Beyond the question of fixed-parameter tractability

Markus Fanebust Dregi

Dissertation for the degree of philosophiae doctor (PhD)

at the University of Bergen

2017

Dissertation date: January 6

Scientific environment

The work of this thesis, both research and writing, was done in the algorithms group at the *Department of Informatics* at the University of Bergen.

Moreover, the project was done entirely under the supervision of Professor Daniel Lokshtanov, and the project received funding from the *Bergen Research Foundation* under the Beating Hardness by Preprocessing grant.

Acknowledgments

First and foremost I would like to thank my supervisor Professor Daniel Lokshtanov. You introduced me to the theory of algorithms almost 10 years ago, and without this introduction I would probably never embarked on my journey in theoretical computer science, let alone written this thesis. I will always be grateful for your enthusiasm, deep insight and elegant overview that sparked my interest and brought my research further.

Second, I would like to thank my wife Anette and my son Mathias for reminding me every day how beautiful life is. For supporting me when I was working late and making the spare time I had during these busy times delightful. I owe you everything! I would also like to thank my parents, siblings, in-laws and friends for your ever lasting support and patience.

The research that has now culminated in this thesis was done in the most friendly research environment I have ever witnessed. I deeply thank the Algorithms group in Bergen for your warm welcome and camaraderie over the years. And for being such an open and fruitful research environment. In particular, I would like to thank Pål, Sigve and Erik for including me in the, back then only, crowd at the masters study hall. And for the friendships that have since evolved during our years as PhD-students. In addition, I sincerely thank you Pål for our countless hours in front of a whiteboard or computer screen, supplemented by an occasional beer or pizza. We have travelled the world together and these years would not have been the same without you! I would also like to thank Pim for our many conversions, and for demonstrating the true beauty of well-formulated sentences, in text as well as in presentations.

I would like thank my fellow students at the University of Oslo during my bachelors, especially Jing and Torgeir, for our joint journey within mathematics, as well our many funny moments together. I would also like to give credit to my teachers, a profession that in general get criticism instead of praise all to often. I truly believe that the scientific skills and the creativity formed by clay and paint, live side by side in thesis as equals.

An big bonus $\star$ to Pål, Pim and Daniel for providing me with invaluable feedback on drafts of this thesis and to Felix for the nice problem environment used throughout this text.

And finally I would like to thank my thesis committee Martin Grohe, Michael Lampis and Fredrik Manne for taking the time to read and evaluate this thesis.

Abstract

Multivariate complexity is a prominent field that over the last decades has developed a rich toolbox, not only to tackle seemingly intractable problems, but also to describe the boundaries of tractability in a richer and more fine-grained way. In this thesis we survey the research directions emerging after the question of fixed-parameter tractability has been settled. That is, we define and exemplify structural parameters, polynomial kernelizations, branching techniques, subexponential time algorithms and parameterized approximation algorithms. In addition, we display techniques for proving lower bounds for all of the above mentioned directions. After this, we give new results within this parameterized framework for several classic graph problems.

The problems studied in this thesis can naturally be divided into two groups; graph modification problems and structural graph problems. With respect to graph modification problems, we study problems where one is to remove a small set of vertices in order to break the graph into small connected components. We also study problems where, instead of deleting vertices, we are to add and/or remove a small number of edges in order to obtain a graph that adheres to a specific set of properties. We resolve several questions in the literature with respect to modifying a graph to a threshold graph or to a chain graph. We prove that editing to such graphs is NP-hard and, under the widely believed Exponential Time Hypothesis (ETH), not solvable in $2^{o(\sqrt{k})} \cdot n^{O(1)}$ time. We also provide polynomial kernels and subexponential time parameterized algorithms running in time $2^{O(\sqrt{k}\log k)} + n^{O(1)}$ for all three edge modification variants into both graph classes.

We also consider edge modifications into $\mathcal{H}$-free graphs, where $\mathcal{H}$ is *any* finite set of forbidden induced subgraphs, on bounded degree input graphs. We prove that for a fixed maximum degree Δ, both edge editing and edge deletion to $\mathcal{H}$-free graphs in at most k operations, admit polynomial kernels with $k^{O(\Delta \log \Delta)}$ vertices. Then, via the framework of cross-compositions we prove that there is a finite set $\mathcal{H}$, such that completion to $\mathcal{H}$-free graphs does not admit a polynomial kernelization algorithm on bounded degree graphs, when parameterized by the bound on the number operations k, unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

With respect to structural graph problems, we first provide several results for bandwidth. We prove that, assuming ETH, there is no significant improvement over the classic dynamic programming algorithm by Saxe [SIAM'80]. In particular, we prove that, assuming ETH, there is no $f(b)n^{o(b)}$ time algorithm for deciding whether the bandwidth of a graph is at most b. This result remains true when restricted to trees of pathwidth at most 2. By the same reduction, we prove that deciding whether the bandwidth of a graph is at most b, when parameterized by b, is W[1]-hard when restricted to the same set of trees. Furthermore, we provide the first approximation algorithm for computing the bandwidth of trees, where the approximation factor depends solely on b. We then extend this result to graphs of bounded treelength, a rich graph class containing among others chordal graphs and graphs of bounded hyperbolicity. We also provide a characterization of graphs of small bandwidth for the same graph classes. In particular, the most general of these results states that a graph of bounded treelength can only have high bandwidth if it has high local density or high pathwidth, or if it contains a slight modification of a bandwidth obstruction introduced by Chung and Seymour [Discrete Mathematics'89].

Finally, we provide a constant factor approximation algorithm for computing the treewidth of a graph that runs in $O(c^k n)$ time. The algorithm either provides a tree decomposition of width $5k + 4$ or concludes that the treewidth of the input graph is larger than k. This algorithm improves several known results, like the one by Robertson and Seymour [JCTB'95] and Reed [STOC'92] and Amir [Algorithmica'10]. We point out that there are many important problems in the literature, for example computing a vertex cover, a dominating set or a steiner tree of a graph, that can be solved in $O(c^k n)$ time if provided a tree decomposition of width at most $O(k)$. The algorithm presented, is the first that can provide such algorithms with a tree decomposition of sufficiently small width, without being the bottleneck of the composed algorithm: That is, an algorithm that first computes a decomposition of width $O(k)$ and then solves the problem in $O(c^k n)$ time.

List of papers

The results of this thesis are based on the following publications:

1. **On the computational complexity of vertex integrity and component order connectivity [DDvH]**.
Pål Grønås Drange, Markus Sortland Dregi and Pim van't Hof. *In* Algorithmica*, pages 1–22. 2016.*

    Part II is based on results from this work.

2. **On the Threshold of Intractability [DDLS15]**.
Pål Grønås Drange, Markus Dregi, Daniel Lokshtanov, and Blair D. Sullivan. *In* ESA 2015*, volume 9294 of* Lecture Notes in Computer Science*, pages 424–436. Springer, 2015.*

    Part III is based on results from this work.

3. **Compressing bounded degree graphs [DDS16]**.
Pål Grønås Drange, Markus Dregi, and R.D. Sandeep. *In* Latin American Symposium on Theoretical Informatics 2015*, pages 362–375. Springer, 2016.*

    Part IV is based on results from this work.

4. **Parameterized complexity of bandwidth on trees [DL14]**.
Markus Dregi and Daniel Lokshtanov. *In* International Colloquium on Automata, Languages, and Programming 2014*, pages 405–416. Springer, 2014.*

    Part V is based on results from this work.

5. **A $c^k n$ 5-Approximation Algorithm for Treewidth [BDD+16]**.
Hans L. Bodlaender, Pål Grønås Drange, Markus Dregi, Fedor V. Fomin, Daniel Lokshtanov and Michał Pilipczuk. *In* SIAM Journal on Computing*, volume 45, number 2, pages 317–378, SIAM, 2016.*

    Part VI is based on results from this work.

All results presented in this thesis have been submitted to journals.

Contents

Part I

Introduction and preliminaries

Chapter 1

Introduction and motivation

Since the formalization of computing, researchers have explored the computability of various tasks, trying to decide which tasks can be solved by a computer and which are resistant to computational efforts. Since the middle of the last century and up until today, an increasing amount of research has been directed towards the question of which problems can be solved efficiently by a computer, and which can not. In the beginning of the 1970's, Cook and Levin independently published their seminal papers [Coo71, Tra84] formally introducing the question of P versus NP. The idea, first stated by Edmonds [Edm65], is that the problems feasible to solve in a reasonable amount of time are the ones that can be solved in polynomial time. As a result P is defined to be exactly these problems. The problems that are hard for the class NP, a class containing P, are believed by most researchers *not* to be solvable in polynomial time and hence to be difficult to solve—if not outright impossible. This classification scheme, where one either manages to solve a problem in polynomial time or prove that a problem is NP-hard, is utilized by researchers in both industry and academia every day.

However, the story often goes as follows; a particular problem is considered important, followed by scientific efforts resulting in the conclusion that the problem is NP-hard. Now what? The problem has been classified as tough to solve, but *we still need to solve it.* In this case, several approaches have been introduced by the computer science community over the years. All of them take the original quest of

"solving the problem optimally in polynomial time"

and relax at least one of the requirements. Either by slightly changing *"the problem"*, while ensuring that the instances we would like to solve are still covered, or by accepting that the instance will not be solved *"optimally"*, but still somewhat satisfactory. The last possibility is to allow spending more than *"polynomial time"* to obtain an optimal solution. In this way we hope to partially circumvent the hardness of the original quest and solve our problem in a satisfying manner.

A guinea pig

Before we continue to illustrate the different relaxations, we are in the need of a guinea pig; i.e., a problem to highlight the various relaxations on. For our purposes the classic graph problem Vertex Cover is perfect. It is easily explained and admits simple results for all the techniques we are going to discuss. In Vertex Cover you are given a graph G and an integer k and the question is whether there exists a subset X of $V(G)$ of size at most k such that all edges of G are incident to at least one vertex in X. Note that Vertex Cover is one the 21 problems Karp proved to be NP-complete as early as in 1972 [Kar72]. Due to this, we do not expect to be able to solve this problem in polynomial time.

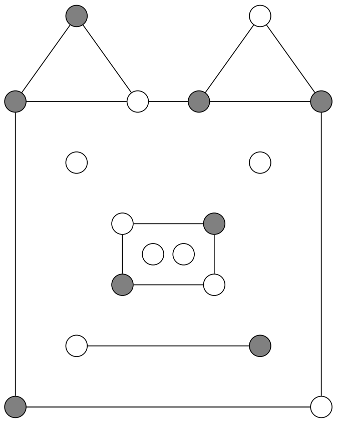


Figure 1.1: A graph together with a gray vertex cover of size 8. Observe how all edges in the graph have at least one gray endpoint.

Approximation algorithms

An extensively studied approach to cope with NP-hardness is to relax the requirement of optimality. Instead of an optimal solution, we are providing a possibly non-optimal solution together with a guarantee for how good the output solution is, either in terms of the optimal solution or the input instance. This is the field of approximation algorithms, a topic on which several books have been written [Hoc96, WS11, Vaz13]. For an optimization problem, OPT denotes the value of the best solutions for an instance and is well-defined as long as there exists a valid solution. For a minimization problem, an algorithm is said to be a c-approximation if the algorithm either gives a solution of value at most $c \cdot \text{OPT}$ or correctly concludes that no solution of value at most OPT exists. Very similarly one can define c-approximation algorithms for maximization problems.

Before we display the folklore 2-approximation algorithm for Vertex Cover we first recall the following: Given an instance (G, k) and a solution X, it holds for every edge uv in G that at least one of u and v is contained in X. The idea of the algorithm is that instead of trying to decide whether u, v or both are to be in the solution, we just add both of them to the solution. Then we remove the two vertices and all incident edges from the graph and repeat the process with another edge until the graph is edgeless. After termination, clearly every edge of G has at least one endpoint in X. The proof of the algorithm being a 2-approximation is based on the observation that whenever the algorithm includes both endpoints of an edge into the solution, any solution, and in particular an optimal one, would have to take at least one of the endpoints. Hence the constructed solution ends up being not more than twice the size of an optimal solution. Interestingly this is the best known approximation algorithm for Vertex Cover and, assuming the unique games conjecture, it is NP-hard to improve upon this [KR08].

Exponential time algorithms

Another option is to relax the requirement of having polynomial running time and move into the field of exponential time algorithms. This is also an extensively studied approach [FK10]. These algorithms do give optimal solutions, but at the expense of running in exponential time. The simplest exponential time algorithm for Vertex Cover goes as follows: For every subset X of $V(G)$ report *yes* if X is a valid solution of size at most k before the algorithm terminates, and *no* if no such X is found. There are 2^n subsets of $V(G)$ and for each of these subsets we spend linear time checking if all edges are covered. Hence, this gives us a $2^n(n+m)$ exact exponential time algorithm. Further improvements can be made by branching techniques evaluating local decisions regarding the solution. The currently best known exact algorithms are by Robson [Rob86] and Bourgeois et al. [BEPvR12], both running in time $O(1.212^n n^{O(1)})$. And while the first has a slightly better base of the exponent, the second one runs in polynomial space.

Restricting input

Yet another approach is to restrict the generality of the problem. It is often the case that the instances of the problem that we are interested in are of a specific structure. Such structures present themselves in problems arising from nature and industry. Suppose that we are only interested in solving Vertex Cover on forests, that is, on graphs without cycles. The following observation is sufficient to devise an optimal algorithm: If u is a leaf in the graph, in other words a vertex with a single neighbor v, there is an optimal solution containing v. This is due to the fact that u can only cover the edge uv, while v can potentially also cover other edges. Observe that a forest always has at least two leaves and remains a forest when deleting vertices, hence being a *hereditary property*. It follows that we can find a leaf u, add its neighbor v to the solution and remove v and all incident edges from the forest. Then, as long as there are edges in the forest, repeat the procedure with another leaf. This yields a linear time algorithm if we are careful enough to keep a list of the leaves in the forest, instead of searching for them.

But what if your graph is not a forest, but still quite close? In such cases you might still be able to devise a polynomial time algorithm. And what if you have several such algorithms, which one should you choose? This brings us to the world of multivariate complexity.

1.1 Multivariate complexity

Occurrences of using multiple variables in complexity can be found as early as 1972 in the seminal paper by Edmonds and Karp [EK72] on maximum flow. In particular, it has been used to describe the complexity of graph algorithms in terms of both the number of vertices (n) and the number of edges (m). However, we all know that the number of edges in a graph is bounded by the square of the number of vertices. And hence, the complexity of various graph algorithms, including breadth-first search, depth-first search, Dijkstra's algorithm and so forth, all could be described only in terms of the number of vertices. But this would imply that a simple graph traversal no longer would be of complexity $O(n+m)$, but $O(n^2)$. The $O(n+m)$ bound indicates that the algorithm would terminate within a second on an instance with a million vertices and edges. However, the promise of the $O(n^2)$ bound is more along the lines of termination within hours. Thus, even though both upper bounds are formally correct, the classic $O(n+m)$ yields a far more descriptive picture of the complexity of the algorithm at hand. And, not only does this finer analysis by introducing more variables give a better description of the complexity of existing algorithms, it also provides a tool for guiding further algorithmic development. This is the very idea that parameterized complexity captures and further develops upon. A field that there has been written numerous, now classic, books on: Downey and Fellows [DF99], Flum and Grohe [FG06], Niedermeier [Nie06], Downey and Fellows [DF13] and Cygan et al. [CFK+15].

1.1.1 A gentle introduction to parameterized complexity

In this thesis we will consider a *parameterized problem* as a problem where each input instance comes with one integer that is predefined as a *parameter*. Based on this we can define the notion of a problem being *fixed-parameter tractable*, or equivalently, belonging to the class FPT. An example of such a problem would be VERTEX COVER parameterized by the requested solution size k. We say that such a problem is *fixed-parameter tractable* if there exists an algorithm solving problem instances of size n with parameter value k in time $O(f(k) \cdot n^{O(1)})$ for some computable function f.

As a first example, we will give such an algorithm for VERTEX COVER parameterized by k, the requested solution size. Recall the observation we made when developing an approximation algorithm for the problem, namely that for any edge uv in the input graph G either u or v is to be in the solution. In the approximation algorithm we exploited this by picking both of the vertices in the solution and then proving that this strategy would yield a solution that is at most twice the size of an optimal solution. Now, however, we have more time at hand, namely a factor $f(k)$, and this we should utilize. What if we tried adding the two vertices, one at the time, to the solution before we recursively cover the remaining edges? First, we could try adding u to the solution and recurse on the instance $(G - u, k - 1)$: Here $G - u$ since all edges incident to u gets covered and $k - 1$ since we are only allowed to add $k - 1$ additional vertices to the solution. If the recursive call decides that $(G - u, k - 1)$ is a yes-instance, we return yes. Otherwise, we recurse on the instance $(G - v, k - 1)$. If both recursive calls fail, we know that (G, k) is a no-instance and hence we return this.

If at any time in the recursion we get an edgeless graph, we are trivially faced with a yes-instance and hence we conclude so. On the other hand, if our graph contains edges and the parameter k is zero we are presented with a trivial no-instance and we return this. This simple, recursive algorithm will always correctly conclude whether the given instance is a yes- or no-instance. It remains to analyze the running time of the algorithm. Observe that any recursive call makes at most two new recursive calls and that the depth of the recursion tree is bounded by $k + 1$. This immediately yields that we make at most $O(2^k)$ recursive calls during the execution of our algorithm. In each call we identify an edge for which we remove each of its endpoints, one by one, from the graph. This takes linear time, and hence the complexity of the branching algorithm is $O(2^k(n + m))$. We can conclude that VERTEX COVER parameterized by k is indeed fixed-parameter tractable.

1.1.2 Harder than FPT

A natural question to ask is whether all parameterized problems are fixed-parameter tractable. So far we have seen that VERTEX COVER is in FPT when parameterized by the solution size. As a first example of a problem that seems to not be in FPT we introduce GRAPH COLORING. Here, you are given a graph G and a number of available colors k as input and the question is whether you can color each vertex in G with one of the k colors such that no two neighbors get the same color. It was proven by Garey, Johnson and Stockmeyer in 1974 that this problem is NP-hard even if you fix k to be 3 [GJS74]. And hence, assuming $\mathsf{P} \neq \mathsf{NP}$, there is no algorithm solving the problem in time $O(f(k) \cdot n^{O(1)})$, since replacing k by 3 would yield a polynomial time algorithm for an NP-hard problem.

This property of being NP-hard already for fixed values of the parameter is often referred to as being para-NP-hard. Not only can we rule out fixed-parameter tractable algorithms for these problems, but we can also rule out parameterized algorithms with running times on the form $f(k)n^{g(k)}$. The parameterized problems that do admit such algorithms belong to the class XP. Despite being polynomial time solvable for every fixed value of the parameter, the polynomial gets worse as the parameter grows. This is in contrast to a fixed-parameter tractable algorithm, where the degree of the polynomial remains the same as the parameter grows.

One can easily observe from the definitions that $\mathsf{FPT} \subseteq \mathsf{XP}$. But it does not stop here, in fact there is an entire hierarchy of classes, called the W-hierarchy, that we do believe to encapsulate problems that are not in FPT, but in XP. Without going into the details of these classes, its structure is as follows:

$$\mathsf{FPT} = \mathsf{W[0]} \subseteq \mathsf{W[1]} \subseteq \mathsf{W[2]} \subseteq \cdots \subseteq \mathsf{W[t]} \subseteq \mathsf{XP},$$

where we believe all containments to be strict. For the readers familiar with nondeterministic Turing machines, we note that the problem of deciding whether a nondeterministic Turing machine will accept a string in at most k nondeterministic steps is W[1]-complete when parameterized by k [DF99]. Due to this, many of the reasons for believing that $\mathsf{P} \neq \mathsf{NP}$ carry over to $\mathsf{FPT} \neq \mathsf{W[1]}$. Instead of going into the hierarchy we will present a classic graph problem that is complete for W[1]. After this we will illustrate how one can use reductions to prove that other problems also are hard for the same class.

Our default starting point for parameterized hardness will be Clique. Here you are given a graph G and an integer k and the question is whether there is a subset of vertices C in G of size k, such that every pair of vertices in C are neighbors. This problem was proven to be NP-complete by Karp [Kar72]. Furthermore, one can observe that the problem is in XP since for any fixed k one can iterate over all subsets of vertices of size k and check whether it is a clique in polynomial time, yielding an $O(n^{k+2})$ time algorithm. It was proven by Downey and Fellows [DF95] that Clique parameterized by the solution size is indeed complete for W[1], and therefore unlikely to be in FPT.

Parameterized reductions

We have established that Clique is W[1]-hard and hence is not believed to admit a fixed-parameter tractable algorithm. What we need is a tool that allows us to extend this and show that other problems are not (or unlikely to be) in FPT. We do this by proving that these problems are at least as hard as Clique and hence that they are at least as hard as all problems in W[1]. For this purpose parameter preserving reductions fits our need. A *parameter preserving reduction* is an algorithm that takes as input an instance (x, k) of a parameterized problem Π and outputs an instance (x', k') of Π' such that the following three conditions are meet:

- the algorithm runs in time $f(k)|x|^{O(1)}$ for a computable function f,
- (x, k) is a yes-instance if and only if (x', k') is a yes-instance and
- $k' \leq g(k)$ for some computable function g.

It is well-known that if a problem A is W[1]-hard and there is a parameter preserving reduction from A to another problem B, then B is also W[1]-hard. To motivate this, consider the case that there is a parameter preserving reduction from A to B and that B is contained in FPT. We claim that then A is also in FPT. In particular, we devise the following algorithm solving A in fixed-parameter tractable time: Given an instance (x, k) of A, we first apply the parameter preserving reduction to obtain an instance (x', k') of B. Then, we apply an algorithm solving (x', k')

in $h(k')|x'|^{O(1)}$ time, whose existence is guaranteed by B $\in$ FPT. Since (x,k) is a yes-instance if and only if (x',k') is a yes-instance, we have now solved the instance (x,k). It remains to analyse the running time of this composite algorithm. First, we spent $f(k)|x|^{O(1)}$ time to obtain (x',k'). Then, we spent $h(k')|x'|^{O(1)}$ time to solve (x',k'). We observe that since the reduction algorithm terminates in $f(k)|x|^{O(1)}$ time, and this includes the time it takes to output the instance (x',k'), it follows that $|x'| \leq f(k)|x|^{O(1)}$. Hence, we get the following total running time:

$$\begin{aligned}
f(k)|x|^{O(1)} + h(k')|x'|^{O(1)} &\leq f(k)|x|^{O(1)} + h(g(k))|x'|^{O(1)} \\
&\leq f(k)|x|^{O(1)} + h(g(k))[f(k)|x|^{O(1)}]^{O(1)} \\
&= [f(k) + h(g(k))f(k)^{O(1)}]|x|^{O(1)} \\
&= f'(k)|x|^{O(1)},
\end{aligned}$$

for a computable function f'. Due to this we conclude that A $\in$ FPT. Observe that if A was a W[1]-hard problem to start with, this proves FPT $=$ W[1], something we consider to be unlikely.

To finish of our discussion about hardness, we give an easy reduction needed later in the thesis, namely when proving lower bounds for Bandwidth in Chapter 22. We now prove that Even Clique, an instance of Clique where k is promised to be an even number, is W[1]-hard.

Theorem 1. Even Clique *is* W[1]*-hard.*

Proof. We give a simple reduction from Clique, which is known to be W[1]-hard [DF95]. Given an instance (G,k) of Clique, if k is even the instance is already a valid instance of Even Clique and the correctness is trivial. Otherwise, let G' be G with a universal vertex added (i.e. a vertex adjacent to every vertex of G) and $k' = k+1$. Clearly, k' is even, so this is a valid instance. If there is a clique of size k in G, then the same clique together with the universal vertex forms a clique of size k' in G'. And the other way around, if there is a clique of size k' in G'. Then there is a subset of this clique of size k not containing the added universal vertex. This is a clique in G of size k and hence our reduction is sound. Since the reduction is parameter preserving it follows immediately that Even Clique is W[1]-hard.

□

1.1.3 Beyond fixed-parameter (in)tractability

A natural question to ask after deciding whether a problem belongs to FPT or not is: What is next? In the same manner as after deciding whether a problem belongs to P or not, the story is far from over after establishing the question of containment in FPT. In this thesis we will explore the different directions this story can take. We will highlight and exemplify the various methods that have developed within computational theory as a result of multivariate complexity.

This journey will be via established and classic results in this first part, as well as trough original research later in the thesis.

The motivation of multivariate complexity is to better capture the complexity of computational tasks. But to do this, it is crucial to chose the right variables. In Section 1.2 we will explore different parameterizations of problems. In particular, we will see how structure of the input can be measured and utilized in the development of algorithms. In Section 1.3 we will see how preprocessing data has evolved within the realm of multivariate complexity. This is formalized as kernels and polynomial kernels. Then, in Section 1.4, we will study how one can obtain faster algorithms. Among others, we will see how preprocessing can significantly improve the running time of algorithms. Last, in Section 1.6 we discuss the role of multivariate complexity within approximation algorithms. This is an application that is used both for obtaining faster algorithms even though the parameterized problem is fixed-parameter tractable and to obtain tractable algorithms when the problem is believed to not be in FPT.

Every method above is complemented with techniques for proving that this method is not applicable for certain problems, under various complexity theoretic assumptions. One should note that this is necessary, as some of these statements, like proving that a problem does not admit a fast fixed-parameter tractable algorithm is a stronger statement than that the problem does not admit a polynomial time algorithm.

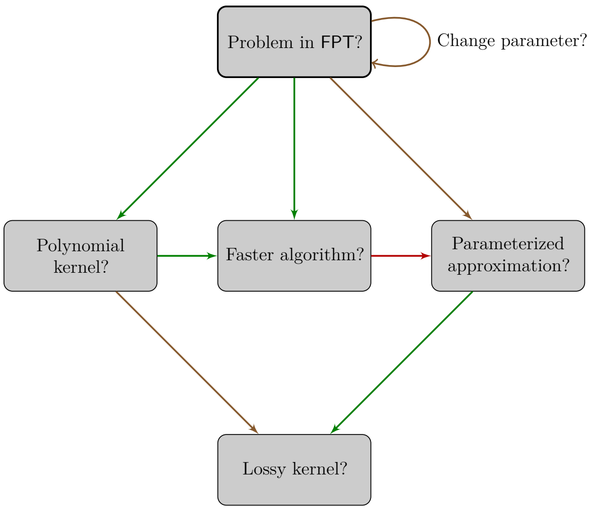


Figure 1.2: Flow chart of relevant research questions in parameterized complexity. Follow green arrows if the question is answered positively, red if negatively and brown in both cases.

In Figure 1.2 we display a natural ordering of research questions asked for a parameterized problem. In particular, we start with deciding whether the parameterized problem is contained in FPT. If this is answered positively, we aim for faster algorithms and efficient preprocessing schemes. One can also hope for faster algorithms by allowing the algorithm to provide approximate answers. Last, one can consider to change the parameter to obtain better algorithms. If there is strong evidence of the problem not being in FPT, we are left with two options. Either we can allow approximate answers or we can change the parameter, hoping that the action yields tractability.

1.1.4 A formal introduction to parameterized complexity

Before we continue we give the formal definitions of the concepts in parameterized complexity discussed so far, as presented in the book by Cygan et al. [CFK+15].

Definition 1.1 ([CFK+15])**.** A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the *parameter*.

Definition 1.2 ([CFK+15])**.** A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, such that given $(x, k) \in \Sigma^s \times \mathbb{N}$, the algorithm $\mathbb{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{O(1)}$. The complexity class containing all fixed-parameter tractable problems is called FPT.

Definition 1.3 ([CFK+15])**.** A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *slice-wise polynomial* if there exists an algorithm $\mathcal{A}$ and two computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathbb{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.

1.2 Various parameterizations

So far we have considered the problems Vertex Cover and Clique parameterized by their solution sizes. And although being a natural way to go about parameterized complexity, it might not be the one that suits your needs. Parameterized complexity is about identifying properties of problem instances that make them computationally tractable or intractable, and then utilize these properties when solving the instances. And with this in mind, it might be that the current parameterization does not capture the tractability of the problem in a good way.

Before continuing we briefly address how we will compare parameters in the context of graph measures. We will consider a graph measure x to be a smaller parameter than another graph measure y, if there exists a computable function f such that for every graph G it holds that $x(G) \leq f(y(G))$. Similarly, we will say that x is larger than y if $x(G) \geq f(y(G))$ for every graph G. If neither of these scenarios applies, we say that the two measures at incomparable parameters.

As a first example, we consider Clique parameterized by the vertex cover number of the input graph. Observe that in a clique all but one of the vertices are to be in a vertex cover. Because of this, we can conclude that the vertex cover number of the graph is lower bounded by the clique number minus one. Hence, we are now considering Clique by a larger parameter than solution size and there might be hope for the problem with the new parameterization to be in FPT. And this does indeed turn out to be the case. We are given a graph G and integers k and ℓ, such that ℓ is an upper bound on the vertex cover number of G. And the question is if there is a clique of size k in G. First, we spend $O(2^\ell(n+m))$ time to obtain a minimal vertex cover X of G. Then, we recall that all but one vertex of a clique in G must be inside X, since $G - X$ is an edgeless graph. Hence, we can brute force the intersection between a maximum clique and the vertex cover Y in time 2^ℓ. And verify in ℓ^2 time that the selected set indeed forms a clique. To finish the computation we need to check if there is a vertex in $V(G) \setminus X$ for which its neighborhood contains Y. The straight forward check of this can be executed in $O(n^2)$ time. In total this yields a running time of $O(2^\ell(n+m)+2^\ell\ell^2n^2) = O(2^\ell\ell^2n^2)$ and hence Clique parameterized by the vertex cover number is in FPT. One can observe that, the opposite scenario, parameterizing Vertex Cover by the clique number of the input graph will yield a hardness result. It was proven by Garey and Johnson [GJ79b] that Vertex Cover is NP-hard on graphs of maximum degree 3, and hence also on graphs of maximum clique size 4. It follows immediately that the aforementioned parameterization of Vertex Cover is para-NP-hard.

Numerous results have been given for problems parameterized by something else than solution size. Jansen and Bodlaender [JB11] proved that Vertex Cover admits a polynomial kernel when parameterized by the feedback vertex set number of the input graph. Similarly, Fellows et al. [FLM+08] studied several NP-hard graph layout problems parameterized by the vertex cover number of the input graph.

Many combinatorial problems can be formulated naturally as an *integer linear program*. And then, one can construct the *linear program relaxation* of this ILP, by not requiring the variables to be integral. A recent, interesting line of research is to parameterize a problem by the difference between the target solution quality and the optimum of the linear program relaxation. Lokshtanov et al. [LNR+14] provided an $O(2.3146^r n^{O(1)})$ time algorithm for Vertex Cover, where r is the difference between the requested solution value k and the optimum of the linear program relaxation. However, the most studied parameterizations, besides solution size itself, are still width parameters, which we will now discuss in more detail.

1.2.1 Width parameters

Width parameters exists in various forms and the goal is to capture structure of the input that can aid significantly in computations. We have previously seen that if we restrict the input graph to be a forest, then Vertex Cover becomes solvable in polynomial time. And maybe the insight gained for forests could carry over to graphs that are almost forests. This quest of quantifying how close you are to a certain computationally simple structure, has inspired numerous width parameters. And among the measures based on forests, and arguably among all width parameters, the one with the most impact is *treewidth*. Its goal is to measure how closely a graph resembles a forest in such a way that it highlights structure that is beneficial for computations.

And which properties is it that makes forests much more computationally tractable than graphs in general? It is the fact that they are cycle-free. When making a local decision, the impact propagates through the instance. But, in the case of a forest, these propagations never meet again, as there are no cycles. Hence, after making a decision for a vertex v one can often reduce the original problem to independent sub-problems, one for each connected component of $G - v$. And this is the feature that treewidth highlights. It gives you access to multiple small sets of vertices in a tree structure, where each set breaks the graph into connected components. The hope is that after making decisions for a provided set X, one can solve each of the components of $G - X$ independently.

Definition 1.4 (Tree decomposition)**.** Given a graph $G = (V, E)$ we say that $\mathcal{T} = (T, \mathcal{X})$, were T is a tree and $\mathcal{X}$ is a collection of subsets of V indexed by $V(T)$, is a *tree decomposition* of G if the following conditions holds:

(i) $\bigcup_{X \in \mathcal{X}} X = V$,

(ii) for every edge $uv \in E$ there exists an $X \in \mathcal{X}$ such that $\{u, v\} \subseteq X$ and

(iii) for every $v \in V$ it holds that $T[\{i \mid v \in X_i\}]$ is connected.

The elements of $\mathcal{X}$ is often referred to as the *bags* of the tree decomposition. And in other words a tree decomposition $\mathcal{T}$ is to satisfy that the union of the bags equals the vertex set of G, that for every edge in G there is a bag containing both of its endpoints and that for every vertex $v \in V$ the vertices in T corresponding to bags containing v form a connected subtree of T. Based on this we can define treewidth as follows.

Definition 1.5 (Treewidth)**.** Given a graph G and a corresponding tree decomposition $\mathcal{T} = (T, \mathcal{X})$ we define the treewidth of $\mathcal{T}$, denoted $\text{tw}(\mathcal{T}, G)$, as $\max_{X \in \mathcal{X}} |X| - 1$. Based on this we define the *treewidth of* G as

$$\text{tw}(G) = \min \text{tw}(\mathcal{T}, G) \text{ where } \mathcal{T} \text{ is a tree decomposition of } G.$$

We say that a tree decomposition $\mathcal{T}$ of G is *optimal* if $\text{tw}(\mathcal{T}, G) = \text{tw}(G)$.

Now, instead of parameterizing a problem by the solution size, we have the option of parameterizing by the structure of the input, namely the treewidth of the input graph. An argument for doing this is that we can, for instance, identify optimal vertex covers in graphs with large solutions if the graph itself has a treelike structure. There are various options for obtaining a tree decomposition, one of them being presented in Section 1.6 and several others in Part VI. But we will leave that part for now and assume that we are given a tree decomposition together with our instance.

The most crucial observation when designing algorithms for tree decompositions goes as follows: For any bag $X_s \in \mathcal{X}$ and pair of vertices $u, v \in V$ it holds that if s separates the bags containing u from the bags containing v in the decomposition tree T, then X_s separates u and v in G. This implies that all communication between u and v in G must be through X_s. In the case of Vertex Cover this can be used as follows: When extending a solution A for a connected component of $G - X_s$ together with X_s, to a solution of the entire graph, the only information we need is which vertices in X_s are contained in A and the cardinality of A. Everything else about A, are "internal workings" and non-critical for the extension.

Based on this we can design a dynamic programming procedure for Vertex Cover on tree decompositions of width t that runs in $2^t t^{O(1)} n$ time. The approach is to root your decomposition by an arbitrary bag and do dynamic programming from the leaves up. The dynamic programming table contains an integer entry for every vertex s of the decomposition tree and every subset A_s of X_s. This integer value is precisely the size of the smallest vertex cover of the graph induced by the union of the bags in the subtree rooted at X_s, such that the vertex cover contains A_s.

Leaf bags: Here one simply map every vertex cover of the graph induced by the bag to their respective sizes and every other subset to some marker of impossibility, like infinity.

After this, we process the bags in a bottom-up fashion, such that when computing the tables for a specific bag, each of its children have already had their tables computed.

Non-leaf bags: We investigate each of the 2^{t+1} different subsets A_s of the bag individually. Since the only interaction between the various connected components in the graph induced by the bags below are via the vertices of the current bag, where you have fixed an interaction with the solution, you can select the best among the appropriate solutions for each of the children and combine them without conflicts.

We can assume the decomposition to have at most $O(n)$ bags (see Section 2). For each bag we inspect all the 2^{t+1} different interactions with an imagined solution and do look-ups in the tables of all of its children. In total, this yields a $2^t t^{O(1)} n$ time algorithm. For a formal description of similar algorithms as the one we just described, as well as correctness proofs, we refer the reader to Cygan et al. [CFK+15].

1.3 Polynomial kernels

Preprocessing is frequently exploited in computationally heavy tasks. Often certain parts of the input instance are easily solvable and can be trimmed away fast before one tackles the core of the instance. One such example would be the preprocessing rule we applied for Vertex Cover on forests. We observed that it would always be at least as good to put the neighbour of a leaf into the solution as the leaf itself. The same preprocessing rule is applicable to any instance, the difference being that in a forest one can always identify a leaf and hence the preprocessing turns into an algorithm. In a graph this is not always the case. Moreover, the existence of a polynomial time preprocessing algorithm for Vertex Cover that guarantees to shrink an instance with n vertices to an instance with $n-1$ vertices, would imply $\mathsf{P} = \mathsf{NP}$. The reason being that if we apply this algorithm repeatedly, we solve the problem in polynomial time. Due to this assumed contradiction, it seems that the classic complexity framework is unfit for studying the theory of preprocessing. A major success however, has been to study preprocessing in the context of parameterized problems. Characterizing the success of a preprocessing procedure in terms of a parameter, yields no contradictions such as above.

Definition 1.6 (Kernel)**.** A parameterized problem Π admits a *kernelization algorithm*, or simply a *kernel*, if there exists computable functions f and g together with a polynomial time algorithm $\mathcal{A}$ that given an instance (x, k) of Π outputs an equivalent instance (x', k') of Π such that the following holds:

(i) $|x'| \leq f(k)$ and

(ii) $k' \leq g(k)$.

If both f and g are polynomial functions we say that Π admits a *polynomial kernelization algorithm*, or *polynomial kernel*.

Before we continue, we would like to highlight a folklore relation between a problem being fixed-parameter tractable and admitting a kernel. We remind the reader that a problem is decidable if there exists an algorithm that solves the problem.

Theorem 2. *A decidable problem Π is in FPT if and only if it admits a kernel.*

Proof. ($\Rightarrow$) Assume that Π belongs to FPT and hence admits an algorithm that given an instance (x, k) solves it in $f(k) \cdot |x|^{O(1)}$ time. We will examine two cases. First, if $|x| \geq f(k)$ it holds that $f(k) \cdot |x|^{O(1)} \leq |x|^{O(1)+1}$. In this case we can run the existing algorithm in polynomial time and based on the answer either output a yes- or no-instance of constant size. It remains to consider the case when $|x| < f(k)$. In this case, the size of the instance is already bounded by a function f depending solely on the parameter, and hence we can output the original instance. This concludes the forward direction of the proof.

($\Leftarrow$) Assume that Π admits a kernel. Since the problem is decidable there exists an algorithm solving the problem. Given input (x, k) we first run the kernelization algorithm and obtain a new instance (x', k'). We then run the algorithm on (x', k'). Observe that the size of the compressed instance depends solely on the parameter and so does the running time. Hence we end up with an algorithm solving the problem in time $O(|x|^{O(1)} + f(k))$ for some function f.

□

When a problem is proven to be fixed-parameter tractable we know by Theorem 2 that the problem also admits a kernel. A natural next question is hence whether or not the problem admits a polynomial kernel. And if so, how small of a polynomial kernel that is possible to obtain. The field of kernelization has grown tremendously the last years. The field has grown from consisting of self-contained, combinatorial arguments to also heavily utilize and further develop results from all over mathematics. One example is the application of the Sunflower Lemma [ER60] by Erdös and Rado to prove that d-Hitting Set admits a polynomial kernel for every fixed d. Another example is the usage of linear programming and the Nemhauser-Trotter theorem to prove that Vertex Cover admits a $2k$ vertex kernel [CKJ01]. However, up until 2008 there was no method for disproving the existence of polynomial kernels. This prevented scientists from being able to prove that it was not their efforts or insight that was lacking, but the polynomial kernel itself. However, the work of Fortnow and Santhanam [FS08] and Bodlaender et al. [BDFH09] resulted in a robust framework for proving that, under reasonable complexity assumptions, polynomial kernels does not exist for certain problems. With this at hand kernelization is today a rigorous field with the tools necessary for exploring the preprocessing tractability of computational problems. We first give an example of a polynomial kernel, before we introduce the above mentioned framework for proving the non-existence of polynomial kernels.

1.3.1 A first polynomial kernel

As an example we will prove that Vertex Cover admits a polynomial kernel with $O(k^2)$ vertices and edges. The result is by Buss and Goldsmith [BG93] and variation of the techniques used are now so common that results obtained in a similar manner are often referred to as a Buss-kernel.

Rule 1.1. *Let (G, k) be an instance of* Vertex Cover *and v an isolated vertex in G. We then reduce the instance to $(G - v, k)$.*

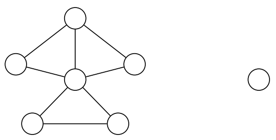


Figure 1.3: Rule 1.1 says that the isolated vertex has no impact on optimal vertex covers for the graph.

Lemma 1.7. *Let (G, k) be an instance of* Vertex Cover *and v an isolated vertex in G. It then holds that (G, k) and $(G - v, k)$ are equivalent.*

Proof. Since v is an isolated vertex it follows that $E(G) = E(G - v)$. Furthermore, observe that no minimal solution would contain v. It follows that every minimal solution of (G, k) is a minimal solution of $(G - v, k)$ and vice versa. □

Rule 1.2. *Let (G, k) be an instance of* Vertex Cover *and v a vertex in G of degree at least $k + 1$. We then reduce the instance to $(G - v, k - 1)$.*

Lemma 1.8. *Let (G,k) be an instance of* Vertex Cover *and v a vertex in G of degree at least $k+1$. It then holds that (G,k) and $(G-v,k-1)$ are equivalent.*

Proof. Assume that (G,k) is a yes-instance and let X be a solution. Furthermore, assume for a contradiction that $v \notin X$. It follows that $N_G(v) \subseteq X$ and since $\deg(v) \geq k+1$ this contradicts X being a solution. Since $v \in X$ it follows that all edges in $G-v$ have an endpoint in $X-v$ and hence $(G-v,k-1)$ is also a yes-instance.

For the other direction, assume that $(G-v,k-1)$ is a yes-instance and that X is a solution. Observe that all edges of G are either present in $G-v$ or incident to v by definition. It follows immediately that $X \cup \{v\}$ is a solution of the instance (G,k) and hence that (G,k) is a yes-instance. □

Theorem 3. Vertex Cover *admits a polynomial kernel with $O(k^2)$ vertices and edges.*

Proof. Observe that Rule 1.1 and 1.2 can be applied exhaustively in polynomial time. Furthermore, by Lemmata 1.7 and 1.8 it follows that the reduced instance is equivalent to the original one.

Let (G,k) be a reduced instance and let X be an optimal solution. By definition every edge in G has an endpoint in X. And since no vertex can be incident to more than k edges due to Rule 1.2, it follows that the number of edges in G are bounded by k^2. Furthermore, from Rule 1.1 no longer being applicable it follows that every vertex is incident to an edge in G and hence $V(G) = X \cup N_G(X)$. Since every vertex in G is of degree at most k, it follows that $|N_G(X)| \leq k \cdot |X|$ and hence that

$$|V(G)| = |X \cup N_G(X)| \leq |X| + |N_G(X)| \leq (k+1)|X| \leq (k+1)k.$$

Hence, it holds that if (G,k) is a yes-instance, then $|V(G)| \leq k(k+1)$. It follows from the contrapositive statements that if $|V(G)| > k(k+1)$ or $|E(G)| > k^2$ we can immediately conclude that (G,k) is a no-instance.

This gives the following kernelization algorithm. Apply Rule 1.1 and 1.2 exhaustively. If the number of vertices in the reduced instance is more than $k(k+1)$ or the number of edges is more than k^2, output a constant size no-instance. Otherwise, output the reduced instance.

□

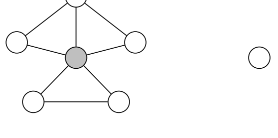


Figure 1.4: Rule 1.2 says that if $k \leq 4$ we have to include the gray vertex in any solution.

1.3.2 Cross Compositions

We will now introduce cross-compositions as presented in the book by Cygan et al. [CFK+15]. For a detailed discussion of the topic, proofs, or multiple examples, we refer the reader to this text. For the original research article we refer the reader to the paper by Bodlaender et al. [BJK14]. After displaying the theory we will give a simple application. First, we will introduce the reduction necessary to prove that a problem does not admit a polynomial kernel.

Definition 1.9 (As presented in [CFK+15])**.** An equivalence relation $\mathcal{R}$ on the set Σ^* is called a *polynomial equivalence relation* if the following conditions are satisfied:

(i) There exists an algorithm, that given strings $x, y \in \Sigma^*$, resolves whether $x \equiv_{\mathcal{R}} y$ in time polynomial in $|x| + |y|$.

(ii) The relation $\mathcal{R}$ restricted to the set $\Sigma^{\leq n}$ has at most $n^{O(1)}$ equivalence classes.

Definition 1.10 (Cross-compositions, as presented in [CFK+15])**.** Let $L \subseteq \Sigma^*$ be a language and $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language. We say that L *cross-composes* into Q if there exists a polynomial equivalence relation $\mathcal{R}$ and an algorithm $\mathcal{A}$, called the *cross-composition*, satisfying the following conditions. The algorithm $\mathcal{A}$ takes as input a sequence of strings $x_1, x_2, \dots x_t \in \Sigma^*$ that are equivalent with respect to $\mathcal{R}$, runs in polynomial time in $\Sigma_{i=1}^{t}|x_i|$, and outputs one instance $(y, k) \in \Sigma^* \times \mathbb{N}$ such that:

(i) $k \leq (\max_{i=1}^{t} |x_i| + \log t)^{O(1)}$ and

(ii) $(y, k) \in Q$ if and only if there exists at least one index i such that $x_i \in L$.

Basically, you want a reduction that can be computed in polynomial time that takes multiple instances of a problem L into a single instance of the parameterized problem Q such that the instance of Q is a yes-instance if and only if at least one of the instances of L is a yes-instance. Furthermore, we want the parameter of the instance of Q to be bounded by a polynomial of the maximum size of the instances of L and the logarithm of the number of instances. Before we give the main result we need to introduce polynomial compressions, a generalization of kernelization algorithms that are allowed to output an instance of a different problem.

Definition 1.11 (Polynomial compression, as presented in [CFK+15])**.** A *polynomial compression* of a parameterized language $Q \subseteq \Sigma^* \times \mathbb{N}$ into a language $R \subseteq \Sigma^*$ is an algorithm that takes as input an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, works in time polynomial in $|x| + k$, and returns a string y such that:

(i) $|y| \leq p(k)$ for some polynomial p and

(ii) $y \in R$ if and only if $(x, k) \in Q$.

We are now ready to state our main theorem in the quest for hardness results within kernelization.

Theorem 4 ([BJK14], as presented in [CFK+15])**.** *Assume that an* NP*-hard language* L *cross-composes into a parameterized language* Q*. Then* Q *does not admit a polynomial compression, unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$*.*

In the problem k-Path one is given a graph G and an integer k and the question is whether there exists a simple path of length k. The problem is well-known to be NP-complete. It was proven to be in FPT by Monien [Mon85] and the fastest known deterministic algorithm runs in time $O(2.6181^k n^{O(1)})$ and is by Zehavi [Zeh15]. We will now argue that although being in FPT, the problem does not admit a polynomial kernel. The argument will be by displaying that the problem cross-composes into itself. And hence, since the problem is NP-hard, it does not admit a polynomial kernel. As input our reduction take multiple non-parameterized instances $(G_1, k), (G_2, k), \ldots, (G_t, k)$ of k-Path. It then outputs the disjoint union of $G_1, G_2, \ldots, G_t$ (i.e. the graph with the disjoint union of the vertex sets as its vertices and the disjoint union of the edge sets as its edges) together with the now parameter k. That is, unless $|V(G_i)| < k$ for an instance, in which case we output a trivial no-instance. First, note that the input instances are equivalent under the relation $\mathcal{R}$ classifying all instances asking for the same path length k as equivalent. Second, we observe that the reduction runs in polynomial time and that the parameter is bounded as required. The last requirement follows directly from the observation that a k-path cannot span multiple connected components and hence must be contained strictly within one of the original instances. It now follows immediately from Theorem 4 that k-Path parameterized by k does not admit a polynomial compression, and hence no polynomial kernel, unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

In our example we did not use the fact that the parameter of the instance output by the cross composition can be dependent on $\log t$. The reason that we did not need this is that we were lucky enough that when simply throwing the instances together with no additional structure, the answer of the new instance turned out to be an OR over the original instances. We are seldom this lucky and often one needs to carefully place gadgets on top of the instances in a manner that after spending some $\log t$ of the budget, exactly one of the instances are activated, and hence the new instances is equivalent to an OR over the original instances. These selection techniques were first utilized by Dom et al. [DLS09]. Applications of this technique will be displayed in Chapter 14 and 15. It should also be noted that it has been proven by Drucker [Dru15] that if a NP-hard problem cross-composes into a problem such that the new instance is a yes-instance if and only if *all* the original instances are yes-instances, yields the same consequences. Namely that the problem admits no polynomial compression unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$. From this it follows by the same type of reduction as above that Treewidth does not admit a polynomial kernel when parameterized by the width at question.

1.3.3 Generalizations and relaxations

One might ask the question of why it is necessary to kernelize into a single instance, or why one should spend strictly polynomial time, as one possibly is to spend exponential time after the preprocessing anyway. Solving multiple small instances is often vastly preferable over a single large one, since this implies a smaller exponent in the running time at the cost of a multiplicative factor in front. Of course, a single small instance would be even better. But, as we have just seen, this is not always possible. Jansen [Jan14] proved that among others k-Path does admit such a kernelization scheme, referred to in the literature as a Turing kernel, on various graph classes including planar graphs, where the problem does not admit a traditional kernel. Multiple other results has appeared giving Turing kernels for various problems that do not admit a traditional kernel.

Lossy kernels

The motivation behind kernelization and preprocessing in general is that it does not hurt to preprocess the instance before doing the final attack on the instance, hoping to shrink the instance to a more manageable size. And although this motivation is true for exact algorithms and in general for many kernelization results, the definition does not ensure this. To be more precise, the traditional definition of kernelization does not mix well with approximation algorithms. There is no mechanism that ensures that if you obtained a c-approximate solution in the preprocessed instance, that this carries over to an approximate solution of the original instance. And this is especially frustrating if you have spent large computational efforts into obtaining such a solution.

This motivations is captured within the framework of lossy kernels introduced by Lokshtanov et al. [LPRS16]. More specifically, a problem admits a polynomial sized α-approximate kernel if there exists a polynomial time algorithm that outputs another instance of the same problem whose size is bounded by a polynomial of the original parameter. And in addition, a polynomial time algorithm exists that given a c-approximate solution of the kernelized instance outputs a $(\alpha \cdot c)$-approximate solution of the original instance. One should note that holding $c = \alpha = 1$ gives more or less the behaviour of a traditional kernel. While, fixing $\alpha = 1$ and letting c vary gives a stronger result as approximations in the processed instance can be lifted to approximations in the original instance of the same approximation ratio. By allowing $\alpha > 1$ we possibly bypass the traditional hardness framework for kernels, making it possible to obtain sensible preprocessing schemes for problems that do not admit polynomial kernels. This makes the framework fitting for further studies of problems that both do and do not admit traditional polynomial kernels. Both positive and negative results exists within this framework, for more information we refer the reader to Lokshtanov et al. [LPRS16].

1.4 Faster algorithms

One can argue that the most natural question to ask after discovering an algorithm is: Can the same task be solved even faster? Enormous efforts are put into this line of research within classic as well as multivariate complexity. There is however a major distinction between the two. In classic complexity it is easy to conclude that one algorithm is faster than the other within the realm of O-notation. This is not always the case when several variables are introduced. Two algorithms with complexity $O(2^k n^2)$ and $O(k^k n)$ are interchangeably better, depending on the relation between the parameter and the instance size. But still, it always makes sense to try to improve upon the existing algorithms. In some cases one makes sacrifices on the way, as in the example above, and in other cases one can devise an algorithm that has a better dependency on both the parameter and the input size. For instance, an $O(2^k n)$ algorithm would be preferable to both of the algorithms above. In this section we will highlight several basic techniques for improving the running time of multivariate algorithms.

1.4.1 Improving by polynomial time reductions

One technique that is frequently utilized both in practice and theory is the one of preprocessing. By applying reduction rules one solves the instance partly before engaging more costly computational efforts. In Section 1.1 we devised an algorithm for Vertex Cover running in $O(2^k(n+m))$ time. Furthermore, by Theorem 3 we know that that Vertex Cover admits a polynomial kernel with $O(k^2)$ edges and vertices.

We will now argue that the two rules of this kernel can be applied exhaustively (i.e. until neither of the rules are applicable anymore) in $O(k(n+m))$ time. This is obtained by marking all vertices of the graph that is to be removed, before we remove them all in one go in the end.

First, we mark all vertices that are to be removed by Rule 1.1 in $O(n+m)$ time. This is done by iterating over all vertices of the graph and whenever a vertex has no neighbors, we mark it for removal. Next, we iterate over the vertices of the graph and if we find a vertex v with at least $k+1$ neighbors, we remove all the edges of the graph incident to v in $O(n+m)$ time, before we mark v for removal. We then restart the iteration, looking for another vertex of high degree. Observe that applying Rule 1.2 more than k times always result in trivial no-instance, as k would be negative. And hence, applying Rule 1.2 as above can be done in $O(k(n+m))$ time in total. It remains to remove all vertices of the graph that is marked. Note that all of these are isolated vertices (i.e. of degree 0). We first make a table that for each vertex, answers how many vertices is of lower index than this vertex and is to be removed. We then copy the entire graph, reducing the value of endpoints of the edges according to the table we just computed. This copying procedure is done in $O(n+m)$ time.

Observe that none of the reduction rules increase the value of the parameter. Hence, given an instance (G,k) we apply the kernelization algorithm to obtain a graph (G', k') in time $O(k(n+m))$. We then apply the branching algorithm from Section 1.1 on the new instance (G', k'). In total this yields a running time of $O(k(n+m) + 2^{k'}(|V(G')| + |E(G')|)) = O(k(n+m) + 2^k k^2)$. We have now successfully separated the exponential dependence on the parameter and the linear dependency on the size of the input graph, which can have a huge impact on the running time.

Another way to view our branching algorithm for Vertex Cover is that for any vertex v in the graph, we either put it into the solution or leave it out. And if we leave it out, we always have to include the entire neighborhood of v in the solution. By consistently branching only on vertices of degree at least 1, this yields the same $O(2^k(n+m))$ complexity as before. However, observe that in the recursive call where v is excluded from the solution, and its neighborhood is included, the higher degree v has, the more vertices we get to put into the solution. Which again will make our algorithm run faster. Specifically, if we could always find a vertex of degree at least two we would obtain a faster algorithm. And now, we can bring back the preprocessing rule we applied for forests in the beginning of this chapter; namely that for a leaf we are always safe to add its neighbor to the solution. By always applying this rule when applicable in the recursion, we can guarantee that when branching on a vertex v, v will have degree at least two. This implies that in one branch our parameter still drops by one, but in the other it drops by at least two.

To analyze the running time we will first bound the number of leaves in a recursion tree of depth k of the algorithm, denoted $L(k)$. In particular, we prove that $L(k) \leq 1.619^k$ by induction. For the base case we observe that $L(0) = 1$. For the induction step we observe that $L(k) \leq L(k-1) + L(k-2)$ and hence the proof follows by the following computation:

$$L(k) \leq L(k-1) + L(k-2) \leq 1.619^{k-1} + 1.619^{k-2} \leq 1.619^k.$$

Since there are no vertices of degree one in the recursion tree, it follows that there are $O(1.619^k)$ vertices in the tree in total. When removing vertices of degree 1, we first compute a list of all the leaves of the graph. And by keeping this list updated while removing vertices and using the same trick as above, where we first mark all vertices to be removed, we can get rid of all degree 1 vertices in $O(n+m)$ time. Hence, we spend $O(n+m)$ time inside each recursive call and get a total running time of $O(1.619^k(n+m))$. When combining this new algorithm with the above implementation of our kernelization algorithm we obtain the running time $O(1.619^k k^2 + k(n+m))$.

1.4.2 Subexponential time algorithms

A natural question would be if we could significantly improve upon our new algorithm for Vertex Cover. The best known algorithm is by Chen, Kanj and Xia [CKX10] and runs in time $O(1.2738^k + kn)$. And although we can continue to bring down the exponent of the algorithm, it is believed that we can not improve the algorithm to a $2^{o(k)}n^{O(1)}$ running time. But this is a discussion for later. There are other problems for which there are algorithms solving them in $2^{o(k)}n^{O(1)}$ time. Such algorithms are referred to as *subexponential time algorithms.*

The problems proven to admit subexponential time algorithms mainly fall into two categories, problems restricted to graph classes and graph modification problems, i.e. problems where one is to modify the graph to adhere to a set of problem specific properties. The most studied modifications are removing vertices and removing, adding or both removing and adding edges. Demaine et. al [DFHT05] proved that classes of problems admit subexponential time algorithms on planar graphs, as well as some larger graph classes. Later, Alon et al. [ALS09] proved that Feedback Arc Set on tournament graphs admits a subexponential time algorithm. Both of these fall into the category of problems on restricted graph classes. After this, numerous results have appeared proving that various graph modification problems also admit subexponential time algorithms [FV13, GKK+15, DFPV15, BFPP14, BFPP16]. All of these results, except the one showing Interval Completion to admit a subexponential time algorithm [BFPP16], rely heavily on the problem also admitting a polynomial kernel. The reason for this is highlighted by the following observation.

Observation 1.12. *Let X be a set of size at most k^c for some constant c, it then holds that*

$$\binom{|X|}{d\sqrt{k}} \leq 2^{cd\sqrt{k}\log k} = 2^{o(k)}.$$

Proof. $\binom{|X|}{d\sqrt{k}} \leq |X|^{d\sqrt{k}} = 2^{d\sqrt{k}\log|X|} = 2^{cd\sqrt{k}\log k}$. □

Consider that you are solving an instance (G, k) of some parameterized graph problem that admits a polynomial kernel. You can then assume that $|V(G)| \leq k^c$ and hence you can enumerate all subsets of size at most $d\sqrt{k}$ in subexponential time. Now, if you can characterize some interesting set of this size, you can try all possibilities for this set and utilize this for solving the remainder of the problem.

In the context of edge modification graph problem one can partition the vertices into two sets, the *cheap* and *expensive* vertices. The cheap vertices are incident to few edges of a fixed optimal solution, say at most $2\sqrt{k}$. And the expensive vertices are incident to more than $2\sqrt{k}$ such edges. Based on this, one can easily prove that there are at most $\sqrt{k}$ expensive vertices in a graph. And hence, by Observation 1.12 one can in subexponential time enumerate all possible sets of expensive vertices. When this is done all vertices, besides this small set of expensive ones, are cheap. And the nice thing about cheap vertices are that the solutions interaction with them is at most $2\sqrt{k}$. This once again lets us utilize Observation 1.12 for a cheap vertex v, because we can enumerate the possible subsets of vertices for which the incidence between v and this vertex is altered by the solution. And hence, we can enumerate the possible neighborhoods of v in the resulting graph. Notice that we cannot do this for all cheap vertices at the same time, as this would be too time consuming. But by carefully selecting cheap vertices to analyze we hope to gain further insight regarding our instance and obtain useful structure that in the end leads to an efficient algorithm.

Edge Modification into split graphs

As a case study we will show how the observations above can be used to obtain $2^{O(\sqrt{k}\log k)}n^{O(1)}$ time algorithms for both edge deletion and completion into split graphs. This is a result that was originally proven, by different methods, by Ghosh et al. [GKK+15]. A graph G is a *split graph* if there exists a partition of the vertex set into two sets C and I such that C is a clique and I is an independent set. Such a partition is called a *split partition*. We denote by $\oplus$ the *symmetric difference* of two sets, specifically we let $X \oplus Y = (X \setminus Y) \cup (Y \setminus X)$. The following result will be crucial, both for this problem and for the subexponential time algorithm given in Chapter 10.

Lemma 1.13. *Given a graph $G = (V, E)$ and an integer k with $|V| = k^{O(1)}$ one can in $2^{O(\sqrt{k}\log k)}$ time generate a collection $\mathcal{P}$ of partitions of V such that for every graph H such that*

- *H is a split graph and*
- *$|E(H) \oplus E(G)| \leq k$*

it holds that a split partition (C, I) of H is an element of $\mathcal{P}$.

Proof. We first prove that the lemma holds if G is a split graph. First, we fix a split partition (C, I) of G. Consider a split graph H on the same vertex set with split partition (C_H, I_H). Observe that

$$|E(H) \oplus E(G)| \geq \binom{|C \setminus C_H|}{2} + \binom{|I \setminus I_H|}{2},$$

since the graphs the sets induces in G and H are complementary. Hence, $|E(H) \oplus E(G)| \leq k$ implies that both $|C \setminus C_H|$ and $|I \setminus I_H|$ are bounded by $2\sqrt{k}$. By applying Lemma 1.12, we can enumerate every possible $C \setminus C_H$ and $I \setminus I_H$ in time $2^{O(\sqrt{k}\log k)}$. Since, the two sets together with (C, I) are sufficient to retrieve (C_H, I_H), the lemma holds assuming that G is a split graph.

We will now prove the full lemma by using the fact that split editing is indeed polynomial time solvable [HS81]. Given G we obtain a split graph G' with the smallest editing distance to G. We apply the lemma on G' with symmetric difference at most $2k$ to obtain a collection $\mathcal{P}$. The correctness of $\mathcal{P}$ with respect to G' follows from G' being a split graph. By the triangle inequality it holds that if the symmetric difference between G and a split graph H is at most k, then the symmetric difference between G' and H is at most $2k$. And hence, $\mathcal{P}$ proves the correctness of the lemma when applied to G with symmetric difference at most k. $\square$

Theorem 5 ([GKK+15])**.** *Both* Split Completion *and* Split Deletion *can be solved in time* $O(2^{O(\sqrt{k}\log k)} + n^{O(1)}$.

Proof. First, we apply the polynomial kernel by Guo [Guo07] to obtain a reduced instance (G, k). We then apply Lemma 1.13 to obtain the candidate split partitions. Last, for every such partition (C', I') we need to add and remove edges so that C' forms a clique and I' an independent set. If an operation that is not available (either removing or adding an edge) is necessary, we immediately continue to the next partition. Otherwise, we count how many legal operations we need to turn G into a split graph with split partition (C', I') by counting either the number of missing edges in $G[C']$ or the number of edges in $G[I']$. If this is at most k we conclude that it is a yes-instance. It no such (C', I') with at most k legal operations needed is found, we conclude that (G, k) is a no-instance. □

1.5 Optimality

It is natural to wonder whether a certain algorithm is optimal, meaning that no faster, or significantly faster algorithm exists. We have discussed algorithms of single exponential, as well as subexponential running time. There are also problems for which the fastest algorithms we know of are on the form $O(k^k n^{O(1)})$ or $O(2^{2^k} n^{O(1)})$. And many problems are resistant to our efforts of trying to improve the exponential dependency of the running times. It appears that maybe there is a finer structure within FPT, partitioning the problems based on how fast fixed-parameter tractable algorithms we can devise for these problems. Our current understanding of complexity seems however to not be sufficient to grasp this finer structure under the assumption $\mathsf{FPT} \neq \mathsf{W[1]}$. Instead, we will utilize the Exponential Time Hypothesis, a complexity theoretic assumption that turns out to be at least as strong as $\mathsf{FPT} \neq \mathsf{W[1]}$.

1.5.1 The Exponential Time Hypothesis

There has been put tremendous efforts, within theory as well as practice, into studying the computational tractability of satisfiability. In this problem, which we will refer to as SAT, one is provided a boolean formula and the question is whether there exists an assignment to the n variables such that the formula evaluates to true. The problem was among the very first problems to be proven to be NP-complete. And although large instances are solved in practice by heuristics every day there is yet to be a significant guaranteed improvement over the algorithm that tries every possible assignment for each variable, running in $O(2^n n^{O(1)})$ time. One can however get below this running time by requiring the formula to be in conjunctive normal form with at most q variables in each clause, also referred to as q-SAT. In particular, for $q = 3$, namely 3-SAT, there is an algorithm by Hertli [Her14] running in time $O(1.308^n n^{O(1)})$. It should be noted that even though 3-SAT is a restriction of SAT, it was among the 21 problems studied by Karp [Kar72] and hence well-known to be NP-hard.

The Exponential Time Hypothesis (ETH) was introduced by Impagliazzo and Paturi [IP01], building upon earlier work by Impagliazzo Paturi and Zane [IPZ01]. ETH has since found its applications in classic complexity, as well as multivariate complexity. We will not give the formal definition of the hypothesis in its original form. And, if interested, we encourage the reader to explore the book on parameterized complexity by Cygan et al. [CFK+15]. We will however say that an immediate consequence of this hypothesis is that 3-SAT cannot be solved in subexponential time in the number of variables, meaning that no $O(2^{o(n)}n^{O(1)})$ time algorithm exists. And hence, 3-SAT can also not be solved in polynomial time, meaning that ETH implies $\mathsf{P} \neq \mathsf{NP}$. However, applying this directly can be rather tedious or even useless for tight bounds as one needs to be careful about how one handles the number of clauses m in the reduction. Applying the Sparsification lemma [IPZ01] resolves this issue and results in the following theorem.

Theorem 6. *Unless ETH fails, there exists a constant $c > 0$ such that no algorithm for* 3-SAT *can achieve running time $O(2^{c(n+m)}(n+m)^d)$ for any constant d. In particular,* 3-SAT *cannot be solved in time $2^{o(n+m)}(n+m)^d$.*

This result allows for a more fine grained classification of problems based on their complexity and rule out certain running times. Among others it has been proved that ETH implies that not only 3-SAT, but also Vertex Cover and many other classic problems, do not admit running times on the form $2^{o(n+m)}n^{O(1)}$. Another hypothesis introduced by Impagliazzo and Paturi [IP01] is the Strong Exponential Time Hypothesis (SETH). It builds upon the observation that no algorithm faster than $O(2^n n^{O(1)})$ has been discovered for SAT and SETH rules out any algorithm on the form $O((2-\epsilon)^n n^{O(1)})$. It can be proven that SETH is at least as strong an assumption as ETH. Furthermore, it should be noted that while ETH is believed to be true by a considerable amount of researchers, SETH is by many considered to be more of a guiding tool saying that improving upon this would imply a major breakthrough within the complexity of SAT. The added power allows researchers to prove that the constants in the exponent of the running times are optimal and for many problems we have tight upper and lower bounds under SETH.

1.5.2 ETH and parameterized problems

ETH has yielded numerous results within parameterized complexity. Among others, it has been proven that assuming ETH, our companion through this text, namely Vertex Cover, does not admit a $2^{o(k)}n^{O(1)}$ time algorithm. The key to further extend the set of such problems is to give a reduction from a problem, for which we already have a running time lower bound based on ETH, to a new problem while keeping the parameter as small as possible. For instance, if a polynomial time reduction[1] from Vertex Cover to some other problem Π exists, taking an instance (G, k) to an instance (X, p) such that $p \leq dk$ for some constant d. Then

[1]Actually one can spend even more time depending on the bound one is trying to prove.

Π does not admit a subexponential time algorithm, assuming ETH. The reason for this is that otherwise, given an instance (G, k) of Vertex Cover one can in polynomial time obtain an equivalent instance (X, p) of Π. This instance is then solvable in $2^{o(p)}|X|^{O(1)}$ time by assumption. And, since $p \le dk$ and $|X| \le |G|^{O(1)}$ due to the running time bound, we obtain a $2^{o(k)} \cdot |G|^{O(1)}$ time algorithm for Vertex Cover, contradicting ETH.

Similarly, if $p \le dk^2$ instead, this would rule out any algorithm with running time $2^{o(\sqrt{p})}|X|^{O(1)}$ under ETH. The reason being that if such an algorithm would exist, one can do the same procedure as previously, and since $2^{o(\sqrt{p})}|X|^{O(1)} = 2^{o(\sqrt{(k^2)})}|G|^{O(1)} = 2^{o(k)}|G|^{O(1)}$ we would once again contradict ETH. Observe that the smaller increase in the parameter is, the better lower bounds we can give.

We will later give such a reduction from the problem $k \times k$-Clique. Here, you are given an instance $(G, k, \mathcal{X})$ where $\mathcal{X} = X_1, \dots, X_k$ is a partition of $V(G)$ into k sets, each of size k. And the question is whether there exists a clique C in G such that $|X_i \cap C| = 1$ for all i. In essence, the proof of the following lower bound is by the same technique as above. However, the reduction goes from SAT and ensures that k is of size $n/\log n$. In this manner, one is able to transfer the implications of ETH from classic to parameterized complexity.

Theorem 7 ([LMS11])**.** *Unless ETH fails, $k \times k$-*Clique *cannot be solved in time $O(2^{o(k \log k)}k^{O(1)})$.*

Similarly, ETH has also been used to prove lower bounds for problems that are W[1]-hard. Because even though a problem is W[1]-hard, there might very well exist a $f(k)n^{g(k)}$ algorithm solving it, were g is such a slow growing function that for any reasonable value of k, it would behave like a fixed-parameter tractable algorithm. There has indeed been developed such algorithms where $g(k) = \log \log k$, which under ETH has been proven to be optimal [Mar08a]. We will base our results on the work by Chen et al. [CHKX06] proving the following results.

Theorem 8. *Assuming ETH, there is no $f(k)n^{o(k)}$ time algorithm for* Clique *for any computable function f.*

By reusing the reduction from the proof of Theorem 1 we can immediately give the following result.

Theorem 9. *Assuming ETH, there is no $f(k)n^{o(k)}$ time algorithm for* Even Clique *for any computable function f.*

Proof. This follows from the fact that the reduction of Theorem 1 increases the parameter by at most an additive factor of one. □

1.6 Parameterized approximation algorithms

Many classic problems appears to not be in FPT. And if you are actually interested in solving instances of the problem, then a negative answer is a setback. However, sometimes a positive answer is as well. It all depends on how it is answered. If you devise an algorithm with running time $2^{2^k}n$ and you want to solve instances were $k = 8$, the algorithm you have will not terminate in a reasonable amount of time. In fact, it might not terminate within the lifetime of the universe on a super computer. It can also be that there exists a $k^k n$ algorithm as well as a $2^k n^2$ algorithm, but your instance is so that you really need a $2^k n$ time algorithm or similar. And even worse, if you proved that the problem is W[1]-hard, you do not even have access to such algorithms.

And just as we did for NP-hard problems, one can try to relax the requirements on the algorithm. We have already relaxed the running time constraints, allowing our algorithm to spend exponential time in the parameter. Depending on the parameter, we have also potentially restricted the structure of the input graph. With this in mind, it seems natural that the next step is to allow the solutions to be non-optimal. And by doing so, we hope that we can find an algorithm satisfying our needs. This allows us to not only apply the tools from parameterized complexity, but also the ones from approximation algorithms.

Since the celebrated graph minors project [RS95] parameterized approximation algorithms have been crucial within the context of computing structural decompositions. Following the approximation algorithm for treewidth by Roberston and Seymour [RS95], for which a variant will be presented in Section 1.6.1, there have been several algorithms developed for treewidth [Ree92, Lag96, Ami10, BDD$^+$16]. Parameterized approximation algorithms was also utilized by Oum and Seymour [OS06] to approximate clique-width and related parameters, which was later improved by Oum [Oum08]. It was also used by Demaine et al. [DHK05] to give a parameterized 2-approximation for Graph Coloring on H-minor free graphs and by Bockenhauer et al. [BHKK07] to give a 2.5-approximation for Traveling Sales Person with Deadlines.

Definition 1.14. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is said to admit a *fixed-parameter tractable approximation algorithm*, or simply a *parameterized approximation algorithm*, if there exists an algorithm $\mathcal{A}$, computable functions $f, g : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^s \times \mathbb{N}$, the algorithm $\mathbb{A}$ either

- outputs a polynomial time verifiable witness of $(x, g(k) \cdot k) \in L$ or
- correctly concludes that $(x, k) \notin \Sigma^s \times \mathbb{N}$

in time bounded by $f(k) \cdot |(x, k)|^{O(1)}$.

In addition to the positive results, there has been provided several hardness results. It was proven by Downey et al. [DFMR08] that approximating Dominating Set within any additive constant factor is W[2]-hard and that there is no fixed-parameter tractable approximation algorithm for Independent Dominating Set unless FPT = W[1]. It was recently proven by Chen and Lin [CL] that Dominating Set has no constant factor parameterized approximation algorithm unless FPT = W[1]. This was build upon the work of Lin [Lin15] proving that Biclique is W[1]-hard.

1.6.1 Approximating treewidth

Treewidth is a structural graph measure that has received tremendous attention. There have been given numerous algorithms to compute tree decompositions and even more algorithms for solving various problems assuming that one is provided with such a decomposition. Of course, the later results all depends on a tree decomposition actually being computed. Despite being extremely well-studied and proven to be in FPT when parameterized by the treewidth, there are no algorithms to compute an optimal width decomposition that are even close to the fast running times one typically get for the algorithms that solves problems given the tree decomposition. And even though at times it can make sense to spend more time preprocessing a graph then actually solving problems on it, since the structure can be reused for other computations, there are limits! And due to this unsatisfying situation, one has turned to fixed-parameter tractable approximation algorithms for help. The idea is to obtain a structure of approximate width, and base the remaining computations on this structure. And even though the approximate structure yields a slightly worse running time for the computations afterwards, all in all this is still highly rewarding. One should note that the typical dynamic programming algorithms over tree decompositions is of the type displayed in Section 1.2, in the sense that given an instance (G, k) of VERTEX COVER together with a tree decompositions of G of width t, we conclude either that G has a vertex cover of size k or that any vertex cover of G is of size larger than k. The algorithm only depends on the width t in the running time and hence the algorithm will, even when provided with a decomposition of non-optimal width, conclude correctly regarding the size of vertex covers in G. Specifically, one should note that an algorithm running in time $O(2^t t^{O(1)} n)$ time, will run in time $O(2^{ct}(ct)^{O(1)} n)$ time when provided with a tree decomposition of width ct.

We will now give a 4-approximation algorithm for treewidth that runs in time $O(27^k k^{O(1)} n^2)$, a result originally given by Robertson and Seymour [RS95]. That is, the algorithm will on input (G, k) either output a tree decomposition of width at most $4k$ or correctly conclude that $\mathrm{tw}(G) \geq k$ in $O(27^k k^{O(1)} n^2)$ time. The algorithm relies heavily on the following result, which is a slightly weaker version of a theorem from Graph Minors II [RS86].

Theorem 10 (Graph Minors II [RS86]). *If $\mathrm{tw}(G) \leq k$ and $S \subseteq V(G)$, then there exists $X \subseteq V(G)$ with $|X| \leq k + 1$ such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S.*

Proof sketch. Let $\mathcal{T}$ be a tree decomposition of G of width at most k. First, for every edge of the decomposition we direct it towards the part of the decomposition for which its union has the largest intersection with S. If there is a tied edge e, then let X be the intersection of the two bags e connects. Let S_r be the intersection of one side of e with S and S_l be the intersection of the other side of e with S. That is, e breaks the decomposition into two connected components L and R. And S_l is the intersection of S and the union of the bags in L and S_r is the intersection of S and the union of the bags in R. Since $|S_l| = |S_r|$ and $X \subseteq S_l \cap S_r$ it follows that $|S_l \setminus X| = |S_r \setminus X|$. Hence, from the observation that S_l, S_r and $X \cap S$ forms a partition of S we conclude that both S_l and S_r have cardinality at most $\frac{1}{2}|S|$. Last, we observe that X is the intersection of two bags and each bag is of size at most $k+1$, hence it follows immediately that $|X| \leq k+1$.

For the rest of the proof we assume that there was no tied edge. Furthermore, for an edge e we let L denote the component of $T - e$ that the e points away from. Hence, by the same argument as above, but considering $|S_l| \leq |S_r|$, we can conclude that for any edge $|S_l| \leq \frac{1}{2}$. Observe that our directed tree is a directed acyclic graph and hence has at least one sink. Assume for a contradiction that there is more than one sink and let x and y be two such sinks. Consider the underlying path in the tree from x to y. Observe that the first edge e_x on the path is pointing at x and that the last edge e_y is pointing at y. Let S_R^x and S_L^x denote S_R and S_L with respect to e_x, and similarly S_R^y and S_L^y denote S_R and S_L with respect to e_y.

Observe that $S_R^y \subseteq S_L^x$ and $S_R^x \subseteq S_L^y$ by definition. And hence, by the direction of the edges we get $|S_L^y| \geq |S_R^x| > |S_L^x| \geq |S_R^y|$, contradicting the direction of e_y.

Now, consider our unique sink x and let X be the vertices in this bag. As the width of the decomposition is bounded by k it follows immediately that $|X| \leq k+1$. Consider any component C of $G - X$ and look at the component of $\mathcal{T} - x$ that contains C. Let e_C be the edge leaving this component pointing at x. By the argument earlier it holds that S_l, the components intersection with S it at most $\frac{1}{2}|S|$ and hence our proof is complete. $\square$

Theorem 11 ([RS95])**.** *There is an algorithm that given a graph $G = (V, E)$, a set $S \subseteq V$ and a $k \in \mathcal{N}$ that either correctly concludes that* $\mathrm{tw}(G) > k$ *or outputs a set $X \subseteq V$ of size at most $k+1$ such that every component of $G - X$ has at most $\frac{2}{3}|S|$ vertices which are in S and runs in time $O(3^{|S|}k^{O(1)}(n+m))$.*

Proof sketch. By Theorem 10 it holds that if $\mathrm{tw}(G) \leq k$ there is a set X such each component's intersection with S is at most $\frac{1}{2}|S|$. First, we will prove that the components of $G - X$ can be assigned left or right in such a way that at most $\frac{2}{3}|S|$ vertices of S are assigned to the left and at most $\frac{2}{3}|S|$ vertices are assigned to the right. Sort the components in decreasing order by intersection with S and assign them, one by one, to the side that contains the fewest vertices of S so far. Let L and R be the number of vertices of S assigned left and right respectively and assume without loss of generality that $L \geq R$. Let A be the number of vertices of S that are contained in the next component to be assigned a direction. We will now prove by induction that $A + R \leq \frac{2}{3}|S|$. If $R = 0$, then since $A \leq \frac{1}{2}|S|$ by Theorem 10 it holds that $A + R \leq \frac{2}{3}|S|$. Otherwise, we observe that due to the ordering of the assignments it holds that $L \geq R \geq A$ and hence the two inequalities $2L \geq R + A$ and $L + R + A \leq |S|$ holds. By a simple substitute we can conclude that $A + R \leq \frac{2}{3}|S|$.

Now, we can enumerate all possibilities of the vertices of S being left, right or in X in time $O(3^{|S|})$. What remains is to identify the vertices of X outside of S or vertices that do the same task. And this task is specifically to separate the vertices of S that are to the left from the ones that are to the right. Due to the existence of X, if the correct partition of S was obtained above, there should exist at most $k + 1 - |S \cap X|$ vertices in $G \setminus S$ such that these vertices together with $S \cap X$ forms a separation of the right and left vertices. These vertices outside S can be identified by a max-flow algorithm. And since we are interested in flows of value at most $k + 1$ this can be computed in $O(k(n + m))$. Hence, if the partition of S sends at most $\frac{2}{3}$ of its vertices in one direction and one can find vertices that in total forms X, we output this set. Otherwise, we conclude that $\text{tw}(G) > k$. $\square$

We are now ready to devise an algorithm that given a graph G and an integer k either outputs a tree decomposition of width $4k + 3$ or correctly concludes that $tw(G) > k$ in time $O(27^k k^{O(1)} n^2)$. The internal workings of the algorithm will be a recursive procedure that is provided with G, k and a set of vertices S containing at most $3k + 3$ vertices. In return, this procedure will give a tree decomposition of G of width at most $4k + 3$ with S contained in its root bag or correctly conclude that $\text{tw}(G) > k$. Our algorithm calls this procedure on G, k, $S = \emptyset$ and outputs the result.

The recursive procedure works as follows. First, until $|S| = 3k + 3$ we add arbitrary vertices of G to S. Then we apply the algorithm of Theorem 11 to G, k and S to obtain an X separating S in a balanced way. If no such X can be found, we conclude that $\text{tw}(G) > k$. Let $C_1, \ldots, C_\ell$ be the connected components of $G - X$. We recursively apply our procedure on $G[C_i \cup X], k, S_i = (C_i \cap S) \cup X$ for each i. Since $|C_i \cap S|$ is bounded by $\frac{2}{3}(3k + 3) = 2k + 2$ and $|X| \leq k + 1$ it follows immediately that $S_i \leq 3k + 3$. If any of the recursive calls conclude that $\text{tw}(G) > k$, we immediately propagate this conclusion up in the recursion tree. Otherwise, we let $S \cup X$ be the root bag of our decomposition and make the roots of the recursively built decompositions for each $C_i \cup X$ be the children of our new root. It can easily be observed that every vertex of the graph appears in a bag and that for every edge its two endpoints appear in a bag together. Forcing S to be in the root bag guarantees that the bags containing a specific vertex are connected in the decomposition and hence we do indeed construct valid tree decompositions. Since $|X| + |S| \leq k + 1 + 3k + 3 = 4k + 4$, the width of the build decomposition is at most $4k + 3$. By standard recursive analysis of the running time combined with the observation that if $\text{tw}(G) \leq k$ then $m \leq kn$ [BF05] we can conclude that the running time of the algorithm is indeed $O(27^k k^{O(1)} n^2)$ (see Chapter 25 for details regarding the analysis).

Theorem 12 ([RS95]). *There is an algorithm that given input (G, k) either*

- *outputs a tree decomposition of G of width at most $4k$ or*
- *correctly concludes that $\text{tw}(G) > k$*

in $O(27^k k^{O(1)} n^2)$ time.

1.7 Concluding remarks

We have now explored the various directions research beyond fixed-parameter (in)tractability can take you. In particular we have considered structural parameterizations and their applications. We have looked at polynomial kernels and displayed the framework of cross-compositions, allowing scientists to prove the non-existence of polynomial kernels for certain problems assuming $\mathsf{NP} \not\subseteq \mathsf{coNP/poly}$. Furthermore, we studied how preprocessing, as well as clever branching techniques, can yield faster and even subexponential time algorithms. We then considered the Exponential Time Hypothesis and its implications, allowing us to give running time lower bounds for various problems, assuming ETH. Last, we looked at how merging the fields of approximation algorithms and parameterized complexity can aid in obtaining tractable algorithms, both for problems in FPT and problems that are believed to not be in FPT. We also listed the sparse amount of results on the non-existence of fixed-parameter tractable approximation algorithms. We end this survey redisplaying the flow-chart from the beginning of the chapter.

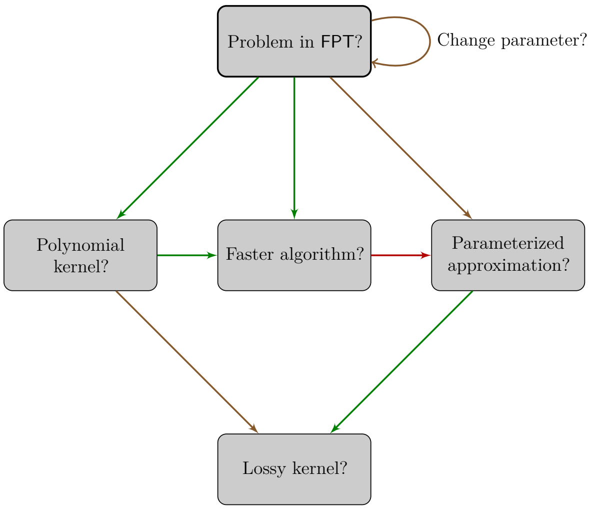


Figure 1.5: Flow chart of relevant research questions in parameterized complexity. Follow green arrows if the question is answered positively, red if negatively and brown in both cases.

1.8 Organization and overview

This thesis is divided into seven parts. In this first part, we have explored the field of parameterized complexity. And we will end it with setting up the necessary definitions and notation used throughout this text. After this follows five sections, where each section evaluates a specific problem with respect to the parameterized methods displayed in this part. In particular, after establishing whether the problem is in FPT or not, we apply parts of the research strategy we have described in this chapter, and visualized in the flow-chart of Figure 1.5, on each and every one of the problems.

For formal definitions of the problems mentioned below, we refer the reader to Chapter 2 or the discussed part. In the last part, Part VII, we conclude with some open problems.

Part II: Here we display some simple results for two graph vulnerability measures. In particular, the problems ask for small vertex sets for which removal breaks the graph into connected components of small size. First, we establish that both of the problems are in FPT parameterized by the sum of the cardinality of the small set disconnecting the graph and the bound on the component size. With this in mind, we first prove that both problems admit polynomial kernels by the same parameterization. Second, we prove that, assuming ETH, one of the algorithms cannot be significantly improved upon.

Part III: Here we study edge modifications into threshold graphs. It follows immediately by the results of Cai [Cai96] that these problem are in FPT, when parameterized by k, the input bound on the number of operations. However, the classic complexity of the editing variant has been stated as an open problem repeatedly. We first establish that THRESHOLD EDITING is NP-hard and that, under ETH, this problem does not admit a $2^{o(\sqrt{k})} \cdot n^{O(1)}$ time algorithm. We then prove that THRESHOLD DELETION, THRESHOLD COMPLETION and THRESHOLD EDITING admit polynomial kernels with $O(k^2)$ vertices. Finally, utilizing the kernel and complementing the lower bound, we give a subexponential time algorithm for THRESHOLD EDITING running in time $2^{\sqrt{k}\log k} + n^{O(1)}$. All the above mentioned results are also given when the target graph class is the closely related class of chain graphs.

Part IV: In this part we study edge modification problems for a more general setup, namely into $\mathcal{H}$-free graphs, where $\mathcal{H}$ is a finite collection of obstructions. Again, it follows by the work of Cai [Cai96] that the problems are in FPT when parameterized by k, the input bound on the number of operations. We prove that both $\mathcal{H}$-FREE EDGE DELETION and $\mathcal{H}$-FREE EDGE EDITING admit polynomial kernels of size $k^{O(\Delta \log \Delta)}$ on graphs of a fixed maximum degree Δ. We complement these results by utilizing the framework of cross-compositions to prove that $\mathcal{H}$-FREE EDGE COMPLETION does not admit a polynomial kernel on graphs of bounded degree. In addition, we prove that neither of three problems above admit polynomial kernels on graphs of bounded degeneracy. All of the lower bounds are under the assumption that $\mathsf{NP} \not\subseteq \mathsf{coNP/poly}$.

Part V: The results of this part is on BANDWIDTH parameterized by b, the solution quality. First, we prove that BANDWIDTH, when parameterized by b, is W[1]-hard. Furthermore, we observe that by the same reduction, the problem admits no $f(b)n^{o(b)}$ time algorithm for any computable function f, assuming ETH. This implies that there is no significant improvement over the classic dynamic programming algorithm by Saxe [Sax80]. Both of these hardness results hold even when restricted to trees of pathwidth at most 2. On the positive side, we give polynomial time approximation algorithms for bandwidth on caterpillars, trees and graphs of bounded treelength. These are the first approximation algorithms for bandwidth on these classes where the approximation factor depends solely on b. In addition to the algorithms, we give for all three of the above graph classes a characterization of graphs of low bandwidth. In particular, we prove that the only way a graph in one of these classes can have high bandwidth is by either having high local density, high pathwidth or containing a less rigid version of an obstruction introduced by Chung and Seymour [CS89].

Part VI: Finally, we study TREEWIDTH parameterized by the requested width k. This extensively studied problem is well-known to be in FPT [Bod89] under this parameterization. However, none of the existing algorithms can provide an exact, nor satisfying approximate, tree decomposition in a time that matches the running time of the typical algorithms exploiting this decomposition. In this part we provide several approximation algorithms for the treewidth of a graph, with the goal of resolving this bottleneck. The main result is a 5-approximation algorithm for treewidth that runs in $O(c^k \cdot n)$ time. Specifically, the algorithm either outputs a tree decomposition of width $5k + 4$ or correctly conclude that the treewidth of the graph is larger than k in time $O(c^k \cdot n)$, for some constant c. This result is obtained via a combination of multiple approaches, among others the algorithm for treewidth presented in Section 1.2. A crucial component of the algorithm is a data structure that is based on an already existing tree decomposition of slightly too large width. After initialization, this data structure can answer various queries for the input graph in $O(c^k \log n)$ time.

Chapter 2

Preliminaries

In this chapter we will introduce all the necessary definitions, as well as the notation used throughout the thesis. It should all be fairly standard and consistent with the union of Diestel [Die05] and Cygan et al. [CFK$^+$15].

2.1 Graphs

All graphs are finite, undirected and simple. Unless explicitly stated otherwise, they are also unweighted. A graph $G = (V, E)$ consists of a vertex set V and an edge set $E \subseteq [V]^2$. We will use $V(G)$ and $E(G)$ to denote the vertices and edges of the graph G. If the graph referred to is clear from the context or is defined as above, we will also simply use V and E. In such a case we will also use n for $|V(G)|$ and m for $|E(G)|$. For a vertex in V or an edge in E we will often refer to these as a vertex or an edge in G respectively.

Neighbors. Two vertices u and v are said to be *adjacent*, or *neighbors*, in G if $uv \in E$. An edge uv is said to be *incident* to the vertices u and v in G. For a set of edges $F \subseteq E(G)$ we will by $V_G(F)$ denote the set of vertices incident to F in G. For a vertex v of G, we use $N_G(v)$ to denote the *open neighborhood* of v, all the vertices adjacent to v in G. We also let $N_G[v] = N_G(v) \cup \{v\}$ denotes the *closed neighborhood* of v. These notions are extended to subsets of vertices as follows: $N_G[X] = \bigcup_{v \in X} N_G[v]$ and $N_G(X) = N_G[X] \setminus X$. Furthermore, we define the *degree* of a vertex v in G, denoted $\deg_G(v)$, as $|N_G(v)|$. For G we denote the minimum degree and maximum degree of the graph as $\delta(G)$ and $\Delta(G)$ respectively.

Subgraphs. For two graphs G and H, we say that H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Furthermore, we say that H is an *induced subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) = E(G) \cap [H]^2$. An induced subgraph of G whose vertices are X is denoted by $G[X]$. We lift the notion of neighborhoods to subgraphs by letting $N_G(H) = N_G(V(H))$ and $N_G[H] = N_G[V(H)]$.

Modifications. When discarding a set of vertices X from a graph G, we will use the notation $G - X$ for the graph $G[V(G) \setminus X]$. And furthermore, if we are removing a single vertex v we will write this as $G - v$, and this is short for $G - \{v\}$. Similarly, when $F \subseteq [V]^2$, we write $G - F$ to denote the graph $(V, E \setminus F)$. For two sets A and B we define the *symmetric difference* of A and B, denoted $A \oplus B$, as the set $(A \setminus B) \cup (B \setminus A)$. For a graph $G = (V, E)$ and $F \subseteq [V]^2$ we define $G \oplus F$ as the graph $(V, E \oplus F)$. In addition, if H is a subgraph of G and $F \subseteq E(G)$ we denote by $H \oplus F$ the graph $H \oplus (F \cap [V(H)]^2)$. Last, by $\overline{G}$ we denote the *complement* of a graph G, i.e., $V(\overline{G}) = V(G)$ and $E(\overline{G}) = [V(G)]^2 \setminus E(G)$.

Separation. We say that a set X of vertices of a graph G *separates* u from v if there is no path from u to v in $G - X$. We generalize this to sets in the natural way; Let X, Y and Z be sets of vertices of a graph G. We say that X separates Y from Z if there is no pair of vertices $y \in Y, z \in Z$ such that y is in the same connected component as z in $G - X$.

Twin classes. For a graph G and a vertex v we define the *true twin class* of v, denoted $\text{ttc}(v)$ as the set $\{u \in V(G) \mid N[u] = N[v]\}$. Similarly, we define the *false twin class* of v, denoted $\text{ftc}(v)$ as the set $\{u \in V(G) \mid N(u) = N(v)\}$. Observe that either $\text{ttc}(v) = \{v\}$ or $\text{ftc}(v) = \{v\}$. From this we define the *twin class* of v, denoted $\text{tc}(v)$ as $\text{ttc}(v)$ if $|\text{ttc}(v)| > |\text{ftc}(v)|$ and $\text{ftc}(v)$ otherwise.

Measures. We define the *distance* between two vertices u and v in G as the minimum number of edges in any path between the two and denote this by $\text{dist}(u, v)$. Similarly, for a vertex v in G and a set of vertices $X \subseteq V(G)$ we define the distance from v to X, denoted $\text{dist}(v, X)$ as $\min_{u \in X} \text{dist}(v, u)$. Given a set of vertices X and a non-negative integer r, we define the *ball around X of radius r*, denoted

$$B(X, r) = \{v \in V(G) \text{ such that } \text{dist}(v, X) \leq r\}.$$

We define the *diameter* of a graph G, denoted $\text{diam}(G)$, as $\max_{u,v \in V} \text{dist}(u, v)$ if G is connected and infinity otherwise. And the *degeneracy* of G, denoted by $\text{dgy}(G)$, as the smallest integer $d \in \mathbb{N}$ such that for every subgraph G' of G, $\delta(G') \leq d$. This is equivalent to saying that there is an ordering of the vertices of G, $v_1, v_2, \ldots, v_n$ such that $\deg_{G_i}(v_i) \leq d$ for every $i \leq n$ where G_i is the graph induced on the vertex set $V_i = \{v_i, v_{i+1}, \ldots, v_n\}$. By $\text{vc}(G)$ we will refer to the size of a minimum vertex cover of G. A *k-coloring* of a graph G is a function from $V(G)$ to $[1, k]$ such that two adjacent vertices are given different values. The *chromatic number* of G, denoted $\chi(G)$ is the minimum k such that there is a k-coloring of G.

Obstructions. An *obstruction set $\mathcal{H}$* is a set of graphs. Given an obstruction set $\mathcal{H}$, a graph G and an induced subgraph H of G we say that H is an *obstruction* in G if H is isomorphic to some element of $\mathcal{H}$. If there is no obstruction H in G we say that G is $\mathcal{H}$-free.

When the graph referred to is clear from the context, we will often skip the subscripts from the notations introduced above, for increased readability.

2.2 Graph classes

A *complete graph* is a graph G such that $E(G) = [V(G)]^2$. Furthermore, a vertex set X in G forms a *clique* if $G[X]$ is a complete graph. On the other side of the spectrum, a graph G is said to be *edgeless* if $E(G) = \emptyset$. And a set X in G forms an *independent set* if $G[X]$ is an edgeless graph. We will use the notation K_n for a complete graph on n vertices. And similarly the notation P_l to define a path on l vertices and C_l a cycle on l vertices. A vertex v is called *simplicial* if $N_G(v)$ forms a clique.

Trees, forests and caterpillars. A *forest* is a graph without cycles and a *tree* is a connected forest. A *caterpillar* is a tree T with a path B as a subgraph, such that all vertices of degree 3 or more lie on B. We then say that B is a *backbone* of T and every connected component of $T - B$ is a *stray* or a *hair*. We say that a caterpillar is of *stray length*, or *hair length*, s if there exists a backbone such that all strays are of size at most s.

Interval and chordal graphs. A graph is *chordal* if it has no induced cycle of length more than 3. It is well-known that if a graph is chordal it has a minimum width tree decomposition such that every bag forms a clique. An *interval graph* is a graph such that there exists a function from $V(G)$ into intervals of $\mathbb{N}$ such that the images of two vertices have a non-empty intersection if and only if the two vertices are adjacent. For every interval graph there is minimal width path decomposition such that every bag forms a clique.

Planar graphs. Informally, a graph is *planar* if there exists a continuous drawing such that no edges are crossing. Kuratowski's theorem states that a graph is planar if and only if it does not contain a subdivision of a K_5 nor a $K_{3,3}$ as a subgraph. For the purposes of this thesis we will only use that planar graphs does not contain these two graphs as subgraphs together with the following theorem.

Proposition 2.1 ([RSST97]). *Given a planar graph G one can in $O(n^2)$ time obtain a 4-coloring of G.*

Bipartite and split graphs. A *bipartite graph* is a graph whose vertex set can be partitioned into two independent sets. A *split graph* is a graph whose vertex set can be partitioned into a clique C and an independent set I; such a partition (C, I) is called a *split partition*. A split graph G with split partition (C, I) and edge set E is denoted by $G = (C, I, E)$. Note that, in general, a split graph can have more than one split partition.

Threshold and chain graphs. A *threshold graph* is a split graph $G = (C, I, E)$ such that the neighborhoods of the vertices in I are nested. In particular, for every $u, v \in I$ it holds that either $N(u) \subseteq N(v)$ or $N(v) \subseteq N(u)$. A *chain graph* is obtained by turning C into an independent set in a threshold graph. For more on these two graph classes we refer the reader to Section 7.1.

2.3 Tree decompositions and related measures

Recall from Section 1.2 that a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ of a graph G consists of a tree $T = (I, M)$ and a collection $\mathcal{X}$ of subsets of $V(G)$ that is indexed by I. Furthermore, it satisfies the following three conditions:

(i) $\bigcup_{X \in \mathcal{X}} X = V$,

(ii) for every edge $uv \in E$ there exists an $X \in \mathcal{X}$ such that $\{u, v\} \subseteq X$ and

(iii) for every $v \in V$ it holds that $T[\{i \mid v \in X_i\}]$ is connected.

The elements of $\mathcal{X}$ are often referred to as the bags of the decomposition. And when referring to the neighbors of a bag X_i, e.g. X_i is a leaf or X_i and X_j are adjacent bags, we are formally referring to properties of i and j in T. The width of a tree decomposition is the cardinality of the largest bag minus one. And the treewidth of a graph, denoted $\operatorname{tw}(G)$, is the minimum width over all tree decompositions. A *path decomposition* $\mathcal{P}$ of a graph is a tree decomposition such that T is a path. And the *pathwidth* of a graph G, denoted $\operatorname{pw}(G)$, is the minimum width over all path decompositions. For a graph G and a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ we define the *treelength* of a tree decomposition, denoted $\operatorname{tl}(G, \mathcal{T})$, as $\max\{\operatorname{dist}_G(u, v) \mid \{u, v\} \subseteq X \in \mathcal{X}\}$. In other words, the treelength of a tree decomposition is the maximum distance in G between two vertices that occur in the same bag in $\mathcal{T}$. We then define the *treelength* of a graph G, denoted $\operatorname{tl}(G)$, as the minimum treelength over all tree decompositions of G. The connected graphs of treewidth one are exactly the trees, the connected graphs of pathwidth one are the caterpillars of hair length one and the graphs of treelength one are the chordal graphs.

Rooted decompositions. When convenient, mostly for algorithmic purposes, we will consider the decomposition tree to be rooted. For each vertex v in G, we denote by T_v the subtree of T such that the corresponding bags all contain v. Furthermore, we will denote the root of this tree by r_v, i.e. the vertex in T_v closest to the root of T.

Non-redundant decompositions. Let $\mathcal{T} = (T, \mathcal{X})$ be a tree decomposition of a graph G. We say that $\mathcal{T}$ is *non-redundant* if for every two unique elements $X, Y \in \mathcal{X}$ it holds that neither is $X \subseteq Y$ nor is $Y \subseteq X$.

Proposition 2.2. *Given a graph G and a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ we can in $O(t^2 n)$ construct a non-redundant tree decomposition $\mathcal{T}' = (T', \mathcal{X}')$ such that*

(i) $\mathcal{X}' \subseteq \mathcal{X}$ *and*

(ii) $|V(T')| \leq |V(G)|$.

It follows from the definition of tree decompositions that one can obtain a non-redundant tree decomposition by contracting an edge in the tree decomposition whenever one of its endpoint bags is a subset of the other. Observe that the constructed tree decomposition satisfies the first condition by default. It is folklore that a non-redundant tree decomposition contains at most $|V(G)|$ bags (See Kleinberg & Tardos [KT06] for a proof). We observe that by property *(i)* both the width and the length of the new decomposition $\mathcal{T}'$ is bounded by the width and length of $\mathcal{T}$ respectively.

A bound on the number of edges. The following result will be used to bound the number of edges in a graph that is (supposedly) of bounded treewidth.

Lemma 2.3 ([BF05])**.** *For every graph G it holds that* $|E(G)| \leq \mathrm{tw}(G) \cdot |V(G)|$.

2.4 Parameterized complexity

We will here briefly recall the necessary concepts within parameterized complexity. We refer to Chapter 1 or the book Parameterized Algorithms [CFK+15] for the exact definitions and more information regarding the mentioned subjects.

Parameterized problems and their tractability A parameterized problem is a collection of pairs (x, k) where we refer to the number k as the parameter. A parameterized problem admits a fixed-parameter tractable algorithm if there is an algorithm solving the problem in $f(k)|x|^{O(1)}$ time for some computable function f. Problems admitting a fixed-parameter tractable algorithm are said to belong to the class FPT. Every parameterized problem solvable in $f(k)|x|^{g(k)}$ for two computable functions f and g, is contained in XP. The W-hierarchy consists of the complexity classes $\mathsf{FPT} = \mathsf{W[0]} \subseteq \mathsf{W[1]} \subseteq \mathsf{W[2]} \subseteq \ldots$ where all containments are believed to be strict. By proving that a problem is $\mathsf{W[1]}$-hard we rule out any fixed-parameter tractable algorithm for this problem, assuming $\mathsf{FPT} \neq \mathsf{W[1]}$. Such a proof is given by a reduction algorithm from a $\mathsf{W[1]}$-hard problem to the problem at hand, such that the reduction algorithm runs in fixed-parameter tractable time, the input and output instance are equivalent with respect to their languages and there is a computable function bounding the parameter of the output instance by the parameter of the input instance. We refer to Sections 1.1.2 and 1.1.4 for the formal definitions.

Kernels. A parameterized problem Π admits a *kernel* if there exists an algorithm that given an instance (x, k) of Π in polynomial time produces an instance (x', k') such that (x, k) is yes-instance of Π if and only if (x', k') is a yes-instance of Π and there is a computable function f such that both $|x'|$ and k' is bounded by $f(k)$. If f is a polynomial function, we say that the kernel is a *polynomial kernel* (see Section 1.3 and in particular Definition 1.6 for more information). It is known (Theorem 2) that all problems in FPT admits a kernel. However, assuming $\mathsf{NP} \not\subseteq \mathsf{coNP/poly}$, we can prove that certain problems do not admit polynomial kernels utilizing the concept of a cross-composition (Definition 1.10).

Faster algorithms and lower bounds. We say that a parameterized problem admits a *subexponential time algorithm* if given an instance (x, k) the algorithm solves the problem in $2^{o(k)}n^{O(1)}$ time. Assuming the Exponential Time Hypothesis, we can prove that some problems does not admit subexponential time algorithms, as well as algorithms of other complexities. We refer to Section 1.4 for more information on subexponential time algorithms and Section 1.5 for more on the Exponential Time Hypothesis and its applications.

Approximation algorithms within parameterized complexity. A parameterized problem is said to admit a *fixed-parameter tractable approximation algorithm* if there is an algorithm that runs in fixed-parameter tractable time and either outputs a solution of quality that depends on the parameter or correctly concludes that the given instance is a no-instance. We refer to Section 1.6 for more information on the subject.

2.5 Miscellaneous

For $\alpha \in \mathbb{N}$, the function $\log^{(\alpha)} n$ is defined as follows:

$$\log^{(\alpha)} n = \begin{cases} \log n & \text{if } \alpha = 1 \\ \log(\log^{(\alpha-1)} n) & \text{otherwise.} \end{cases}$$

If a function f is defined on a set X and $Y \subseteq X$ we will use the notation $f(Y)$ for $\cup_{y \in Y} f(y)$. For intervals of natural numbers we will use the notation $[n]$ for the interval $[1, \ldots, n]$.

2.6 Problems

We now enumerate the most relevant problems for this thesis.

Vertex subset problems

Vertex Cover

Input: A graph $G = (V, E)$ and an integer k.
Question: Is there an $X \subseteq V$ with $|X| \leq k$ such that $G - X$ is an edgeless graph?

Clique

Input: A graph $G = (V, E)$ and an integer k.
Question: Is there an $X \subseteq V$ with $|X| = k$ such that X forms a clique?

Dominating Set

Input: A graph $G = (V, E)$ and an integer k.
Question: Is there an $X \subseteq V$ with $|X| \leq k$ such that $N[X] = V$?

Vertex Integrity

Input: A graph $G = (V, E)$ and an integer p.
Question: Is there an $X \subseteq V$ such that every connected component of $G - X$ has at most $p - |X|$ vertices?

Component Order Connectivity

Input: A graph $G = (V, E)$ and two integers k and ℓ.
Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that every connected component of $G - X$ has at most ℓ vertices?

$\mathcal{H}$-free Vertex Deletion

Input: A graph $G = (V, E)$ and an integer k.
Question: Is there an $X \subseteq V$ with $|X| \leq k$ such that $G - X$ is $\mathcal{H}$-free?

Edge modification problems

For all problems below we assume $\mathcal{H}$ to be a finite set of graphs.

Threshold Editing

Input:	A graph $G = (V, E)$ and an integer k.
Question:	Is there a set $F \subseteq [V]^2$ with $\lvert F\rvert \leq k$ such that $G \oplus F$ is a threshold graph?

Chain Editing

Input:	A graph $G = (V, E)$ and an integer k.
Question:	Is there a set $F \subseteq [V]^2$ with $\lvert F\rvert \leq k$ such that $G \oplus F$ is a chain graph?

$\mathcal{H}$-free Edge Editing

Input:	A graph $G = (V, E)$ and an integer k.
Question:	Is there a set $F \subseteq [V]^2$ with $\lvert F\rvert \leq k$ such that $G \oplus F$ is $\mathcal{H}$-free?

For all three problems above we similarly define their completion and deletion variants by requiring the set F to be in $[V]^2 \setminus E$ and E respectively.

Structural problems

Bandwidth

Input:	A graph $G = (V, E)$ and an integer b.
Question:	Is there a linear ordering α of V such that for every $uv \in E$ it holds that $\lvert\alpha(u) - \alpha(v)\rvert \leq b$?

Treewidth

Input:	A graph $G = (V, E)$ and an integer t.
Question:	Is $\operatorname{tw}(G) \leq t$?

Part II

Vulnerability measures

Chapter 3

Introduction

Motivated by a multitude of practical applications, many different vulnerability measures of graphs have been introduced in the literature over the past few decades. The vertex and edge connectivity of a graph, although undoubtedly being the most well-studied of these measures, often fail to capture the more subtle vulnerability properties of networks that one might wish to consider, such as the number of resulting components, the size of the largest or smallest component that remains, and the largest difference in size between any two remaining components. The two vulnerability measures we study in this chapter, *vertex integrity* and *component order connectivity*, take into account not only the number of vertices that need to be deleted in order to break a graph into pieces, but also the number of vertices in the largest component that remains.

The *vertex integrity* of an unweighted graph G is defined as $\iota(G) = \min\{|X| + n(G - X) \mid X \subseteq V(G)\}$, where $n(G - X)$ is the number of vertices in the largest connected component of $G - X$. This vulnerability measure was introduced by Barefoot, Entringer, and Swart [BES87] in 1987. For an overview of structural results on vertex integrity, including combinatorial bounds and relationships between vertex integrity and other vulnerability measures, we refer the reader to a survey on the subject by Bagga et al. [BBG$^+$92]. We mention here only known results on the computational complexity of determining the vertex integrity of a graph.

The Vertex Integrity (VI) problem takes as input an n-vertex graph G and an integer p, and asks whether $\iota(G) \leq p$. This problem was shown to be NP-complete, even when restricted to planar graphs, by Clark, Entringer, and Fellows [CEF87]. On the positive side, Fellows and Stueckle [FS89] showed that the problem can be solved in $O(p^{3p}n)$ time, and is thus fixed-parameter tractable when parameterized by p. In the aforementioned survey, Bagga et al. [BBG$^+$92] mention that Vertex Integrity can be solved in $O(n^3)$ time when the input graph is a tree or a cactus graph. Kratsch, Kloks, and Müller [KKM97] provided polynomial time algorithms for several other graph classes.

Ray and Deogun [RD94] were the first to study the more general Weighted Vertex Integrity (wVI) problem. This problem takes as input an n-vertex graph G, a weight function $w : V(G) \to \mathbb{N}$, and an integer p. The task is to decide if there exists a set $X \subseteq V(G)$ such that the weight of X plus the weight of a heaviest component of $G - X$ is at most p. Using a reduction from 0-1 Knapsack, Ray and Deogun [RD94] identified several graph classes on which the Weighted Vertex Integrity problem is weakly NP-complete. In particular, their result implies that the problem is weakly NP-complete on trees, bipartite graphs, series-parallel graphs, and regular graphs, and therefore also on superclasses such as chordal graphs and comparability graphs.

We now turn our attention to the second vulnerability measure studied in this chapter. For any positive integer ℓ, the *ℓ-component order connectivity* of a graph G is defined to be the cardinality of a smallest set $X \subseteq V(G)$ such that $n(G - X) < \ell$. We refer to the survey by Gross et al. [GHI+13] for more background on this graph parameter. Motivated by the definitions of ℓ-component order connectivity and the Weighted Vertex Integrity problem, we introduce the Weighted Component Order Connectivity (wCOC) problem. This problem takes as input a graph G, a weight function $w : V(G) \to \mathbb{N}$, and two integers k and ℓ. The task is to decide if there exists a set $X \subseteq V(G)$ such that the weight of X is at most k and the weight of a heaviest component of $G - X$ is at most ℓ. Observe that the Weighted Component Order Connectivity problem can be interpreted as a more refined version of Weighted Vertex Integrity. We therefore find it surprising that, to the best of our knowledge, the Weighted Component Order Connectivity problem has not yet been studied in the literature. We do however point out that the techniques described by Kratsch et al. [KKM97] yield polynomial-time algorithms for the unweighted version of the problem on interval graphs, circular-arc graphs, permutation graphs, and trapezoid graphs, and that very similar problems have received some attention recently [BAMSN13, GHI+13].

A parameterized perspective

Observe that an instance (G, k, ℓ) of the unweighted problem Component Order Connectivity with $\ell = 1$ is equivalent to the instance (G, k) of Vertex Cover. This, together with the NP-hardness of Vertex Integrity [CEF87] gives that all of the above problems are NP-complete on planar graphs. Motivated by this, we study them by applying the flow chart in Figure 1.2. Recall that Fellows and Stueckle [FS89] proved that VI can be solved in $O(p^{3p}n)$ time on general graphs. In Chapter 4 we prove that even Weighted Vertex Integrity is in FPT and admits a polynomial kernel when parameterized by p.

We then consider the parameterized complexity of COC and wCOC. We first prove that when parameterizing COC by either k or ℓ, the problem is W[1]-hard, even when restricted to split graphs. By changing the parameter to $k + \ell$, we manage to prove that for this dual parameterization, wCOC is indeed in FPT by a simple branching argument. Somewhat surprisingly, we then prove that it is not possible to improve significantly over this algorithm assuming ETH. Finally, we show that wCOC admits a polynomial kernel with at most $k\ell(k + \ell) + k$ vertices, where each vertex has weight at most $k + \ell$.

The arguments for these results are clean and simplistic and we hence believe that they are a nice introduction to several of the concepts that are to appear in this thesis and that was surveyed in Chapter 1.

3.1 Preliminaries

Let G be a graph, $V(G)$ and $E(G)$ denote the vertex set and edge set of G, respectively, and $w : V(G) \to \mathbb{N}_+ = \{1, 2, \ldots\}$ a weight function on the vertices of G. The weight of a subset $X \subseteq V(G)$ is defined as $w(X) = \sum_{v \in X} w(v)$. We define $w_{\text{cc}}(G)$ to be the weight of a heaviest component of G, i.e., $w_{\text{cc}}(G) = \max\{w(V(G_i)) \mid 1 \leq i \leq r\}$, where $G_1, \ldots, G_r$ are the components of G. The *weighted vertex integrity* of G is defined as

$$\iota(G) = \min\{w(X) + w_{\text{cc}}(G - X) \mid X \subseteq V(G)\},$$

where $G - X$ denotes the graph obtained from G by deleting all the vertices in X. Any set $X \subseteq V(G)$ for which $w(X) + w_{\text{cc}}(G - X) = \iota(G)$ is called an *ι-set* of G. We consider the following two decision problems:

Weighted Vertex Integrity

Input:	A graph G, a weight function $w : V(G) \to \mathbb{N}_+$, and an integer p.
Question:	Is $\iota(G) \leq p$?

Weighted Component Order Connectivity

Input:	A graph G, a weight function $w : V(G) \to \mathbb{N}_+$, and two integers k and ℓ.
Question:	Is there a set $X \subseteq V(G)$ with $w(X) \leq k$ such that $w_{\text{cc}}(G - X) \leq \ell$?

The unweighted versions of these two problems, where $w(v) = 1$ for every vertex $v \in V(G)$, are called Vertex Integrity (VI) and Component Order Connectivity (COC), respectively.

Chapter 4

Vertex Integrity

In this chapter we prove that WEIGHTED VERTEX INTEGRITY as well as VERTEX INTEGRITY both are in FPT when parameterized by p. Furthermore, we provide a cubic vertex kernel.

4.1 A fixed-parameter tractable algorithm

We will now present a fixed-parameter tractable algorithm for WEIGHTED VERTEX INTEGRITY. First the algorithm searches for a connected set of vertices U of weight at least $p+1$. Observe that at least one of the vertices in U will have to be in a solution X. Based on this, for every every vertex v in U we try to include this vertex in X and recurse on $G-v$ together with $p-w(v)$. But before we give the algorithm, we will prove that we can assume the instance to have at most pn edges.

Lemma 4.1. *Given an instance (G, w, p) of* WEIGHTED VERTEX INTEGRITY *there is an algorithm that in $O(pn)$ time outputs an equivalent instance (G', w, p) such that $|E(G')| \leq (p-1)|V(G')|$.*

Proof. Suppose that (G, w, p) is a yes-instance. Then there exists a set $X \subseteq V(G)$ such that $w(X) + w_{\text{cc}}(G-X) \leq p$. Let $G_1, \dots, G_r$ be the components of $G-X$. Since every vertex has weight at least 1, it holds that $|X \cup V(G_i)| \leq w(X \cup V(G_i)) \leq p$ for each $i \in \{1, \dots, r\}$. Observe that G has a path decomposition of width at most $p-1$ whose bags are exactly the sets $X \cup V(G_i)$. This implies that the pathwidth, and hence the treewidth, of G is at most $p-1$. It is well-known that every n-vertex graph of treewidth at most t has at most tn edges [BF05]. We thus conclude that if (G, w, p) is a yes-instance, then $m \leq (p-1)n$. Our algorithm can therefore start counting edges and either output the given instance if $m \leq (p-1)n$ and otherwise a constant no-instance (G', w', p') with $|E(G')| \leq (p-1)|V(G')|$. $\square$

Theorem 13. WEIGHTED VERTEX INTEGRITY *can be solved in $O((p+1)^p n)$ time.*

Proof. Let (G, w, p) be an instance of Weighted Vertex Integrity. First, we apply Lemma 4.1 to ensure that $m \leq (p-1)n$. We now describe a simple branching algorithm that solves the problem. At each step of the algorithm, we use a depth-first search to find a set U of at most $p+1$ vertices such that $G[U]$ is connected and $w(U) \geq p+1$. If such a set does not exist, then every component of the graph under consideration has weight at most p, so the empty set is an ι-set of the graph and we are done. Otherwise, we know that any ι-set of the graph contains a vertex of U. We therefore branch into $|U| \leq p+1$ subproblems: for every $v \in U$, we create the instance $(G-v, w, p-w(v))$, where we discard the instance in case $p - w(v) < 0$. The parameter p decreases by at least 1 at each branching step and the algorithm terminates at depth $p-1$. Due to this, the search tree of the algorithm has depth at most $p-1$. Since T is a $(p+1)$-ary tree, it contains $((p+1)^p - 1)/p = O((p+1)^{p-1})$ nodes in total. Due to the assumption that $m \leq (p-1)n$, the depth-first search at each step can be performed in time $O(pn)$. This yields an overall running time of $O((p+1)^{p-1}pn) = O((p+1)^p n)$. □

4.2 A polynomial kernel

In this section we prove that Weighted Vertex Integrity admits a polynomial kernel with at most p^3 many vertices. By Lemma 4.1 this immediately yields a p^4 bound on the number of edges. The kernel consists of two reduction rules, followed by a combinatorial argument showing that after these two rules have been exhaustively applied, there are at most p^3 vertices left in any yes-instance.

Rule 4.1. *Let (G, w, p) be an instance of* Weighted Vertex Integrity *and $C_1, \ldots, C_r$ the connected components of G in decreasing order with respect to total weight. If $r > p+1$ we output the instance (G', w, p) where $G' = G[\cup_{i=1}^{p+1} C_i]$.*

In other words, we remove all but the $p+1$ heaviest components of the instance. The intuition is that our solution can interact with at most p of the components and that this interaction better be among the heavier components. We observe that the rule can be applied in $O(n+m)$ time.

Lemma 4.2. *Let (G, w, p) be an instance of* Weighted Vertex Integrity *and apply Rule 4.1 to obtain the instance (G', w, p). Then (G, w, p) is a yes-instance if and only if (G', w, p) is a yes-instance.*

Proof. To prove that this rule is safe, it suffices to prove that (G, w, p) is a yes-instance if the new instance (G', w, p) is a yes-instance, as the reverse direction trivially holds. Suppose (G', w, p) is a yes-instance. Then there exists a set $X \subseteq V(G')$ such that $w(X) + w_{\mathrm{cc}}(G' - X) \leq p$. Since $|X| \leq w(X) \leq p$ and G' has exactly $p+1$ components, there exists an index $i \in \{1, \ldots, p+1\}$ such that X does not contain any vertex from G_i. Since $w_{\mathrm{cc}}(G_i) \geq w_{\mathrm{cc}}(G_j)$ for every $j \in \{p+2, \ldots, r\}$, it holds that $w_{\mathrm{cc}}(G - X) = w_{\mathrm{cc}}(G' - X)$. Hence

$w(X) + w_{\mathrm{cc}}(G - X) = w(X) + w_{\mathrm{cc}}(G' - X) \leq p$, implying that $\iota(G) \leq p$ and that (G, w, p) is a yes-instance.

□

The second reduction rule is along the lines of the classic high degree reduction rule for Vertex Cover. If the closed neighborhood of a vertex induces some connected graph that cannot be fixed without separating the vertices, it is safe to include this vertex into your solution.

Rule 4.2. *Let (G, w, p) be an instance of* Weighted Vertex Integrity *and $v \in V$ a vertex such that $w(N[v]) > p$. If $p - w(v) \geq 0$ we output the instance $(G - v, w, p - w(v))$ and otherwise a trivial no-instance.*

Lemma 4.3. *Let (G, w, p) be an instance of* Weighted Vertex Integrity *and apply Rule 4.2 to obtain the instance (G', w', p'). Then (G, w, p) is a yes-instance if and only if (G', w', p') is a yes-instance.*

Proof. It suffices to show that if (G, w, p) is a yes-instance, then v belongs to any ι-set of G. Suppose (G, w, p) indeed is a yes-instance, and let X be an ι-set of G. Then $w(X) + w_{\mathrm{cc}}(G - X) = \iota(G) \leq p$. For contradiction, suppose that $v \notin X$. Consider the component H of $G - X$ that contains v. Since every vertex of $N_G[v]$ belongs either to X or to the component H, and $w(N_G[v]) > p$ by assumption, we find that $w(X) + w_{\mathrm{cc}}(H) > p$. But this implies that $w(X) + w_{\mathrm{cc}}(G - X) \geq w(X) + w_{\mathrm{cc}}(H) > p$, yielding the desired contradiction.

□

We will now prove that the two rules given are sufficient to obtain a polynomial kernel with the promised bounds.

Theorem 14. Weighted Vertex Integrity *admits a kernel with at most p^3 vertices and p^4 edges, where each vertex has weight at most p.*

Proof. Let (G, w, p) be our input instance and (G', w, p') the instance obtained after exhaustively applying Rules 4.1 and 4.2. Observe that $p' \leq p$. We assume that $p \geq 2$, as otherwise we can trivially solve the instance (G, w, p). We claim that if (G', w, p') is a yes-instance, then $|V(G')| \leq p^3$. Suppose (G', w, p') is a yes-instance, and let $X \subseteq V(G')$ be an ι-set of G'. Then $w(X) + w_{\mathrm{cc}}(H) \leq p' \leq p$ for every component H of $G' - X$. This, together with the fact that every vertex in G' has weight at least 1, implies that $|X| \leq p$ and $|V(H)| \leq w_{\mathrm{cc}}(H) \leq p - w(X) \leq p - |X|$ for every component H of $G' - X$. Since the first reduction rule cannot be applied on the instance (G', w, p'), we know that G' has at most $p' + 1 \leq p + 1$ components. If $X = \emptyset$, then each of these components contains at most p vertices, so $|V(G')| \leq (p + 1)p \leq p^3$, where the last inequality follows from the assumption that $p \geq p' \geq 2$.

Now suppose $|X| \geq 1$. Observe that every vertex in X has degree at most $p' \leq p$ due to the assumption that the second reduction rule cannot be applied. Hence, every vertex of X is adjacent to at most p components of $G' - X$, implying that there are at most p^2 components of $G' - X$ that are adjacent to X. Since G' itself has at most $p + 1$ components, at least one of which contains a vertex of X, we find that $G' - X$ has at most $p^2 + p$ components in total. Recall that each of these components contains at most $p - |X|$ vertices. We conclude that $|V(G')| \leq (p^2 + p)(p - |X|) + |X| \leq p^3$, where we use the assumption that $|X| \geq 1$. Due to the second reduction rule, each vertex in G' has weight at most p.

Finally, we apply the algorithm of Lemma 4.1 to bound the number of edges by $p \cdot p^3 = p^4$. It remains to argue that our kernelization algorithm runs in polynomial time. Observe that the execution of any reduction rule strictly decreases the number of vertices in the graph, so each rule is applied only a polynomial number of times. The observation that each rule can be executed in polynomial time completes the proof.

$\square$

Chapter 5

Component Order Connectivity

We will now investigate the parameterized complexity and kernelization complexity of Component Order Connectivity and Weighted Component Order Connectivity. As mentioned in the introduction, both problems are para-NP-hard when parameterized by ℓ due to the fact that Component Order Connectivity is equivalent to Vertex Cover when $\ell = 1$. First, we will prove that when restricted to split graphs, both problems are W[1]-hard when parameterized by k or by ℓ. After this, we will prove that both problems are fixed-parameter tractable when parameterized by both k and ℓ. We will then prove that this algorithm is optimal, before we end with a polynomial kernel.

5.1 Hardness on split graphs

We will now prove that parameterized by either k or ℓ, there is little hope for tractability. More specifically, we will prove that by both the aforementioned parameterizations the problem is W[1]-hard, even when restricted to split graphs.

Given a graph G, the *incidence split graph* of G is the split graph $G^* = (C^*, I^*, E^*)$ whose vertex set consists of a clique $C^* = \{v_x \mid x \in V(G)\}$ and an independent set $I^* = \{v_e \mid e \in E(G)\}$, and where two vertices $v_x \in C^*$ and $v_e \in I^*$ are adjacent if and only if the vertex x is incident with the edge e in G. The following lemma will be used in the proofs of hardness results.

Lemma 5.1. *Let $G = (V, E)$ be a graph, $G^* = (C^*, I^*, E^*)$ its incidence split graph, and $k < |V|$ a non-negative integer. Then the following statements are equivalent:*

(i) G has a clique of size k;

(ii) there exists a set $X \subseteq C^*$ such that $|X| \leq k$ and $|X| + n(G^* - X) \leq |V| + |E| - \binom{k}{2}$;

(iii) there exists a set $X \subseteq C^*$ such that $|X| \leq k$ and $n(G^* - X) \leq |V| + |E| - \binom{k}{2} - k$.

Proof. Let $n = |V|$ and $m = |E|$. We first prove that (i) implies (iii). Suppose G has a clique S of size k. Let $X = \{v_x \in C^* \mid x \in S\}$ denote the set of vertices in G^* corresponding to the vertices of S. Similarly, let $Y = \{v_e \in I^* \mid e \in E(G[S])\}$ denote the set of vertices in G^* corresponding to the edges in G both endpoints of which belong to S. Observe that $|Y| = \binom{k}{2}$ due to the fact that S is a clique of size k in G. Now consider the graph $G^* - X$. In this graph, every vertex of Y is an isolated vertex, while every vertex of $I^* \setminus Y$ has at least one neighbor in the clique $C^* \setminus X$. This implies that $n(G^* - X) = n + m - \binom{k}{2} - k$.

Since (iii) trivially implies (ii), it remains to show that (ii) implies (i). Suppose there exists a set $X \subseteq C^*$ such that $|X| \leq k$ and $|X| + n(G^* - X) \leq |V| + |E| - \binom{k}{2}$. Let $Z \subseteq I^*$ be the set of vertices in I^* both neighbors of which belong to X. Observe that $|Z| \leq \binom{|X|}{2}$ and $n(G^* - X) = n + m - |X| - |Z|$. Hence

$$n + m - \binom{k}{2} \geq |X| + n(G^* - X) = n + m - |Z| \geq n + m - \binom{|X|}{2},$$

which implies that $\binom{k}{2} \leq \binom{|X|}{2}$. Since $|X| \leq k$ by assumption, we find that $|X| = k$ and all the above inequalities must be equalities. In particular, we find that $|Z| = \binom{|X|}{2}$. We conclude that the vertices in G that correspond to X form a clique of size k in G. □

Theorem 15. *Component Order Connectivity is* W[1]*-hard on split graphs when parameterized by* k.

Proof. We give a reduction from the W[1]-hard problem Clique. Let (G, k) be an instance of Clique with $n = |V(G)|$ and $m = |E(G)|$. Let $G^* = (C^*, I^*, E^*)$ be the split incidence graph of G, and let $\ell = n + m - \binom{k}{2}$. By Lemma 5.1, there is a clique of size k in G if and only if there exists a set $X \subseteq C^*$ such that $|X| \leq k$ and $n(G^* - X) \leq n + m - \binom{k}{2}$. This immediately implies that (G, k) is a yes-instance of Clique if and only if (G^*, k, ℓ) is a yes-instance of Component Order Connectivity. □

Theorem 16. *Component Order Connectivity is* W[1]*-hard on split graphs when parameterized by* ℓ.

Proof. We give a reduction from Clique. Let (G, q) be an instance of Clique, and construct $G^\dagger = (V^\dagger, E^\dagger)$, where $V^\dagger = \{v_x \mid x \in V(G)\} \cup \{w_e \mid e \in E(G)\}$ and $E^\dagger = \{w_{e_1} w_{e_2} \mid e_1, e_2 \in E(G)\} \cup \{v_x w_e \mid \text{vertex } x \text{ incident to edge } e \text{ in } G\}$. Define $C^\dagger = \{v_e \mid e \in E(G)\}$ and $I^\dagger = V^\dagger \setminus C^\dagger$. We also define $k = |E(G)| - \binom{q}{2}$ and $\ell = \binom{q}{2} + q$. We will show that (G, q) is a yes-instance of Clique if and only if $(G^\dagger, k, \ell)$ is a yes-instance of COC.

First assume (G, q) is a yes-instance of Clique, and let $Q \subseteq V(G)$ be a clique of size q. Define $Q^\dagger = \{w_e \mid e = uv \text{ for } u, v \in Q\}$. Let $X^\dagger = C^\dagger \setminus Q^\dagger$ and consider $|X^\dagger|$ and $G^\dagger - X^\dagger$. Observe that $|X^\dagger| = |C^\dagger \setminus Q^\dagger| = |C^\dagger| - |Q^\dagger| = |E(G)| - \binom{q}{2} = k$. Also note that the neighborhood of $Q^\dagger$ in $I^\dagger$ has size exactly q. Hence the component of $G^\dagger - X^\dagger$ containing the vertices of $Q^\dagger$ has $|Q^\dagger| + q = \binom{q}{2} + q = \ell$ vertices, while every other component of $G^\dagger - X^\dagger$ contains exactly one vertex. This implies that $(G^\dagger, k, \ell)$ is a yes-instance of COC.

For the reverse direction, suppose that $(G^\dagger, k, \ell)$ is a yes-instance of COC. Then there exists a set $X^\dagger \subseteq V^\dagger$ such that $|X^\dagger| \leq k$ and $n(G^\dagger - X^\dagger) \leq \ell$; let us call such a set $X^\dagger$ a *deletion set*. Without loss of generality, assume that among all deletion sets, $X^\dagger$ contains the smallest number of vertices from $I^\dagger$. We claim that $X^\dagger \cap I^\dagger = \emptyset$, i.e., $X^\dagger \subseteq C^\dagger$.

For contradiction, suppose there is a vertex $v \in X^\dagger \cap I^\dagger$. If all the neighbors of v belong to $X^\dagger$, then $X^\dagger \setminus \{v\}$ is a deletion set, contradicting the choice of $X^\dagger$. Hence we may assume that there exists a vertex $w \in N_{G^\dagger}(v) \setminus X^\dagger$. Let D be the component of $G^\dagger - X^\dagger$ containing w. Observe that every component of $G^\dagger - X^\dagger$ other than D has exactly one vertex, so $|V(D)| = n(G^\dagger - X^\dagger)$. Let $X' = X^\dagger \setminus \{v\}$, and let D' be the component of $G^\dagger - X'$ containing v and w. It is clear that $|V(D')| = |V(D)| + 1$ and all components of $G^\dagger - X'$ other than D' contain exactly one vertex. Finally, let $X'' = (X^\dagger \setminus \{v\}) \cup \{w\}$. Then every component of $G^\dagger - X''$ has at most $|V(D')| - 1 \leq |V(D)|$ vertices, implying that $n(G^\dagger - X'') \leq n(G^\dagger - X^\dagger)$. Hence X'' is a deletion set, contradicting the choice of $X^\dagger$. This contradiction proves that $X^\dagger \subseteq C^\dagger$.

Observe that $|C^\dagger \setminus X^\dagger| = |C^\dagger| - |X^\dagger| \geq |E(G)| - k = \binom{q}{2}$. Let $Q^\dagger$ be any subset of $C^\dagger \setminus X^\dagger$ of size $\binom{q}{2}$. Let D be the component of $G^\dagger - X^\dagger$ containing $Q^\dagger$. Since $X^\dagger$ is a deletion set, $|V(D)| \leq \ell = \binom{q}{2} + q$. This implies that $Q^\dagger$ has at most q neighbors in $I^\dagger$. By construction of $G^\dagger$, it holds that $Q^\dagger$ has exactly q neighbors in $I^\dagger$. These q neighbors correspond to a clique of size q in G. □

5.2 An optimal algorithm

Since we have just proved that the problems at hand are W[1]-hard when considering either k or ℓ as the parameter, we will now focus our attention to the problem parameterized by both k and ℓ. At first, this might appear to not be in line with our definition of a parameterized problem (Definition 1.1). However, for classifications with respect to containment in FPT or whether the problem admits a polynomial kernel, we can consider the parameter as $k + \ell$. In this section we will first give a simple branching algorithm and then prove that no major improvement is possible unless ETH fails.

Lemma 5.2. *Given an instance (G, w, k, ℓ) of* Weighted Component Order Connectivity *there is an algorithm that in $O(pn)$ time outputs an equivalent instance (G', w, k, ℓ) such that $|E(G')| \leq (k + \ell)|V(G')|$.*

Proof. Observe that if an instance (G, w, k, ℓ) of Weighted Component Order Connectivity is a yes-instance, then the $(G, w, k + \ell)$ is a yes-instance of Weighted Vertex Integrity. Hence, the same argument as in the proof of Lemma 4.1 applies.

□

Theorem 17. *There is an algorithm solving* Weighted Component Order Connectivity *in time $O((\ell+1)^k(k+\ell)n) = 2^{O(k\log\ell)}n$.*

Proof. We describe a simple branching algorithm that solves the problem. At each step of the algorithm, we use a depth-first search to find a set $U \subseteq V(G)$ of at most $\ell+1$ vertices such that $w_{\mathrm{cc}}(G[U]) \geq \ell+1$ and $G[U]$ induces a connected subgraph. If such a set does not exist, then every component of the graph has weight at most ℓ, so we are done. Otherwise, we know that any solution contains a vertex of U. We therefore branch into $|U| \leq \ell+1$ subproblems: for every $v \in U$, we create the instance $(G-v, w, k-w(v), \ell)$, where we discard the instance in case $k-w(v) < 0$. Since the parameter k decreases by at least 1 at each branching step, the corresponding search tree T has depth at most k. Since T is an $(\ell+1)$-ary tree of depth at most k, it has at most $((\ell+1)^{k+1}-1)/((\ell+1)-1) = O((\ell+1)^k)$ nodes. Due to the assumption that $m \leq (k+\ell-1)n$, the depth-first search at each step can be performed in time $O(n+m) = O((k+\ell)n)$. This yields an overall running time of $O((\ell+1)^k(k+\ell)n) = 2^{O(k\log\ell)}n$.

□

The next result shows that even if the input graph is a split graph, we cannot significantly improve the algorithm above assuming ETH. In particular we prove that, under ETH, no $2^{o(k\log\ell)}n^{O(1)}$-time algorithm can exist.

Theorem 18. *There is no $2^{o(k\log\ell)}n^{O(1)}$ time algorithm for* Component Order Connectivity*, even when restricted to split graphs, unless the ETH fails.*

Proof. For a contradiction, suppose there exists an algorithm $\mathbb{A}$ solving Component Order Connectivity in time $2^{o(k\log\ell)}n^{O(1)}$. Let $(G, \mathcal{X})$ be an instance of $k \times k$-Clique, where $\mathcal{X} = \{X_1, \dots, X_k\}$. We assume that G contains no edge whose endpoints belong to the same set X_i, as an equivalent instance can be obtained by deleting all such edges from G. Due to this assumption, it holds that $(G, \mathcal{X})$ is a yes-instance of $k \times k$-Clique if and only if G contains a clique of size k.

Now let $G^* = (C^*, I^*, E^*)$ be the incidence split graph of G, and let $\ell = |V(G)| + |E(G)| - \binom{k}{2}$. By the definition of $k \times k$-Clique, we have that $|V(G)| = k^2$ and $|E(G)| \leq k^2(k^2-1)/2$. This implies that the graph G^* has at most $k^2 + k^2(k^2-1)/2 \leq k^4$ vertices, and that $\ell \leq k^4$. By Lemma 5.1, it holds that (G^*, k, ℓ) is a yes-instance of Component Order Connectivity if and only if G has a clique of size k. Hence, using algorithm $\mathbb{A}$, we can decide in time $2^{o(k\log k^4)}k^{O(1)} = 2^{o(k\log k)}$ whether or not $(G, \mathcal{X})$ is a yes-instance of $k \times k$-Clique, which by Theorem 7 is only possible if the ETH fails.

□

5.3 A polynomial kernel

We conclude this chapter by showing that the Weighted Component Order Connectivity problem admits a polynomial kernel. The arguments in the proof of Theorem 19 are similar to, but slightly different from, those in the proof of Theorem 14. The rules of this kernel lie even closer to the kernel for Vertex Cover displayed in Section 1.3.1. In particular, if a connected component is small, we remove it from the graph. And if the closed neighborhood of a vertex is too costly to fix without removing the vertex itself, we immediately remove it and adjust k accordingly.

Rule 5.1. *Let (G, w, k, ℓ) be an instance of wCOC and C a connected component of G such that $w(C) \leq \ell$. We then reduce the instance to $(G - C, w, k, \ell)$.*

The correctness of the above rule follows immediately from the observation that no minimal solution deletes any of the vertices of C and hence the entire component is irrelevant.

Rule 5.2. *Let (G, w, k, ℓ) be an instance of wCOC and $v \in V$ a vertex such that $w(N[v]) > k + \ell$. If $k - w(v) \geq 0$ we output the instance $(G - v, w, k - w(v), \ell)$ and otherwise a trivial no-instance.*

Lemma 5.3. *Let (G, w, k, ℓ) be an instance of wCOC and $v \in V$ a vertex such that $w(N[v]) > k + \ell$. Then v is contained in every solution X.*

Proof. Let us first show that v belongs to any solution for the instance (G, w, k, ℓ) if such a solution exists. This follows from the observation that deleting any set $X \subseteq V(G) \setminus \{v\}$ with $w(X) \leq k$ from G yields a graph G' such that $w(N_{G'}[v]) > \ell$. For the same reason, there exists no solution if $w(v) > k$. □

Theorem 19. Weighted Component Order Connectivity *admits a kernel with at most $k\ell(k + \ell) + k$ vertices, where each vertex has weight at most $k + \ell$.*

Proof. Given an instance (G, w, k, ℓ) let (G', w, k', ℓ) be the instance that we obtain after exhaustively applying Rules 5.1 and 5.2. Observe that $k' \leq k$, while the parameter ℓ did not change in the kernelization process.

Suppose X is a solution for this instance. Then $w(X) \leq k' \leq k$, which implies that X contains at most k vertices. For every component H of $G' - X$, it holds that $|H| \leq w_{\text{cc}}(H) \leq \ell$, furthermore H is adjacent to at least one vertex of X, as otherwise our second reduction rule could have been applied. Moreover, the fact that the first reduction rule cannot be applied guarantees that $w(N_G[v]) \leq k + \ell$ for every $v \in V(G')$. In particular, this implies that every vertex in X has degree at most $k + \ell$. We find that $G - X$ has at most $k(k + \ell)$ components, each containing at most ℓ vertices. We conclude that if (G', w, k', ℓ') is a yes-instance, then $|V(G')| \leq k\ell(k + \ell) + k$. The observation that each vertex in G' has weight at most $k + \ell$ due to the second reduction rule completes the proof. □

Chapter 6

Concluding remarks

In this part we have proved Vertex Integrity and Component Order Connectivity (as well as their weighted variants) to be in FPT and to admit polynomial kernels. Kumar & Lokshtanov [KL16] recently observed that one can improve the running time of the algorithm to $(\ell + 0.0755)^k n^{O(1)}$ by reducing the instance to an instance of $(l + 1)$-Hitting Set. We show that both of these algorithms are in some sense optimal, assuming ETH. The reduction does however not rule out an algorithm solving the problem in time $c^{k+\ell} n^{O(1)}$ for some constant c. Hence, as a first step, it would be interesting to see an algorithm with running time $f(\ell) \cdot 2^{O(k)} \cdot n^{O(1)}$ for some computable function f. And finally, can we prove that there is no $c^{k+\ell} \cdot n^{O(1)}$ time algorithm for COC?

The kernel size was recently improved independently by Xiao [Xia16] and Kumar & Lokshtanov [KL16] to $O(\ell k)$ vertices. And while the first runs in polynomial time, the later obtains a $2\ell k$ kernel, matching the best known kernels for Vertex Cover when $\ell = 1$. We note that this later kernelization algorithm runs in $n^{O(\ell)}$ time and hence in polynomial time for every fixed ℓ.

It is also interesting to study the problem parameterized by various structural parameters. Motivated by restricting epidemics, Enright and Meeks [EM15] gave an algorithm for solving COC running in time $O((t\ell))^{2t} n)$, where t is the treewidth of the input graph. Can we prove that COC is W[1]-hard when parameterized by k and t instead?

Finally, although Component Order Connectivity is not in FPT parameterized by either k or ℓ, there might still exist tractable parameterized approximation algorithms for the problem. We note that an $(\ell + 1)$-approximation for COC is obtained by applying the same type of argument as we did for the 2-approximation for Vertex Cover in Chapter 1. This can also be obtained via the above mentioned reduction to $(\ell + 1)$-Hitting Set [KL16]. We also note that an $O(\log \ell)$-fixed-parameter tractable approximation algorithm for COC when parameterized by ℓ was presented recently by Lee [Lee16].

Part III

Editing to nested neighborhoods

Chapter 7

Introduction

In this part we mainly study the computational complexity of two edge modification problems, namely editing to threshold graphs and editing to chain graphs. Graph modification problems ask whether a given graph G can be transformed to have a certain property using a small number of edits (such as deleting/adding vertices or edges), and have been the subject of significant previous work [SST04, CJL03, Dam06, DLL$^+$06, NG13].

In the Threshold Editing problem, we are given as input an n-vertex graph $G = (V, E)$ and a non-negative integer k. The objective is to find a set F of at most k pairs of vertices such that G minus any edges in F plus all non-edges in F is a threshold graph. A graph is a *threshold graph* if it can be constructed from the empty graph by repeatedly adding either an isolated vertex or a universal vertex [BLS99].

Threshold Editing

Input:	A graph $G = (V, E)$ and a non-negative integer k
Question:	Is there a set $F \subseteq [V]^2$ of size at most k such that $G \oplus F$ is a threshold graph.

The computational complexity of Threshold Editing has repeatedly been stated as open, starting from Natanzon et al. [NSS01], and then more recently by Burzyn et al. [BBD06], and again very recently by Liu, Wang, Guo and Chen [LWGC12]. Natanzon et al. [NSS01] showed that Threshold Editing can be solved in polynomial time on bounded degree input graphs. We resolve the general problem by showing that the problem is indeed NP-hard.

Graph editing problems are well-motivated by problems arising in the applied sciences, where we often have a predicted model from domain knowledge, but observed data fails to fit this model exactly. In this setting, edge modification corresponds to correcting false positives (and/or false negatives) to obtain data that is consistent with the model. Threshold Editing has specifically been of recent interest in the social sciences, where Brandes et al. are using distance to threshold graphs in work on axiomatization of centrality measures [Bra14, SB15]. More generally, editing to threshold graphs and their close relatives *chain graphs* arises in the study of sparse matrix multiplications [Yan81a]. Chain graphs are the bipartite analogue of threshold graphs (see Definition 7.7), and here we also establish hardness of Chain Editing.

A parameterized perspective

Having settled the NP-hardness of these problems, we turn to studying ways of dealing with their intractability. Cai's theorem [Cai96] shows that Threshold Editing and Chain Editing are *fixed-parameter tractable*, i.e., solvable in $f(k) \cdot n^{O(1)}$ time where k is the edit distance from the desired model (graph class); However, by the reductions used to prove NP-hardness we also obtain lower bounds in the order of $2^{o(\sqrt{k})} \cdot n^{O(1)}$ under ETH, and thus leave a gap. We continue to show that it is in fact the lower bound which is tight (up to logarithmic factors in the exponent) by giving a subexponential time algorithm for both problems.

A crucial first step in our algorithms is to preprocess the instance, reducing to a kernel of size polynomial in the parameter. We give quadratic kernels for all three edge variants of modifying your graph into either a threshold or chain graph. This answers (affirmatively) a recent question of Liu, Wang and Guo [LWG14]—whether the previously known kernel, which has $O(k^3)$ vertices, for Threshold Completion (equivalently Threshold Deletion) can be improved.

7.1 Graph classes with nested neighborhoods

A split graph is a graph $G = (V, E)$ whose vertex set can be partitioned into two sets C and I such that $G[C]$ is a complete graph and $G[I]$ is edgeless, i.e., C is a clique and I an independent set [BLS99]. For a split graph G we say that a partition (C, I) of $V(G)$ forms a *split partition* of G if $G[C]$ induces a clique and $G[I]$ an independent set. A split partition (C, I) is called a *complete* split partition if for every vertex $v \in I$, $N(v) = C$. If G admits a complete split partition, we say that G is a complete split graph.

The original definition of a *threshold graph* is the following: A graph is a threshold graph if it is possible to assign real weights to the vertices $w \colon V \to [0, 1]$ such that two vertices u and v are adjacent if and only if $w(u) + w(v) \geq 1$. We now give two characterizations of threshold graphs that will be more useful for us:

Proposition 7.1 ([MP95]). *A graph G is a threshold graph if and only if G has a split partition (C, I) such that the neighborhoods of the vertices in I are nested, i.e., for every pair of vertices v and u in I, either $N(v) \subseteq N[u]$ or $N(u) \subseteq N[v]$.*

Proposition 7.2 ([BLS99]). *A graph G is a threshold graph if and only if G does not have a C_4, P_4 nor a $2K_2$ as an induced subgraph. Thus, the threshold graphs are exactly the $\{C_4, P_4, 2K_2\}$-*free graphs *(see Figure 7.1).*

Motivated by the characterization of threshold graphs in Propositions 7.2 and 7.8, we define threshold obstructions (also see Figure 7.1).

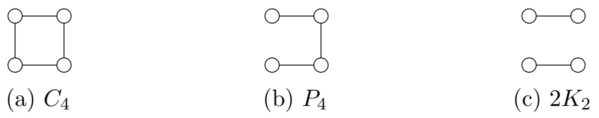


Figure 7.1: *Threshold graphs* are $\{C_4, P_4, 2K_2\}$-free.

Definition 7.3 ($\mathcal{H}$, Threshold obstruction)**.** A graph H is a *threshold obstruction* if it is isomorphic to a member of the set $\{C_4, P_4, 2K_2\}$ and a *chain obstruction* if it is isomorphic to a member of the set $\{C_3, 2K_2, C_5\}$. If it is clear from the context, we will often use the term obstruction for both threshold and chain obstructions and denote the set of obstructions by $\mathcal{H}$.

The following partitioning of threshold graphs will be instrumental in the development of the polynomial kernels.

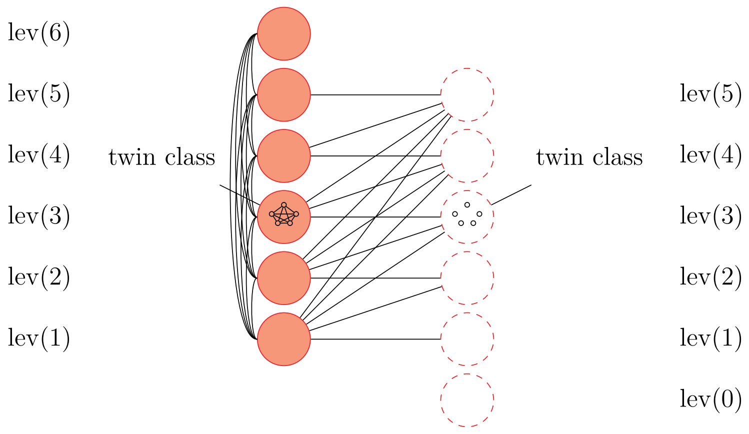


Figure 7.2: A threshold partition—the left hand side is the clique and the right hand side is an independent set, each bag contains a twin class. All bags are non-empty, otherwise two twin classes on the opposite side would collapse into one, except possibly the two extremal bags.

Definition 7.4. We say that $(\mathcal{C}, \mathcal{I}) = (\langle C_1, \ldots, C_t \rangle, \langle I_1, \ldots, I_t \rangle)$ forms a *threshold partition* of G if the following holds (see Figure 7.2 for an illustration):

- (C, I) is a split partition of G, where $C = \bigcup_{i \leq t} C_i$ and $I = \bigcup_{i \leq t} I_i$,
- C_i and I_i are twin classes in G for every i
- $N[C_j] \subset N[C_i]$ and $N(I_i) \subset N(I_j)$ for every $i < j$.
- Finally, we demand that for every $i \leq t$, $(C_i, I_{\geq i})$ form a complete split partition of the graph induced by $C_i \cup I_{\geq i}$.

We furthermore define, for every vertex v in G, $\mathrm{lev}(v)$ as the number i such that $v \in C_i \cup I_i$ and we denote each level $L_i = C_i \cup I_i$.

In a threshold decomposition we will refer to C_i for every i as a *clique fragment* and I_i as a *independent fragment.* Furthermore, we will refer to a vertex in $\cup\mathcal{C}$ as a *clique vertex* and a vertex in $\cup\mathcal{I}$ as an *independent vertex.*

Proposition 7.5 (Threshold decomposition)**.** *A graph G is a threshold graph if and only if G admits a threshold partition.*

Proof. Suppose that G is a threshold graph and therefore admits a nested ordering of the neighborhoods of vertices of each side [HIS81]. We show that partitioning the graph into partitions depending only on their degree yields the levels of a threshold partition. The clique side is naturally defined as the maximal set of highest degree vertices that form a clique. Suppose now for contradiction that this did not constitute a threshold partition. By definitions, every level consists of twin classes, and also, for two twin classes I_i and I_j, since their neighborhoods are nested in the threshold graph, their neighborhoods are nested in the threshold partition as well. So what is left to verify is that $(C_i, I_{\geq i})$ is a complete split partition of $G[C_i \cup I_{\geq i}]$. But that follows directly from the assumption that G admitted a nested ordering and C_i is a true twin class.

For the reverse direction, suppose G admits a threshold partition $(\mathcal{C}, \mathcal{I})$. Consider any four connected vertices a, b, c, d. We will show that they can not form any of the induced obstructions (see Figure 7.1). For the $2K_2$ and C_4, it is easy to see that at most two of the vertices can be in the clique part of the decomposition—and they must be adjacent since it is a clique—and hence there must be an edge in the independent set part of the decomposition, which contradicts the assumption that $\mathcal{C}, \mathcal{I}$ was a threshold partition. So suppose now that a, b, c, d forms a P_4. Again with the same reasoning as above, the middle edge b, c must be contained in the clique part, hence a and d must be in the independent set part. But since the neighborhoods of a and d should be nested, they cannot have a private neighbor each, hence either ac or bd must be an edge, which contradicts the assumption that a, b, c, d induced a P_4. This concludes the proof. □

Lemma 7.6. *For every instance (G, k) of* Threshold Editing *or* Threshold Completion *it holds that there exists an optimal solution F such that for every pair of vertices $u, v \in V(G)$, if $N_G(u) \subseteq N_G[v]$ then $N_{G\oplus F}(u) \subseteq N_{G\oplus F}[v]$.*

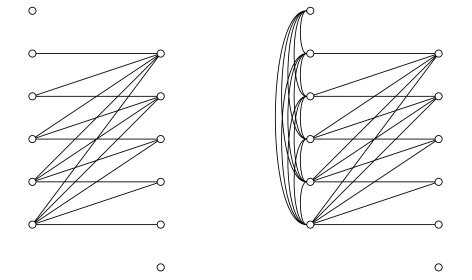
Proof. Let us define, for any editing set F and two vertices u and v, the set

$$F_{v\leftrightarrow u} = \{e \mid e' \in F \text{ and } e \text{ is } e' \text{ with } u \text{ and } v \text{ switched}\}.$$

Suppose F is an optimal solution for which the above statement does not hold. Then $N_G(u) \subseteq N_G[v]$ and $N_{G\oplus F}(v) \subseteq N_{G\oplus F}[u]$ (see Proposition 7.1). But then it is easy to see that we can flip edges in an ordering such that at some point, say after flipping F^0, u and v are twins in this intermediate graph $G \oplus F^0$. Let $F^1 = F \setminus F^0$. It is clear that for $G' = G \oplus (F^0 \cup F^1_{v\leftrightarrow u})$, $N_{G'}(u) \subseteq N_{G'}[v]$. Since $|F| \geq |F^0 \cup F^1_{v\leftrightarrow u}|$, the claim holds. □

Chain graphs

Chain graphs are the bipartite graphs whose neighborhoods of the vertices on one of the sides form an inclusion chain. It follows that the neighborhoods on the opposite side form an inclusion chain as well. If this is the case, we say that the neighborhoods are *nested*. The relation to threshold graphs is obvious, see Figure 7.3 for a comparison. The problem of completing edges to obtain a chain graph was introduced by Golumbic [Gol04] and later studied by Yannakakis [Yan81a], Feder, Mannila and Terzi [FMT09] and finally by Fomin and Villanger [FV13] who showed that CHAIN COMPLETION when given a bipartite graph whose bipartition must be respected is solvable in subexponential time.



(a) A chain graph (b) A threshold graph

Figure 7.3: Illustration of the similarities between chain and threshold graphs. Note that the nodes drawn can be replaced by twin classes of any size, even empty. However, if on one side of a level there is an empty class, the other two levels on the opposite side will collapse to a twin class. See Proposition 7.5.

Definition 7.7 (Chain graph)**.** A bipartite graph $G = (A, B, E)$ is a chain graph if there is an ordering of the vertices of A, $a_1, a_2, \ldots, a_{|A|}$ such that $N(a_1) \subseteq N(a_2) \subseteq \cdots \subseteq N(a_{|A|})$.

From the following proposition, it follows that chain graphs are characterized by a finite set of forbidden induced subgraphs and hence are subject to Cai's theorem [Cai96].

Proposition 7.8 ([BLS99])**.** *Let G be a graph. The following are equivalent:*

- *G is a chain graph.*
- *G is bipartite and $2K_2$-free.*
- *G is $\{2K_2, C_3, C_5\}$-free.*
- *G can be constructed from a threshold graph by removing all the edges in the clique partition.*

Since they have the same structure as threshold graphs, it is natural to talk about a *chain decomposition* $(\mathcal{A}, \mathcal{B})$ of a bipartite graph G with bipartition (A, B). We say that $(\mathcal{A}, \mathcal{B})$ is a chain decomposition for a chain graph G if and only if $(\mathcal{A}, \mathcal{B})$ is a threshold decomposition for the corresponding threshold graph G' where A is made into a clique.

Some additional tools

As part of our kernelization algorithms we obtain a set X and analyse the neighborhoods of vertices in $G - X$ inside X. To do this in a natural manner we need the following notion.

Definition 7.9 (Realizing)**.** For a graph G and a set of vertices $X \subseteq V(G)$ we say that a vertex $v \in V(G) \setminus X$ is *realizing* $Y \subseteq X$ if $N_X(v) = Y$. Furthermore, we say that a set $Y \subseteq X$ is *being realized* if there is a vertex $v \in V(G) \setminus X$ such that v is realizing Y.

In our kernel for Chain Editing, when analyzing the neighborhoods realized inside the aforementioned set X we obtain the following set system.

Definition 7.10 (Laminar set system)**.** A set system $\mathcal{F}$ over a finite ground set U is *laminar* if for every two sets $X_1 \in \mathcal{F}$ and $X_2 \in \mathcal{F}$, either $X_1 \subseteq X_2$, $X_2 \subseteq X_1$, or $X_1 \cap X_2 = \emptyset$.

One property that is crucial and we will use of laminar set systems is that their sizes are bounded linearly by the ground set, as the following lemma attests.

Lemma 7.11 (Folklore)**.** *Let $\mathcal{F}$ be a laminar set system over a finite ground set U. Then the cardinality of $\mathcal{F}$ is at most* $2|U|$.

Proof sketch. By associating the elements of U with the leaves of a rooted tree where each internal non-root node has degree at least three, a non-empty element of F corresponds exactly to a rooted induced subtree. Since this tree can have at most $2|U| - 1$ nodes, by adding the possibility of the empty set, we obtain the result. □

Chapter 8

NP-hardness and lower bounds

In this chapter we will prove that a number of editing problems are NP-hard. In addition, assuming ETH, we will prove that the same problems do not admit $2^{o(\sqrt{k})}n^{O(1)}$ time algorithms. The later is obtained as the number of allowed edits (k) is bounded by the product of the number of variables and the number of clauses in the 3-SAT-instance given to the reduction algorithm. And hence Theorem 6, stating that there is no $2^{o(n+m)}(n+m)^{O(1)}$ algorithm for 3-SAT assuming ETH, is applicable. In particular, we obtain the conclusions above for the following problems: Threshold Editing, Chain Editing and Chordal Editing.

8.1 Lower bounds for Threshold Editing

Recall that a boolean formula ϕ is in 3-CNF-SAT if it is in conjunctive normal form and each clause has at most three variables. Our hardness reduction is from the problem 3Sat, where we are given a 3-CNF-SAT formula ϕ and asked to decide whether ϕ admits a satisfying assignment. We will denote by $\mathcal{C}_\phi$ the set of clauses, and by $\mathcal{V}_\phi$ the set of variables in a given 3-CNF-SAT formula ϕ. An *assignment* for a formula ϕ is a function $\alpha\colon \mathcal{V}_\phi \to \{\texttt{true}, \texttt{false}\}$. Furthermore, we assume we have some natural lexicographical ordering $<_{\text{lex}}$ of the clauses $c_1, \ldots, c_{|\mathcal{C}_\phi|}$ and the same for the variables $v_1, \ldots, v_{|\mathcal{V}_\phi|}$, hence we may write, for some variables x and y, that $x <_{\text{lex}} y$. To immediately get an impression of the reduction we aim for, the construction is depicted in Figure 8.1.

Construction

Recall that we want to form a graph G_ϕ and pick an integer k_ϕ so that (G_ϕ, k_ϕ) is a yes-instance of Threshold Editing if and only if ϕ is satisfiable. We will design G_ϕ to be a split graph, so that the split partition is forced to be maintained in any threshold graph within distance k_ϕ of G_ϕ, where $k_\phi = |\mathcal{C}_\phi| \cdot (3|\mathcal{V}_\phi| - 1)$. Given ϕ, we first create a clique of size $6|\mathcal{V}_\phi|$; To each variable $x \in \mathcal{V}_\phi$, we associate six vertices of this clique, and order them in the following manner

$$v_a^x, v_b^x, v_\perp^x, v_\top^x, v_c^x, v_d^x.$$

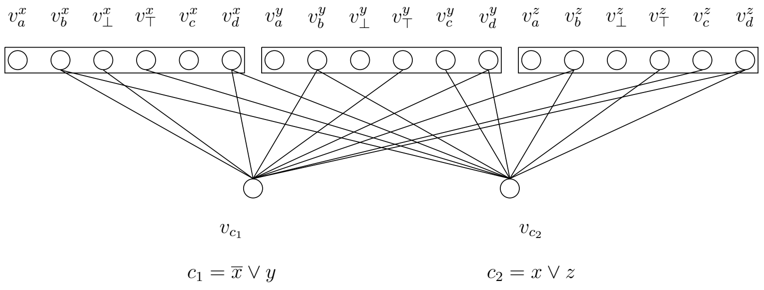

v_a^x v_b^x $v_\perp^x$ $v_\top^x$ v_c^x v_d^x v_a^y v_b^y $v_\perp^y$ $v_\top^y$ v_c^y v_d^y v_a^z v_b^z $v_\perp^z$ $v_\top^z$ v_c^z v_d^z
v_{c_1}
v_{c_2}
$c_1 = \overline{x} \vee y$
$c_2 = x \vee z$

Figure 8.1: The connections of a clause and a variable. All the vertices on the top (the variable vertices) belong to the clique, while the vertices on the bottom (the clause vertices) belong to the independent set. The vertices in the left part of the clique have higher degree than the vertices of the right part of the clique, whereas all the clause vertices (in the independent set) will all have the same degree, namely $3 \cdot |\mathcal{V}_\phi|$.

We will throughout the reduction refer to this ordering as π_ϕ: π_ϕ is a partial order which has

$$v_a^x <_{\pi_\phi} v_b^x <_{\pi_\phi} v_\top^x, v_\perp^x <_{\pi_\phi} v_c^x <_{\pi_\phi} v_d^x,$$

and for every two vertex $v_\star^x$ and $v_\star^y$ with $x <_{\text{lex}} y$, we have $v_\star^x <_{\pi_\phi} v_\star^y$. Observe that we do not specify which comes first of $v_\top^x$ and $v_\perp^x$—this is the choice that will result in the assignment α for ϕ.

We enforce this ordering by adding $O(k_\phi^2)$ vertices in the independent set; Enforcing that v_1 comes before v_2 in the ordering is done by adding $k_\phi + 1$ vertices in the independent set incident to all the vertices coming before v_1, including v_1. Since swapping the position of v_1 and v_2 would demand at least $k_\phi + 1$ edge modifications and k_ϕ is the intended budget, in any yes-instance, v_1 ends up before v_2 in the ordering of the clique.

We proceed adding the clause gadgets; For every clause $c \in \mathcal{C}_\phi$, we add one vertex v_c to the independent set. Hence, the size of the independent set is $O(|\mathcal{C}_\phi| + k_\phi^2)$. For a variable x occurring in c, we add an edge between v_c and $v_\perp^x$ if it occurs negatively, and between v_c and $v_\top^x$ otherwise. In addition, we make v_c incident to v_b^x and v_d^x.

For a variable z which does not occur in a clause c, we make v_c adjacent to v_b^z, v_c^z, and v_d^z. To complete the reduction, we add $4(k_\phi + 1)$ isolated vertices; $k_\phi + 1$ vertices to the left in the independent set, $k_\phi + 1$ vertices to the right in the independent set, and $k_\phi + 1$ to the left and $k_\phi + 1$ to the right in the clique. This ensures that no vertex will move from the clique to the independent set partition, and vice versa.

Properties of the Constructed Instance

Before proving the Theorem 20, and specifically Lemma 8.4, we may observe the following, which may serve as an intuition for the idea of the reduction. When we consider a fixed permutation of the variable gadget vertices (the clique side), the only thing we need to determine for a clause vertex v_c, is the *cut-off point*: the point in π_ϕ at which the vertex v_c will no longer have any neighbors. Observing that no vertex v_i^x swaps places with any other v_j^x for $i, j \in \{a, b, c, d\}$, and that no $v_\star^x$ changes with $v_\star^y$ for $x, y \in \mathcal{V}_\phi$, consider a fixed permutation of the variable vertices. We charge the clause vertices with the edits incident to the clause vertex. Since the budget is $k_\phi = |\mathcal{C}| \cdot (3|\mathcal{V}_\phi| - 1)$, and every clause needs at least $3|\mathcal{V}_\phi| - 1$, to obtain a solution (upcoming Lemma 8.2) we need to charge every clause vertex with exactly $3|\mathcal{V}_\phi| - 1$ edits. Figure 8.2 illustrates the charged cost of a clause vertex.

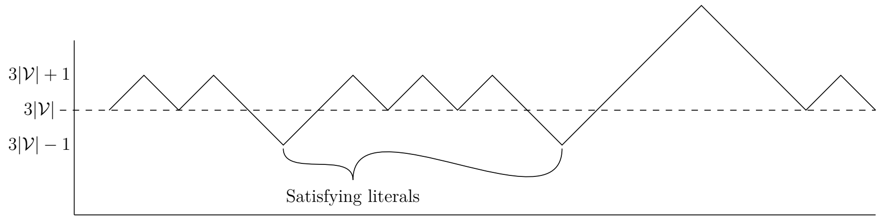


Figure 8.2: The cost with which we charge a clause vertex depends on the cut-off point; The x-axis denotes the point in the lexicographic ordering which separates the vertices adjacent to the clause vertex from the vertices not adjacent to the clause vertex.

Observation 8.1. *The graph G_ϕ resulting from the above procedure is a split graph and when $k_\phi = |\mathcal{C}| \cdot (3|\mathcal{V}_\phi| - 1)$, if H is a threshold graph within distance k_ϕ of G_ϕ, H must have the same clique-maximizing split partition as G_ϕ.*

Lemma 8.2. *Let (G_ϕ, k_ϕ) be a yes-instance to* Threshold Editing *constructed from a 3-CNF-SAT formula ϕ with $|F| \leq k_\phi$ a solution. For any clause vertex v_c, at least $3|\mathcal{V}_\phi| - 1$ edges in F are incident to v_c.*

Proof. By the properties of π_ϕ, we know that the only vertices we may change the order of are those corresponding to $v_\top^\star$ and $v_\perp^\star$. Pick any index in π_ϕ for which we know that v_c is adjacent to all vertices on the left hand side and non-adjacent to all vertices on the right hand side. Let L_c be the set of variables whose vertices are completely adjacent to v_c and R_c the corresponding set completely non-adjacent to v_c. By construction, v_c has exactly three neighbors in each variable and thus these variable gadgets contribute $3(|L_c| + |R_c|)$ to the budget. If $L_c \cup R_c = \mathcal{V}_\phi$, we are done, as v_c needs at least $3|\mathcal{V}_\phi|$ edits here.

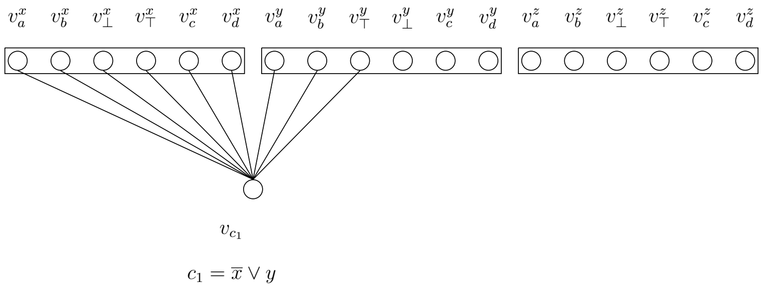

v_a^x v_b^x $v_\perp^x$ $v_\top^x$ v_c^x v_d^x v_a^y v_b^y $v_\top^y$ $v_\perp^y$ v_c^y v_d^y v_a^z v_b^z $v_\perp^z$ $v_\top^z$ v_c^z v_d^z
v_{c_1}
$c_1 = \overline{x} \vee y$

Figure 8.3: The edited version when y satisfies c_1. We have added three edges to the gadget x and deleted three edges to the gadget z, and added the edge to v_a^y and deleted the edge to v_d^y, that is, we have edited exactly $3 \cdot 2 + 2 = 3(|\mathcal{V}| - 1) + 2 = 3|\mathcal{V}| - 1$ edges incident to c_1. Notice that if $v_\perp^y$ was coming before $v_\top^y$, we would have to choose a different variable to satisfy c_1.

Suppose therefore that there is a variable x whose vertex v_a^x is adjacent to v_c and v_d^x is non-adjacent to v_c. But then we have already deleted the existing edge $v_c v_d^x$ and added the non-existing edge $v_c v_a^x$. This immediately gives a lower bound on $3(|\mathcal{V}_\phi| - 1) + 2 = 3|\mathcal{V}_\phi| - 1$ edits. □

Proof of Correctness

Lemma 8.3. *If there is an editing set F of size at most k_ϕ for an instance (G_ϕ, k_ϕ) constructed from a 3-CNF-SAT formula ϕ, and $|F(v_c)| = 3|\mathcal{V}_\phi| - 1$, then the $<_{\text{lex}}$-highest vertex connected to v_c corresponds to a variable satisfying the clause c.*

Proof. From the proof of Lemma 8.2, we observed that for a clause c to be within budget, we must choose a cut-off point within a variable gadget, meaning that there is a variable x for which v_c is adjacent to v_a^x and non-adjacent to v_d^x.

We now distinguish two cases, *(i)* x is a variable occurring (w.l.o.g. positively) in c and *(ii)* x does not occur in c. For *(i)*, v_c was adjacent to v_b^x, $v_\top^x$, and v_d^x. By assumption, we add the edge to v_a^x and delete the edge to v_d^x. But then we have already spent the entire budget, hence the only way this is a legal editing, $v_\top^x$ must come before $v_\perp^x$, and hence satisfies v_c. See Figure 8.3.

For *(ii)* we have that v_c was adjacent to v_b^x, v_c^x, and v_d^x. Here we, again by assumption, add the edge to v_a^x and delete the edge to v_d^x. This alone costs two edits, so we are done. But observe that these two edits alone are not enough, hence if we want to achieve the goal of $3|\mathcal{V}_\phi| - 1$ edited edges, the cut-off index must be inside a variable gadget corresponding to a variable occurring in c, i.e. *(i)* must be the case. □

Lemma 8.4. *A 3-CNF-SAT formula ϕ is satisfiable if and only if (G_ϕ, k_ϕ) is a yes-instance to* Threshold Editing.

Proof of Lemma 8.4. For the forwards direction, let ϕ be a satisfiable 3-CNF-SAT formula where $\alpha\colon \mathcal{V}_\phi \to \{\texttt{true}, \texttt{false}\}$ is any satisfying assignment, and (G_ϕ, k_ϕ) the Threshold Editing instance as described above.

Now, let $\alpha\colon \mathcal{V}_\phi \to \{\texttt{true}, \texttt{false}\}$ be a satisfying assignment, and (G_ϕ, k_ϕ) the Threshold Editing instance as described above, and let π be any permutation of the vertices of the clique side with the following properties

- for every $x <_{\text{lex}} y \in \mathcal{V}_\phi$, we have $v_\star^x <_\pi v_\star^y$,
- for every $x \in \mathcal{V}_\phi$, we have $v_a^x <_\pi v_b^x <_\pi v_\top^x < v_c^x <_\pi v_d^x$ and $v_a^x <_\pi v_b^x <_\pi v_\bot^x < v_c^x <_\pi v_d^x$, and finally
- for every $x \in \mathcal{V}_\phi$, we have $v_\bot^x <_\pi v_\top^x$ if and only if $\alpha(x) = \texttt{false}$.

We now show how to construct the threshold graph H_ϕ^π from the constructed graph G_ϕ by editing exactly $k_\phi = |\mathcal{C}| \cdot (3|\mathcal{V}_\phi| - 1)$ edges. For a clause c, let x be any variable satisfying c. If x appears positively, add every non-existing edge from v_c to every vertex $v \leq_\pi v_\top^x$ and delete all the rest. If x appears negated, use $v_\bot^x$ instead. We break the remainder of the proof in the forward direction into two claims:

Claim 8.5. *H_ϕ^π is a threshold graph.*

Proof of Claim 8.5. Let G_ϕ and π be given, both adhering to the above construction. Since G_ϕ was a split graph, π a total ordering of the elements in the independent set part and every vertex of the clique part of H_ϕ^π sees a prefix of the vertices of the independent set, their neighborhoods are naturally nested. Hence H_ϕ^π is a threshold graph by Proposition 7.1. □

Claim 8.6. $\left|E(G_\phi) \oplus E(H_\phi^\pi)\right| = k_\phi$.

Proof of Claim 8.6. Since we did not edit any of the edges within the clique part nor the independent set part, we only need to count the number of edits going between a clause vertex and the variable vertices. Let c be any clause and x the lexicographically smallest variable satisfying c. Suppose furthermore, without loss of generality, that x appears positively in c and has thus $\alpha(x) = \texttt{true}$. We now show that $|F(v_c)| = 3|\mathcal{V}_\phi| - 1$, and since c was arbitrary, this concludes the proof of the claim. Since v_c is adjacent to exactly three vertices per variable, and non-adjacent to exactly three vertices per variable, we added all the edges to the vertices appearing before x and removed all the edges to the vertices appearing after x. This cost exactly $3(|\mathcal{V}_\phi| - 1) = 3|\mathcal{V}_\phi| - 3$, hence we have two edges left in our budget for c. Moreover, the edge $v_c v_a^x$ was added and the edge $v_c v_d^x$ was deleted. Now, c is adjacent to every vertex to the before, and including, x, and non-adjacent to all the vertices after x. The budget used was $3(|\mathcal{V}_\phi| - 1) + 2 = 3|\mathcal{V}_\phi| - 1$. Hence, the total number of edges edited to obtain H_ϕ^π is $\sum_{c \in \mathcal{C}} 3|\mathcal{V}_\phi| - 1 = |\mathcal{C}| \cdot (3|\mathcal{V}_\phi| - 1) = k_\phi$. □

This shows that if ϕ is satisfiable, then (G_ϕ, k_ϕ) is a yes-instance of Threshold Editing.

In the reverse direction, let (G_ϕ, k_ϕ) be a constructed instance from a given 3-CNF-SAT formula ϕ and let F be a minimal editing set such that $G_\phi \oplus F$ is a threshold graph and $|F| \leq k_\phi$. We aim to construct a satisfying assignment $\alpha\colon \mathcal{V}_\phi \to \{\texttt{true}, \texttt{false}\}$ from $G_\phi \oplus F$. By Observation 8.1, $H = G_\phi \oplus F$ has the same split partition as G_ϕ. By construction, we have enforced the ordering, π_ϕ, of each of the vertices corresponding to the variables. Thus, we know exactly how H looks, with the exception of the internal ordering of each literal and its negation. Construct the assignment α as described above, i.e., $\alpha(x) = \texttt{false}$ if and only if $v^x_\bot <_\pi v^x_\top$.

By Lemmata 8.2 and 8.3, it follows directly that α is a satisfying assignment for ϕ which concludes the proof of the main lemma. □

The above lemma shows that there is a polynomial time many-one (Karp) reduction from 3Sat to Threshold Editing so we may wrap up the main theorem of this section. Lemma 8.4 implies Theorem 20, that Threshold Editing is NP-complete, even on split graphs.

Theorem 20. Threshold Editing *is* NP*-complete, even on split graphs.*

Proof. Split Threshold Editing is clearly in NP and that the problem is NP-complete follows immediately from combining Lemma 8.4 with Observation 8.1. □

For the sake of the next section, devoted to the proof of Theorem 22, we define the following annotated version of editing to threshold graphs. In this problem, we are given a split graph and we are asked to edit the graph to a threshold graph while respecting the split partition.

Split Threshold Editing

Input:	A split graph $G = (V, E)$ with split partition (C, I), and an integer k.
Question:	Is there an editing set $F \subseteq C \times I$ of size at most k such that $G \oplus F$ is a threshold graph?

Corollary 8.7. Split Threshold Editing *is* NP*-complete.*

Theorem 21. *Assuming ETH, neither* Threshold Editing *nor* Split Threshold Editing *are solvable in* $2^{o(\sqrt{k})} \cdot n^{O(1)}$ *time.*

Proof. Observe that the size of the editing set of the instance output by the reduction is the product of the number of variables and the number of clauses (up to constant factors) in the input 3-SAT-instance. Hence, we apply Theorem 6 to obtain the result. □

8.2 Lower bounds for Chain Editing

A bipartite graph $G = (A, B, E)$ is a *chain graph* if the neighborhoods of A are nested (which necessarily implies the neighborhoods of B are nested as well). Recalling Proposition 7.8, chain graphs are closely related to threshold graphs; Given a bipartite graph $G = (A, B, E)$, if one replaces A (or B) by a clique, the resulting graph is a threshold graph if and only if G was a chain graph.

It immediately follows from the above exposition that the following problem is NP-complete. This problem has also been referred to as CHAIN EDITING in the literature (for instance in the work by Guo [Guo07]).

BIPARTITE CHAIN EDITING

Input:	A bipartite graph $G = (A, B, E)$ and an integer k
Question:	Is there $F \subseteq A \times B$ with $\lvert F\rvert \leq k$ s.t. $G \oplus F$ is a chain graph?

Observe that we in this problem are given a bipartite graph together with a bipartition, and we are asked to respect the bipartition in the editing set.

Corollary 8.8. *The problem* BIPARTITE CHAIN EDITING *is* NP*-complete.*

Proof. We reduce from SPLIT THRESHOLD EDITING. Recall that to this problem, we are given a split graph $G = (V, E)$ with split partition (C, I), and an integer k, and asked whether there is an editing set $F \subseteq C \times I$ of size at most k such that $G \oplus F$ is a threshold graph. Since a chain graph is a threshold graph with the edges in the clique partition removed (Proposition 7.8), it follows that $G \oplus F$ with all the edges in the clique partition removed is a chain graph.

Let (G, k) be the input to SPLIT THRESHOLD EDITING and let (C, I) be the split partition. Remove all the edges in C to obtain a bipartite graph $G' = (A, B, E')$. Now it follows directly from Proposition 7.8 that (G, k) is a yes-instance to SPLIT THRESHOLD EDITING if and only if (G', k) is a yes-instance to BIPARTITE CHAIN EDITING. □

CHAIN EDITING

Input:	A graph $G = (V, E)$ and a non-negative integer k
Question:	Is there a set F of size at most k s.t. $G \oplus F$ is a chain graph?

Theorem 22. CHAIN EDITING *is* NP*-complete, even on bipartite graphs.*

Proof. Reduction from BIPARTITE CHAIN EDITING. Let $G = (A, B, E)$ be a bipartite graph and consider the input instance (G, k) to BIPARTITE CHAIN EDITING. We now show that adding $2(k + 1)$ new edges to G to obtain a graph $G' = (V, E')$, gives us that (G', k) is a yes-instance for CHAIN EDITING if and only if (G, k) is a yes-instance for BIPARTITE CHAIN EDITING.

Let $G = (A, B, E)$ be a bipartite graph and k a positive integer. Add $k + 1$ new vertices $a_1, \cdots a_{k+1}$ to A and make them universal to B, and add $k + 1$ new vertices $b_1, \cdots b_{k+1}$ to B and make them universal to A. Call the resulting graph $G' = (V, E')$. The following claim follows immediately from the construction.

Claim 8.9. *If $G' \oplus F$ is a chain graph with $|F| \leq k$, then $G' \oplus F$ has bipartition $(A \cup \{a_1, \ldots, a_{k+1}\}, B \cup \{b_1, \ldots, b_{k+1}\})$.*

It follows that for any input instance (G, k) to Bipartite Chain Editing, the instance (G', k) as constructed above is a yes-instance for Chain Editing if and only if (G, k) is a yes-instance for Bipartite Chain Editing. □

Theorem 23. *Assuming ETH, there is no algorithm solving neither* Chain Editing *nor* Bipartite Chain Editing *in time $2^{o(\sqrt{k})} \cdot n^{O(1)}$.*

Proof. In both these cases we reduced from Split Threshold Editing without changing the parameter k. Hence this follows immediately from the above exposition and from Theorem 21. □

8.3 Lower bounds for Chordal Editing

We will now combine our previous result on Chain Editing with the following observation of Yannakakis to provide lower bounds for Chordal Editing. Yannakakis showed [Yan81a], while proving the NP-completeness of Chordal Completion (more often known as Minimum Fill-In [FV13]), that a bipartite graph can be transformed into a chain graph by adding at most k edges if and only if the cobipartite graph formed by completing the two sides can be transformed into a chordal graph by adding at most k edges.

We will first give an intermediate problem that makes the proof simpler. Let $G = (A, B, E)$ be a cobipartite graph. Define the problem Cobipartite Chordal Editing to be the problem which on input (G, k) asks if we can edit at most k edges between A and B, i.e., does there exist an editing set $F \subseteq A \times B$ of size at most k, such that $G \oplus F$ is a chordal graph. That is, Cobipartite Chordal Editing asks for the bipartition A, B to be respected.

Cobipartite Chordal Editing

Input:	A cobipartite graph $G = (A, B, E)$ and an integer k
Question:	Does there exist a set $F \subseteq A \times B$ of size at most k such that $G \oplus F$ is a chordal graph?

We will use the following observation to prove the above theorem:

Lemma 8.10. *If $G = (A, B, E)$ is a bipartite graph, and $G' = (A, B, E')$ is the cobipartite graph constructed from G by completing A and B, then F is an optimal edge editing set for* Bipartite Chain Editing *on input (G, k) if and only if F is an optimal edge editing set for* Cobipartite Chordal Editing *on input (G', k).*

Proof. Let F be an optimal editing set for Bipartite Chain Editing on input (G, k) and suppose that $G' \oplus F$ has an induced cycle of length at least four. Since G' is cobipartite, it has a cycle of length exactly four. Let $a_1 b_1 b_2 a_2 a_1$ be this cycle.

But then it is clear that a_1b_1, a_2b_2 forms an induced $2K_2$ in $G \oplus F$, contradicting the assumption that F was an editing set.

For the reverse direction, suppose F is an optimal edge editing set for Cobipartite Chordal Editing on input (G', k) only editing edges between A and B. Suppose for the sake of a contradiction that $G \oplus F$ was not a chain graph. Since F only goes between A and B, $G \oplus F$ is bipartite and hence by the assumption must have an induced $2K_2$. This obstruction must be on the form a_1b_1, a_2b_2, but then $a_1b_1b_2a_2a_1$ is an induced C_4 in $G' \oplus F$ which is a contradiction to the assumption that $G' \oplus F$ was chordal. Hence $G \oplus F$ is a chain graph. □

Corollary 8.11. Cobipartite Chordal Editing *is* NP-*complete.*

Theorem 24. Chordal Editing *is* NP-*hard.*

Proof. Let $(G = (A, B, E), k)$ be a cobipartite graph as input to Cobipartite Chordal Editing. Our reduction is as follows. Create $G' = (A' \cup B', E')$ as follows:

- $A' = A \cup \{a_1, a_2, \ldots, a_{k+1}\}$,
- $B' = B \cup \{b_1, b_2, \ldots, b_{k+1}\}$,
- $E' = E \cup \bigcup_{i \leq k+1, b \in B'} \{a_i b\} \cup \bigcup_{i,j \leq k+1} \{a_i a_j, b_i b_j\}$

Finally, we create G'' as follows. For every edge a_ia_j create $k+1$ new vertices adjacent to only a_i and a_j. Do the same thing for every edge b_ib_j. This forces none of the edges in A' to be removed and none of the edges in B' to be removed.

Claim 8.12. *The instance of* Chordal Editing (G'', k) *is equivalent to the instance* (G, k) *to* Cobipartite Chordal Editing.

Proof of claim. The proof of the above claim is straight-forward. If we delete an edge within A (resp. B), we create at least $k + 1$ cycles of length 4, each of which uses at least one edge to delete, hence in any yes-instance, we do not edit edges within A (resp. B). Furthermore, any chordal graph remains chordal when adding a simplicial vertex, which is exactly what the $k + 1$ new vertices are. □

From the claim it follows that (G'', k) is a yes-instance to Chordal Editing if and only if (G, k) is a yes-instance to Cobipartite Chordal Editing. The theorem follows immediately from Corollary 8.11. □

Theorem 25. *Assuming ETH, there is no algorithm solving* Chordal Editing *in time* $2^{o(\sqrt{k})} \cdot n^{O(1)}$.

Chapter 9

Polynomial kernels

First we give kernels with quadratically many vertices for the following three problems: Threshold Completion, Threshold Deletion, and Threshold Editing, answering a recent question of Liu, Wang and Guo [LWG14]. Then we continue by providing kernels with quadratically many vertices for Chain Completion, Chain Deletion, and Chain Editing. Our kernelization algorithms use techniques similar to the previous result that Trivially Perfect Editing admits a polynomial kernel [DP15]. Observe that the class of threshold graphs is closed under taking complements. It follows that for every instance (G, k) of Threshold Completion, $(\bar{G}, k)$ is an equivalent instance of Threshold Deletion (and vice versa). Almost the same trick applies to Chain Deletion. Due to this, we restrict our attention to the completion and editing variants for the remainder of the section.

Before proceeding, we observe that our kernelization algorithms do not modify any edges, and only change the budget in the case that we discover that we have a no-instance (in which case we return $(H, 0)$, where H is an obstruction in G). The only modification of the instance is to delete vertices, hence the kernelized instance is an induced subgraph of the original graph. Since the parameter is never increased, we obtain *proper kernels*.

9.1 Modifications into Threshold Graphs

Outline of the Kernelization Algorithm

The kernelization algorithm consists of a twin reduction rule and an irrelevant vertex rule. The twin reduction rule is based on the observation that any obstruction containing vertices from a large enough twin class will have to be handled by edges not incident to the twin class. From this observation, we may conclude that for any twin class, we may keep only a certain amount without affecting the solutions.

A key concept of the irrelevant vertex rule is what will be referred to as a *threshold-modulator*. A threshold-modulator is a set of vertices X in G of linear size in k, such that for every obstruction H in G one can add and remove edges in $[X]^2$

to turn H into a non-obstruction. First, we prove that we can in polynomial time either obtain such a set X or conclude correctly that the instance is a no-instance. The observation that $G - X$ is a threshold graph will be exploited heavily and we now fix a threshold decomposition $(\mathcal{C}, \mathcal{I})$ of $G - X$. We then prove that the idea of Proposition 7.1 can be extended to vertices in $G - X$ when considering their neighborhoods in G. In other words, the neighborhoods of the vertices in $G - X$ are nested also when considering G. This immediately yields that the number of subsets of X that are being realized is bounded linearly in the size of X and hence also in k.

We now either conclude that the graph is small or we identify a sequence of levels in the threshold decomposition containing many vertices, such that all the clique vertices and all the independent set vertices in the sequence have identical neighborhoods in X, respectively. The crux is that in the middle of such a sequence there will be a vertex that is replaceable by other vertices in every obstruction and hence is irrelevant. Such a sequence is obtained by discarding all levels in the decomposition that are extremal with respect to a subset Y of X, meaning that there either are no levels above or underneath that contain vertices realizing Y. One can prove that in this process, only a quadratic number of vertices are discarded and from this we obtain a kernel.

The Twin Reduction Rule

First, we introduce the twin reduction rule as described above. For the remainder of the section we will assume this rule to be applied exhaustively and hence we can assume all twin classes to be small.

Rule 9.1 (Twin reduction rule)**.** *Let (G, k) be an instance of* Threshold Completion *or* Threshold Editing *and v a vertex in G such that $|\operatorname{tc}(v)| > 2k + 2$. We then reduce the instance to $(G - v, k)$.*

Lemma 9.1. *Let G be a graph and v a vertex in G such that $|\operatorname{tc}(v)| > 2k+2$. Then for every k we have that (G, k) is a yes-instance of* Threshold Completion *(or* Threshold Editing*) if and only if $(G - v, k)$ is a yes-instance of* Threshold Completion *(resp.* Threshold Editing*).*

Proof. For readability we only consider Threshold Completion, however the exact same proof works for Threshold Editing. Let $G' = G - v$. It trivially holds that if (G, k) is a yes-instance, then also (G', k) is a yes-instance. This is due to the fact that removing a vertex never will create new obstructions.

Now, let (G', k) be a yes-instance and assume for a contradiction that (G, k) is a no-instance. Let F be an optimal solution of (G', k) and W an obstruction in $(G \oplus F, k)$. Since W is not an obstruction in G' it follows immediately that v is in W. Furthermore, since $|F| \leq k$ it follows that there are two vertices $a, b \in \operatorname{tc}(v) \setminus \{v\}$ that F is not incident to. Also, one can observe that no obstruction contains more than two vertices from a twin class and hence we can assume without loss of generality that b is not in W. It follows that $N_{G \oplus F}(v) \cap (W - v) =$

$N_G(v) \cap (W - v) = N_G(b) \cap (W - v) = N_{G'}(b) \cap (W - v)$ and hence the graph induced on $V(W) \oplus \{b, v\}$ is an obstruction in $G' \oplus F$, contradicting that F is a solution. ☐

The Modulator

To obtain an $O(k^2)$ kernel we aim at an irrelevant vertex rule. However, this requires some tools. The first one is the concept of a threshold-modulator, as defined below.

Definition 9.2 (Threshold modulator)**.** Let G be a graph and $X \subseteq V(G)$ a set of vertices. We say that X is a *threshold-modulator* of G if for every obstruction W in G it holds that there is a set of edges F in $[X]^2$ such that $W \oplus F$ is not an obstruction.

Less formally, a set X is a threshold-modulator of a graph G if for every obstruction W in G you can edit edges between vertices in X to turn W into a non-obstruction. Our kernelization algorithm will heavily depend on finding a small threshold-modulator X and the fact that $G - X$ is a threshold graph.

Lemma 9.3. *There is a polynomial time algorithm that given a graph G and an integer k either*

- *outputs a threshold-modulator X of G such that $|X| \leq 4k$ or*
- *correctly concludes that (G, k) is a no-instance of both* Threshold Completion *and* Threshold Editing*.*

Proof. Let X_1 be the empty set and $\mathcal{W} = \{W_1, \ldots, W_t\}$ the set of all obstructions in G. We execute the following procedure for every W_i in $\mathcal{W}$: If $W_i \oplus F$ is an obstruction for every $F \subseteq [X_i \cap V(W_i)]^2$ we let $X_{i+1} = X_i \cup V(W_i)$, otherwise we let $X_{i+1} = X_i$. After we have considered all obstructions we let $X = X_{t+1}$. If $|X| > 4k$ we conclude that (G, k) is a no-instance, otherwise we output X.

Since all obstructions are finite the algorithm described clearly runs in polynomial time. We now argue that X is a threshold-modulator of G. If W_i was added to X_{i+1}, we let F be all the non-edges of W. Since $W \oplus F$ is isomorphic to K_4 it follows immediately that $W \oplus F$ is not an obstruction. If W_i was not added to X_{i+1}, let F the set found in $[X_i \cap V(W_i)]^2$ such that $W_i \oplus F$ is not an obstruction. Observe that $F \subseteq [X]^2$ and hence X is a threshold-modulator.

It remains to prove that if $|X| > 4k$ then (G, k) is a no-instance of Threshold Editing. Observe that it will follow immediately that (G, k) is a no-instance of Threshold Completion. Since every obstruction consists of four vertices there was at least $k + 1$ obstructions added during the procedure. Assume without loss of generality that $W_1, \ldots, W_{k+1}$ was added. Observe that by construction, a solution must contain an edge in $[X_{i+1} - X_i]^2$ for every $i \in [k + 1]$ and hence contains at least $k + 1$ edges. ☐

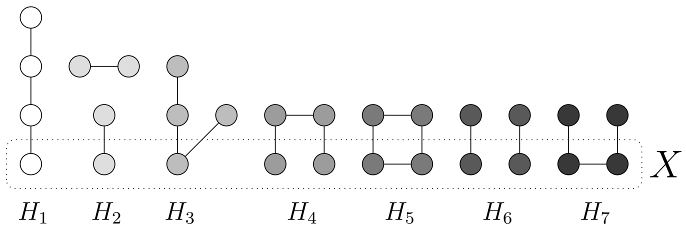


Figure 9.1: Some of the intersections of an obstruction with a threshold-modulator X that will not occur by definition. More specifically the ones necessary for the proof of the kernel.

Obtaining Structure

We now exploit the threshold-modulator and its interaction with the remaining graph to obtain structure. First, we prove that the neighborhoods of the vertices outside of X are nested and that the number of realized sets in X are bounded linearly in k.

Lemma 9.4. *Let G be a graph and X a threshold-modulator. For every pair of vertices u and v in $G - X$ it holds that either $N(u) \subset N[v]$ or $N(v) \subset N[u]$.*

Proof. Assume otherwise for a contradiction and let u' be a vertex in $N(u) \setminus N[v]$ and v' a vertex in $N(v) \setminus N[u]$. Let $W = G[\{u, v, u', v'\}]$ and observe that uu' and vv' are edges in W and uv' and vu' are non-edges in W by definition. Hence, no matter if some of the edges uv and $u'v'$ are present or not, W is an obstruction in G (see Figure 9.1 for an illustration). Since $u'v'$ is the only pair in W possibly with both elements in X this contradicts X being a threshold-modulator. □

Lemma 9.5. *Let G be a graph and X a corresponding threshold-modulator, then*

$$|\{N_X(v) \textit{ for } v \in V(G) \setminus X\}| \leq |X| + 1.$$

Or in other words, there are at most $|X| + 1$ sets of X that are being realized.

Proof. Let u and v be two vertices in $G - X$. It follows directly from Lemma 9.4 that either $N_X(v) \subseteq N_X(u)$ or $N_X(v) \supseteq N_X(u)$. The result follows immediately. □

With the definition of the modulator and the basic properties above, we are now ready to extract more vertices from the instance, aiming at many consecutive levels that have the same neighborhood in X for the clique, and independent set vertices, respectively. This will lead up to our irrelevant vertex rule.

Let G be a graph, X a threshold-modulator and $(\mathcal{C}, \mathcal{I})$ a threshold partition of $G - X$. Letting P denote either C or I, we say that a subset $Y \subseteq X$ has its *upper extreme* in P_i if P_i realizes Y and for every $j > i$ it holds that P_j does not realize Y. Similarly, a subset $Y \subseteq X$ has its *lower extreme* in P_i if P_i realizes Y and for every $j < i$ it holds that P_j does not realize Y. We say that $Y \subseteq X$ is *extremal* in P_i if Y has its upper or lower extreme in Y. Observe that every $Y \subseteq X$ is extremal in at most two clique fragments and two independent set fragments. We continue having P denote either C or I.

Lemma 9.6. *Let G be a graph, X a threshold-modulator and $(\mathcal{C},\mathcal{I})$ a threshold partition of $G - X$. For every $Y \subseteq X$ it holds that if Y has its lower extreme in P_ℓ and upper extreme in P_u, then for every vertex $v \in P_i$ with $i \in [\ell+1, u-1]$ it holds that $N_X(v) = Y$.*

Proof. Let Y be a subset of X with C_ℓ and C_u being its lower and upper extremes in the clique respectively. By definition there is a vertex $u \in C_\ell$ and a vertex $w \in C_u$ such that $N_X(u) = N_X(w) = Y$. Let i be an integer in $[\ell+1, u-1]$ and a vertex $v \in C_i$. By the definition of a threshold partition it holds that $N_{G-X}(w) \subset N_{G-X}(v) \subset N_{G-X}(u)$. It follows from Lemma 9.4 that $N(w) \subset N[v]$ and that $N(v) \subset N[u]$. Hence,

$$Y = N_X(w) \subseteq N_X(v) \subseteq N_X(u) = Y$$

and we conclude that $N_X(v) = Y$. Since i and v was arbitrary, the proof is complete. □

Definition 9.7 (Important, Outlying, and Regular)**.** We say that P_i in the partition is *important* if there is a $Y \subseteq X$ such that Y has its extreme in P_i. Furthermore, a level L_i is important if C_i or I_i is important. Let f be the smallest number such that $| \cup_{i \leq f} C_i| \geq 2k+2$ and r the largest number such that $| \cup_{i \geq r} I_i| \geq 2k+2$. A level L_i is *outlying* if $i \leq f$ or $i \geq r$. All other levels of the decomposition are *regular* and a vertex is regular, outlying or important depending on the type of the level it is contained in.

Lemma 9.8. *Let G be a graph and X a threshold-modulator of G of size at most $4k$. Then every threshold partition of $G - X$ has at most $16k + 4$ important levels.*

Proof. The result follows immediately from the definition of important levels and Lemma 9.5. □

Lemma 9.9. *Let G be a graph, X a threshold-modulator of G and $(\mathcal{C},\mathcal{I})$ a threshold partition of $G - X$, then for every set $Y \subseteq X$ there are at most two important clique fragments (independent fragments) realizing Y.*

Proof. We first prove the statement for clique fragments. Let Y be a subset of X and $i < j < k$ three integers. Assume for a contradiction that C_i, C_j and C_k are important clique fragments all realizing Y. By definition there are vertices $u \in C_i$, $v \in C_j$ and $w \in C_k$ such that $N_X(u) = N_X(v) = N_X(w) = Y$. Furthermore, there is a vertex $v' \in C_j$ such that $N_X(v') \neq Y$ since C_j is important and Y does not have an extreme in C_j. By the definition of threshold partitions, we have that $N_{G-X}(w) \subset N_{G-X}(v') \subset N_{G-X}(u)$. Lemma 9.4 immediately implies that $N(w) \subset N[v']$ and $N(v') \subset N[u]$ and since $\{u, v', w\} \subseteq \cup\mathcal{C}$ it holds that

$N[u] \subseteq N[v'] \subseteq N[w]$. Since $N_X(v') \neq Y$, we have $N_X(w) \subset N_X(v') \subset N_X(u)$, which contradicts the definition of w and u since $N_X(u) = N_X(w)$. By a symmetric argument, the statement also holds for independent fragments. □

Lemma 9.10. *Let G be a graph, X a threshold-modulator of G of size at most $4k$ and $(\mathcal{C}, \mathcal{I})$ a threshold partition of $G - X$. Then there are at most $64k^2 + 80k + 16$ important vertices in $G - X$.*

Proof. Let Y be the set of all vertices contained in a important clique or independent fragment and let Z be the set of all important vertices. Observe that $Y \subseteq Z$ and that every C_i or I_i contained in $Z \setminus Y$ is a twin class in G by definition. By Lemma 9.8 there are at most $16k + 4$ important levels and since the twin-rule has been applied exhaustively it holds that $|Z \setminus Y| \leq (16k+4)(2k+2) = 32k^2+40k+8$.

Let A be a subset of X and B the vertices in Y such that their neighborhood in X is exactly A. Let D be a C_i or I_i contained in Y and observe that $D \cap B$ is a twin class in G and hence $|D \cap B| \leq 2k + 2$. And hence it follows from Lemma 9.9 that $|B| \leq 8k + 8$. Furthermore, we know from Lemma 9.5 that there are at most $4k + 1$ realized in X and hence $|Y| \leq (8k + 8)(4k + 1) = 32k^2 + 40k + 8$. It follows immediately that $|Z| \leq 64k^2 + 80k + 16$, completing the proof. □

Lemma 9.11. *Let G be a graph, X a threshold-modulator of G of size at most $4k$ and $(\mathcal{C}, \mathcal{I})$ a threshold partition of $G - X$. Then there are at most $80k^2 + 112k + 32$ important and outlying vertices in total in $G - X$.*

Proof. By Lemma 9.10 it follows that there are at most $64k^2 + 80k + 16$ vertices that are important and possibly outlying. It follows from Lemma 9.6 that if a level is not important its vertices are covered by at most two twin classes in G and hence the level contains at most $4k + 4$ vertices. By definition there are at most $4k+4$ outlying levels and hence at most $(4k+4)(4k+4) = 16k^2+32k+16$ vertices which are outlying, but not important. The result follows immediately. □

Lemma 9.12. *Let G be a graph, X a threshold-modulator of G, v a regular vertex in some threshold partition $(\mathcal{C}, \mathcal{I})$ of $G - X$, $C = \cup\mathcal{C}$ and $I = \cup\mathcal{I}$. Then for every $F \subseteq [V(G)]^2$ such that $G \oplus F$ is a threshold graph, $|F| \leq k$ and every split partition (C_F, I_F) of $G \oplus F$ we have:*

- *$v \in C$ if and only if $v \in C_F$ and*
- *$v \in I$ if and only if $v \in I_F$.*

Proof. Observe that the two statements are equivalent and that it is sufficient to prove the forward direction of both statements. First, we prove that $v \in C$ implies that $v \in C_F$. Let Y be the set of outlying vertices in $I \cap N_G(v)$ and recall that $|Y| > 2k+1$ by definition. Observe that at most $2k$ vertices in Y are incident to F and hence there are two vertices u, u' in Y that are untouched by F. Clearly, u and u' are not adjacent in $G \oplus F$ and hence we can assume without loss of generality that u is in I_F. Since u is untouched by F, v is adjacent to u by the definition of outlying vertices and hence v is not in I_F. A symmetric argument gives that $v \in I$ implies that $v \in I_F$ and hence our argument is complete. □

The Irrelevant Vertex Rule

We have now obtained the structure necessary to give our irrelevant vertex rule. But before stating the rule, we need to define these consecutive levels with similar neighborhood and what it means for a vertex to be in the middle of such a collection of levels.

Definition 9.13 (Large strips, central vertices)**.** Let G be a graph, X a threshold-modulator and $(\mathcal{C}, \mathcal{I})$ a threshold partition of $G - X$. A *strip* is a maximal set of consecutive levels which are all regular and we say that a strip is *large* if it contains at least $16k + 13$ vertices. For a strip $S = ([C_a, I_a], \ldots, [C_b, I_b])$ a vertex $v \in C_i$ is *central* if $a \leq i \leq b$ and $|\cup_{j \in [a,i-1]} C_j| \geq 2k+2$ and $|\cup_{j \in [i+1,b]} C_j| \geq 2k+2$. Similarly we say that a vertex $v \in I_i$ is *central* if $a \leq i \leq b$ and $| \cup_{j \in [a,i-1]} I_j| \geq 2k + 2$ and $| \cup_{j \in [i+1,b]} I_j| \geq 2k + 2$. Furthermore, we say that a vertex v is *central in G* if there exists a threshold-modulator X of size at most $4k$ and a threshold decomposition of $G - X$ such that v is central in a large strip.

Lemma 9.14. *If a strip is large it has a central vertex.*

Proof. Let $S = ([C_a, I_a], \ldots, [C_b, I_b])$ be a large strip. First, we consider the case when $| \cup_{i \in [a,b]} C_i| \geq | \cup_{i \in [a,b]} I_i|$. Observe that $| \cup_{i \in [a,b]} C_i| \geq 8k + 7$. Let i be the smallest number such that $| \cup_{j \in [a,i-1]} C_j| \geq 2k + 2$. It follows immediately from $|C_{i-1}| \leq 2k + 2$ that $| \cup_{j \in [a,i-1]} C_j| \leq 4k + 3$. Furthermore, since $|C_i| \leq 2k + 2$ it follows that $| \cup_{j \in [i+1,b]} C_j| \geq 8k + 7 - (2k + 2 + 4k + 3) = 2k + 2$. And hence any vertex in C_i is central. A symmetric argument for the case $| \cup_{i \in [a,b]} C_i| < | \cup_{i \in [a,b]} I_i|$ completes the proof. $\square$

Rule 9.2 (Irrelevant vertex rule)**.** *If (G, k) be an instance of* Threshold Completion *or* Threshold Editing *and v is a central vertex in G, reduce to $(G - v, k)$.*

Lemma 9.15. *Let (G, k) be an instance, X a threshold-modulator and v a central vertex in G. Then (G, k) is a yes-instance of* Threshold Editing *(*Threshold Completion*) if and only if $(G - v, k)$ is a yes-instance.*

Proof. For readability we only consider Threshold Editing, however the exact same proof works for Threshold Completion. For the forwards direction, for any vertex v, if (G, k) is a yes-instance, then $(G - v, k)$ is also a yes-instance. This holds since threshold graphs are hereditary.

For the reverse direction, let $(G - v, k)$ be a yes-instance and assume for a contradiction that (G, k) is a no-instance. Let F be a solution of $(G - v, k)$ satisfying Lemma 7.6, and let $G' = G \oplus F$. By assumption, (G, k) is a no-instance, so specifically, G' is not a threshold graph. Let W be an obstruction in G'. Clearly $v \in W$ since otherwise there is an obstruction in $(G - v) \oplus F$, so consider $Z = V(W) - v$. For convenience we will use N' to denote neighborhoods in G' and specifically for any set $Y \subseteq V(G')$, $N'_Y(v) = N_{G'}(v) \cap Y$. Furthermore, let $(\mathcal{C}, \mathcal{I})$ be a threshold decomposition of $G - X$ such that there is a large strip S for which v is central. We will now consider the case when v is in the clique of

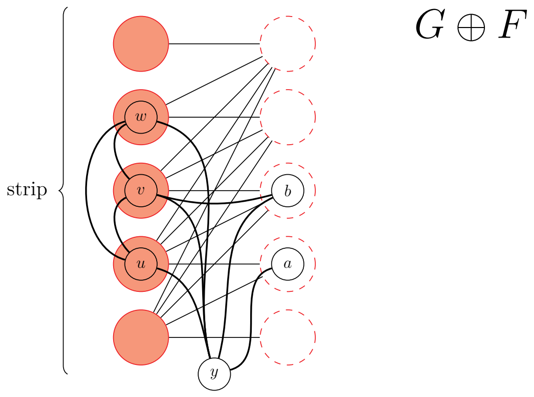


Figure 9.2: The vertex v was a center vertex in a strip and $W = \{v, a, b, y\}$ was assumed to be an obstruction.

$G - X$. Since $|F| \leq k$ and S is a large strip it follows immediately that there are two clique vertices w and w' in S in higher levels than v that is not incident to F. Observe that $\{w, w', v\}$ forms a triangle and that W contains no such subgraph. Hence, we can assume without loss of generality that $w \notin V(W)$. Similarly, we obtain a clique vertex u in a lower level than v in S such that $u \notin W$.

Observe that $G'[Z \cup \{u\}]$ is not an obstruction and hence $N_Z(u) = N'_Z(u) \neq N'_Z(v) = N_Z(v)$. Since u and v are clique vertices from the same strip it is true that $N_X(v) = N_X(u)$ and hence there is an independent vertex a in Z such that $\text{lev}(u) \leq \text{lev}(a) < \text{lev}(v)$ (see Definition 7.4). In other words u is adjacent to a while v and w are not. By a symmetric argument we obtain a vertex b such that $\text{lev}(v) \leq \text{lev}(b) < \text{lev}(w)$, meaning that both u and v are adjacent to b while w is not. Let y be last vertex of Z, meaning that $\{v, y, a, b\} = V(W)$. Observe that a and b are regular vertices and hence it follows from Lemma 9.12 that for every threshold partition of G' it holds that $\{a, b\}$ are independent vertices.

Recall that u, v, w, a, b are all regular and hence they are in the same partitions in G' as in $G - X$ by Lemma 9.12. Furthermore, since W is an obstruction and a is neither adjacent to v nor b in G' it holds that y and a are adjacent in G'. It follows that y is a clique vertex in G' and hence it is adjacent to both u and w in G'. Since u and w are not incident to F by definition, they are adjacent to y also in G. Since u, v, w are regular and from the same strip it follows that v is adjacent to y in both G and G'. Observe that the only possible adjacency not yet decided in W is the one between b and y. However, for W to be an obstruction it should not be present. Hence y is adjacent to a but not to b in G'. By definition $N_G(a) \subseteq N_G(b)$, however by the last observation this is not true in G'. This contradicts that F satisfies Lemma 7.6. A symmetric argument gives a contradiction for the case when v is an independent vertex and hence the proof is complete. $\square$

The above lemma shows the soundness of the irrelevant vertex rule, Rule 9.2, and we may therefor apply it exhaustively. The following theorem wraps up the goal of this section.

Theorem 26. *The following three problems admit kernels with at most* $336k^2 + 388k + 92$ *vertices:* Threshold Deletion, Threshold Completion *and* Threshold Editing.

Proof. Assume that Rules 9.1 and 9.2 have been applied exhaustively. If this process does not produce a threshold-modulator, we can safely output a trivial no-instance by Lemma 9.3. Hence, we can assume that we have a threshold-modulator X of size at most $4k$ and that the reduction rules cannot be applied. By Lemma 9.11 we know that there are at most $80k^2+112k+32$ vertices in $G-X$ that are not regular. Furthermore, every regular vertex is contained in a strip and by Lemma 9.8 there are at most $16k+5$ such strips. Since the reduction rules cannot be applied, no strip is large, and hence they contain at most $16k+12$ vertices each. Since every vertex in G is either in X, or considered regular, outlying or important this gives us $4k + 80k^2 + 112k + 32 + (16k + 5)(16k + 12) = 336k^2 + 388k + 92$ vertices in total. □

9.2 Modifications into Chain Graphs

In this section we provide kernels with quadratically many vertices for Chain Deletion, Chain Completion and Chain Editing. Due to the fundamental similarities between modification to chain and threshold graphs we omit the full proof and instead highlight the differences between the two proofs. Observe that the only proofs for the threshold kernels that explicitly applies the obstructions are those of Lemmata 9.3, 9.4 and 9.15 and hence these will receive most of our attention.

The twin reduction rule goes through immediately and hence our first obstacle is the modulator. Luckily, this is a minor one. Recall from Definition 7.3 that the obstructions now are $\mathcal{H} = \{2K_2, C_3, C_5\}$; We thus get a chain-modulator X of size $5k$, as the largest obstruction contains five vertices. Besides this detail, the proof goes through exactly as it is.

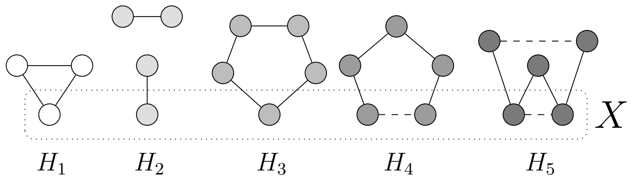


Figure 9.3: Some of the intersections of an obstruction with a chain-modulator X that by definition will not occur. Dashed edges represent edges that could or could not be there. These are the intersections necessary for the proof of the kernel.

An Additional Step

Before we continue with the remainder of the proof we need an additional step. Namely to discard all vertices that are isolated in $G - X$. We will prove that by doing this we discard at most $O(k^2)$ vertices. Now, if the irrelevant vertex rule concludes that the graph is small, then the graph is small also when we reintroduce the discarded vertices. And if we find an irrelevant vertex, we remove it and reintroduce the discarded vertices before we once again apply our reduction rules. Due to the locality of our arguments, this is a valid approach.

Lemma 9.16. *For a graph G and a corresponding chain-modulator X there are at most $10k^2 + 12k + 2$ isolated vertices in $G - X$.*

Proof. Let I be the set of isolated vertices in $G - X$. We will prove that $\mathcal{F} = \{N_X(v) \mid v \in I\}$ is laminar (see Definition 7.10) and hence by Lemma 7.11 it holds that $|\mathcal{F}| \leq |X| + 1 \leq 5k + 1$. It follows immediately, due to the twin reduction rule, that there are at most $(5k + 1)(2k + 2) = 10k^2 + 12k + 2$ independent vertices in $G - X$.

Assume for a contradiction that there are vertices u, v and w in I such that there exists $u' \in N_X(u) \setminus N_X(v)$ and $v' \in N_X(v) \setminus N_X(u)$ with $\{u', v'\} \subseteq N_X(w)$. These vertices intersect with the modulator as a variant of the forbidden H_5 in Figure 9.3 and hence we get a contradiction. □

Nested Neighborhoods

From now on we will assume in all of our arguments that there are no isolated vertices in $G - X$. The next difference is with respect to Lemma 9.4, which is just not true anymore. The lemma provided us with the nested structure of the neighborhoods in the modulator and was crucial for most of the proofs. As harmful as this appears to be at first, it turns out that we can prove a weaker version that is sufficient for our needs.

Lemma 9.17 (New, weaker version of Lemma 9.4)**.** *Let G be a graph and X a chain-modulator. For every pair of vertices u and v in the* same bipartition *of $G - X$ it holds that either $N(u) \subseteq N(v)$ or $N(v) \subseteq N(u)$.*

Proof. Let u and v be two vertices from the same bipartition of $G - X$. By the definition of chain graphs we can assume that $N_{G-X}(u) \subseteq N_{G-X}(v)$. Assume for a contradiction that the lemma is not true. Then there is a vertex $u' \in N_X(u) \setminus N_X(v)$ and a vertex v' in $N_X(v) \setminus N_X(u)$. By definition, u and v are not adjacent. Since there are no isolated vertices in $G - X$ there is a vertex $a \in N_{G-X}(u) \subseteq N_{G-X}(v)$. Observe that if a is adjacent to either u' or v' we get a C_3 that only has one vertex in X, which is a contradiction (see H_1 in Figure 9.3). However, if a is not adjacent to both u' and v' then $\{u, v, u', v', a\}$ forms the same interaction with the modulator as H_4 in Figure 9.3 and hence our proof is complete. □

One can observe that Lemma 9.17 is a sufficiently strong replacement for Lemma 9.4 since all proofs are applying the lemma to vertices from only one partition of

$G - X$. The only exception is the proof of Lemma 9.5, but by applying Lemma 9.17 on one partition at the time we obtain the following bound instead:

$$|\{N_X(v) \text{ for } v \in V(G) \setminus X\}| \leq 2|X| + 2.$$

An Irrelevant Vertex Rule

It only remains to prove that the irrelevant vertex rule can still be applied with this new set of obstructions. Although the strategy is the same, the details are different and hence we provide the proof in full detail.

Lemma 9.18. *Let (G, k) be an instance, X a threshold-modulator and v a central vertex in G. Then (G, k) is a yes-instance of* Chain Editing *(*Chain Completion*) if and only if $(G - v, k)$ is a yes-instance.*

Proof. For readability we only consider Chain Editing, however the exact same proof works for Chain Completion. For the forwards direction, for any vertex v, if (G, k) is a yes-instance, then $(G - v, k)$ is also a yes-instance. This holds since chain graphs are hereditary.

For the reverse direction, let $(G - v, k)$ be a yes-instance and assume for a contradiction that (G, k) is a no-instance. Let F be a solution of $(G - v, k)$ satisfying Lemma 7.6, and let $G' = G \oplus F$. By assumption, (G, k) is a no-instance, so specifically, G' is not a chain graph. Let W be an obstruction in G'. Clearly $v \in W$, since otherwise there is an obstruction in $(G - v) \oplus F$. Let $Z = V(W) - v$. For convenience we will use N' to denote neighborhoods in G' and specifically for any set $Y \subseteq V(G')$, $N'_Y(v) = N_{G'}(v) \cap Y$. Furthermore, let $(\mathcal{A}, \mathcal{B})$ be a chain decomposition of $G - X$ such that there is a large strip S for which v is central. Let $A = \cup\mathcal{A}$ and $B = \cup\mathcal{B}$. We will now consider the case when v is in A. Since $|F| \leq k$ and S is a large strip it follows immediately that there are two vertices w and w' in $A \cap S$ in higher levels than v that is not incident to F. Observe that $\{w, w', v\}$ forms an independent set of size three and that W contains no such subgraph. Hence, we can assume without loss of generality that $w \notin V(W)$. Similarly, we obtain a vertex u in A at a lower level than v in S such that $u \notin W$.

Observe that $G'[Z \cup \{u\}]$ is not an obstruction and hence $N_Z(u) = N'_Z(u) \neq N'_Z(v) = N_Z(v)$. Since u and v are vertices in A from the same strip it is true that $N_X(v) = N_X(u)$ and hence there is a vertex a in $Z \cap B$ such that $\text{lev}(u) \leq \text{lev}(a) < \text{lev}(v)$. In other words u is adjacent to a, while v and w are not. By a symmetric argument we obtain a vertex b such that $\text{lev}(v) \leq \text{lev}(b) < \text{lev}(w)$, meaning that both u and v are adjacent to b while w is not. We now fix a chain decomposition $(\mathcal{A}', \mathcal{B}')$ and let $A' = \cup\mathcal{A}'$ and $B' = \cup\mathcal{B}'$. Observe that a and b are regular vertices and hence it follows from the chain version of Lemma 9.12 that $\{a, b\}$ is in B'. This yields immediately that W is not a C_3 (since a and b are not adjacent) and hence we are left the cases of W being a $2K_2$ or a C_5.

We now consider the case when W is isomorphic to a $2K_2$. Let y be the last vertex of Z, meaning that $\{v, y, a, b\} = V(W)$. Observe that since W is a

$2K_2$ it holds that y is adjacent to a, but not to b. However, in G it holds that $N(a) \subseteq N(b)$ and hence F is not satisfying Lemma 7.6, which is a contradiction.

Hence we are left with the case that W is isomorphic to a C_5. Let y, x be the last vertices of Z. Observe that all vertices in W should be of degree two and hence a is adjacent to both x and y. Recall that a is in B' and observe that u is in A' by the same reasoning. Due to their adjacency to a, also x and y is in A'. It follows immediately that u, x and y form an independent set in $(G - v) \oplus F$. Since u and v are not touched by F and in the same strip it follows that v, x and y form an independent set in G'. We observe that by this W can not be isomorphic to a C_5. The argument for the case when $v \in B$ is symmetrical and hence the proof is complete. $\square$

We immediately obtain our kernelization results for modifications into chain graphs by the same wrap up as for threshold graphs.

Theorem 27. *The following three problems admit kernels with at most $O(k^2)$ vertices:* Chain Deletion, Chain Completion *and* Chain Editing.

Chapter 10

Subexponential time algorithms

In this chapter we give a subexponential time algorithm for Threshold Editing. We also show that we can modify the algorithm to work with Chain Editing. Combined with the results of Fomin and Villanger [FV13] and Drange et al. [DFPV15], we now have complete information on the subexponentiality of edge modification to threshold and chain graphs.

10.1 Editing to Threshold Graphs

We will throughout refer to a *solution* F. In this case, we are assuming a given input instance (G, k), and then F is a set of at most k edges such that $G \oplus F$ is a threshold graph.

A brief explanation of the algorithm The algorithm consists of four parts, the first of which is the kernelization algorithm described in Chapter 9. This gives in polynomial time an equivalent instance (G, k) with the guarantee that $|V(G)| = O(k^2)$. We may observe that this is a *proper kernel*, i.e., the reduced instance's parameter is bounded by the original parameter. This allows us to use time subexponential in the kernelized parameter.

The second step in the algorithm selects a potential split partitioning of G. We show that the number of such partitionings is bounded subexponentially in k, and that we can enumerate them all in subexponential time. This step actually also immediately implies that editing[1], completing and deleting to split graphs can be solved in subexponential time, however all of this was known [HS81, GKK$^+$15]. The main part of this step is Lemma 1.13. For the remainder of the algorithm, we may thus assume that the input instance is a split graph, and that the split partition needs to be preserved, that is, we focus on solving Split Threshold Editing.

The third and fourth steps of the algorithm consists of repeatedly finding special kind of separators and solving structured parts individually; Step three consists of locating so-called *cheap vertices* (see Definition 10.2 for a formal explanation).

[1]Indeed, editing to split graphs is solvable in linear time [HS81].

These are vertices, v, whose neighborhood is almost correct, in the sense that there is an optimal solution in which v is incident to only $O(\sqrt{k})$ edges. The dichotomy of cheap and expensive vertices gives us some tools for decomposing the graph. Specific configurations of cheap vertices allow us to extract three parts, one part is a highly structured part, the second part is a provably small part which me may brute force, and the last part we solve recursively. All of which is done in subexponential time $2^{O(\sqrt{k} \log k)}$.

Henceforth we will have in mind a "target graph" $H = G \oplus F$ with threshold partitioning $(\mathcal{C}, \mathcal{I})$. We refer to the set of edges F as the *solution*, and assume $|F| \leq k$. A crucial part of the algorithm is to enumerate all vertex sets of size at most $O(\sqrt{k})$. Observation 1.12 shows that this is indeed doable and we will use this result throughout this section without necessarily referring to it.

Enumerating the potential partitions

After kernelizing the instance the next step of the subexponential time algorithm is to compute the potential split partitionings of the input instance. Since we are given a general graph, we do not know which vertices will go to the clique and which will go to the independent set. However, we now show that there is at most subexponentially many potential split partitionings. That is, there are subexponentially many partitionings of the vertex set into (C, I) such that it is possible to edit the input graph to a threshold graph with the given partitioning while not exceeding the prescribed budget.

Definition 10.1 (Potential split partition)**.** Given a graph G and an integer k (called the budget), for C and I a partitioning of $V(G)$ we call (C, I) a *potential split partition* of G provided that

$$\binom{|C|}{2} - E(C) + E(I) \leq k.$$

That is, the cost of making G into a split graph with the prescribed partitioning does not exceed the budget.

Lemma 1.13, saying that we can enumerate all potential split partitions in subexponential time, will be crucial in our algorithm, as our algorithm presupposes a fixed split partition. Using this result, we may in subexponential time compute every possible split partition within range, and run our algorithm for completion to threshold graphs on each of these split graphs.

Cheap or Expensive?

We will from now on assume that all our input graphs $G = (V, E)$ are split graphs provided with a split partition (C, I), and that we are to solve Split Threshold Editing, that is, we have to respect the split partitioning. We are allowed to do this with subexponential time overhead, as per the previous section and specifically Lemma 1.13. In addition, we assume that $|V(G)| = O(k^2)$.

Given an instance (G, k) and a solution F, we define the *editing number* of a vertex v, denoted $\mathrm{en}_G^F(v)$, to be the number of edges in F incident to a vertex v. When G and F are clear from the context, we will simply write $\mathrm{en}(v)$. A vertex v will be referred to as *cheap* if $\mathrm{en}(v) \leq 2\sqrt{k}$ and *expensive* otherwise. We will call a set of vertices $U \subseteq V$ *small* provided that $|U| \leq 2\sqrt{k}$ and *large* otherwise.

Definition 10.2. Given an instance (G, k) with solution F, we call a vertex v *cheap* if $\mathrm{en}(v) \leq 2\sqrt{k}$.

The following observation will be used extensively.

Observation 10.3. *If $U \subseteq V(G)$ is a large set, then there exists a cheap vertex in U, or contrapositively: if a set $U \subseteq V(G)$ has only expensive vertices, then U is small. Specifically it follows that in any yes-instance (G, k) where F is a solution, there are at most $2\sqrt{k}$ expensive vertices.*

This gives the following win-win situation: If a set X is small, then we can "guess" it, which means that we can in subexponential time enumerate all candidates, and otherwise, we can guess a cheap vertex inside the set and its "correct" neighborhood. In particular, since the set of expensive vertices is small, we can guess it in the beginning. For the remainder of the proof we will assume that the graph G is a labeled graph, where some vertices are labeled as cheap and others as expensive. There will never be more than $2\sqrt{k}$ vertices labeled expensive. The idea is that we guess the expensive vertices at the start of the algorithm and then bring this information along when we recurse on subgraphs.

From now on, we assume that we are solving Split Threshold Editing on a graph with at most $O(k^2)$ vertices and a set of at most $2\sqrt{k}$ vertices are labeled expensive.

Splitting Pairs and Unbreakable Segments

Definition 10.4 (Splitting pair)**.** Let G be a graph, k an integer, F a solution of (G, k) and $(\mathcal{C}, \mathcal{I})$ a threshold decomposition of $G \oplus F$. We then say that the vertices $u \in I_a$ and $v \in C_b$ is a *splitting pair* if

- $a < b$,
- u and v are cheap,
- $\cup_{a<i<b} L_i$ consists of only expensive vertices. Recall from Definition 7.4 that $L_i = C_i \cup I_i$.

Definition 10.5 (Unbreakable)**.** Let G be a graph, k an integer, F a solution of (G, k) and $(\mathcal{C}, \mathcal{I})$ a threshold decomposition of $G \oplus F$. We then say that a sequence of levels $(C_a, I_a), (C_{a+1}, I_{a+1}), \ldots, (C_b, I_b)$ is an *unbreakable segment* if there is no splitting pair in the vertex set $\cup_{i \in [a,b]}(C_i \cup I_i)$.

Furthermore, we say that an instance (G, k) is *unbreakable* if there exists an optimal solution F and a threshold decomposition $(\mathcal{C}, \mathcal{I})$ of $G \oplus F$ such that the entire decomposition is an unbreakable segment. We also say that such a decomposition is a *witness* of G being unbreakable.

Definition 10.6. Let G be a graph and $(\mathcal{C}, \mathcal{I})$ a threshold decomposition of $G \oplus F$ for some solution F. Then we say that i is a *transfer level* if

- for every $j > i$ it holds that C_j contains no cheap vertices and
- for every $j < i$ it holds that I_j contains no cheap vertices.

Lemma 10.7. *Let (G, k) be a yes-instance of* Split Threshold Editing *with solution F such that G is unbreakable and $(\mathcal{C}, \mathcal{I})$ a witness. Then there is a transfer level in $(\mathcal{C}, \mathcal{I})$.*

Proof. Suppose for a contradiction that the lemma is false. Let a be maximal such that C_a contains a cheap vertex and b minimum such that I_b contains a cheap vertex. Since $i = a$ clearly satisfies the first condition, it must be the case that $b < a$. Increment b as long as $b+1 < a$ and there is a cheap vertex in $\cup_{i \in (b,a)} I_i$. Then decrement a as long as $b + 1 < a$ and there is a cheap vertex in $\cup_{i \in (b,a)} C_i$. Let u be a cheap vertex in C_a and v a cheap vertex in C_b. It follows from the procedure that they both exist. Observe that u, v is indeed a splitting pair, which is a contradiction to G being unbreakable and $(\mathcal{C}, \mathcal{I})$ being a witness. $\square$

Lemma 10.8. *Let (G, k) be an instance of* Split Threshold Editing *such that G is unbreakable and $(\mathcal{C}, \mathcal{I})$ a witness of this. Then the number of levels in $(\mathcal{C}, \mathcal{I})$ is at most $2\sqrt{k} + 1$.*

Proof. Let i be the transfer level in $(\mathcal{C}, \mathcal{I})$. It is guaranteed to exist by Lemma 10.7. Observe that for every $j > i$ it holds that C_i consists of expensive vertices and for every $j < i$ it holds that I_i consists of expensive vertices. It follows immediately that every level besides i contains at least one expensive vertex. As there are at most $2\sqrt{k}$ such vertices the result follows immediately. $\square$

Lemma 10.9. *Let (G, k) be an instance of* Split Threshold Editing *such that G is unbreakable, $(\mathcal{C}, \mathcal{I})$ is a witness of this and F a corresponding solution. If X is the set of cheap vertices in G then $(G \oplus F)[X]$ forms a complete split graph.*

Proof. Let t be the transfer level of the decomposition, u a cheap vertex in C_i and v a cheap vertex in I_j for some i and j. By the definition of t it holds that $i \leq t \leq j$. It follows immediately that u and v are adjacent in $G \oplus F$ and the proof is complete. $\square$

We will now describe the algorithm `unbreakAlg`. It takes as input an instance $(G, (C, I), k)$ of Split Threshold Editing, with the assumption that G is unbreakable and has split partition (C, I), and returns either an optimal solution F

for (G, k) where $|F| \leq k$ or correctly concludes that (G, k) is a no-instance. Assume that (G, k) is a yes-instance. Then there exists an optimal solution F and a threshold decomposition $(\mathcal{C}, \mathcal{I})$ of $G \oplus F$ that is a witness of G being unbreakable. First, we guess the number of levels ℓ in the decomposition, and by Lemma 10.8, we have that $\ell \in [0, 2\sqrt{k} + 1]$ and the transfer level $t \in [0, \ell]$. Then we guess where the at most $2\sqrt{k}$ vertices that are expensive in G are positioned in $(\mathcal{C}, \mathcal{I})$. Observe that from this information we can obtain all edges between expensive vertices in F. Finally, we put every cheap vertex in the level that minimizes the cost of fixing its adjacencies into the expensive vertices while respecting that t is the transfer level. From this information we can obtain all adjacencies between cheap and expensive vertices in F. Since the cheap vertices induces a complete split graph, we reconstructed F and hence we return it.

Lemma 10.10. *Given an instance (G, k) of* Split Threshold Editing *with G being unbreakable,* `unbreakAlg` *either gives an optimal solution or correctly concludes that (G, k) is a no-instance in time $2^{O(\sqrt{k} \log k)}$.*

Proof. Since the algorithm goes through every possible value for ℓ and t (according to Lemmata 10.7 and 10.8), and every possible placement of the expensive vertices, the only thing remaining to ensure is that the cheap vertices are placed correctly. However, since the cheap vertices form a complete split graph (according to Lemma 10.9), the only cost associated with a cheap vertex is the number of expensive vertices in the opposite side it is adjacent to. However, their placement is fixed, so we simply greedily minimize the cost of the vertex by putting it in a level that minimizes the number of necessary edits.

If we get a solution from the above procedure, this solution is optimal. On the other hand, if in every branch of the algorithm we are forced to edit more than k edges, then either (G, k) is a no-instance, or G is not unbreakable. Since the assumption of the algorithm is that G is unbreakable, we conclude that the algorithm is correct. □

Divide and Conquer

We now explain the main algorithm. The algorithm takes as input a graph G, together with a split partition (C, I) and a budget k. In addition, it takes a vertex set S which the algorithm is supposed to find an optimal solution for. The algorithm is recursive and either finds a splitting pair, in which it recurses on a subset of S, and if there is no splitting pair, then $G[S]$ is unbreakable, and thus it simply runs `unbreakAlg` on S. To avoid unnecessary recomputations, it uses memoization to solve already computed inputs.

The algorithm `solveAlg`$(G, (C, I), k, S)$ returns an optimal solution for the instance $(G[S], k)$, respecting the given split partition (C, I) in the following manner:

(i) Run `unbreakAlg`$(G[S], (C \cap S, I \cap S), k)$.

(ii) For every pair of cheap vertices $u \in I$ and $v \in C$, together with their correct neighborhoods N_u and N_v, and every pair of subsets $C_X \subseteq C$ and $I_X \subseteq I$ of expensive vertices we do the following: Let $X = I_X \cup C_X$, $R_C = N_u$, $U_I = N_v \cap I$, $R_I = I \setminus (X \cup U_I)$ and $U_C = S \setminus (X \cup R_C \cup U_I \cup R_I)$. Now, $U = U_I \cup U_C$ is the unbreakable segment, X is the set of expensive vertices between the splitting pair, and $R = R_I \cup R_C$ is the remaining vertices. We now

   (i) Run `unbreakAlg`$(G[U], (C \cap U, I \cap U), k)$ yielding a solution F_U,

   (ii) solve $G[X]$ optimally by brute force since it has size at most $2\sqrt{k}$, giving a solution F_X, and

   (iii) recursively call `solveAlg`$(G, (C, I), k, R)$ to solve the instance corresponding to the remaining vertices yielding F_R.

   Finally we return F, the union of F_U, F_X, and F_R together with all edges from $C \cap R$ and $I \cap (X \cup U)$, and all edges from $C \cap X$ to $I \cap U$.

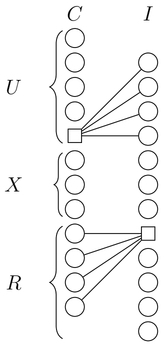


Figure 10.1: The partitioning of the vertex sets according to `solveAlg`. The square bags are the bags containing the splitting pair, U is an unbreakable segment and the bags of X contains exclusively expensive vertices. The edges drawn indicates the neighborhoods of the splitting pair across the partitions.

In *(i)* we consider the option that there are no splitting pairs in G. In *(ii)* (see Figure 10.1) we guess the uppermost splitting pair in the partition and the neighborhood of these two vertices. Then we guess all of the expensive vertices that live in between the two levels of the splitting pair. Observe that these expensive vertices together with the splitting pair partition the levels into three consecutive sequences. The upper one, U is an unbreakable segment, the middle, X are the expensive vertices and the lower one, R is simply the remaining graph. When we apply `unbreakAlg` on the upper part, brute force the middle one and recurse with `solveAlg` on the lower part, we get individual optimal solutions for each three, finally we may merge the solutions and add all the remaining edges (see end of *(2)*).

Lemma 10.11. *Given a split graph $G = (V, E)$ with split partition (C, I),* `solveAlg` *either returns an optimal solution for* Split Threshold Editing *on input $(G,(C,I),k,V)$, or correctly concludes that (G,k) is a no-instance.*

Proof. If (G,k) with split partition (C,I) is a yes-instance of Split Threshold Editing there is a solution F with threshold decomposition $(\mathcal{C},\mathcal{I})$ and a sequence of pairs $(u_1,v_1),(u_2,v_2),\ldots,(u_t,v_t)$ such that u_1,v_1 is the splitting pair highest in $(\mathcal{C},\mathcal{I})$, and u_2,v_2 in the highest splitting pair in the graph induced by the vertices in and below the level of v_1, etc. Since we in a state $(G,(C,I),k,S)$ try every possible pair of such cheap vertices and every possible neighborhood and set of expensive vertices, we exhaust all possibilities for any threshold editing of S of at most k edges. Hence, if there is a solution, an optimal solution is returned.

Thus, if ever an F is constructed of size $|F| > k$, we can safely conclude that there is no editing set $F^\star \subseteq C \times I$ of size at most k such that $G \oplus F^\star$ is a threshold graph. □

Lemma 10.12. *Given a split graph $G = (V, E)$ with split partition (C, I) and an integer k with $|V(G)| = O(k^2)$, the algorithm* `solveAlg` *terminates in time $2^{O(\sqrt{k}\log k)}$ on input $(G,(C,I),k,V)$.*

Proof. By charging a set S for which `solveAlg` is called with input $(G,(C,I),k,S)$ every operation except the recursive call, we need to *(I)* show that there are at most $2^{O(\sqrt{k}\log k)}$ many sets $S \subseteq V$ for which `solveAlg` is called, and *(II)* that the work done inside one such call is at most $2^{O(\sqrt{k}\log k)}$.

For Case *(I)*, we simply note that when `solveAlg` is called with a set S, the sets R on which we recurse are uniquely defined by u, v, N_u, N_v, X, and there are at most $O(k^4) \cdot 2^{O(\sqrt{k}\log k)^3} = 2^{O(\sqrt{k}\log k)}$ such configurations, so at most $2^{O(\sqrt{k}\log k)}$ sets are charged. Case *(II)* follows from the fact that we guess two vertices, u and v and three sets, N_u, N_v and X. For each choice we run `unbreakAlg`, which runs in time $2^{O(\sqrt{k}\log k)}$ by Lemma 10.10, and the brute force solution takes time $2^{O(\sqrt{k}\log(\sqrt{k}))}$. The recursive call is charged to a smaller set, and merging the solutions into the final solution we return, F, takes polynomial time.

The two cases show that we charge at most $2^{O(\sqrt{k}\log k)}$ sets with $2^{O(\sqrt{k}\log k)}$ work, and hence `solveAlg` completes after $2^{O(\sqrt{k}\log k)}$ steps. □

To conclude we observe that Theorem 28 follows directly from the above exposition. Given an input (G,k) to Threshold Editing, from the previous section we can in polynomial time obtain an equivalent instance with at most $O(k^2)$ vertices. Furthermore, by Lemma 1.13 we may in time $2^{O(\sqrt{k}\log k)}$ time assume we are solving the problem Split Threshold Editing. Finally, by Lemmata 10.11 and 10.12, the theorem follows.

Theorem 28. Threshold Editing *admits a $2^{o(\sqrt{k}\log k)} + n^{o(1)}$ subexponential time algorithm.*

By restricting the algorithm to either only add or remove edges, we reprove a result by Drange et al. [DFPV15].

Theorem 29 ([DFPV15]). THRESHOLD COMPLETION *and* THRESHOLD DELETION *admits* $2^{o(\sqrt{k}\log k)} + n^{o(1)}$ *subexponential time algorithms.*

10.2 Editing to Chain Graphs

We finally describe which steps are needed to change the algorithm above into an algorithm correctly solving CHAIN EDITING in subexponential time.

The main difference between CHAIN EDITING and THRESHOLD EDITING is that it is far from clear that the number of bipartitions is subexponential, that is, is there a bipartite equivalent of the bound of the potential split partitions as in Lemma 1.13? If we were able to enumerate all such "potential bipartitions" in subexponential time, we could simply run a very similar algorithm to the one above on the problem BIPARTITE CHAIN EDITING, where we are asked to respect the bipartition (see Section 8.2 for the definition of this problem).

It turns out that we indeed are able to enumerate all such potential bipartitions within the allowed time:

Lemma 10.13. *There is an algorithm which, given an instance* (G, k) *for* CHAIN EDITING, *enumerates* $\binom{|V|}{O(\sqrt{k})} = 2^{O(\sqrt{k}\log|V|)}$ *bipartite graphs* $H = (A, B, E')$ *with* $|E \oplus E'| \leq k$ *such that if* (G, k) *is a yes-instance, then one output* (H, k) *will be a yes-instance for* BIPARTITE CHAIN EDITING, *and furthermore is any yes-instance* (H, k) *is output, then* (G, k) *is a yes-instance. This also holds for the deletion and completion versions.*

Proof. We first mention that it is trivial to change the below proof into the proofs for the deletion and completion versions; One simply disallow one of the operations. So we will prove only the editing version. Furthermore, it is clear to see that if any output instance (H, k) is a yes-instance for BIPARTITE CHAIN EDITING, then (G, k) was a yes-instance for CHAIN EDITING.

Consider any solution $H = (A, B, E')$ for an input instance (G, k). If either $\min\{|A|, |B|\} \leq 5\sqrt{k}$, then we can simply guess every such in subexponential time. Hence, we assume that both sides of H are large. But this means, by Observation 10.3, that both A and B have cheap vertices. Let v_A be a cheap vertex as low as possible in A and v_B be a cheap vertex as high as possible in B. It immediately follows from the same observation that the set of vertices below v_A, A_X is a set of expensive vertices, and the same for the vertices above v_B, B_X. Since v_A and v_B, we know that we can in subexponential time correctly guess their neighborhoods in H and we can similarly guess A_X and B_X.

Now, since we know v_A, v_B, $N_H(v_A)$ and $N_H(v_B)$, as well as A_X and B_X, the only vertices we do now know where to place, are the vertices in A which are in the levels above $\operatorname{lev}(v_B)$, call them A_Y, and the vertices in b which are in the levels below $\operatorname{lev}(v_A)$. However, we know which set this is, that is, we know $Z = A_Y \cup B_Y$. Define now $A_M = A \setminus (A_Y \cup A_X \cup \{v_A\})$ and similarly $B_M = B \setminus (B_Y \cup B_X \cup \{v_B\})$. These are the vertices living in the middle of A and B, respectively.

We now know that the vertices of Z should form an independent set. This follows from the fact that A_M and B_M are both non-empty. Hence, the vertices of A_Y are in higher levels than all of B_Y, and since there are no edges going from a vertex in A to a vertex lower in B, and each of A and B are independent sets, Z must be an independent set.

The following is the crucial last step. We can in subexponential time guess the partitioning of levels of both A_X and of B_X, since they are both of sizes at most $2\sqrt{k}$. When knowing these levels, we can greedily insert each vertex in Z into either A and B by pointwise minimizing the cost; A vertex $z \in Z$ can safely be places in the level of A or B which minimizes the cost of making it adjacent to only the vertices of B_X above its level, or by making it adjacent to only the vertices below its level in A_X. □

Given the above lemma, we may work on the more restricted problem, Bipartite Chain Editing. The rest of the algorithm actually goes through without any noticeable changes:

Theorem 30. Chain Editing *is solvable in time* $2^{O(\sqrt{k}\log k)} + n^{O(1)}$.

Proof. On input (G, k) we first run the kernelization algorithm from Section 9.2, and then we enumerate every potential bipartition according to Lemma 10.13. Now, for each bipartition (A, B) we make A into a clique, and run the Split Threshold Editing algorithm from Section 10.1 (see also Proposition 7.8).

Now, (G, k) is a yes-instance if and only if there is a bipartition (A, B) such that when making A into a clique, the resulting instance is a yes- instance for Split Threshold Editing. □

Theorem 31. Chain Deletion *and* Chain Completion *are solvable in time* $2^{O(\sqrt{k}\log k)} + n^{O(1)}$.

Proof. By allowing edges only to be deleted or added in the algorithm and definitions, we obtain the result. □

Chapter 11

Concluding remarks

In this part we showed that the problems of editing edges to obtain a threshold graph and editing edges to obtain a chain graph are NP-complete. The latter solves a conjecture in the positive from Natanzon et al. [NSS01] and both results answer open questions from Sharan [Sha02], Burzyn et al. [BBD06], and Mancini [Man08].

On the positive side, we show that both Threshold Editing and Chain Editing admit quadratic kernels, i.e., given a graph (G, k), we can in polynomial time find an equivalent instance (G', k) where $|V(G')| = O(k^2)$, and furthermore, G' is an induced subgraph of G. We also show that these results hold for the deletion and completion variants as well, and these results answer open questions by Liu et al. [LWG14] in a recent survey on kernelization complexity of graph modification problems.

Finally we show that both problems admit subexponential time algorithms of complexity $2^{O(\sqrt{k}\log k)} + n^{O(1)}$. This answers a recent open question by Liu et al. [LWY+15]. These are the only two graph classes known towards which completion, deletion, and editing all are NP-complete, admit polynomial kernels and are subexponential parameterized time solvable.

There are now algorithms solving Threshold Editing (Theorem 28), Chain Editing (Theorem 30) and Chordal Editing[1] (Cao & Marx [CM14]) in time $2^{O(\sqrt{k}\log k)}n^{O(1)}$. Furthermore, assuming ETH, we have now established that there is no $2^{o(\sqrt{k})}n^{O(1)}$ time algorithm for any of these problems. This leaves a complexity gap and it would be interesting to see if we can achieve tight bounds, by improving the algorithms, the lower bounds or both.

[1]Here, the authors take Chain Editing to allow vertex deletions.

Part IV

Modifications in graphs of bounded degree

Chapter 12

Introduction

Graph modification problems have been a fundamental part of computational graph theory throughout its history [GJ79a, A1. Graph Theory]. In these classic problems you are to apply at most k modifications to an input graph G to make it adhere to a specific set of properties, where both the modifying operations and the target properties are problem specific. Unfortunately, even when considering vertex deletion to hereditary graph classes, the modification problems often regarded as the most tractable, almost all of them are NP-complete [LY80]. A similar dichotomy is yet to appear for edge modification problems and hence the classic complexity landscape seems far more involved. However, various results display the NP-hardness of the edge variants as well [Yan81b, EMC88, BBD06].

We will restrict our attention to hereditary graph classes characterized by finite sets of forbidden induced subgraphs. Hence, for every graph class studied there is a finite set of graphs $\mathcal{H}$ such that a graph G is in the graph class if and only if no graph in $\mathcal{H}$ is an induced subgraph of G. In this situation Cai's theorem [Cai96] shows that all $\mathcal{H}$-free modification problems are *fixed-parameter tractable*, that is, they are all solvable in time $f(k) \cdot n^{O(1)}$. And furthermore, every vertex deletion problem admits a classic $O(k^d)$ polynomial kernel, where d is the maximum number of vertices in a graph in $\mathcal{H}$, based on the sunflower lemma [FG06, AK10]. However, for edge modification problems we are far away from such a classification of the complexity landscape. In particular, P_4-free edge deletion admits a polynomial kernel [GHPP13], C_4-free edge deletion does not [CC15] and $K_{1,3}$-deletion (also referred to as claw-deletion) remains open.

The edge modification problems characterized by a finite set of forbidden induced subgraphs $\mathcal{H}$ are often referred to as $\mathcal{H}$-Free Edge Completion, $\mathcal{H}$-Free Edge Deletion and $\mathcal{H}$-Free Edge Editing, where one is to add, remove or both add and remove k edges to make the graph $\mathcal{H}$-free.

Gramm, Guo, Hüffner and Niedermeier [GGHN09], and Guo [Guo07] gave kernels for several graph modification problems to graph classes characterized by a finite set of forbidden induced subgraphs. Several positive results followed, which led Fellows, Langston, Rosamond, and Shaw to ask whether all $\mathcal{H}$-free modification problems admit polynomial kernels [FLRS07].

This was refuted by Kratsch and Wahlström [KW13] who showed that for

$\mathcal{H} = \{H\}$ where H is a certain graph on seven vertices, $\mathcal{H}$-Free Edge Deletion, as well as $\mathcal{H}$-Free Edge Editing, does not admit a polynomial kernel unless NP $\subseteq$ coNP/poly.[1] Without stating it explicitly, but revealed by a more careful analysis of the inner workings of their proofs, Kratsch and Wahlström actually showed something even stronger; namely that the result holds when restricted to 6-degenerate graphs, both for the deletion and for the editing version.

This line of research was followed up by Guillemot et al. [GHPP13] showing large classes of simple graphs for which $\mathcal{H}$-Free Edge Deletion is incompressible, which was further developed by Cai and Cai [CC15]; Combining these results, we now know that when H is a path or a cycle, $\mathcal{H}$-Free Edge Deletion, Editing *and* Completion is compressible if and only if H has at most three edges, that is, only for the simplest graphs. Although being classified for a single path or cycle, there is still a vast number of obstructions for which we do not know the kernelization complexity.

Bounded degree

In dealing with the inherent intractability of graph modification problems, Natanzon, Shamir, and Sharan [NSS01] suggested to study $\mathcal{H}$-Free Edge Deletion on bounded degree input graphs. Recently, following this direction of research, Aravind, Sandeep and Sivadasan [ASS14] were able to show that as long as every graph $H \in \mathcal{H}$ is connected, the problem $\mathcal{H}$-Free Edge Deletion admits a polynomial kernel of size

$$\Delta^{f(\mathcal{H})} \cdot k^{g(\mathcal{H},\Delta)},$$

for two functions g and f. In particular, since $\mathcal{H}$ is fixed for a specific problem, this yields a polynomial kernel for every fixed maximum degree Δ.

One should note that many modification problems remain NP-complete for bounded degree graphs. Komusiewicz and Uhlmann [KU12] showed that even for simple cases like $\mathcal{H} = \{P_3\}$, the path on three vertices, $\mathcal{H}$-Free Edge Deletion—also known as Cluster Deletion—is NP-complete, even on graphs of maximum degree 6. Later, it was also shown that P_4-free Edge Deletion and Editing (Cograph Editing) and $\{C_4, P_4\}$-free Edge Deletion and Editing (Trivially Perfect Editing) [DP15] had similar properties; NP-complete, even on graphs of maximum degree 4.

A parameterized perspective

Since the problems already are FPT for general graphs, and hence also for graphs of bounded degree, we focus on providing polynomial kernels. The first result presented is several, simultaneously applicable improvements upon the above mentioned result by Aravind, Sandeep and Sivadasan [ASS14]. First, we are able to remove the condition requiring all graphs of $\mathcal{H}$ to be connected. As

[1] NP $\subseteq$ coNP/poly implies that PH is contained in Σ_3^p. It is widely believed that PH does not collapse, and hence it is also believed that NP $\not\subseteq$ coNP/poly. We will throughout this section assume that NP $\not\subseteq$ coNP/poly.

many interesting graph classes (threshold graphs, split graphs e.g.) are described by disconnected forbidden subgraphs, this is a major extension. Second, we complement it by proving that the same kernels can be obtained when considering $\mathcal{H}$-Free Edge Editing. And third, we improve the kernel dependency on $\Delta(G)$. The novelty of our approach lies within a better understanding of how forbidden subgraphs are introduced when edges are modified in the input graph. Due to this, we can localize the crucial part of the instance even when both forbidden subgraphs and modifications are spread throughout the graph.

We continue by providing several hardness results. First, we prove that somewhat surprisingly the positive result does not extend to the completion variant. More specifically, there exists a finite set $\mathcal{H}$ such that $\mathcal{H}$-Free Edge Completion does not admit a polynomial kernel, even on input graphs of maximum degree 5, unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$. The intuition behind this distinction is that for both the deletion and editing variants, one has the possibility of preventing propagation by locally removing all involved edges. This is realized by Lemmata 13.9 and 13.10. However, this is not the case for completion problems. In particular, one can be forced to complete a single obstruction in such a way that this creates new obstructions, that again when fixed creates new obstructions and so forth. This is utilized among others in the selector tree of Section 14.1.

Furthermore, we prove that for both $\mathcal{H}$-Free Edge Editing and $\mathcal{H}$-Free Edge Deletion there is no hope for polynomial kernels, even when restricted to 2-degenerate graphs. It can easily be observed that the same proofs can be applied to generalize the results to K_9-minor free graphs.

We now have complete information on the kernelization complexity of edge and vertex modification problems when the target graph class is characterized by a finite set of forbidden induced subgraphs, on bounded degree and 2-degenerate input graphs. Recall that the yes answer for the vertex deletion version on general graphs is obtained by a simple reduction from the $\mathcal{H}$-Free Vertex Deletion problem to the d-Hitting Set problem, which, using the sunflower lemma [ER60], can be shown to admit a polynomial kernel [AK10].

	bounded degree	2-degenerate
Deletion	Yes ([ASS14], Theorem 32)	No (Theorem 34)
Completion	No (Theorem 33)	No (Theorem 33)
Editing	Yes (Theorem 32)	No (Theorem 35)
Vertex deletion	Yes	Yes

Table 12.1: Overview of polynomial kernelization complexity for graph modification on bounded degree and degenerate input graphs. The table shows that there is no distinction between disconnected graphs, and that the completion variant is notoriously incompressible—bounded degree does not help compressing completion problems.

12.1 Preliminaries

For a finite obstruction set $\mathcal{H}$, we define $\mathrm{diam}_{\mathcal{H}}$ to be the maximum diameter over all components over all graphs in $\mathcal{H}$. Note that this differs from the diameter of a disconnected graph, which we defined as infinity. The size of the largest graph in $\mathcal{H}$ we denote by $n_{\mathcal{H}} = \max\{|V(H)| \text{ for } H \in \mathcal{H}\}$.

Definition 12.1 (H-packing)**.** Given a graph G and an obstruction H we say that $\mathcal{X} \subseteq 2^{V(G)}$ forms an *H-packing* in G if

(i) $G[X]$ and H are isomorphic for every $X \in \mathcal{X}$, and

(ii) X and Y are disjoint for every $X, Y \in \mathcal{X}$.

Observation 12.2. *Given a graph G and an obstruction H we can obtain a maximal H-packing $\mathcal{X}$ in $O(|V(H)|^2 \cdot n^{|V(H)|})$ time.*

Planar Cubic Vertex Cover

We have two main polynomial kernel hardness results, both of which will be reduced from the problem Cubic Planar Vertex Cover, which is the famous Vertex Cover problem restricted to regular planar input graphs of degree three. The following result shows the validity of reducing from this restricted instance:

Proposition 12.3 ([Moh01])**.** Vertex Cover *is* NP*-complete on cubic planar graphs.*

It is well known that planar graphs can be recognized in polynomial time, so an algorithm can simply reject the input if the graph is not regular or non-planar. When we later will make a cross-composition argument, we will reduce from Cubic Planar Vertex Cover. We may allow us, on $t = 2^r$ instances of Cubic Planar Vertex Cover on n vertices, m edges and a budget of $k' \leq n$, to have a budget of size bounded by $(\log(t))^{O(1)} + |G_i|^{O(1)} = \log(t)^{O(1)} + (n + m + k')^{O(1)}$.

An upper bound

The following lemma will be used when analyzing the size of the polynomial kernel.

Lemma 12.4. *For $x > 1$ it holds that $\frac{1}{\log_x \frac{x+1}{x}} \leq (x+1)\ln x$.*

Proof. First, we observe that

$$\frac{1}{\log_x \frac{x+1}{x}} = \frac{1}{\frac{\ln \frac{x+1}{x}}{\ln x}} = \frac{\ln x}{\ln \frac{x+1}{x}}.$$

Hence, it remains to prove that $\frac{1}{\ln \frac{x+1}{x}} \leq x+1$, or equivalently that $\frac{1}{x+1} \leq \ln \frac{x+1}{x}$. This follows from the following calculation:

$$\begin{aligned}
\ln \frac{x+1}{x} &= \ln(x+1) - \ln(x) \\
&= \int_x^{x+1} \frac{1}{y} dy \\
&\geq (x+1-x) \cdot \min \left\{ \frac{1}{y} \mid y \in [x, x+1] \right\} \\
&= \frac{1}{x+1}.
\end{aligned}$$

□

Chapter 13

A polynomial kernel

In this section we prove that for any finite set of obstructions $\mathcal{H}$, deleting or editing at most k edges to make an input graph of bounded degree $\mathcal{H}$-free admits polynomial kernels. More precisely, both $\mathcal{H}$-Free Edge Editing and $\mathcal{H}$-Free Edge Deletion admits polynomial kernels on bounded degree graphs.

The argument consists of two parts. First, we identify a set of critical vertices in the input graph G, called the obstruction core Z. Based on this set we can decompose any set of modifications F in G. The decomposition leads to the construction of a set of vertices in the graph, called the extended obstruction core Z^+. The first crucial property of Z^+ is that if F modifies $G[Z^+]$ into an $\mathcal{H}$-free graph, then F also modifies G into an $\mathcal{H}$-free graph. In other words, whichever obstructions we want to remove in the input graph should be done within the extended obstruction core. The second crucial property is that the extended obstruction core can be proved to live within a ball around the obstruction core, were the radius depends on how well the solution decomposes. This ball will in the end constitute the kernel. In the second part of the argument we prove that every minimal solution decomposes well. Hence we can bound the size of the ball containing the extended obstruction core and obtain a polynomial kernel.

We point out that we have considered the editing variant of the problem where we are allowed to surpass the original maximum degree in the graph by adding edges. However, it is true that there is always a solution that at most doubles the maximum degree of the graph since if more edges are added one might as well remove all edges incident to the vertex. The validity of this is proved in Lemma 13.10. Furthermore, it can be argued that the studied version of the problem is the most general one. This is due to the fact that adding every super graph of the star with $\Delta(G) + 1$ leaves to the obstruction set ensures that any solution respects the current maximum degree.

Before we continue, we observe that both the case when the maximum degree of the graph is 0 and when $k = 0$ are instances that can be solved in polynomial time. Hence, for the remainder of the section we will assume that $\Delta(G)$ and k are positive.

13.1 Cores and layers

In this section we introduce the concepts of obstruction cores and extended obstruction cores. They are heavily based on the notion of shattered obstructions, which are the set of obstructions we get from $\mathcal{H}$ if we take every connected component as an obstruction. It follows immediately that every shattered obstruction is connected. Recall from the preliminaries that the size of the largest graph in $\mathcal{H}$ is denoted by $n_{\mathcal{H}}$.

Definition 13.1 (Shattered obstructions)**.** Given a set of obstructions $\mathcal{H}$ we define the *shattered obstructions*, denoted $\mathcal{H}^\star$, as follows:

$$\{C \mid C \text{ is a connected component of } H \text{ and } H \text{ is a graph in } \mathcal{H}\}.$$

Based on shattered obstructions we now define an obstruction core and explain how such a set of not too large size can be obtained.

Definition 13.2 (Obstruction core)**.** Let (G, k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion). We then say that a set $Z \subseteq V(G)$ is an *obstruction core in* G if for every shattered obstruction H in G it holds that either:

(i) $V(H) \subseteq Z$ or

(ii) there is an H-packing in $G[Z]$ of size at least $(\Delta(G) + 1) \cdot n_{\mathcal{H}} + 2k + 1$.

Observation 13.3. *Given an instance (G, k) of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion) we can in $O(|\mathcal{H}^\star| n^{n_{\mathcal{H}}})$ time obtain an obstruction core Z in G of size at most*

$$n_{\mathcal{H}}|\mathcal{H}^\star|((\Delta(G) + 1) \cdot n_{\mathcal{H}} + 2k + 1).$$

Proof. Let Z be the empty set initially. Then for every shattered obstruction H we find a maximal H-packing $\mathcal{X} = X_1, \ldots, X_t$ and add the following set $\bigcup_{i=1}^{p} X_i$ to Z, where $p = \min(t, (\Delta(G) + 1) \cdot n_{\mathcal{H}} + 2k + 1)$. The time complexity follows from Observation 12.2. □

The next definitions are the ones of layer decompositions and core extensions, arguably the most central definitions of the kernelization algorithm. They are both with respect to a fixed obstruction core Z and a solution F. The goal is to describe how solutions behave in G. We decompose a solution into several layers such that the first layer consists of the edges of F that are contained in Z. The second layer consists of the edges of F that are contained in scattered obstructions created when the modifications in Z was done, and so forth. The extended core is a set of vertices encapsulating all scattered obstructions either in $G[Z]$ or created in G when doing the modifications of the layers. Observe that when computing the kernel, no such solution F is available and the analysis with respect to this set is done solely to ensure that the kernelized instance is equivalent to the original one.

Layer decompositions and core extensions

Let (G, k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion), $F \subseteq [V(G)]^2$ and Z an obstruction core. We construct the *layer decomposition* $F_1, \ldots, F_\ell$ of F as follows: Let $G_1 = G$, $R_1 = F$ and $Z_1 = Z$. Then, inductively we construct the set $X = R_i \cap [Z_i]^2$. If X is empty we stop the process, otherwise we let $F_i = X$, $G_{i+1} = G_i \oplus F_i$ and $R_{i+1} = R_i \setminus F_i$. Furthermore, we let

$$W_{i+1} = \{v \in H \mid H \text{ is a shattered obstruction in } G_{i+1} \text{ with } [V(H)]^2 \cap F_i \neq \emptyset\}.$$

Based on this we let $Z_{i+1} = Z_i \cup W_{i+1}$. With the construction above in mind we introduce some notation and terminology:

Definition 13.4 (Intermediate graphs and the extended core)**.** We will refer to G_i as the *i'th intermediate graph*, R_i as the *i'th remainder*, Z_i as the *i'th core extension* and ℓ as the *solution depth* (all with respect to G, Z and F). Furthermore, we will refer to $G^+ = G_{\ell+1}$ as the *resulting graph* and $Z^+ = Z_{\ell+1}$ as the *extended core.*

The next lemma shows that if there is an obstruction in some intermediate graph such that for every connected component of the obstruction, the component is either inside the corresponding core extension or not modified at all so far by the layers, then there is an isomorphic obstruction contained entirely within the core extension. The intuition is that any untouched connected component has a large packing in Z and hence it can be replaced by an isomorphic subgraph inside Z that both avoids the modifications and the neighborhood of the rest of the obstruction.

Lemma 13.5. *Let (G, k) be an instance of $\mathcal{H}$-*Free Edge Editing*($\mathcal{H}$-*Free Edge Deletion*), Z an obstruction core of G, and $F \subseteq [V(G)]^2$ with $|F| \leq k$ and $F_1, \ldots, F_\ell$ a layer decomposition of F. For an integer $j \in [1, \ell + 1]$ let G_j be the intermediate graph and Z_j the core extension with respect to G, Z and F. Let H be an obstruction in G_j with connected components $H_1, \ldots, H_t$ such that every H_i satisfies either:*

(i) $V(H_i) \subseteq Z_j$ or

(ii) $H_i = G[V(H_i)]$.

Then there is an obstruction H' in G_j isomorphic to H with $V(H') \subseteq Z_j$ and $V(H') \setminus V(H) \subseteq Z$.

Proof. For convenience we denote neighborhoods in G_j by N_j. Let H' be the disjoint union of every H_i such that $V(H_i) \subseteq Z_j$ and $\mathcal{L}$ the list containing every H_i not added to H'. Let H_i be an element of $\mathcal{L}$. We will now prove that there is an H_i' in $G_j[Z_j \setminus N_j[H']]$ such that H_i and H_i' are isomorphic. Let $\mathcal{X}_i$ be the maximal H_i-packing obtained when constructing Z. Since $V(H_i)$ is not contained in Z_j (and hence Z) and H_i's edges are as in G it holds that $|\mathcal{X}_i| \geq (\Delta(G)+1) \cdot n_{\mathcal{H}} + 2k + 1$ by the definition of obstruction cores. This yields that

$(\Delta(G)+1)\cdot n_{\mathcal{H}}+2k+1$ of the elements of the packing was added to Z. Furthermore, we observe that $|V(H')| \leq n_{\mathcal{H}}$ and hence that $|N_G(H')| \leq \Delta \cdot n_{\mathcal{H}}$. It follows immediately that $|N_j(H')| \leq \Delta \cdot n_{\mathcal{H}} + k$ and hence that $|N_j[H']| \leq (\Delta+1)\cdot n_{\mathcal{H}}+k$. By the previous arguments it follows that there is an H_i-packing in $G_j[Z \setminus N_j[H']]$ of size at least $k+1$. And hence, by the pigeon hole principle there is an H_i' isomorphic to H_i in $G_j[Z_j \setminus N_j[H']]$ such that $[V(H_i')]^2$ and F are disjoint.

To complete the proof we do the following for every H_i in $\mathcal{L}$. We find an H_i' as described above, add H_i' to H' and remove H_i from $\mathcal{L}$. Since $H_1, \dots, H_t$ are the connected components of H it follows that H and H' are isomorphic. Furthermore, $V(H')$ is clearly contained in Z_j and $V(H') \setminus V(H)$ in Z. □

This possibility of moving obstructions to the inside of core extensions immediately yields several very useful lemmata. The first one gives the true power of an extended core, namely that if a set of edges is a solution for the graph induced on its extended core it also is a solution for the entire graph.

Lemma 13.6. *Let (G, k) be an instance of $\mathcal{H}$-*Free Edge Editing*($\mathcal{H}$-*Free Edge Deletion*), Z an obstruction core of G, and $F \subseteq [V(G)]^2$. Construct the layer decomposition $F_1, \dots, F_\ell$ of F with respect to Z, let $F' = \cup_{i=1}^{\ell} F_i$ and let Z^+ be the extended core with respect to Z and F. It then holds that:*

$$(G \oplus F')[Z^+] \text{ is } \mathcal{H}\text{-free if and only if } G \oplus F' \text{ is } \mathcal{H}\text{-free.}$$

Proof. Recall that $G^+ = G \oplus F'$. It is trivial that if there is an obstruction H in $G^+[Z^+]$ then H is also an obstruction in G^+. For the other direction, let H be an obstruction in G^+ and $H_1, \dots, H_t$ the connected components of H. Observe that by the definition of Z^+ it holds that every H_i satisfies either *(i)* or *(ii)* of Lemma 13.5 with $j = \ell + 1$. It follows that there is an obstruction H' in G^+ with $V(H') \subseteq Z^+$. Hence H' is an obstruction in $G^+[Z^+]$, which completes the argument. □

Next, we prove that a layer decomposition is a partition of the solution, given that the solution is minimal.

Lemma 13.7. *Let (G, k) be an instance of $\mathcal{H}$-*Free Edge Editing*($\mathcal{H}$-*Free Edge Deletion*), Z an obstruction core of G, F a minimal solution and $F_1, \dots, F_\ell$ the layer decomposition of F with respect to Z. It then holds that $F_1, \dots, F_\ell$ forms a partition of F.*

Proof. Let F_i and F_j be two layers with $i < j$. It follows immediately from the definition of layer decompositions that $F_j \subseteq R_j \subseteq R_i \setminus F_i$ and hence F_i and F_j are disjoint. For convenience we let $F' = \cup_{i \in [1,\ell]} F_i$. We now prove that $F' = F$. It follows from the definition of layer decompositions that $F' \subseteq F$. Assume for a contradiction that $F' \subsetneq F$. Consider the final graph $G^+ = G \oplus F'$. If G^+ is $\mathcal{H}$-free it follows that F is not a minimal solution, yielding a contradiction.

Hence, we can assume that G^+ is not $\mathcal{H}$-free. It follows immediately from Lemma 13.6 that $G^+[Z^+]$ is also not $\mathcal{H}$-free. Furthermore, we know by the definition of layer decompositions that $G^+[Z^+] = (G \oplus F)[Z^+]$. And hence $G \oplus F$ is not $\mathcal{H}$-free, contradicting that F is a solution. □

The following lemma gives us a partial tool for encapsulating an extended core without knowing the solution beforehand. The next section is dedicated to turning this partial tool into something useful. Recall that $\mathrm{diam}_{\mathcal{H}}$ is the maximum diameter of a connected component of a graph in $\mathcal{H}$.

Lemma 13.8. *Let (G, k) be an instance of $\mathcal{H}$-*Free Edge Editing*($\mathcal{H}$-*Free Edge Deletion*), Z an obstruction core of G, $F \subseteq [V(G)]^2$ and Z^+ the extended core with respect to Z and F. It then holds that*

$$Z^+ \subseteq B(Z, \ell \cdot \mathrm{diam}_{\mathcal{H}}).$$

Proof. Let $Z_1, \ldots, Z_{\ell+1}$ be the extended cores. Instead of proving the lemma directly we prove the following, stronger claim:

($\star$) For every Z_i it holds that $Z_i \subseteq B(Z, (i-1) \cdot \mathrm{diam}_{\mathcal{H}})$.

Since $Z^+ = Z_{\ell+1}$, it is clear that ($\star$) implies the lemma. The proof of ($\star$) is by induction. First, we observe that ($\star$) holds for $i = 1$ by the definition of balls, since $Z = Z_1$. Assume for the induction step that ($\star$) holds for i. Let v be a vertex in Z_{i+1}. If v is also in Z_i we are done by assumption. Hence, we assume v to be a vertex in $Z_{i+1} \setminus Z_i$. Or in other words, v is in W_{i+1}. By definition there is a scattered obstruction H in G_{i+1} and an edge uw in F_i such that both u, v and w are in H. Observe that the distance between u and v is at most $\mathrm{diam}_{\mathcal{H}}$ and recall that u is in $Z_i \subseteq B(Z, (i-1) \cdot \mathrm{diam}_{\mathcal{H}})$. It follows immediately that v is in $B(Z, i \cdot \mathrm{diam}_{\mathcal{H}})$ and hence the proof is complete. □

13.2 Solutions are shallow

In this section we prove that the depth of any solution is bounded logarithmically by the size of the solution. This, combined with Lemma 13.8 gives that linearly in k many balls of logarithmic radius is sufficient to encapsulate an extended core. To motivate that we obtain a polynomial kernel, observe that a ball of logarithmic radius in a bounded degree ball is of polynomial size.

First, we prove that when considering any layer we can always find a set of vertices of the same size, for which removal would result in an $\mathcal{H}$-free graph. Next we prove that as long as the graph is not very small, removing a set of vertices from the graph has the same effect as modifying the graph such that the set becomes a set of isolates.

Lemma 13.9. *Let (G, k) be an instance of $\mathcal{H}$-*Free Edge Editing*($\mathcal{H}$-*Free Edge Deletion*), Z an obstruction core of G, F a minimal solution of the instance and $F_1, \ldots, F_\ell$ the layer partition of F with respect to Z. For every $i \in [1, \ell]$ there exist a set Y with $Y \leq |F_i|$ such that $G_i - Y$ is $\mathcal{H}$-free.*

Proof. We construct Y as follows: For every edge uv in F_i we add to Z the endpoint furthest away from Z. If it is a tie, we choose an arbitrary endpoint.

Assume for a contradiction that $G_i - Y$ is not $\mathcal{H}$-free. Let H be an obstruction in $G_i - Y$ and $H_1, \ldots, H_t$ the connected components of H.

First, we consider the case when $i = 1$. We then apply a modification of the proof of Lemma 13.5. The idea is as follows: Let H' be the disjoint union of the components of H contained in Z and H_x a component not in Z. Then there is an H_x-packing of size $k + 1$ in Z avoiding the closed neighborhood of H'. We observe that Y intersects with at most k of the elements of the packing and hence we can find a subgraph H'_x in $G[Z]$ not intersecting with Y such that H_x and H'_x are isomorphic. Add H'_x to H' and continue with the next component not contained in Z. It follows immediately that H' is also an obstruction in G_2. By definition $G_2[Z] = G^+[Z]$ and hence H' is an obstruction in G^+. This contradicts F being a solution.

If $i \geq 2$ it holds that Y and Z are disjoint. This is true since if both endpoints of an edge are included in Z, the edge would be in F_1 and not F_i. It holds by the definition of Y and H that $[V(H)]^2 \cap F_i$ is empty. Furthermore, by the definition of layer decompositions it holds that if some connected component H_x of H intersects with some F_j with $j < i$ then $V(H_x) \subseteq Z_{j+1} \subseteq Z_i$. Hence we can apply Lemma 13.5 to obtain an obstruction H' in G_i with $V(H') \subseteq Z_i$. Since $V(H) \subseteq V(G) \setminus Y$ and $V(H') \setminus V(H) \subseteq Z$ it follows that H' is an obstruction in $G_i \setminus Y$. It follows immediately that H' is also an obstruction in G_{i+1}. By definition $G_{i+1}[Z_i] = G^+[Z_i]$ and hence H' is an obstruction in G^+. This contradicts F being a solution and completes the proof. □

We now show that if we the graph is big compared to the budget, the bounded degree and $\mathcal{H}$, then deleting vertices and deleting edges are in some sense equivalent.

Lemma 13.10. *Let (G, k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion), X a set of vertices in G and E_X the set of edges incident to vertices in X. It then holds that either*

(i) $|V(G)| < |X| + k + 2(\Delta(G) + 1)n_{\mathcal{H}}$ *or*

(ii) the instances $(G - X, k')$ and $(G - E_X, k')$ are equivalent for every k'.

Proof. We assume that *(i)* does not apply and prove that this implies *(ii)*. It is trivial that if $(G - E_X, k')$ is a yes-instance then $(G - X, k')$ is also a yes-instance. For the other direction, assume for a contradiction that $(G - X, k')$ is a yes-instance and that $(G - E_X, k')$ is a no-instance. Let F be a solution of $(G - X, k')$. For convenience we define $G_V = (G - X) \oplus F$ and $G_E = (G - E_X) \oplus F$. Let H be an obstruction in G_E and B the set of vertices $V(H) \setminus X$. By construction $G_E[V(H) \cap X]$ is an independent set. Observe that $G_V[B] = G_E[B]$

and that $|N_{G_E}(V(H))| \leq \Delta(G) \cdot n_{\mathcal{H}} + k$. It follows immediately that

$$\begin{aligned}
&|V(G_E) \setminus (X \cup N_{G_E}[V(H)])| \\
&\geq |V(G_E)| - |X| - |N_{G_E}[V(H)]| \\
&\geq |X| + k + 2(\Delta(G)+1)n_{\mathcal{H}} - |X| - n_{\mathcal{H}} - \Delta(G) \cdot n_{\mathcal{H}} - k \\
&= 2(\Delta(G)+1)n_{\mathcal{H}} - n_{\mathcal{H}} - \Delta(G) \cdot n_{\mathcal{H}} \\
&= (\Delta(G)+1)n_{\mathcal{H}}.
\end{aligned}$$

And hence we can obtain an independent set I of size $|X \cap V(H)|$ that is entirely outside of both X and $N_{G_E}[V(H)]$. Let $H' = G_V[I \cup B]$ and observe that H' is isomorphic to H, contradicting G_V being $\mathcal{H}$-free. □

With the two previous lemmata in mind we present the main idea of the shallowness of a solution. Intuitively, if for any level of a decomposed solution we do a factor $\Delta(G)$ more modifications in the future than we do in this particular level, we could instead remove a set of edges related to this layer and stop any further propagation. This ensures that in any optimal solution the size of the union of the remaining layers are bounded by the size of the current layer and the maximum degree of the graph.

Lemma 13.11. *Given an instance (G, k) of $\mathcal{H}$-*Free Edge Editing *($\mathcal{H}$-*Free Edge Deletion*), an obstruction core Z, an optimal solution F and its layer decomposition $F_1, \ldots, F_\ell$ it holds that either*

(i) $|V(G)| < k + 2(\Delta(G)+1) \cdot n_{\mathcal{H}}$ *or*

(ii) $\Delta(G) \cdot |F_i| \geq |R_{i+1}|$ *for every $i \in [1, \ell]$.*

Proof. We assume that *(i)* does not apply and hence that $|V(G)| \geq k + (\Delta(G) + 2) \cdot n_{\mathcal{H}}$. Assume for a contradiction that there is an $i \in [1, \ell]$ such that *(ii)* does not hold. Specifically, i is so that $\Delta(G) \cdot |F_i| < |R_{i+1}|$. By Lemma 13.9 there is a set of vertices Y with $|Y| \leq |F_i|$ such that $G_i - Y$ is $\mathcal{H}$-free. It follows by Lemma 13.10 with $k' = 0$ that $G_i - E_Y$ is also $\mathcal{H}$-free. Let $F' = (\cup_{j \in [1,i-1]} F_j) \cup E_Y$ and observe that $G \oplus F'$ is $\mathcal{H}$-free. By the following calculations:

$$|F'| \leq |\cup_{j \in [1,i-1]} F_j| + |E_Y| < |\cup_{j \in [1,i-1]} F_j| + |R_{i+1}| = |F|$$

we conclude that $|F'| < |F|$. This contradicts the optimality of $|F|$ and hence our proof is complete. □

Lemma 13.12. *Given a instance (G, k) of $\mathcal{H}$-*Free Edge Editing *($\mathcal{H}$-*Free Edge Deletion*), an optimal solution F and its layer decomposition $F_1, \ldots, F_\ell$ it holds that either*

(i) $|V(G)| < k + 2(\Delta(G)+1) \cdot n_{\mathcal{H}}$ *or*

(ii) $\ell \leq 1 + \log_{\frac{\Delta(G)+1}{\Delta(G)}} |F|$.

Proof. Assume that *(i)* does not hold and hence that $|V(G)| \geq k+2(\Delta(G)+1)\cdot n_{\mathcal{H}}$. It follows immediately that *(ii)* in Lemma 13.11 applies.

$$\begin{aligned}
|F| &= |R_1| = |F_1| + |R_2| \\
&\geq \frac{|R_2|}{\Delta(G)} + |R_2| = \frac{\Delta(G)+1}{\Delta(G)} \cdot |R_2| = \frac{\Delta(G)+1}{\Delta(G)} \cdot (|F_2| + |R_3|) \\
&\geq \cdots \geq \left(\frac{\Delta(G)+1}{\Delta(G)}\right)^{\ell-1} \cdot |R_\ell| \\
&= \left(\frac{\Delta(G)+1}{\Delta(G)}\right)^{\ell-1} \cdot |F_\ell| \\
&\geq \left(\frac{\Delta(G)+1}{\Delta(G)}\right)^{\ell-1}
\end{aligned}$$

This gives that $\ell \leq 1 + \log_{\frac{\Delta(G)+1}{\Delta(G)}} |F|$ and hence the argument is complete. □

13.3 Obtaining the kernel

We now have all the necessary tools for providing the kernels. We reduce the graph to a ball of small radius around any obstruction core Z and by this obtain a kernelized instance. Both the size bounds and the correctness of the reduction rule follows by combining the tools developed during the section. For readability, we denote $\mathrm{diam}_{\mathcal{H}}$ by D and $\Delta(G)$ by Δ for the remainder of this section.

Rule 13.1. *Let (G,k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion). If $|V(G)| < k + 2(\Delta+1)\cdot n_{\mathcal{H}}$ we output (G,k). Otherwise, we find an obstruction core Z of G and return $(G[B(Z,r)],k)$ where $r = D \cdot (1 + \log_{\frac{\Delta+1}{\Delta}} k)$.*

The rule can clearly be applied in polynomial time; From Observation 13.3, we find Z in polynomial time and then it only remains to compute a breadth-first search from Z to depth r. This is the only rule and it is applied once. Thus the returned kernelized instance is either (G,k) or $(G[B(Z,r)],k)$, which yields a proper kernel. We proceed now to prove that the rule is sound (Lemma 13.13) and that the kernel indeed is a polynomial kernel (Lemma 13.14) before we wrap up this section with Theorem 32.

Lemma 13.13. *Let (G,k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion) and (G',k) the instance obtained when applying Rule 13.1 to (G,k). Then (G,k) is a yes-instance if and only if (G',k) is a yes-instance.*

Proof. It follows immediately from G' being an induced subgraph of G that if (G,k) is a yes-instance, then so is (G',k). For the other direction, let (G',k) be a yes-instance and let Z be the obstruction core of G obtained when applying Rule 13.1. Let F be an optimal solution of (G',k) and construct the layer decomposition

$F_1, \dots, F_\ell$ and the core extensions Z_i with respect to Z and F in G. It follows from Lemmata 13.8 and 13.12 that Z^+ is contained in G'. We observe that by construction Z is a valid obstruction core in G' and $F_1, \dots, F_\ell$ a valid layer decomposition of F in G'. It follows from Lemma 13.7 that $F_1, \dots, F_\ell$ forms a partition of F. By assumption it holds that $G'[Z^+] \oplus F$ is $\mathcal{H}$-free. And since $G'[Z^+] = G[Z^+]$ it follows by Lemma 13.6 that $G \oplus F$ is $\mathcal{H}$-free.

□

The following lemma shows that the kernel given by the rule, is actually a polynomial kernel.

Lemma 13.14. *Let (G, k) be an instance of $\mathcal{H}$-Free Edge Editing ($\mathcal{H}$-Free Edge Deletion) and (G', k) the instance obtained when applying Rule 13.1 to (G, k). Then the number of vertices in G' is at most*

$$6n_{\mathcal{H}}^2|\mathcal{H}^\star|\Delta^{D+2}k^{1+D(\Delta+1)\ln(\Delta)}.$$

Proof. If $|V(G)| < k + 2(\Delta+1) \cdot n_{\mathcal{H}} \leq 6n_{\mathcal{H}}\Delta k$, the result follows immediately. Otherwise, by Observation 13.3 we can assume that $|Z| \leq n_{\mathcal{H}}^2|\mathcal{H}^\star|((\Delta+1)+2k+1)$. If $\Delta = 1$ it holds that $|B(Z,r)| \leq 2|Z| \leq 2n_{\mathcal{H}}^2|\mathcal{H}^\star|(2k+3) \leq 6n_{\mathcal{H}}^2|\mathcal{H}^\star|k$. From now on, we assume $\Delta \geq 2$ and we will use the bound $|Z| \leq 6n_{\mathcal{H}}^2|\mathcal{H}^\star|\Delta k$. Observe that $|B(Z,r)| \leq |Z|\Delta^{r+1}$ and hence it remains to bound Δ^{r+1}. First, we obtain the following

$$\begin{aligned}
\Delta^{r+1} &= \Delta^{1+D(1+\log_{\frac{\Delta+1}{\Delta}} k)} \\
&= \Delta^{D+1}\Delta^{D\cdot\log_{\frac{\Delta+1}{\Delta}} k} \\
&= \Delta^{D+1}\Delta^{D\cdot\log_\Delta k/\log_\Delta(\frac{\Delta+1}{\Delta})} \\
&= \Delta^{D+1}k^{D/\log_\Delta(\frac{\Delta+1}{\Delta})}.
\end{aligned}$$

Since $\Delta \geq 2$ we can apply Lemma 12.4 to obtain $|Z| \leq \Delta^{D+1}k^{D(\Delta+1)\ln(\Delta)}$. It follows immediately that

$$\begin{aligned}
|B(Z,r)| &\leq |Z|\Delta^{r+1} \\
&\leq 6n_{\mathcal{H}}^2|\mathcal{H}^\star|\Delta k\Delta^{D+1}k^{D(\Delta+1)\ln(\Delta)} \\
&= 6n_{\mathcal{H}}^2|\mathcal{H}^\star|\Delta^{D+2}k^{1+D(\Delta+1)\ln(\Delta)}.
\end{aligned}$$

□

Theorem 32. *For every $\mathcal{H}$ both $\mathcal{H}$-Free Edge Deletion and $\mathcal{H}$-Free Edge Editing admit kernels with at most $k^{O(\Delta\log(\Delta))}$ vertices on graphs of maximum degree Δ.*

Proof. The result follows immediately from Lemmata 13.13 and 13.14. □

Chapter 14

Hardness for completion

This chapter is devoted to proving that for the *completion operation*, there is no hope for a general polynomial kernelization result on bounded degree graphs, even when the target graph class is characterized by a finite sets of forbidden connected graphs. To this aim, we give a finite set $\mathcal{H}$ of connected graphs for which $\mathcal{H}$-Free Edge Completion does not admit a polynomial kernel unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$. The result here is purely a classification result, as it will be clear that the size of $\mathcal{H}$—albeit finite—is quite large. We will throughout this chapter refer to $\mathcal{H}$-Free Edge Completion with the intended meaning that $\mathcal{H}$ is a finite set of connected forbidden induced subgraphs determined later.

We use OR-cross-composition and reduce from Cubic Planar Vertex Cover. In this problem we are given a planar cubic graph G (i.e., $\delta(G) = \Delta(G) = 3$), and an integer k' and asked to find a vertex cover of size k', that is, decide whether $\mathrm{vc}(G) \leq k'$.

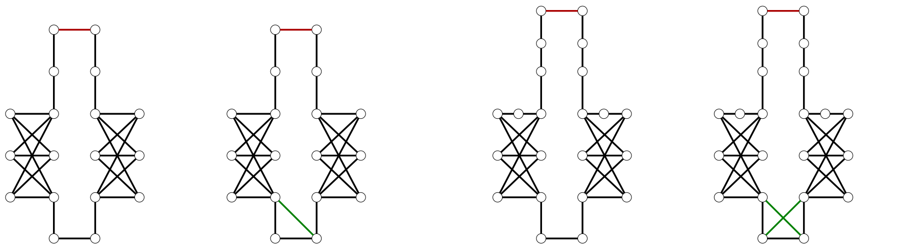

(a) Selector tree forbidden
(b) Selector tree ok
(c) Duplicator forbidden
(d) Selector tree forbidden

Figure 14.1: The selector tree gadget (on the left) and the duplicator gadget (on the right). The former has two possible completions, either to the left, or to the right. The latter (the duplicator) has only one legal completion, adding both edges.

Outline

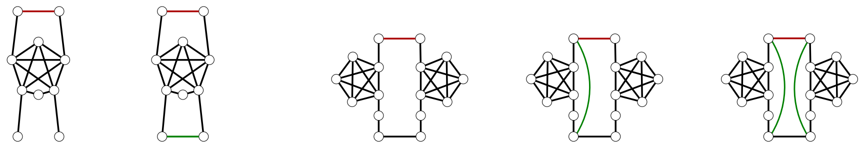
We will define four *base graphs*. These graphs will be our main building blocks, and they are all non-planar, in the way that they contain, as a minor, a K_5 or a $K_{3,3}$. This means that we do not have to worry that they will appear in our problem instance to Cubic Planar Vertex Cover, whose graphs are all planar. The base graphs will all be added to $\mathcal{H}$ together with all their supergraphs, except for a few supergraphs. These supergraphs will act as selectors, or as propagators or duplicators when only one completion is allowed. The size of $\mathcal{H}$ will be bounded by the sum of all possible completions for each of the base graphs. The base graphs are the following:

(i) Selector tree (Figure 14.1a, one of the allowed completions is depicted in Figure 14.1b)

(ii) Duplicator gadget (Figure 14.1c, the unique allowed completion is depicted in Figure 14.1d)

(iii) Propagator gadget (Figure 14.2a, the unique allowed completion is depicted in Figure 14.2b)

(iv) Vertex selector gadget (Figure 14.2c, two of the allowed completions are depicted in Figures 14.2d and 14.2e)



(a) Propagator forbidden (b) Propagator ok (c) Vertex selector forbidden (d) Vertex selector ok (e) Vertex selector ok

Figure 14.2: The propagator gadget (on the left) and the vertex selector gadget (on the right). The propagator has as purpose to simply separate two gadgets. The vertex selector gadget is the final gadget used in the vertex cover reduction. Three completion of the latter gadget is allowed, shortcutting either the left or the right edge, or shortcutting both, corresponding to adding one or two endpoints of an edge to the vertex cover.

Let $U_{\texttt{SEL}}$ be the selector tree gadget as depicted in Figure 14.1a. We define $U_{\texttt{SEL}}{\uparrow}$ to be the set of all supergraphs of $U_{\texttt{SEL}}$ *except* the two graphs isomorphic to the completed selector tree, depicted in Figure 14.1b. For the three other base graphs, we do the same thing; Let $U_{\texttt{DUP}}$ be the duplicator gadget and $U_{\texttt{DUP}}{\uparrow}$ all supergraphs except the one depicted in Figure 14.1d. Finally we construct $U_{\texttt{PRO}}{\uparrow}$ and $U_{\texttt{VER}}{\uparrow}$ for the propagator and vertex selector gadgets. These sets together comprise $\mathcal{H}$: Let

$$\mathcal{H} = U_{\texttt{SEL}}{\uparrow} \cup U_{\texttt{DUP}}{\uparrow} \cup U_{\texttt{PRO}}{\uparrow} \cup U_{\texttt{VER}}{\uparrow} .$$

The OR-cross-composition will take as input $t = 2^r$ instances of the NP-complete problem Cubic Planar Vertex Cover (recall Proposition 12.3) where we may assume they are on the form

$$(G_1, k'), (G_2, k'), \ldots, (G_t, k'),$$

that is the parameter is the same across all instances, in addition to that $|V(G_i)| = |V(G_j)|$ for all $i, j \leq t$, hence also $|E(G_i)| = |E(G_j)|$. The restriction that t is a power of two is not necessary—we could attach any *cul-de-sac* gadget, e.g., the propagator with an extra edge in the K_5, on the end of the selector tree for every index $i > t$—but makes the proofs simpler to conceptualize.

We reduce to a single instance of $\mathcal{H}$-Free Edge Completion (G, k) with budget $k = \log t + k' + 3\hat{m} - 1$, where $\hat{m}$ is the smallest power of two that is at least $m = |E(G_i)|$. The graph G will have a single induced obstruction. The budget will be tight with $\log t$ edges forced in the selector gadget, selecting instance (G_i, k'), then $3\hat{m} - 1$ edges will be used to construct the graph copy $\hat{G}_i$. The remaining part of the budget is the k' edges corresponding to a vertex cover of G_i. Hence $k = (|G_i| + \log t)^{O(1)}$.

14.1 Selector Tree

In this section we describe how to construct a selector tree that will be used in the OR-cross-composition. For now, let us fix t to be the number of instances provided as input to the reduction, and let $G_1, \ldots, G_t$ be the cubic planar graphs. Furthermore, denote by k' the budget for the input instance. We may assume that t is a power of two. Let U_{SEL} be the graph depicted in Figure 14.1a. Denote by v_1 and v_2 the two top vertices of U_{SEL}. Denote the vertices on the bottom on the path (including endpoints) between the bicliques with a_1, b_2, a_2, b_1, in order.

Let T be a complete binary tree on $t/2 = 2^{r-1}$ leaves. Replace each leaf node ℓ_i with U_i, a copy of U_{SEL}. Note that this is one U_i for each two instances, i.e., we have twice as many instances as we have leaves. For each two siblings U_1 and U_2 in the tree, replace its parent v by U_v and identify v_1, v_2 of U_1 with a_1 and a_2, and equivalently, identify v_1, v_2 of U_2 with b_1 and b_2. The tree construction T is depicted in Figure 14.3. Finally, for every U_i, except the one corresponding to the root, we remove the edge v_1v_2.

We define $h(T)$ to be the height of T, i.e., $h(T) = \log(t)$ (see Figure 14.3).

Lemma 14.1. *The constructed selector tree has degree bounded by* 5.

Proof. It is easy to verify that the vertices with maximum degree in T are the vertices corresponding to a_1 and b_1, i.e., the vertices of $K_{3,3}$ identified with top vertices in a child. These have degree 5. □

Definition 14.2. A solution F of (T, k) is said to *select* i if the $\lfloor i/2 \rfloor$th leaf has its corresponding a_1a_2 (or b_1b_2 if i is even) in F.

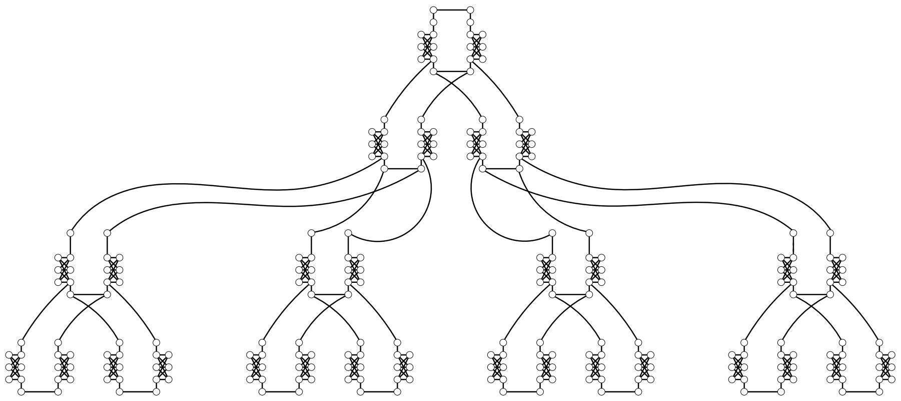


Figure 14.3: The selector tree T is a complete binary tree with nodes replaced by copies of $U_{\texttt{SEL}}$. For an input t (e.g. $t = 16$ in this case), we construct the complete binary tree with $t/2$ leaves. The tree has height $h(T) = \log(t) = 4$ and needs a budget of 4.

Lemma 14.3. *Given an instance* $(T, h(T))$ *of* $\mathcal{H}$-Free Edge Completion*, the following holds:* $(T, h(T))$ *is a yes-instance, and* $(T, h(T) - 1)$ *is a no-instance.*

Proof. By strong induction on $h(T)$. Let $h(T) = 1$, i.e., we have one copy of $U_{\texttt{SEL}}$. We know that $U_{\texttt{SEL}}$ is forbidden, and that U can be eliminated by adding one edge. This concludes the base case.

Suppose the statement is true for all $h(T') \leq n-1$. Construct T with $h(T) = n$ by taking two copies of T' of height $h(T) - 1$ and attaching them to a new root U_r. Clearly, again, we can take any solution of one of the subtrees and add a corresponding edge for U_r selecting the subtree with the solution. Suppose now that T had a solution of size $h(T) - 1 = h(T')$. We know that the root U_r must have one edge. But then $h(T')$ has a solution of size $h(T') - 1$, contradicting the induction hypothesis. See Figure 14.3. □

With the tightness proven, we are ready to state the main lemma of the selector tree:

Lemma 14.4 (Selector lemma)**.** *Given an instance* $(T, h(T))$ *of* $\mathcal{H}$-Free Edge Completion*, any solution* F *of size at most* $h(T)$ *selects exactly one* i*.*

Proof. As witnessed in the proof above, every solution is on the form of a path from root to a leaf. For a budget of $h(T)$ edges, we can select exactly one leaf, which is the i in the statement. □

From Lemmata 14.3 and 14.4 we get the interface we wanted from this section; a tree T constructed as above, together with the corresponding budget, must be handled in one specific way, by adding one edge in all constructed $U_{\texttt{SEL}}$s in a path from a leaf to the root.

Corollary 14.5. *A budget of* $\log(t) = h(T)$ *is sufficient and necessary to eliminate all obstructions from* T.

14.2 Instance Activator

In this section, we create for a given m, an instance activator M which consists of one propagator on the top, $\hat{m} - 1$ duplicators, and then $\hat{m}$ new propagators. The first propagator will be the interface to the *instance selector tree* in the previous section, attached to an appropriate place in the selector tree. In particular, the two top vertices will be identified with a_1 and a_2 (or b_1 and b_2) in a leaf of the selector tree. The $\hat{m}$ bottom propagators will add edges in a single instance that is to be activated by the choice of i.

We construct the graph M_i for every $i \leq t$ by creating a binary tree of duplicators, copying the choice of i from the selector tree without increasing the degree of the constructed graph above 5. A propagator is positioned on top of the tree and attached to the selector tree. Finally, to every leaf of the binary tree we also attach a propagator. The construction M_i is displayed in Figure 14.4.

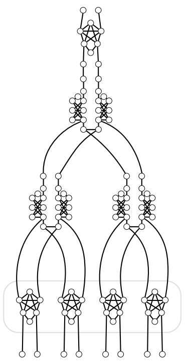


Figure 14.4: Instance activator M_i for $\hat{m} = 4$, with budget $b = 11 = 3\hat{m} - 1$.

Lemma 14.6. *When the ith propagator gadget is activated, all* $\hat{m}$ *edges out of* M_i *are activated and this uses budget* $2(\hat{m} - 1) + 1 + \hat{m} = 3\hat{m} - 1$, *not counting the activation edge on the top.*

Proof. The propagator gadget (Figure 14.2a) has only one possible completion (Figure 14.2b), and the same holds for the duplicator gadget (Figure 14.1c, the completion is depicted in Figure 14.1d). This implies that once the top propagator is activated, all edges in every duplicator will be added, and finally, again, all the $\hat{m}$ propagators will be activated. This sums up to the claimed budget. □

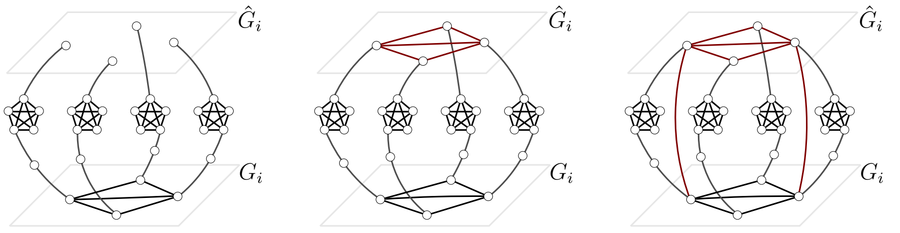
14.3 Cubic Planar Vertex Cover reduction

With the instance activator M_i and hence $\hat{m}$ edges that can be forced inside the instance at hand, we are ready to construct the actual vertex cover reduction. Let (G_i, k') be an instance of Cubic Planar Vertex Cover. We are to construct an instance P_i such that, when activated, any successful completion corresponds to a vertex cover in G_i. This is done as follows: First, we add G_i to P_i. Then, for every vertex $v \in V(G_i)$ we add a vertex $\hat{v}$ and a K_5 to P_i. Finally, we connect v to the K_5 via a path of length 2 and $\hat{v}$ directly to a different vertex of the constructed clique. See Figure 14.5a.

For every edge $uv \in E(G_i)$ we select a unique bottom propagator of the instance activator M_i and identify the two vertices $\hat{u}$ and $\hat{v}$ with the two bottom vertices of the propagator. We say that the instance G_i is activated if the selector tree selected i and hence the instance activator M_i was completed. Furthermore, we say that P_i is activated, or completed into $\hat{P}_i$, if for for every edge uv in $E(G_i)$ it holds that the edge $\hat{u}\hat{v}$ was added by the solution.

Lemma 14.7. *When G_i is activated, P_i must be completed into $\hat{P}_i$ in any minimal completion.*

Proof. By Lemma 14.4 we know that the selector tree selects exactly one i. When the i'th instance activator is activated, it follows by Lemma 14.6 that all output edges of the corresponding M_i was completed by the solution. By construction, P_i is then activated. □



(a) The planar graph gadget P_i before the edges of G_i has been added to $\hat{G}_i$.

(b) The completed graph, with edges added to $\hat{G}_i$. This depicts $\hat{P}_i$.

(c) The completed graph, with edges added to $\hat{G}_i$ and the vertex cover. This depicts $\hat{P}_i$ with vc.

Figure 14.5: The vertex cover gadget. On the left, P_i, the graph before the edges of $\hat{G}_i$ have been added. In the middle, the graph $\hat{P}_i$, when the edges of $\hat{G}_i$ have been added. On the right, the completed $\hat{P}_i$ with the vertex cover solution.

Lemma 14.8. *For (G_i, k) of* Cubic Planar Vertex Cover *the constructed graph P_i has $\Delta(P_i) \leq 5$.*

Proof. Every vertex of G_i is of degree three and the corresponding vertices in P_i is of degree 4. The vertices between $V(G_i)$ and the K_5's are of degree 2 and the vertices in the K_5 of degree 5. Last, we consider the copies of the vertices of $V(G_i)$. We observe that every $\hat{v}$ has one neighbor in P_i and one neighbor in M_i. The result follows immediately. □

Lemma 14.9. *P_i is $\mathcal{H}$-free.*

Proof. Since every forbidden structure contains as a subgraph either a K_5 or a $K_{3,3}$, we know that neither G_i nor $\hat{G}_i$ contains an obstruction since these graphs are both planar. Hence the obstructions must use the newly introduced vertices. Observe, that the only obstructions that contain K_5's with only two neighbors is the vertex selector and some of its super graphs. However, in these obstructions all vertices are contained in at least one cycle, which is not true for the vertices on the paths between $V(G_i)$ and the K_5's in P_i. □

Lemma 14.10. *$\hat{P}_i$ has exactly one obstruction per edge in G_i.*

Proof. By the same argument as above, namely that each K_5 have degree at most two to the outside, we know that the obstructions appearing in $\hat{P}_i$ must be instances of the vertex selector. Furthermore, it must utilize exactly two of the K_5's inserted above two vertices u and v in $V(G_i)$. Last, for the paths between the K_5's to be of the correct length, both uv and $\hat{u}\hat{v}$ must be connected in $\hat{P}_i$. □

Lemma 14.11. *$(\hat{P}_i, k')$ is yes for $\mathcal{H}$-Free Edge Completion if and only of G_i has a vertex cover of size k'.*

Proof. The first thing we want to observe is that for any obstruction using vu and $\hat{v}\hat{u}$, adding an edge $v\hat{v}$ or $u\hat{u}$ is sufficient to eliminate the obstruction, and furthermore does not create any new obstructions. The latter holds since any obstruction must be on the form vu and $\hat{v}\hat{u}$ and K_5s between v and $\hat{v}$, and u and $\hat{u}$. For the former statement, the following, even stronger statement holds: For any vertex $v \in G_i$, adding the edge $v\hat{v}$ will eliminate every obstruction in which v and $\hat{v}$ is.

With those two observations, we are ready to prove the lemma statement. In the forwards direction, let $(\hat{P}_i, k')$ be a yes-instance with F a solution. Since there is one obstruction per edge, and any solution will be of the form $\bigcup_{v \in V(F) \cap V(G_i)} \{v\hat{v}\}$, we will show that $\bigcup_{v \in V(F) \cap V(G_i)} \{v\}$ is a vertex cover of G_i. Suppose there is an edge uv which has not been covered by a vertex. Then uv and $\hat{u}\hat{v}$ together with the K_5's form an obstruction. This contradicts the assumption that F was a solution and concludes the forwards direction.

In the reverse direction, let C be a vertex cover of G_i of size at most k'. For each vertex $v \in C$, add the edge between v and $\hat{v}$, its corresponding vertex in $\hat{G}_i$. We claim that this graph is $\mathcal{H}$-free. Suppose there is still an obstruction. This obstruction has form uv and $\hat{u}\hat{v}$ with K_5's between. There are three allowed completions of the obstruction, namely of $\{u\hat{u}\}$, $\{v\hat{v}\}$, and $\{u\hat{u}, v\hat{v}\}$. However, none of them has been added by the construction, hence $u \notin C$ and $v \notin C$. It follows that C is not a vertex cover since uv is an edge. □

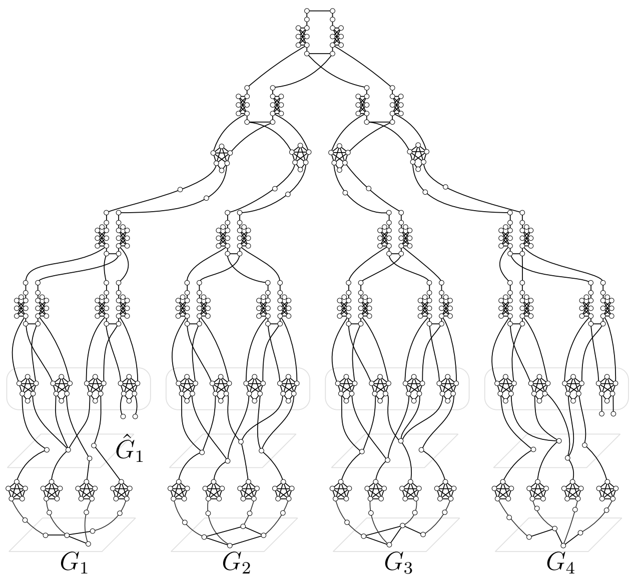


Figure 14.6: A complete reduction G for four graphs G_1, G_2, G_3, G_4. However, in these examples, the graphs are subcubic, and does not have the same number of edges. The intuition for the gadget is that if $k' = 1$ and G_1 has a vertex cover of size 1, then we can chose to complete the edges such that we solve G_1.

14.3.1 Wrapping up the cross-composition

We now combine the gadgets from the previous sections, the selector tree from Section 14.1, the instance activator from Section 14.2 and the vertex cover reduction from the previous section, Section 14.3. The goal is to have the tree T activate one instance activator M_i, which in turn adds all the edges of $\hat{G}_i$ in a vertex cover reduction P_i, thus constructing $\hat{P}_i$. This finally creates one induced copy of the forbidden *vertex selector gadget* (Figure 14.2c) for each edge of G_i.

Lemma 14.12. *Let G be the constructed instance above for input $(G_1, \dots, G_t, k)$. Then $\Delta(G) = 5$.*

Proof. First, observe that any vertex inside an instance activator is of degree at most 5. The remaining vertices are covered by Lemmata 14.1 and 14.8. □

Lemma 14.13. *If (G_i, k') is a yes-instance, then F_T (the solution selecting i) together with the edges F_M of the instance activator M_i and a solution F_i for G_i is a solution of G.*

Proof. Invoke Lemmata 14.4, 14.6 and 14.11. □

Lemma 14.14. *Let (G, k) be a yes-instance constructed from the reduction with $k = \log t + k' + 3\hat{m} - 1$ and F a minimal solution of size at most k. Then there is an i such that:*

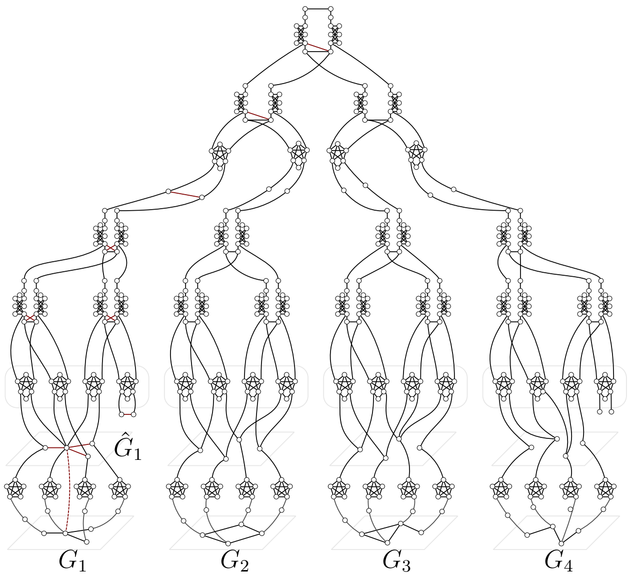


Figure 14.7: A completed reduction G where we have selected G_1 to be the instance we want to solve, and chosen the universal vertex of G_1 as our vertex cover. The budget is exactly $k' = \log t + 2m + k$.

- $F = F_i \cup F_T \cup F_M$ *and*
- G_i, k' *is a yes-instance of* Vertex Cover

Proof. Corollary 14.5 showed that it is necessary and sufficient with a budget of $\log t$ to eliminate all obstructions from T. Furthermore, Lemma 14.4 showed that there is exactly one i such that G_i is activated. When G_i has been activated, we need, by Lemma 14.6 to add $3\hat{m} - 1$ edges to M_i. Finally, by Lemma 14.7 we are left with a completed $\hat{P}_i$, which by Lemma 14.11 is completable using at most k' edges if and only if G_i is a yes-instance for Cubic Planar Vertex Cover. □

From the discussions in this section, it is clear that given a set of $t = 2^r$ many graphs $G_1, \ldots, G_t$ and a natural number k', we can construct in polynomial time an instance (G, k) which is a yes-instance for $\mathcal{H}$-Free Edge Completion if and only if there is an $i \leq t$ such that (G_i, k') is a yes-instance of Cubic Planar Vertex Cover. This is the or-equivalence part of the definition of or-cross-composition (Definition 1.10). The second part of the definition is that the parameter value must be polynomially bounded, and this is clear since $k \leq (\hat{m} + k')^{O(1)} + \log t \leq |G_i|^{O(1)} + \log t$. That is, (G, k) is subject to or-cross-composition. As we observed in Lemma 14.12, $\Delta(G) = 5$, and hence applying Theorem 4 yields the following theorem.

Theorem 33. *There exists a finite set $\mathcal{H}$ such that $\mathcal{H}$-*Free Edge Completion *does not admit a polynomial kernel, even on input graphs of maximum degree* 5*, unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

Chapter 15

Hardness for degenerate graphs

In this chapter, we prove that, unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$, there is no polynomial kernel for C_{11}-Free Edge Deletion even on 2-degenerate graphs. We use or-cross-composition and reduce from Cubic Planar Vertex Cover. We first give an equivalent annotated version of the problem and after this we only consider instances of this problem. The annotated version has an extra input, which is a subset R of the edge set of the input graph, such that the edges in R are forbidden to be a part of any valid solution.

15.1 Annotated version

We now give an annotated version of the problem at hand, allowing us to make certain edges undeletable. The input to this problem will be a triple, (G, k, R), where G and k are as before, and R is the set of edges we are not allowed to delete. We also note that this is not a parameterized problem of the form $\mathcal{G} \times \mathbb{N}$, but this is only a technicality.

Annotated C_{11}-Free Edge Deletion parameterized by $G, k, R \subseteq E(G)$

Input:	k
Question:	Is there an $F \subseteq E(G)$ of size at most k with $F \cap R = \emptyset$ such that $G - F$ is C_{11}-free?

When we say that F is a solution of an instance (G, k, R) of Annotated C_{11}-Free Edge Deletion, it is implicit that $|F| \leq k$ and hence may not be explicitly mentioned. The following construction gives a parameterized reduction from Annotated C_{11}-Free Edge Deletion to C_{11}-Free Edge Deletion. The construction is depicted in Figure 15.1.

Construction

Given an instance (G, k, R) of Annotated C_{11}-Free Edge Deletion, we output an instance (G_R, k) of C_{11}-Free Edge Deletion as follows: For every edge $\{u, v\} \in R$, add k vertices adjacent to only u and v, and k paths of eight vertices with the endpoints adjacent to u and v, respectively.

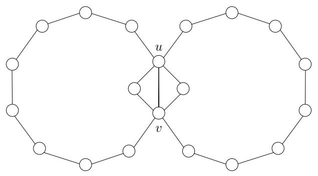


Figure 15.1: The edge $\{u, v\}$ is in R. The edge is made a chord to k number of edge disjoint C_{11}s. Only two such C_{11}s are displayed in the figure.

The input graph of the construction is an induced subgraph of the output graph of the construction. Hence, we identify every vertex and edge of the input graph with the corresponding vertex and edge of the output graph. The following observation shows that the newly introduced vertices cannot be a part of any induced C_{11} in any subgraph of G_R if all edges in R are part of the subgraph.

Observation 15.1. *Let (G, k, R) be an instance of* Annotated C_{11}-Free Edge Deletion. *Let (G_R, k) be obtained by applying the construction described above on (G, k, R). Let $E' \subseteq E(G_R)$ be such that $E' \cap R = \emptyset$. Let x be any vertex introduced in the construction. Then, x is not a part of any induced C_{11} in $G_R - E'$.*

Proof. For a contradiction, assume that there is an induced C_{11} with a vertex set C in $G_R - E'$ such that $x \in C$. Let $\{u, v\}$ be the edge in G such that x is a part of a C_{11}, with the chord $\{u, v\}$, introduced in the construction. Since $x \in C$ and C induces a C_{11} in $G_R - E'$, C has a subset C' which contains all the vertices in the path containing u, v and x (with the end points u and v). Since $\{u, v\} \in R$, by assumption, $\{u, v\} \notin E'$. Hence, C' induces either a C_3 or a C_{10} in $G_R - E'$. Hence, C does not induce a C_{11} in $G_R - S$. □

Lemma 15.2. Annotated C_{11}-Free Edge Deletion *is parameter equivalent to* C_{11}-Free Edge Deletion.

Proof. We need to give a strong parameterized reductions from C_{11}-Free Edge Deletion to Annotated C_{11}-Free Edge Deletion and Annotated C_{11}-Free Edge Deletion to C_{11}-Free Edge Deletion. In the reverse direction, let (G, k) be an instance of C_{11}-Free Edge Deletion. It is straight forward to verify that (G, k) is a yes-instance of C_{11}-Free Edge Deletion if and only if $(G, k, \emptyset)$ is a yes-instance of Annotated C_{11}-Free Edge Deletion.

In the forward direction, let (G, k, R) be an instance of Annotated C_{11}-Free Edge Deletion. Apply the construction described above on (G, k, R) to obtain (G_R, k). Clearly, the construction can be applied in polynomial time. We need to prove that (G, k, R) is a yes-instance of Annotated C_{11}-Free Edge Deletion if and only if (G_R, k) is a yes-instance of C_{11}-Free Edge Deletion. Let (G, k, R) be a yes-instance with a minimum solution F. We claim that $G_R - F$ is C_{11}-free.

For a contradiction, assume that there exists an induced C_{11} with a vertex set C in $G_R - F$. Since $G - F$ is an induced subgraph of $G_R - F$, an induced C_{11} in $G_R - S$ must have a vertex x introduced in the construction, which is a contradiction by Observation 15.1. Conversely, assume that (G_R, k) is a yes-instance with a minimum solution F_R. Since G is an induced subgraph of G_R and $G_R - F_R$ is C_{11}-free, $G - F_R$ is C_{11}-free. □

The following lemma shows that the construction described above does not alter the degeneracy of the graph.

Lemma 15.3. *Let* (G, k, R) *be an instance of* Annotated C_{11}-Free Edge Deletion *such that* $\text{dgy}(G) \geq 2$. *Let* (G_R, k) *be obtained by applying the construction described above on* (G, k, R). *Then,* $\text{dgy}(G_R) = \text{dgy}(G)$.

Proof. Let $\text{dgy}(G) = d \geq 2$. All the vertices in G_R which are newly introduced by the construction have degree two. Deleting all of them from G_R gives G. Therefore, since $\text{dgy}(G) \geq 2$, $\text{dgy}(G_R) = \text{dgy}(G)$. □

Lemma 15.2 and Lemma 15.3 make sure that it is enough to prove the kernelization lower bound of Annotated C_{11}-Free Edge Deletion on degenerate graphs to prove the same for C_{11}-Free Edge Deletion. Corollary 15.4 directly follows from both these lemmas.

Corollary 15.4. *For every* $d \geq 2$, C_{11}-Free Edge Deletion *admits polynomial kernelization on* d*-degenerate graphs if and only if* Annotated C_{11}-Free Edge Deletion *admits polynomial kernelization on* d*-degenerate graphs.*

Proof. By Lemma 15.2, the two problems are parameter equivalent, and by Lemma 15.3, this holds even when restricting the input to degenerate graphs. Hence, it follows immediately that C_{11}-Free Edge Deletion admits a polynomial kernelization on d-degenerate graphs if and only if Annotated C_{11}-Free Edge Deletion admits a polynomial kernelization on d-degenerate graphs. □

We note that we work solely with Annotated C_{11}-Free Edge Deletion from this point on.

15.2 Selector tree

In this section, we describe how to construct a selector tree that will be used in the cross composition later.

Consider a complete binary tree T_r with $t = 2^{r-1}$ leaves, where r is the height of T. For every non-leaf vertex v in T_r arbitrarily identify a child of v as the *first child* of v and the other child as the *second child* of v. The basic building block of the selector tree is a C_{11}. Consider a labeling of vertices in a C_{11} from v_1 to v_{11} such that v_i is adjacent to $v_{(i \mod 11)+1}$, for $1 \leq i \leq 11$. Replace each vertex in T_r with a labeled C_{11}. The C_{11} replacing the root vertex is the *root*-C_{11} and those C_{11}s replacing the leaf vertices are *leaf*-C_{11}s. Let u be any vertex in T_r. Let the C_{11} introduced by replacing u has the vertex set U. We use the notations C_u and C_U analogously to denote this C_{11}. Let u be a non-leaf vertex in T_r. Let U be the labeled vertex set of C_{11} replaced for u. Let u_1 and u_2 be the first and second child of u respectively in T_r. Let U_1 and U_2 be the vertex sets of C_{11}s replaced for u_1 and u_2 respectively. Then C_U is called the *parent-*C_{11} of both C_{U_1} and C_{U_2}. Similarly, C_{U_1} and C_{U_2} are called *first child-*C_{11} and *second child-*C_{11} respectively of C_U. How C_{U_1} and C_{U_2} are glued to C_U is described below:

- Identify v_1 and v_4 of the U_1 with v_7 and v_6 of U respectively.
- Identify v_1 and v_4 of the U_2 with v_{10} and v_9 of U respectively.

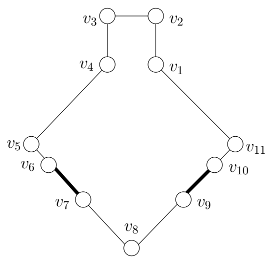


Figure 15.2: C_{11} - the basic building block of the selector tree and a valid labeling of its vertices. The edges $\{v_6, v_7\}$ and $\{v_9, v_{10}\}$ are the first deletable edge and the second deletable edge of the C_{11} respectively.

Let the graph obtained from T_r by the process described above be T. We call T a *selector tree.* For every C_{11} C, we call the edges $\{v_6, v_7\}$ and $\{v_9, v_{10}\}$ as the *first deletable edge* and the *second deletable edge* respectively of C. Let R_T be the set of all edges in T other than the first and second deletable edges of every C_{11}. We call every edge in R_T *non-deletable* and every edge in $E(T) \setminus R_T$ *deletable.* Root-C_{11} is at level 1 and the leaf-C_{11}s are at level r in T. Let A be a parent-C_{11} of a C_{11} B. Then, if B is at level ℓ, then A is at level $\ell - 1$. Figure 15.3 depicts the structure of T when $r = 3$.

Lemma 15.5. *Let T be a selector tree. Then, T is 2-degenerate.*

Proof. Deleting all vertices with degree two from T gives a matching, constituted by the deletable edges. □

Consider a path $P = (u_1, u_2, \ldots, u_r)$ from a leaf u_1 to the root u_r of a complete binary tree T_r such that u_{i+1} is the parent of u_i for $1 \leq i \leq r - 1$. We call such a path as a *leaf-root* path of T_r. Based on P, we define two sets of edges E_1 and E_2 of the selector tree T. Let U_i denotes the set of vertices of the C_{11} introduced by replacing the vertex u_i, for $1 \leq i \leq r$. For every U_i, for $2 \leq i \leq r$, if u_{i-1} is the first child of u_i, then both E_1 and E_2 contain the first deletable edge of

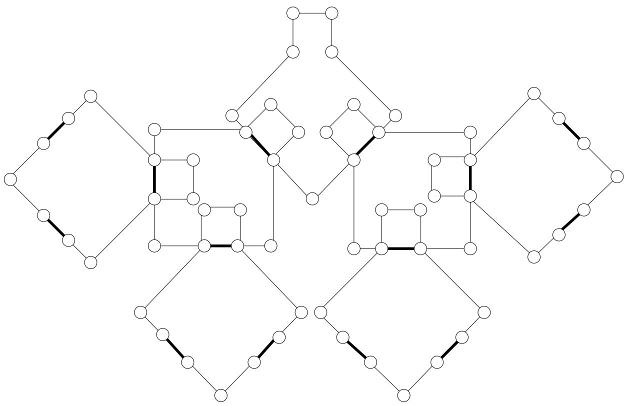


Figure 15.3: The selector tree T when $r = 3$. Every edge other than the bold edges is in R_T.

C_{U_i} and if u_{i-1} is the second child of u_i, then both E_1 and E_2 contain the second deletable edge of C_{U_i}. Additionally, E_1 contains the first deletable edge of C_{U_1} and E_2 contains the second deletable edge of C_{U_1}. We call E_1 and E_2 as the *first solution* and *second solution* of (T, r, R_T) based on P. Next, we prove that E_1 and E_2 are in fact solutions of (T, r, R_T).

Lemma 15.6. *Let T be a selector tree constructed from a complete binary tree T_r for some $r \geq 1$. Let $P = (u_1, u_2, \ldots, u_r)$ be a leaf-root path in T_r and let E_1 and E_2 are the first solution and second solution respectively of (T, r, R_T) based on P. Then, $|E_1| = |E_2| = r$ and $T - E_1$ and $T - E_2$ are C_{11}-free.*

Proof. Since both E_1 and E_2 have exactly one edge from the set of C_{11}s at every level of T, $|E_1| = |E_2| = r$. We prove that E_1 is a solution of (T, r, R_T) by induction on r. The same arguments apply for E_2. If $r = 1$, then T is a C_{11} and E_1 contains its first deletable edge. Hence, $T - E_1$ is C_{11}-free. Assume that the argument is true when a selector tree T is obtained from T_{r-1}. Now, let T be obtained from T_r. Without loss of generality, assume that u_1 is the first child of u_2. Delete all vertices with degree two in T of the leaf-C_{11}s to obtain T'. Clearly, T' is isomorphic to a selector tree obtained from T_{r-1} and $P' = (u_2, \ldots, u_r)$ is a leaf-root path in T_{r-1}. Let E_1' be the first solution of $(T', r-1, R_{T'})$ based on P'. Clearly, $E_1' \subseteq E_1$. By the induction hypothesis, $T' - E_1'$ is C_{11}-free. It is straight forward to verify that $T - E_1'$ has a single induced C_{11} formed by the vertex set U_1 of C_{u_1}. This single induced C_{11} is killed by the first deletable edge e (which is part of E_1) of C_{U_1}. Since deleting e does not create any induced C_{11} in T, $T - E_1$ is C_{11}-free. $\square$

We label the deletable edges in the leaf C_{11}s in T from e_1 to e_{2^r} in an arbitrary order. We call them as the *deletable leaf edges* of T.

Definition 15.7. Let T be a selector tree obtained from a complete binary tree T_r as described above for some $r \geq 1$. A subset of edges E' of T *selects an integer* i if deletable leaf edge e_i is in E'.

Lemma 15.8. *Let T be a selector tree constructed from T_r as described above for some $r \geq 1$, and let (T, r, R_T) be an instance of* Annotated C_{11}-Free Edge Deletion. *Then:*

(i) Let E' be a subset of edges of T such that $T - E'$ is C_{11}-free. Then E' selects an integer i, for some $i \in [1, 2^r]$.

(ii) For any integer $i \in [1, 2^r]$, there exists a solution E' of (T, r, R_t) selecting the integer i.

(iii) Let E' be a subset of edges of T such that $T - E'$ is C_{11}-free. Then $|E'| \geq r$.

Proof. *(i)* Since the root-C_{11} is an induced C_{11} in T, $|E'| > 0$. For a contradiction, assume that E' does not contain any deletable leaf edge. Let ℓ be the maximum level such that E' has an edge e from a C_{11} at level ℓ. Without loss of generality assume that e is the first deletable edge of a C_{11} formed by a vertex set U. Let U' be the vertex set of the first child-C_{11} of C_U. Since $C_{U'}$ is in the level $\ell + 1$, E' does not contain any edge from it. Hence U' induces a C_{11} in $T - E'$, which is a contradiction.

(ii) Assume that the deletable leaf edge e_i is an edge in a leaf-C_{11} C_U with the vertex set U. Let u be the vertex in T_r such that u in T_r is replaced by C_U in T. Let P be a path from u to the root vertex in T_r. Let E_1 and E_2 be the first and second solution respectively of (T, r, R_T) based on P. Clearly, either E_1 or E_2 contains e_i. By Lemma 15.6, both E_1 and E_2 are solutions of (T, r, R_T). Hence either E_1 or E_2 selects i.

(iii) We apply induction on r. When $r = 1$, T is a C_{11} and hence $|E'| = 1$. Assume that the statement is true when the selector tree T' is obtained from T_{r-1}. Since T' is an induced subgraph of T, there must be a subset of edges E'' of T' such that $E'' \subseteq E'$ and $T' - E''$ is C_{11}-free. By the induction hypothesis, $|E''| \geq r - 1$. By (i), E'' selects an integer $g \in [1, 2^{r-1}]$ in T'. Let the deletable leaf edge e_g of T' be the edge in a leaf-C_{11} C_{U_g} with a vertex set U_g. Without loss of generality, assume that e_g is the first deletable edge of C_{U_g}. Then there is an induced C_{11} in $T - E''$ formed by the first child-C_{11} of C_{U_g}. Hence $|E'| \geq r - 1 + 1$. □

15.3 Cubic Planar Vertex Cover reduction

We introduce a *star gadget* for each instance of Cubic Planar Vertex Cover. A star gadget is a union of four edge disjoint P_3s where the P_3s share a single endpoint. The shared endpoint will be called the *root vertex* of the star gadget. The four vertices in the star gadget with degree one will be called the *leaf vertices*. The gadget is shown in Figure 15.4. Using this gadget, we give a polynomial time reduction from Cubic Planar Vertex Cover to Annotated C_{11}-Free

Edge Deletion. This will be used in the or-cross-composition to be described in the next section.

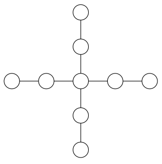


Figure 15.4: The star gadget. The vertex with degree four is called the root vertex and the vertices with degree one are called leaf vertices of the star gadget.

Reduction. Let (G, k) be an instance of Cubic Planar Vertex Cover. Obtain a four-coloring χ of G by using Proposition 2.1. We can safely assume that G has at least four vertices and χ uses exactly four colors. Let the four colors be $\{c_1, c_2, c_3, c_4\}$. The four leaf vertices of the star gadget are colored using the colors c_1, c_2, c_3 and c_4 respectively. We make a vertex in G and a leaf vertex in star gadget adjacent if both are of same color. We subdivide every edge in G by replacing it with a path of length five. Let the resultant graph be $G^\dagger$. Let $R^\dagger$ be the set of all edges in $G^\dagger$ except the edges whose one endpoint is in the star gadget and the other endpoint is in G. This completes the reduction.

The set of original G vertices in $G^\dagger$ is denoted by V_G. V_{two} denotes the set of vertices in $G^\dagger$ originated by subdividing the edges in G. All the vertices in $G^\dagger$ obtained by subdividing an edge $\{u, v\}$ (including u and v) is denoted by $S_{\{u,v\}}$. V_{star} denotes the vertices in the star gadget. For a vertex $v \in V_G$, we denote the vertex in V_{star} adjacent to v by $s(v)$. The vertices in the path of length four in $G^\dagger[V_{star}]$ with the colored end points x and y is denoted by $P_{\{x,y\}}$.

Lemma 15.9. *Let (G, k) be an instance of* Cubic Planar Vertex Cover. *Let $(G^\dagger, k, R^\dagger)$ be obtained by the reduction described above. A vertex set $C \subseteq V(G^\dagger)$ induces a C_{11} in $G^\dagger$ if and only if C is $S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$, where $\{u, v\}$ is an edge in G.*

Proof. Clearly, for any edge $\{u, v\}$ in G, $S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$ induces a C_{11} in $G^\dagger$. Since the girth of G is at least three and every edge in G is subdivided into a path of length five, $G^\dagger[V_G]$ has girth at least 15. Hence $G^\dagger[V_G]$ does not have an induced C_{11}. Since the star gadget has only nine vertices, $G^\dagger[V_{star}]$ does not have an induced C_{11}. Hence, every induced C_{11} in $G^\dagger$ must contain vertices from V_{star} and vertices from $V_G \cup V_{two}$. Let C be a set of vertices which induces a C_{11} in $G^\dagger$. It is straight forward to verify that C contains exactly six vertices $S_{\{u,v\}}$ from V_G, where $\{u, v\}$ is an edge in G and C must be of the form $S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$. □

Lemma 15.10. *Let (G, k) be an instance of* Cubic Planar Vertex Cover. *Let $(G^\dagger, k, R^\dagger)$ be obtained by using the reduction from (G, k) as described above. Then, $(G^\dagger, k, R^\dagger)$ is a yes-instance of* Annotated C_{11}-Free Edge Deletion *if and only if (G, k) is a yes-instance of* Cubic Planar Vertex Cover.

Proof. Let $(G^\dagger, k, R^\dagger)$ be a yes-instance of Annotated C_{11}-Free Edge Deletion. Let E' be a solution of size at most k of $(G^\dagger, k, R^\dagger)$. Let V' be the set of all vertices in V_G such that for every vertex $v' \in V'$, there is an edge $\{u, v'\}$ in E'. We claim that V' is a vertex cover of size at most k of (G). Since every edge in E' has one endpoint in V_G and the other endpoint in V_{star}, $|V'| \leq k$. By Lemma 15.9, for every edge $\{u, v\}$ in G, there is a C_{11} induced by $C = S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$. Since $\{u, s(u)\}$ and $\{v, s(v)\}$ are the only deletable edges, either of them must be in E'. Hence, for every edge $\{u, v\}$ in G, either u or v is in V'.

Conversely, let (G, k) be a yes-instance of Cubic Planar Vertex Cover. Let V' be a vertex cover of size at most k of G. Let $E' = \{\{u, s(u)\} : u \in V'\}$, i.e., E' is the set of edges between vertices in V' and the vertices in V_{star}. By Lemma 15.9, every induced C_{11} with a vertex set C in $G^\dagger$ corresponds to an edge $\{u, v\}$ in G and is of the form $S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$. For every edge $\{u, v\}$ in G, E' contains either $\{u, s(u)\}$ or $\{v, s(v)\}$. Hence, E' kills all the original induced C_{11}s in $G^\dagger$. It is straight forward to verify that deleting E' from $G^\dagger$ does not introduce new induced C_{11}s. □

Lemma 15.11. *Let (G, k) be an instance of* Cubic Planar Vertex Cover. *Let $(G^\dagger, k, R^\dagger)$ be obtained by the reduction described above. Then, $G^\dagger$ is 2-degenerate.*

Proof. Deleting all vertices with degree two in $G^\dagger$ results in a graph which is a disjoint union of star graphs ($K_{1,s}$ for $s \geq 0$). Hence, for any set of vertices $V' \subseteq V(G^\dagger)$, $G^\dagger[V']$ induces a graph having a vertex with degree at most two. □

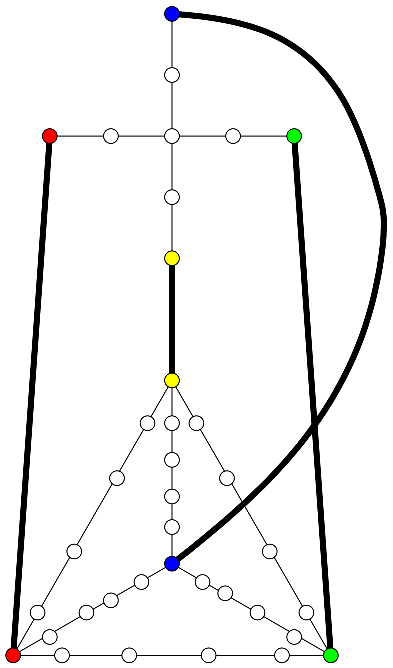
Let (G, k) be an instance of Cubic Planar Vertex Cover. Let $(G^\dagger, k, R^\dagger)$ be obtained by the reduction applied on (G, k) as described above. Introduce four edges in $G^\dagger$ where the one endpoint of the edges is the root vertex of the star gadget in $G^\dagger$ and the other endpoints of the four edges are the four leaf vertices of the star gadget respectively. Let the graph obtained be $G^\ddagger$. The four newly introduced edges are deletable. Hence, the set of non-deletable edges of $G^\ddagger$ is $R^\ddagger = R^\dagger$. We call this process *short-circuiting* of $G^\dagger$.

Lemma 15.12. *Let (G, k) be an instance of* Cubic Planar Vertex Cover. *Let $(G^\dagger, k, R^\dagger)$ be obtained by applying the reduction described above. Short-circuit $G^\dagger$ to obtain $G^\ddagger$. Then:*

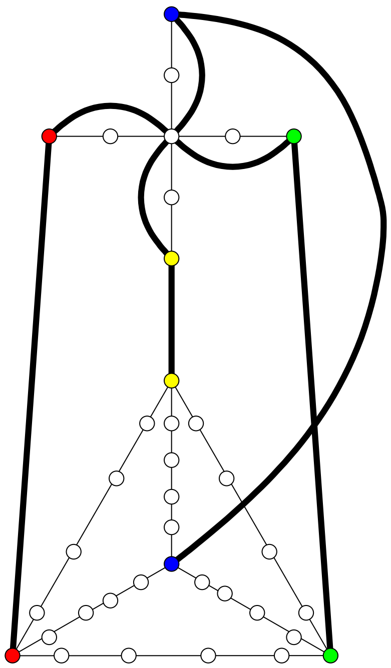
(i) There is no induced C_{11} in $G^\ddagger$.

(ii) $G^\ddagger$ is 2-degenerate.

Proof. (i) By Lemma 15.9, every induced C_{11} in $G^\dagger$ has a vertex set $S_{\{u,v\}} \cup P_{\{s(u),s(v)\}}$ where $\{u, v\}$ is any edge in G. By short-circuiting, the distance between $s(u)$ and $s(v)$ becomes two and hence every original induced C_{11}s are destroyed. It is straight forward to verify that short-circuiting does not introduce new C_{11}s. *(ii)* Deleting all vertices of degree two from $G^\ddagger$ results a forest, which is 2-degenerate. □



(a) The reduction from an instance of Cubic Planar Vertex Cover to an instance of Annotated C_{11}-Free Edge Deletion. Here, the input graph is a K_4. This graph is denoted by $G^{\dagger}$.



(b) Short-circuiting an instance of Annotated C_{11}-Free Edge Deletion obtained by the reduction. This graph is denoted by $G^{\ddagger}$.

15.4 Wrapping up the cross-composition

Here, we give an OR-cross-composition of Cubic Planar Vertex Cover instances into an instance of Annotated C_{11}-Free Edge Deletion.

Construction. Let $\{(G_i, k)\}$, for $1 \leq i \leq 2^r$ (for some $r > 1$) be a set of 2^r instances of Cubic Planar Vertex Cover. Let T be a selector tree as constructed in Section 15.2 (from a complete binary tree T_r) with 2^r deletable leaf edges and let R_T be the set of non-deletable edges in T obtained by the construction. For each (G_i, k), apply the reduction described in Section 15.3 to obtain an instance $(G_i^{\dagger}, k)$ of Annotated C_{11}-Free Edge Deletion. Short-circuit (as described in Section 15.3) each $G_i^{\dagger}$ to obtain $G_i^{\ddagger}$. For $1 \leq i \leq 2^r$, *connect* $G_i^{\ddagger}$ with the i^{th} deletable leaf edge $\{u_i, v_i\}$ of T by the following steps:

- Introduce a path of length two between every leaf vertex of the star gadget in $G_i^{\ddagger}$ and u_i.

- Introduce a path of length five between the root vertex of the star gadget in $G_i^\ddagger$ and v_i.
- Introduce a path of length three between u_i and v_i.

Let G be the constructed graph. Let R be $\bigcup R_i^\ddagger \cup R_T \cup R_N$, where R_N is the set of all edges introduced in the above steps. This completes the construction.

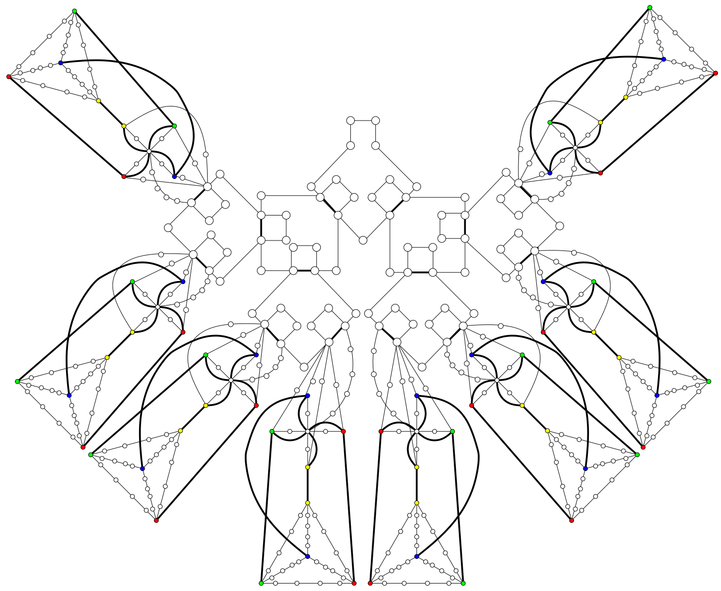


Figure 15.5: OR-cross-composition. For convenience, all the Cubic Planar Vertex Cover instances are shown as K_4. Here, $r = 3$ and all except the bold edges are in R.

Lemma 15.13. *Let $(G, k + r + 4, R)$ be obtained by applying the construction described above on 2^r instances (G_i, k) of* Cubic Planar Vertex Cover*, for some $r > 1$. Then G is 2-degenerate.*

Proof. Deleting all vertices with degree two introduced in the construction of G described above, gives a graph which is a disjoint union of $G_i^\ddagger$ and T. By Lemma 15.5, T is 2-degenerate and by Lemma 15.12 (ii), $G_i^\ddagger$ is 2-degenerate for $1 \leq i \leq 2^r$. □

The next two lemmata prove the "OR" part of the OR-cross-composition.

Lemma 15.14. *Let $\{(G_i, k)\}$ be the set of 2^r (for some $r > 1$) instances of* Cubic Planar Vertex Cover. *Let $(G, k + r + 4, R)$ be obtained from $\{(G_i, k)\}$s and a selector tree T (obtained from a complete binary tree T_r in Section 15.2) by the construction described above. For some j, such that $1 \leq j \leq 2^r$, let (G_j, k) be a* *yes-instance of* Cubic Planar Vertex Cover. *Then* $(G, k+r+4, R)$ *is a yes-instance of* Annotated C_{11}-Free Edge Deletion.

Proof. Let $(G_j^\dagger, R_j^\dagger, k)$ be the Annotated C_{11}-Free Edge Deletion instance obtained by the reduction from (G_j, k) described in Section 15.3. Let V_j be a vertex cover of size at most k of G_j and let F_j be the corresponding solution of $(G_j^\dagger, R_j^\dagger, k)$ as given by the proof of Lemma 15.10. By Lemma 15.8 (ii), there exists a solution for (T, r, R_T) selecting the integer j. Let F_T be such a solution. Let $E_j = \{e_1, e_2, e_3, e_4\}$ be the set of edges used to short-circuit $G_j^\dagger$ (in Section 15.3) to obtain $G_j^\ddagger$. Then, we claim that $F = F_T \cup F_j \cup E_j$ is a solution of $(G, k+r+4, R)$.

Since $|F_T| \leq r$ and $|F_j| \leq k$, we obtain $|F| \leq k+r+4$. We need to prove that $G - F$ is C_{11}-free. Since F_T selects j, F_T contains the j^{th} deletable leaf edge $e_j = \{u_j, v_j\}$ of T. Deleting e_j from G creates four induced C_{11}s corresponds to the four leaf vertices of the star gadget of $G_j^\ddagger$. Each of the induced C_{11} is constituted by $\{x_{leaf}, x_{root}\}$, $P_{\{x_{root}, v_j\}}$, $P_{\{v_j, u_j\}}$ and $P_{\{u_j, x_{leaf}\}}$, where x_{leaf} is a leaf vertex and x_{root} is the root vertex of the star gadget in $G_j^\ddagger$ and $P_{\{y,z\}}$ denotes the path introduced between y and z in the construction described above. Each of these induced C_{11}s has a single unique deletable edge $\{x_{leaf}, x_{root}\}$, which is the edge used to short circuit $G_j^\dagger$ to obtain $G_j^\ddagger$ in Section 15.3. Hence every edge in E_j must be in every solution of $G - F_T$. Deleting E_j from G leaves an induced $G_j^\dagger$. Now, $G_j^\dagger - F_j$ is C_{11}-free. It is straight forward to verify that $G - F$ does not have any induced C_{11} having at least one vertex from T and at least one vertex from $G_j^\ddagger$ and there is no induced C_{11} in $G - F$ containing at least one vertex from $G_i^\ddagger$ for $i \neq j$. Hence $G - F$ is C_{11}-free. □

Lemma 15.15. *Let* $\{(G_i, k)\}$ *be the set of* 2^r *(for some* $r > 1$*) instances of* Cubic Planar Vertex Cover. *Let* $(G, k+r+4, R)$ *be obtained from* $\{(G_i, k)\}$*s and* T *(obtained in Section 15.2 from a complete binary tree* T_r*) by the construction described above. Let* $(G, k+r+4, R)$ *be a yes-instance. Then there exists some integer* j*, such that* $1 \leq j \leq 2^r$*, and* (G_j, k) *is a yes-instance of* Cubic Planar Vertex Cover.

Proof. Let F be a solution of size at most $k+r+4$. Since T is an induced subgraph of G, there exists a set of edges E' of T such that $E' \subseteq F$ and $T - E'$ is C_{11}-free. By Lemma 15.8 (iii), $|E'| \geq r$. By Lemma 15.8 (i), E' selects an integer $j \in [1, 2^r]$ in T. Hence, $T - E'$ has four induced C_{11}s corresponding to four edges used to short-circuit $G^\dagger$ to form $G^\ddagger$. It implies that those four edges must be in F. Hence there must be an $F_j \subseteq F$ such that $|F_j| \leq k$ and $G_j^\dagger - F_j$ is C_{11}-free. Now by Lemma 15.10, (G_j, k) is a yes-instance of Cubic Planar Vertex Cover. □

Theorem 34. *The problem* C_{11}-free Edge Deletion *does not admit a polynomial kernel on degenerate graphs unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

Proof. By Corollary 15.4, it is enough to prove the incompressibility of Annotated C_{11}-Free Edge Deletion. Lemmata 15.14 and 15.15 prove the "or" part of the cross-composition of Cubic Planar Vertex Cover into Annotated C_{11}-Free Edge Deletion described in this section. Since the parameter of the cross-composed instance is $k + r + 4$ where r is $\log t$ where t is the number of instances of Cubic Planar Vertex Cover used for the cross-composition, the "pb" condition is satisfied. By Proposition 12.3, Cubic Planar Vertex Cover is NP-complete. Now, the theorem follows from Theorem 4. $\square$

Chapter 16

Hardness for editing on degenerate input

In this section we sketch how to modify the reductions from Chapters 14 and 15 to push the edge editing kernelization hardness to 2-degenerate graphs. We define $\mathrm{sd}(G)$ for a graph G to be a graph G' which is G with every edge subdivided once. Observe that for any graph G, $\mathrm{sd}(G)$ is 2-degenerate.

Theorem 35. *There is a finite set of connected graphs $\mathcal{H}$ such that $\mathcal{H}$-*Free Edge Editing *does not admit a polynomial kernel, even on 2-degenerate graphs, unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

This theorem gives us lower bounds for 2-degenerate graphs in Table 12.1 in Chapter 12.

Proof sketch. We use insights obtained in Chapters 14 and 15; Combining the annotated problem definition used to show kernelization lower bounds for edge deletion on 2-degenerate graphs together with the forbidding of all possible supergraphs from the kernelization lower bounds for the completion problem allow us to force an editing problem to become a deletion problem.

The combination of insights referred to above is as follows. We may use the reduction from Chapter 15 to combine $t = 2^r$ instances of Cubic Planar Vertex Cover into one instance of $\mathcal{H}$-Free Edge Editing of size polynomial in $n + t$ with budget polynomially bounded in $k + \log t$. We use the supergraph construction from Chapter 14 to enforce that no edges are added: Instead of using C_{11} we take a C_{11} but with one vertex replaced by a $\mathrm{sd}(K_{3,3})$; Instead of the subdivided four-leafed star, we take the subdivided four-leafed star where the center vertex is replaced by $\mathrm{sd}(K_5)$. Finally, we need a gadget to force no edge being deleted. Consider the graph $\mathrm{sd}(K_6 - e)$, where $K_6 - e$ is the clique on six vertices with one edge deleted. These constructions ensure that our forbidden graphs never appear in a planar graph, and we may therefore treat the selector gadget and vertex cover gadgets separately from the planar graphs, and furthermore, all these graphs are 2-degenerate.

Having the obstruction set $\mathcal{H}$ to be all supergraphs of the base graphs, except that we allow $\mathrm{sd}(K_6 + e) + v_1v_2$, where v_1 and v_2 are the two degree four-vertices. We have managed to turn $\mathcal{H}$-Free Edge Editing into a problem of the form $\mathcal{H}$-Free Edge Deletion, and we have also made it possible to consider an annotated version, where we disallow the deletion of certain edges.

What remains is to show how to model the vertex cover, but this is easily achievable in the same manner as was done in Chapter 14. This concludes the proof sketch of Theorem 35.

Chapter 17

Concluding remarks

We have proved that for any finite set $\mathcal{H}$ of forbidden induced subgraphs, both $\mathcal{H}$-Free Edge Editing and $\mathcal{H}$-Free Edge Deletion admit polynomial kernelizations on bounded degree input graphs. This extends and generalizes the result of Aravind et al. [ASS14], who showed that $\mathcal{H}$-Free Edge Deletion admits a polynomial kernel when $\mathcal{H}$ is connected on bounded degree input.

We also provided two lower bounds:

(i) For a finite set $\mathcal{H}$ of connected graphs, $\mathcal{H}$-Free Edge Completion does not admit a polynomial kernel on bounded degree input graphs, unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

(ii) Under the same assumption, C_{11}-Free Edge Deletion does not admit a polynomial kernel on 2-degenerate graphs, nor does $\mathcal{H}$-Free Edge Editing.

Since there is a finite set $\mathcal{H}$ of connected graphs such $\mathcal{H}$-Free Edge Completion does not admit a polynomial kernel, we encourage a further study of these problems. We leave it as an open problem whether there is a dichotomy for when $\mathcal{H}$-Free Edge Completion admits a polynomial kernel, restricted to bounded degree graphs and connected, forbidden induced subgraphs.

There are several problems that do not admit subexponential time algorithms on bounded degree graphs, assuming ETH. In particular, Komusiewicz and Uhlmann [KU12] proved that neither Cluster Editing nor Cluster Deletion admits $2^{o(k)}n^{O(1)}$ time algorithms on graphs of bounded degree, assuming ETH. Furthermore, Drange and Pilipczuk [DP15] obtained similar results for Trivially Perfect Editing and Drange et al. [DRVS15] for Star Forest Editing, both on bounded degree input graphs. We note that all the target graph classes mentioned above can be characterized by a finite set of forbidden induced subgraphs. A more fine-grained study of the various sets of obstructions in relation to admitting subexponential time algorithms on graphs of bounded degree would be interesting.

Part V

Bandwidth

Chapter 18

Introduction

A layout, or linear ordering, of a graph G is a bijection $\alpha : V(G) \to \{1, \ldots, |V(G)|\}$, and the bandwidth of the layout α is the maximum over all edges $uv \in E(G)$ of $|\alpha(u) - \alpha(v)| \leq b$. The bandwidth of G is the smallest integer b such that G has a layout of bandwidth b. In the Bandwidth problem we are given as input a graph G and an integer b and the goal is to determine whether the bandwidth of G is at most b. In the optimization variant we are given G and the task is to find a layout with smallest possible bandwidth.

The problem arises in sparse matrix computations, where given an $n \times n$ matrix A and an integer k, the goal is to decide whether there is a permutation matrix P such that PAP^T is a matrix whose all non-zero entries lie within the k diagonals on either side of the main diagonal. Standard matrix operations such as inversion and multiplication as well as Gaussian elimination can be sped up considerably if the input matrix A can be transformed into a matrix PAP^T of small bandwidth [GL81].

Bandwidth is one of the most well-studied NP-complete [GJ79a, Pap76] problems. The problem remains NP-complete even on very restricted subclasses of trees, such as caterpillars of hair length at most 3 [Mon86]. Furthermore, it is NP-hard to approximate the bandwidth within any constant factor, even on trees [DFU11]. The best approximation algorithm for Bandwidth on general graphs is by Dungan and Vempala [DV01], this algorithm has approximation ratio $(\log n)^3$. For trees Gupta [Gup00] gave a slightly better approximation algorithm with ratio $(\log n)^{9/4}$, while for caterpillars a $O(\frac{\log n}{\log \log n})$-approximation [FT09] can be achieved.

Polynomial-time algorithms for the exact computation of bandwidth are known for a few graph classes including caterpillars of hair length at most 2 [APSZ81], cographs [Yan97], interval graphs [KV90] and bipartite permutation graphs [HKM09].

One could argue that the Bandwidth problem is most interesting when the bandwidth of the graph is very small compared to the size of the graph. Indeed, when the bandwidth of G is *constant* the matrix operations discussed above can be implemented in linear time. For each $b \geq 1$ it is possible to recognize the graphs with bandwidth at most b in time $2^{\mathcal{O}(b)}n^{b+1}$ using the classic algorithm of

Saxe [Sax80]. At this point it is very natural to ask how much Saxe's algorithm can be improved. We will prove that assuming the Exponential Time Hypothesis of Impagliazzo, Paturi and Zane [IPZ01, IP01], no significant improvement is possible, even on very restricted subclasses of trees. In particular we show that assuming ETH, there is no $f(b)n^{o(b)}$ time algorithm for BANDWIDTH on trees of pathwidth at most 2. The proof also implies that BANDWIDTH is $W[1]$-hard on trees of pathwidth at most 2.

As a counterweight to the bad news we give the first approximation algorithm for BANDWIDTH of graphs of bounded treelength whose approximation ratio depends only on the bandwidth b, and not on the size of the graph. Specifically, we give a polynomial time algorithm that given as input a graph G of treelength at most ℓ and an integer b, either correctly concludes that the bandwidth of G is greater than b or outputs a layout of quality at most $(b\ell)^{O(b^2\ell)}$.

A graph G has treelength ℓ if there exist a tree decomposition of G such that for every pair of vertices in a bag, the distance between them in G is at most ℓ. This measure was introduced by Dourisboure and Gavoille [DG07]. The graphs of bounded treelength are a rich graph class containing among others trees, chordal graphs and graphs of bounded hyperbolicity [CDE$^+$08]. In order to obtain our result, we define a new graph measure based on tree decompositions called induced treelength, where the graph induced by each bag of the decomposition is to be of diameter at most ℓ. We then prove that induced treelength and treelength differ by at most a factor 3 and that we can approximate induced treelength in polynomial time.

As a critical subroutine for our main algorithm we develop an approximation algorithm for trees. More specifically, we give a polynomial time algorithm that given as input a tree T and integer b either correctly concludes that the bandwidth of T is greater than b or outputs a layout of width at most $b^{O(b)}$. This algorithm again, utilizes an approximation algorithm for the bandwidth of caterpillars with ratio $O(b^3)$. Our algorithm for trees outperforms the $(\log n)^{9/4}$-approximation algorithm of Gupta [Gup00] whenever $b = o(\frac{\log \log n}{\log \log \log n})$. Our algorithm is the first *parameterized approximation* algorithm for the BANDWIDTH problem on trees, that is an algorithm with approximation ratio $g(b)$ and running time $f(b)n^{O(1)}$. A parameterized approximation algorithm for the closely related TOPOLOGICAL BANDWIDTH problem has been known for awhile [Mar08b], while the existence of a parameterized approximation algorithm for BANDWIDTH, even on trees was unknown prior to this work.

An interesting aspect of our approximation algorithm is the way we lower bound the bandwidth of the input graph G. It is well known that the bandwidth of a graph G is lower bounded by its *pathwidth*, and by its *local density*. One might wonder how far these lower bounds could be from the true bandwidth of G. It was conjectured that the answer to this question is "not too far", in particular that any graph with pathwidth c_1 and local density c_2 would have bandwidth at most c_3 where c_3 is a constant depending only on c_1 and c_2. Chung and Seymour [CS89] gave a counterexample to this conjecture by constructing a special kind of trees, called *cantor combs*, with pathwidth 2, local density at most 10, and bandwidth approximately $\frac{\log n}{\log \log n}$. Our approximation algorithms output one of these three obstructions in the case that the bandwidth is large and hence shows that the only structures driving up the bandwidth of graphs of bounded treelength, trees and caterpillars, are indeed pathwidth, local density and cantor comb-like subgraphs. Hence, we give a characterization of the graphs of bounded treelength that also has bounded bandwidth.

Related Work

There is a vast literature on the Bandwidth problem. For an example the problem has been extensively studied from the perspective of approximation algorithms [DFU11, DV01, Fei00, FT09, Gup00], parameterized complexity [BFH94, GHK$^+$11, Sax80], polynomial time algorithms on graph classes [APSZ81, HKM09, KV90, Yan97], and graph theory [CCDG82, CS89]. We focus here on the study of algorithms for Bandwidth for small values of b.

Dragan, Köhler and Leitert [DKL14] gave a constant factor approximation of Bandwidth for graphs of bounded pathlength, the linear version of treelength. We note that none of the results we obtain are covered by this. In particular trees, and hence graphs of bounded treelength, are highly non-linear. However, graphs of bounded pathlength are also of bounded treelength.

Following the $2^{O(b)}n^{b+1}$ time algorithm of Saxe [Sax80], published in 1980, there was no progress on algorithms for the recognition of graphs of constant bandwidth. With the advent of parameterized complexity in the late 80's and early 90's [DF99] it became an intriguing open problem whether one could improve the algorithm of Saxe to remove the dependency on b in the exponent of n, and obtain a $f(b)n^{O(1)}$ time algorithm.

In a seminal paper from 1994, Bodlaender, Fellows, and Hallet [BFH94] proved that a number of layout problems do not admit fixed-parameter tractable algorithms unless $\mathsf{FPT} = \mathsf{W[t]}$ for every $t \geq 1$, a collapse considered by many to be almost as unlikely as $\mathsf{P} = \mathsf{NP}$. In the same paper Bodlaender, Fellows, and Hallet [BFH94] claim that their techniques can be used to show that a $f(b)n^{O(1)}$ time algorithm for Bandwidth would also imply $\mathsf{FPT} = \mathsf{W[t]}$ for every $t \geq 1$. Downey and Fellows ([DF99], page 468) further claim that the techniques of [BFH94] imply that even a fixed-parameter tractable algorithm for Bandwidth *on trees* would yield the same collapse. Unfortunately a full version of [BFH94] substantiating these claims is yet to appear.

18.1 Preliminaries

Linear orderings and bandwidth

A *linear ordering* α of a set S is a bijection between S and $[|S|]$. Given a graph $G = (V, E)$ and a linear ordering α over V, the *bandwidth* of α denoted $\mathrm{bw}(G, \alpha) = \max_{uv \in E} |\alpha(u) - \alpha(v)|$. And furthermore, the bandwidth of G denoted

$\mathrm{bw}(G) = \min\{\mathrm{bw}(G, \alpha) \mid \alpha \text{ is a linear ordering over } V\}$. We say that α is a k-*bandwidth ordering* of a graph G if $\mathrm{bw}(G, \alpha) \leq k$. And we say that a bandwidth ordering α of G is optimal if $\mathrm{bw}(G, \alpha) = \mathrm{bw}(G)$.

Sparse orderings

Let u and v be a pair of vertices of a graph G and α an ordering of $V(G)$. We then say that u is left of v in α if $\alpha(u) < \alpha(v)$ and that u is right of v if $\alpha(v) < \alpha(u)$. A sparse ordering β of a graph G is an injective function from $V(G)$ to $\mathbb{Z}$. And the bandwidth of a sparse ordering β of G, denoted $\mathrm{bw}(G, \beta) = \max_{uv \in E} |\beta(u) - \beta(v)|$. We say that a linear ordering α of G is a compression of a sparse ordering β of G if for every pair of vertices u, v in G it holds that $\beta(u) < \beta(v)$ if and only if $\alpha(u) < \alpha(v)$.

Inclusion intervals

For a graph T, an integer b and a b-bandwidth ordering α we provide the following definitions. Given a set of vertices $Y \subseteq V(T)$ we define the *inclusion interval of* Y, denoted $I(Y)$ as $[\min \alpha(Y), \max \alpha(Y)]$ and for two vertices u and v we define $I(u, v)$ as $I(\{u, v\})$ or equivalently $[\min\{\alpha(u), \alpha(v)\}, \max\{\alpha(u), \alpha(v)\}]$. Given a subgraph H of T we define $I(H)$ as $I(V(H))$. Whenever necessary, we will use subscripts to avoid confusion about which ordering is considered.

Stretched path and passing through

For a subpath $\hat{P}_l = \{v_1, \ldots, v_l\}$ of a graph T we say that $\hat{P}_l$ is *stretched* with respect to a b-bandwidth ordering α if $|\alpha(v_{i+1}) - \alpha(v_i)| = b$ for every $i \in [1, l)$. Observe that as α is injective, stretched implies either $\alpha(v_1) < \alpha(v_2) < \cdots < \alpha(v_l)$ or $\alpha(v_l) < \cdots < \alpha(v_2) < \alpha(v_1)$. Furthermore, we say that a path P *passes through* some subgraph H in α if $I(H) \subseteq I(P)$.

Lower bounds

Definition 18.1 (Local density)**.** For a graph G we define the *density* of G as

$$\mathrm{ds}(G) = \frac{|V(G)| - 1}{\mathrm{diam}(G)}.$$

Based on this we define *local density* as $\rho(G) = \max_{G' \subseteq G} \mathrm{ds}(G')$.

The following proposition will be used repeatedly in our arguments.

Proposition 18.2 (Folklore)**.** *For every graph G it holds that $\rho(G) \leq \mathrm{bw}(G)$ and $\mathrm{pw}(G) \leq \mathrm{bw}(G)$.*

18.2 Overview of the algorithms

We will provide parameterized approximation algorithms for BANDWIDTH on increasingly general graph classes. We start out with an algorithm for caterpillars. Then, utilizing the algorithm for caterpillars, we give a parameterized approximation algorithm for trees. Finally, using the algorithm for trees, we devise an algorithm for graphs of bounded treelength.

Given a caterpillar T and a positive integer b, `CatAlg` either returns a $48(b+1)b^2$-bandwidth ordering of T or an obstruction forcing the bandwidth of T above b. To obtain this we define an obstruction for bandwidth on caterpillars inspired by Chung & Seymour [CS89] and search for these objects. Based on the appearance of these objects in T we construct an interval graph such that either the interval graph has low chromatic number or the bandwidth of T is large. If the interval graph has low chromatic number we use a coloring of this graph to give a low bandwidth layout of T. Together with the algorithm we also obtain a characterization of caterpillars of low bandwidth.

Given a tree T and positive integers b and p such that $\mathrm{pw}(T) \leq p$, `TreeAlg` either returns a $(7680b^6)^b$-bandwidth ordering of T or an obstruction to the bandwidth of T being low. The high level outline of the algorithm is as follows. The algorithm first decomposes the tree into several connected components of smaller pathwidth and recurses on these. Then it builds a host graph for T that is a caterpillar and applies `CatAlg` on the host graph. Finally, it combines the result of `CatAlg` with the results from the recursive calls, to give a $(7680b^6)^b$-bandwidth ordering of T. Also for trees, we obtain a characterization for having low bandwidth. We note that the bandwidth of the obtained ordering can be lowered to $(1152b^3)^b$ if we do not require the algorithm to output an obstruction.

Last, we give an algorithm `TreeLengthAlg` that given a graph G of treelength at most ℓ and a positive integer b either returns a $(b\ell)^{O(b^2\ell)}$-bandwidth ordering of G or an obstruction to G having bandwidth at most b. The algorithm applies the algorithm for trees on the decomposition tree and plugs the content of the bags into the returned ordering. Furthermore, we prove that obstructions obtained when computing the bandwidth of the tree decomposition carry over to obstructions in the input graph. More specifically, we prove that if a graph of bounded treelength has bounded pathwidth, bounded local density and does not contain a large skewed Cantor comb as a subgraph, it does have bounded bandwidth.

Chapter 19

Caterpillars

The bandwidth of caterpillars is, somewhat surprisingly, a well-studied problem. Assmann et al. [APSZ81] proved that the bandwidth of caterpillars of stray length 1 and 2 is polynomial time computable. Monien [Mon86] completed the story of polynomial time computability by proving that BANDWIDTH on caterpillars of stray length 3 is NP-hard. Furthermore, Haralambides et al. [HMM91] gave an $O(\log n)$ approximation algorithm, which later was improved to $O(\log n/ \log\log n)$ by Feige & Talwar [FT09]. We now give the first parameterized approximation algorithm of BANDWIDTH on caterpillars, namely a $O(b^2)$-approximation. In addition, we give the first characterization of caterpillars of low bandwidth.

19.1 Skewed Cantor combs

Chung & Seymour [CS89] defined *Cantor combs.* These are very special caterpillars defined in such a way that they have small local density, but high bandwidth. The definition of Cantor combs is very strict - it precisely defines the length of all the paths in the caterpillars. For our purposes we need a more flexible definition which captures all caterpillars that are "similar enough" to Cantor combs. We call such caterpillars *skewed Cantor combs*, and we will prove that they also have high bandwidth. Our algorithm will scan for skewed Cantor combs as an obstruction for bandwidth and if none of big enough size are found it will construct a $O(b^3)$-bandwidth ordering based on the appearance of smaller versions of these objects.

For positive integers b and k we now define a *skewed b-Cantor comb* of depth k, denoted $S_{b,k}$ inductively as follows: $S_{b,1}$ is a path of length one. For the induction step to be well-defined we mark two vertices of every skewed b-Cantor comb as end vertices. For an $S_{b,1}$ the two vertices are the end vertices. For $k > 1$ we start with two skewed b-Cantor combs of depth $k-1$, lets call them S and S' and furthermore let x, y and x', y' be their end vertices respectively. Connect y to x' by a path P of length at least two. Furthermore, let Q be a stray connected to an internal vertex v of P. Mark x and y' as the end vertices of the construction and let B be the path from x to y'. The path B will be referred to as the *backbone* of the skewed Cantor comb. Let d be the maximum distance from v to any vertex in B. If Q has at least $2bd$ vertices we say that the graph described is a skewed b-Cantor comb of depth k. And furthermore, that the skewed Cantor comb is *centered around* the stray Q and that Q is a *level* k stray in the skewed Cantor comb. For skewed Cantor combs such that $k = b$, we will also use the simplistic notation S_b for $S_{b,b}$.

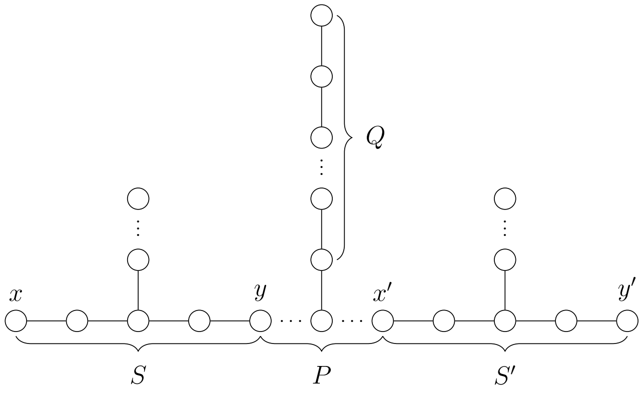


Figure 19.1: A skewed b-Cantor comb of depth 2 for some b.

The goal of the remainder of this section is to prove that the bandwidth of an S_b is at least b and hence that skewed Cantor combs can be utilized as obstructions for having low bandwidth. But before we do this, we will prove that for any optimal bandwidth ordering of any skewed Cantor comb it holds that one of the edges stretched the farthest is embedded intersecting with the embedding of the backbone. It seems natural that the bottleneck of the bandwidth is indeed close to the backbone, as this is where all the vertices of degree more than two are. And as the lemma below shows, this is indeed the case.

Lemma 19.1. *Let S be a skewed b-Cantor comb of depth k with backbone B and α an optimal bandwidth ordering of S. Then there exists an edge uv of S such that $I(u, v)$ intersects $I(B)$ and $|\alpha(u) - \alpha(v)| = \mathrm{bw}(S)$.*

Proof. Let C_B be the connected component of $S[\alpha^{-1}(I(B))]$ that contains B. Observe that $S - C_B$ is a collection of paths, with each path being a subpath of a stray and having exactly one neighbor in C_B. We will now construct a new ordering β based on α. First, for every vertex $x \in C_B$ we let $\beta(x) = \alpha(x)$.

Let L contain every vertex $u \in N(C_B)$ such that $\alpha(u) < I(B)$ and R contain every vertex $v \in N(C_B)$ such that $I(B) < \alpha(v)$. Then exhaustively apply the following rule; if there are vertices $u, v \in L \cup R$ that are connected in $S - C_B$, we discard the vertex furthest away from C_B from its respective set (L or R). Note that by construction, if two such vertices exists, their distance from C_B will be different. Now, every vertex in $V(S) \setminus (C_B \cup R \cup L)$ is connected to exactly one vertex in $R \cup L$. Let L occupy the positions in β immediately to the left of C_B in the same order as they appear in α. Then similarly embed R in β immediately to the right of C_B. Observe that for every two vertices u and v embedded so far it holds that $|\beta(u) - \beta(v)| \leq |\alpha(u) - \alpha(v)|$. Furthermore, let u and v be an edge stretched the farthest by β so far. Observe that $I_\beta(u, v)$ intersects with $I_\beta(B)$ and that $|\beta(u) - \beta(v)| \geq \max(|L|, |R|)$.

We will now embed the remaining vertices. Let u be the rightmost vertex of L with respect to β and observe that since u is of degree at most two and have exactly one neighbor already embedded, it holds that u has at most one neighbor not embedded by β. If this neighbor v exists, we embed v at the rightmost position to the left of L in β and add v to L. Finally, independently of the existence of v, we remove u from L. Observe that $|\beta(u) - \beta(v)| \leq |L|$ and that v has exactly one embedded neighbor. We continue this process as long as L is not empty, before we apply a symmetrical process to R. The process just described will be referred to as rolling out L and R.

Note that we might have assigned non-positive positions to some vertices while rolling out L. We fix this by incrementing all assignments of β simultaneously until all values are positive. After this, we compress β to obtain a proper ordering.

Observe that β is a valid bandwidth ordering. Since no new vertices are embedded within $I_\beta(B)$ while rolling out L and R it still holds that for every edge uv such that $I_\beta(u, v)$ intersects $I_\beta(B)$ it follows that $|\beta(u) - \beta(v)| \leq |\alpha(v) - \alpha(u)|$. If follows from the argument from the rolling out process that for every edge uv such that $I(u, v)$ and $I(B)$ are not intersecting, it holds that $|\beta(u) - \beta(v)| \leq \max(|L|, |R|)$. This has several implications. First, it follows that $\text{bw}(S, \beta) \leq \text{bw}(S, \alpha)$ and since α is optimal, so is β. Second, it also follows that there exists an edge uv with $I_\beta(u, v)$ intersecting $I_\beta(B)$ so that $|\beta(u) - \beta(v)| = \text{bw}(S)$. And third, for the very same edge uv, it holds that $|\beta(u) - \beta(v)| \leq |\alpha(u) - \alpha(v)|$. And due to the construction it holds that $I_\alpha(u, v)$ intersects $I_\alpha(B)$. And hence there exists an edge uv such that $I_\alpha(u, v)$ intersects $I_\alpha(B)$ and $|\alpha(u) - \alpha(v)| \geq |\beta(u) - \beta(v)| = \text{bw}(S)$. Due to α being optimal we obtain equality and hence the proof is complete. □

We are now ready to prove the main result of this section, namely that skewed Cantor combs are of large bandwidth. The proof of this lemma is inspired by the one for Cantor combs given by Chung and Seymour ([CS89], Lemma 2.1).

Lemma 19.2. *For $b \geq k \geq 1$, the bandwidth of any $S_{b,k}$ is at least k.*

Proof. Assume for a contradiction that there is a $\hat{S}_{b,k}$ such that $\text{bw}(\hat{S}_{b,k}) < k$. Furthermore, assume without loss of generality that k is the smallest such value with respect to b. Observe that $k > 1$. Let α be an ordering of $\hat{S}_{b,k}$ of bandwidth at most $k - 1$. Let S, S', P and Q be as in the definition of skewed Cantor combs. By assumption the bandwidth of both S and S' are $k - 1$. Let B, B_S and $B_{S'}$ be the backbones of $\hat{S}_{b,k}, S$ and S' respectively.

Let β be the compressed version of α when restricted to S. Since α is of bandwidth $k - 1$, it follows that β is of bandwidth at most $k - 1$ and by our assumption β is an optimal bandwidth ordering of S. By Lemma 19.1 we know that there exists an edge uv in S such that $I_\beta(u, v) \cap I_\beta(B_S)$ is non-empty and

$|\beta(u) - \beta(v)| = k - 1$. It follows that $I_\alpha(u, v) \cap I_\alpha(B_S)$ is non-empty and $|\alpha(u) - \alpha(v)| = k - 1$. In the same manner we obtain an edge $u'v'$ from S'. Assume without loss of generality that $\alpha(u) < \alpha(v)$ and that $\alpha(u') < \alpha(v')$.

Observe that $\alpha^{-1}(I_\alpha(u, v)) \subseteq S$ and that $\alpha^{-1}(I_\alpha(u', v')) \subseteq S'$. It follows directly that the inclusion intervals has an empty intersection with P. Let q be the vertex in $N(Q)$. We can assume without loss of generality that $\alpha(v) < \alpha(q)$, as otherwise we can reverse α and relabel u, v, u' and v' to adhere to the assumptions above. There are two cases to consider, either $\alpha(q) < \alpha(u')$ or $\alpha(v') < \alpha(q)$.

First, we consider the case when $\alpha(q) < \alpha(u')$. Recall that $b \geq k > 1$. It follows that $|E(B)| \geq 4$ and hence that $|I(B)| \leq (k-1)|E(B)| + 1 < b|E(B)| \leq |V(Q)|$. It follows that there is a vertex $q' \in Q$ such that $\alpha(q') \notin I(B)$. First, we consider the case when $\alpha(q') < I(B)$. It follows that $\alpha(q') < \alpha(u) < \alpha(v) < \alpha(q)$. Since there is a path from q' to q disjoint from S, it follows that $I(u, v)$ must contain a vertex of Q, which is a contradiction to $\alpha^{-1}(I(u, v)) \subseteq S$. The second case follows by a symmetrical argument for S'.

It remains to consider the case when $\alpha(v') < \alpha(q)$. Recall that that $I(u, v)$ and $I(u', v')$ are disjoint. Again we consider two cases. First, let $\alpha(v) < \alpha(u')$. We are then in the situation that $\alpha(v) < \alpha(u') < \alpha(v') < \alpha(q)$ and since there is a path from v to q avoiding S' it follows that this path has a non-empty intersection with $I(u', v')$, which is a contradiction. The case $\alpha(v') < \alpha(u)$ follows by a symmetric argument and hence the proof is complete. □

From now on, whenever talking about an *obstruction* we are either referring to a skewed Cantor comb of appropriate size, a graph of high density or a tree of high pathwidth.

19.2 Directions

Given a caterpillar T and a backbone $B = \{b_1, \ldots, b_\ell\}$ we say that B is *maximized* if there is no longer backbone of T. We also define $\text{pos}(P)$ for every stray P in T with respect to B, as the integer i such that P is attached to the vertex b_i. Furthermore, we let $|P|$ denote $|V(P)|$. We will now, given a stray Q define X_Q and Y_Q. X_Q will contain strays P to the left of Q that can be used to recursively build a skewed b-Cantor comb around Q. The first bound in the definition below assures that for a skewed b-Cantor comb S_P around P it holds that all of S_P is strictly to the left of Q. The second bound assures that Q is long enough when considering its distance away from vertices in S_P. Similarly, Y_Q consists of strays to the right of Q that can be utilized for building a skewed b-Cantor comb around Q.

Definition 19.3. Let T be a caterpillar, $B = \{b_1, \ldots, b_k\}$ a backbone of T and b a positive integer. For every stray Q we let

- $X_Q = \left\{P \mid \text{pos}(P) + \frac{|P|}{2(b+1)} < \text{pos}(Q) \text{ and } \text{pos}(Q) - \frac{|Q|}{2(b+1)} \leq \text{pos}(P) - \frac{|P|}{2(b+1)}\right\}$ and
- $Y_Q = \left\{P \mid \text{pos}(Q) < \text{pos}(P) - \frac{|P|}{2(b+1)} \text{ and } \text{pos}(P) + \frac{|P|}{2(b+1)} \leq \text{pos}(Q) + \frac{|Q|}{2(b+1)}\right\}$.

Let dep be a function from the strays of T with respect to the backbone B into $\mathbb{N}$ such that there is a skewed $(b+1)$-Cantor comb centered around Q of depth $\operatorname{dep}(Q)$. The exception is if $\operatorname{dep}(Q)$ is 0, then the stray is so short that we ignore it and we hence make no promises with respect to skewed $(b+1)$-Cantor combs. We will now describe an algorithm `FindSCC`, that given a caterpillar T, a maximized backbone B of T and an integer b computes such a function dep. First, for every stray Q it sets $\operatorname{dep}(Q)$ as 2 if $|Q| \geq 4(b+1)$ and 0 otherwise. After this, it searches for a stray Q that is such that both x_Q and y_Q are at least $\operatorname{dep}(Q)$, were $x_Q = \max \operatorname{dep}(X_Q)$ and $y_Q = \max \operatorname{dep}(Y_Q)$. If such a Q is found, it increases $\operatorname{dep}(Q)$ by one. This process continues until no such stray Q can be found or dep evaluates to $b+1$ for a stray.

Lemma 19.4. *Given a caterpillar T, a maximized backbone B and an integer b, the algorithm* ***`FindSCC`*** *outputs in $O(bn^3)$ time a function* dep *such that for every stray Q in T there is a skewed $(b+1)$-Cantor comb of depth* $\operatorname{dep}(Q)$ *centered around Q in T.*

Proof. We first prove that the algorithm terminates in $O(bn^3)$ time. The initial set up of evaluating all strays to either 0 or 2 can be carried out in $O(n)$ time. After this, it iterates over all strays Q in $O(n)$ and for every such stray spends $O(n)$ time computing x_Q and y_Q. Hence, we increment the function for one stray in at most $O(n^2)$ time. Since there are at most n strays and the function value of each of these strays are incremented at most $b+1$ times, it follows that the algorithm terminates in $O(bn^3)$ time.

It remains to prove that for every stray Q there is a skewed $(b+1)$-Cantor comb of depth $\operatorname{dep}(Q)$ centered around Q in T. We prove this by induction. If $\operatorname{dep}(Q)$ evaluates to 0, no promises are made regarding skewed Cantor combs. If $\operatorname{dep}(Q)$ evaluates to 2 however, there should be a skewed $(b+1)$-Cantor comb of depth 2 centered around Q. Recall that in this case it holds that $|Q| \geq 4(b+1)$. And since the backbone is maximized the distance from $N(Q)$ to a leaf of the backbone is at least $4(b+1) \geq 4$. It follows that there is a skewed $(b+1)$-Cantor comb of depth 1 immediately at each side of Q. Observe that this completes the argument for the initial set up.

We now consider a stray Q such that both x_Q and y_Q evaluates to $k = \operatorname{dep}(Q)$. There is a stray P in X_Q with a skewed $(b+1)$-Cantor comb of depth k, from now on referred to as S_P, centered around P. Let d_P be the maximum distance from $N(P)$ to a vertex on the backbone of S_P and recall from the definition of skewed Cantor combs that $d_P \leq |P|/[2(b+1)]$. It follows immediately from the first bound in the definition of X_Q that S_P lies entirely to the left of $N(Q)$ on the backbone B. Let d^l_Q be the maximum distance from $N(Q)$ to a vertex in S_P. We need to argue that $|Q| \geq 2(b+1)d^l_Q$. From the second condition of X_Q we know that $\operatorname{pos}(Q) - \frac{|Q|}{2(b+1)} \leq \operatorname{pos}(P) - \frac{|P|}{2(b+1)}$ and hence the argument regarding Q can be finished by the following calculation

$$\text{pos}(Q) - \frac{|Q|}{2(b+1)} \leq \text{pos}(P) - \frac{|P|}{2(b+1)}$$
$$\Longrightarrow$$
$$\frac{|Q|}{2(b+1)} \geq \text{pos}(Q) - \text{pos}(P) + \frac{|P|}{2(b+1)}$$
$$\Longrightarrow$$
$$\frac{|Q|}{2(b+1)} \geq \text{pos}(Q) - \text{pos}(P) + d_P \geq d^l_Q$$
$$\Longrightarrow$$
$$|Q| \geq 2(b+1)d^l_Q.$$

By applying a symmetric argument to Y_Q we get that Q satisfies the requirements for there being a skewed $(b+1)$-Cantor comb of depth $k+1$ centered around Q. Hence it is indeed correct to increment the value of dep(Q).

□

The reader should note that `FindSCC` does not detect all skewed Cantor combs. But, as it turns out, these stricter versions are sufficient for our purposes. Given a function dep we say that Q is *pushed east* if $x_Q \geq y_Q$ and *pushed west* otherwise ($x_Q < y_Q$). From now on, we will assume that the function applied when evaluation whether a stray is pushed west or east, is the depth function calculated by running `FindSCC`. We will now build an interval graph that represents the various strays in the caterpillar. We will use a coloring of this graph to shift the embeddings of the various strays in such a way that they will not collide. To ease the reading of the proofs we will use big_b for the expression $48(b+1)b^2$ and small_b for the expression $12b^2$. For the most part, we will only use that big_b is a factor $4(b+1)$ larger than small_b.

Definition 19.5. For a caterpillar T, a maximized backbone $B = \{b_1, \dots, b_l\}$ of T and a positive integer b we define the *directional stray graph* as the following interval graph: for every stray P add the interval

- $[\text{pos}(P)\text{big}_b - \text{small}_b|P|, \text{pos}(P)\text{big}_b]$ if P is pushed west and
- $[\text{pos}(P)\text{big}_b, \text{pos}(P)\text{big}_b + \text{small}_b|P|]$ otherwise.

We say that an interval originating from a stray pushed west is *west oriented* and vice versa. The interval originating from P is said to start at $\text{pos}(P)\text{big}_b$.

Lemma 19.6. *Let T be a caterpillar, b a positive integer, G_I a directional stray graph of T and x and y two natural numbers such that $x < y$. Then there is an algorithm that in time $O(n)$ either concludes that there are*

- *at most $2b$ intervals of length at least $y - x$ in G_I starting within $[x, y]$ or*
- *outputs a graph $H \subseteq G$ such that* $\mathrm{ds}(H) > b$.

Proof. First, we check if there are at most $2b$ intervals starting within $[x, y]$ of length at least $y - x$. If this is the case, we output this conclusion. Otherwise, let K be a set of $2b + 1$ intervals of length at least $y - x$ starting within $[x, y]$. Let x' be the smallest number such that $x \leq x'$ and x' is divisible by big_b and y' the largest number such that $y' \leq y$ and y' is divisible by big_b. Observe that all intervals in K has their starting point within $[x', y']$ by construction. Observe that if $x' = y'$ then $\deg(T) \geq b + 1$ and hence we can easily obtain an $H \subseteq T$ with $\mathrm{ds}(H) > b$. For the rest of the proof we assume $x' < y'$.

Consider the minimum connected, induced subgraph H of T containing the vertices of the strays corresponding to the intervals in K. We will consider H with respect to the backbone such that the strays of H are exactly the ones corresponding to intervals in K. Let $z = y' - x'$ and observe that every stray in H contains at least $q = z/\mathrm{small}_b$ vertices and that the backbone of H is of length $r = z/\mathrm{big}_b$. Finally, we remove leaves in H until all strays are of length exactly q and output H. It remains to argue that $\mathrm{ds}(H) > b$.

$$\begin{aligned}
\mathrm{ds}(H) &= \frac{|V(H)| - 1}{\mathrm{diam}(H)} \\
&= \frac{(2b+1)q + r + 1 - 1}{2q + r} \\
&= \frac{2(b+1)q + r}{2q + r} \\
&= \frac{2(b+1) \cdot 4(b+1)r + r}{2 \cdot 4(b+1)r + r} \\
&= \frac{8b^2 + 16b + 9}{8b + 9} \\
&> \frac{8b^2 + 9b}{8b + 9} \\
&= \frac{b(8b+9)}{8b+9} = b
\end{aligned}$$

□

We will now prove that for a caterpillar T, integer b and a corresponding directional stray graph G_I we can either obtain a coloring of G_I with few colors or an obstruction forcing the bandwidth of T above b. We will utilize the coloring later to obtain an approximation. The argument is based on that if no such coloring exists, we know that there is a large clique in G_I. We process the east oriented intervals in this clique by decreasing starting value and maintain a skewed b-Cantor comb of depth d at the very right of the clique. By applying Lemma 19.6 we either find a stray at an appropriate distance from the maintained cantor comb that we use to build a deeper cantor comb or a dense subgraph $H \subseteq G$. If we find an appropriate stray, we know since it is pushed east, that there is a corresponding skewed b-Cantor comb of depth d on the other side of the stray. In this manner we iteratively build a large skewed $(b+1)$-Cantor comb.

Lemma 19.7. *Let T be a caterpillar, b a positive integer and G_I some directional stray graph of T. Then there is an algorithm that in $O(bn)$ time outputs either*

- *a coloring γ of G_I using less than* small_b *colors,*
- *a graph $H \subseteq G$ such that* $\text{ds}(H) > b$ *or*
- *a skewed b-Cantor comb $\hat{S}_{b+1} \subseteq G$.*

Proof. An optimal coloring γ of G_I can be found in $O(n)$ time by Golumbic [Gol04]. If γ uses less then $\text{small}_b = 12b^2$ colors, we output it. Otherwise, we know that there is a number w such that at least $12b^2$ of the intervals of G_I contains w. This follows from the well-known result that $\chi(G_I)$ equals the size of the maximum clique of G_I, since G_I is an interval graph. Let I be the set of all east oriented intervals containing w and assume without loss of generality that I is of size at least $6b^2$.

Discard the elements of I with the highest starting value and let $[x^1, y^1]$ be a discarded element. If we discarded more than $2b$ elements, we output $H = G[N[b_\ell]]$ for $\ell = x^1/\text{big}_b$. Observe that H has at least $2b+2$ vertices, a diameter of 2 and hence density strictly larger than b. Either H was output or we have at least $6b^2 - 2b$ elements left in I. We will start by giving a lower bound on the length of the intervals in I. Consider an element $[x, y]$ of shortest length in I. By definition $x < x^1 \leq y$ and by construction $x^1 - x \geq \text{big}_b$. It follows that $y - x \geq \text{big}_b$ and hence that all elements of I are of length at least big_b.

Let $[x_2, y_2]$ be a shortest interval in I and recall that the stray P^2 corresponding to the interval is attached to the backbone vertex b_{c_2} for $c_2 = x_2/\text{big}_b$. Furthermore, $|P^2| = (y_2 - x_2)/\text{small}_b \geq \text{big}_b/\text{small}_b = 4(b+1)$. Since the backbone used when constructing G_I is maximized it follows that the distance from b_{c_2} to any endpoint of the backbone is at least 8 and hence there is an $S_{b+1,2}$ centered around b_{c_2}.

Discard all intervals with starting points within $[x_2 - 2(y_2 - x_2), y_2]$ in I. By dividing the interval into five subintervals of equal length and applying Lemma 19.6 to each of the subintervals, we either discarded at most $6b$ intervals or obtained a subgraph $H \subseteq G$ with $\text{ds}(H) > b$. If such an H was obtained we output it, otherwise we continue. Let $[x_3, y_3]$ be a shortest interval in I and recall that the stray P^3 corresponding to the interval is attached to the backbone vertex b_{c_3} for $c_3 = x_3/\text{big}_b$. We will now argue that P^2 is contained in Y_{P^3}. Since $x_3 < x_2 - 2(y_2 - x_2)$ we know that $y_2 - x_2 < \frac{1}{2}(x_2 - x_3)$. It follows that

$$y_2 - x_2 < \frac{1}{2}(x_2 - x_3)$$
$$\Longrightarrow$$
$$\frac{2(y_2 - x_2)}{\text{big}_b} < \frac{x_2 - x_3}{\text{big}_b}$$
$$\Longrightarrow$$
$$\frac{y_2 - x_2}{2(b+1)\text{small}_b} < \frac{x_2}{\text{big}_b} - \frac{x_3}{\text{big}_b}$$
$$\Longrightarrow$$
$$\frac{|P^2|}{2(b+1)} < \text{pos}(P^2) - \text{pos}(P^3)$$
$$\Longrightarrow$$
$$\text{pos}(P^3) < \text{pos}(P^2) - \frac{|P^2|}{2(b+1)}.$$

And hence the first constraint of being in Y_{P^3} is satisfied. Since $x_3 < x_2 \leq y_3$ it holds that $y_3 - x_3 \geq x_2 - x_3$. Based on this, we get

$$y_3 - x_3 \geq x_3 - x_2 > \frac{1}{2}(x_2 - x_3) + (y_2 - x_2)$$
$$\Longrightarrow$$
$$\frac{y_3 - x_3}{2(b+1)\text{small}_b} > \frac{x_2 - x_3}{2 \cdot 2(b+1)\text{small}_b} + \frac{y_2 - x_2}{2(b+1)\text{small}_b}$$
$$\Longrightarrow$$
$$\frac{y_3 - x_3}{2(b+1)\text{small}_b} > \frac{x_2 - x_3}{\text{big}_b} + \frac{y_2 - x_2}{\text{small}_b}$$
$$\Longrightarrow$$
$$\frac{|P^3|}{2(b+1)} > c_2 - c_3 + \frac{|P^2|}{2(b+1)}$$
$$\Longrightarrow$$
$$\frac{|P^3|}{2(b+1)} + \text{pos}(P^3) > \text{pos}(P^2) + \frac{|P^2|}{2(b+1)}.$$

And hence the second constraint of being in Y_{P^3} is satisfied. Since $[x_3, y_3]$ is east oriented it holds that $x_{P^3} \geq 2$ and hence $\text{dep}(P^3) \geq 3$. It follows by Lemma 19.4 that there is a skewed $(b+1)$-Cantor comb of depth 3 centered around P^3. Discard all intervals with starting points within $[x_3 - 2(y_3 - x_3), x_3]$ and repeat the argument to obtain a $S_{b+1,4}$. We keep repeating the argument until we obtain a S_{b+1} or in the process discover a subgraph H with density strictly larger then b.

Notice that we are discarding at most $6b$ vertices each time, repeating the procedure $b-1$ times and I contains at least $6b^2 - 2b > 6b(b-1)$ intervals. Each step of the computation after obtaining the coloring can easily be executed in $O(n)$ time. The only exception from this is when building the skewed $(b+1)$-Cantor comb, were we identify the skewed Cantor comb to the left of P^i. However, this can be taken care of by storing for each stray the two strays that forced the increment of its depth. And then, only after completing the iterations we identify the entire subgraph to be returned. Since at most $b-1$ iterations are necessary to build the required skewed $(b+1)$-Cantor comb of depth $(b+1)$ we end up with a total running time of $O(bn)$.

□

19.3 Algorithm, correctness and obstructions

We are now ready to give the approximation algorithm for Bandwidth on caterpillars. In addition, due to the nature of the algorithm, we will obtain a sufficient set of obstructions for guaranteeing caterpillars to have low bandwidth.

Algorithm 1: `CatAlg`

```
Input: A caterpillar $T$ and a positive integer $b$.
Output: A $48(b+1)b^2$-bandwidth ordering of $T$ or an obstruction $H \subseteq T$.
Let $B = \{b_1, \dots, b_k\}$ be a maximized backbone of $T$.
Construct the directional stray graph $G_I$ of $T$ with respect to $B$.
Find a minimum coloring of $G_I$.
if $\chi(G_I) \geq 12b^2$ then
    return an obstruction $H \subseteq G$.
end
Let $\alpha(b_i) = \mathrm{big}_b(n+i)$.
Let $\mathcal{P}$ be the collection of strays in $T$ with respect to $B$.
For every stray $P$ in $\mathcal{P}$ let $C(P)$ be the color of the corresponding interval.
for every $P \in \mathcal{P}$ do
    Let $p_1, \dots, p_k$ be the vertices of $P$ such that $\mathrm{dist}(B, p_i) < \mathrm{dist}(B, p_{i+1})$
    for every $i$.
    if $P$ is pushed west then
        Let $\alpha(p_i) = \mathrm{big}_b[n + \mathrm{pos}(P)] + C(P) - i \cdot \mathrm{small}_b$ for every $i$.
    end
    else
        Let $\alpha(p_i) = \mathrm{big}_b[n + \mathrm{pos}(P)] + C(P) + (i-1) \cdot \mathrm{small}_b$ for every $i$.
    end
end
return Compressed version of $\alpha$.
```

Lemma 19.8. *There exists an algorithm that given a caterpillar T and a positive integer b in time $O(bn^3)$ either returns*

- *a* $48(b+1)b^2$*-bandwidth ordering of* T*,*
- *a graph* $H \subseteq T$ *such that* $\mathrm{ds}(H) > b$ *or*
- *a skewed* b*-Cantor comb* $\hat{S}_{b+1} \subseteq T$*.*

Proof. Recall that `FindSCC` runs in $O(bn^3)$ time. Furthermore, by Lemma 19.7 we can obtain a coloring using less then $12b^2$ colors or an obstruction H in time $O(bn)$. Observe that every other step of the algorithm trivially runs in $O(n)$ time. And hence the algorithm runs in $O(bn^3)$ time. If when applying the algorithm from Lemma 19.7 we obtain an obstruction H, we return this H. Otherwise, we have a coloring of G_I using few colors.

We will now prove that α is a sparse ordering of $V(T)$ of bandwidth at most $\mathrm{big}_b = 48(b+1)b^2$. It is clear that for any edge $uv \in E(T)$ it holds that $|\alpha(u) - \alpha(v)| \leq \mathrm{big}_b$. It remains to prove that α is an injective function. Assume for a contradiction that there are two vertices u, v such that $\alpha(u) = \alpha(v)$. Observe that $\alpha(u) \equiv 0 \bmod (\mathrm{big}_b)$ if and only if u is a backbone vertex of T. This comes from the fact that $\chi(G_I) < 12b^2 = \mathrm{small}_b$ and that small_b divides big_b. And since it is clear from the algorithm that no two vertices of the backbone are given the same position we can assume that neither u nor v is a backbone vertex. It holds that $\alpha(u) \equiv c(P) \bmod (\mathrm{small}_b)$ where P is the stray containing u. Observe that the algorithm gives unique positions to all vertices from the same stray and hence u and v must belong to two different strays given the same color. Let P_u be the stray containing u and P_v the strain containing v. Furthermore, let $[x_u, y_u]$ and $[x_v, y_v]$ be the corresponding intervals in G_I. Observe that $I(P_u) \subseteq [x_u, y_u]$ and $I(P_v) \subseteq [x_v, y_v]$ and hence $[x_u, y_u] \cap [x_y, y_v] \neq \emptyset$, which is a contradiction to the obtained coloring being a proper coloring.

□

Theorem 36. *There exists an algorithm that given a caterpillar* T *and a positive integer* b *either returns a* $48(b+1)b^2$*-bandwidth ordering of* T *or correctly concludes that* $\mathrm{bw}(T) > b$ *in time* $O(bn^3)$*.*

Proof. This follows from Lemmata 19.2 and 19.8 together with Proposition 18.2. □

Theorem 37. *Given a caterpillar* T *and a positive integer* b *it either holds that*

- $\mathrm{bw}(T) \leq 48(b+1)b^2$*,*
- $\rho(T) > b$ *or*
- T *contains a* $S_{b+1,b+1}$ *as a subgraph.*

Proof. This follows directly from Lemma 19.8. □

Chapter 20

Trees

The aim of this chapter is to give an approximation algorithm for Bandwidth on trees, namely an $(1152b^3)^b$-approximation. This algorithm crucially depends on the $O(b^2)$-approximation for caterpillars presented in the previous section. In addition, we provide the first characterization of trees of low bandwidth.

20.1 Recursive path decompositions

In this section we will present some decomposition results crucial for our algorithm. First we define recursive path decompositions, which will allow us to partition our graph into several components of slightly lower complexity. The recursive decomposition is used to call the algorithm recursively on easier instances, and then combine the layouts of these instances to a low bandwidth layout of the input tree.

Definition 20.1. Let T be a tree and $P, T^1, \ldots, T^t$ induced subgraphs of T such that $V(T) = \bigcup V(T^i) \cup V(P)$. We then say that $(P, T^1, \ldots, T^t)$ is a *p-recursive path decomposition* of T if P is a path in T and for every i it holds that

- T^i is a connected component of $T - V(P)$ and
- $\text{pw}(T^i) < p$.

Lemma 20.2. *Given a tree T of pathwidth at most p, a p-recursive path decomposition $(P, T^1, \ldots, T^t)$ of T can be found in $O(pn)$ time.*

Proof. It was proven by Skodinis [Sko03] that given a tree T and an integer p one can find a path decomposition $\mathcal{P}$ of T of width p or correctly conclude that $\text{pw}(T) > p$ in time $O(pn)$. Let X and Y be the leaf bags of $\mathcal{P}$. By standard techniques we can assume X and Y to be non-empty. Let u, v be two, not necessarily distinct vertices such that $u \in X$ and $v \in Y$. Let P be the path in T from u to v. Observe that for every bag Z of $\mathcal{P}$ it is true that $Z \cap P$ is non-empty. Hence, if we remove all the vertices of P from T and $\mathcal{P}$ we obtain a path decomposition of $T - V(P)$ of width $p - 1$. It follows that for every connected component T^i of $T - V(P)$ it holds that $\text{pw}(T^i) \leq p - 1$. To complete the proof, observe that both P and the connected components of $T - V(P)$ can be found in $O(n)$ time by graph traversals.

□

We will now introduce a caterpillar that can be built given a tree and a p-recursive path decomposition. This caterpillar is the easier instance mentioned before that we will utilize as a host graph to insert the embeddings of the smaller instances into. This caterpillar will have P as its backbone.

Definition 20.3. Let T be a tree and $(P, T^1, \ldots, T^t)$ a p-recursive path decomposition of T. We construct the *simplified instance* T_S of T with respect to $(P, T^1, \ldots, T^t)$ as follows. First we add P to T_S. Then, for every T^i we first add a path P^i such that $|V(P^i)| = |V(T^i)|$ and then we add an edge from one endpoint of P^i to $N(T^i)$.

Lemma 20.4. *Let T be a tree, $(P, T^1, \ldots, T^t)$ be a p-recursive path decomposition of T and T_S the corresponding simplified instance, then* $\text{bw}(T_S) \leq 2\,\text{bw}(T)$

Proof. Let α be an optimal bandwidth ordering of T. We will now give an ordering β of T_S such that $\text{bw}(T_S, \beta) \leq 2\,\text{bw}(T, \alpha)$. For every $v \in P$, let $\beta(v) = 2\alpha(v)$.

Let $W = \alpha(T^i)$ and observe that for every $x \in W$ such that y is the smallest element in W larger than X it follows by the connectivity of T^i that $y - x \leq \text{bw}(T)$. First, consider the case when at least half of W is less than $\alpha(N(T^i))$. For every $w \in W$ such that $w < \alpha(N(T^i))$, add $2w$ and $2w + 1$ to the initially empty set Z. Let $P^i = \{p_1, \ldots, p_m\}$ such that $\text{dist}(P, p_j) < \text{dist}(P, p_{j+1})$ for every j. For j from 1 to m, let $\beta(p_j)$ be the largest value in Z and discard $\beta(p_j)$ from Z. Observe that for every j it holds that $|\beta(p_j) - \beta(p_{j+1})| \leq 2\,\text{bw}(T)$. And furthermore, $|\beta(p_1) - \beta(N(P^i))| \leq 2\,\text{bw}(T)$. If at least half of W is larger than $\alpha(N(T^i))$ apply a symmetric construction.

To conclude the argument we need to prove that β never maps two distinct vertices of T_S on the same position. It is easy to verify that this never happens for two vertices on P or two vertices in the same tree T^i. Consider now a vertex $u \in V(T^i)$ and a vertex $v \in V(T^j)$ for $i \neq j$. It follows that $\beta(u)/2 \in \alpha(T^i)$ and $\beta(v)/2 \in \alpha(T^j)$, contradicting $i \neq j$. The argument for one vertex in T^i and one in P is identical. We obtain that $\text{bw}(T_S) \leq \text{bw}(T_S, \beta) \leq 2\,\text{bw}(T, \alpha) = 2\,\text{bw}(T)$. □

The algorithm will utilize the following operation when combining the recursively built solutions in the host caterpillar. Let T be a graph, v a vertex of T and α a b-bandwidth ordering of T. Let β' be a sparse ordering such that for every $u \in T$

$$\beta'(u) = \begin{cases} 2[\alpha(v) - \alpha(u)] & \text{if } \alpha(u) \leq \alpha(v) \text{ and} \\ 2[\alpha(u) - \alpha(v)] - 1 & \text{otherwise.} \end{cases}$$

and let β be the bandwidth ordering obtained by compressing β'. We then say that β is α *folded* around v. Observe that $\text{bw}(T, \beta) \leq 2\,\text{bw}(T, \alpha)$.

20.2 Lifting obstructions

We have already proven that the bandwidth of the simplified instance is closely connected to the bandwidth of the original instance. During this section, we will prove that obstructions that occur in the simplified instance can be translated into obstructions in the original instance. By this, we connect the bandwidth of the simplified and original instance even more.

Lemma 20.5. *Let T be a tree, $(P, T^1, \dots, T^T)$ be a p-recursive path decomposition of T, T_S the corresponding simplified instance and H_S a subgraph of T_S. There is an algorithm that in $O(n)$ time outputs $H \subseteq T$ such that* $\mathrm{ds}(H) \geq \mathrm{ds}(H_S)$.

Proof. First, we add the vertices $V(H_S) \cap P$ to H. Then, for every stray P^i in T_S and corresponding subtree T^i of T, we add $|V(H_S) \cap V(P^i)|$ of the vertices closest to P in T^i, to H. Finally, we let $H = T[V(H)]$, or in other words add all the edges of T between vertices in H to H. This completes the construction of H and can be carried out in $O(n)$ time.

Observe that $|V(H_S)| = |V(H)|$ and hence it remains prove that $\mathrm{diam}(H) \leq \mathrm{diam}(H_S)$. Let u and v be two vertices of H. If $u, v \in T^i$ for some i, it follows immediately that $\mathrm{dist}_H(u, v) \leq |V(H) \cap V(T^i)| - 1 = |V(H_S) \cap V(P^i)| - 1 \leq \mathrm{diam}(H_S)$. Otherwise, we let u' be the vertex farthest away from P in $P^i \cap V(H)$ if $u \in T^i$ for some i and u otherwise. Similarly, we obtain a vertex v'. Observe that the shortest path between u and v has the same intersection with P as the shortest path between u' and v'. Furthermore, it holds that $\mathrm{dist}_H(u, P) \leq \mathrm{dist}_{H_S}(u', P)$ and similarly that $\mathrm{dist}_H(v, P) \leq \mathrm{dist}_{H_S}(v', P)$. It follows immediately that $\mathrm{dist}_H(u, v) \leq \mathrm{dist}_{H_S}(u', v')$ and hence that $\mathrm{diam}(H) \leq \mathrm{diam}(H_S)$. □

Lemma 20.6. *Let T be a tree, b a positive integer, $(P, T^1, \dots, T^T)$ a p-recursive path decomposition of T, T_S the corresponding simplified instance and $\hat{S}_\ell \subseteq T_S$ a skewed ℓ-Cantor comb of depth ℓ were $\ell = 4b^2 + 1$. There is an algorithm that in $O(n)$ time either outputs*

- *a subgraph $H \subseteq T$ such that* $\mathrm{ds}(H) > b$ *or*
- *a skewed Cantor comb $\hat{S}_{b+1} \subseteq T$.*

Proof. First, we consider the case when $b = 1$ separately. In this case $\ell = 5$ and hence there is a vertex v in T_S of degree at least 3. Note that this vertex is in P and hence also has degree at least 3 in T. Let $H = T[N_T[v]]$ and observe that $\mathrm{ds}(H) > 1$. In this case we output H. For the remainder of the proof we assume $b \geq 2$.

Let P^i be a stray of T_S that intersects with $\hat{S}_\ell$ and let $Q = P^i[V(P^i) \cap V(\hat{S}_\ell)]$. First, we argue that if T^i has radius r at most $|V(Q)|/(2.5b)$, it follows that $\mathrm{ds}(T^i) > b$. In this case we immediately output $H = T^i$. Since $\ell > b \geq 2$, we know by the definition of skewed Cantor combs that $|V(Q)| \geq 4\ell \geq 68$ and hence that $|V(Q)| - 1 \geq 67|V(Q)|/68$. The claim follows directly from the calculation below.

$$\mathrm{ds}(T^i) = \frac{|V(T^i)|-1}{\mathrm{diam}(T^i)} \geq \frac{|V(Q)|-1}{2r} \geq \frac{67|V(Q)|}{136r} \geq \frac{67|V(Q)|}{136\frac{|V(Q)|}{2.5b}} = \frac{67\cdot 2.5b}{136} > b.$$

We can now assume that the radius of T^i is larger than $|V(Q)|/(2.5b)$. It follows that we can replace the path Q in $\hat{S}_\ell$ by a path in T^i of at least

$$\frac{|V(Q)|}{2.5b} \geq \frac{2\ell d}{2.5b} = 2\left(\frac{\ell}{2.5b}\right)d$$

vertices (were d is from the definition of skewed Cantor combs). After doing this transformation for each stray P^i we are left with a skewed $\lfloor \ell/2.5b \rfloor$-Cantor comb of depth ℓ. And since

$$\frac{l}{2.5b} = \frac{4b^2+1}{2.5b} \geq b+1.$$

it follows that we did indeed obtain a $S_{b+1,\ell}$. By definition it contains a $(b+1)$-Cantor comb of depth $(b+1)$ as a subgraph and hence we output this S_{b+1}. We note that all of the steps above can be executed in $O(n)$ time. □

20.3 Algorithm and Correctness

We are now ready to describe algorithm `TreeAlg` and prove its correctness. Note that `TreeAlg` provides obstructions of low bandwidth and due to this has a worse approximation ratio than what we get in Theorem 38, when we only require the algorithm to correctly conclude that $\mathrm{bw}(T) > b$. The details of this can be found in the proof of Theorem 38.

Lemma 20.7. *Given a tree T and two integers p an b such that $\mathrm{pw}(T) \leq p$, `TreeAlg` terminates in $O(pbn^3)$ time.*

Proof. We start by analyzing the time complexity of the computations done in a specific execution of `TreeAlg` given T', p', b as input, disregarding the recursive calls. The calls to `CatAlg` require $O(b|V(T')|^3)$ time. Finding a p-recursive path decomposition can be done in $O(p|V(T')|) = O(b|V(T')|)$ time by Lemma 20.2. Constructing T'_S can trivially be done in $O(|V(T')|)$ time. By Lemmata 20.5 and 20.6 we know that we can build the obstruction H in $O(|V(T')|)$ time. And furthermore, constructing all the β's require $\sum_{i=1}^{t} O(|T^i|) = O(|V(T')|)$ time. Last, we observe that constructing α requires $O(|V(T')|)$ time. It follows that the time complexity of the computations done in a specific call to `TreeAlg` is $O(b|V(T')|^3)$.

Let $n = |V(T)|$ and $T_1, \ldots, T_l$ the trees given as input at a specific recursion level. Observe that $T_1, \ldots, T_l$ are pairwise disjoint and hence it follows that the time complexity of a recursion level is $\sum_{i=1}^{l} O(b|V(T_1)|^3) = O(bn^3)$. Furthermore, as p is decreased by one at each recursion level it follows that `TreeAlg` runs in time $O(pbn^3)$. □

```
Algorithm 2: TreeAlg
  Input: A tree $T$ and positive integers integers $p$ and $b$ such that $\mathrm{pw}(T) \leq p$.
  Output: A $(7680b^6)^p$-bandwidth ordering of $T$ or an obstruction $H \subseteq T$.
  if $p = 1$ then
    return CatAlg$(T, b)$
  end
  Find a $p$-recursive path decomposition $(P, T^1, \ldots, T^t)$ of $T$.
  Let $\alpha_1 =$ TreeAlg$(T^1, p-1, b), \ldots, \alpha_t =$ TreeAlg$(T^t, p-1, b)$.
  if an obstruction $H$ is returned in one of the recursive calls then
    return $H$
  end
  Let $T_s$ be the simplified instance of $T$ with respect to $(P, T^1, \ldots, T^t)$.
  Let $\alpha_s =$ CatAlg$(T_s, 4b^2)$.
  if an obstruction $H_S \subseteq T_S$ is returned then
    Build obstruction $H \subseteq T$ from $H_S$.
    return $H$
  end
  For every $i$, let $\beta_i$ be $\alpha_i$ folded around $N(P) \cap T^i$.
  For every $v \in P$, let $\alpha(v) = \alpha_s(v)$.
  For every $P_i$ of $T_s$ and every $v \in P_i$ of distance $d$ from $P$ in $T_s$, let
  $\alpha(\beta_i^{-1}(d)) = \alpha_s(v)$.
  return $\alpha$
```

Lemma 20.8. *Given a tree T and positive integers b and p such that $\mathrm{pw}(T) \leq p$, it holds that* `TreeAlg` *in time $O(pbn^3)$ either returns*

- *a $(7680b^6)^p$-bandwidth ordering of T,*
- *a subgraph $H \subseteq T$ such that $\mathrm{ds}(H) > b$ or*
- *a skewed Cantor comb $\hat{S}_{b+1} \subseteq T$.*

Proof. The running time follows directly from Lemma 20.7 and hence it remains to prove the correctness of the algorithm. This we will do by induction on p. For $p = 1$ the correctness follows directly from the correctness of `CatAlg` and hence it remains to prove the induction step.

First, we consider the case when the algorithm returned an obstruction H. If this was done after computing $\alpha_1, \ldots, \alpha_t$, then the obstruction returned from the recursive computation is also a subgraph of T and can hence be returned as is. Otherwise, an obstruction H_S was returned after applying `CatAlg` on T_S. If $\mathrm{ds}(H_S) > 4b^2$ it follows from Lemma 20.5 that we can obtain a subgraph $H \subseteq T$ with $\mathrm{ds}(H_S) > 4b^2$. If H_S is a skewed $4b^2+1$-Cantor comb of depth $4b^2+1$, we can by Lemma 20.6 either obtain an $H \subseteq T$ with $\mathrm{ds}(H) > b$ or a skewed $(b+1)$-Cantor comb of depth $(b+1)$ in T.

It remains to consider the case when the algorithm returns a bandwidth ordering α. Then, by the induction hypothesis α_i is a $(7680b^6)^{p-1}$-bandwidth ordering of T^i for every i. Furthermore, α_s is a $3840b^6$-bandwidth ordering for T_s, since $48(4b^2+1)(4b^2)^2 \geq 48 \cdot 5b^2 \cdot 16b^4 = 3840b^6$. Let u and v be two neighbouring vertices of T. If u and v are vertices in P it follows from $\mathrm{bw}(T_s, \alpha_s) \leq 3840b^6$ that $|\alpha(u) - \alpha(v)| \leq 3840b^6$. Next, we consider the case when either u or v is a vertex in P. Assume without loss of generality that $u \in P$ and let T^j be such that $v \in T^j$. By the definition of β_j it follows that $\beta_j(v) = 1$. It follows that $|\alpha(u) - \alpha(v)| = |\alpha_s(u) - \alpha_s(w)|$ where $\mathrm{dist}(u, w) = 1$, and hence u and w are neighbours in T_s and it follows directly that $|\alpha(u) - \alpha(v)| \leq 3840b^6$. We will now consider the case when u and v are vertices of T^j for some j. Let u' be the vertex in P^j of distance $\beta(u)$ from P and v' the vertex in P^j of distance $\beta(v)$ from P. It follows that

$$\begin{aligned}
|\alpha(u) - \alpha(v)| &= |\alpha_s(u') - \alpha_s(v')| \\
&\leq \mathrm{dist}(u', v')3840b^6 \\
&= |\beta_j(u) - \beta_j(v)|3840b^6 \\
&\leq |\alpha_j(u) - \alpha_j(v)|7680b^6 \\
&\leq (7680b^6)^p
\end{aligned}$$

completing the proof. □

Note that one in the case of $p = 1$ also could solve the instance exactly by Assmann [APSZ81], which would slightly reduce the approximation factor.

Theorem 38. *There exists an algorithm that given a tree T and a positive integer b either returns a $(1152b^3)^b$-bandwidth ordering of T or correctly concludes that $\mathrm{bw}(T) > b$ in time $O(b^2n^3)$.*

Proof. Since we are not required to obtain obstructions we apply `CatAlg` to T_S with $2b$ instead of $4b^2$ in `TreeAlg`. If `CatAlg` returns $48(2b+1)(2b)^2$-bandwidth ordering we obtain a $(1152b^3)^p$-bandwidth ordering by repeating the calculations from the proof of Lemma 20.8. If it returns an obstruction it follows from Lemmata 18.2 and 19.2 that $\mathrm{bw}(T_S) > 2b$. And hence, by Lemma 20.4 it follows that $\mathrm{bw}(T) > b$. Recall that $p < b$ ($\mathrm{pw}(T) \leq \mathrm{bw}(T)$) and hence the proof is complete.

□

Theorem 39. *Given a tree T and a positive integer b it holds that either*

- $\mathrm{bw}(T) \leq (7680b^6)^b$*,*
- $\mathrm{pw}(T) > b$*,*
- $\rho(T) > b$ *or*
- *T contains a $S_{b+1,b+1}$ as a subgraph.*

Proof. If $\mathrm{pw}(T) > b$ we are done. Otherwise, we apply Lemma 20.8. □

Chapter 21

Graphs of bounded treelength

In this chapter we give an approximate algorithm for BANDWIDTH on graphs of bounded treelength. Namely a $(54b\ell)^{49b^2\ell}$-approximation, where ℓ it the treelength of the input graph and b is the requested bandwidth. In addition we characterize graphs that have both low treelength and low bandwidth. For the larger part of this chapter we assume the input graph to be connected and only at the end do we resolve this. The algorithm is very simple and will apply the algorithm for trees on a decomposition tree and then utilize this ordering to create an ordering for the input graph. The main part of this chapter is dedicated to proving that the various obstructions for trees having low bandwidth can be transformed from obstruction for the decomposition tree into obstructions of the input graph. But first, we introduce a new graph measure that will be necessary for our success.

21.1 Induced treelength

In this section we define a new graph measure called *induced treelength*. In addition to require that every two vertices in a bag are close in G, we require a witness of this to be in the bag. In other words, we require the diameter of the graph induced by every bag to have bounded diameter. This stronger requirement makes it easier to route obstructions in the decomposed graph, based on obstructions in the decomposition tree. And as we will prove below, the two measures are closely related.

Definition 21.1 (Induced treelength)**.** Given a graph G and a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ of G, we say that $\mathcal{T}$ is of *length* ℓ if for for every $X \in \mathcal{X}$ it holds that $\operatorname{diam}(G[X]) \leq \ell$. Furthermore, we define the *induced treelength* of G, denoted $\operatorname{ltl}(G)$, as the minimum length over all tree decompositions of G.

Observation 21.2. *For every graph G it holds that* $\operatorname{tl}(G) \leq \operatorname{ltl}(G)$.

Proof. This follows immediately from the observation that a tree decomposition of length ℓ is also a tree decomposition of treelength at most ℓ. □

Lemma 21.3. *There is an algorithm that given a graph G and a tree decomposition $\mathcal{T}$ of G of treelength ℓ, outputs a tree decomposition $\mathcal{T}'$ of G such that the length of $\mathcal{T}'$ is at most 3ℓ in time $(|V(T)| + n)^{O(1)}$.*

Proof. Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G of treelength at most ℓ. For a bag $X^i \in \mathcal{X}$ and two vertices $u, v \in X^i$ we will now construct the tree decomposition $\mathcal{T}_{(i,u,v)} = (T, \mathcal{X}_{(i,u,v)})$ as follows: Fix a shortest path P from u to v in G. Since the treelength of $\mathcal{T}$ is at most ℓ, it follows that the length of P is at most ℓ. Then, we construct $\mathcal{X}_{(i,u,v)}$ as follows: If $X^j \in \mathcal{X}$ and P are vertex disjoint, we let $X^j_{(i,u,v)} = X^j$. And otherwise, we let $X^j_{(i,u,v)} = X^j \cup V(P)$. In both cases, we add $X^j_{(i,u,v)}$ to $\mathcal{X}_{(i,u,v)}$. This completes the construction of $\mathcal{T}_{(i,u,v)}$. We would like to point out that both tree decompositions use the same tree T and that there hence is a natural bijection between the bags in the two decompositions based on the index j in the construction above.

First, we argue that $\mathcal{T}_{(i,u,v)}$ is a valid tree decomposition. The vertex and edge requirement follow directly from every bag in $\mathcal{T}_{(i,u,v)}$ being a superset of the corresponding bag in $\mathcal{T}$. It remains to argue that the connectivity requirement is satisfied. Consider a vertex $v \in V(G)$. If v is not a vertex in P, then the bags containing v induce the same subtree in T in both $\mathcal{T}$ and $\mathcal{T}'$. Now, we consider the case when v is a vertex in P. Let $P = (u = x_1, x_2, x_c = v)$ and observe that since the edge requirement is satisfied, there is for every j a bag containing both x_j and x_{j+1}. Hence, the subgraph of T induced by the bags having a non-empty intersection with P is a connected subtree. It follows that $\mathcal{T}_{(i,u,v)}$ is a valid tree decomposition of G.

For every j we define $H_j = G[X^j_{(i,u,v)}]$. By construction, it holds that $\text{dist}_{H_i}(u, v) \leq \ell$. Finally, we observe that for every j and every vertex $v \in (X^j_{(i,u,v)} \setminus X^j)$, it holds that $\text{dist}_{H_j}(v, X^j) \leq \ell$. This is true since every time we added $V(P)$ to a bag, we did so because the bag already contained some of the vertices of P. And since P is a path of length ℓ, the claim follows immediately.

We are now ready to construct $\mathcal{T}' = (T, \mathcal{X}')$. Note that once again, the tree decomposition will be over the same tree T. For every $j \in V(T)$ we define X'^j as

$$X'^j = \bigcup_{i \in V(T)} \bigcup_{u,v \in X^i} X^j_{(i,u,v)}.$$

That is, the bag X'^j is constructed by first constructing $X^j_{(i,u,v)}$ for every choice of (i, u, v), and then taking the union of $X^j_{(i,u,v)}$ for all choices of (i, u, v).

By repeatedly applying the argument above, $\mathcal{T}'$ is a valid tree decomposition of G. We define $H'_j = G[X'^j]$. By construction, and the argument above, it follows that for every $X^j \in \mathcal{X}$ and every pair of vertices $u, v \in X^j$, it holds that $\text{dist}_{H'_j}(u, v) \leq \ell$. Furthermore, by the argument above, it follows that for every $j \in V(T)$ and every vertex $v \in X'^j \setminus X^j$ it holds that $\text{dist}_{H'_j}(v, X^j) \leq \ell$. Hence, for every bag X'^j it holds that $\text{diam}(G[X'^j]) \leq 3\ell$.

It remains to analyze the running time. Observe that there are polynomial in n and $|V(T)|$ many combinations for (i, u, v). Furthermore, we can identify P

and construct $\mathcal{X}_{(i,u,v)}$ in polynomial time. And finally, we can build a single X'^{i} in polynomial time, as it is the union of polynomially many sets of polynomial size. Hence, the running time is polynomial in $n + |V(T)|$. □

Corollary 21.4. *For every graph G it holds that* $\operatorname{ltl}(G) \leq 3\operatorname{tl}(G)$.

Lemma 21.5. *Given a graph G and an integer ℓ there is an algorithm that in time $n^{O(1)}$ either*

- *outputs a tree decomposition $\mathcal{T}$ of G of length at most 9ℓ or*
- *correctly concludes that both $\operatorname{ltl}(G) \geq \ell$ and $\operatorname{tl}(G) \geq \ell$*

in time $n^{O(1)}$.

Proof. By Dourisboure et al. [DG07] we can obtain a 3-approximation of treelength in $O(nm)$ time. If this algorithm reports that $\operatorname{tl}(G) \geq \ell$ it follows by Observation 21.2 that $\operatorname{ltl}(G) \geq \operatorname{tl}(G) \geq \ell$. Otherwise, we can transform the tree decomposition of treelength at most 3ℓ into a tree decomposition of induced treelength at most 9ℓ by Lemma 21.5 in $n^{O(1)}$ time.

□

Observe that the procedure to obtain non-redundant tree decompositions described in Chapter 2 also preserves the induced treelength of a decomposition.

21.2 Induced treelength and local density

We will now give some basic properties regarding tree decompositions of bounded length when the graph that is being decomposed is of bounded local density. Similar properties were recently provided by Belmonte et al. [BFGR16] for general graphs. However, the bounds we obtain are much better due to the underlying graphs being of bounded local density.

Lemma 21.6. *Let G be a graph, b a positive integer and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G of length ℓ. It then holds for every $X_i \in \mathcal{X}$ that if $|X_i| > b\ell + 1$, then $\operatorname{ds}(G[X_i]) > b$.*

Proof. Assume that there is an $X_i \in \mathcal{X}$ with $|X_i| \geq b\ell + 2$. Then the result follows immediately from the following calculation:

$$\operatorname{ds}(G[X_i]) = \frac{|X_i| - 1}{\operatorname{diam}(G[X_i])} \geq \frac{b\ell + 2 - 1}{\ell} > b.$$

□

Lemma 21.7. *Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a non-redundant tree decomposition of G. For every $\mathcal{Y} \subseteq \mathcal{X}$, it holds that*

$$\left|\bigcup \mathcal{Y}\right| \geq |\mathcal{Y}|.$$

Proof. We prove this by induction on the cardinality of $\mathcal{Y}$. Due to $\mathcal{T}$ being non-redundant, no bag is empty and hence the claim holds for $|\mathcal{Y}| = 1$. For the induction step, consider the minimal subtree $T_{\mathcal{Y}}$ of T that contains all the vertices corresponding to the bags in $\mathcal{Y}$. Let $Y \in \mathcal{Y}$ be a bag that corresponds to a leaf in $T_{\mathcal{Y}}$ and let $Z \in \mathcal{Y}$ be its only neighbor bag in $T_{\mathcal{Y}}$. Let $\mathcal{Y}' = \mathcal{Y} - Y$ and observe that $Y \cap Z = Y \cap (\cup \mathcal{Y}')$. Since $\mathcal{T}$ is compressed, it follows that $Y \setminus Z$ and hence $Y \setminus (\cup \mathcal{Y}')$ is non-empty. By the induction hypothesis we get

$$\left|\bigcup \mathcal{Y}\right| \geq \left|\bigcup \mathcal{Y}'\right| + |Y \cap Z| \geq |\mathcal{Y}'| + |Y \cap Z| \geq |\mathcal{Y}'| + 1 = |\mathcal{Y}|.$$

□

Definition 21.8. Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G. For a vertex $v \in V(G)$ we define the *frequency of v in $\mathcal{T}$*, denoted $\mathrm{freq}(v, \mathcal{T})$, as

$$|\{X_i \in \mathcal{X} \mid v \in X_i\}|.$$

Furthermore, we define the *frequency of $\mathcal{T}$* as $\mathrm{freq}(\mathcal{T}) = \max_{v \in V(G)} \mathrm{freq}(v, \mathcal{T})$.

Lemma 21.9. *Let G be a graph, b a positive integer and $\mathcal{T} = (T, \mathcal{X})$ a non-redundant tree decomposition of G of length ℓ and width t. Then there is an algorithm that in $O(tn + m)$ time either*

- *correctly concludes that* $\mathrm{freq}(\mathcal{T}) \leq 2b\ell$ *or*
- *outputs a $H \subseteq G$ with* $\mathrm{ds}(H) > b$.

Proof. We can in $O(tn)$ time go through all bags of the decomposition and compute the frequencies for all vertices. If the frequencies are bounded by $2b\ell$ we are done. Otherwise, let v be a vertex of frequency at least $2b\ell + 1$ and let Y the union of all the bags containing v. We then output $H = G[Y]$. Hence, H can be constructed in $O(tn + m)$ time.

It remains to prove that $\mathrm{ds}(H) > b$. Let $\mathcal{T}'$ be the tree decomposition of H composed by all the bags in $\mathcal{T}$ containing v and the corresponding subtree. Observe that $\mathcal{T}'$ is indeed a non-redundant tree decomposition of H. We then remove v from all the bags of $\mathcal{T}'$. Since v was in all bags of $\mathcal{T}'$ it holds that $\mathcal{T}'$ is a non-redundant tree decomposition of $H - v$. By applying Lemma 21.7 we get that $|V(H - v)| \geq 2b\ell + 1$ and hence $|V(H)| > 2b\ell + 1$. We observe that the distance from v to any vertex in H is at most ℓ. Hence, we finish the proof by the following calculation:

$$\mathrm{ds}(H) = \frac{|V(H)| - 1}{\mathrm{diam}(H)} > \frac{2b\ell + 1 - 1}{2\ell} = b.$$

□

21.3 Lifting obstructions of high density

The goal of the following section is to prove that if there is a subtree of high density in the tree of a tree decomposition of bounded length, then we can identify a subgraph in the graph itself of high density. But first, we prove that the distance between two vertices in the graph cannot be much larger than the distance between two bags containing them.

Lemma 21.10. *Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G of length ℓ. For two vertices $u, v \in G$ such that $u \in X_i \in \mathcal{X}$ and $v \in X_j \in \mathcal{X}$ it holds that*

$$dist_H(u, v) \leq \ell(\mathrm{dist}_T(i, j) + 1)$$

were H is G induced on the union of all the bags on the path from X_i to X_j.

Proof. Consider the shortest path $(i = p_1, p_2, \ldots, p_t = j)$ from i to j in T. We will now construct a walk P from u to v in H. First, we set z as u initially and add z to P. Now, we move from bag X_{p_i} to $X_{p_{i+1}}$ by adding a shortest path from z to a vertex in $w \in (X_{p_i} \cap X_{p_{i+1}})$ to P and set $z = w$. Observe that since the length of the decomposition is bounded by ℓ, so is the length of the path we added to P. Finally, we end up in X_j and add a shortest from z to u. In total, we combined $\mathrm{dist}_T(i, j) + 1$ paths of length ℓ with overlapping endpoints. Hence, the length of P is bounded by $\ell(\mathrm{dist}_T(i, j) + 1)$.

□

Lemma 21.11. *Let G be a graph, b a positive integer, $\mathcal{T} = (T, \mathcal{X})$ a non-redundant tree decomposition of length ℓ and width t and $H_T \subseteq T$ a subtree with $\mathrm{ds}(H_T) > 2b\ell$. Then there is an algorithm that in $O(tn)$ time outputs a graph $H \subseteq G$ with $\mathrm{ds}(H) > b$.*

Proof. Let V_H be the union of all the bags corresponding to vertices in H_T and let $H = G[V_H]$. We output H and observe that H can indeed be computed in $O(tn)$ time. It remains to prove that $\mathrm{ds}(H) > b$. By Lemma 21.7 we know that $|V(H)| \geq |V(H_T)|$. Furthermore, by Lemma 21.10 we know that $\mathrm{diam}(H) \leq \ell(\mathrm{diam}(H_T) + 1)$. And hence

$$\mathrm{ds}(H) = \frac{|V(H)| - 1}{\mathrm{diam}(H)} \geq \frac{|V(H_T)| - 1}{\ell(\mathrm{diam}(H_T) + 1)} \geq \frac{|V(H_T)| - 1}{2\ell\,\mathrm{diam}(H_T)} > \frac{1}{2\ell} 2b\ell = b,$$

which completes the proof.

□

21.4 Lifting skewed Cantor combs

We will now prove that given a sufficiently large skewed Cantor comb in the tree of a tree decomposition of bounded length, we can either identify a smaller, but still sufficiently large, skewed Cantor comb or a dense subgraph in the decomposed graph. First, we will prove that any skewed Cantor comb can be transformed into a skewed Cantor comb of smaller depth such that the vertices for which the strays are attached and the end vertices are all far apart from each other. This is used to ensure that when selecting vertices from bags were strays are attached in the tree decomposition, we are selecting vertices that are of some constant distance from each other in the original graph.

Note that in a Cantor comb all vertices have degree 1, 2 or 3. Hence, all vertices of even degree have degree exactly 2. Most vertices have degree 2, the ones of degree 1 and 3 are "special" — they are leaves or branch points.

Lemma 21.12. *Let $\hat{S}_k$ be a skewed k-Cantor comb of depth k and d a non-negative integer with $d < k - 1$. Then one can in $O(dn)$ time find a skewed Cantor comb $\hat{S}_{k,k-d} \subseteq \hat{S}_k$ such that*

- *the backbone of $\hat{S}_{k,k-d}$ is a subset of the backbone of $\hat{S}_k$ and*
- *the distance between two vertices of odd degree on the backbone is at least $3 \cdot 2^d - 1$.*

Proof. Recall that by the definition of skewed Cantor combs the degree of a vertex is either one, two or three. Furthermore, for any skewed Cantor comb of depth at least two it holds that every path between two vertices of degree one goes through a vertex of degree three. And hence, since $0 \leq d < k - 1 \Rightarrow k - d \geq 2$, it is sufficient to bound the distance from vertices of degree three to vertices of odd degree on the backbone.

We prove a slightly stronger statement by induction on d. Specifically, we prove that the distance between two vertices of degree three is at least $3 \cdot 2^d$ and that the distance between a vertex of degree three and a vertex of degree one is at least $3 \cdot 2^d - 1$. For $d = 0$, the statement follows by the definition of skewed Cantor combs. Specifically, the distance between any two vertices of degree three is at least 3 and the distance between a vertex of degree three and a vertex of degree one is at least 2. For the induction step we assume that we have found an appropriate skewed Cantor comb $\hat{S}_{k,k-(d-1)}$. We then remove every stray of level 2 (the "smallest" strays) from the skewed Cantor comb. We observe that this new skewed Cantor comb is of depth $d - k$ and its backbone is unchanged. The path between two vertices of degree three goes through a vertex that used to be of degree three (before we removed the strays) and hence the distance is at least $3 \cdot 2^{d-1} + 3 \cdot 2^{d-1} = 3 \cdot 2^d$. Similarly, the path between a vertex of degree three and a vertex of degree one goes through a vertex that used to be of degree three. It follows that the distance between the two is bounded by $3 \cdot 2^{d-1} + 3 \cdot 2^{d-1} - 1 = 3 \cdot 2^d - 1$.

Observe that removing the strays can be done in $O(n)$ time and that this process is repeated at most d times. Hence we can find the skewed Cantor comb in $O(dn)$ time.

□

Next we will prove that if two vertices are picked from two different bags that are far apart in the decomposition, then this distance has some carry over to the decomposed graph and the distance between the sampled vertices. We will use this to ensure that when building strays of the skewed Cantor comb in the graph, using strays from the skewed Cantor comb in the decomposition, these newly built strays are indeed long enough to be used.

Observation 21.13. *Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G. For every two bags X_i and X_j such that $\operatorname{dist}_T(i, j) \geq \operatorname{freq}(\mathcal{T})$ it holds that X_i and X_j are disjoint.*

Lemma 21.14. *Let G be a graph and $\mathcal{T} = (T, \mathcal{X})$ a tree decomposition of G. For every two vertices $u \in X_i$ and $v \in X_j$ it holds that $\operatorname{dist}_G(u, v) \geq \lfloor \operatorname{dist}_T(i, j)/\operatorname{freq}(\mathcal{T}) \rfloor$.*

Proof. Let P_T be the shortest path from i to j in T and P a shortest path from u to v in G. It is well-known that for every $k \in V(P_T)$ it holds that X_k separates u from v in G. It follows that P intersects X_k for every $k \in V(P_T)$. We prove the statement by induction on $d = \lfloor \operatorname{dist}_T(i, j)/\operatorname{freq}(\mathcal{T}) \rfloor$. The statement trivially holds for $d = 0$. For the induction step, we assume that the statement holds for $d - 1$. Now, let k be the vertex on P_T that has distance exactly $(d - 1) \cdot \operatorname{freq}(\mathcal{T})$ and let x be a vertex in $X_k \cap V(P)$. By assumption if holds that $\operatorname{dist}_G(u, x) \geq \operatorname{dist}_T(i, k)/\operatorname{freq}(\mathcal{T}) = (d - 1)$. Now, since $\operatorname{dist}_T(k, j) \geq \operatorname{freq}(\mathcal{T})$ it follows by Observation 21.13 that X_k and X_j are disjoint. And hence, $\operatorname{dist}_G(x, v) \geq 1$, implying that $\operatorname{dist}_G(u, v) = \operatorname{dist}_G(u, x) + \operatorname{dist}_G(x, v) \geq (d - 1) + 1 = d$.

□

We are now ready to prove that given a skewed Cantor comb in the decomposition, we can build a smaller skewed Cantor comb in the graph.

Lemma 21.15. *Let G be a graph, b a positive integer, $\mathcal{T} = (T, \mathcal{X})$ a non-redundant tree decomposition of length ℓ and width t and $S \subseteq T$ a skewed k-Cantor comb of depth k for $k = 8b^2\ell^2$. There is an algorithm that in $O(tn^2)$ time either outputs*

- *a subgraph $H \subseteq G$ such that $\operatorname{ds}(H) > b$ or*
- *a skewed Cantor comb $\hat{S}_{b+1} \subseteq G$.*

Proof. First, we apply Lemma 21.9 and either conclude that $\operatorname{freq}(\mathcal{T}) \leq 2b\ell$ or we output a subgraph of G of density more than b. From now on we assume that the first case applied. We then apply Lemma 21.12 to obtain a skewed Cantor comb $S' = \hat{S}_{k,k-d} \subseteq S$ with $d = \operatorname{freq}(\mathcal{T})$. It follows that the distance between two vertices of odd degree in S' is at least $3 \cdot 2^{\operatorname{freq}(\mathcal{T})} - 1 \geq 3\operatorname{freq}(\mathcal{T})$.

Let $B' = (b'_1, \dots, b'_q)$ be the backbone of S' and x a vertex in $X_{b'_1} \setminus X_{b'_2}$. Let b'_i be the degree three vertex closest to b'_1 and s'_i the leaf vertex of the stray attached to b'_i. Let $z_i \in V(G)$ be a vertex that is contained only in the bag $X_{s'_i}$. Observe that such a vertex must exist since s'_i is a leaf and $\mathcal{T}$ is non-redundant. Then, let

H be a path from x to z_i consisting of only vertices from bags on the path from $X_{b'_1}$ to $X_{s'_i}$.

Let y_i be a vertex in $V(H) \cap X_{b'_i}$. Then, let b'_j be the degree three vertex closest to b'_i in T with $i < j$ and s'_j the leaf of the stray attached to b'_j. Let z_j be a vertex that is contained only in $X_{s'_j}$ and let P' be a shortest path from y_i to z_j consisting of only vertices from bags on the path from $X_{b'_i}$ to $X_{s'_j}$. Observe that the union of H and P' might very well contain cycles. Let y'_i be the last vertex on P' that is also contained in H and observe that $y'_i \in X_{b'_i}$. We delete all vertices prior to y'_i from P' and add P' to H. Observe that H is a caterpillar. We continue this procedure as long as we can find a new vertex of degree three with a higher index. Finally, we let y be a vertex in $X_{b'_q} \setminus X_{b'_{q-1}}$ and find a path from H to y that we also add to H, removing cycles in the same manner as above.

We claim that H is a S_{k_1,k_2} with $k_1 \geq b+1$ and $k_2 \geq 8b^2\ell^2 - 2b\ell$ and hence contains a S_{b+1} as a subgraph. We identify and output this very subgraph. Observe that H can be built and the subgraph identified in $O(tn^2)$ time.

Observe that H is indeed a caterpillar and let B be the backbone of H. The bound for k_2 follows immediately from our application of Lemma 21.12. First, we will prove that the vertices of odd degree on B, or in other words the leaves and the vertices with a stray attached, are sufficiently spread out. Then, we will finish the proof by arguing that the strays are sufficiently long. Recall that the distance between two vertices of odd degree in B' is at least $3\text{freq}(\mathcal{T})$. Observe that by construction the leaves in B originates from bags that are leaves in B'. And similarly, that the vertices of B that are of degree three originates from bags of degree three in B'. It follows, by Lemma 21.14 that the distance between vertices of odd degree in H is at least 3. Which is sufficient for our purposes.

Let Q' be a stray in S' and Q the corresponding stray in H. By construction the leaf of Q is contained in the leaf bag of Q' and the neighbor of Q is contained in the neighbor bag of Q'. By applying Lemma 21.14 it follows that Q consists of at least $\lfloor (|Q'|/\text{freq}(\mathcal{T}) \rfloor \geq \lfloor |Q'|/2b\ell \rfloor$ vertices. Let d_Q be the distance d from the definition of skewed Cantor combs for the one centered around Q in H and similarly $d_{Q'}$ for the one centered around Q' in S'. Consider a vertex v of the backbone of the skewed Cantor comb centered around Q that is of maximum distance away from Q. Or in other words, $\text{dist}_H(N_H(Q), v) = d_Q$. By construction, the bag closest to $N_T(Q')$ containing v is at most distance $d_{Q'} + 1$ away from $N_T(Q')$. It follows by Lemma 21.10 that $d_Q \leq \ell(d_{Q'} + 2)$. And hence, we finish the proof with the following calculation

$$|V(Q)| \geq \left\lfloor \frac{|V(Q')|}{2b\ell} \right\rfloor \geq \left\lfloor \frac{2 \cdot 8b^2\ell^2 d_{Q'}}{2b\ell} \right\rfloor = 8b\ell d_{Q'} \geq 4b\ell(d_{Q'}+2) \geq 4bd_Q \geq 2(b+1)d_Q$$

using that $k \geq 2$ and hence that $d_{Q'} \geq 2$. □

21.5 Lifting obstructions of high pathwidth

We will now prove that the final case that demonstrates that a tree has high bandwidth, namely having high pathwidth can be lifted from a decomposition to the decomposed graph. More specifically, we will prove that given that the decomposition tree is of high pathwidth, we can obtain a subtree of the graph that is also of high pathwidth. The proof of this builds upon the classification of trees of high pathwidth by Ellis et al. [EST94] that is restated as Lemma 21.17.

Definition 21.16. For a tree T and integer p we say that a vertex $v \in V(T)$ is *p-forcing* if there are at least three components in $T - v$ of pathwidth at least $p - 1$.

Lemma 21.17 (Ellis et al. [EST94])**.** *Let T be a tree and p an integer. It then holds that $\mathrm{pw}(T) \geq p$ if and only if there is a p-forcing vertex in T.*

Theorem 40 (Ellis et al. [EST94])**.** *Given a tree T and an integer p there is an algorithm that in $O(n)$ time decides if $\mathrm{pw}(T) \leq p$.*

Observation 21.18. *Given a tree T and an integer p such that $\mathrm{pw}(T) \geq p$, there is an algorithm that in $O(n^2)$ time outputs a p-forcing vertex in T.*

Proof. This is achieved by doing a linear search over all candidates for v and then for each component of $T - v$ apply the algorithm from Theorem 40. □

Note that it might very well be possible to identify forcing vertices faster than above. However, this observation is sufficient for our purposes and is a very easy application of existing knowledge.

Lemma 21.19. *There is an algorithm that given a graph G, a positive integer b and a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ of G of length ℓ, width t and frequency f such that $\mathrm{pw}(T) \geq bf$ in time $O(bfn^2)$ outputs a subtree $T_G \subseteq G$ such that $\mathrm{pw}(T_G) \geq b$.*

Proof. We will now describe a recursive algorithm that adheres to the specifications in the statement. If $b = 0$, we return an arbitrary vertex in G. Otherwise, let $p = bf$ and find a p-forcing vertex v in T using the algorithm from Observation 21.18. Let T^1, T^2 and T^3 be three subtrees of $T - v$ of pathwidth at least $p - 1$. For every T^i we do the following: Let $T^i_1 = T^i$ and find a $(p-1)$-forcing vertex v_1 in T^i_1. Then, let T^i_2 be a subtree of $T^i_1 - v_1$ of pathwidth at least $p - 2$ that maximizes its distance to v. Let v_2 be a $(p-2)$-forcing vertex of T^i_2 and observe that $1 \leq \mathrm{dist}_T(v, v_1) < \mathrm{dist}_T(v, v_2)$. We continue this iterative process until we obtain v_{p-f-1} and T^i_{p-f}. Observe that $\mathrm{dist}_T(v, v_{p-f-1}) \geq f - 1$ and hence $\mathrm{dist}(v, T^i_{p-f}) \geq f$. Furthermore, observe that $\mathrm{pw}(T^i_{p-f}) \geq p - f$. We let G^i be G induced on the union of all the bags corresponding to vertices in T^i_{p-f} and $\mathcal{T}^i = (T^i_{p-f}, \mathcal{X}')$, the tree decomposition of G^i when restricting $\mathcal{T}$ to T^i_{p-f}. Due to Observation 21.13 it follows that $V(G^i)$ and X_v are disjoint. We now apply the algorithm recursively on $(G^i, b^i = b-1, \mathcal{T}^i)$ to obtain a subtree $T_G^i \subseteq G$ such that $\mathrm{pw}(T_G^i) \geq b-1$.

In the end we have computed the subtrees T_G^1, T_G^2 and T_G^3 of G, all of pathwidth at least $p-f$. Since for every i it holds that $V(G^i)$ and X_v are disjoint and v separates T_{p-f}^i from T_{p-f}^j for $i \neq j$, it holds that T_G^1, T_G^2 and T_G^3 are vertex disjoint subgraphs of G. Let T_G be the disjoint union of $G[X_v], T_G^1, T_G^2$ and T_G^3 together with shortest paths from T_G^1, T_G^2 and T_G^3 to X_v in G. Observe that T_G might not be tree, but that $T_G - X_v$ is a forest. We exhaustively remove edges and vertices in $G[X_v]$ from T_G as long as T_G remains connected and return the resulting T_G.

We prove the correctness of the algorithm by induction. First, we observe that the pathwidth of G drops by f for every recursive call and hence the pathwidth of G stays at least bf as b drops by one for each recursive call. If $b = 0$, the correctness trivially holds. For the induction step we assume the algorithm to be correct for $b-1$ and hence that the pathwidth of T_G^i is at least $b-1$ for all i. It remains to prove that $\mathrm{pw}(T_G)$ is indeed at least b. Consider the tree $T' = T_G[N[X_v] \cap T_G]$. By construction, T' is a tree with exactly three leaves. And hence, it is well-known that T' contains exactly one vertex v' of degree three and that all other vertices are of degree at most two. Furthermore, v' separates all the leaves. We immediately observe that $T_G - v'$ has exactly three components of pathwidth at least $b-1$ and hence, by Lemma 21.17 the pathwidth of T_G is at least b.

Finally, we will argue about the running time of the algorithm. First, we observe that inside a recursive call on the graph G_i with $n_i = |V(G_i)|$ and $m_i = |E(G_i)|$ the algorithm spends $O(fn_i^2)$ time identifying v_{p-f} and T_{p-f}^i. After this it spends $O(n_i + m_i)$ time building the initial T_G and $O(w^2)$ time trimming it down to a tree. This yields a running time of $O(fn_i^2)$ inside a specific function call. In total, for all the recursive calls with $b = x$ we spend $\sum O(fn_i^2) = O(fn^2)$ due to $\sum n_i = n$. The recursion depth is bounded by $b+1$ and hence we end up with a total running time of $O(bfn^2)$.

□

21.6 Algorithm and correctness

We now have the necessary theory to describe the algorithm for BANDWIDTH on graphs of bounded treelength. The algorithm is surprisingly simplistic. First, we obtain a tree decomposition of approximate induced treelength. Then we verify that the width, frequency and pathwidth of the decomposition tree is indeed low and if not build an obstruction. If all checks are good, we apply our algorithm for trees directly on the decomposition tree. And then we either plug the content of the bags into the ordering of the decomposition tree or we output an obstruction.

Lemma 21.20. *Given a connected graph G and positive integers b and ℓ such that $\mathrm{tl}(G) \leq \ell$, it holds that `TreeLengthAlg` in time $n^{O(1)}$ either returns*

- *a $(54b\ell)^{49b^2\ell}$-bandwidth ordering of G,*
- *a subgraph $H \subseteq G$ such that $\mathrm{ds}(H) > b$,*
- *a skewed Cantor comb $\hat{S}_{b+1} \subseteq G$ or*
- *a subtree $T_H \subseteq G$ such that $\mathrm{pw}(T_H) > b$.*

Proof. We obtain the tree decomposition $\mathcal{T}$ of induced treelength at most 9ℓ by the algorithm of lemma 21.5 in $n^{O(1)}$ time. Furthermore, by Proposition 2.2 we can assume the decomposition to be non-redundant. Then, by Lemmata 21.6 and 21.9 we can obtain $H \subseteq G$ of high density if either the width or the frequency of $\mathcal{T}$ is large in $O(b\ell n)$ time. Using the algorithm from Theorem 40 we can check if $\mathrm{pw}(T) \leq 18(b+1)b\ell$ in $O(n)$ time. If $\mathrm{pw}(T) > 18b(b+1)\ell$ it follows by Lemma 21.19 that we can obtain a subtree $T_G \subseteq G$ such that $\mathrm{pw}(T_G) \geq b+1$ in $O(\ell b^2 n^2)$ time. If `TreeAlg` returns an obstruction it follows from Lemma 20.8 that either we get a $H \subseteq T$ with $\mathrm{ds}(H) > 648b^2\ell^2$ or a skewed Cantor comb $\hat{S}_{648b^2\ell^2+1} \subseteq T$. If the first case applies it follows from Lemma 21.11 that we can obtain a subgraph $H \subseteq G$ such that $\mathrm{ds}(H) > 648b^2\ell^2/9\ell = 72b^2\ell > b$ in time $O(b\ell n)$. If the later applies if follows from Lemma 21.15 that we in time $O(b\ell n^2)$ can obtain either a subgraph $H \subseteq G$ with $\mathrm{ds}(H) > b$ or a skewed Cantor comb $\hat{S}_{b+1} \subseteq G$.

Note that no matter the result of applying `TreeAlg` it terminates in time $O((b^2\ell^2)^2 n^3) = O(b^4\ell^4 n^3)$ by Lemma 20.8. It remains to consider the case when `TreeAlg` returns an ordering α_T of T of bandwidth at most

$$(7680(648b^2\ell^2)^6)^{2b(b+1)\ell} \leq (6 \cdot 10^{20} \cdot b^{12}\ell^{12})^{4b^2\ell} \leq (54b\ell)^{48b^2\ell}.$$

It is clear that all vertices are embedded by α exactly once and hence it is a valid ordering of $V(G)$. Now, consider an edge uv in G and let i be the integer for which u gets embedded by α and j the integer for which v gets embedded. Observe that $u \in X_i$ and $v \in X_j$. Furthermore, since $\mathcal{T}$ is a tree decomposition there is an X_k such that $u, v \in X_k$. It follows by Observation 21.13 that $\mathrm{dist}_T(i,k) < \mathrm{freq}(\mathcal{T}) \leq 18b\ell$ and similarly that $\mathrm{dist}_T(j,k) < \mathrm{freq}(\mathcal{T}) \leq 18b\ell$. It follows by the triangle inequality that $\mathrm{dist}_T(i,j) \leq 36b\ell$. And hence, $|\alpha_T(i) - \alpha_T(j)| \leq 36b\ell \cdot \mathrm{bw}(T, \alpha_T)$. By construction of α it holds that $|\alpha(u) - \alpha(v)| \leq w(\mathcal{T})(|\alpha_T(i) - \alpha_T(j)| + 1)$ and hence

$$\begin{aligned}
|\alpha(u) - \alpha(v)| &\leq w(\mathcal{T})(|\alpha_T(i) - \alpha_T(j)| + 1) \\
&\leq (9b\ell + 1)(36b\ell \cdot \mathrm{bw}(T, \alpha_T) + 1) \\
&\leq 370b^2\ell^2 \cdot \mathrm{bw}(T, \alpha_T) \\
&\leq 370b^2\ell^2 \cdot (54b\ell)^{48b^2\ell} \\
&\leq (54b\ell)^{49b^2\ell}.
\end{aligned}$$

Since uv was an arbitrary edge from G it follows that $\mathrm{bw}(G,\alpha) \leq (54b\ell)^{49b^2\ell}$ and our proof is complete.

□

Algorithm 3: `TreeLengthAlg`

```
Input: A connected graph $G$ and positive integers integers $\ell$ and $b$ such
       that $\mathrm{tl}(T) \leq \ell$.
Output: A $(54b\ell)^{49b^2\ell}$-bandwidth ordering of $G$ or an obstruction $H \subseteq G$.

Compute a tree decomposition $\mathcal{T} = (T, \mathcal{X})$ of $G$ of length at most $9\ell$.
if $w(\mathcal{T}) > 9b\ell + 1$ or $\mathrm{freq}(\mathcal{T}) > 18b\ell$ then
    Find $H \subseteq G$ such that $\mathrm{ds}(H) > b$.
    return $H$
end
if $\mathrm{pw}(T) > 18b(b+1)\ell$ then
    Find subtree $T_G \subseteq G$ such that $\mathrm{pw}(T_G) > b$.
    return $T_G$
end
Let $\alpha_T = \mathtt{TreeAlg}(T, 648b^2\ell^2)$.
if an obstruction $H_T \subseteq T$ is returned then
    Build obstruction $H \subseteq G$ from $H_T$.
    return $H$
end
Let $\alpha$ be an empty ordering.
for $i \in [1, |V(T)|]$ do
    Let $Y$ be the vertices in $X_{\alpha_T^{-1}(i)}$ that are not embedded by $\alpha$.
    Let $Y$ populate the left-most available positions of $\alpha$.
end
return $\alpha$
```

Theorem 41. *Given a graph G and positive integers b and ℓ such that $\mathrm{tl}(G) \leq \ell$, there exists an algorithm that either outputs a $(b\ell)^{O(b^2\ell)}$-bandwidth ordering of G or correctly concludes that $\mathrm{bw}(G) > b$ in polynomial time.*

Proof. First, we apply Lemma 21.20 for every connected component of G. If the algorithm returns bandwidth orderings, we concatenate them and output this new ordering. In the case we obtain a subgraph $H \subseteq G$ such that $\mathrm{ds}(H) \geq b$, it follows from Proposition 18.2 that $\mathrm{bw}(G) > b$. In the case we obtain a subtree $T_G \subseteq G$ such that $\mathrm{pw}(T_G) > b$ it follows immediately that $\mathrm{pw}(G) > b$ and hence again by Proposition 18.2 we conclude that $\mathrm{bw}(G) > b$. In the case we obtain a skewed Cantor comb $\hat{S}_{b+1} \subseteq G$ it follows from Lemma 19.2 that $\mathrm{bw}(G) > b$.

□

Corollary 21.21. *Given a chordal graph G and a positive integer b, there is an algorithm that in polynomial time either outputs a $b^{O(b^2)}$-bandwidth ordering or correctly concludes that $\mathrm{bw}(G) > b$.*

Theorem 42. *Given a graph G and positive integers b and ℓ such that* $\mathrm{tl}(G) \leq \ell$ *it holds that either*

- $\mathrm{bw}(G) \leq (54b\ell)^{49b^2\ell}$,
- $\rho(G) > b$,
- *G contains a skewed Cantor comb $\hat{S}_{b+1}$ as a subgraph or*
- *G contains a tree T_G as a subgraph such that* $\mathrm{pw}(T_G) > b$.

Proof. The result follows from applying Lemma 21.20 on every connected component of G. □

Chapter 22

Lower bounds

In this chapter we will give a reduction from Even Clique to Bandwidth with a linear blowup of the "solution size". For the rest of this section we will refer to the solution size of the instance of Even Clique as k and the bandwidth of the resulting Bandwidth instance as b. From this reduction, we conclude that Bandwidth is W[1]-hard when parameterized by b. And that Bandwidth cannot be solved in $f(b)n^{o(b)}$ time for any computable function f, assuming ETH. Both of the results above hold even when restricted to trees of pathwidth at most 2. A consequence of the later result is that there is no significant improvement over the dynamic programming algorithm by Saxe [Sax80], even for trees of pathwidth at most 2.

22.1 A gentle introduction to the reduction

We will now give an informal description of the reduction. We hope it will provide the reader with some intuition of why p-Bandwidth is as hard as it is. As already mentioned, the reduction will be from instances (G, k) of p-Even Clique to instances (T, b) of p-Bandwidth. To obtain the results of Theorem 44 we must first of all ensure that (G, k) is a yes-instance if and only if (T, b) is a yes-instance. And furthermore, we require T to be a tree of size polynomial in $|V(G)|$ and k, and that the path-width of T is at most 2. Last, for obtaining the lower bounds based on ETH we require b to be of size $O(k)$.

We start, by providing some boundaries for b-bandwidth orderings of T. Meaning that we force specific parts of T to be the leftmost and rightmost elements of every such ordering. This is done by introducing two stars with $2b$ leaves and adding a path from one of the leaves of the first star to one of the leaves of the second. The two stars will be referred to as walls and the path between them as the main path. Observe that for both of the walls, the leaves must occupy the $2b$ positions closest to the center in any b-bandwidth ordering. It follows that the main path must be within the inclusion interval of the two walls, since otherwise the main path would pass through a wall and one of its edges must be stretched too far.

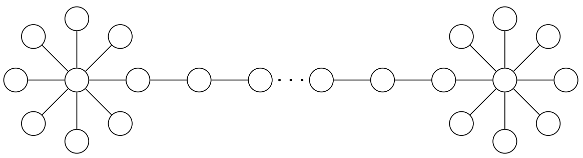


Figure 22.1: An illustration of the walls for $b = 4$.

We are now controlling the first and last vertices in any b-bandwidth ordering of the graph and hence it is time to start encoding our instance of Even Clique. To keep control, the rest of T will be attached to the main path. Before we continue, we select one of the walls and base an ordering of the reduction graph on this selection. This wall will from now on be referred to as the first wall and the other wall will be referred to as the last wall. We then attach k paths, from now on referred to as threads, to the vertex of the main path that is also a leaf of the first wall. Each thread will encode a selection of a vertex in G, and then we will check whether this set of vertices in fact forms a clique or not.

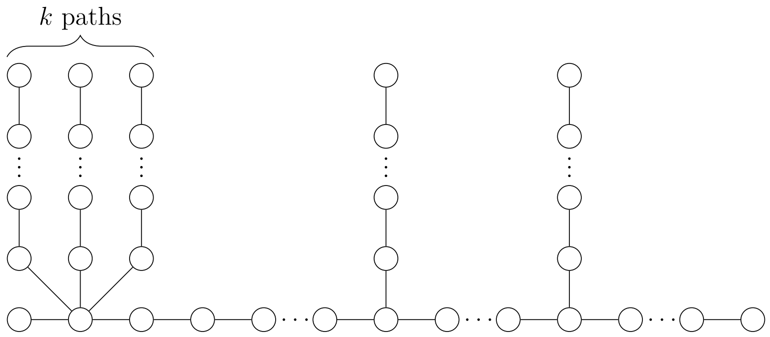


Figure 22.2: An illustration of the paths of the reduction graph.

To control how information propagates through a bandwidth ordering, we introduce gates. A k-gate is a vertex on the main path with $2(b-k-1)$ leaves attached to it, that is in addition to the two neighbours it has on the main path. The goal is to force every thread to pass through every k-gate. Then every thread will position two vertices within the positions of distance at most b away from the center of the gate. And hence there will be $2(b-k-1)+2k+2 = 2b$ vertices that have to be positioned close to the center, leaving no available room.

A hole is basically two vertices on the main path with some extra space in between. This extra space is obtained by attaching not so many leaves to the two vertices. A knot is a large star centered at one of the threads. The idea is that a knot requires so much space that it cannot be positioned close to a gate. And hence, if a subpath of the main path consists of only gates and holes, a knot that is to be positioned within the inclusion interval of this subpath must be positioned in a hole.

Before we start the process of embedding gadgets on the main path and the threads, we need a guarantee ensuring that any resulting bandwidth ordering will behave nicely. Consider the following situation, we have a graph T and a b-bandwidth ordering α of T. T contains $k+1$ disjoint paths, one of the paths P being of length l such that all the other paths are passing through P in α. In addition there is a set of $(l-1)(b-k-1)$ vertices X disjoint from all the paths, such that the image of X is contained in the inclusions interval of P. Lemma 22.3 then tells us that that P must be stretched with respect to α, meaning that the vertices of P appear in the same order in T as in α up to reversion and that the distance between two consecutive vertices is b. Furthermore, each of the paths passing through will position exactly one vertex in between any two consecutive vertices of P. As the reader probably can image, we will apply this result with the main path as P and the threads as the paths passing through. This will ensure that how and in which order the vertices appear in α is highly similar to how they are ordered in T. See Figure 22.3 for an illustration of the phenomenon.

We will now start to embed gadgets. First, we introduce three long sequences of gates on the main path. These sequences naturally partitions our graph into nine sectors. We will refer to them as the first wall, the first wasteland, the first gateland, the selector, the middle gateland, the validator, the last gateland, the last wasteland and the last wall. See Figure 22.4 for an illustration. By making the threads very long, one can force them to pass through every gate. This together with the lemma described above implies that the sectors will appear in the same order in any b-bandwidth ordering as they do in the graph up to reversion.

We aim at forcing a large set over vertices to be embedded in between the first and the last wasteland. It follows that this part of the main path will be stretched and every thread will position exactly one vertex in between every two consecutive vertices of the main path. Recall that the threads are to encode which vertices we take as our clique. This will be done by how much of the thread is positioned within the inclusion interval of the first wasteland before it starts its journey towards the last wasteland. And the job of the wastelands are exactly

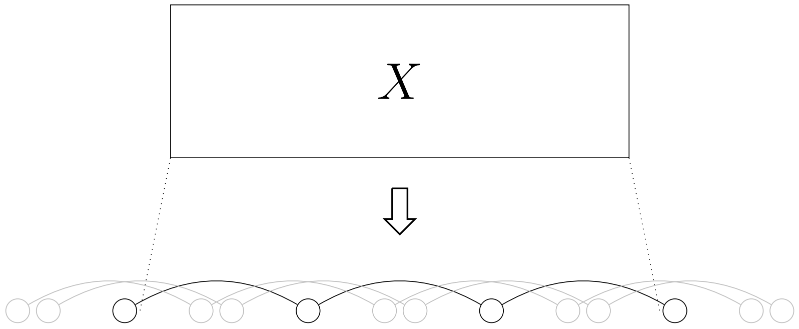


Figure 22.3: An illustration of Lemma 22.3. The black path is P and the grey are the ones passing through P.

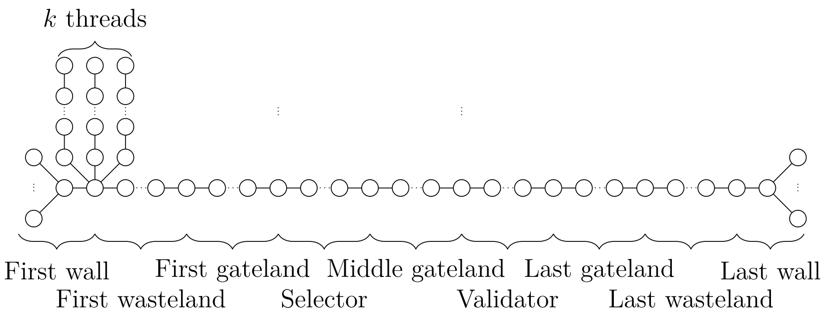


Figure 22.4: The sectors of our reduction graph.

this, to handle the slack or waste produced by different choices of vertices to form the clique.

We now describe how we enforce the selection of vertices in a manner that allows us to extract this information in a useful way in the validator. First, we order the vertices of G by labeling them with numbers from 1 to n. Basically, we want there to be a linear function describing the number of vertices positioned in the first wasteland by a single thread, given the label of the vertex this thread choose. This is obtained by embedding n holes within the selector, with a certain number of gates in between every pair of consecutive holes. Then we embed a knot on each thread. The idea is that each thread must position its knot within a hole and every hole can contain at most one knot. Which hole the knot is positioned in gives the vertex the thread selects for the clique.

We should now ensure that the selected vertices forms a clique in G. This is done by the validator. The validator is partitioned into $2n - 1$ zones. The first $n-1$ and last $n-1$ zones are referred to as neutral zones and nothing is embedded on this part of the main path. The middle zone is referred to as the validation zone. Like the selector, also the validator zone consists of n holes separated by a series of gates. Now the idea is to embed the adjacency matrix of G on the threads, row by row, in such a way that if vertex i is selected by the thread, then the part representing row number i of the matrix is positioned within the validator zone. The matrix will be represented as follows: Partition the subpath of the thread representing row i into n parts. At part number i we embed a knot. And then, for every non-neighbour j we will attach a leaf to part j. What will happen is that when the vertices are selected the corresponding holes in the validator will be filled up by knots. And then, if two vertices are not adjacent there will also be a leaf that should be positioned within the same hole as a knot. And this there will not be room for. Furthermore, if a vertex is not selected there will not be a knot in the corresponding hole, so that it can contain as many leaves as necessary. The last crucial observation is that in the neutral zone, there is room for both leaves and knot to co-exist close in the bandwidth ordering.

The observant reader might recall that we promised some large set of vertices that should be embedded within the first and the last wasteland. This will be handled by attaching paths of appropriate size right after both the first and the second gateland. By making every hole and gate within the selector and validator into $(k+1)$-holes and $(k+1)$-gates these paths can travel around in the two sectors filling up the remaining space.

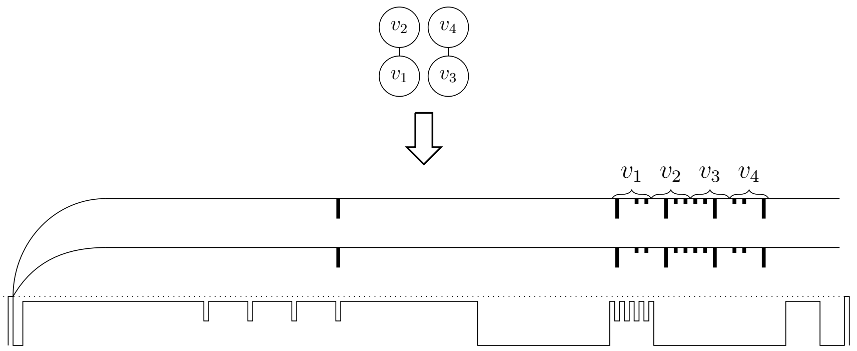


Figure 22.5: A graph (a $2K_2$) and a visualization of the output of the reduction when given this graph as input.

In Figure 22.5 we visualize the output of the reduction given a graph G that forms a $2K_2$. The bottom line illustrates the amount of space that is occupied by default at the various positions of any bandwidth ordering. And the dotted line illustrates the threshold of how much space there is available (the value of b). We see the peaks at the beginning and end that represents the walls that nothing can pass. And also the two wastelands next to the walls, with room for the threads to pile up. To the right of the first wasteland we find the selector. There are four holes, one for each of the vertices in the input graph. And at the corresponding positions of the threads there is a knot, represented by a block. Which of these holes the thread puts its knot into, decides which of the vertices in G it select to be in the clique. The reader should imagine that each thread will be embedded in an even and nice manner from the first to the last wasteland, without changing direction or similar. This ensures that the choices made in the selector propagates into the validation part of the reduction.

Next is the validator and we see the validation zone in the middle as four consecutive holes. On the threads, the adjacency matrix is embedded, row by row. We put a knot if we are at row i, position i. And otherwise, either a leaf or nothing, represented by the smaller blocks. Specifically, it uv is not an edge in G we do attach a leaf on the position representing v at the row of u and vice versa. Otherwise, we attach nothing. Observe that when a thread selects a specific vertex, the row corresponding to this vertex is positioned within the validation zone. We see that a single knot (big block) can fit within a hole, but then there is no room for an additional leaf (small block). However, two leaves (smaller blocks) easily fit within a hole. The holes of the validation zone that are filled by knots are exactly the vertices selected by the threads. And if two vertices are selected that are not adjacent, one would have to position an addition leaf within an already filled hole, which is not possible. For the vertices not selected there is sufficient room, independent of their adjacencies with the selected vertices.

Observe that there is available room both in the selector. Specifically, in the holes of the vertices that were not selected to be in the clique. And also, possibly some room available within the validation zone depending on the number of non-adjacencies that are pulled into this zone. Last, there is a lot of empty space in all of the neutral zones, ensuring that for the rows of the vertices that are not selected to be in clique, it is possible to position a leaf on top of a knot. All of this available room is filled up by the two fillers of the reduction, ensuring that the threads are not piling up at various locations. We are now done with the informal introduction and for the details we refer to the rest of this section.

22.2 Tools

In this section we give some definitions and results for bandwidth which are crucial for our reduction.

Lemma 22.1. *Let (T, b) be an instance of p-Bandwidth and $\hat{P}_2, P^1, \ldots, P^k$ be $k+1$ disjoint subpaths of T. Given a b-bandwidth ordering α such that $P^1, \ldots, P^k$ pass through $\hat{P}_2$ and there is a set of vertices X disjoint from $\hat{P}_2, P^1, \ldots, P^k$ such that $|X| \geq b-k-1$ and $\alpha(X) \subseteq I(\hat{P}_2)$, then $|\alpha(P^i) \cap I(\hat{P}_2)| = 1$ for every i.*

Proof. Let $\hat{P}_2 = (u, v)$ and assume without loss of generality that $\alpha(u) < \alpha(v)$. From $|I(\hat{P}_2)| \leq b+1$ and

$$\begin{aligned}
|I(\hat{P}_2)| &= |I(\hat{P}_2) \cap \alpha(V(T))| \\
&\geq |I(\hat{P}_2) \cap \alpha(\bigcup P^i \cup X \cup \hat{P}_2)| \\
&= |I(\hat{P}_2) \cap \alpha(\bigcup P^i)| + |I(\hat{P}_2) \cap \alpha(X)| + |I(\hat{P}_2) \cap \alpha(\hat{P}_2)| \\
&\geq |I(\hat{P}_2) \cap \alpha(\bigcup P^i)| + b - k + 1
\end{aligned}$$

it follows that $|I(\hat{P}_2) \cap \alpha(\bigcup P^i)| \leq k$.

Assume for a contradiction that there is a j_1 such that $|\alpha(P^{j_1}) \cap I(\hat{P}_2)| \neq 1$. Then, since $|I(\hat{P}_2) \cap \alpha(\bigcup P^i)| \leq k$ it follows that there is a j_2 such that $|\alpha(P^{j_2}) \cap I(\hat{P}_2)| = 0$. For a path P^i let (v_l^i, v_r^i) maximize $\alpha(v_l^i)$ among the edges in P^i with $\alpha(v_l^i) < \alpha(u)$ and $\alpha(v) < \alpha(v_r^i)$. Let P^j be the path minimizing $\alpha(v_l^j)$ among all paths P^i such that $|\alpha(P^i) \cap I(\hat{P}_2)| = 0$. It follows that for every path P^i either $|\alpha(P^i) \cap I(\hat{P}_2)| \geq 1$ or $|\alpha(P^i) \cap I(v_l^j, u)| \geq 1$. Hence for each i it holds that $|I(v_l^j, v_r^j) \cap \alpha(P^i)| \geq 1$. Furthermore, observe that $|I(v_l^j, v_r^j) \cap \alpha(P^j)| \geq 2$. It follows that

$$\begin{aligned}
|I(v_l^j, v_r^j)| &\geq |I(v_l^j, v_r^j) \cap \alpha(X)| + |I(v_l^j, v_r^j) \cap \alpha(\hat{P}_2)| + |I(v_l^j, v_r^j) \cap \alpha(\bigcup P^i)| \\
&\geq (b-k-1) + 2 + (k+1) \\
&\geq b+2
\end{aligned}$$

Observe that X, $\hat{P}_2$ and $\bigcup P^i$ are disjoint and hence the first line above is valid. Since (v_l^j, v_r^j) is an edge in T and $|I(v_l^j, v_r^j)| \geq b + 2$ we have a contradiction to α being a b-bandwidth ordering and hence our proof is complete. $\square$

Corollary 22.2. *Let (T, b) be an instance of p-*Bandwidth *and $\hat{P}_2, P^1, \ldots, P^k$ be $k+1$ disjoint subpaths of T. Given a b-bandwidth ordering α such that $P^1, \ldots, P^k$ pass through $\hat{P}_2$ and there is a set of vertices X disjoint from $\hat{P}_2, P^1, \ldots, P^k$ such that $|X| \geq b - k - 1$ and $\alpha(X) \subseteq I(\hat{P}_2)$, then $|X| = b - k - 1$.*

Proof. Assume for a contradiction that $|X| \geq b - k$. Apply Lemma 22.1 to obtain $|\alpha(P^i) \cap I(\hat{P}_2)| = 1$ for every i. It follows that

$$\begin{aligned}
|I(\hat{P}_2)| &\geq |I(\hat{P}_2) \cap \alpha(X \cup \hat{P}_2 \cup \bigcup P^i)| \\
&\geq |I(\hat{P}_2) \cap \alpha(X)| + |I(\hat{P}_2) \cap \alpha(\hat{P}_2)| + |I(\hat{P}_2) \cap \alpha(\bigcup P^i)| \\
&\geq (b - k) + 2 + k \\
&\geq b + 2.
\end{aligned}$$

which is a contradiction to α being a b-bandwidth ordering. $\square$

We are now ready to prove the result illustrated in Figure 22.3.

Lemma 22.3. *Let (T, b) be an instance of p-*Bandwidth *and $\hat{P}_l, P^1, \ldots, P^k$ be $k+1$ disjoint subpaths of T. Given a b-bandwidth ordering α such that $P^1, \ldots, P^k$ pass through $\hat{P}_l$ and there is a set of vertices X disjoint from $\hat{P}_l, P^1, \ldots, P^k$ such that $|X| \geq (l-1)(b-k-1)$ and $\alpha(X) \subseteq I(\hat{P}_l)$, then $\hat{P}_l$ is stretched with respect to α and $|P^i \cap I(\hat{P}_2)| = 1$ for every i and every $\hat{P}_2 \subseteq \hat{P}_l$.*

Proof. We start by proving $\alpha(v_1) < \alpha(v_2) < \cdots < \alpha(v_l)$ or $\alpha(v_l) < \cdots < \alpha(v_2) < \alpha(v_1)$. Assume otherwise for a contradiction. Then there exists three vertices v_{j-1}, v_j and v_{j+1} such that either $\max\{\alpha(v_{j-1}), \alpha(v_{j+1})\} < \alpha(v_j)$ or $\alpha(v_j) < \min\{\alpha(v_{j-1}), \alpha(v_{j+1})\}$. Since all properties of the lemma is preserved with respect to reversing α, we can assume without loss of generality that $\max\{\alpha(v_{j-1}), \alpha(v_{j+1})\} < \alpha(v_j)$. We define a function f from proper subsets of $V(\hat{P}_\ell)$ into $V(\hat{P}_\ell)$ as follows, $f(B) = v_d$ such that $d = \min\{i \mid v_i \in \hat{P}_l \setminus B \text{ and } \{v_{i-1}, v_{i+1}\} \cap B \neq \emptyset\}$. In other words, f gives you the smallest indexed vertex in the open neighbourhood of B. Notice that since $\hat{P}_l$ is connected f is a well-defined function. We will now define $a_1, \ldots, a_t$ and $B_1, \ldots, B_t$. First let $a_1 = \alpha^{-1}(\min\{\alpha(\hat{P}_l)\})$ and $B_1 = \{a_1\}$. Then we let $a_i = f(B_{i-1})$ and $B_i = I(a_1, a_i) \cap \hat{P}_l$ as long as $B_{i-1} \neq \hat{P}_l$. Observe that $B_{i-1} \subset B_i$.

First we will prove that $t < l$. Assume otherwise for a contradiction, clearly then $t = l$. It follows by the construction and our assumption that $\{a_1, \ldots, a_i\} = B_i$ for every i. And by a simple induction we get that $T[\{a_1, \ldots, a_i\}]$ is connected, since this clearly holds for $i = 1$ and for $i > 1$ observe that a_i has a neighbour in B_{i-1} by construction. Recall that j is so that $\max\{\alpha(v_{j-1}), \alpha(v_{j+1})\} < \alpha(v_j)$ and let c be so that $a_c = v_j$. Since v_j is separating v_{j-1} and v_{j+1} in $\hat{P}_l$ and $v_j \notin B_{c-1}$ it follows that $\{v_{j-1}, v_{j+1}\} \not\subseteq B_{c-1}$. Furthermore, since $\max\{\alpha(v_{j-1}), \alpha(v_{j+1})\} < \alpha(v_j)$ it holds that $\{v_{j-1}, v_j, v_{j+1}\} \subseteq B_c$. But this contradicts $\{a_1, \ldots, a_i\} = B_i$ and hence we know that $t < l$. It follows, due to the pidgin hole principle, that there is a d such that

$$|I(a_{d-1}, a_d) \cap \alpha(X)| \geq \frac{(\ell-1)(b-k-1)}{\ell-2} > b-k-1.$$

By construction there is a neighbour a' of a_d among $a_1, \ldots, a_{d-1}$. Observe that $|I(a', a_d) \cap \alpha(X)| > b-k-1$ and apply Corollary 22.2 with $\hat{P}_2 = (a', a_d)$ to obtain a contradiction. Hence we can conclude that $\alpha(v_1) < \alpha(v_2) < \cdots < \alpha(v_l)$ or $\alpha(v_l) < \cdots < \alpha(v_2) < \alpha(v_1)$.

We will now prove that $|P^i \cap I(\hat{P}_2)| = 1$ for every i and every $\hat{P}_2 \subseteq \hat{P}_l$. Observe that if there is a $\hat{P}_2$ such that $|I(\hat{P}_2) \cap \alpha(X)| \neq b-k-1$, then there is a $\hat{P}_2'$ such that $I(\hat{P}_2') \cap \alpha(X)| > b-k-1$. But this contradicts Corollary 22.2 and hence we get that $|I(\hat{P}_2) \cap \alpha(X)| = b-k-1$ for every $\hat{P}_2 \subseteq \hat{P}_l$ and then it follows directly from Lemma 22.1 that $|P^i \cap I(\hat{P}_2)| = 1$ for every $\hat{P}_2 \subseteq \hat{P}_l$. Hence

$$\begin{aligned}
|I(\hat{P}_2)| &\geq |I(\hat{P}_2) \cap \alpha(X \cup \hat{P}_2 \cup \bigcup P^i)| \\
&\geq |I(\hat{P}_2) \cap \alpha(X)| + |I(\hat{P}_2) \cap \alpha(\hat{P}_2)| + |I(\hat{P}_2) \cap \alpha(\bigcup P^i)| \\
&\geq b-k-1+2+k \\
&\geq b+1
\end{aligned}$$

and it follows that $\hat{P}_l$ is stretched with respect to α. □

Corollary 22.4. *Let* (T, b) *be an instance of* Bandwidth *and* $\hat{P}_l, P^1, \ldots, P^k$ *be* $k+1$ *disjoint subpaths of* T*. Given a* k*-bandwidth ordering* α *such that* $P^1, \ldots, P^k$ *passes through* $\hat{P}_l$ *and there is a set of vertices* X *disjoint from* $\hat{P}_l, P^1, \ldots, P^k$ *such that* $|X| \geq (l-1)(b-k-1)$ *and* $\alpha(X) \subseteq I(\hat{P}_l)$*, then* $|X| = (l-1)(b-k-1)$*.*

Proof. Assume for a contradiction that $|X| > (l-1)(b-k-1)$. Then there is a $\hat{P}_2 \subseteq \hat{P}_l$ such that $|X \cap I(\hat{P}_2)| \geq b-k$ which is a contradiction by Corollary 22.2. □

22.3 Gadgets

We will now introduce the gadgets used for the reduction. They will all be defined on paths of various lengths. And later on when we say that a gadget is embedded on some path, this means that the path referred to together with some of its neighbours is an instantiation of the gadget.

Definition 22.5. Let (T, b) be an instance of p-Bandwidth and H be a subgraph of T with a vertex labeled *in* and another vertex labeled *out*. We say that H is *functioning* in T if T contains two walls W_{in} and W_{out} such that

- W_{in}, W_{out} and H are disjoint,
- there is a path P_{in} from *in* to W_{in} avoiding $(H - in)$ and W_{out} and

- there is a path P_{out} from *out* to W_{out} avoiding $(H - out)$, W_{in} and P_{in}.

If H is functioning in T let $W_{in}(H,T), W_{out}(H,T), P_{in}(H,T)$ and $P_{out}(H,T)$ denote a witness of this.

Walls

A *wall* is a star with $2b$ leaves. The high degree vertex of a wall W will be referred to as the *center* of the wall. We will turn the endpoints of the main path into walls to control the endpoints of all valid b-bandwidth orderings. The next lemma gives us this behaviour.

Lemma 22.6. *Let (T, b) be an instance of* Bandwidth *such that T contains two disjoint walls W_1 and W_2 with centers c_1 and c_2 as subgraphs. Let H be a connected component of $T - (W_1 \cup W_2)$ connected by edges to both walls in T. Then, for every b-bandwidth ordering α of T and every vertex $v \in H$ it follows that $\alpha(v) \in I_\alpha(c_1, c_2)$.*

Proof. Assume without loss of generality that $\alpha(c_1) < \alpha(c_2)$. First, we assume for a contradiction that $\alpha(v) < \alpha(c_1)$. Let u_l be the leaf in W_1 minimizing α and u_r the leaf maximizing α. Furthermore, let P^1 be a path from v to c_2 in $T[V(H) \cup W_2]$ and $\hat{P}_3$ the path (u_l, c_1, u_r). Observe that P^1 passes through $\hat{P}_3$, since $\alpha(W_1) = [\alpha(c_1) - b, \alpha(c_1) + b]$. Let $X = V(W_1) - \hat{P}_3$ and note that $|X| = 2b - 2$. Apply Corollary 22.4 on $\hat{P}_3, P^1$ and X to obtain a contradiction, since $(3-1)(b-1-1) = 2b-4 < 2b-2 = |X|$. For $\alpha(v) > \alpha(c_2)$ we apply a symmetric argument and hence our proof is complete. □

Gates

For an integer $k \geq 0$ a *k-gate*, denoted Π_k, is a star with $2(b-k)$ leaves. The function of the k-gate will be to reduce the number of paths passing this point to at most k. The high degree vertex of the star will be referred to as the *center* of the gate. In addition one leaf will be labeled *in* and another labeled *out*.

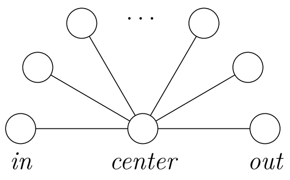


Figure 22.6: A k-gate with the special vertices marked with tags below.

Lemma 22.7. *Let (T, b) be an instance of* Bandwidth *such that T contains a gate Π_k and paths $P^1, \ldots, P^k$ as disjoint subgraphs with Π_k being functioning in $T - (\bigcup P^i)$. Given a b-bandwidth ordering α such that* $\max\{\alpha(W_{in}(\Pi_k, T - \bigcup p^i))\} <$ $\min\{\alpha(W_{out}(\Pi_k, T - \bigcup p^i))\}$ *and every path P^i passes through the gate it follows that:*

(i) $\alpha(N[\mathit{center}]) \subseteq B \subseteq \alpha(\bigcup P^i \cup N[\mathit{center}])$,

(ii) $\alpha(\mathit{in}) < \alpha(\mathit{center}) < \alpha(\mathit{out})$ *and*

(iii) $|\alpha(P^i) \cap B_l| = |\alpha(P^i) \cap B_r| = 1$ *for every* $i \in [1, k]$

where $c = \alpha(\mathit{center})$, $B = [c - b, c + b]$, $B_l = \{i \in B \mid i < c\}$ *and* $B_r = \{i \in B \mid c < i\}$.

Proof. We start by proving *(iii).* For every path P^i we know that there are $u, v \in P^i$ such that $\alpha(u) < \min \alpha(\Pi_k)$ and $\max \alpha(\Pi_k) < \alpha(v)$. Assume that $u \notin B_l$ and follow the path from u to v until you reach the first vertex u' such that $\alpha(u') \geq c - b$. Let u'' be the vertex we reached right before u'. From the definition of α it follows that $\alpha(u') - \alpha(u'') \leq b$ and hence $u' \in B_l$ and $|P^i \cap B_l| = 1$. Reverse α and apply the argument on the path from v to u to obtain $|P^i \cap B_r| = 1$.

We continue by proving *(i).* It follows directly from the fact that $\mathrm{bw}(T, \alpha) \leq b$ that $N[\mathit{center}] \subseteq B$. Since $|B \cap (\bigcup P^i \cup N[\mathit{center}])| = |B \cap \bigcup P^i| + |B \cap N[\mathit{center}]| = 2k + 2(b - k) + 1 = 2b + 1$ and $|B| = 2b + 1$, it follows that $B \subseteq \bigcup P^i \cup N[\mathit{center}]$. It remains to prove *(ii).* Observe that $\max \alpha(W_{\mathit{in}}) < \min\{\alpha(\mathit{in}), \alpha(\mathit{center})\}$ by Lemma 22.6. Assume for a contradiction that $\alpha(\mathit{in}) > \alpha(\mathit{center})$. Since $P_{\mathit{in}}(\Pi_k, T - (\bigcup P^i))$ is a path from *in* to $W_{\mathit{in}}(\Pi_k, T - (\bigcup P^i))$ and the bandwidth of α is b it follows that $|B \cap W_{\mathit{in}}(\Pi_k, T - (\bigcup P^i)| \geq 2$, but this contradicts *(i)* and hence $\alpha(\mathit{in}) < \alpha(\mathit{center})$. A symmetric argument gives us $\alpha(\mathit{center}) < \alpha(\mathit{out})$ and our proof is complete. □

Knots and Holes

Assuming $b \geq 2k + 14$ and b to be dividable by 4 we give the following two definitions. A *k-knot* is a path $P = (\mathit{first}, \mathit{center}, \mathit{last})$ with $\frac{3}{2}b - k - 1$ leaves attached to *center*. A *k-hole* consists of a path $P = (\mathit{in}, \mathit{in\ center}, \mathit{out\ center}, \mathit{out})$ with $\frac{3}{4}b - k - 1$ leaves attached to both *in center* and *out center*.

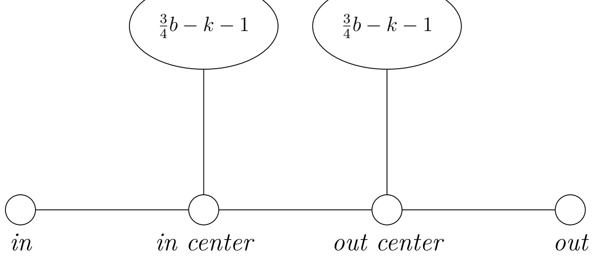


Figure 22.7: A hole. The ellipse shaped vertices represent some number of leaves.

Lemma 22.8. *Let* (T, b) *be an instance of* p-Bandwidth *such that* T *contains the following disjoint subgraphs; a k-hole* H *and paths* $P^1, \ldots, P^k$ *with a k-knot* K *embedded on one of the paths. Furthermore, let* H *be functioning in*

$T - (\bigcup P^i)$. *Given a b-bandwidth ordering α such that $P^1, \ldots, P^k$ passes though H,* $\max\{\alpha(W_{in}(\Pi_k, T - \bigcup p^i))\} < \min\{\alpha(W_{out}(\Pi_k, T - \bigcup p^i))\}$ *and $I(K \cup H) \subseteq I(in, out)$ it holds that*

(i) $\alpha(in) < \alpha(in\ center) < \alpha(out\ center) < \alpha(out)$*,*

(ii) $|I(\hat{P}_2) \cap \alpha(P^i)| = 1$ *for every i and every* $\hat{P}_2 \subset (in, in\ center, out\ center, out)$ *and*

(iii) $\alpha(in\ center) < \alpha(center) < \alpha(out\ center)$*.*

Proof. First we prove the correctness of statements *(i)* and *(ii)*. Let $\hat{P}_4 = (in, in\ center, out\ center, out)$ and let X_c, X_i and X_o be the set of leaves attached to *center*, *in center* and *out center* respectively. Apply Lemma 22.3 with $X = X_i \cup X_c \cup X_o$ to obtain *(ii)* and either $\alpha(in) < \alpha(in\ center) < \alpha(out\ center) < \alpha(out)$ or $\alpha(out) < \alpha(out\ center) < \alpha(in\ center) < \alpha(in)$ since $|X| = 2\left(\frac{3}{4}b - k - 1\right) + \frac{3}{2}b - k - 1 = (4-1)(b-k-1)$. Assume for a contradiction that $\alpha(out) < \alpha(out\ center) < \alpha(in\ center) < \alpha(in)$. Then there is a vertex $v \in P_{in}(H, T - (\bigcup P^i)) \cap \alpha^{-1}(I(H)) \setminus \{in\}$. Apply Corollary 22.4 with $X = X_i \cup X_c \cup X_o \cup \{v\}$ to get a contradiction and hence *(i)* holds.

It remains to prove *(iii)*. Assume for a contradiction that $\alpha(center)$ is not in $I(in\ center, out\ center)$. First, we consider the case when $\alpha(center)$ is in $I(in, in\ center)$. It follows from Lemma 22.3 that $\hat{P}_4$ is stretched and hence $X_i \cup X_c \subseteq I(\hat{P}_3)$ for $\hat{P}_3 = (in, in\ center, out\ center)$. Apply Corollary 22.4 with $X = X_i \cup X_c$ to obtain a contradiction since $|X| = \frac{3}{4}b - k - 1 + \frac{3}{2}b - k - 1 = \frac{9}{4}b - 2k - 2 > (3-1)(b-k-1)$. The case when $\alpha(center) \in I(out\ center, out)$ follows by the same kind of argument.

□

22.4 The Reduction

We now give a reduction from an instance (G, k) of Even Clique to an instance (T, b) of Bandwidth. The correctness and implications will be given in the two following sections. Recall that the resulting instance T can be divided into eleven parts. Nine of them lie on the main path and will in the future be referred to as the sectors of the main path. The nine sectors are the first wall, the first wasteland, the first gateland, the selector, the middle gateland, the validator, the last gateland, the last wasteland and the last wall. The two other components will be referred to as threads and fillers. Each of the components have a specific purpose with respect to how a b-bandwidth ordering can look like. The walls will force everything else to be positioned within them. The threads are k paths attached to the first wasteland and each of them represents a vertex in the supposed clique in G. To encode how G looks like we attach leaves to the threads, which will be referred to as the dangelments of the threads. How much of a thread that is in the inclusion interval of the first wasteland decides which vertex in G this thread represents.

To propagate this information the threads are made so long that they will have to enter the inclusion interval of the last wasteland. The selectors job is to make sure that the decisions made by the threads are unique and valid. The validator will verify that the selected vertices in fact is a clique. And the fillers and the gatelands will control how information propagates between the other components.

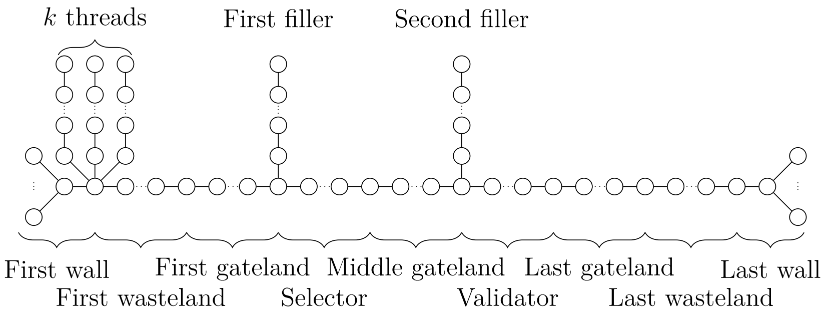


Figure 22.8: A subgraph of T with the components marked.

When discussing vertices and subgraphs of T we will apply an ordering based on the distance from the center of the first wall, the leftmost wall in Figure 22.8. We will say that a vertex u comes before a vertex v if u is closer to the center of the first wall than v. Furthermore, we will label the vertices of the main path $u_1, u_2, \dots$ with u_i being before u_{i+1}. For subgraphs, we will compare the minimized distance over all vertices in each subgraph. To complete our construction we need an ordering of the vertices of G, we therefore let $V(G) = \left\{v_1, \dots, v_{|V(G)|}\right\}$. Given an instance of Even Clique we set $b = 4k + 16$ and construct T according to the description that follows, to produce an instance (T, b) of Bandwidth.

The First Wall, Wasteland and Gateland

To ensure enough space for the gadgets in the validator we introduce the *pull-factor* p, which will correspond to the distance from the *in* vertex of a hole in the selector to the *in* vertex of the next hole. The pull-factor is $4n + 3$ in our reduction, but will for convenience mostly be referred to as p.

The first sector we will embed is the first wall. This is done by turning u_1 into the center of a wall by attaching leaves to it. Second comes the first wasteland. This is done by attaching nothing to the vertices u_2 until u_{m_1} for $m_1 = pnk + 2$. Note that u_2 is the vertex for which the threads are connected. After this we embed bm_1 consecutive k-gates from u_{m_1} to $u_{(2b+1)m_1}$ to create the first gateland. This is done in such a way that the *in* vertex of the i'th gate is the *out* vertex of the $i - 1$'th gate.

The Selector

The selector will control the choices done by the threads. The idea is to let the selector have $|V(G)|$ sparse intervals, namely holes, and let each of the threads have a big knot, which can only be placed within such an interval. The vertex selected by a thread is then decided by which hole its knot is placed within.

The embedding of the selector starts where the first gateland ended, at vertex $u_{(2b+1)m_1}$. Note that this is the vertex where the first filler is attached in Figure 22.8. We now embed $|V(G)|$ many $(k+1)$ holes with $(p-3)/2$ consecutive $(k+1)$-gates in between every consecutive pair of holes on the path $(u_{(2b+1)m_1}, \dots, u_{(2b+1)m_1+p(n-1)+3})$. After this we embed $b(p(n-1)+3)$ consecutive $(k+1)$-gates. In total, the selector is embedded on the vertices $(u_{(2b+1)m_1}, \dots, u_{m_2})$ for $m_2 = (2b+1)m_1 + (2b+1)(p(n-1)+3)$.

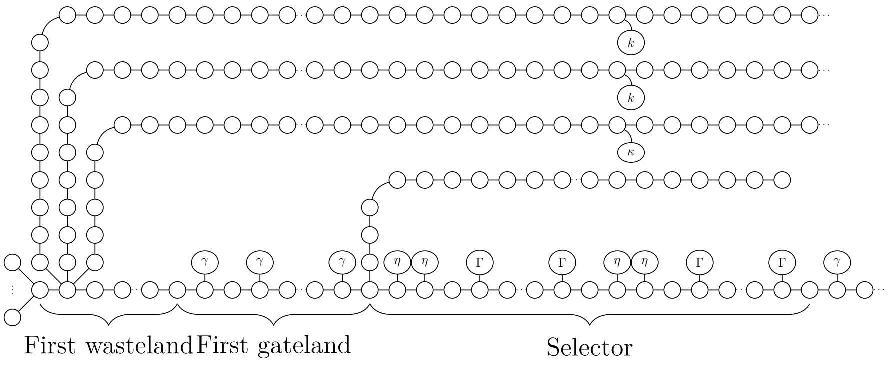


Figure 22.9: Illustration of the selector where $\gamma = 2(b-k-1)$, $\Gamma = 2(b-k-2)$, $\eta = \frac{3}{4}b-k-1$ and $\kappa = \frac{3}{2}b-k-1$.

The Middle Gateland

The middle gateland consist of bm_2 consecutive k-gates, embedded on the main path from vertex u_{m_2} to vertex $u_{(2b+1)m_2}$.

The Validator

We will now give the validator. Its job is to verify that the selected vertices of the threads in fact is a clique. The validator starts with $n-1$ neutral zones, followed by a validation zone and another $n-1$ neutral zones. After this there will be $b(2n-1)(4n+3)$ consecutive $(k+1)$-gates. A neutral zone is a P_{4n+4}. The zones will be joined by sharing endpoints in the same style as the gadgets in the selector. The validation zone consists of a $\hat{P}_{4n+4}$ where there is n many $(k+1)$-holes sharing endpoints embedded on the last $3n+1$ vertices. The validator is hence embedded on the vertices $(u_{(2b+1)m_2}, \dots, u_{m_3})$ for $m_3 = (2b+1)m_2 + (2b+1)(2n-1)(4n+3)$.

The Last Gateland, Wasteland and Wall

The last gateland consists of bm_3 consecutive k-gates embedded on the vertices $(u_{m_3}, \ldots, u_{(2b+1)m_3})$. After this we embed the last wasteland, which means that we leave the vertices $(u_{(2b+1)m_3}, u_{b^2(2b+1)m_3})$ untouched. Finally we turn the vertex $u_{b^2(2b+1)m_3+1}$ into the center of the last wall by attaching leaves to it.

The Threads and Their Danglements

We will now describe the threads and their danglements. As they are all isomorphic, it is sufficient to describe one of them. Let us name the vertices on the path that constitutes a thread by $t_2, \ldots$ with $t_2 = u_2$. The leaves neighbouring to the thread will be referred to as its danglements. First we turn $t_{(2b+1)m_1+1}$ into the center of a $(k+1)$-knot by attaching leaves to it. Starting at vertex $t_{(2b+1)m_2+(n-1)(4n+3)}$ we consider n consecutive, disjoint P_{4n+3}. For $\hat{P}_{4n+3}$ number i we do the following. We divide the $\hat{P}_{4n+3}$ into disjoint subpaths, first a $\hat{P}_{n+3}$ followed by n many P_3's. Consider the j'th P_3. If $i = j$ we turn the middle vertex of the P_3 into the center of a $(k+1)$-knot. Otherwise we attach a single leaf to the middle vertex if $(v_i, v_j) \notin E(G)$. This leaf will be referred to as a non-neighbouring leaf. After this we extend the thread with additional $b(2b+1)m_3$ vertices.

The Fillers

A fillers job is to fill up all available room within a sector of T to force this part of the main path to be stretched. To accomplish this we let the filler connected to $u_{(2b+1)m_1}$ be of length $(n-k)(\frac{3}{2}b - k - 2) + (2b+1)(p(n-1)+3)$. And the filler attached to $u_{(2b+1)m_2}$ to be of length $(b-1)(4n+3)(2n-1) + 2b(4n+3)(2n-1) - (k(2n-1)(4n+3) + k(n(\frac{3}{2}b - k - 2) + n^2 - n - 2m) + 2n(\frac{3}{4}b - k - 2))$.

22.4.1 Correctness

With the next lemmas we will prove the correctness of the reduction. After this we will continue by giving the implications of this reduction, which are the main results of this section. Recall that $b = 4k + 16$ and $p = 4n + 3$.

Lemma 22.9. *Given a yes-instance (G, k) of* Even Clique *the reduction instance (T, b) is a yes-instance of* Bandwidth.

Proof. We will now give a sparse ordering α of bandwidth $b = 4k + 16$, meaning that the image of α might not be an interval. To obtain a proper bandwidth ordering one can just compress α. During the description of α a *position* is a number in $\mathbb{N}$ that will be in the image of α and a vertex v is said to be *positioned* if the value $\alpha(v)$ has been given. Furthermore, we will say that v is *positioned at* c if $\alpha(v) = c$. By *reserving* a position for a subgraph H of T we guarantee that if a vertex will be positioned at that specific position, it will be a vertex of H. And by a position being *available* we will mean that no vertex has been positioned at that specific position so far. Let $C_k = \{c_1, \ldots, c_k\}$ be a k-clique in G.

For a vertex u_i on the main path let $\alpha(u_i) = bi + 1$. We continue by positioning the remainders of the two walls. Let c_f be the center of the first wall and L_f be the neighbouring leaves of c_f. Let $\alpha(L_f) = [\alpha(c_f) - b, \alpha(c_f) + b - 1] \setminus \{\alpha(c_f)\}$ in some arbitrary way. Similarly for the last wall, let $\alpha(L_l) = [\alpha(c_l) - b - 1, \alpha(c_l) + b] \setminus \{\alpha(c_l)\}$. Observe that for every two vertices u and v of T such that both $\alpha(u)$ and $\alpha(v)$ has been described, it holds that $\alpha(u) \neq \alpha(v)$. Furthermore, if uv is an edge in T it is true that $|\alpha(u) - \alpha(v)| \leq b$.

Order the threads of T and name them $\tau_1, \ldots, \tau_k$. Let u and v be two neighbours on the main path such that neither u nor v is the center of a wall and so that $\alpha(u) < \alpha(v)$. Observe that there is $b-1$ available positions within $I(u, v)$. Reserve the k positions in the middle of $I(u, v)$, one for each of the k threads. If there are two positions equally close to the middle, take the leftmost one. The leftmost is reserved for the first thread, the second to leftmost for the second thread and so forth.

For every vertex c_i of the clique we let j_i be such that $c_i = v_{j_i}$. Consider hole number j_i on the main path starting at the first wall, with h_1, h_2, h_3 and h_4 being the vertices on the main path for which the hole is embedded on such that $\alpha(h_1) < \alpha(h_2) < \alpha(h_3) < \alpha(h_4)$. Thus h_1, h_2, h_3 and h_4 are the *in*, *in center*, *out center* and *out* vertices of the hole respectively. Let c be the center of the first knot on τ_i and r the reserved position for τ_i in α within $I(h_2, h_3)$. We then set $\alpha(c) = r$ and complete the following procedure in the left (and right) direction on the thread τ_i. Let P be the path from c to $N(u_2) \cap V(\tau_i)$ (or to the end of the thread). If every vertex of P is positioned we stop. Otherwise, let u be the vertex closest to c on P not yet positioned. Furthermore, let $\hat{P}_2$ be the rightmost (leftmost) P_2 on the main path to the left (right) of the hole such that the position reserved for τ_i is available in $I(\hat{P}_2)$. If $\hat{P}_2$ is not part of any wasteland we set $\alpha(u)$ to this reserved position and continue. Otherwise we consider two cases. If we are right of r we position u at the leftmost position within $I(\hat{P}_2)$ that is either not reserved yet, or reserved for τ_i. If we are left of r we again consider two cases. Either there are exactly as many positions to the left of r reserved for τ_i as there are vertices before c not yet positioned. In that case we position u at the reserved position for τ_i within $I(\hat{P}_2)$. Otherwise, we position u at the rightmost position in $I(\hat{P}_2)$ that is either not reserved yet, or reserved for τ_i. Observe that if uv is an edge of τ_i there are positions x and y that are reserved for τ_i such that $y > x$ and $y - x = b$ and $\alpha(u)$ and $\alpha(v)$ are contained in $[x, y]$. It follows that $|\alpha(u) - \alpha(v)| \leq b$.

Note that the number of vertices on a thread that will be positioned to the left of r is $(2b + 1)m_1$ and that, by construction, $2bm_1$ of these will be within the inclusion interval of the first gateland. Hence it can be observed that there are at most km_1 vertices from the threads within the inclusion interval of the first wasteland. Recall that the distance from u_2 to the first vertex of the first gateland is $m_1 - 2$. Hence there are $(b - 1)(m_1 - 2) > km_1$ available positions within the inclusion interval of the first wasteland, before we position the threads. By the same kind of argument there are $(b - 1)(b^2 - 1)(2b + 1)m_3$ available positions in the inclusion interval of the last wasteland before positioning the thread. Recall that the length of a thread is bounded above by

$$\begin{aligned}&(2b+1)m_2 + (n-1)(4n+3) + (4n+3)n + b(2b+1)m_3\\ <&(2b+3)m_2 + b(2b+1)m_3\\ <&2b(2b+3)m_3.\end{aligned}$$

It follows that for every pair of vertices u and v such that both $\alpha(u)$ and $\alpha(v)$ has been described if holds that $\alpha(u) \neq \alpha(v)$.

Recall that every k-gate of T is embedded on the main path. And hence for every k-gate in T there are k paths passing through it with respect to α. Hence there are $2(b-k-1)$ positions available between the left and the right leaf and the rest of the leaves can be positioned in any way within this interval. Clearly, for every pair of vertices u and v of T, such that both $\alpha(u)$ and $\alpha(v)$ are described it holds that $\alpha(u) \neq \alpha(v)$. And furthermore, if uv is an edge of a k-gate it holds that $|\alpha(u) - \alpha(v)| \leq b$. For every $\hat{P}_2$ on the main path such that $\hat{P}_2$ is not in a subgraph of a wasteland and there are available positions in $I(\hat{P}_2)$ we reserve the position to the right of the k positions reserved for the threads, for the fillers. Observe that any $\hat{P}_2$ such that this position is not available either is a subgraph of a wasteland or a k-gate (which has no available positions).

We will now position the leaves of the knots. Let K be a knot in T. The center c of K is a vertex of a thread and hence $\alpha(c)$ has already been described. Let $\hat{P}_2$ be the P_2 of the main path such that $\alpha(c) \in I(\hat{P}_2)$. Position the leaves attached to c as close to the middle of $I(\hat{P}_2)$ as possible by only using available positions, that are not reserved. If there are two such positions equally close to the middle, we take the leftmost one. Let $\hat{P}_4$ be the P_4 of the main path such that $\hat{P}_2$ contains the internal vertices of $\hat{P}_4$. It can be observed, by where the knots are embedded on the thread and where the threads are positioned in α, that $\hat{P}_4$ is either a subgraph of a hole or a neutral zone. Furthermore, if c' is the center of some other knot and $\hat{P}_2'$ is the P_2 of the main main such that $\alpha(c')$ is contained in its inclusion interval, then it can be observed that $\hat{P}_2'$ and $\hat{P}_4$ are disjoint. Hence, we can observe that there are $2(b-k-2)$ positions available and non-reserved within $[\alpha(c)-b, \alpha(c)+b]$. Recall that a knot consists of $\frac{3}{2}b-k-2$ leaves and that $b = 4k+16$, and hence $2(b-k-2) \geq \frac{3}{2}b-k-2$. It follows that for every two vertices u and v of T such that both $\alpha(u)$ and $\alpha(v)$ have been described, $\alpha(u) \neq \alpha(v)$. Furthermore, it uv is an edge of T it holds that $|\alpha(u)-\alpha(v)| \leq b$.

Let $\hat{P}_4 = (u_h, u_{h+1}, u_{h+2}, u_{h+3})$ be some subpath of the main path such that a hole is embedded on it. Position the leaves attached to u_{h+1} to the leftmost non-reserved, available positions and the leaves attached to u_{h+2} to the rightmost non-reserved, available positions, within $I(\hat{P}_4)$. Furthermore, for the leaves representing non-neighbours, position it at the position available and not reserved closest to its neighbour. If there are two such positions, any of the two will do. It can be observed, by where the knots are embedded on the threads and where the knots and positioned that no two knots are positioned within the inclusion interval of a hole. And furthermore, that at most k non-adjacency leaves are positioned within the inclusion interval of a hole. At last, since C_k is a clique it holds that no knot and non-neighbour leaf is positioned with the inclusion interval of a hole. Recall that k is even and hence $\frac{3}{2}b - k - 2 = 5k + 22$ is even. It follows that the leaves of a knot is evenly distributed among the two sides of the center. Recall that the number of leaves in a hole is $\frac{3}{2}b - 2k - 4$. There are $3k$ vertices from the threads positioned within the inclusion interval of $\hat{P}_4$ and there are $3b - 3k - 6$ leaves attached to one hole and one knot. Since there are more than k leaves attached to a knot, it can be observed that for any two vertices u and v such that at least u or v is positioned within the inclusion interval of $\hat{P}_4$ it holds that $\alpha(u) \neq \alpha(v)$. And furthermore, if uv is an edge in T it holds that $|\alpha(u) - \alpha(v)| \leq b$.

Consider danglements positioned within the inclusion interval of a $\hat{P}_4$ that is a subgraph of a neutral zone. One can observe that there is at most k non-neighbouring leaves and at most one clique positioned within the inclusion interval of $\hat{P}_4$. And hence the same argument as above can be applied to show that for every two vertices u and v of T such that both $\alpha(u)$ and $\alpha(v)$ has been described, it holds that $\alpha(u) \neq \alpha(v)$. Furthermore, if uv is an edge of T it is true that $|\alpha(u) - \alpha(v)| \leq b$.

It remains to describe the positioning of each of the fillers. Let u be the vertex on the filler closest to the main path not yet positioned and r lowest value bigger than the α-value of the intersection vertex between the filler and the main path that is not taken. Set $\alpha(u) = r$ and continue. Recall that the length of the path where the selector is embedded is $(n-1)p + 3 + 2b(p(n-1)+3)$, and hence there were $(b-1)((n-1)p+3+2b(p(n-1)+3))$ available positions within the inclusion interval of the selector after only the main path had been positioned. Observe that the threads now occupies $k((n-1)p+3+2b(p(n-1)+3))$ of these positions, the $k+1$-gates $((p-3)(n-1)/2 + b(p(n-1)+3))2(b-k-2)$ of the positions, the knots $k(\frac{3}{2}b - k - 2)$ positions, the holes $2n(\frac{3}{4}b - k - 2)$ positions and the filler $(n-k)(\frac{3}{2}b - k - 2) + (2b+1)(p(n-1)+3)$. By substituting p by $4n+3$ and b by $4k+16$ one can verify that the vertices positioned equals the amount of positions available within the inclusion interval of the selector. The expression for the once available positions within the inclusion interval of the selector S and the number of vertices now positioned within it, disregarding the main path, namely X, is given below.

$$\begin{aligned}
S = {} & k((n-1)p + 3 + 2b(p(n-1)+3)) \\
& + ((p-3)(n-1)/2 + b(p(n-1)+3))2(b-k-2) \\
& + k(\frac{3}{2}b - k - 2) + 2n(\frac{3}{4}b - k - 2) \\
& + (n-k)(\frac{3}{2}b - k - 2) + (2b+1)(p(n-1)+3) \\
= {} & (b-1)((n-1)p + 3 + 2b(p(n-1)+3)) = X.
\end{aligned}$$

It follows that for every two vertices u and v such that both $\alpha(u)$ and $\alpha(v)$ have been described, it holds that $\alpha(u) \neq \alpha(v)$. Recall that for every $\hat{P}_2$ that is a subgraph of the selector there was a position reserved for the filler. And hence for every edge uv of the first filler, there are positions reserved for the filler, x

and y such that $y - x = b$ and $\alpha(u)$ and $\alpha(v)$ is contained within $[x, y]$. It follows directly that $|\alpha(u) - \alpha(v)| \leq b$. For the second filler, we observe that there were $(b-1)(4n+2)(2n-1)$ available positions within the inclusion interval of the validator when only the main path had been positioned. And furthermore, now the n holes occupies $2n(\frac{3}{4}b-k-2)$ of these positions, the threads $k(2n-1)(4n+2)$ of the positions, the knots $kn(\frac{3}{2}b-k-2)$ and the non-neighbouring leaves $k(n^2-n-2m)$. By a similar argument as for the first filler, one can prove that for every u and v of T it holds that $\alpha(u) \neq \alpha(v)$ and if uv is an edge of T then $|\alpha(u) - \alpha(v)| \leq b$. This completes the description of α and the argument is complete. □

Given a reduced instance (T, b) and a b-bandwidth ordering α we say that a k-gate in T is *blocked* with respect to α if every thread in T pass through the gate.

Lemma 22.10. *Let (T, b) be the result of the reduction for an instance of* Even Clique *and α a b-bandwidth ordering of T. Then every every thread passes through every k-gate. And in particular, ever k-gate in T is blocked with respect to α.*

Proof. By Lemma 22.6 we know that the first wall is either the leftmost or the rightmost elements of α. Observe that every k-gate in T is blocked with respect to α if and only if every k-gate in T is blocked with respect to α reversed. Hence it is sufficient to prove that every k-gate is blocked when the first wall is the leftmost elements of α.

Assume for a contradiction that there is a k-gate Π and a thread τ such that τ is not passing through Π. Let P be the path from u_2 to the *out* vertex of Π and let $X = V(\tau) - u_2$. By Lemma 22.6 we know that $\alpha(u_2) = \min \alpha(\tau)$ and that $\alpha(u_2) < \min \alpha(\Pi)$. It follows by the definition of passing through that $\max \alpha(\tau) \leq \max \alpha(\Pi)$ and hence $\alpha(X) \subseteq I(P)$. Recall that $|E(P)| \leq (2b+1)m_3 - 2$ and $|X| > b(2b+1)m_3$. It follows directly that $|I(P)| \leq b((2b+1)m_3 - 2) + 1 < b(2b+1)m_3 < |X|$ which is a contradiction. □

Recall that the main path of the reduction instance consist of 9 sectors, namely the first wall, the first wasteland, the first gateland, the selector, the middle gateland, the validator, the last gateland, the last wasteland and the last wall. See Figure 22.8 for an illustration. The lemma below shows that the sectors will appear in the same order in α as they do in the instance, up to reversion.

Lemma 22.11. *Let (T, b) be the result of the reduction for an instance of* Even Clique *and α a b-bandwidth ordering of T such that the first wall is mapped to the leftmost elements of α. If u and v are vertices from two different sectors such that u comes before v in T, then it holds that $\alpha(u) \leq \alpha(v)$.*

Proof. If at least one of the vertices are in one of the walls, the lemma follows directly from Lemma 22.6. We will now consider two cases. First, we consider the case when there is a k-gate Π with center c embedded on the inner vertices of the path from u to v. We make c adjacent to $\alpha^{-1}([\alpha(c) - b, \alpha(c) + b)$ and observe that c is now the center of a wall and α is still a b-bandwidth ordering of the graph.

Apply Lemma 22.6 on the first wall and the new wall to obtain $\alpha(u) \leq \alpha(c)$ and on the new wall and the last wall to obtain $\alpha(c) \leq \alpha(v)$. It follows immediately that $\alpha(u) \leq \alpha(v)$.

It remains to consider the case when there is no k-gate embedded on the inner vertices of the path from u to v. It follows, by construction, that either u or v is a vertex of a k-gate. First, let us consider the case when u is a vertex of a k-gate. Recall that the vertices the gate is embedded on is named $in, c = center$ and out and let P be the path from out to v. It follows by Lemmata 22.7 and 22.10 that $\alpha(P)$ and $[\alpha(c) - b, \alpha(c) + b]$ intersects in only one element, namely $\alpha(out)$, and that $\alpha(in) < \alpha(c) < \alpha(out)$. Since α is a b-bandwidth ordering it follows that $\alpha(out) = \min \alpha(P)$ and hence $\alpha(u) \leq \alpha(out) \leq \alpha(v)$. The case when v is a vertex of a k-gate follows by a symmetrical argument. □

Let P_F, P_M and P_L be the paths from the center of the first gate to the center of the last gate in the first gateland, the middle gateland and the last gateland respectively.

Lemma 22.12. *Let (T, b) be the result of the reduction for some instance of* Even Clique *and α a b-bandwidth ordering of T, then*

- *P_F, P_M and P_L are stretched with respect to α and*
- *for the centers of two k-gates c_1 and c_2 such that c_1 comes before c_2 in T it holds that $\alpha(c_1) < \alpha(c_2)$.*

Proof. This follows directly from Lemmata 22.7, 22.10 and 22.11. □

Let Π_F and Π_L be the first and last k-gate in T, and c_F and c_L their centers respectively. Furthermore, let P_R be the path from c_F to c_L.

Lemma 22.13. *Let (T, b) be the result of the reduction for some instance of* Even Clique *and α a b-bandwidth ordering of T. If $u \neq u_2$ is a vertex of a thread, such that the degree of u is at least three, then $\alpha(u) \in I(P_R)$.*

Proof. By Lemma 22.6 we know that the first wall is either the leftmost or the rightmost elements of α. Observe that u is mapped within the inclusion interval of P_R by α if and only if u is mapped within the inclusion interval of P_R by α reversed. Hence it is sufficient to prove that $\alpha(u) \in I(P_R)$ when the first wall is the leftmost elements of α.

Assume for a contradiction that there is a vertex $u \neq u_2$ of some thread, such that u has degree at least three and $\alpha(u) \notin I(P_R)$. It follows from Lemmata 22.11 and 22.12 that either $\alpha(u) < \alpha(c_F)$ or $\alpha(c_L) < \alpha(u)$. First, we consider the case when $\alpha(u) < \alpha(c_F)$. Let P_L^τ be the path from u_2 to u, except u_2 and P_R^τ the path from u to the last vertex of the thread. Furthermore, let P be the path from u_2 to c_F. By Lemmata 22.7, 22.10 and 22.11 we get that $\alpha(P_L^\tau) \subseteq I(P)$. Recall that $|V(P_L^\tau)| \geq (2b+1)m_1 - 1$ and that $|E(P)| = m_1$. It follows immediately that $|I(P)| \leq bm_1 + 1 < (2b+1)m_1 - 1 \leq |V(P_L^\tau)|$ and hence we obtain a contradiction.

It remains to consider the case when $\alpha(c_L) < \alpha(u)$. Let P^τ be the path from u_2 to u and P the path from u_2 to c_L except u_2. By assumption $\alpha(u_2) < \min \alpha(P)$ and hence $\alpha(P) \subseteq I(P^\tau)$. Recall that $|E(P^\tau)| < m_3$ and that $|V(P)| = (2b+1)m_3 - 3$. It follows that $|I(P^\tau)| < bm_3 + 1 < (2b+1)m_3 - 3 = |V(P)|$, which is a contradiction. $\square$

Lemma 22.14. *Let (T, b) be the result of the reduction for an instance of* Even Clique *and α a b-bandwidth ordering of T. Then*

- $|\alpha(\tau_i) \cap I(\hat{P}_2)| = 1$ *for every thread τ_i and every subpath $\hat{P}_2$ of P_R and*
- *P_R is stretched with respect to α.*

Proof. By Lemma 22.6 we know that the first wall is either the leftmost or the rightmost elements of α. Observe that P_R is stretched with respect to α if and only if P_R is stretched with respect to α reversed. It follows that it is sufficient to prove that the lemma holds when the first wall is the leftmost elements of α.

Let $Z = \alpha^{-1}(I(P_R))$ and observe that there are at most $2b$ vertices in $N(Z)$. Furthermore, observe that every leaf of a gate or a hole is either within $I(P_R)$ or a neighbor of Z. It follows from Lemma 22.10 that Lemma 22.7 applies to all k-gates of T. Furthermore, by Lemma 22.11 it follows that the neighbors of the fillers are positioned after the first gateland and before the last gateland. And hence by Lemmata 22.10 and 22.11 and the fact that α is a b-bandwidth ordering, it follows that both fillers are positioned within $I(P_R)$. By Lemma 22.13 it holds that every vertex v that is a danglement, its neighbor is positioned within $I(P_R)$. And hence v is either in $I(P_R)$ or a neighbor of Z. Below you find a table giving an overview of how many vertices not on the main path, each type of gadget contributes with to $N[Z]$.

Type of vertices	Amount
Knots	$k(n+1)(\frac{3}{2}b - k - 2)$
Holes	$4n(\frac{3}{4}b - k - 2)$
First filler	$(n-k)(\frac{3}{2}b - k - 2) + (2b+1)(p(n-1)+3)$
Second filler	$(b-1)(4n+3)(2n-1) + 2b(4n+3)(2n-1) - (k(2n-1)(4n+3) + k(n(\frac{3}{2}b - k - 2) + n^2 - n - 2m) + 2n(\frac{3}{4}b - k - 2))$
k-gates	$2(b-k-1)b(m_1 + m_2 + m_3)$
$(k+1)$-gates	$2(b-k-2)((n-1)(p-3)/2 + b(p(n-1)+3) + b(2n-1)(4n+3))$
non-neighbouring leaves	$k(n^2 - n - 2m)$

It follows from Lemma 22.6 that there are two vertices of the main path within $N(Z)$ and from Lemma 22.10 that there are $2k$ vertices from the threads in Z. Let X be all leaves in gates, holes and knots and non-neighbouring leaves and all the vertices in the fillers that are positioned within $I(P_R)$. We know that $|X|$ is at least the sum of the numbers in the table above, minus $2b - 2k - 2$. And hence it can be verified that $|X| = (b-k-1)((2b+1)m_3 - m_1 - 2)$. By construction it follows that $|E(P_R)| = (2b+1)m_3 - m_1 - 2$. It follows that we can apply Lemma 22.3 to complete the proof.

□

Name the holes of the selector such that the first hole is called H_1 and the last hole is H_n. Let (T, b) be a resulting instance of the reduction and α a b-bandwidth ordering of T. Furthermore, let H_i be a hole of T embedded on the path (v_1, v_2, v_3, v_4) such that v_1 comes before v_4 in T. We say that a thread τ is *selecting* i, if the center c of the first knot of the thread is positioned so that $\alpha(c) \in I(v_2, v_3)$.

Lemma 22.15. *Let (T, b) be the result of the reduction for the instance (G, k) of p-Even Clique and α a b-bandwidth ordering of T. Then every thread in T selects a unique integer in $[n]$.*

Proof. By Lemma 22.6 we know that the first wall is either the leftmost or the rightmost elements of α. Observe that every thread in T selects an unique integer with respect to α if and only if every thread in T selects an unique integer with respect to α reversed. It follows that it is sufficient to prove that the lemma holds when the first wall is the leftmost elements of α.

Let us consider a thread τ with vertices $(u_2 = t_2, t_3, \dots)$, where c is the center of the first knot K of τ. Furthermore, let c_F be the center of the first gate in the first gateland, c_M the center of the last gate in the middle gateland and c_L the center of the last gate in the last gateland. We first prove that $\alpha(c) \in I(c_F, c_M)$. We know that $\alpha(c) \in I(P_R)$ by Lemma 22.13 and hence in $I(c_F, c_L)$ by Lemma 22.14. Assume for a contradiction that $\alpha(c) \notin I(c_F, c_M)$, it follows that $\alpha(c) \in I(c_M, c_L)$. Let P^τ be the path from u_2 to c and P the path from u_3 to c_M. By Lemma 22.11 it follows that $\alpha(P) \subseteq I(P^\tau)$. Recall that $|E(P^\tau)| = (2b+1)m_1 - 1$ and that $V(P) = (2b+1)m_2 - 3$. A contradiction follows immediately, since $I(P^\tau) \leq b((2b+1)m_1 - 1) + 1 < (2b+1)m_2 - 3 \leq V(P)$. And hence we can assume $\alpha(c) \in I(c_F, c_M)$.

We will now prove that there is a hole H_i such that $\alpha(c) \in I(H_i)$. Assume for a contradiction that $\alpha(c) \notin I(H_i)$ for every i. Let $\hat{P}_2 = (p_1, p_2)$ be the P_2 of the main path such that $\alpha(c) \in I(\hat{P}_2)$. It follows by construction, that either p_1 or p_2 is the center of a gate. Observe that the leaves attached to c, p_1 and p_2 must be positioned within a $\hat{P}_4$. And due to Lemma 22.14 there are $4 + 3k$ vertices from the main path and the threads within $I(\hat{P}_4)$. Recall that there are $\frac{3}{2}b - k - 2$ leaves attached to c and at least $2(b-k-2)$ leaves attached to $\hat{P}_2$. This adds up to $4 + 3k + \frac{3}{2}b - k - 2 + 2b - 2k - 4 = \frac{7}{2}b - 2 > 3b + 1$ and hence we get a contradiction.

Let H_i be embedded on the path (v_1, v_2, v_3, v_4) such that v_1 comes before v_4 in T. Observe that due to Lemma 22.14 there is a position within the inclusion interval of the last $(k+1)$-gate of the selector that only the first filler can take. Due to our tight budget when it comes to positions within $I(P_R)$ (see the proof of Lemma 22.14) it follows that the first filler must take this position. And hence for every hole in the selector, the $(k+1)$-gate immediately before and after will be passed by the first filler. It follows that Lemma 22.7 is applicable on the $(k+1)$-gates in the selector and hence $\alpha(K) \subseteq I(v_1, v_4)$. Furthermore, due to Lemma 22.14 we know that $I(H_i) \subseteq I(v_1, v_4)$. And hence we can apply Lemma 22.8 to obtain that $\alpha(c) \in I(v_2, v_3)$.

It remains to prove that the threads selects unique integers. Assume otherwise for a contradiction and let τ and τ' be two threads selecting the same integer i. Hence there are two knots K and K' such that $\alpha(K) \cup \alpha(K') \subseteq I(H_i) \subseteq I(v_1, v_4)$. And since $|I(v_1, v_4)| = 3b+1 < 6b - 4k - 8 = 2(\frac{3}{2}b - k - 2) + 2(\frac{3}{4}b - k - 2) = |V(K) \cup V(K') \cup H_i|$ and hence we get our contradiction and the proof is complete. □

Lemma 22.16. *Let (T, b) be the result of the reduction for the instance (G, k) of* Even Clique *and α a b-bandwidth ordering of T. Then the set*

$$\{v_i \mid \textit{there is a thread selecting } i\}$$

is a clique in G.

Proof. By Lemma 22.6 we know that the first wall is either the leftmost or the rightmost elements of α. Observe that the set of integers selected by the threads with respect to α is the same as the one selected with respect to α reversed. It follows that it is sufficient to prove that the lemma holds when the first wall is the leftmost elements of α.

Let A be the set of selected integers and $C = \{v_i \mid i \in A\}$. From Lemma 22.15 we know that the size of both A and C is k. Assume for a contradiction that there are two vertices v_a and v_b in C such that v_a and v_b are not neighbours in G. Let τ_a be the thread selecting a and τ_b the thread selecting b. One can observe that by construction and Lemma 22.14 there is a hole H in the validation zone and a knot K_a with center c_a embedded on τ_a such that $\alpha(c_a) \in I(H)$.

Let (v_1, v_2, v_3, v_4) be the path that H is embedded on, such that v_1 comes before v_4 in T. From Lemma 22.14 one can observe that there is a position within the inclusion interval of the last $(k+1)$-gate in the validator that only the second filler can take. Due to our tight budget when it comes to positions within $I(P_R)$ (see the proof of Lemma 22.14) it follows that the second filler must take this position. It follows that Lemma 22.7 is applicable on the $(k+1)$-gates immediately before and after H. Hence it follows by Lemma 22.8 that $\alpha(K) \cup \alpha(H) \subseteq I(v_1, v_4)$.

From the construction of T and Lemma 22.14 one can observe that the vertex of τ_b positioned within $I(v_2, v_3)$ has a non-neighbouring leaf attached. It follows that there are $3(k+1)+4$ vertices from the threads, the filler and the main path positioned within $I(v_1, v_4)$. Furthermore, the knot contributes with $\frac{3}{2}b - k - 2$ leaves to $I(v_1, v_4)$ and the hole with $2(\frac{3}{4}b - k - 2)$. And in addition the non-neighbouring leaf must be positioned within $I(v_1, v_4)$. It follows that $3b + 1 = |I(v_1, v_4)| \leq 3(k+1) + 4 + \frac{3}{2}b - k - 2 + 2(\frac{3}{4}b - k - 2) + 1 = 3b + 7 - 2 - 4 + 1 = 3b + 2$ which is a contradiction and the proof is complete. □

Lemma 22.17. *Given an instance (G, k) of p-*CLIQUE *the reduction instance (T, b) of p-*BANDWIDTH *has a b-bandwidth ordering if and only if there is a clique of size k in G.*

Proof. This follows immediately by Lemmata 22.9, 22.15 and 22.16. □

22.5 Consequences

Recall from Chapter 1 that EVEN CLIQUE is W[1]-hard when parameterized by k (Theorem 1) and does not admit an $f(k)n^{o(k)}$ time algorithm unless ETH fails (Theorem 9). We now combine this with our reduction to obtain the promised results.

Theorem 43. BANDWIDTH *parameterized by b is* W[1]*-hard, even when the input graph is restricted to trees of pathwidth at most* 2*.*

Proof. The result follows directly from Lemma 22.17 and Theorem 1, together with the observations that the graph constructed by the reduction is a tree of pathwidth at most 2 and that $b = f(k)$. □

Theorem 44. BANDWIDTH *does not admit an $f(b)n^{o(b)}$ time algorithm, even when the input graph is restricted to trees of pathwidth at most* 2*, unless ETH fails.*

Proof. The result follows directly from Lemma 22.17 and Theorem 9, together with the observations that the graph constructed by the reduction is a tree of pathwidth at most 2 and that $b = O(k)$. □

Chapter 23

Concluding remarks

Lower bounds

In this part we have shown that the classic $2^{O(b)}n^{b+1}$ time dynamic programming algorithm of Saxe [Sax80] for the Bandwidth problem is essentially optimal, even on trees of pathwidth at most 2. On trees of pathwidth 1, namely caterpillars with hair length 1, the problem is known to be polynomial time solvable.

The gadgets introduced for the reduction provide a framework for selecting k vertices of the graph and testing properties regarding the neighborhoods of these vertices. Within the same framework, one can replace the danglements that are positioned within the validation zone in the validator by a somewhat small object for each vertex that this vertex does not have in its closed neighborhood. By somewhat small, we mean a set of leaves that are so that $k-1$ of these can fit within a hole while k cannot. We can in addition redefine the length of the second filler so that the validator is forced to be stretched. By doing so we ensure that no vertex in the graph is not within the closed neighborhood of the selected set. Or in other words, that the selected set is a dominating set in the graph.

This edited reduction proves that Bandwidth is indeed W[2]-hard. Recently, Chen and Lin [CL] proved that Dominating Set does not admit any constant factor approximation in $f(b)n^{O(1)}$ time, unless FPT = W[1]. Can one make the reduction presented here robust enough to imply a similar result for Bandwidth?

Algorithms and obstructions

On the positive side, we gave the first approximation algorithm for Bandwidth on graphs of bounded treelength with approximation ratio being a function of b and independent of n. Based on this one can ask if Bandwidth admit a parameterized approximation algorithm on general graphs?

One can observe that the exponential approximation factor for graphs of bounded treelength stems from the algorithm for trees. Hence, this is the most interesting algorithm to improve upon with respect to approximation factor. Does Bandwidth admit an approximation algorithm on trees with approximation ratio polynomial in b? What if one allows the algorithm to have running time $f(b)n^{O(1)}$?

Our approximation algorithm is based on pathwidth, local density and a new obstruction to bounded bandwidth called skewed Cantor combs. And given that your graph is of bounded treelength, we prove that these are the only obstructions that can prevent bandwidth from being small. It is natural to ask whether this extends to general graphs? Does there exist a function f such that any graph G with pathwidth at most c_1, local density at most c_2, and containing no S_{c_3} as a subgraph has bandwidth at most $f(c_1, c_2, c_3)$?

Part VI

Treewidth

Chapter 24

Introduction

Since its invention in the 1980s, the notion of treewidth has come to play a central role in an enormous number of fields, ranging from very deep structural theories to highly applied areas. An important (but not the only) reason for the impact of the notion is that many graph problems that are intractable on general graphs become efficiently solvable when the input is a graph of bounded treewidth. In most cases, the first step of an algorithm is to find a tree decomposition of small width and the second step is to perform a dynamic programming procedure on the tree decomposition.

In particular, if a graph on n vertices is given together with a tree decomposition of width k, many problems can be solved by dynamic programming in time $2^{O(k)}n$, i.e., single-exponential in the treewidth and linear in n. Many of the problems admitting such algorithms have been known for over thirty years [Bod88] but new algorithmic techniques on graphs of bounded treewidth [BCKN13] as well as new problems motivated by various applications (just a few of many examples are [ABDR12, Gil11, KvHK02, RPBD12]) continue to be discovered. While a reasonably good tree decomposition can be derived from the properties of the problem sometimes, in most of the applications, the computation of a good tree decomposition is a challenge.

Hence the natural question here is what can be done when no tree decomposition is given. In other words, is there an algorithm that for a given graph G and integer k, in time $2^{O(k)}n$ either correctly reports that the treewidth of G is more than k, or finds an optimal solution to our favorite problem (finds a maximum independent set, computes the chromatic number, decides if G is Hamiltonian, etc.)? To answer this question it would be sufficient to have an algorithm that in time $2^{O(k)}n$ either reports correctly that the treewidth of G is more that k, or constructs a tree decomposition of width at most ck for some constant c.

However, the lack of such algorithms has been a bottleneck, both in theory and in practical applications of the treewidth concept. The existing approximation algorithms give us the choice of running times of the form $2^{O(k)}n^2$, $2^{O(k \log k)}n \log n$, or $k^{O(k^3)}n$, see Table 24.1. Remarkably, the newest of these current record holders is now almost 20 years old. This "newest record holder" is the linear time algorithm of Bodlaender [Bod96] that given a graph G, decides if the treewidth of G is at most k, and if so, gives a tree decomposition of width at most k in $O(k^{O(k^3)}n)$ time. The improvement by Perković and Reed [PR00] is only a factor polynomial in k faster, however, if the treewidth is larger than k, it gives a subgraph of treewidth more than k with a tree decomposition of width at most $2k$, leading to an $O(n^2)$ algorithm for the fundamental disjoint paths problem. Recently, a version running in logarithmic space was found by Elberfeld et al. [EJT10], but its running time is not linear.

Reference	Approximation	$f(k)$	$g(n)$
Arnborg et al. [ACP87]	exact	$O(1)$	$O(n^{k+2})$
Robertson & Seymour [RS95]	$4k+3$	$O(3^{3k})$	n^2
Lagergren [Lag96]	$8k+7$	$2^{O(k \log k)}$	$n \log^2 n$
Reed [Ree92]	$8k+O(1)^1$	$2^{O(k \log k)}$	$n \log n$
Bodlaender [Bod96]	exact	$O(k^{O(k^3)})$	n
Amir [Ami10]	$4.5k$	$O(2^{3k}k^{3/2})$	n^2
Amir [Ami10]	$(3+2/3)k$	$O(2^{3.6982k}k^3)$	n^2
Amir [Ami10]	$O(k \log k)$	$O(k \log k)$	n^4
Feige et al. [FHL08]	$O(k \cdot \sqrt{\log k})$	$O(1)$	$n^{O(1)}$
This part	$3k+4$	$2^{O(k)}$	$n \log n$
This part	$5k+4$	$2^{O(k)}$	n

Table 24.1: Overview of treewidth algorithms. Here k is the treewidth and n is the number of vertices of an input graph G. Each of the algorithms outputs in time $f(k) \cdot g(n)$ a decomposition of width given in the Approximation column.

We now present the first constant factor approximation algorithm for the treewidth graph such that its running time is single exponential in treewidth and linear in the size of the input graph.

Of independent interest are a number of techniques that we use to obtain the result and the intermediate result of an algorithm that either tells that the treewidth is larger than k or outputs a tree decomposition of width at most $3k+4$ in time $2^{O(k)}n \log n$.

Related results and techniques

The basic shape of our algorithm is along the same lines as about all of the treewidth approximation algorithms [Ami10, BGHK95, FHL08, Lag96, Ree92, RS95], i.e., a specific scheme of repeatedly finding separators. If we ask for polynomial time approximation algorithms for treewidth, the currently best result is that of [FHL08] that gives in polynomial (but not linear) time a tree decomposition of width $O(k \cdot \sqrt{\log k})$ where k is the treewidth of the graph. Their work also gives a polynomial time approximation algorithm with ratio $O(|V_H|^2)$ for H-minor free graphs. By Austrin et al. [APW12], assuming the Small Set Expansion Conjecture, there is no polynomial time approximation algorithm for treewidth with a constant performance ratio.

An important element in the algorithms is the use of a data structure that allows to perform various queries in time $O(c^k \log n)$ each, for some constant c. This data structure is obtained by adding various new techniques to old ideas from the area of dynamic algorithms for graphs of bounded treewidth [Bod93, CSTV93, CZ98, CZ00, Hag00].

A central element in the data structure is a tree decomposition of the input graph of bounded (but too large) width such that the tree used in the tree decomposition is binary and of logarithmic depth. To obtain this tree decomposition, we combine the following techniques: following the scheme of the exact linear time algorithms [Bod96, PR00], but replacing the call to the dynamic programming algorithm of Bodlaender and Kloks [BK96] by a recursive call to our algorithm, we obtain a tree decomposition of G of width at most $10k + 9$ (or $6k + 9$, in the case of the $O(c^k n \log n)$ algorithm of Chapter 26.)

We use a result by Bodlaender and Hagerup [BH98] that this tree decomposition can be turned into a tree decomposition with a logarithmic depth binary tree in linear time. We then turn this shallow tree decomposition into a data structure that we can make queries to regarding the input graph.

Implementing the approximation algorithm for treewidth by Robertson and Seymour [RS95] using our data structure immediately gives a 3-approximation algorithm for treewidth running in time $O(c^k n \log n)$; This algorithm is explained in detail in Chapter 26. Additional techniques are needed to speed this algorithm up. We build a series of algorithms, with running times of the forms $O(c^k n \log\log n)$, $O(c^k n \log\log\log n), \ldots$, etc. Each algorithm "implements" Reed's algorithm [Ree92], but with a different procedure to find balanced separators of the subgraph at hand, and stops when the subgraph at hand has size $O(\log n)$. In the latter case, we call the previous algorithm of the series on this subgraph.

Finally, to obtain a linear time algorithm, we consider two cases, one case for when n is "small" (with respect to k), and one case when n is "large", where we consider n to be small if

$$n \leq 2^{2^{c_0 k^3}}, \text{ for some constant } c_0.$$

For small values of n, we apply the $O(c^k n \log\log n)$ algorithm from Chapter 27. This will yield a linear running time in n since $\log\log n \leq k$. For larger values of n, we show that the linear time algorithms of Bodlaender [Bod96] or Perković and Reed [PR00] can be implemented in truly linear time, without any overhead depending on k. This seemingly surprising result can be obtained roughly as follows.

We explicitly construct a finite state tree automaton of the dynamic programming algorithm in time double exponential in k. In this case, double exponential in k is in fact linear in n. This automaton is then applied on an *expression tree* constructed from our tree decomposition and this results in an algorithm running in time $2^{O(k)}n$. Viewing a dynamic programming algorithm on a tree decomposition as a finite state automaton traces back to early work by Fellows and Langston [FL], see e.g., also [AF93]. Our algorithm assumes the RAM model of computation [Sav98], and the only aspect of the RAM model which is exploited by our algorithm is the ability to look up an entry in a table in constant time, independently of the size of the table. This capability is crucially used in almost every linear time graph algorithm including breadth first search and depth first search.

Chapter 25

Proof outline

This part combines several different techniques. Instead of directly giving the full proofs with all details, we first give in this section a more intuitive (but still quite technical) outline of the results and techniques. The roadmap of this outline is as follows: first, we briefly explain some constant factor approximation algorithms for treewidth upon which our algorithm builds. First we give a variant of the algorithm by Robertson and Seymour [RS95] (which was already presented in Section 1.6), which within a constant factor, approximates treewidth with a running time $O(c^k n^2)$. Then, in Section 25.2 we discuss the $O(k^{O(k)} n \log n)$ algorithm by Reed [Ree92]. After this, we sketch in Section 25.3 the proof of our new $O(c^k n \log n)$ 3-approximation for treewidth, building upon the earlier discussed algorithms by Robertson and Seymour and by Reed. This algorithm needs a technical lemma, of which the main graph theoretic ideas are sketched in Sections 25.4 and 25.5. The algorithm needs a specific data structure: we exploit having a tree decomposition of bounded (but still too large) width to perform several queries in $O(c^k \log n)$ time; this is sketched in Section 25.6. The algorithm with running time $O(c^k n \log n)$ is used as the first in a series of algorithm, with running times $O(c^k n \log\log n)$, $O(c^k n \log\log\log n)$, etc, each calling the previous one as a subroutine; this is sketched in Section 25.7. How we obtain from this series of algorithms our final $O(c^k n)$ algorithm then is sketched in Section 25.8.

25.1 The algorithm from Graph Minors XIII

The engine behind the algorithm is a lemma that states that graphs of treewidth k have balanced separators of size $k + 1$. In particular, for any way to assign non-negative weights to the vertices there exists a set X of size at most $k + 1$ such that the total weight of any connected component of $G \setminus X$ is at most half of the total weight of G. We will use the variant of the lemma where some vertices have weight 1 and some have weight 0.

Lemma 25.1 (Graph Minors II [RS86]). *If $\mathrm{tw}(G) \leq k$ and $S \subseteq V(G)$, then there exists $X \subseteq V(G)$ with $|X| \leq k + 1$ such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S.*

We note that the original version of [RS86] is seemingly stronger: it gives bound $\frac{1}{2}|S \setminus X|$ instead of $\frac{1}{2}|S|$. However, we do not need this stronger version and we find it more convenient to work with the weaker. The set X with properties ensured by Lemma 25.1 will be called a *balanced S-separator*, or a *$\frac{1}{2}$-balanced S-separator*. More generally, for an *β-balanced S-separator* X every connected component of $G \setminus X$ contains at most $\beta|S|$ vertices of S. If we omit the set S, i.e., talk about separators instead of S-separators, we mean $S = V(G)$ and balanced separators of the whole vertex set.

The proof of Lemma 25.1 is not too hard; start with a tree decomposition of G with width at most k and orient every edge of the decomposition tree towards the side which contains the larger part of the set S. Two edges of the decomposition can not point "in different directions", since then there would be disjoint parts of the tree, both containing more than half of S. Thus there has to be a node in the decomposition tree such that all edges of the decomposition are oriented towards it. The bag of the decomposition corresponding to this node is exactly the set X of at most $k + 1$ vertices whose deletion leaves connected components with at most $\frac{1}{2}|S|$ vertices of S each.

The proof of Lemma 25.1 is constructive if one has access to a tree decomposition of G of width less than k. The algorithm does not have such a decomposition at hand, after all we are trying to compute a decomposition of G of small width. Thus we have to settle for the following algorithmic variant of the lemma [RS86].

Lemma 25.2 ([RS95]). *There is an algorithm that given a graph G, a set S and a $k \in \mathbb{N}$ either concludes that $\mathrm{tw}(G) > k$ or outputs a set X of size at most $k + 1$ such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S and runs in time $O(3^{|S|}k^{O(1)}(n + m))$.*

Proof sketch. By Lemma 25.1 there exists a set X' of size at most $k + 1$ such that every component of $G \setminus X'$ has at most $\frac{1}{2}|S|$ vertices which are in S. A simple packing argument shows that the components can be assigned to left or right such that at most $\frac{2}{3}|S|$ vertices of S go left and at most $\frac{2}{3}|S|$ go right. Let S_X be $S \cap X'$ and let S_L and S_R be the vertices of S that were put left and right respectively. By trying all partitions of S in three parts the algorithm correctly guesses S_X, S_L and S_R. Now X' separates S_L from S_R and so the minimum vertex cut between S_L and S_R in $G \setminus S_X$ is at most $|X' \setminus S_X| \leq (k + 1) - |S_X|$. The algorithm finds using max-flow a set Z of size at most $(k + 1) - |S_X|$ that separates S_L from S_R in $G \setminus S_X$. Since we are only interested in a set Z of size at most $k - |S_X|$ one can run max-flow in time $O((n + m)k^{O(1)})$. Having found S_L, S_R, S_X and Z the algorithm sets $X = S_X \cup Z$, L to contain all components of $G \setminus X$ that contain vertices of S_L and R to contain all other vertices. Since every component C of $G \setminus X$ is fully contained in L or R, the bound on $|C \cap S|$ follows.

If no partition of S into S_L, S_R, S_X yielded a cutset Z of size at most $(k + 1) - |S_X|$, this means that $\mathrm{tw}(G) > k$, which the algorithm reports. □

The algorithm takes as input G, k and a set S on at most $3k+3$ vertices, and either concludes that the treewidth of G is larger than k or finds a tree decomposition of width at most $4k+3$ such that the top bag of the decomposition contains S.

On input G, S, k the algorithm starts by ensuring that $|S| = 3k+3$. If $|S| < 3k+3$ the algorithm just adds arbitrary vertices to S until equality is obtained. Then the algorithm applies Lemma 25.2 and finds a set X of size at most $k+1$ such that each component C_i of $G \setminus X$ satisfies $|C_i \cap S| \leq \frac{2|S|}{3} \leq 2k+2$. Thus for each C_i we have $|(S \cap C_i) \cup X| \leq 3k+3$. For each component C_i of $G \setminus X$ the algorithm runs itself recursively on $(G[C_i \cup X], (S \cap C_i) \cup X, k)$.

If either of the recursive calls returns that the treewidth is more than k then the treewidth of G is more than k as well. Otherwise we have for every component C_i a tree decomposition of $G[C_i \cup X]$ of width at most $4k+3$ such that the top bag contains $(S \cap C_i) \cup X$. To make a tree decomposition of G we make a new root node with bag $X \cup S$, and connect this bag to the roots of the tree decompositions of $G[C_i \cup X]$ for each component C_i. It is easy to verify that this is indeed a tree decomposition of G. The top bag contains S, and the size of the top bag is at most $|S| + |X| \leq 4k+4$, and so the width if the decomposition is at most $4k+3$ as claimed.

The running time of the algorithm is governed by the recurrence

$$T(n,k) = O(3^{|S|}k^{O(1)}(n+m)) + \sum_{C_i} T(|C_i \cup X|, k) \tag{25.1}$$

which solves to $T(n,k) \leq (3^{3k}k^{O(1)}n(n+m))$ since $|S| = 3k+3$ and there always are at least two non-empty components of $G \setminus X$. Finally, if $|E(G)| > nk$ the algorithm can safely output that $\mathrm{tw}(G) > k$ by Lemma 2.3. After this, running the algorithm above takes time $O(3^{3k}k^{O(1)}n(n+m)) = O(3^{3k}k^{O(1)}n^2)$.

25.2 The approximation algorithm of Reed

Reed [Ree92] observed that the running time of the algorithm of Robertson and Seymour [RS95] can be sped up from $O(n^2)$ for fixed k to $O(n \log n)$ for fixed k, at the cost of a worse (but still constant) approximation ratio, and a $k^{O(k)}$ dependence on k in the running time, rather than the 3^{3k} factor in the algorithm of Robertson and Seymour. We remark here that Reed [Ree92] never states explicitly the dependence on k of his algorithm, but a careful analysis shows that this dependence is in fact of order $k^{O(k)}$. The main idea of this algorithm is that the recurrence in Equation 25.1 only solves to $O(n^2)$ for fixed k if one of the components of $G \setminus X$ contains almost all of the vertices of G. If one could ensure that each component C_i of $G \setminus X$ had at most $c \cdot n$ vertices for some fixed $c < 1$, the recurrence in Equation 25.1 solves to $O(n \log n)$ for fixed k. To see that this is true we simply consider the recursion tree. The total amount of work done at any level of the recursion tree is $O(n)$ for a fixed k. Since the size of the components considered at one level is always a constant factor smaller than the size of the components considered in the previous level, the number of levels is only $O(\log n)$ and we have $O(n \log n)$ work in total.

By using Lemma 25.1 with $S = V(G)$ we see that if G has treewidth at most k, then there is a set X of size at most $k+1$ such that each component of $G \setminus X$ has size at most $\frac{n}{2}$. Unfortunately if we try to apply Lemma 25.2 to *find* an X which splits G in a balanced way using $S = V(G)$, the algorithm of Lemma 25.2 takes time $O(3^{|S|}k^{O(1)}(n+m)) = O(3^n n^{O(1)})$, which is exponential in n. Reed [Ree92] gave an algorithmic variant of Lemma 25.1 especially tailored for the case where $S = V(G)$.

Lemma 25.3 ([Ree92]). *There is an algorithm that given G and k, runs in time $O(k^{O(k)}n)$ and either concludes that $\mathrm{tw}(G) > k$ or outputs a set X of size at most $k+1$ such that that every component of $G \setminus X$ has at most $\frac{3}{4}n$ vertices.*

Let us remark that Lemma 25.3 as stated here is never explicitly proved in [Ree92], but it follows easily from the arguments given there.

Having Lemmata 25.2 and 25.3 at hand, we show how to obtain an 8-approximation of treewidth in time $O(k^{O(k)}n \log n)$. The algorithm takes as input G, k and a set S on at most $6k+6$ vertices, and either concludes that the treewidth of G is at least k, or finds a tree decomposition of width at most $8k+7$ such that the top bag of the decomposition contains S.

On input G, S, k the algorithm starts by ensuring that $|S| = 6k+6$. If $|S| < 6k+6$ the algorithm just adds vertices to S until equality is obtained. Then the algorithm applies Lemma 25.2 and finds a set X_1 of size at most $k+1$ such that each component C_i of $G \setminus X_1$ satisfies $|C_i \cap S| \leq \frac{2}{3}|S| \leq 4k+4$. Now the algorithm applies Lemma 25.3 and finds a set X_2 of size at most $k+1$ such that each component C_i of $G \setminus X_2$ satisfies $|C_i| \leq \frac{3}{4}|V(G)| \leq \frac{3}{4}n$. Set $X = X_1 \cup X_2$. For each component C_i of $G \setminus X$ we have that $|(S \cap C_i) \cup X| \leq 6k+6$. For each component C_i of $G \setminus X$ the algorithm runs itself recursively on $(G[C_i \cup X], (S \cap C_i) \cup X, k)$.

If either of the recursive calls returns that the treewidth is more than k then the treewidth of G is more than k as well. Otherwise we have for every component C_i a tree decomposition of $G[C_i \cup X]$ of width at most $8k+7$ such that the top bag contains $(S \cap C_i) \cup X$. Similarly as before, to make a tree decomposition of G we make a new root node with bag $X \cup S$, and connect this bag to the roots of the tree decompositions of $G[C_i \cup X]$ for each component C_i. It is easy to verify that this is indeed a tree decomposition of G. The top bag contains S, and the size of the top bag is at most $|S| + |X| \leq |S| + |X_1| + |X_2| \leq 6k+6+2k+2 = 8k+8$, and the width of the decomposition is at most $8k+7$ as claimed.

The running time of the algorithm is governed by the recurrence

$$T(n,k) \leq O\left(k^{O(k)}(n+m)\right) + \sum_{C_i} T(|C_i \cup X|, k) \tag{25.2}$$

which solves to $T(n,k) = O(k^{O(k)}(n+m)\log n)$ since each C_i has size at most $\frac{3}{4}n$. By Lemma 2.3 we have $m \leq kn$ and so the running time of the algorithm is upper bounded by $O(k^{O(k)}n \log n)$.

25.3 Faster than Reed

The goal of this section is to sketch a proof of the existence of an approximation algorithm either producing a tree decomposition of width at most $3k + 4$ or correctly concluding that $\mathrm{tw}(G) > k$ in time $O(c^k n \log n)$. A full proof can be found in Chapter 26.

The algorithm employs the same recursive compression scheme which is used in Bodlaender's linear time algorithm [Bod96] and the algorithm of Perković and Reed [PR00]. The idea is to solve the problem recursively on a smaller instance, expand the obtained tree decomposition of the smaller graph to a "good, but not quite good enough" tree decomposition of the instance in question, and then use this tree decomposition to either conclude that $\mathrm{tw}(G) > k$ or find a decomposition of G which is good enough. A central concept in this recursive approach of Bodlaender's algorithm [Bod96] is the definition of an improved graph:

Definition 25.4. Given a graph G and an integer k, the *improved* graph of G, denoted G_I, is obtained by adding an edge between each pair of vertices with at least $k + 1$ common neighbors of degree at most k in G.

Intuitively, adding the edges during construction of the improved graph cannot spoil any tree decomposition of G of width at most k, as the pairs of vertices connected by the new edges will need to be contained together in some bag anyway. This is captured in the following lemma.

Lemma 25.5. *Given a graph G and an integer $k \in \mathbb{N}$, it holds that $\mathrm{tw}(G) \leq k$ if and only if $\mathrm{tw}(G_I) \leq k$.*

If $|E(G)| = O(kn)$, which is the case in graphs of treewidth at most k, the improved graph can be computed in $O(k^{O(1)} \cdot n)$ time using radix sort [Bod96].

A vertex $v \in G$ is called *simplicial* if its neighborhood is a clique, and *I-simplicial*, if it is simplicial in the improved graph G_I. The intuition behind I-simplicial vertices is as follows: all the neighbors of an I-simplicial vertex must be simultaneously contained in some bag of any tree decomposition of G_I of width at most k, so we can safely remove such vertices from the improved graph, compute the tree decomposition, and reintroduce the removed I-simplicial vertices. The crucial observation is that if no large set of I-simplicial vertices can be found, then one can identify a large matching, which can be also used for a robust recursion step. The following lemma, which follows from the work of Bodlaender [Bod96], encapsulates all the main ingredients that we will use.

Lemma 25.6. *There is an algorithm working in $O(k^{O(1)}n)$ time that, given a graph G and an integer k, either*

(i) returns a maximal matching in G of cardinality at least $\frac{|V(G)|}{O(k^6)}$,

(ii) returns a set of at least $\frac{|V(G)|}{O(k^6)}$ I-simplicial vertices, or,

(iii) correctly concludes that the treewidth of G is larger than k.

Moreover, if a set X of at least $\frac{|V(G)|}{O(k^6)}$ I-simplicial vertices is returned, and the algorithm is in addition provided with some tree decomposition $\mathcal{T}_I$ of $G_I \setminus X$ of width at most k, then in $O(k^{O(1)} \cdot n)$ time one can turn $\mathcal{T}_I$ into a tree decomposition $\mathcal{T}$ of G of width at most k, or conclude that the treewidth of G is larger than k.

Lemma 25.6 allows us to reduce the problem to a *compression* variant where we are given a graph G, an integer k and a tree decomposition of G of width $O(k)$, and the goal is to either conclude that the treewidth of G is at least k or output a tree decomposition of G of width at most $3k + 4$. Our approximation algorithm has two parts: an algorithm for the compression step and an algorithm for the general problem that uses the algorithm for the compression step together with Lemma 25.6 as black boxes. We now state the properties of our algorithm for the compression step in the following lemma.

Lemma 25.7. *There exists an algorithm which on input $G, k, S_0, \mathcal{T}_{apx}$, where*

(i) $S_0 \subseteq V(G)$, $|S_0| \leq 2k + 3$,

(ii) $G \setminus S_0$ is connected, and

(iii) $\mathcal{T}_{apx}$ is a tree decomposition of G of width at most $O(k)$,

in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 3k + 4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.

We now give a proof of an algorithm as promised in the beginning of this section and later formalized in Theorem 45, assuming the correctness of Lemma 25.7. In particular we will give an algorithm that runs in $O(c^k n \log n)$ time and either outputs a tree decomposition of width at most $3k + 4$ or correctly concludes that the treewidth of the input graph is strictly more than k. The correctness of the lemma will be argued in Sections 25.4 and 25.5.

Proof of Theorem 45. Our algorithm will in fact solve a slightly more general problem. Here we are given a graph G, an integer k and a set S_0 on at most $2k + 3$ vertices, with the property that $G \setminus S_0$ is connected. The purpose of S_0 in the final algorithm lies in the recursive step: when we recursively apply our algorithm to different connected components, we need to ensure that we are able to connect the tree decomposition of the different connected components onto the already connected tree decomposition without blowing up the width too much. The algorithm will either conclude that $\mathrm{tw}(G) > k$ or output a tree decomposition of width at most $3k + 4$ such that S_0 is the root bag. To get a tree decomposition of any (possibly disconnected) graph it is sufficient to run this algorithm on each connected component with $S_0 = \emptyset$. The algorithm proceeds as follows. It first applies Lemma 25.6 on $(G, 3k + 4)$. If the algorithm of Lemma 25.6 concludes that $\mathrm{tw}(G) > 3k + 4$ the algorithm reports that $\mathrm{tw}(G) > 3k + 4 > k$.

If the algorithm finds a matching M in G with at least $\frac{|V(G)|}{O(k^6)}$ edges, it contracts every edge in M and obtains a graph G'. Since G' is a minor of G we know that $\mathrm{tw}(G') \leq \mathrm{tw}(G)$. The algorithm runs itself recursively on $(G', k, \emptyset)$, and either concludes that $\mathrm{tw}(G') > k$ (implying $\mathrm{tw}(G) > k$) or outputs a tree decomposition of G' of width at most $3k+4$. Uncontracting the matching in this tree decomposition yields a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width at most $6k+9$ [Bod96]. Now we can run the algorithm of Lemma 25.7 on $(G, k, S_0, \mathcal{T}_{\text{apx}})$ and either obtain a tree decomposition of G of width at most $3k+4$ and S_0 as the root bag, or correctly conclude that $\mathrm{tw}(G) > k$.

If the algorithm finds a set X of at least $\frac{|V(G)|}{O(k^6)}$ I-simplicial vertices, it constructs the improved graph G_I and runs itself recursively on $(G_I \setminus X, k, \emptyset)$. If the algorithm concludes that $\mathrm{tw}(G_I \setminus X) > k$ then $\mathrm{tw}(G_I) > k$ implying $\mathrm{tw}(G) > k$ by Lemma 25.5. Otherwise we obtain a tree decomposition of $G_I \setminus X$ of width at most $3k+4$. We may now apply Lemma 25.6 and obtain a tree decomposition $\mathcal{T}_{\text{apx}}$ of G with the same width. Note that we can not just output $\mathcal{T}_{\text{apx}}$ directly, since we can not be sure that S_0 is the top bag of $\mathcal{T}_{\text{apx}}$. However we can run the algorithm of Lemma 25.7 on $(G, k, S_0, \mathcal{T}_{\text{apx}})$ and either obtain a tree decomposition of G of width at most $3k+4$ and S_0 as the root bag, or correctly conclude that $\mathrm{tw}(G) > k$.

It remains to analyze the running time of the algorithm. Suppose the algorithm takes time at most $T(n, k)$ on input (G, k, S_0) where $n = |V(G)|$. Running the algorithm of Lemma 25.6 takes $O(k^{O(1)}n)$ time. Then the algorithm either halts, or calls itself recursively on a graph with at most $n - \frac{n}{O(k^6)} = n(1 - \frac{1}{O(k^6)})$ vertices taking time $T(n(1 - \frac{1}{O(k^6)}), k)$. Then the algorithm takes time $O(k^{O(1)}n)$ to either conclude that $\mathrm{tw}(G) > k$ or to construct a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$. In the latter case we finally run the algorithm of Lemma 25.7, taking time $O(c^k \cdot n \log n)$. This gives the following recurrence:

$$T(n, k) \leq O\left(c^k \cdot n \log n\right) + T\left(n\left(1 - \frac{1}{O(k^6)}\right), k\right)$$

The recurrence leads to a geometric series and solves to $T(n, k) \leq O(c^k k^{O(1)} \cdot n \log n)$, completing the proof. For a thorough analysis of the recurrence, see Equations 26.1 and 26.2 in Chapter 26. Pseudocode for the algorithm described here is given in Algorithm 4 in Chapter 26. □

25.4 A compression algorithm

We now proceed to give a sketch of a proof for a slightly weakened form of Lemma 25.7. The goal is to give an algorithm that given as input a graph G, an integer k, a set S_0 of size at most $6k + 6$ such that $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$, runs in time $O(c^k n \log n)$ and either correctly concludes that $\mathrm{tw}(G) > k$ or outputs a tree decomposition of G of width at most $8k + 7$. This chapter does not contain a full proof of this variant of Lemma 25.7—we will discuss the proof of Lemma 25.7 in Section 25.5. The aim of this section is to demonstrate that the recursive scheme of Section 25.3 together with a nice trick for finding balanced separators is already sufficient to obtain a factor 8 approximation for treewidth running in time $O(c^k n \log n)$. A variant of the trick used in this section for computing balanced separators turns out to be useful in our final $O(c^k n)$ time 5-approximation algorithm.

The route we follow here is to apply the algorithm of Reed described in Section 25.2, but instead of using Lemma 25.3 to find a set X of size $k+1$ such that every connected component of $G \setminus X$ is small, finding X by dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ in time $O(c^k n)$. There are a few technical difficulties with this approach.

The most serious issue is that, to the best of our knowledge, the only known dynamic programming algorithms for balanced separators in graphs of bounded treewidth take time $O(c^k n^2)$ rather than $O(c^k n)$: in the state, apart from a partition of the bag, we also need to store the cardinalities of the sides which gives us another dimension of size n. We now explain how it is possible to overcome this issue. We start by applying the argument in the proof of Lemma 25.1 on the tree decomposition $\mathcal{T}_{\text{apx}}$ and get in time $O(k^{O(1)} n)$ a partition of $V(G)$ into L_0, X_0 and R_0 such that there are no edges between L_0 and R_0, $\max(|L_0|, |R_0|) \leq \frac{3}{4}n$ and $|X_0| \leq \mathrm{w}(\mathcal{T}_{\text{apx}}) + 1$. For every way of writing $k+1 = k_L + k_X + k_R$ and every partition of X_0 into $X_L \cup X_X \cup X_R$ with $|X_X| = k_X$ we do the following.

First we find in time $O(c^k n)$ using dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ a partition of $L_0 \cup X_0$ into $\hat{L}_L \cup \hat{X}_L \cup \hat{R}_L$ such that there are no edges from $\hat{L}_L$ to $\hat{R}_L$, $|\hat{X}_L| \leq k_L + k_X$, $X_X \subseteq \hat{X}_L$, $X_R \subseteq \hat{R}_L$ and $X_L \subseteq \hat{L}_L$ and the size $|\hat{L}_L|$ is maximized.

Then we find in time $O(c^k n)$ using dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ a partition of $R_0 \cup X_0$ into $\hat{L}_R \cup \hat{X}_R \cup \hat{R}_R$ such that there are no edges from $\hat{L}_R$ to $\hat{R}_R$, $|\hat{X}_R| \leq k_R + k_X$, $X_X \subseteq \hat{X}_R$, $X_R \subseteq \hat{R}_R$ and $X_L \subseteq \hat{L}_R$ and the size $|\hat{R}_R|$ is maximized. Let $L = \hat{L}_L \cup \hat{L}_R$, $R = \hat{R}_L \cup \hat{R}_R$ and $X = X_L \cup X_R$. The sets L, X, R form a partition of $V(G)$ with no edges from L to R and $|X| \leq k_L + k_X + k_R + k_X - k_X \leq k+1$.

It is possible to show using a combinatorial argument (see Lemma 29.5 in Chapter 29) that if $\mathrm{tw}(G) \leq k$ then there exists a choice of k_L, k_X, k_R such that $k+1 = k_L + k_X + k_R$ and partition of X_0 into $X_L \cup X_X \cup X_R$ with $|X_X| = k_X$ such that the above algorithm will output a partition of $V(G)$ into X, L and R such that $\max(|L|, |R|) \leq \frac{8n}{9}$. Thus we have an algorithm that in time $O(c^k n)$ either finds a set X of size at most $k+1$ such that each connected component of $G \setminus X$ has size at most $\frac{8n}{9}$ or correctly concludes that $\mathrm{tw}(G) > k$.

The second problem with the approach is that the algorithm of Reed is an 8-approximation algorithm rather than a 3-approximation. Thus, even the sped up version does not quite prove Lemma 25.7. It does however yield a version of Lemma 25.7 where the compression algorithm is an 8-approximation. In the proof of Theorem 45 there is nothing special about the number 3 and so one can use this weaker variant of Lemma 25.7 to give a 8-approximation algorithm for treewidth in time $O(c^k n \log n)$. We will not give complete details of this algorithm, as we will shortly describe a proof of Lemma 25.7 using a quite different route.

It looks difficult to improve the algorithm above to an algorithm with running time $O(c^k n)$. The main hurdle is the following: both the algorithm of Robertson and Seymour [RS95] and the algorithm of Reed [Ree92] find a separator X and proceed recursively on the components of $G \setminus X$. If we use $O(c^k \cdot n)$ time to find the separator X, then the total running time must be at least $O(c^k \cdot n \cdot d)$ where d is the depth of the recursion tree of the algorithm. It is easy to see that the depth of the tree decomposition output by the algorithms equals (up to constant factors) the depth of the recursion tree. However there exist graphs of treewidth k such that no tree decomposition of depth $o(\log n)$ has width $O(k)$ (take for example powers of paths). Thus the depth of the constructed tree decompositions, and hence the recursion depth of the algorithm must be at least $\Omega(\log n)$.

Even if we somehow managed to reuse computations and find the separator X in time $O(c^k \cdot \frac{n}{\log n})$ on average, we would still be in trouble since we need to pass on the list of vertices of the connected components of $G \setminus X$ that we will call the algorithm on recursively. At a first glance this has to take $O(n)$ time and then we are stuck with an algorithm with running time $O((c^k \cdot \frac{n}{\log n} + n) \cdot d)$, where d is the recursion depth of the algorithm. For $d = \log n$ this is still $O(c^k n + n \log n)$ which is slower than what we are aiming at. In Section 25.5 we give a proof of Lemma 25.7 that *almost* overcomes these issues.

25.5 A better compression algorithm

We give a sketch of the proof of Lemma 25.7. The goal is to give an algorithm that given as input a connected graph G, an integer k, a set S_0 of size at most $2k + 3$ such that $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ of G, runs in time $O(c^k n \log n)$ and either correctly concludes that $\text{tw}(G) > k$ or outputs a tree decomposition of G of width at most $3k + 4$ with top bag S_0.

Our strategy is to implement the $O(c^k n^2)$ time 4-approximation algorithm described in Section 25.1, but make some crucial changes in order to (a) make the implementation run in $O(c^k n \log n)$ time, and (b) make it a 3-approximation rather than a 4-approximation. We first turn to the easier of the two changes, namely making the algorithm a 3-approximation.

To get an algorithm that satisfies all of the requirements of Lemma 25.7, but runs in time $O(c^k n^2)$ rather than $O(c^k n \log n)$ we run the algorithm described in Section 25.1 setting $S = S_0$ in the beginning. Instead of using Lemma 25.2 to find a set X such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S, we apply Lemma 25.1 to show the *existence* of an X such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S, and do dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ in time $O(c^k n)$ in order to find such an X. Going through the analysis of Section 25.1 but with X satisfying that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S shows that the algorithm does in fact output a tree decomposition with width $3k + 4$ and top bag S_0 whenever $\mathrm{tw}(G) \leq k$.

It is somewhat non-trivial to do dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ in time $O(c^k n)$ in order to find an X such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S. The problem is that $G \setminus X$ could potentially have many components and we do not have time to store information about each of these components individually. The following lemma, whose proof can be found in Section 29.4.2, shows how to deal with this problem.

Lemma 25.8. *Let G be a graph and $S \subseteq V(G)$. Then a set X is a balanced S-separator if and only if there exists a partition (M_1, M_2, M_3) of $V(G) \setminus X$, such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$.*

Lemma 25.8 shows that when looking for a balanced S-separator we can just look for a partition of G into four sets X, M_1, M_2, M_3 such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$. This can easily be done in time $O(c^k n)$ by dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$. This yields the promised algorithm that satisfies all of the requirements of Lemma 25.7, but runs in time $O(c^k n^2)$ rather than $O(c^k n \log n)$.

We now turn to the most difficult part of the proof of Lemma 25.7, namely how to improve the running time of the algorithm above from $O(c^k n^2)$ to $O(c^k n \log n)$ in a way that gives hope of a further improvement to running time $O(c^k n)$. The $O(c^k n \log n)$ time algorithm we describe now is based on the following observations:

(i) In any recursive call of the algorithm above, the considered graph is an induced subgraph of G. Specifically the considered graph is always $G[C \cup S]$ where S is a set with at most $2k+3$ vertices and C is a connected component of $G \setminus S$.

(ii) The only computationally hard step, finding the balanced S-separator X, is done by dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of G.

These observations give some hope that one can reuse the computations done in the dynamic programming when finding a balanced S-separator for G during the computation of balanced S-separators in induced subgraphs of G. This plan can be carried out in a surprisingly clean manner and we now give a rough sketch of how it can be done.

We start by preprocessing the tree decomposition using an algorithm of Bodlaender and Hagerup [BH98]. This algorithm is a parallel algorithm and here we state its sequential form. Essentially, Proposition 25.9 lets us assume without loss of generality that the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ has depth $O(\log n)$:

Proposition 25.9 (Bodlaender and Hagerup [BH98])**.** *There is an algorithm that, given a tree decomposition of width k with $O(n)$ nodes of a graph G, finds a rooted binary tree decomposition of G of width at most $3k + 2$ with depth $O(\log n)$ in $O(kn)$ time.*

In Chapter 29 we will describe a data structure with the following properties. The data structure takes as input a graph G, an integer k and a tree decomposition $\mathcal{T}_{\text{apx}}$ of width $O(k)$ and depth $O(\log n)$. After an initialization step which takes $O(c^k n)$ time the data structure allows us to do certain operations and queries. At any point of time the data structure is in a certain *state*. The operations allow us to change the state of the data structure. Formally, the state of the data structure is a 3-tuple (S, X, F) of subsets of $V(G)$ and a vertex π called the "pin", with the restriction that $\pi \notin S$. The initial state of the data structure is that $S = S_0$, $X = F = \emptyset$, and π is an arbitrary vertex of $G \setminus S_0$. The data structure allows operations that change S, X or F by inserting/deleting a specified vertex, and move the pin to a specified vertex in time $O(c^k \log n)$.

For a fixed state of the data structure, the *active component* U is the component of $G \setminus S$ which contains π. The data structure allows the query findSSeparator which outputs in time $O(c^k \log n)$ either an S-balanced separator $\hat{X}$ of $G[U \cup S]$ of size at most $k+1$, or $\perp$, which means that $\text{tw}(G[S \cup U]) > k$.

The algorithm of Lemma 25.7 runs the $O(c^k n^2)$ time algorithm described above, but *uses the data structure* to find the balanced S-separator in time $O(c^k \log n)$ instead of doing dynamic programming over $\mathcal{T}_{\text{apx}}$. All we need to make sure is that the S in the state of the data structure is always equal to the set S for which we want to find the balanced separator, and that the active component U is set such that $G[U \cup S]$ is equal to the induced subgraph we are working on. Since we always maintain that $|S| \leq 2k+3$ we can change the set S to anywhere in the graph (and specifically into the correct position) by doing $k^{O(1)}$ operations taking $O(c^k \log n)$ time each.

At a glance, it looks like viewing the data structure as a black box is sufficient to obtain the desired $O(c^k n \log n)$ time algorithm. However, we haven't even used the sets X and F in the state of the data structure, or described what they mean! The reason for this is of course that there is a complication. In particular, after the balanced S-separator $\hat{X}$ is found—how can we recurse into the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$? We need to move the pin into each of these components one at a time, but if we want to use $O(c^k \log n)$ time in each recursion step, we cannot afford to spend $O(|S \cup U|)$ time to compute the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$. We resolve this issue by pushing the problem into the data structure, and showing that the appropriate queries can be implemented there. This is where the sets X and F in the state of the data structure come in.

Recall that the data structure is a 3-tuple (S, X, F) of subsets of $V(G)$ together with a pin π. The role of the second argument X in these triples in the data structure is that when queries to the data structure depending on X are called, X equals the set $\hat{X}$, i.e., the balanced S-separator found by the query findSSeparator. The set F is a set of "finished pins" whose intention is the following: when the algorithm calls itself recursively, we use findNextPin to find a connected component U' of $G[S \cup U] \setminus (S \cup \hat{X})$, with the restriction that U' does not contain any vertices of F. After it has finished computing a tree decomposition of

$G[U' \cup N(U')]$ with $N(U')$ as its top bag, it selects an arbitrary vertex of U' and inserts it into F.

The query findNextPin finds a new pin π' in some component U' of $G[S \cup U] \setminus (S \cup \hat{X})$ that does not contain any vertices of F. And finally, the query findNeighborhood allows us to find the neighborhood $N(U')$, which in turn allows us to call the algorithm recursively in order to find a tree decomposition of $G[U' \cup N(U')]$ with $N(U')$ as its top bag.

At this point it should be clear that the $O(c^k n^2)$ time algorithm described in the beginning of this section can be implemented using $O(k^{O(1)})$ calls to the data structure in each recursive step, thus spending only $O(c^k \log n)$ time in each recursive step. Pseudocode for this algorithm can be found in Algorithm 6. The recurrence bounding the running time of the algorithm then becomes

$$T(n,k) \leq O(c^k \log n) + \sum_{U_i} T(|U_i \cup \hat{X}|, k).$$

Here $U_1, \ldots, U_q$ are the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$. This recurrence solves to $O(c^k n \log n)$, proving Lemma 25.7. A full proof of Lemma 25.7 assuming the data structure as a black box may be found in Section 26.2.

25.6 The data structure

We sketch the main ideas in the implementation of the data structure. The goal is to set up a data structure that takes as input a graph G, an integer k and a tree decomposition $\mathcal{T}_{\text{apx}}$ of width $O(k)$ and depth $O(\log n)$, and initializes in time $O(c^k n)$. The *state* of the data structure is a 4-tuple (S, X, F, π) where S, X and F are vertex sets in G and $\pi \in V(G) \setminus S$. The initial state of the data structure is $(S_0, \emptyset, \emptyset, v)$ where v is an arbitrary vertex in $G \setminus S_0$. The data structure should support operations that insert (delete) a single vertex to (from) S, X and F, and an operation to change the pin π to a specified vertex. These operations should run in time $O(c^k \log n)$. For a given state of the data structure, set U to be the component of $G \setminus S$ that contains π. The data structure should also support the following queries in time $O(c^k \log n)$.

- findSSeparator: Assuming that $|S| \leq 2k+3$, return a set $\hat{X}$ of size at most $k+1$ such that every component of $G[S \cup U] \setminus \hat{X}$ contains at most $\frac{1}{2}|S|$ vertices of S, or conclude that $\text{tw}(G) > k$.

- findNextPin: Return a vertex π' in a component U' of $G[S \cup U] \setminus (S \cup \hat{X})$ that does not contain any vertices of F.

- findNeighborhood: Return $N(U)$ if $|N(U)| < 2k+3$ and $\perp$ otherwise.

Suppose for now that we want to set up a much simpler data structure. Here the state is just the set S and the only query we want to support is findSSeparator which returns a set $\hat{X}$ such that every component of $G \setminus (S \cup \hat{X})$ contains at most $\frac{1}{2}|S|$ vertices of S, or conclude that $\mathrm{tw}(G) > k$. At our disposal we have the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of width $O(k)$ and depth $O(\log n)$. To set up the data structure we run a standard dynamic programming algorithm for finding $\hat{X}$ given S. Here we use Lemma 25.8 and search for a partition of $V(G)$ into (M_1, M_2, M_3, X) such that $|X| \leq k + 1$, there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$. This can be done in time $O(c^k k^{O(1)} n)$ and the tables stored at each node of the tree decomposition have size $O(c^k k^{O(1)})$. This finishes the initialization step of the data structure. The initialization step took time $O(c^k k^{O(1)} n)$.

We will assume without loss of generality that the top bag of the decomposition is empty. The data structure will maintain the following invariant: after every change has been performed the tables stored at each node of the tree decomposition correspond to a valid execution of the dynamic programming algorithm on input (G, S). If we are able to maintain this invariant, then answering findSSeparator queries is easy: assuming that each cell of the dynamic programming table also stores solution sets (whose size is at most $k+1$) we can just output in time $O(k^{O(1)})$ the content of the top bag of the decomposition!

But how to maintain the invariant and support changes in time $O(c^k \log n)$? It turns out that this is not too difficult: the content of the dynamic programming table of a node t in the tree decomposition depends only on S and the dynamic programming tables of t's children. Thus, when the dynamic programming table of the node t is changed, this will only affect the dynamic programming tables of the $O(\log n)$ ancestors of t. If the dynamic program is done carefully, one can ensure that adding or removing a vertex to/from S will only affect the dynamic programming tables for a single node t in the decomposition, together with all of its $O(\log n)$ ancestors. Performing the changes amounts to recomputing the dynamic programming tables for these nodes, and this takes time $O(c^k k^{O(1)} \log n)$.

It should now be plausible that the idea above can be extended to work also for the more complicated data structure with the more advanced queries. Of course there are several technical difficulties, the main one is how to ensure that the computation is done in the connected component U of $G \setminus S$ without having to store “all possible ways the vertices in a bag could be connected below the bag”. We omit the details of how this can be done in this outline. The full exposition of the data structure can be found in Chapter 29.

25.7 An almost linear time algorithm

We now sketch how the algorithm of the previous section can be sped up, at the cost of increasing the approximation ratio from 3 to 5. In particular we argue that for every $\alpha \in \mathbb{N}$, there exists an algorithm which in $O(c_\alpha^k \cdot n \log^{(\alpha)} n)$ time either computes a tree decomposition of width at most $5k + 3$ or correctly concludes that $\mathrm{tw}(G) > k$. This is formalized in Theorem 46.

Observe that the algorithm from Section 25.3 satisfies the conditions for $\alpha = 1$. We will show how one can use the algorithm for $\alpha = 1$ in order to obtain an algorithm for $\alpha = 2$. In particular we aim at an algorithm which given a graph G and an integer k, in $O(c_2^k \cdot n \log\log n)$ time for some $c_2 \in \mathbb{N}$ either computes a tree decomposition of G of width at most $5k+3$ or correctly concludes that $\mathrm{tw}(G) > k$.

We inspect the $O(c^k n \log n)$ algorithm for the compression step described in Section 25.5. It uses the data structure of Section 25.6 in order to find balanced separators in time $O(c^k \log n)$. The algorithm uses $O(c^k \log n)$ time on each recursive call regardless of the size of the induced subgraph of G it is currently working on. When the subgraph we work on is big this is very fast. However, when we get down to induced subgraphs of size $O(\log\log n)$ the algorithm of Robertson and Seymour described in Section 25.1 would spend $O(c^k(\log\log n)^2)$ time in each recursive call, while our presumably fast algorithm still spends $O(c^k \log n)$ time. This suggests that there is room for improvement in the recursive calls where the considered subgraph is very small compared to n.

The overall structure of our $O(c_2^k \log\log n)$ time algorithm is identical to the structure of the $O(c^k \log n)$ time algorithm of Theorem 45. The only modifications happen in the compression step. The compression step is also similar to the $O(c^k \log n)$ algorithm described in Section 25.5, but with the following caveat. The data structure query findNextPin finds the *largest* component where a new pin can be placed, returns a vertex from this component, and also returns the size of this component. If a call of findNextPin returns that the size of the largest yet unprocessed component is less than $\log n$ the algorithm does not process this component, nor any of the other remaining components in this recursive call. This ensures that the algorithm is never run on instances where it is slow. Of course, if we do not process the small components we do not find a tree decomposition of them either. A bit of inspection reveals that what the algorithm will do is either conclude that $\mathrm{tw}(G) > k$ or find a tree decomposition of an induced subgraph of G' of width at most $3k+4$ such that for each connected component C_i of $G \setminus V(G')$, *(a)* $|C_i| \leq \log n$, *(b)* $|N(C_i)| \leq 2k+3$, and *(c)* $N(C_i)$ is fully contained in some bag of the tree decomposition of G'.

How much time does it take the algorithm to produce this output? Each recursive call takes $O(c^k \log n)$ time and adds a bag to the tree decomposition of G' that contains some vertex which was not yet in $V(G')$. Thus the total time of the algorithm is upper bounded by $O(|V(G')| \cdot c^k \log n)$. What happens if we run this algorithm, then run the $O(c^k n \log n)$ time algorithm of Theorem 45 on each of the connected components of $G \setminus V(G')$? If either of the recursive calls return that the treewidth of the component is more than k then $\mathrm{tw}(G) > k$. Otherwise we have a tree decomposition of each of the connected components with width $3k+4$. With a little bit of extra care we find tree decompositions of the same width of $C_i \cup N(C_i)$ for each component C_i, such that the top bag of the decomposition contains $N(C_i)$. Then all of these decompositions can be glued together with the decomposition of G' to yield a decomposition of width $3k+4$ for the entire graph G.

The running time of the above algorithm can be bounded as follows. It takes

$O(|V(G')| \cdot c^k \log n)$ time to find the partial tree decomposition of G', and

$$O(\sum_i c_2^k |C_i| \log |C_i|) = O(c_2^k \log\log n \cdot \sum_i |C_i|) = O(c_2^k n \log\log n)$$

time to find the tree decompositions of all the small components. Thus, if $|V(G')| = O(\frac{n}{\log n})$ the running time of the first part would be $O(c^k n)$ and the total running time would be $O(c_2^k n \log\log n)$.

How big can $|V(G')|$ be? In other words, if we inspect the algorithm described in Section 25.1, how big part of the graph does the algorithm see before all remaining parts have size less than $\log n$? The bad news is that the algorithm could see almost the entire graph. Specifically if we run the algorithm on a path it could well be building a tree decomposition of the path by moving along the path and only terminating when reaching the vertex which is $\log n$ steps away from from the endpoint. The good news is that the algorithm of Reed described in Section 25.2 will get down to components of size $\log n$ after decomposing only $O(\frac{n}{\log n})$ vertices of G. The reason is that the algorithm of Reed also finds balanced separators of the considered subgraph, ensuring that the size of the considered components drop by a constant factor for each step down in the recursion tree.

Thus, if we augment the algorithm that finds the tree decomposition of the subgraph G' such that that it also finds balanced separators of the active component and adds them to the top bag of the decomposition before going into recursive calls, this will ensure that $|V(G')| = O(\frac{n}{\log n})$ and that the total running time of the algorithm described in the paragraphs above will be $O(c_2^k n \log\log n)$. The algorithm of Reed described in Section 25.2 has a worse approximation ratio than the algorithm of Robertson and Seymour described in Section 25.1. The reason is that we also need to add the balanced separator to the top bag of the decomposition. When we augment the algorithm that finds the tree decomposition of the subgraph G' in a similar manner, the approximation ratio also gets worse. If we are careful about how the change is implemented we can still achieve an algorithm with running time $O(c_2^k n \log\log n)$ that meets the specifications of Theorem 46 for $\alpha = 2$.

The approach to improve the running time from $O(c^k n \log n)$ to $O(c_2^k n \log\log n)$ also works for improving the running time from $O(c_\alpha^k \cdot n \log^{(\alpha)} n)$ to $O(c_{\alpha+1}^k \cdot n \log^{(\alpha+1)} n)$. Running the algorithm that finds in $O(c^k n)$ time the tree decomposition of the subgraph G' such that all components of $G \setminus V(G')$ have size $\log n$ and running the $O(c_\alpha^k \cdot n \log^{(\alpha)} n)$ time algorithm on each of these components yields an algorithm with running time $O(c_{\alpha+1}^k \cdot n \log^{(\alpha+1)} n)$.

In the above discussion we skipped over the following issue. How can we compute a small balanced separator for the active component in time $O(c^k \log n)$? It turns out that also this can be handled by the data structure. The main idea here is to consider the dynamic programming algorithm used in Section 25.4 to find balanced separators in graphs of bounded treewidth, and show that this algorithm can be turned into a $O(c^k \log n)$ time data structure query. We would like to remark here that the implementation of the trick from Section 25.4 is significantly more involved than the other queries: we need to use the approximate tree decomposition not only for fast dynamic programming computations, but also to locate the separation (L_0, X_0, R_0) on which the trick is employed. A detailed explanation of how this is done can be found at the end of Section 29.4.4. This completes the proof sketch of Theorem 46. A full proof can be found in Chapter 27.

25.8 Obtaining linear time

The algorithm(s) of the previous section are in fact already $O(c^k n)$ algorithms unless n is astronomically large compared to k. If, for example, $n \leq 2^{2^{2^k}}$ then $\log^{(3)} n \leq k$ and so $O(c^k n \log^{(3)} n) \leq O(c^k kn)$. Thus, to get an algorithm which runs in $O(c^k n)$ it is sufficient to consider the cases when n is really, really big compared to k. The recursive scheme of Section 25.3 allows us to only consider the case where *(a)* n is really big compared to k and *(b)* we have at our disposal a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$.

For this case, consider the dynamic programming algorithm of Bodlaender and Kloks [BK96] that given G and a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$ either computes a tree decomposition of G of width k or concludes that $\text{tw}(G) > k$ in time $O(2^{O(k^3)}n)$. The dynamic programming algorithm can be turned into a tree automaton based algorithm [FL, AF93] with running time $O(2^{2^{O(k^3)}} + n)$ if one can inspect an arbitrary entry of a table of size $O(2^{2^{O(k^3)}})$ in constant time. If $n \geq \Omega(2^{2^{O(k^3)}})$ then inspecting an arbitrary entry of a table of size $O(2^{2^{O(k^3)}})$, means inspecting an arbitrary entry of a table of size $O(n)$, which one can do in constant time in the RAM model. Thus, when $n \geq \Omega(2^{2^{O(k^3)}})$ we can find an optimal tree decomposition in time $O(n)$. When $n = O(2^{2^{O(k^3)}})$ the $O(c^k n \log^{(3)} n)$ time algorithm of the previous section runs in time $O(c^k kn)$.

This concludes the outline of the proof of the linear time algorithm. A full explanation of how to handle the case where n is much bigger than k can be found in Chapter 28.

Chapter 26

A faster approximation algorithm

In this section, we provide formal details of the proof of the following statement:

Theorem 45. *There exists an algorithm which given a graph G and an integer k, either computes a tree decomposition of G of width at most $3k+4$ or correctly concludes that $\mathrm{tw}(G) > k$, in time $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$.*

In fact, the algorithm that we present, is slightly more general. The main procedure, $\mathtt{Alg}_1$, takes as input a connected graph G, an integer k, and a subset of vertices S_0 such that $|S_0| \leq 2k+3$. Moreover, we have a guarantee that not only G is connected, but $G \setminus S_0$ as well. $\mathtt{Alg}_1$ runs in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ and either concludes that $\mathrm{tw}(G) > k$, or returns a tree decomposition of G of width at most $3k+4$, such that S_0 is the root bag. Clearly, to prove Theorem 45, we can run $\mathtt{Alg}_1$ on every connected component of G separately using $S_0 = \emptyset$. Note that computation of the connected components takes $O(|V(G)| + |E(G)|) = O(kn)$ time, since if $|E(G)| > kn$, then we can safely output that $\mathrm{tw}(G) > k$.

The presented algorithm $\mathtt{Alg}_1$ uses two subroutines. As described in Chapter 25, $\mathtt{Alg}_1$ uses a reduction approach; in short words, we either apply a reduction step, or find an approximate tree decomposition of width $O(k)$ on which a compression subroutine $\mathtt{Compress}_1$ can be employed. In this compression step we are either able to find a refined, compressed tree decomposition of width at most $3k+4$, or again conclude that $\mathrm{tw}(G) > k$.

The algorithm $\mathtt{Compress}_1$ starts by initializing the data structure (see Chapter 25 for an intuitive description of the role of the data structure), and then calls a subroutine `FindTD`. This subroutine resembles the algorithm of Robertson and Seymour (see Chapter 25): it divides the graph using balanced separators, recurses on the different connected components, and combines the subtrees obtained for the components into the final tree decomposition.

26.1 The main procedure

Algorithm $\mathtt{Alg}_1$, whose layout is proposed as Algorithm 4, runs very similarly to the algorithm of Bodlaender [Bod96]; we provide here all the necessary details for the sake of completeness, but we refer to [Bod96] for a wider discussion.

First, we apply Lemma 25.6 on graph G for parameter $3k+4$. We either immediately conclude that $\mathrm{tw}(G) > 3k+4$, find a set of I-simplicial vertices of size at least $\frac{n}{O(k^6)}$, or a matching of size at least $\frac{n}{O(k^6)}$. Note that in the application of Lemma 25.6 we ignore the fact that some of the vertices are distinguished as S_0.

If a matching M of size at least $\frac{n}{O(k^6)}$ is found, we employ a similar strategy as in [Bod96]. We first contract the matching M to obtain G'; note that if G had treewidth at most k then so does G'. Then we apply $\mathtt{Alg}_1$ recursively to obtain a tree decomposition $\mathcal{T}'$ of G' of width at most $3k+4$, and having achieved this we decontract the matching M to obtain a tree decomposition $\mathcal{T}$ of G of width at most $6k+9$: every vertex in the contracted graph is replaced by at most two vertices before the contraction. Finally, we call the sub-procedure $\mathtt{Compress}_1$, which given G, S_0, k and the decomposition $\mathcal{T}$ (of width $O(k)$), either concludes that $\mathrm{tw}(G) > k$, or provides a tree decomposition of G of width at most $3k+4$, with S_0 as the root bag. $\mathtt{Compress}_1$ is given in details in the next section.

In case of obtaining a large set X of I-simplicial vertices, we proceed similar to Bodlaender [Bod96]. We compute the improved graph, remove X from it, apply $\mathtt{Alg}_1$ on $G_I \setminus X$ recursively to obtain its tree decomposition $\mathcal{T}'$ of width at most $3k+4$, and finally reintroduce the missing vertices of X to obtain a tree decomposition $\mathcal{T}$ of G of width at most $3k+4$ (recall that reintroduction can fail, and in this case we may conclude that $\mathrm{tw}(G) > k$). Observe that the decomposition $\mathcal{T}$ satisfies all the needed properties, with the exception that we have not guaranteed that S_0 is the root bag. However, to find a decomposition that has S_0 as the root bag, we may again make use of the subroutine $\mathtt{Compress}_1$, running it on input G, S_0, k and the tree decomposition $\mathcal{T}$. Lemma 25.6 ensures that all the described steps, apart from the recursive calls to $\mathtt{Alg}_1$ and $\mathtt{Compress}_1$, can be performed in $O(k^{O(1)} \cdot n)$ time. Note that the I-simplicial vertices can safely be reintroduced since we used Lemma 25.6 for parameter $3k+4$ instead of k.

Let us now analyze the running time of the presented algorithm, provided that the running time of the subroutine $\mathtt{Compress}_1$ is $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$. Since all the steps of the algorithm (except for calls to subroutines) can be performed in $O(k^{O(1)} \cdot n)$ time, the time complexity satisfies the following recurrence relation:

$$T(n) \quad \leq \quad O(k^{O(1)} \cdot n) + O(c^k \cdot n \log n) + T\left(\left(1 - \frac{1}{C \cdot k^6}\right) n\right); \qquad (26.1)$$

Here C is the constant hidden in the O-notation in Lemma 25.6. By unraveling the recurrence into a geometric series, we obtain that

$$\begin{aligned} T(n) \quad &\leq \quad \sum_{i=0}^{\infty} \left(1 - \frac{1}{Ck^6}\right)^i O(k^{O(1)} \cdot n + c^k \cdot n \log n) \qquad (26.2) \\ &= \quad Ck^6 \cdot O(k^{O(1)} \cdot n + c^k \cdot n \log n) = O(c_1^k \cdot n \log n), \end{aligned}$$

for some $c_1 > c$.

Algorithm 4: `Alg`$_1$

```
Input: A connected graph $G$, an integer $k$, and $S_0 \subseteq V(G)$ s.t.
       $|S_0| \leq 2k+3$ and $G \setminus S_0$ is connected.
Output: A tree decomposition $\mathcal{T}$ of $G$ with $\mathrm{w}(\mathcal{T}) \leq 3k+4$ and $S_0$ as the
        root bag, or conclusion that $\mathrm{tw}(G) > k$.

  Run algorithm of Lemma 25.6 for parameter $3k+4$
  if Conclusion that $\mathrm{tw}(G) > 3k+4$ then
      return $\perp$
  end
  if G has a matching M of cardinality at least $\frac{n}{O(k^6)}$ then
      Contract $M$ to obtain $G'$.
      $\mathcal{T}' \leftarrow$ Alg$_1(G', k)$        /* $\mathrm{w}(\mathcal{T}') \leq 3k+4$ */
      if $\mathcal{T}' = \perp$ then
          return $\perp$
      else
          Decontract the edges of $M$ in $\mathcal{T}'$ to obtain $\mathcal{T}$.
          return Compress$_1(G, k, \mathcal{T})$
      end
  end
  if G has a set X of at least $\frac{n}{O(k^6)}$ I-simplicial vertices then
      Compute the improved graph $G_I$ and remove $X$ from it.
      $\mathcal{T}' \leftarrow$ Alg$_1(G_I \setminus X, k)$        /* $\mathrm{w}(\mathcal{T}') \leq 3k+4$ */
      if $\mathcal{T}' = \perp$ then
          return $\perp$
      end
      Reintroduce vertices of $X$ to $\mathcal{T}'$ to obtain $\mathcal{T}$.
      if Reintroduction failed then
          return $\perp$
      else
          return Compress$_1(G, k, \mathcal{T})$
      end
  end
```

26.2 Compression

In this section we provide the details of the implementation of the subroutine `Compress`$_1$. The main goal is encapsulated in the following lemma.

Lemma 26.1 (Lemma 25.7, restated)**.** *There exists an algorithm which on input $G, k, S_0, \mathcal{T}_{apx}$, where*

(i) $S_0 \subseteq V(G)$, $|S_0| \leq 2k+3$,

(ii) $G \setminus S_0$ is connected, and

(iii) $\mathcal{T}_{apx}$ is a tree decomposition of G of width at most $O(k)$,

in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 3k+4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.

The subroutine's layout is given as Algorithm 5. Roughly speaking, we first initialize the data structure $\mathcal{DS}$, with $G, k, S_0, \mathcal{T}$ as input, and then run a recursive algorithm `FindTD` that constructs the decomposition itself given access to the data structure. See Figure 26.1 for an overview of the operations of our data structure $\mathcal{DS}$. The decomposition is returned by a pointer to the root bag. The data structure interface will be explained in the following paragraphs, and its implementation is given in Chapter 29. We refer to Chapter 25 for a brief, intuitive outline.

Algorithm 5: `Compress`$_1(G, k, \mathcal{T})$.

```
Input: Connected graph $G$, $k \in \mathbb{N}$, a set $S_0$ s.t. $|S_0| \leq 2k+3$ and $G \setminus S_0$ is
       connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ with $\mathrm{w}(\mathcal{T}_{\text{apx}}) = O(k)$
Output: Tree decomposition of $G$ of width at most $3k+4$ with $S_0$ as the
        root bag, or conclusion that $\mathrm{tw}(G) > k$.
  Initialize data structure $\mathcal{DS}$ with $G, k, S_0, \mathcal{T}_{\text{apx}}$
  return FindTD()
```

The initialization of the data structure takes $O(c^k n)$ time (see Lemma 29.1). The time complexity of `FindTD`, given in Section 26.3, is $O(c^k \cdot n \log n)$.

26.3 The recursive algorithm

The subroutine `FindTD` works on the graph G with two disjoint vertex sets S and U distinguished. Intuitively, S is small (of size at most $2k+3$) and represents the root bag of the tree decomposition under construction. U in turn, stands for the part of the graph to be decomposed below the bag containing S, and is always one of the connected components of $G \setminus S$. As explained in Chapter 25, we cannot afford storing U explicitly. Instead, we represent U in the data structure by an arbitrary vertex π (called the *pin*) belonging to it, and implicitly define U to be the connected component of $G \setminus S$ that contains π. Formally, the behavior of the subroutine `FindTD` is encapsulated in the following lemma; Herein, the *state* of the data structure $\mathcal{DS}$ is the content of its tables, see Section 29.1 (specifically Figure 29.2). A short description of the state is given in the next paragraphs.

Lemma 26.2. *There exists an algorithm that, given access to the data structure $\mathcal{DS}$ in a state such that $|S| \leq 2k+3$, computes a tree decomposition $\mathcal{T}$ of $G[U \cup S]$ of width at most $3k+4$ with S as a root bag, or correctly reports that that $\mathrm{tw}(G[U \cup S]) > k$. If the algorithm is run on $S = \emptyset$ and $U = V(G)$, then its running time is $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$.*

Operation / Query	**Description**
$\text{set}_\pi(v)$	sets v as current pin, π
$\text{get}_\pi()$	gives π
$\text{get}_S()$	gives the set S
$\text{insert}_S(v)$	inserts v to S
$\text{insert}_X(v)$	inserts v to X
$\text{insert}_F(v)$	inserts v to F
$\text{clear}_S()$	clears S (sets it to $\emptyset$)
$\text{clear}_X()$	clears X
$\text{clear}_F()$	clears F
findNeighborhood()	Gives neighborhood of U in S
findSSeparator()	gives a balanced S-separator
findNextPin()	gives a pair (π', l) such that in the connected component of $G - (S \cup X)$ containing π', there is no vertex in F, l is the size of the component, and the component is a largest such or $\perp$ if no such π' exists.

Figure 26.1: Data structure operations. Every operation takes time $O(c^k \log n)$. In our algorithm we will give the insert methods sets of vertices of size $k^{O(1)}$. Note that this can be implemented with a loop over the elements and that due to the size of the sets the delay can be hidden in the c^k-factor.

The data structure is initialized with $S = S_0$ and π set to an arbitrary vertex of $G \setminus S_0$; as we have assumed that $G \setminus S_0$ is connected, this gives $U = V(G) \setminus S_0$ after initialization. Therefore, Lemma 26.2 immediately yields Lemma 25.7.

A gentle introduction to the data structure

Before we proceed to the implementation of the subroutine `FindTD`, we give a quick description of the interface of the data structure $\mathcal{DS}$: what kind of queries and updates it supports, and what is the running time of their execution. The details of the data structure implementation will be given in Chapter 29.

The state of the data structure is, in addition to $G, k, \mathcal{T}$, three subsets of vertices, S, X and F, and the pin π with the restriction that $\pi \notin S$. S and π uniquely imply the set U, defined as the connected component of $G \setminus S$ that contains π. The intuition behind these sets and the pin is the following:

- S is the set that will serve as a root bag for some subtree,
- π is a vertex which indicates the current active component,
- U is the current active component, the connected component of $G \setminus S$ containing π,
- X is a balanced S-separator (of $G[S \cup U]$) and

- F is a set of vertices marking the connected components of $G[S \cup U] \setminus (S \cup X)$ as "finished".

The construction of the data structure $\mathcal{DS}$ is heavily based on the fact that we are provided with some tree decomposition of width $O(k)$. Given this tree decomposition, the data structure can be initialized in $O(c^k \cdot n)$ time for some $c \in \mathbb{N}$. At the moment of initialization we set $S = X = F = \emptyset$ and π to be an arbitrary vertex of G. During the run of the algorithm, the following updates can be performed on the data structure:

- insert/remove a vertex to/from S, X, or F;
- mark/unmark a vertex as a pin π.

All of these updates will be performed in $O(c^k \cdot \log n)$ time for some $c \in \mathbb{N}$.

Finding a better tree decomposition

The pseudocode of the algorithm `FindTD` is given as Algorithm 6. Its correctness is proven in Claim 26.3, and its time complexity is proven as Claim 26.4. The subroutine is provided with the data structure $\mathcal{DS}$, and the following invariants hold at each time the subroutine is called and exited:

- $S \subseteq V(G)$, $|S| \leq 2k+3$,
- π exists, is unique and $\pi \notin S$,
- $X = F = \emptyset$ and
- the state of the data structure is the same on exit as it was when the function was called.

The latter means that when we return, be it a tree decomposition or $\perp$, the algorithm that called `FindTD` will have S, X, F and π as they were before the call.

We now describe the consecutive steps of the algorithm `FindTD`; the reader is encouraged to follow these steps in the pseudocode, in order to be convinced that all the crucial, potentially expensive computations are performed by calls to the data structure.

First we apply query findSSeparator, which either finds a $\frac{1}{2}$-balanced S-separator in $G[S \cup U]$ of size at most $k+1$, or concludes that $\mathrm{tw}(G) > k$. The running time of this query is $k^{O(1)}$. If no such separator can be found, by Lemma 25.1 we infer that $\mathrm{tw}(G[S \cup U]) > k$ and we can terminate the procedure. Otherwise we are provided with such a separator `sep`, which we add to X in the data structure. Moreover, for a technical reason, we also add the pin π to `sep` (and thus also to X), so we end up with having $|\texttt{sep}| \leq k+2$.

The next step is a loop through the connected components of $G[S \cup U] \setminus (S \cup \texttt{sep})$. This part is performed using the query findNextPin. Query findNextPin, which runs in constant time, either finds an arbitrary vertex u of a connected component of $G[S \cup U] \setminus (S \cup X)$ that does not contain any vertex from F, or concludes that each of these components contains some vertex of F. After finding u, we mark u by putting it to F and proceed further, until all the components are marked. Having achieved this, we have obtained a list `pins`, containing exactly one vertex from each connected component of $G[S \cup U] \setminus (S \cup \texttt{sep})$. We remove all the vertices on this list from F, thus making F again empty.

It is worth mentioning that the query findNextPin not only returns some vertex u of a connected component of $G[S \cup U] \setminus (S \cup \texttt{sep})$ that does not contain any vertex from F, but also provides the size of this component as the second coordinate of the return value. Moreover, the components are being found in decreasing order with respect to sizes. In this algorithm we do not exploit this property, but it will be crucial for the linear-time algorithm.

The set X will no longer be used, so we remove all the vertices of `sep` from X, thus making it again empty. On the other hand, we add all the vertices from `sep` to S. The new set S obtained in this manner will constitute the new bag, of size at most $|S| + |\texttt{sep}| \leq 3k + 5$. We are left with computing the tree decompositions for the connected components below this bag, which are pinpointed by vertices stored in the list `pins`.

We iterate through the list `pins` and process the components one by one. For each vertex $u \in \texttt{pins}$, we set u as the new pin by unmarking the old one and marking u. Note that the set U gets redefined and now is the connected component containing considered u. First, we find the neighborhood of U in S. This is done using query findNeighborhood, which in $O(k)$ time returns either this neighborhood, or concludes that its cardinality is larger than $2k + 3$. However, as X was a $\frac{1}{2}$-balanced S-separator, it follows that this neighborhood will always be of size at most $2k + 3$ (a formal argument is contained in the proof of correctness). We continue with $S \cap N(U)$ as our new S and recursively call `FindTD` in order to decompose the connected component under consideration, with its neighborhood in S as the root bag of the constructed tree decomposition. `FindTD` either provides a decomposition by returning a pointer to its root bag, or concludes that no decomposition can be found. If the latter is the case, we may terminate the algorithm providing a negative answer.

After all the connected components are processed, we merge the obtained tree decompositions. For this, we use the function `build`(S, X, C) which, given sets of vertices S and X and a set of pointers C, constructs two bags $B = S$ and $B' = S \cup X$, makes C the children of B', B' the child of B and returns a pointer to B. This pointer may be returned from the whole subroutine, after doing a clean-up of the data structure.

Invariants

Now we show that the stated invariants indeed hold. Initially $S = X = F = \emptyset$ and $\pi \in V(G)$, so clearly the invariants are satisfied. If no S-separator is found, the algorithm returns without changing the data structure and hence the invariants trivially hold in this case. Since both X and F are empty or cleared before return

Algorithm 6: `FindTD`

```
Data: Data structure $\mathcal{DS}$
Output: Tree decomposition of width at most $3k + 4$ of $G[S \cup U]$ with $S$ as
        root bag or conclusion that $\mathrm{tw}(G) > k$.
    $\mathrm{old_S} \leftarrow \mathcal{DS}.\mathrm{get}_S()$
    $\mathrm{old}_\pi \leftarrow \mathcal{DS}.\mathrm{get}_\pi()$
    $\mathtt{sep} \leftarrow \mathcal{DS}.\mathrm{findSSeparator}()$
    if $\mathtt{sep} = \perp$ then
        return $\perp$        /* safe to return: the state not changed */
    end
    $\mathcal{DS}.\mathrm{insert}_X(\mathtt{sep})$
    $\mathcal{DS}.\mathrm{insert}_X(\pi)$
    $\mathtt{pins} \leftarrow \emptyset$
    while $(u, l) \leftarrow \mathcal{DS}.\mathrm{findNextPin}() \neq \perp$ do
        $\mathtt{pins}.\mathit{append}(u)$
        $\mathcal{DS}.\mathrm{insert}_F(u)$
    end
    $\mathcal{DS}.\mathrm{clear}_X()$
    $\mathcal{DS}.\mathrm{clear}_F()$
    $\mathcal{DS}.\mathrm{insert}_S(\mathtt{sep})$
    $\mathtt{bags} \leftarrow \emptyset$
    for $u \in \mathtt{pins}$ do
        $\mathcal{DS}.\mathrm{set}_\pi(u)$
        $\mathtt{bags}.\mathit{append}(\mathcal{DS}.\mathrm{findNeighborhood}())$
    end
    $\mathrm{children} \leftarrow \emptyset$
    for $u, b \in \mathtt{pins}, \mathtt{bags}$ do
        $\mathcal{DS}.\mathrm{set}_\pi(u)$
        $\mathcal{DS}.\mathrm{clear}_S()$
        $\mathcal{DS}.\mathrm{insert}_S(b)$
        $\mathrm{children}.\mathit{append}(\mathtt{FindTD}())$
    end
    $\mathcal{DS}.\mathrm{clear}_S()$
    $\mathcal{DS}.\mathrm{insert}_S(\mathrm{old_S})$
    $\mathcal{DS}.\mathrm{set}_\pi(\mathrm{old}_\pi)$
    if $\perp \in \mathrm{children}$ then
        return $\perp$        /* postponed because of rollback of $S$ and $\pi$ */
    end
    return $\mathtt{build}(\mathrm{old_S}, \mathtt{sep}, \mathrm{children})$
```

or recursing, $X = F = \emptyset$ holds. Furthermore, as S is reset to $\mathrm{old_S}$ (consult Algorithm 6 for the variable names used) and the pin to old_π before returning, it follows that the state of the data structure is reverted upon returning.

The size of $S = \emptyset$ is trivially less than $2k + 3$ when initialized. Assume that for some call to `FindTD` we have that $|\text{old}_S| \leq 2k+3$. When recursing, S is the neighborhood of some component C of $G[\text{old}_S \cup U] \setminus (\text{old}_S \cup \texttt{sep})$ (note that we refer to U before resetting the pin). This component is contained in some component C' of $G[\text{old}_S \cup U] \setminus \texttt{sep}$, and all the vertices of old_S adjacent to C must be contained in C'. Since `sep` is a $\frac{1}{2}$-balanced old_S-separator, we know that C' contains at most $\frac{1}{2}|\text{old}_S|$ vertices of old_S. Hence, when recursing we have that $|S| \leq \frac{1}{2}|\text{old}_S| + |\texttt{sep}| = \frac{1}{2}(2k+3) + k + 2 = 2k + \frac{7}{2}$ and, since $|S|$ is an integer, it follows that $|S| \leq 2k+3$.

Finally, we argue that the pin π is never contained in S. When we obtain the elements of `pins` (returned by query findNextPin) we know that $X = \texttt{sep}$ and the data structure guarantees that the pins will be from $G[\text{old}_S \cup U] \setminus (\text{old}_S \cup \texttt{sep})$. When recursing, $S = b \subseteq (\text{old}_S \cup \texttt{sep})$ and $\pi \in \texttt{pins}$, so it follows that $\pi \notin S$. Assuming $\pi \notin \text{old}_S$, it follows that π is not in S when returning, and our argument is complete. From here on we will safely assume that the invariants indeed hold.

Correctness

We now show that the algorithm `FindTD` actually does what we need, that is, provided that the treewidth of the input graph G is at most k, it outputs a tree decomposition of width at most $3k+4$ with S as a root bag.

Claim 26.3. *The algorithm* `FindTD` *is correct, that is*

(i) if $\text{tw}(G) \leq k$, `FindTD` *returns a valid tree decomposition of* $G[S \cup U]$ *of width at most* $3k+4$ *with* S *as a root bag and*

(ii) if `FindTD` *returns* $\perp$ *then* $\text{tw}(G) > k$.

Proof. We start by proving (ii). Suppose the algorithm returns $\perp$. This happens only if at some point we are unable to find a balanced S-separator for an induced subgraph $G' = G[S \cup U]$. By Lemma 25.1 the treewidth of G' is more than k. Hence $\text{tw}(G) > k$ as well.

To show (i) we proceed by induction on the height of the recursion tree. In the induction we prove that the algorithm creates a tree decomposition, and we therefore argue that the necessary conditions are satisfied, namely

- the bags have size at most $3k+5$,
- every vertex and every edge is contained in some bag,
- for each $v \in V(G)$ the subtree of bags containing v is connected, and finally
- S is the root bag.

The base case is at the leaf of the obtained tree decomposition, namely when $U \subseteq S \cup \texttt{sep}$. Then we return a tree decomposition containing two bags, B and B' where $B = \{S\}$ and $B' = \{S \cup \texttt{sep}\}$. Clearly, every edge and every vertex of $G[S \cup U] = G[S \cup \texttt{sep}]$ is contained in the tree decomposition. Furthermore, since the tree has size two, the connectivity requirement holds and finally, since $|S| \leq 2k+3$ (invariant) and $\texttt{sep} \leq k+2$ it follows that $|S \cup \texttt{sep}| \leq 3k+5$. Note that due to the definition of the base case, the algorithm will find no pins and hence it will not recurse further. Clearly, letting $B = \{S\}$ be the root bag fulfills the requirements.

The induction step is as follows. Assuming, by the induction hypothesis, that all recursive calls to `FindTD`() correctly returned what was promised, we now consider the case when we have successfully completed all the calls for each of the connected components (the line containing children.*append*(`FindTD`()) in Algorithm 6), and return $\texttt{build}(\text{old}_\text{S}, \texttt{sep}, \text{children})$.

Since $U \nsubseteq S \cup \texttt{sep}$, the algorithm have found some pins $\pi_1, \pi_2, \ldots, \pi_d$ and the corresponding components $C_1, C_2, \ldots, C_d$ in $G[S \cup U] \setminus (S \cup \texttt{sep})$. Let $N_i = N(C_i) \cap (S \cup \texttt{sep})$. By the induction hypothesis the algorithm gives us valid tree decompositions $\mathcal{T}_i$ of $G[N_i \cup C_i]$. Note that the root bag of $\mathcal{T}_i$ consists of the vertices in N_i. By the same argument as for the base case, the two bags $B = S$ and $B' = S \cup \texttt{sep}$ that we construct have appropriate sizes.

Let v be an arbitrary vertex of $S \cup U$. If $v \in S \cup \texttt{sep}$, then it is contained in B'. Otherwise there exists a unique i such that $v \in C_i$. It then follows from the induction hypothesis that v is contained in some bag of $\mathcal{T}_i$.

It remains to show that the edge property and the connectivity property hold. Let uv be an arbitrary edge of $G[S \cup U]$. If u and v both are in $S \cup \texttt{sep}$, then the edge is contained in B'. Otherwise, assume without loss of generality that u is in some component C_i. Then u and v are in $N_i \cup C_i$ and hence they are in some bag of $\mathcal{T}_i$ by the induction hypothesis.

Finally, for the connectivity property, let v be some vertex in $S \cup U$. If $v \notin S \cup \texttt{sep}$, then there is a unique i such that $v \in C_i$, hence we can apply the induction hypothesis. So assume that $v \in S \cup \texttt{sep} = B'$. Let A be some bag of $\mathcal{T}$ containing v. We will complete the proof by proving that there is a path of bags containing v from A to B'. If A is B or B', then this follows directly from the construction. Otherwise there exists a unique i such that A is a bag in $\mathcal{T}_i$. Observe that v is in N_i as it is in $S \cup \texttt{sep}$. By the induction hypothesis the bags containing v in $\mathcal{T}_i$ are connected and hence there is a path of bags containing v from A to the root bag R_i of $\mathcal{T}_i$. By construction B' contains v and the bags B' and R_i are adjacent. Hence there is a path of bags containing v from A to B' and as A was arbitrary chosen, this proves that the bags containing v form a connected subtree of the decomposition.

Now all we need to show is that S is the root bag of the tree decomposition we return, but that is precisely what is returned in the algorithm. We return (see the last line of Algorithm 6) the output of the function $\texttt{build}(\text{old}_\text{S}, \texttt{sep}, \text{children})$, but as described above, this function builds the tree decomposition consisting of the two bags old_S and $\text{old}_\text{S} \cup \texttt{sep}$, together with children, and outputs a pointer to old_S, which is exactly S.

This concludes the proof of Claim 26.3. □

Complexity

The final part needed to prove the correctness of Lemma 26.2, and thus conclude the algorithm of this section, is that the running time of `FindTD` does not exceed what was stated, namely $O(c^k \cdot n \log n)$. That is the last part of this section and is formalized in the following claim:

Claim 26.4. *The invocation of* `FindTD` *in the algorithm* $\texttt{Compress}_1$ *runs in* $O(c^k \cdot n \log n)$ *time for some* $c \in \mathbb{N}$*.*

Proof. First we simply observe that at each recursion step, we add the previous pin to S and create two bags. Since a vertex can only be added to S one time during the entire process, at most $2n$ bags are created. Hence the number of bags is bounded, and if we partition the used running time between the bags, charging each bag with at most $O(c^k \cdot \log n)$ time, it follows that `FindTD` runs in $O(c^k \cdot n \log n)$ time.

We now charge the bags. For a call $\mathcal{C}$ to `FindTD`, let B and B' be as previously with $R_1, \ldots, R_d$ the children of B'. Let $\mathcal{T}_i$ be the tree decomposition of the recursive call which has R_i as the root bag. We will charge B' and $R_1, \ldots, R_d$ for the time spent on $\mathcal{C}$. Notice that as R_i will correspond to B in the next recursion step, each bag will only be charged by one call to `FindTD`. We charge B' with everything in $\mathcal{C}$ not executed in the two loops iterating through the components, plus with the last call to findNextPin that returned $\perp$.

Now, since every update and query in the data structure is executed in $O(c^k \cdot \log n)$ time, and there is a constant number of queries charged to B', it follows that B' is charged with $O(c^k \cdot \log n)$ time. For each iteration in one of the loops we consider the corresponding π_i and charge the bag R_i with the time spent on this iteration. As all the operations in the loops can be performed in $O(c^k \cdot \log n)$ time, each R_i is charged with at most $O(c^k \cdot \log n)$ time.

Since our tree decomposition has at most $2n$ bags and each bag is charged with at most $O(c^k \cdot \log n)$ time, it follows that `FindTD` runs in $O(c^k \cdot n \log n)$ time and the proof is complete. $\square$

Chapter 27

Obtaining almost linear time

In this section we provide formal details of the proof of the following statement:

Theorem 46. *For every $\alpha \in \mathbb{N}$, there exists an algorithm which, given a graph G and an integer k, in $O(c_\alpha^k \cdot n \log^{(\alpha)} n)$ time for some $c_\alpha \in \mathbb{N}$ either computes a tree decomposition of G of width at most $5k+4$ or correctly concludes that $\mathrm{tw}(G) > k$.*

In the proof we give a sequence of algorithms $\mathtt{Alg}_\alpha$ for $\alpha = 2, 3, \ldots$; $\mathtt{Alg}_1$ has been already presented in the previous section. Each $\mathtt{Alg}_\alpha$ in fact solves a slightly more general problem than stated in Theorem 46, in the same manner as $\mathtt{Alg}_1$ solved a more general problem than the one stated in Theorem 45. Namely, every algorithm $\mathtt{Alg}_\alpha$ gets as input a connected graph G, an integer k and a subset of vertices S_0 such that $|S_0| \le 4k+3$ and $G \setminus S_0$ is connected, and either concludes that $\mathrm{tw}(G) > k$ or constructs a tree decomposition of width at most $5k+4$ with S_0 as the root bag. The running time of $\mathtt{Alg}_\alpha$ is $O(c_\alpha^k \cdot n \log^{(\alpha)} n)$ for some $c_\alpha \in \mathbb{N}$; hence, in order to prove Theorem 46 we can again apply $\mathtt{Alg}_\alpha$ to every connected component of G separately, using $S_0 = \emptyset$.

The algorithms $\mathtt{Alg}_\alpha$ are constructed inductively; by that we mean that $\mathtt{Alg}_\alpha$ will call $\mathtt{Alg}_{\alpha-1}$, which again will call $\mathtt{Alg}_{\alpha-2}$, and all the way until $\mathtt{Alg}_1$, which was given in the previous section. Let us remark that a closer examination of our algorithms in fact shows that the constants c_α in the bases of the exponents of consecutive algorithms can be bounded by some universal constant. However, of course the constant factor hidden in the O-notation depends on α.

In the following we present a quick outline of what will be given in this section. For $\alpha = 1$, we refer to the previous section, and for $\alpha > 1$, $\mathtt{Alg}_\alpha$ and $\mathtt{Compress}_\alpha$ are described in this section, in addition to the subroutine `FindPartialTD`.

- $\mathtt{Alg}_\alpha$ takes as input a graph G, an integer k and a vertex set S_0 with similar assumptions as in the previous section, and returns a tree decomposition $\mathcal{T}$ of G of width at most $5k+4$ with S_0 as the root bag. The algorithm is almost exactly as $\mathtt{Alg}_1$ given as Algorithm 4, except that it uses $\mathtt{Compress}_\alpha$ for the compression step.

- $\mathtt{Compress}_\alpha$ is an advanced version of $\mathtt{Compress}_1$ (see Algorithm 5), it allows S_0 to be of size up to $4k+3$ and gives a tree decomposition of width at most

$5k+4$ in time $O({c'_\alpha}^k \cdot n \log^{(\alpha)} n)$ for some $c'_\alpha \in \mathbb{N}$. It starts by initializing the data structure, and then it calls `FindPartialTD`, which returns a tree decomposition $\mathcal{T}'$ of an induced subgraph $G' \subseteq G$. The properties of G' and $\mathcal{T}'$ are as follows. All the connected components $C_1, \ldots C_p$ of $G \setminus V(G')$ are of size less than $\log n$. Furthermore, for every connected component C_j, the neighborhood $N(C_j)$ in G is contained in a bag of $\mathcal{T}'$. Intuitively, this ensures that we are able to construct a tree decomposition of C_j and attach it to $\mathcal{T}'$ without blowing up the width of $\mathcal{T}'$. More precisely, for every connected component C_j, the algorithm constructs the induced subgraph $G_j = G[C_j \cup N(C_j)]$ and calls $\texttt{Alg}_{\alpha-1}$ on G_j, k, and $N(C_j)$. The size of $N(C_j)$ will be bounded by $4k+3$, making the recursion valid with respect to the invariants of $\texttt{Alg}_\alpha$. If this call returned a tree decomposition $\mathcal{T}_j$ with a root bag $N(C_j)$, we can conveniently attach $\mathcal{T}_j$ to $\mathcal{T}'$; otherwise we conclude that $\mathrm{tw}(G[C_j \cup N(C_j)]) > k$ so $\mathrm{tw}(G) > k$ as well.

- `FindPartialTD` differs from `FindTD` in two ways. First, we use the fact that when enumerating the components separated by the separator using query findNextPin, these components are identified in the descending order of cardinalities. We continue the construction of partial tree decomposition in the identified components only as long as they are of size at least $\log n$, and we terminate the enumeration when we encounter the first smaller component. It follows that all the remaining components are smaller then $\log n$; these remainders are exactly the components $C_1, \ldots C_p$ that are left not decomposed by $\texttt{Alg}_\alpha$, and on which $\texttt{Alg}_{\alpha-1}$ is run.

  The other difference is that the data structure has a new *flag*, `whatsep`, which is set to either u or s and is alternated between calls. If $\texttt{whatsep} = s$, we use the same type of separator as `FindTD` did, namely findSSeparator, but if $\texttt{whatsep} = u$, then we use the (new) query findUSeparator. Query findUSeparator, instead of giving a balanced S-separator, provides a $\frac{8}{9}$-balanced U-separator, that is, a separator that splits the whole set U of vertices to be decomposed in a balanced way. Using the fact that on every second level of the decomposition procedure the whole set of available vertices shrinks by a constant fraction, we may for example observe that the resulting partial tree decomposition will be of logarithmic depth. More importantly, it may be shown that the total number of constructed bags is at most $O(n/\log n)$ and hence we can spend $O(c_\alpha^k \cdot \log n)$ time constructing each bag and still obtain running time linear in n.

In all the algorithms that follow we assume that the cardinality of the edge set is at most k times the cardinality of the vertex set, because otherwise we may immediately conclude that treewidth of the graph under consideration is larger than k and terminate the algorithm.

27.1 The main procedure

The procedure $\texttt{Alg}_\alpha$ works exactly as $\texttt{Alg}_1$, with the exception that it applies Lemma 25.6 for parameter $5k+4$ instead of $3k+4$, and calls recursively $\texttt{Alg}_\alpha$ and $\texttt{Compress}_\alpha$ instead of $\texttt{Alg}_1$ and $\texttt{Compress}_1$. The running time analysis is exactly the same, hence we omit it here.

27.2 Compression algorithm

The following lemma defines the behavior of the compression algorithm $\texttt{Compress}_\alpha$.

Lemma 27.1. *For every integer $\alpha \geq 1$ there exists an algorithm, which on input $G, k, S_0, \mathcal{T}_{apx}$, where*

(i) $S_0 \subseteq V(G)$, $|S_0| \leq 4k+3$,

(ii) G and $G \setminus S_0$ are connected and

(iii) $\mathcal{T}_{apx}$ is a tree decomposition of G of width at most $O(k)$

in $O({c'_\alpha}^k \cdot n \log^{(\alpha)} n)$ time for some $c'_\alpha \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 5k+4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.

The outline of the algorithm $\texttt{Compress}_\alpha$ for $\alpha > 1$ is given as Algorithm 7. Having initialized the data structure using $\mathcal{T}_{\text{apx}}$, the algorithm asks $\texttt{FindPartialTD}$ for a partial tree decomposition $\mathcal{T}'$, and then the goal is to decompose the remaining small components and attach the resulting tree decompositions in appropriate places of $\mathcal{T}'$.

First we traverse $\mathcal{T}'$ in linear time and store information on where each vertex appearing in $\mathcal{T}'$ is forgotten in $\mathcal{T}'$. More precisely, we compute a map $\texttt{forgotten} : V(G) \to V(\mathcal{T}') \cup \{\bot\}$, where for every vertex v of G we either store $\bot$ if it is not contained in $\mathcal{T}'$, or we remember the top-most bag B_i of $\mathcal{T}'$ such that $v \in B_i$ (the connectivity requirement of the tree decomposition ensures that such B_i exists and is unique). The map $\texttt{forgotten}$ may be very easily computed via a DFS traversal of the tree decomposition: when accessing a child node i from a parent i', we put $\texttt{forgotten}(v) = i$ for each $v \in B_i \setminus B_{i'}$. Moreover, for every $v \in B_r$, where r is the root node, we put $\texttt{forgotten}(v) = r$. Clearly, all the vertices not assigned a value in $\texttt{forgotten}$ in this manner, are not contained in any bag of $\mathcal{T}'$, and we put value $\bot$ for them. Let W be the set of vertices contained in $\mathcal{T}'$, i.e., $W = \bigcup_{i \in V(\mathcal{T}')} B_i$.

Before we continue, let us show how the map $\texttt{forgotten}$ will be used. Suppose that we have some set $Y \subseteq W$, and we have a guarantee that there exists a node i of $\mathcal{T}'$ such that B_i contains the whole Y. We claim the following: then one of the bags associated with $\texttt{forgotten}(v)$ for $v \in Y$ contains the whole Y. Indeed, take the path from i to the root of the tree decomposition $\mathcal{T}'$, and consider the last node

i' of this path whose bag contains the whole Y. It follows that $i' = \texttt{forgotten}(v)$ for some $v \in Y$ and $Y \subseteq B_{i'}$, so the claim follows. Hence, we can locate the bag containing Y in $O(k^{O(1)} \cdot |Y|)$ time by testing each of $|Y|$ candidate nodes $\texttt{forgotten}(v)$ for $v \in Y$.

The next step of the algorithm is locating the vertices which has not been accounted for, i.e., those assigned $\perp$ by `forgotten`. The reason each of these vertices has not been put into the tree decomposition, is precisely because the size of its connected component C of $G \setminus W$, is smaller than $\log n$. The neighborhood of this component in G is $N(C)$, and this neighborhood is guaranteed to be of size at most $4k + 3$ and contained in some bag of $\mathcal{T}'$ (a formal proof of this fact will be given when presenting the algorithm `FindPartialTD`, i.e., in Lemma 27.2).

Let $C_1, C_2, \ldots, C_p$ be all the connected components of $G \setminus W$, i.e., the connected components *outside* the obtained partial tree decomposition $\mathcal{T}'$. To complete the partial tree decomposition into a tree decomposition, for every connected component C_j, we construct a graph $G_j = G[C_j \cup N(C_j)]$ that we then aim to decompose. These graphs may be easily identified and constructed in $O(k^{O(1)} \cdot n)$ time using depth-first search as follows.

We iterate through $V(G)$, and for each vertex v such that $\texttt{forgotten}(v) = \perp$ and v was not visited yet, we apply a depth-first search on v to identify its component C. During this depth-first search procedure, we terminate searching and return from a recursive call whenever we encounter a vertex from W. In this manner we identify the whole component C, and all the visited vertices of W constitute exactly $N(C)$. Moreover, the edges traversed while searching are exactly those inside C or between C and $N(C)$. To finish the construction of G_j, it remains to identify edges between vertices of $N(C)$. Recall that we have a guarantee that $N(C) \subseteq W$ and $N(C)$ is contained in some bag of $\mathcal{T}'$. Using the map `forgotten` we can locate some such bag in $O(k^{O(1)})$ time, and in $O(k^{O(1)})$ time check which vertices of $N(C)$ are adjacent in it, thus finishing the construction of G_j. Observe that during the presented procedure we traverse each edge of the graph at most once, and for each of at most n components C we spend $O(k^{O(1)})$ time on examination of $N(C)$. It follows that the total running time is $O(k^{O(1)} \cdot n)$.

Having constructed G_j, we run the algorithm $\texttt{Alg}_{\alpha-1}$ on G_j using $S_0 = N(C_j)$. Note that in this manner we have that both G_j and $G_j \setminus S_0$ are connected, which are requirements of the algorithm $\texttt{Alg}_{\alpha-1}$. If $\texttt{Alg}_{\alpha-1}$ concluded that $\operatorname{tw}(G_j) > k$, then we can consequently answer that $\operatorname{tw}(G) > k$ since G_j is an induced subgraph of G. On the other hand, if $\texttt{Alg}_{\alpha-1}$ provided us with a tree decomposition $\mathcal{T}_j$ of G_j having $N(C_j)$ as the root bag, then we may simply attach this root bag as a child of the bag of $\mathcal{T}'$ that contains the whole $N(C_j)$. Any such bag can be again located in $O(k^{O(1)})$ time using the map `forgotten`.

Algorithm 7: $\texttt{Compress}_\alpha$

```
Input: Connected graph $G$, $k \in \mathbb{N}$, a set $S_0$ s.t. $|S_0| \leq 4k+3$ and $G \setminus S_0$ is
       connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ with $\mathrm{w}(\mathcal{T}_{\text{apx}}) = O(k)$
Output: Tree decomposition of $G$ of width at most $5k+4$ with $S_0$ as the
        root bag, or conclusion that $\mathrm{tw}(G) > k$.

  Initialize data structure $\mathcal{DS}$ with $G, k, S_0, \mathcal{T}_{\text{apx}}$
  $\mathcal{T}' \leftarrow$ FindPartialTD()
  if $\mathcal{T}' = \perp$ then
      return $\perp$
  end
  Create the map forgotten : $V(G) \to V(\mathcal{T}')$ using a DFS traversal of $\mathcal{T}'$
  Construct components $C_1, C_2, \ldots, C_p$ of $G \setminus W$, and graphs
  $G_j = G[C_j \cup N(C_j)]$ for $j = 1, 2, \ldots, p$
  for $j = 1, 2, \ldots, p$ do
      $\mathcal{T}_j \leftarrow \texttt{Alg}_{\alpha-1}$ on $G_j, k, N(C_j)$
      if $\mathcal{T}_j = \perp$ then
          return $\perp$
      end
      Locate a node $i$ of $\mathcal{T}'$ s.t. $N(C_j) \subseteq B_i$, by checking forgotten($v$) for
      each $v \in N(C_j)$
      Attach the root of $\mathcal{T}_j$ as a child of $i$
  end
  return $\mathcal{T}'$
```

Correctness and complexity

In this section we prove Lemma 27.1 and Theorem 46, and we proceed by induction on α. To this end we will assume the correctness of Lemma 27.2, which will be proved later, and which describes behavior of the subroutine `FindPartialTD()`.

For the base case, $\alpha = 1$, we use $\texttt{Compress}_1$ given as Algorithm 5. When its correctness was proved we assumed $|S_0| \leq 2k+3$ and this is no longer the case. However, if $\texttt{Alg}_1$ is applied with $|S_0| \leq 4k+3$ it will conclude that $\mathrm{tw}(G) > k$ or give a tree decomposition of width at most $5k+4$. The reason is as follows; Assume that `FindTD` is applied with the invariant $|S| \leq 4k+3$ instead of $2k+3$. By the same argument as in the original proof this invariant will hold, since $\frac{1}{2}(4k+3) + k + 2 \leq 4k+3$. The only part of the correctness (and running time analysis) affected by this change is the width of the returned decomposition, and when the algorithm adds the separator to S it creates a bag of size at most $(4k+3)+(k+2) = 5k+5$ and hence our argument for the base case is complete. For the induction step, suppose that the theorem and lemma hold for $\alpha - 1$. We show that $\texttt{Compress}_\alpha$ is correct and runs in $O({c'_\alpha}^k \cdot n \log^{(\alpha)} n)$ time. This immediately implies correctness and complexity of $\texttt{Alg}_\alpha$, in the same manner as in Chapter 26.

To prove correctness of $\texttt{Compress}_\alpha$, suppose that $\mathcal{T}'$ is a valid tree decomposition for some $G' \subseteq G$ that we have obtained from `FindPartialTD`. Observe that if

$\mathcal{T}' = \bot$, then $\mathrm{tw}(G) > k$ by Lemma 27.2. Otherwise, let $C_1, \ldots, C_p$ be the connected components of $G \setminus W$, and let $G_j = G[C_j \cup N(C_j)]$ for $j = 1, 2, \ldots, p$. Let $\mathcal{T}_j$ be the tree decompositions obtained from application of the algorithm $\mathtt{Alg}_{\alpha-1}$ on graphs G_j. If $\mathcal{T}_j = \bot$ for any j, we infer that $\mathrm{tw}(G_j) > k$ and, consequently, $\mathrm{tw}(G) > k$. Assume then that for all the components we have indeed obtained valid tree decompositions, with $N(C_j)$ as root bags. It can be easily seen that since $N(C_j)$ separates C_j from the rest of G, then attaching the root of $\mathcal{T}_j$ as a child of any bag containing the whole $N(C_j)$ gives a valid tree decomposition; the width of this tree decomposition is the maximum of widths of $\mathcal{T}'$ and $\mathcal{T}_j$, which is at most $5k + 3$. Moreover, if we perform this operation for all the components C_j, then all the vertices and edges of the graph will be contained in some bag of the obtained tree decomposition.

We now proceed to the time complexity of $\mathtt{Compress}_\alpha$. The first thing done by the algorithm is the initialization of the data structure and running `FindPartialTD` to obtain $\mathcal{T}'$. Application of `FindPartialTD` takes $O(c^k n)$ time by Lemma 27.2, and so does initialization of the data structure (see Chapter 29). As discussed, creation of the `forgotten` map and construction of the graphs G_j takes $O(k^{O(1)} \cdot n)$ time.

Now, the algorithm applies $\mathtt{Alg}_{\alpha-1}$ to each graph G_j. Let n_j be the number of vertices of G_j. Note that

$$\sum_{j=1}^{p} n_j = \sum_{j=1}^{p} |C_j| + \sum_{j=1}^{p} |N(C_j)| \leq n + p \cdot (4k+3) \leq (5k+3)n.$$

Moreover, as $n_j \leq \log n + (4k+3)$, it follows from concavity of $t \to \log^{(\alpha-1)} t$ that

$$\log^{(\alpha-1)} n_j \leq \log^{(\alpha-1)}(\log n + (4k+3)) \leq \log^{(\alpha)} n + \log^{(\alpha-1)}(4k+3).$$

By the induction hypothesis, the time complexity of $\mathtt{Alg}_{\alpha-1}$ on G_j is $O({c'_{\alpha-1}}^k \cdot n_j \log^{(\alpha-1)} n_j)$ for some $c'_{\alpha-1} \in \mathbb{N}$, hence we spend $O({c'_{\alpha-1}}^k \cdot n_j \log^{(\alpha-1)} n_j)$ time for G_j. Attaching each decomposition $\mathcal{T}_j$ to $\mathcal{T}'$ can be done in $O(k^{O(1)})$ time.

Let C_α denote the complexity of $\mathtt{Compress}_\alpha$ and A_α the complexity of $\mathtt{Alg}_\alpha$. By applying the induction hypothesis and by taking $c'_\alpha > \max\{c, c_{\alpha-1}\}$ in order to hide polynomial factors depending on k, we analyze the complexity of $\mathtt{Compress}_\alpha$:

$$\begin{aligned}
C_\alpha(n,k) &= O(c^k \cdot n) + \sum_{j=1}^{n} A_{\alpha-1}(n_j, k) \\
&= O(c^k \cdot n) + \sum_{j=1}^{p} O(c^k_{\alpha-1} \cdot n_j \log^{(\alpha-1)} n_j) \\
&\leq O(c^k \cdot n) + \sum_{j=1}^{p} O(c^k_{\alpha-1} \cdot n_j(\log^{(\alpha)} n + \log^{(\alpha-1)}(4k+3))) \\
&= O({c'_\alpha}^k \cdot n) + \sum_{j=1}^{p} O(c^k_{\alpha-1} \cdot n_j \log^{(\alpha)} n) \\
&\leq O({c'_\alpha}^k \cdot n) + (5k+3)n \cdot O(c^k_{\alpha-1} \cdot \log^{(\alpha)} n) = O({c'_\alpha}^k \cdot n \log^{(\alpha)} n).
\end{aligned}$$

We conclude that $\texttt{Compress}_\alpha$ is both correct and that it runs in $O({c'_\alpha}^k \cdot n \log^{(\alpha)} n)$ time for some $c'_\alpha \in \mathbb{N}$.

Recall that the only difference between $\texttt{Alg}_\alpha$ and $\texttt{Alg}_1$ is which compression subroutine is called. Hence, the correctness of $\texttt{Alg}_\alpha$ follows in the same manner as the correctness of $\texttt{Alg}_1$; see the previous section. The time analysis of $\texttt{Alg}_\alpha$ is also very similar to the time analysis of $\texttt{Alg}_1$ as given in Section 26.1; taking $c_\alpha > c'_\alpha$, we obtain a time complexity of $O(c_\alpha^k n \log^{(\alpha)} n)$ using Equations 26.1 and 26.2 with c_α and c'_α instead of c_1 and c. And hence our induction step is complete and the correctness of Lemma 25.7 and Theorem 46 follows. The only assumption we made was that of the correctness of Lemma 27.2, which will be given immediately.

27.3 Obtaining a partial tree decomposition

The following lemma describes behavior of the subroutine `FindPartialTD`.

Lemma 27.2. *There exists an algorithm that, given data structure $\mathcal{DS}$ in a state such that $|S| \leq 4k+3$ if* `whatsep` $= s$ *or $|S| \leq 3k+2$ if* `whatsep` $= u$*, in time $O(c^k n)$ either concludes that $\mathrm{tw}(G[U \cup S]) > k$, or give a tree decomposition $\mathcal{T}'$ of $G' \subseteq G[U \cup S]$ such that*

- *the width of the decomposition is at most $5k+4$ and S is its root bag;*
- *for every connected component C of $G[U \cup S] \setminus V(G')$, the size of the component is less than $\log n$, its neighborhood is of size at most $4k+3$, and there is a bag in the decomposition $\mathcal{T}'$ containing this whole neighborhood.*

The pseudocode of `FindPartialTD` is presented as Algorithm 8. Recall Figure 26.1 for the operations on the data structure. The algorithm proceeds very similarly to the subroutine `FindTD`, given in Chapter 29. The main differences are the following.

- We alternate usage of findSSeparator and findUSeparator between the levels of the recursion to achieve that the resulting tree decomposition is also balanced. A special flag in the data structure, `whatsep`, that can be set to s or u, denotes whether we are currently about to use findSSeparator or findUSeparator, respectively. When initializing the data structure we set `whatsep` $= s$, so we start with finding a balanced S-separator.
- When identifying the next components using query findNextPin, we stop when a component of size less than $\log n$ is discovered. The remaining components are left without being decomposed.

The new query findUSeparator, provided that we have the data structure with S and π distinguished, gives a $\frac{8}{9}$-balanced separator of U in $G[U]$ of size at most $k+1$. That is, it returns a subset Y of vertices of U, with cardinality at most $k+1$, such that every connected component of $G[U] \setminus Y$ has at most $\frac{8}{9}|U|$ vertices. If such a separator cannot be found (which is signalized by $\perp$), we may safely conclude that $\mathrm{tw}(G[U]) > k$ and, consequently $\mathrm{tw}(G) > k$. The running time of query findUSeparator is $O(c^k \cdot \log n)$.

We would like to remark that the usage of balanced U-separators make it not necessary to add the pin to the obtained separator. Recall that this was a technical trick that was used in Chapter 26 to ensure that the total number of bags of the decomposition was linear.

Correctness

The invariants of Algorithm 8 are as for Algorithm 6, except for the size of S, in which case we distinguish whether `whatsep` is s or u. In the case of s the size of S is at most $4k+3$ and for u the size of S is at most $3k+2$.

If `whatsep` $= u$ then, since $|S| \leq 3k+2$ and we add an U-separator of size at most $k+1$ and make this our new S, the size of the new S will be at most $4k+3$ and we set `whatsep` $= s$. For every component C on which we recurse, the cardinality of its neighborhood (S at the moment of recursing) is therefore bounded by $4k+3$. So the invariant holds when `whatsep` $= u$.

We now show that the invariant holds when `whatsep` $= s$. Now $|\mathrm{old}_\mathrm{S}| \leq 4k+3$. We find $\frac{1}{2}$-balanced S-separator `sep` of size at most $k+1$. When recursing, the new S is the neighborhood of some component C of $G[\mathrm{old}_\mathrm{S} \cup U] \setminus (\mathrm{old}_\mathrm{S} \cup \texttt{sep})$ (note that we refer to U before resetting the pin). This component is contained in some component C' of $G[\mathrm{old}_\mathrm{S} \cup U] \setminus \texttt{sep}$, and all the vertices of old_S adjacent to C must be contained in C'. Since `sep` is a $\frac{1}{2}$-balanced old_S-separator, we know that C' contains at most $\frac{1}{2}|\mathrm{old}_\mathrm{S}|$ vertices of old_S. Hence, when recursing we have that $|S| \leq \frac{1}{2}|\mathrm{old}_\mathrm{S}| + |\texttt{sep}| = \frac{1}{2}(4k+3) + k + 1 = 3k + \frac{5}{2}$ and, since $|S|$ is an integer, it follows that $|S| \leq 3k+2$. Hence, the invariant also hold when `whatsep` $= s$.

Note that in both the checks we did not assume anything about the size of the component under consideration. Therefore, it also holds for components on which we do not recurse, i.e., those of size at most $\log n$, that the cardinalities of their neighborhoods will be bounded by $4k+3$.

The fact that the constructed partial tree decomposition is a valid tree decomposition of the subgraph induced by vertices contained in it, follows immediately from the construction, similarly as in Chapter 26. A simple inductive argument also shows that the width of this tree decomposition is at most $5k+4$: at each step of the construction, we add two bags of sizes at most $(4k+3)+(k+1) \leq 5k+5$ to the obtained decompositions of the components, which by inductive hypothesis are of width at most $5k+4$.

Finally, we show that every connected component of $G[S \cup U] \setminus V(G')$ has size at most $\log n$ and that the neighborhood of each of these connected component is contained in some bag on the partial tree decomposition $\mathcal{T}'$. First, by simply breaking out of the loop shown in Algorithm 8 at the point we get a pair (π, l) such that $l < \log n$, we are guaranteed that the connected component of $G[S \cup U] \setminus \texttt{sep}$ containing π has size less than $\log n$, and so does every other connected component of $G[S \cup U]$ not containing a vertex from F and which has not been visited by $\mathcal{DS}$.findNextPin(). Furthermore, since immediately before we break out of the loop due to small size we add $S \cup \texttt{sep}$ to a bag, we have ensured that the neighborhood of any such small component is contained in this bag. The bound on the size of this neighborhood has been already argued.

Complexity

Finally, we show that the running time of the algorithm is $O(c^k \cdot n)$. The data structure operations all take time $O(c^k \log n)$ and we get the data structure $\mathcal{DS}$ as input.

The following combinatorial lemma will be helpful to bound the number of bags in the tree decomposition produced by `FindPartialTD`. We aim to show that the tree decomposition $\mathcal{T}'$ contains at most $O(n/\log n)$ bags, so we will use the lemma with $\mu(i) = w_i/\log n$, where i is a node in a tree decomposition $\mathcal{T}'$ and w_i is the number of vertices in $G[U]$ when i is added to $\mathcal{T}'$. Having proven the lemma, we can show that the number of bags is bounded by $O(\mu(r)) = O(n/\log n)$, where r is the root node of $\mathcal{T}'$.

Lemma 27.3. *Let T be a rooted tree with root r. Assume that we are given a measure $\mu : V(T) \to \mathbb{R}$ with the following properties:*

(i) $\mu(v) \geq 1$ for every $v \in V(T)$,

(ii) for every vertex v, let $v_1, v_2, \ldots, v_p$ be its children, we have that $\sum_{i=1}^{p} \mu(v_i) \leq \mu(v)$, and

(iii) there exists a constant $0 < C < 1$ such that for for every two vertices v, v' such that v is a parent of v', it holds that $\mu(v') \leq C \cdot \mu(v)$.

Then $|V(T)| \leq \left(1 + \frac{1}{1-C}\right)\mu(r) - 1$.

Proof. We prove the claim by induction with respect to the size of $V(T)$. If $|V(T)| = 1$, the claim trivially follows from property (i). We proceed to the induction step.

Let $v_1, v_2, \ldots, v_p$ be the children of r and let $T_1, T_2, \ldots, T_p$ be subtrees rooted in $v_1, v_2, \ldots, v_p$, respectively. If we apply the induction hypothesis to trees $T_1, \ldots, T_p$, we infer that for each $i = 1, 2, \ldots, p$ we have that $|V(T_i)| \leq \left(1 + \frac{1}{1-C}\right)\mu(v_i) - 1$. By summing the inequalities we infer that:

$$|V(T)| \;\leq\; 1 - p + \left(1 + \frac{1}{1-C}\right)\sum_{i=1}^{p} \mu(v_i).$$

We now consider two cases. Assume first that $p \geq 2$; then:

$$|V(T)| \;\leq\; 1 - 2 + \left(1 + \frac{1}{1-C}\right)\sum_{i=1}^{p} \mu(v_i) \leq \left(1 + \frac{1}{1-C}\right)\mu(r) - 1,$$

and we are done. Assume now that $p = 1$; then

$$\begin{aligned}|V(T)| &\leq \left(1+\frac{1}{1-C}\right)\mu(v_1) \leq C\left(1+\frac{1}{1-C}\right)\mu(r)\\ &= \left(1+\frac{1}{1-C}\right)\mu(r) - (2-C)\mu(r) \leq \left(1+\frac{1}{1-C}\right)\mu(r) - 1,\end{aligned}$$

and we are done as well. □

We now prove the following claim.

Claim 27.4. *The partial tree decomposition $\mathcal{T}'$ contains at most $42n/\log n$ nodes.*

Proof. Let us partition the set of nodes $V(\mathcal{T}')$ into two subsets. At each recursive call of `FindPartialTD`, we create two nodes: one associated with the bag old_S, and one associated with the bag $\mathrm{old}_S \cup \mathtt{sep}$. Let I_{small} be the set of nodes associated with bags old_S, and let I_{large} the the set of remaining nodes, associated with bags $\mathrm{old}_S \cup \mathtt{sep}$. As bags are always constructed in pairs, it follows that $|I_{\mathrm{small}}| = |I_{\mathrm{large}}| = \frac{1}{2}|V(\mathcal{T}')|$. Therefore, it remains to establish a bound on $|I_{\mathrm{small}}|$.

We now further partition I_{small} into three parts: I^{s}_{small}, $I^{u,\mathrm{int}}_{\mathrm{small}}$, and $I^{u,\mathrm{leaf}}_{\mathrm{small}}$:

- I^{s}_{small} consists of all the nodes created in recursive calls where $\mathtt{whatsep} = s$.
- $I^{u,\mathrm{leaf}}_{\mathrm{small}}$ consists of all the nodes created in recursive calls where $\mathtt{whatsep} = u$, and moreover the algorithm did not make any more recursive calls to `FindTD` (in other words, all the components turned out to be of size smaller than $\log n$).
- $I^{u,\mathrm{int}}_{\mathrm{small}}$ consists of all the remaining nodes created in recursive calls where $\mathtt{whatsep} = u$, that is, such that the algorithm made at least one more call to `FindTD`.

We aim at bounding the size of each of the sets I^{s}_{small}, $I^{u,\mathrm{int}}_{\mathrm{small}}$, and $I^{u,\mathrm{leaf}}_{\mathrm{small}}$ separately.

We first claim that $|I^{u,\mathrm{leaf}}_{\mathrm{small}}| \leq n/\log n$. For each node in $I^{u,\mathrm{leaf}}_{\mathrm{small}}$, consider the set of vertices strictly below the bag. By construction, such a set consists of a number of components, each with less than $\log n$ vertices. However, as a recursive call to create the bag was made at the parent, the total size of these components must be at least $\log n$. Now observe that these associated sets are pairwise disjoint for the bags in $I^{i,\mathrm{leaf}}_{\mathrm{small}}$; so it follows that $|I^{u,\mathrm{leaf}}_{\mathrm{small}}| \leq n/\log n$.

We now claim that $|I^{u,\mathrm{int}}_{\mathrm{small}}| \leq |I^{s}_{\mathrm{small}}|$. Indeed, if with every node $i \in I^{u,\mathrm{int}}_{\mathrm{small}}$ we associate any of its grandchildren belonging to I^{s}_{small}, whose existence is guaranteed by the definition of $I^{u,\mathrm{int}}_{\mathrm{small}}$, we obtain an injective map from $I^{u,\mathrm{int}}_{\mathrm{small}}$ into I^{s}_{small}.

We are left with bounding $|I^{s}_{\mathrm{small}}|$. We use the following notation. For a node $i \in V(\mathcal{T}')$, let w_i be the number of vertices strictly below i in the tree decomposition $\mathcal{T}'$, also counting the vertices outside the tree decomposition. Note that by the construction it immediately follows that $w_i \geq \log n$ for each $i \in I_{\mathrm{small}}$.

We now make use of Lemma 27.3. Recall that vertices of I^{s}_{small} are exactly those that are in levels whose indices are congruent to 1 modulo 4, where the root has level 1; in particular, $r \in I^s_{\text{small}}$. We define a rooted tree T as follows. The vertex set of T is I^s_{small}, and for every two nodes $i, i' \in I^s_{\text{small}}$ such that i' is an ancestor of i exactly 4 levels above (grand-grand-grand-parent), we create an edge between i and i'. It is easy to observe that T created in this manner is a rooted tree, with r as the root.

We can now construct a measure $\mu : V(T) \to \mathbb{R}$ by taking $\mu(i) = w_i / \log n$. Let us check that μ satisfies the assumptions of Lemma 27.3 for $C = \frac{8}{9}$. Property (i) follows from the fact that $w_i \geq \log n$ for every $i \in I_{\text{small}}$. Property (ii) follows from the fact that the parts of the components on which the algorithm recurses below the bags are always pairwise disjoint. Property (iii) follows from the fact that between every pair of parent, child in the tree T we have used a $\frac{8}{9}$-balanced U-separator. Application of Lemma 27.3 immediately gives that $|I^s_{\text{small}}| \leq 10n/\log n$.

As $|I^s_{\text{small}}| \leq 10n/\log n$, we have that $|I^{u,\text{int}}_{\text{small}}| \leq |I^s_{\text{small}}| \leq 10n/\log n$, thus $|I_{\text{small}}| \leq |I^s_{\text{small}}| + |I^{u,inte}_{\text{small}}| + |I^{u,leaf}_{\text{small}}| \leq 21n/\log n$. Hence $|V(\mathcal{T}')| \leq 2 \cdot |I_s m| \leq 42n/\log n$. $\square$

To conclude the running time analysis of `FindPartialTD`, we provide a similar charging scheme as in Chapter 26. More precisely, we charge every node of $\mathcal{T}'$ with $O(c^k \cdot \log n)$ running time; Claim 27.4 ensures us that then the total running time of the algorithm is then $O(c^k \cdot n)$.

Let $B = \text{old}_S$ and $B' = \text{old}_S \cup \texttt{sep}$ be the two bags constructed at some call of `FindPartialTD`. All the operations in this call, apart from the two loops over the components, take $O(c^k \cdot \log n)$ time and are charged to B'. Moreover, the last call of findNextPin, when a component of size smaller than $\log n$ is discovered, is also charged to B'. As this call takes $O(1)$ time, B' is charged with $O(c^k \cdot \log n)$ time in total.

We now move to examining the time spent while iterating through the loops. Let B_j be the root bag of the decomposition created for graph G_j. We charge B_j with all the operations that were done when processing G_j within the loops. Note that thus every such B_j is charged at most once, and with running time $O(c^k \cdot \log n)$. Summarizing, every bag of $\mathcal{T}'$ is charged with $O(c^k \cdot \log n)$ running time, and we have at most $42n/\log n$ bags, so the total running time of `FindPartialTD` is $O(c^k \cdot n)$.

Algorithm 8: `FindPartialTD`

```
Data: Data structure $\mathcal{DS}$
Output: Partial tree decomposition of width at most $k$ of $G[S \cup U]$ with $S$
        as root bag or conclusion that $\mathrm{tw}(G) > k$.
  $\mathrm{old}_S \leftarrow \mathcal{DS}.\mathrm{get}_S()$
  $\mathrm{old}_\pi \leftarrow \mathcal{DS}.\mathrm{get}_\pi()$
  $\mathtt{old_w} \leftarrow \mathcal{DS}.\mathtt{whatsep}$
  if $\mathcal{DS}.\mathtt{whatsep} = s$ then
      $\mathtt{sep} \leftarrow \mathcal{DS}.\mathrm{findSSeparator}()$
      $\mathcal{DS}.\mathtt{whatsep} \leftarrow u$
  else
      $\mathtt{sep} \leftarrow \mathcal{DS}.\mathrm{findUSeparator}()$
      $\mathcal{DS}.\mathtt{whatsep} \leftarrow s$
  end
  if $\mathtt{sep} = \perp$ then
      $\mathcal{DS}.\mathtt{whatsep} \leftarrow \mathtt{old_w}$
      return $\perp$        /* safe to return: the state not changed */
  end
  $\mathcal{DS}.\mathrm{insert}_X(\mathtt{sep})$
  $\mathtt{pins} \leftarrow \emptyset$
  while $(u, l) \leftarrow \mathcal{DS}.\mathrm{findNextPin}() \neq \perp$ and $l \geq \log n$ do
      $\mathtt{pins}.\mathit{append}(u)$
      $\mathcal{DS}.\mathrm{insert}_F(u)$
  end
  $\mathcal{DS}.\mathrm{clear}_X()$
  $\mathcal{DS}.\mathrm{clear}_F()$
  $\mathcal{DS}.\mathrm{insert}_S(\mathtt{sep})$
  $\mathtt{bags} \leftarrow \emptyset$
  for $u \in \mathtt{pins}$ do
      $\mathcal{DS}.\mathrm{set}_\pi(u)$
      $\mathtt{bags}.\mathit{append}(\mathcal{DS}.\mathrm{findNeighborhood}())$
  end
  $\mathrm{children} \leftarrow \emptyset$
  for $u, b \in \mathtt{pins}, \mathtt{bags}$ do
      $\mathcal{DS}.\mathrm{set}_\pi(u)$
      $\mathcal{DS}.\mathrm{clear}_S()$
      $\mathcal{DS}.\mathrm{insert}_S(b)$
      $\mathrm{children}.\mathit{append}(\mathtt{FindPartialTD}())$
  end
  $\mathcal{DS}.\mathtt{whatsep} \leftarrow \mathtt{old_w}$
  $\mathcal{DS}.\mathrm{clear}_S()$
  $\mathcal{DS}.\mathrm{insert}_S(\mathrm{old}_S)$
  $\mathcal{DS}.\mathrm{set}_\pi(\mathrm{old}_\pi)$
  if $\perp \in \mathrm{children}$ then
      return $\perp$        /* postponed because of rollback of $S$ and $\pi$ */
  end
  return $\mathtt{build}(\mathrm{old}_S, \mathtt{sep}, \mathrm{children})$
```

Chapter 28

A linear time algorithm

In this section we give the main result of this part, i.e., we prove Theorem 47 (stated below) and discuss how it follows from a combination of the previous sections and a number of existing results with some modifications.

Theorem 47. *There exists an algorithm that, given an n-vertex graph G and an integer k, in time $2^{O(k)}n$ either outputs that the treewidth of G is larger than k, or constructs a tree decomposition of G of width at most $5k + 4$.*

The algorithm distinguishes between two cases depending on whether or not $n \leq 2^{2^{c_0 k}}$ for some constant c_0. If this is the case, we may simply run $\texttt{Alg}_2$ which will work in linear time since $n \log \log n = O(nk^{O(1)})$ and hence $O(c_2^k n \log \log n) = O(c^k n)$ for some constant c. Otherwise we will construct a tree automaton being a version of the dynamic programming algorithm by Bodlaender and Kloks [BK96]. The crucial insight is to perform the automaton construction once, before the whole algorithm, and then use the table lookups to implement automaton's run in time $O(k^{O(1)}n)$. These techniques combined will give us a 5-approximation algorithm for treewidth in time single exponential in k and linear in n.

The section is organized as follows. In Section 28.1, we describe how to construct the automaton. In Section 28.1.1 we show how to solve the instance when n astronomically big compared to k, and finally in Section 28.1.2 we wrap up and give the proof of Theorem 47.

28.1 Automata

This section is devoted to the proof of Lemma 28.2. We (1) define nice expression trees, (2) explain the relationship between dynamic programming on tree decompositions and nice expression trees, (3) describe a table lookup procedure and finally (4) show how to construct an actual tree decomposition, provided that the automaton decides whether the treewidth of the input graph is at most k.

Lemma 28.1 (Bodlaender and Kloks [BK96])**.** *There is an algorithm that, given a graph G, an integer k, and a nice tree decomposition of G of width at most ℓ*

with $O(n)$ bags, either decides that the treewidth of G is more than k, or finds a tree decomposition of G of width at most k in time $O(2^{O(k\ell^2)}n)$.

In our algorithm, we need to separate the automaton's construction from running it on the tree decomposition. We will use the standard notion of a deterministic tree automaton working on ranked trees with at most two children per node, labeled with symbols from some finite alphabet Σ. The automaton has a finite set of states, whereas transitions compute the state corresponding to a node based on the states of the children and the symbol of Σ placed on the node.

Our version of the above result can be hence expressed as follows.

Lemma 28.2. *There are algorithms* $\texttt{Alg}_{\text{pre}}$ *and* $\texttt{Alg}_{\text{run}}$ *and constants c_p and c_r, such that*

- *Algorithm* $\texttt{Alg}_{\text{pre}}$ *gets as input integers $k, \ell \in \mathbb{N}$, where k is the treewidth we want to decide whereas ℓ is the width of the input tree decomposition, and constructs in time $O(2^{2^{c_p k\ell^2}})$ an automaton $\mathcal{A}_{k,\ell}$.*
- *Algorithm* $\texttt{Alg}_{\text{run}}$ *gets as input a graph G, with a nice tree decomposition of G of width at most ℓ with $O(n)$ bags, an integer k, and the automaton $\mathcal{A}_{k,\ell}$, and either decides that the treewidth of G is more than k, or finds a tree decomposition of G of width at most k in time $O(k^{c_r}n)$.*

By the exact same recursive techniques as in Sections 26.1 and 27.1, which originates from Bodlaender's algorithm [Bod96], we can focus our attention on the case when we are given a tree decomposition of the input graph of slightly too large width.

Nice expression trees

The dynamic programming algorithm in Bodlaender and Kloks [BK96] is described with help of so called *nice tree decompositions*[1]. As we need to represent a nice tree decomposition as a labeled tree with the label alphabet of size being a function of k, we use a slightly different notion of *labeled nice tree decomposition.* The formalism is quite similar to existing formalisms, e.g., the operations on k-terminal graphs by Borie [Bor88], or construction terms used by Lokshtanov et al. [LPPS14].

A labeled terminal graph is a 4-tuple $G = (V, E, X, f)$, with (V, E) a graph, $X \subseteq V(G)$ a set of *terminals*, and $f \colon X \to \mathbb{N}$ an injective mapping of the terminals to non-negative integers, which we call *labels*. A k-labeled terminal graph is a labeled terminal graph with the maximum label at most k, i.e., $\max_{x\in X} f(x) \leq k$. Let O_k be the set of the following operations on k-terminal graphs.

Leaf$_\ell$(): gives a k-terminal graph with one vertex v, no edges, with v a terminal with label ℓ.

[1]See the beginning of Chapter 29 for a definition of *nice* tree decompositions.

Introduce$_{\ell,S}$(G): $G = (V(G), E(G), X, f)$ is a k-terminal graph, ℓ a non-negative integer, and $S \subseteq \{1, \ldots, k\}$ a set of labels. If there is a terminal vertex in G with label ℓ, then the operation returns G, otherwise it returns the graph obtained by adding a new vertex v, making v a terminal with label ℓ, and adding edges $\{v, w\}$ for each terminal $w \in X$ with $f(w) \in S$. I.e., we make the new vertex adjacent to each existing terminal whose label is in S.

Forget$_\ell$(G): Again $G = (V(G), E(G), X, f)$ is a k-terminal graph. If there is no vertex $v \in X$ with $f(v) = \ell$, then the operation returns G, otherwise, we turn v into a non-terminal, i.e., we return the k-terminal graph $(V(G), E(G), X \setminus \{v\}, f')$ for the vertex v with $f(v) = \ell$, and f' is the restriction of f to $X \setminus \{v\}$.

Join(G,H): $G = (V(G), E(G), X, f)$ and $H = (V(H), E(H), Y, g)$ are k-terminal graphs. If the range of f and g are not equal, then the operation returns G. Otherwise, the result is obtained by taking the disjoint union of the two graphs, and then identifying pairs of terminals with the same label.

Note that for given k, O_k is a collection of $k + k \cdot 2^k + k + 1$ operations. When the treewidth is k, we work with $k+1$-terminal graphs. The set of operations mimics closely the well known notion of nice tree decompositions (see e.g., Kloks [Klo94] or Bodlaender [Bod98]).

Proposition 28.3. *Suppose a tree decomposition of G is given of width at most k with m bags. Then, in time linear in n and polynomial in k, we can construct an expression giving a graph isomorphic to G in terms of operations from O_{k+1} with the length of the expression $O(mk)$.*

Proof. First build with standard methods a nice tree decomposition of G of width k; this has $O(mk)$ bags, and $O(m)$ join nodes. Now, construct the graph $H = (V(H), E(H))$, with $V(H) = V(G)$ and for all $v, w \in V(H)$, $\{v, w\} \in E(H)$, if and only if there is a bag i with $v, w \in X_i$. It is well known that H is a chordal super graph of G with maximum clique size $k+1$ (see e.g., Bodlaender [Bod98]). Use a greedy linear time algorithm to find an optimal vertex coloring c of H (see Golumbic [Gol80, Section 4.7].)

Now, we can transform the nice tree decomposition to the expression as follows: each leaf bag that contains a vertex v is replaced by the operation Leaf$_{c(v)}$, i.e., we label the vertex by its color in H. We can now replace bottom up each bag in the nice tree decomposition by the corresponding operation; as we labeled vertices with the color in H, we have that all vertices in a bag have different colors, which ensures that a Join indeed performs identifications of vertices correctly. Bag sizes are bounded by $k+1$, so all operations belong to O_{k+1}. □

View the expression as a rooted tree with every node labeled by an operation from O_{k+1}: leaves are labeled with the Leaf operation, and binary nodes have the Join-label. Thus, every node has at most two children. To each node i of the tree, we can associate a graph G_i; the graph G_r associated to the root node r is isomorphic to G. Call such a labeled rooted tree a *nice expression tree* of width k.

Dynamic programming and finite state tree automata

The discussion in this paragraph holds for all problems invariant under isomorphism. Note that the treewidth of a graph is also invariant under isomorphisms. We use ideas from the early days of treewidth, see e.g., Fellows and Langston [FL] or Abrahamson and Fellows [AF93].

A dynamic programming algorithm on nice tree decompositions can be viewed also as a dynamic programming algorithm on a nice expression tree of width k. Suppose that we have a dynamic programming algorithm that computes in bottom-up order for each node of the expression tree a table with at most $r = O(1)$ bits per table, and to compute a table, only the label of the node (type of operation) and the tables of the children of the node are used. As we will argue in the next sections, the DP algorithm for treewidth from Bodlaender and Kloks [BK96] is indeed of this form, if we see k as a fixed constant. Such an algorithm can be seen as a finite state tree automaton: the states of the automaton correspond to the at most $2^r = O(1)$ different tables; the alphabet are the $O(1)$ different labels of tree nodes.

To decide if the treewidth of G is at most k, we first explicitly build this finite state tree automaton, and then execute it on the expression tree. For actually building the corresponding tree decomposition of G of width at most k, if existing, some more work has to be done, which is described later.

Table lookup implementation of dynamic programming

The algorithm of Bodlaender and Kloks [BK96] (see especially Definition 5.9 therein) builds for each node in the nice tree decomposition a *table of characteristics*. Each characteristic represents the "essential information" of a partial tree decomposition of width at most k of the graph associated with the bag. More precisely, if i is a node of the input tree decomposition, then we need to succintly encode partial tree decompositions of graph G_i. With each such partial decomposition we associate its *characteristic*. Bodlaender and Kloks argue that for the rest of the computation, we only need to remember a bounded-size family of characteristics. More precisely, we will say that a set of characteristics $\mathcal{F}$ for the node i is *full* if the following holds: if there is a tree decomposition of the whole graph, then there is also a tree decomposition whose restriction to G_i has a characteristic belonging to $\mathcal{F}$. The crucial technical result of Bodlaender and Kloks [BK96] is that there is a full set of characteristics $\mathcal{F}$ where each characteristic has size bounded polynomially in k and ℓ, assuming that we are given an expression tree of width ℓ and want to test if the treewidth is at most k. Moreover, this set can be effectively constructed for each node of the input tree decomposition using a bottom-up dynamic programming. Inspection of the presentation of [BK96] easily shows that the number of possible characteristics, for which we need to store whether they are included in the full set or not, is bounded by $2^{O(k \cdot \ell^2)}$. Thus, the algorithm of Bodlander and Kloks needs to remember $2^{O(k \cdot \ell^2)}$ bits of information in every dynamic programming table, one per every possible characteristic.

We now use that we represent the vertices in a bag, i.e., the terminals, by labels from $\{1, \ldots, \ell + 1\}$, where ℓ is the width of the nice expression tree. Thus, we have a set $C_{k,\ell}$ (only depending of k and ℓ) with $|C_{k,\ell}| = 2^{O(k \cdot \ell^2)}$ that contains all possible characteristics that can possibly belong to a computed full set. Then, each table is just a subset of $C_{k,\ell}$, i.e., an element of $\mathcal{P}(C_{k,\ell})$, where $\mathcal{P}(\cdot)$ denotes the powerset. This in turn means that we can view the decision variant of the dynamic programming algorithm of Bodlaender and Kloks as a tree automaton working over rooted trees over the alphabet $O_{\ell+1}$ (with at most two children per node). Namely, the state set of the automaton is $\mathcal{P}(C_{k,\ell})$, and transitions correspond to the formulas for combining full sets of characteristics that are given by Bodlaender and Kloks [BK96].

The first step of the proof of Lemma 28.2 is to explicitly construct the described tree automaton. We can do this as follows. Enumerate all characteristics in $C_{k,\ell}$, and number them $c_1, \ldots, c_s$, where $s = 2^{O(k \cdot \ell^2)}$. Enumerate all elements of $\mathcal{P}(C_{k,\ell})$, and number them $t_1, \ldots, t_{s'}$, $s' = 2^{2^{O(k \cdot \ell^2)}}$; store with t_i the elements of its set.

Then, we compute the transition function $\delta\colon O_{\ell+1} \times \{1, \ldots, s'\} \times \{1, \ldots, s'\} \to \{1, \ldots, s'\}$. In terms of tree automaton view, δ computes the state of a node given its symbol and the states of its children. (If a node has less than two children, the third, and possibly the second argument are ignored.) In terms of the DP algorithm, if we have a tree node i with operation $o \in O_{\ell+1}$, and the children of i have tables corresponding to t_α and t_β, then $\delta(o, \alpha, \beta)$ gives the number of the table obtained for i by the algorithm. To compute one value of δ, we just execute one part of the algorithm of Bodlaender and Kloks. Suppose we want to compute $\delta(o, \alpha, \beta)$. We build the tables T_α and T_β corresponding to t_α and t_β, and execute the step of the algorithms of Bodlaender and Kloks for a node with operation o whose children have tables T_α and T_β. (If the node is not binary, we ignore the second and possibly both tables.) Then, look up what is the index of the resulting table; this is the value of $\delta(o, \alpha, \beta)$.

We now estimate the time to compute δ. We need to compute $O(2^\ell \cdot \ell \cdot s'^2) = O(2^{2^{O(k \cdot \ell^2)}})$ values; each executes one step of the DP algorithm and does a lookup in the table, which is easily seen to be bounded again by $O(2^{2^{O(k \cdot \ell^2)}})$, so the total time to compute δ is still bounded by $O(2^{2^{O(k \cdot \ell^2)}})$. To decide if the treewidth of G is at most k, given a nice tree decomposition of width at most ℓ, we thus carry out the following steps:

- Compute δ.
- Transform the nice tree decomposition to a nice expression tree of width ℓ.
- Compute bottom-up (e.g., in post-order) for each node i in the expression tree a value q_i, equal to the state of the automaton at i in the run, as follows. If node i is labeled by operation $o \in O_{\ell+1}$ and its children have pre-computed values q_{j_1}, q_{j_2}, we have $q_i = \delta(o, q_{j_1}, q_{j_2})$. If i has less than two children, we take some arbitrary argument for the values of missing children.

In this way, q_i corresponds to the table computed by the DP algorithm of Bodlaender and Kloks [BK96].

- If the value q_r for the root of the expression tree corresponds to the empty set, then the treewidth of G is more than k, otherwise the treewidth of G is at most k [BK96].

If our decision algorithm decides that the treewidth of G is more than k, we reject, and we are done. Otherwise, we need to do additional work to construct a tree decomposition of G of width at most k, which is described next.

Constructing tree decompositions

After the decision algorithm has determined that the treewidth of G is at most k, we need to find a tree decomposition of G of width at most k. Again, the discussion is necessarily not self contained and we refer to details given in Bodlaender and Kloks [BK96, Section 6].

Basically, each table entry (characteristic) in the table of a join node is the result of a combination of a characteristic from the table of the left child and a characteristic from the table of the right child. More precisely, a pair of characteristics for the children can be *combined* into a characteristic of a node. Bodlaender and Kloks give a constructive procedure that verifies whether the combination of two characteristic is possible to perform, and if so then it computes the result; this algorithm was already used implicitly in the previous section when we described how δ is computed. The characteristics that are included in the table of a node are exactly those that can be obtained as combinations of characteristics included in the tables of children. Similarly, for nodes with one child, each characteristic is the result of an operation to a characteristic in the table of the child node, and characteristics stored for a node are exactly those obtainable in this manner. Leaf nodes represent a graph with one vertex, and we have just one tree decomposition of this graph, and thus one table entry in the table of a leaf node.

We now describe how, given a run of the automaton (equivalently, filling of the DP tables), to reconstruct one exemplary tree decomposition of the graph. This construction can be thought of a variant of the well-known technique of *back-links* used for recovering solutions from dynamic programming tables. First, let us define an auxiliary function γ, as follows. This function has four arguments: an operation from $O_{\ell+1}$, the index of a characteristic (a number between 1 and s), and the indices of two states (numbers between 1 and $s' = 2^s$). As value, γ yields $\perp$ or a pair of two indices of characteristics. The intuition is as follows: suppose we have a node i in the nice expression tree labeled with o, an index c_i of a characteristic of a (not yet known) tree decomposition of G_i, and indices of the tables of the children of i, say t_{j_1} and t_{j_2}. Then $\gamma(o, c_i, t_{j_1}, t_{j_2})$ should be an (arbitrarily chosen) pair (c_{j_1}, c_{j_2}) such that c_i is the result of the combination of c_{j_1} and c_{j_2} (in case o is the join operation) or of the operation as described in the previous paragraph to c_{j_1} (in case o is an operation with one argument; c_{j_2} can have any value and is ignored). If no such pair exists, the output of γ is $\perp$.

To compute γ, we can perform the following steps for each 4-tuple o, c_i, t_{j_1}, t_{j_2}. Let $S_1 \in \mathcal{P}(C_{k,\ell})$ be the set corresponding to t_{j_1}, and $S_2 \in \mathcal{P}(C_{k,\ell})$ be the set corresponding to t_{j_2}. For each $c \in S_1$ and $c' \in S_2$, see if a characteristic c and a characteristic c' can be combined (or, in case of a unary operation, if the relevant operation can be applied to c) to obtain c_1. If we found at least one such pair, we return an arbitrarily selected one (say, lexicographically first); if no combination gives c_1, we return $\perp$. Again, in case of unary operations o, we ignore c'. We do not need γ in case o is a leaf operation, and can give any return values in such cases. One can easily see that the computation of γ uses again $2^{2^{O(k\cdot\ell^2)}}$ time.

The first step of our construction phase is to build γ, as described above. After this, we select a characteristic from $C_{k,\ell}$ for each node in the nice expression tree, as follows. As we arrived in this phase, the state of the root bag corresponds to a nonempty set of characteristics, and we take an arbitrary characteristic from this set (e.g., the first one from the list). Now, we select top-down in the expression tree (e.g., in pre-order) a characteristic for each node. Leaf nodes always receive the characteristic of the trivial tree decomposition of a graph with one vertex. In all other cases, if node i has operation o and has selected characteristic c, the left child of i has state t_{j_1} and the right child of i has state t_{j_2} (or, take any number, e.g., 1, if i has only one child, i.e., o is a unary operation), look up the precomputed value of $\gamma(o, c, t_{j_1}, t_{j_2})$. As c is a characteristic in the table that is the result of $\delta(o, t_{j_1}, t_{j_2})$, we have that $\gamma(o, c, t_{j_1}, t_{j_2}) \neq \perp$, so suppose $\gamma(o, c, t_{j_1}, t_{j_2})$ is the pair (c', c''). We associate c' as characteristic with the left child of i, and (if i has two children) c'' as characteristic with the right child of i.

At this point, we have associated a characteristic with each node in the nice expression tree. These characteristics are precisely the same as the characteristics that are computed in the constructive phase of the algorithm from Bodlaender and Kloks [BK96, Section 6], with the sole difference that we work with labeled terminals instead of the "names" of the vertices (i.e., in Bodlaender and Kloks [BK96], terminals / bag elements are identified as elements from $V(G)$).

From this point on, we can follow without significant changes the algorithm from Bodlaender and Kloks [BK96, Section 6]: bottom-up in the expression tree, we build for each node i, a tree decomposition of G_i whose characteristic is the characteristic we just selected for i, together with a number of pointers from the characteristic to the tree decomposition. The decomposition for G_i can be constructed from the (pre-computed) decompositions for the children of i using amortized time $(k+\ell)^{O(1)}$ for additional work needed for combining the partial decompositions, so that the whole reconstruction algorithm works in time $O((k+\ell)^{O(1)} \cdot n)$. Again, the technical details can be found in Bodlaender and Kloks [BK96], our only change is that we work with terminals labeled with integers in $\{1, \ldots, \ell+1\}$ instead of bag vertices.

At the end of this process, we obtain a tree decomposition of the graph associated with the root bag $G_r = G$ whose characteristic belongs to the set corresponding to the state of r. As we only work with characteristics of tree decompositions of width at most k, we obtained a tree decomposition of G of width at most k.

All work we do, except for the pre-computation of the tables of δ and γ, is linear in n and polynomial in k; the time for the pre-computation does not depend on n, and is bounded by $2^{2^{O(k\ell^2)}}$. Note that once δ and γ are computed, retrieving their values is done by a table lookup that takes constant time in the RAM model. This ends the description of the proof of Lemma 28.2.

28.1.1 Astronomic n: Proof of Lemma 28.4

We now state and prove Lemma 28.4, which is a variant of a result from Bodlaender [Bod96].

Lemma 28.4. *There exists constants c_p and c_r and an algorithm, that given an n-vertex graph G and an integer k, in time $O(2^{2^{c_p k^3}} + k^{c_r} n)$, either outputs that the treewidth of G is larger than k, or constructs a tree decomposition of G of width at most k.*

The $O(f(k)n)$ algorithm for treewidth by Bodlaender [Bod96] makes a number of calls to an algorithm by Bodlaender and Kloks [BK96]. More precisely, Lemma 28.4 is obtained by combining Lemmata 28.1 and 28.2, but modifying it by replacing the explicit construction of tables to lookup of states in an explicitly constructed automaton.

We modify the algorithm as follows. Before the whole procedure, in the preprocessing phase we once construct the automaton $\mathcal{A}_{k,\ell}$ using algorithm $\mathtt{Alg}_{\mathrm{pre}}$ from Lemma 28.2 with $\ell = 2k + 1$. Then, instead of calling the algorithm of Lemma 28.1 in Step 5, we call algorithm $\mathtt{Alg}_{\mathrm{run}}$ from Lemma 28.2. Thus we obtain an algorithm for treewidth for fixed k that uses $O(2^{2^{c_p k^3}})$ time once for constructing the automaton, and then has a recursive procedure whose running time is given by

$$T(n) = T\left((1 - \Omega(\frac{1}{k^6})n\right) + O(k^{O(1)}n),$$

which solves to $T(n) = O(k^{O(1)}n)$. We can conclude that using the algorithm of Lemma 28.2 as subroutine inside Bodlaender's algorithm gives an algorithm for treewidth that uses $O(2^{2^{O(k^3)}} + k^{O(1)}n)$ time, and thus Lemma 28.2 together with the insights of earlier sections in this paper and the results from Bodlaender [Bod96] implies Lemma 28.4.

28.1.2 Wrapping up

Proof of Theorem 47. Now, Theorem 47 follows easily from the results in previous sections and Lemma 28.4, in the following way: The algorithm distinguishes between two cases. The first case is when n is “sufficiently small” compared to k. By this, we mean that $n \leq 2^{2^{c_p k^3}}$ for the value of $c_p \in \mathbb{N}$ in Lemma 28.4. The other case is when this is *not* the case. For the first case, we can apply $\texttt{Alg}_2$ from Theorem 46. Since n is sufficiently small compared to k we can observe that $\log \log n = k^{O(1)}$, resulting in a $2^{O(k)}n$ time algorithm. In the second case, we use the algorithm of Lemma 28.4; as $n > 2^{2^{c_p k^3}}$, the algorithm uses $O(k^{c_r} n)$ time. $\square$

Chapter 29

The data structure

In this chapter we give a complete explanation of the data structure that has been instrumental for developing the algorithms given in the previous chapters.

29.1 Overview of the data structure

Assume we are given a tree decomposition $(\{B_i \mid i \in I\}, T = (I, F))$ of G of width $O(k)$. First we turn our tree decomposition into a tree decomposition of depth $O(\log n)$, keeping the width to $t = O(k)$, by the work of Bodlaender and Hagerup [BH98]. Furthermore, by standard arguments we turn this decomposition into a *nice* tree decomposition in $O(t^{O(1)} \cdot n)$ time, that is, a decomposition of the same width and satisfying following properties:

- All the leaf bags, as well as the root bag, are empty.
- Every node of the tree decomposition is of one of four different types:
  - **Leaf node**: a node i with $B_i = \emptyset$ and no children.
  - **Introduce node**: a node i with exactly one child j such that $B_i = B_j \cup \{v\}$ for some vertex $v \notin B_j$; we say that v is *introduced* in i.
  - **Forget node**: a node i with exactly one child j such that $B_i = B_j \setminus \{v\}$ for some vertex $v \in B_i$; we say that v is *forgotten* in i.
  - **Join node**: a node i with two children j_1, j_2 such that $B_i = B_{j_1} = B_{j_2}$.

The standard technique of turning a tree decomposition into a nice one includes (i) adding paths to the leaves of the decomposition on which we consecutively introduce the vertices of corresponding bags; (ii) adding a path to the root on which we consecutively forget the vertices up to the new root, which is empty; (iii) introducing paths between every non-root node and its parent, on which we first forget all the vertices that need to be forgotten, and then introduce all the vertices that need to be introduced; (iv) substituting every node with $d > 2$ children with a balanced binary tree of $O(\log d)$ depth. It is easy to check that after performing these operations, the tree decomposition has depth at most $O(t \log n)$ and contains at most $O(t \cdot n)$ bags. Moreover, using folklore preprocessing routines, in $O(t^{O(1)} \cdot n)$ time we may prepare the decomposition for algorithmic uses, e.g., for each bag compute and store the list of edges contained in this bag. We omit here the details of this transformation and refer to Kloks [Klo94].

In the data structure, we store a number of tables: three special tables that encode general information on the current state of the graph, and one table per query. The information stored in the tables reflect some choice of subsets of $V(G)$, which we will call the *current state of the graph.* More precisely, at each moment the following subsets will be distinguished: S, X, F and a single vertex π, called the pin. The meaning of these sets is described in Chapter 26. On the data structure we can perform the following updates: adding and removing vertices to S, X, F and marking and unmarking a vertex as a pin. In the following table we gather the tables used by the algorithm, together with an overview of the running times of updates. The meaning of the table entries uses terminology that is described in the following sections.

The following lemma follows from each of the entries in the table below, and will be proved in this section:

Lemma 29.1. *The data structure can be initialized in $O(c^k n)$ time.*

Table	Meaning
$P[i]$	Boolean value $\pi \in W_i$
$C[i][(S_i, U_i)]$	Connectivity information on U_i^{ext}
$CardU[i][(S_i, U_i)]$	Integer value $\lvert U_i^{\text{ext}} \cap W_i \rvert$
$T_1[i][(S_i, U_i)]$	Table for query findNeighborhood
$T_2[i][(S_i, U_i)][\psi]$	Table for query findSSeparator
$T_3[i][(S_i, U_i, X_i, F_i)]$	Table for query findNextPin
$T_4[i][(S_i, U_i)][\psi]$	Table for query findUSeparator

Figure 29.1: Description of tables.

Table	Update	Initialization
$P[i]$	$O(t \cdot \log n)$	$O(t \cdot n)$
$C[i][(S_i, U_i)]$	$O(3^t \cdot t^{O(1)} \cdot \log n)$	$O(3^t \cdot t^{O(1)} \cdot n)$
$CardU[i][(S_i, U_i)]$	$O(3^t \cdot t^{O(1)} \cdot \log n)$	$O(3^t \cdot t^{O(1)} \cdot n)$
$T_1[i][(S_i, U_i)]$	$O(3^t \cdot k^{O(1)} \cdot \log n)$	$O(3^t \cdot k^{O(1)} \cdot n)$
$T_2[i][(S_i, U_i)][\psi]$	$O(9^t \cdot k^{O(1)} \cdot \log n)$	$O(9^t \cdot k^{O(1)} \cdot n)$
$T_3[i][(S_i, U_i, X_i, F_i)]$	$O(6^t \cdot t^{O(1)} \cdot \log n)$	$O(6^t \cdot t^{O(1)} \cdot n)$
$T_4[i][(S_i, U_i)][\psi]$	$O(5^t \cdot k^{O(1)} \cdot \log n)$	$O(5^t \cdot k^{O(1)} \cdot n)$

Figure 29.2: Complexities of the tables.

We now proceed to the description of the table P, and then to the two tables C and $CardU$ that handle the important component U. The tables T_1, T_2, T_3 are described together with the description of realization of the corresponding queries. Whenever describing the table, we argue how the table is updated during updates of the data structure, and initialized in the beginning.

29.2 The pin table

In the table P, for every node i of the tree decomposition we store a boolean value $P[i]$ equal to $(\pi \in W_i)$. We now show how to maintain the table P when the data structure is updated. The table P needs to be updated whenever the pin π is marked or unmarked. Observe, that the only nodes i for which the information whether $\pi \in W_i$ changed, are the ones on the path from r_π to the root of the tree decomposition. Hence, we can simply follow this path and update the values. As the tree decomposition has depth $O(t \log n)$, this update can be performed in $O(t \cdot \log n)$ time. As when the data structure is initialized, no pin is assigned, P is initially filled with $\bot$.

29.3 Maintaining the important component

Before we proceed to the description of the queries, let us describe what is the reason of introducing the pin π. During the computation, the algorithm recursively considers smaller parts of the graph, separated from the rest via a small separator: at each step we have distinguished set S and we consider only one connected component U of $G \setminus S$. Unfortunately, we cannot afford recomputing the tree decomposition of U at each recurrence call, or even listing the vertices of U. Therefore we employ a different strategy for identification of U. We will distinguish one vertex of U as a representative pin π, and U can then be defined as the set of vertices reachable from π in $G \setminus S$. Instead of recomputing U at each recursive call we will simply change the pin.

In order to make the operation to change the pin more efficient, we store additional information in the tables. For each node i of the tree decomposition, we not only have an entry in its table for the current value of the pin π, but in order to quickly update information when the pin is changed, instead store entries for each possible intersection of U with B_i. Thus, when the pin is changed and thus the important set is changed, we are prepared and the information is already available in the table: information needs to be recomputed on two paths to the root in the tree decomposition, corresponding the previous and the next pin, while for subtrees unaffected by the change we do not need to recompute anything as the tables stored there already contain information about the new U as well—as they contain information for *every possible* new U. As the tree decomposition is of logarithmic depth, the update time is logarithmic instead of linear.

We proceed to the formal description. We store the information about U in two special tables: C and $CardU$. As we intuitively explained, tables C and $CardU$ store information on the connectivity behavior in the subtree, for every possible interaction of U with the bag. Formally, for every node of the tree decomposition i

we store an entry for every member of the family of *signatures* of the bag B_i. A signature of the bag B_i is a pair (S_i, U_i), such that S_i, U_i are disjoint subsets of B_i. Clearly, the number of signatures is at most $3^{|B_i|}$.

Let i be a node of the tree decomposition. For a signature $\phi = (S_i, U_i)$ of B_i, let $S_i^{\mathrm{ext}} = S_i \cup (S \cap W_i)$ and U_i^{ext} consists of all the vertices reachable in $G_i \setminus S_i^{\mathrm{ext}}$ from U_i or π, providing that it belongs to W_i. Sets S_i^{ext} and U_i^{ext} are called *extensions* of the signature ϕ; note that given S_i and U_i, the extensions are defined uniquely. We remark here that the definition of extensions depend not only on ϕ but also on the node i; hence, we will talk about extensions of signatures only when the associated node is clear from the context.

We say that signature ϕ of B_i with extensions S_i^{ext} and U_i^{ext} is *valid* if it holds that

(i) $U_i^{\mathrm{ext}} \cap B_i = U_i$,

(ii) if $U_i \neq \emptyset$ and $\pi \in W_i$ (equivalently, $P[i]$ is true), then the component of $G[U_i^{\mathrm{ext}}]$ that contains π contains also at least one vertex of U_i.

Intuitively, invalidity means that ϕ cannot contain consistent information about intersection of U and G_i. The second condition says that we cannot fully forget the component of π, unless the whole U_i^{ext} is already forgotten.

Formally, the following invariant explains what is stored in tables C and $CardU$:

- if ϕ is invalid then $C[i][\phi] = CardU[i][\phi] = \bot$;
- otherwise, $C[i][\phi]$ contains an equivalence relation R consisting of all pairs of vertices $(a, b) \in U_i$ that are connected in $G_i[U_i^{\mathrm{ext}}]$, while $CardU[i][\phi]$ contains $|U_i^{\mathrm{ext}} \cap W_i|$.

Note that in this definition we actually ignore the information about the membership of vertices of B_i in sets S, F, X in the current state of the graph: the stored information depends only on the membership of forgotten vertices to these sets, whereas the signature of the bag overrides the actual information about the membership of vertices of the bag. In this manner we are prepared for possible changes in the data structure, as after an update some other signature will reflect the current state of the graph. Moreover, it is clear from this definition that during the computation, the membership of any vertex v in sets S, F, X in the current state of the graph is being checked only in the single node r_v when this vertex is being forgotten; we use this property heavily to implement the updates efficiently.

We now explain how for every node i, entries of $C[i]$ and $CardU[i]$ can be computed using the entries of these tables for the children of i. We consider different cases, depending on the type of node i.

Case 1: Leaf node. If i is a leaf node then $C[i][(\emptyset, \emptyset)] = \emptyset$ and $CardU[i][(\emptyset, \emptyset)] = 0$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i; we would like to compute $R_i = C[i][\phi]$. Let ϕ' be a natural projection of ϕ onto B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$. Let $R_j = C[j][\phi']$. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in S_i$. If we introduce a vertex from S_i, then it follows that extensions of $U_i = U_j$ are equal. Therefore, we can put $C[i][\phi] = C[j][\phi']$ and $CardU[i][\phi] = CardU[j][\phi']$.

Case 2.2: $v \in U_i$. In the beginning we check whether conditions of validity are not violated. First, if v is the only vertex of U_i and $P[i] = \top$, then we simply put $C[i][\phi] = \bot$: condition (ii) of validity is violated. Second, we check whether v is adjacent only to vertices of S_j and U_j; if this is not the case, we put $C[i][\phi] = \bot$ as condition (i) of validity is violated.

If the validity checks are satisfied, we can infer that the extension U_i^{ext} of U_i is extension U_j^{ext} of U_j with v added; this follows from the fact that B_j separates v from W_j, so the only vertices of U_i^{ext} adjacent to v are already belonging to U_j. Now we would like to compute the equivalence relation R_i out of R_j. Observe that R_i should be basically R_j augmented by connections introduced by the new vertex v between its neighbors in B_j. Formally, R_i may be obtained from R_j by merging equivalence classes of all the neighbors of v from U_j, and adding v to the obtained equivalence class; if v does not have any neighbors in U_j, we put it as a new singleton equivalence class. Clearly, $CardU[i][\phi] = CardU[j][\phi']$.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$. We first check whether the validity constraints are not violated. As v is separated from W_j by B_j, the only possible violation introduced by v is that v is adjacent to a vertex from U_j. In this situation we put $C[i][\phi] = CardU[i][\phi] = \bot$, and otherwise we can put $C[i][\phi] = C[j][\phi']$ and $CardU[i][\phi] = CardU[j][\phi']$, because extensions of ϕ and ϕ' are equal.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and define extensions S_i^{ext}, U_i^{ext} for this signature. Observe that there is at most one valid signature $\phi' = (S_j, U_j)$ of B_j for which $S_j^{\text{ext}} = S_i^{\text{ext}}$ and $U_j^{\text{ext}} = U_i^{\text{ext}}$, and this signature is simply ϕ with w added possibly to S_i or U_i, depending whether it belongs to S_i^{ext} or U_i^{ext}: the three candidates are $\phi_S = (S_i \cup \{w\}, U_i)$, $\phi_U = (S_i, U_i \cup \{w\})$ and $\phi_0 = (S_i, U_i)$. Moreover, if ϕ is valid then so is ϕ'. Formally, in the following manner we can define signature ϕ', or conclude that ϕ is invalid:

- if $w \in S$, then $\phi' = \phi_S$;
- otherwise, if $w = \pi$ then $\phi' = \phi_U$;
- otherwise, we look into entries $C[j][\phi_U]$ and $C[j][\phi_0]$. If
  - (i) $C[j][\phi_U] = C[j][\phi_0] = \bot$ then ϕ is invalid, and we put $C[i][\phi] = CardU[i][\phi] = \bot$;
  - (ii) if $C[j][\phi_U] = \bot$ or $C[j][\phi_0] = \bot$, we take $\phi' = \phi_0$ or $\phi' = \phi_U$, respectively;

(iii) if $C[j][\phi_U] \neq \bot$ and $C[j][\phi_0] \neq \bot$, it follows that w must be a member of a component of $G_i \setminus S_i^{\text{ext}}$ that is fully contained in W_i and does not contain π. Hence we take $\phi' = \phi_0$.

The last point is in fact a check whether $w \in U_i^{\text{ext}}$: whether w is connected to a vertex from U_i in G_i, can be looked up in table $C[j]$ by adding or not adding w to U_i, and checking the stored connectivity information. If $w \in S_i^{\text{ext}}$ or $w \in U_i^{\text{ext}}$, we should be using the information for the signature with S_i or U_i updated with w, and otherwise we do not need to add w anywhere.

As we argued before, if ϕ is valid then so does ϕ', hence if $C[j][\phi'] = \bot$ then we can take $C[i][\phi] = CardU[i][\phi] = \bot$. On the other hand, if ϕ' is valid, then the only possibility for ϕ to be invalid is when condition (ii) cease to be satisfied. This could happen only if $\phi' = \phi_U$ and w is in a singleton equivalence class of $C[j][\phi']$ (note that then the connected component corresponding to this class needs to necessarily contain π, as otherwise we would have $\phi' = \phi_0$). Therefore, if this is the case, we put $C[i][\phi] = CardU[i][\phi] = \bot$, and otherwise we conclude that ϕ is valid and move to defining $C[i][\phi]$ and $CardU[i][\phi]$.

Let now $R_j = C[j][\phi']$. As extensions of ϕ' and ϕ are equal, it follows directly from the maintained invariant that R_i is equal to R_j with w removed from its equivalence class. Moreover, $CardU[i][\phi]$ is equal to $CardU[j][\phi']$, possibly incremented by 1 if we concluded that $\phi' = \phi_U$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. From the maintained invariant it follows that $C[i][\phi]$ is a minimum transitive closure of $C[j_1][\phi_1] \cup C[j_2][\phi_2]$, or $\bot$ if any of these entries contains $\bot$. Similarly, $CardU[i][\phi] = CardU[j_1][\phi_1] + CardU[j_2][\phi_2]$.

We now explain how to update tables C and $CardU$ in $O(3^t \cdot t^{O(1)} \cdot \log n)$ time. We perform a similar strategy as with table P: whenever some vertex v is included or removed from S, or marked or unmarked as a pin, we follow the path from r_v to the root and fully recompute the whole tables $C, CardU$ in the traversed nodes using the formulas presented above. At each step we recompute the table for some node using the tables of its children; these tables are up to date since they did not need an update at all, or were updated in the previous step. Observe that since the alignment of v in the current state of the graph is accessed only in computation for r_v, the path from r_v to the root of the decomposition consists of all the nodes for which the tables should be recomputed. Note also that when marking or unmarking the pin π, we must first update P and then C and $CardU$. The update takes $O(3^t \cdot t^{O(1)} \cdot \log n)$ time: re-computation of each table takes $O(3^t \cdot t^{O(1)})$ time, and we perform $O(t \log n)$ re-computations as the tree decomposition has depth $O(t \log n)$.

Similarly, tables C and $CardU$ can be initialized in $O(3^t \cdot t^{O(1)} \cdot n)$ time by processing the tree in a bottom-up manner: for each node of the tree decomposition, in $O(3^t \cdot t^{O(1)})$ time we compute its table based on the tables of the children, which were computed before.

29.4 Queries

In our data structure we store one table per query. In this section, we describe each of the queries. We do this by first introducing an invariant for the entries we query, then how this information can be computed once we have computed the entries for all the children.

We then discuss how to perform updates and initialization of the tables, as they are based on the same principle as with tables C and $CardU$. The queries themselves can be performed by reading a single entry of the data structure, with the exception of findUSeparator, whose implementation is more complex.

29.4.1 Query for neighbors

We begin the description of the queries with the simplest one, namely findNeighborhood. This query lists all the vertices of S that are adjacent to U. In the algorithm we have an implicit bound on the size of this neighborhood, which we can use to cut the computation when the accumulated list grows too long. We use ℓ to denote this bound; in our case we have that $\ell = O(k)$.

findNeighborhood

Output:	A list of vertices of $N(U) \cap S$, or marker '$\boxtimes$' if their number is larger than ℓ.
Time:	$O(\ell)$

Let i be a node of the tree decomposition, let $\phi = (S_i, U_i)$ be a signature of B_i, and let $U_i^{\text{ext}}, S_i^{\text{ext}}$ be extensions of this signature. In entry $T_1[i][\phi]$ we store the following:

- if ϕ is invalid then $T_1[i][\phi] = \perp$;
- otherwise $T_1[i][\phi]$ stores the list of elements of $N(U_i^{\text{ext}}) \cap S_i^{\text{ext}}$ if there is at most ℓ of them, and $\boxtimes$ if there is more of them.

Note that the information whether ϕ is invalid can be looked up in table C. The return value of the query is stored in $T[r][(\emptyset, \emptyset)]$.

We now present how to compute entries of table T_1 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_1[i][(\emptyset, \emptyset)] = \emptyset$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i; we would like to compute $T_1[i][\phi] = L_i$. Let ϕ' be a natural intersection of ϕ with B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$. Let $T_1[j][\phi'] = L_j$. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in S_i$. If we introduce a vertex from S_i, we have that U-extensions of ϕ and ϕ' are equal. It follows that L_i should be simply list L_j with v appended if it is adjacent to any vertex of $U_j = U_i$. Note here that v cannot be adjacent to any vertex of $U_i^{\text{ext}} \setminus U_i$, as B_j separates v from W_j. Hence, we copy the list L_j and append v if it is adjacent to any vertex of U_j and $L_j \neq \boxtimes$. However, if the length of the new list exceeds the ℓ bound, we replace it by $\boxtimes$. Note that copying the list takes $O(\ell)$ time, as its length is bounded by ℓ.

Case 2.2: $v \in U_i$. If we introduce a vertex from U_i, then possibly some vertices of S_i gain a neighbor in U_i^{ext}. Note here that vertices of $S_i^{\text{ext}} \setminus S_i$ are not adjacent to the introduced vertex v, as B_j separates v from W_j. Hence, we copy list L_j and append to it all the vertices of S_i that are adjacent to v, but were not yet on L_j. If we exceed the ℓ bound on the length of the list, we put $\boxtimes$ instead. Note that both copying the list and checking whether a vertex of S_i is on it can be done in $O(\ell)$ time, as its length is bounded by ℓ.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$. In this case extensions of ϕ and ϕ' are equal, so it follows from the invariant that we may simply put $T[i][\phi] = T[j][\phi']$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i. Define ϕ' in the same manner as in the Forget step in the computation of C. As extensions of ϕ and ϕ' are equal, it follows that $T_1[i][\phi] = T_1[j][\phi']$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. It follows that $T_1[i][\phi]$ should be the merge of lists $T_1[j_1][\phi_1]$ and $T_1[j_2][\phi_2]$, where we remove the duplicates. Of course, if any of these entries contains $\boxtimes$, we simply put $\boxtimes$. Otherwise, the merge can be done in $O(\ell)$ time due to the bound on lengths of $T_1[j_1][\phi_1]$ and $T_1[j_2][\phi_2]$, and if the length of the result exceeds the bound ℓ, we replace it by $\boxtimes$.

Similarly as before, for every addition and removal of vertex v to/from S, or marking and unmarking v as a pin, we can update table T_1 in $O(3^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_1 can be initialized in $O(3^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_1 must be performed after updating/initializing tables P and C.

29.4.2 Query for finding bag separators

We now move to the next query, namely finding a balanced S-separator. By Lemma 25.1, as $G[U \cup S]$ has treewidth at most k, such a $\frac{1}{2}$-balanced S-separator of size at most $k+1$ always exists. We therefore implement the following query.

findSSeparator

Output:	A list of elements of a $\frac{1}{2}$-balanced S-separator of $G[U \cup S]$ of size at most $k+1$, or $\perp$ if no such exists.
Time:	$O(t^{O(1)})$

Before we proceed to the implementation of the query, we show how to translate the problem of finding a S-balanced separator into a partitioning problem.

Lemma 29.2 (Lemma 25.8, restated)**.** *Let G be a graph and $S \subseteq V(G)$. Then a set X is a balanced S-separator if and only if there exists a partition (M_1, M_2, M_3) of $V(G) \setminus X$, such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$.*

The following combinatorial observation is crucial in the proof of Lemma 25.8.

Lemma 29.3. *Let $a_1, a_2, \ldots, a_p$ be non-negative integers such that $\sum_{i=1}^{p} a_i = q$ and $a_i \leq q/2$ for $i = 1, 2, \ldots, p$. Then there exists a partition of these integers into three sets, such that sum of integers in each set is at most $q/2$.*

Proof. Without loss of generality assume that $p > 3$, as otherwise the claim is trivial. We perform a greedy procedure as follows. At each time step of the procedure we have a number of sets, maintaining an invariant that each set is of size at most $q/2$. During the procedure we gradually merge the sets, i.e., we take two sets and replace them with their union. We begin with each integer in its own set. If we arrive at three sets, we end the procedure, thus achieving a feasible partition of the given integers. We therefore need to present how the merging step is performed.

At each step we choose the two sets with smallest sums of elements and merge them (i.e., replace them by their union). As the number of sets is at least 4, the sum of elements of the two chosen ones constitute at most half of the total sum, so after merging them we obtain a set with sum at most $q/2$. Hence, unless the number of sets is at most 3, we can always apply this merging step. □

Proof of Lemma 25.8. One of the implications is trivial: if there is a partition (M_1, M_2, M_3) of $G \setminus X$ with the given properties, then every connected component of $G \setminus X$ must be fully contained either in M_1, M_2, or M_3, hence it contains at most $|S|/2$ vertices of S. We proceed to the second implication.

Assume that X is a balanced S-separator of G and let $C_1, C_2, \ldots, C_p$ be connected components of $G \setminus X$. For $i = 1, 2, \ldots, p$, let $a_p = |S \cap C_i|$. By Lemma 29.3, there exists a partition of integers a_i into three sets, such that the sum of elements of each set is at most $|S|/2$. If we partition vertex sets of components $C_1, C_2, \ldots, C_p$ in the same manner, we obtain a partition (M_1, M_2, M_3) of $V(G) \setminus X$ with postulated properties. □

Lemma 25.8 shows that, when looking for a balanced S-separator, instead of trying to bound the number of elements of S in each connected component of $G[U \cup S] \setminus X$

separately, which could be problematic because of connectivity condition, we can just look for a partition of $G[U \cup S]$ into four sets with prescribed properties that can be checked locally. This suggest the following definition of table T_2.

In table T_2 we store entries for every node i of the tree decomposition, for every signature $\phi = (S_i, U_i)$ of B_i, and every 8-tuple $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$ where

- (M_1, M_2, M_3, X) is a partition of $S_i \cup U_i$,
- m_1, m_2, m_3 are integers between 0 and $|S|/2$,
- and x is an integer between 0 and $k+1$.

This 8-tuple ψ will be called the *interface*, and intuitively it encodes the interaction of a potential solution with the bag. Observe that the set U is not given in our graph directly but rather via connectivity information stored in table C, so we need to be prepared also for all the possible signatures of the bag; this is the reason why we introduce the interface on top of the signature. Note however, that the number of possible pairs (ϕ, ψ) is at most $9^{|B_i|} \cdot k^{O(1)}$, so for every bag B_i we store $9^{|B_i|} \cdot k^{O(1)}$ entries.

We proceed to the formal definition of what is stored in table T_2. For a fixed signature $\phi = (S_i, U_i)$ of B_i, let $(S_i^{\text{ext}}, U_i^{\text{ext}})$ be its extension, we say that partitioning $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ of $S_i^{\text{ext}} \cup U_i^{\text{ext}}$ is an *extension consistent* with interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$, if:

- $X^{\text{ext}} \cap B_i = X$ and $M_j^{\text{ext}} \cap B_i = M_j$ for $j = 1, 2, 3$;
- there is no edge between vertices of M_j^{ext} and $M_{j'}^{\text{ext}}$ for $j \neq j'$;
- $|X^{\text{ext}} \cap W_i| = x$ and $|M_j^{\text{ext}} \cap W_i| = m_j$ for $j = 1, 2, 3$.

In entry $T_2[i][\phi][\psi]$ we store:

- $\perp$ if ϕ is invalid or no consistent extension of ψ exists;
- otherwise, a list of length x of vertices of $X^{\text{ext}} \cap W_i$ in some consistent extension of ψ.

The query findSSeparatorcan be realized in $O(t^{O(1)})$ time by checking entries in the table T, namely $T[r][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, \emptyset, m_1, m_2, m_3, x)]$ for all possible values $0 \leq m_j \leq |S|/2$ and $0 \leq x \leq k+1$, and outputting the list contained in any of them that is not equal to $\perp$, or $\perp$ if all of them are equal to $\perp$.

We now present how to compute entries of table T_2 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_2[i][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, \emptyset, 0, 0, 0, 0)] = \emptyset$, and all the other interfaces are assigned $\perp$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and an interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let ϕ', ψ' be natural intersections of ϕ, ψ with B_j, respectively, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$ and $\psi' = (M_1 \cap B_j, M_2 \cap B_j, M_3 \cap B_j, X \cap B_j, m_1, m_2, m_3, x)$. Let $T_2[j][\phi'][\psi'] = L_j$. We consider some sub-cases, depending on the alignment of v in ϕ and ψ. The cases with v belonging to M_1, M_2 and M_3 are symmetric, so we consider only the case for M_1.

Case 2.1: $v \in X$**.** Note that every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, every extension consistent with ψ' can be extended to an extension consistent with ψ by adding v to the extension of X. Hence, it follows that we can simply take $L_i = L_j$.

Case 2.2: $v \in M_1$**.** Similarly as in the previous case, every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, if we are given an extension consistent with ψ', we can add v to M_1 and make an extension consistent with ψ if and only if v is not adjacent to any vertex of M_2 or M_3; this follows from the fact that B_j separates v from W_j, so the only vertices from M_2^{ext}, M_3^{ext} that v could be possibly adjacent to, lie in B_j. However, if v is adjacent to a vertex of M_2 or M_3, we can obviously put $L_i = \bot$, as there is no extension consistent with ψ: property that there is no edge between M_1^{ext} and $M_3^{\text{ext}} \cup M_3^{\text{ext}}$ is broken already in the bag. Otherwise, by the reasoning above we can put $L_i = L_j$.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$**.** Again, in this case we have one-to-one correspondence of extensions consistent with ψ with ψ' after trimming to B_j, so we may simply put $L_i = L_j$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i, and some interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let $\phi' = (S_j, U_j)$ be the only extension of signature ϕ to B_j that has the same extension as ϕ; ϕ' can be deduced by looking up which signatures are found valid in table C in the same manner as in the forget step for computation of table C. We consider three cases depending on alignment of w in ϕ':

Case 3.1: $w \notin S_j \cup U_j$**.** If w is not in $S_j \cup U_j$, then it follows that we may put $L_i = T_2[j][\phi'][\psi']$: extensions of ψ consistent with ψ correspond one-to-one to extensions consistent with ψ'.

Case 3.2: $w \in S_j$**.** Assume that there exist some extension consistent with ψ denoted $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$. In this extension, vertex w is either in M_1^{ext}, M_2^{ext}, M_3^{ext}, or in X^{ext}. Let us define the corresponding interfaces:

- $\psi_1 = (M_1 \cup \{w\}, M_2, M_3, X, m_1 - 1, m_2, m_3, x)$;
- $\psi_2 = (M_1, M_2 \cup \{w\}, M_3, X, m_1, m_2 - 1, m_3, x)$;
- $\psi_3 = (M_1, M_2, M_3 \cup \{w\}, X, m_1, m_2, m_3 - 1, x)$;

- $\psi_X = (M_1, M_2, M_3, X \cup \{w\}, m_1, m_2, m_3, x - 1)$.

If any of integers $m_1 - 1, m_2 - 1, m_3 - 1, x - 1$ turns out to be negative, we do not consider this interface. It follows that for at least one $\psi' \in \{\psi_1, \psi_2, \psi_3, \psi_X\}$ there must be an extension consistent with ψ': it is just the extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$. On the other hand, any extension consistent with any of interfaces $\psi_1, \psi_2, \psi_3, \psi_X$ is also consistent with ψ. Hence, we may simply put $L_i = T_2[i][\phi'][\psi']$, and append w on the list in case $\psi' = \psi_X$.

Case 3.3: $w \in U_j$**.** We proceed in the same manner as in Case 3.2, with the exception that we do not decrement m_j by 1 in interfaces ψ_j for $j = 1, 2, 3$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider a signature $\phi = (S_i, U_i)$ of B_i, and an interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. Assume that there is some extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ consistent with ψ. Define $m_q^p = |W_{j_p} \cap M_q|$ and $x^p = |W_{j_p} \cap X|$ for $p = 1, 2$ and $q = 1, 2, 3$; note that $m_q^1 + m_q^2 = m_q$ for $q = 1, 2, 3$ and $x^1 + x^2 = x$. It follows that in G_{j_1}, G_{j_2} there are some extensions consistent with $(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)$ and $(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)$, respectively—these are simply extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ intersected with V_i, V_j, respectively. On the other hand, if we have some extensions in G_{j_1}, G_{j_2} consistent with $(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)$ and $(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)$ for numbers m_p^q, x^p such that $m_q^1 + m_q^2 = m_q$ for $q = 1, 2, 3$ and $x^1 + x^2 = x$, then the point-wise union of these extensions is an extension consistent with $(M_1, M_2, M_3, X, m_1, m_2, m_3, x)$. It follows that in order to compute L_i, we need to check if for any such choice of m_p^q, x^p we have non-$\perp$ entries in

- $T_2[j_1][\phi_1][(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)]$ and
- $T_2[j_2][\phi_2][(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)]$.

This is the case, we put the union of the lists contained in these entries as L_i, and otherwise we put $\perp$. Note that computing the union of these lists takes $O(k)$ time as their lengths are bounded by k, and there is $O(k^4)$ possible choices of m_p^q, x^p to check.

Similarly as before, for every addition and removal of vertex v to and from S or marking and unmarking v as a pin, we can update table T_2 in $O(9^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_2 can be initialized in $O(9^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_2 must be performed after updating/initializing tables P and C.

29.4.3 Query for next pin

We now proceed to the next query. Recall that at each point, the algorithm maintains the set F of vertices marking components of $G[U \cup S] \setminus (X \cup S)$ that have been already processed. A component is marked as processed when one of its vertices is added to F. Hence, we need a query that finds the next component to process by returning any of its vertices. As in the linear-time approximation algorithm we need to process the components in decreasing order of sizes, the query in fact provides a vertex of the largest component.

findNextPin

Output:	A pair (u, ℓ), where *(i)* u is a vertex of a component of $G[U \cup S] \setminus (X \cup S)$ that does not contain a vertex from F and is of maximum size among such components, and *(ii)* ℓ is the size of this component; or, $\bot$ if no such component exists.
Time:	$O(1)$

To implement the query we create a table similar to table C, but with entry indexing enriched by subsets of the bag corresponding to possible intersections with X and F. Formally, we store entries for every node i, and for every signature $\phi = (S_i, U_i, X_i, F_i)$, which is a quadruple of subsets of B_i such that (i) $S_i \cap U_i = \emptyset$, (ii) $X_i \subseteq S_i \cup U_i$, (iii) $F_i \subseteq U_i \setminus X_i$. The number of such signatures is equal to $6^{|B_i|}$.

For a signature $\phi = (S_i, U_i, X_i, F_i)$, we say that $(S_i^{\text{ext}}, U_i^{\text{ext}}, X_i^{\text{ext}}, F_i^{\text{ext}})$ is the extension of ϕ if (i) $(S_i^{\text{ext}}, U_i^{\text{ext}})$ is the extension of (S_i, U_i) as in the table C, (ii) $X_i^{\text{ext}} = X_i \cup (W_i \cap X)$ and $F_i^{\text{ext}} = F_i \cup (W_i \cap F)$. We may now state what is stored in entry $T_3[i][(S_i, U_i, X_i, F_i)]$:

- if (S_i, U_i) is invalid then we store $\bot$;
- otherwise we store:
  - an equivalence relation R between vertices of $U_i \setminus X_i$, such that $(v_1, v_2) \in R$ if and only if v_1, v_2 are connected in $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$;
  - for every equivalence class K of R, an integer m_K equal to the number of vertices of the connected component of $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$ containing K, which are contained in W_i, or to $\bot$ if this connected component contains a vertex of F_i^{ext};
  - a pair (u, m), where m is equal to the size of the largest component of $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$ not containing any vertex of F_i^{ext} or U_i, while u is any vertex of this component; if no such component exists, then $(u, m) = (\bot, \bot)$.

Clearly, query findNextPinmay be implemented by outputting the pair (u, m) stored in the entry $T_3[r][(\emptyset, \emptyset, \emptyset, \emptyset)]$, or $\bot$ if this pair is equal to $(\bot, \bot)$.

We now present how to compute entries of table T_3 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures (S_i, U_i, X_i, F_i) for which (S_i, U_i) is valid, as for the invalid ones we just put value $\bot$.

Case 1: Leaf node. If i is a leaf node then $T_3[i][(\emptyset, \emptyset, \emptyset, \emptyset)] = (\emptyset, \emptyset, (\bot, \bot))$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute $T_3[i][\phi] = (R_i, (m^i_K)_{K \in R_i}, (u_i, m_i))$. Let ϕ' be a natural projection of ϕ onto B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j, X_i \cap B_j, F_i \cap B_j)$. Let $T_3[j][\phi'] = (R_j, (m^j_K)_{K \in R_j}, (u_j, m_j))$; note that this entry we know, but entry $T_3[i][\phi]$ we would like to compute. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in U_i \setminus (X_i \cup F_i)$. If we introduce a vertex from $U_i \setminus (X_i \cup F_i)$, then the extension of ϕ is just the extension of ϕ' plus vertex v added to U_i^{ext}. If we consider the equivalence classes of R_i, then these are equivalence classes of R_j but possibly some of them have been merged because of connections introduced by vertex v. As B_j separates v from W_j, v could only create connections between two vertices from $B_j \cap (U_j \setminus X_j)$. Hence, we can obtain R_i from R_j by merging all the equivalence classes of vertices of $U_j \setminus X_j$ adjacent to v; the corresponding entry in sequence $(m_K)_{K \in R_i}$ is equal to the sum of entries from the sequence $(m^j_K)_{K \in R_j}$ corresponding to the merged classes. If any of these entries is equal to $\bot$, we put simply $\bot$. If v was not adjacent to any vertex of $U_j \setminus X_j$, we put v in a new equivalence class K with $m_K = 0$. Clearly, we can also put $(u_i, m_i) = (u_j, m_j)$.

Case 2.2: $v \in (U_i \setminus X_i) \cap F_i$. We perform in the same manner as in Case 2.2, with the exception that the new entry in sequence $(m_K)_{K \in R_i}$ will be always equal to $\bot$, as the corresponding component contains a vertex from F_i^{ext}.

Case 2.3: $v \in S_i \cup X_i$. In this case we can simply put $T_3[i][\phi] = T_3[j][\phi']$ as the extensions of ϕ and ϕ' are the same with the exception of v being included into X_i^{ext} and/or into S_i^{ext}, which does not influence information to be stored in the entry.

Case 2.4: $v \in B_i \setminus (S_i \cup U_i)$. In this case we can simply put $T_3[i][\phi] = T_3[j][\phi']$ as the extensions of ϕ and ϕ' are equal.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute

$$T_3[i][\phi] = (R_i, (m^i_K)_{K \in R_i}, (u_i, m_i)).$$

Let $(S_i^{\text{ext}}, U_i^{\text{ext}}, X_i^{\text{ext}}, F_i^{\text{ext}})$ be extension of ϕ. Observe that there is exactly one signature $\phi' = (S_j, U_j, X_j, F_j)$ of B_j with the same extension as ϕ, and this signature is simply ϕ with w added possibly to S_i, U_i, X_i or F_i, depending whether it belongs to S_i^{ext}, U_i^{ext}, X_i^{ext}, or F_i^{ext}. Coloring ϕ' may be defined similarly as in case of forget node for table C; we just need in addition to include w in X_i^{ext} or F_i^{ext} if it belongs to X or F, respectively.

Let $T_3[j][\phi] = (R_j, (m^j_K)_{K \in R_j}, (u_j, m_j))$. As the extensions of ϕ and ϕ' are equal, it follows that we may take R_i equal to R_j with w possibly excluded from its equivalence class. Similarly, for every equivalence class $K \in R_i$ we put m^i_K equal to $m^j_{K'}$, where K' is the corresponding equivalence class of R_j, except the class that contained w which should get the previous number incremented by 1, providing it was not equal to $\bot$. We also put $(u_i, m_i) = (u_j, m_j)$ except the situation, when we forget the last vertex of a component of $G[U_j^{\text{ext}} \setminus X_j^{\text{ext}}]$: this is the case when w is in $U_j \setminus X_j$ and constitutes a singleton equivalence class of R_j. Let then $m^j_{\{w\}}$ be the corresponding entry in sequence $(m^j_K)_{K \in R_j}$. If $m^j_{\{w\}} = \bot$, we simply put $(u_i, m_i) = (u_j, m_j)$. Else, if $(u_j, m_j) = (\bot, \bot)$ or $m^j_{\{w\}} > m_j$, we put $(u_i, m_i) = (w, m^j_{\{w\}})$, and otherwise we put $(u_i, m_i) = (u_j, m_j)$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute $T_3[i][\phi] = (R_i, (m^i_K)_{K \in R_i}, (u_i, m_i))$. Let $\phi_1 = (S_i, U_i, X_i, F_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i, X_i, F_i)$ be a signature of B_{j_2}. Let $T_3[j_1][\phi_1] = (R_{j_1}, (m^{j_1}_K)_{K \in R_{j_1}}, (u_{j_1}, m_{j_1}))$ and $T_3[j_2][\phi_2] = (R_{j_2}, (m^{j_2}_K)_{K \in R_{j_2}}, (u_{j_2}, m_{j_2}))$. Note that equivalence relations R_{j_1} and R_{j_2} are defined on the same set $U_i \setminus X_i$. It follows from the definition of T_3 that we can put:

- R_i to be the minimum transitive closure of $R_{j_1} \cup R_{j_2}$;
- for every equivalence class K of R_i, m^i_K equal to the sum of (i) numbers $m^{j_1}_{K_1}$ for $K_1 \subseteq K$, K_1 being an equivalence class of R_{j_1}, and (ii) numbers $m^{j_2}_{K_2}$ for $K_2 \subseteq K$, K_2 being an equivalence class of R_{j_2}; if any of these numbers is equal to $\bot$, we put $m^i_K = \bot$;
- (u_i, m_i) to be equal to (u_{j_1}, m_{j_1}) or (u_{j_2}, m_{j_2}), depending whether m_{j_1} or m_{j_2} is larger; if any of these numbers is equal to $\bot$, we take the second one, and if both are equal to $\bot$, we put $(u_i, m_i) = (\bot, \bot)$.

Similarly as before, for every addition and removal of vertex v to and from S, to and from X, to and from F, or marking and unmarking v as a pin, we can update table T_3 in $O(6^t \cdot t^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_3 can be initialized in $O(6^t \cdot t^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_3 must be performed after updating/initializing tables P and C.

29.4.4 Query for finding a graph separator

In this section we implement the last query, needed for the linear-time algorithm; the query is significantly more involved than the previous one. The query specification is as follows:

findUSeparator

Output:	A list of elements of a $\frac{8}{9}$-balanced separator of $G[U]$ of size at most $k+1$, or $\bot$ if no such exists.
Time:	$O(c^t \cdot k^{O(1)} \cdot \log n)$

Note that Lemma 25.1 guarantees that in fact $G[U]$ contains a $\frac{1}{2}$-balanced separator of size at most $k+1$. Unfortunately, we are not able to find a separator with such a good guarantee on the sizes of the sides; the difficulties are explained in Chapter 25. Instead, we again make use of the precomputed approximate tree decomposition to find a balanced separator with slightly worse guarantees on the sizes of the sides.

In the following we will also use the notion of a *balanced separation*. For a graph G, we say that a partition (L, X, R) of $V(G)$ is an *α -balanced separation of G*, if there is no edge between L and R, and $|L|, |R| \leq \alpha|V(G)|$. The *order* of a separation is the size of X. Clearly, if (L, X, R) is an α-balanced separation of G, then X is an α-balanced separator of G. By folklore (see the proof of Lemma 25.2) we know that every graph of treewidth at most k has a $\frac{2}{3}$-balanced separation of order at most $k+1$.

Express searching for a balanced separator as a maximization problem

Before we start explaining the query implementation, we begin with a few definitions that enable us to express finding a balanced separator as a simple maximization problem.

Definition 29.4. Let G be a graph, and T_L, T_R be disjoint sets of terminals in G. We say that a partition (L, X, R) of $V(G)$ is a *terminal separation of G of order ℓ*, if the following conditions are satisfied:

(i) $T_L \subseteq L$ and $T_R \subseteq R$;

(ii) there is no edge between L and R;

(iii) $|X| \leq \ell$.

We moreover say that (L, X, R) is *left-pushed* (*right-pushed*) if $|L|$ ($|R|$) is maximum among possible terminal separations of order ℓ.

Pushed terminal separations are similar to important separators of Marx [Mar06], and their number for fixed T_L, T_R can be exponential in ℓ. Pushed terminal separations are useful for us because of the following lemma, that enables us to express finding a small balanced separator as a maximization problem, providing that some separator of a reasonable size is given.

Lemma 29.5. *Let G be a graph of treewidth at most k and let (A_1, B, A_2) be some separation of G, such that $|A_1|, |A_2| \leq \frac{3}{4}|V(G)|$. Then there exists a partition (T_L, X_B, T_R) of B and integers k_1, k_2 with $k_1 + k_2 + |X_B| \leq k+1$, such that if G_1, G_2 are $G[A_1 \cup (B \setminus X_B)]$ and $G[A_2 \cup (B \setminus X_B)]$ with terminals T_L, T_R, then*

(i) there exist a terminal separations of G_1, G_2 of orders k_1, k_2, respectively;

(ii) for any left-pushed terminal separation (L_1, X_1, R_1) of order k_1 in G_1 and any right-pushed separation (L_2, X_2, R_2) of order k_2 in G_2, the triple $(L_1 \cup T_L \cup L_2, X_1 \cup X_B \cup X_2, R_1 \cup T_R \cup R_2)$ is a terminal separation of G of order at most $k+1$ with $|L_1 \cup T_L \cup L_2|, |R_1 \cup T_R \cup R_2| \leq \frac{7}{8}|V(G)| + \frac{|X|+(k+1)}{2}$.

Proof. As the treewidth of G is at most k, there is a separation (L, X, R) of G such that $|L|, |R| \leq \frac{2}{3}|V(G)|$ and $|X| \leq k + 1$ by folklore [see the proof of lemma 25.2]. Let us set $(T_L, X_B, T_R) = (L \cap B, X \cap B, R \cap B)$, $k_1 = |X \cap A_1|$ and $k_2 = |X \cap A_2|$. Observe that $X \cap A_1$ and $X \cap A_2$ are terminal separations in G_1 and G_2 of orders k_1 and k_2, respectively, hence we are done with (i). We proceed to the proof of (ii).

Let us consider sets $L \cap A_1$, $L \cap A_2$, $R \cap A_1$ and $R \cap A_2$. Since (A_1, B, A_2) and (L, X, R) are $\frac{1}{4}$- and $\frac{1}{3}$- balanced separations, respectively, we know that:

- $|L \cap A_1| + |L \cap A_2| + |B| \geq \frac{1}{3}|V(G)| - (k + 1)$;
- $|R \cap A_1| + |R \cap A_2| + |B| \geq \frac{1}{3}|V(G)| - (k + 1)$;
- $|L \cap A_1| + |R \cap A_1| + (k + 1) \geq \frac{1}{4}|V(G)| - |B|$;
- $|L \cap A_2| + |R \cap A_2| + (k + 1) \geq \frac{1}{4}|V(G)| - |B|$.

We claim that either $|L\cap A_1|, |R\cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$, or $|L\cap A_2|, |R\cap A_1| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Assume first that $|L \cap A_1| < \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Observe that then $|L \cap A_2| \geq \frac{1}{3}|V(G)| - |B| - (k + 1) - (\frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}) \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Similarly, $|R \cap A_1| \geq \frac{1}{4}|V(G)| - |B| - (k + 1) - (\frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}) \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. The case when $|R \cap A_2| < \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$ is symmetric. Without loss of generality, by possibly flipping separation (L, X, R), assume that $|L \cap A_1|, |R \cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$.

Let (L_1, X_1, R_1) be any left-pushed terminal separation of order k_1 in G_1 and (L_2, X_2, R_2) be any right-pushed terminal separation of order k_2 in G_2. By the definition of being left- and right-pushed, we have that $|L_1 \cap A_1| \geq |L \cap A_1| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$ and $|R_2 \cap A_2| \geq |R \cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Therefore, we have that $|L_1 \cup T_L \cup L_2| \leq \frac{7}{8}|V(G)| + \frac{|B|+(k+1)}{2}$ and $|L_1 \cup T_L \cup L_2| \leq \frac{7}{8}|V(G)| + \frac{|B|+(k+1)}{2}$. $\square$

The idea of the rest of the implementation is as follows. First, given an approximate tree decomposition of with $O(k)$ in the data structure, in logarithmic time we will find a bag B_{i_0} that splits the component U in a balanced way. This bag will be used as the separator B in the invocation of Lemma 29.5; the right part of the separation will consist of vertices contained in the subtree below B_{i_0}, while the whole rest of the tree will constitute the left part. Lemma 29.5 ensures us that we may find some balanced separator of U by running two maximization dynamic programs: one in the subtree below B_{i_0} to identify a right-pushed separation, and one on the whole rest of the tree to find a left-pushed separation. As in all the other queries, we will store tables of these dynamic programs in the data structure, maintaining them with $O(c^t \log n)$ update times.

Case of a small U

At the very beginning of the implementation of the query we read $|U|$, which is stored in the entry $CardU[r][(\emptyset, \emptyset)]$. If it turns out that $|U| < 36(k+t+2) = O(k)$, we perform the following explicit construction. We apply a depth-first search from π to identify the whole U; note that this search takes $O(k^2)$ time, as U and S are bounded linearly in k. Then we build subgraph $G[U]$, which again takes $O(k^2)$ time. As this subgraph has $O(k)$ vertices and treewidth at most k, we may find its $\frac{1}{2}$-balanced separator of order at most $k+1$ in c^k time using a brute-force search through all the possible subsets of size at most $k+1$. This separator may be returned as the result of the query. Hence, from now on we assume that $|U| \geq 36(k+t+2)$.

Tracing U

We first aim to identify bag B_{i_0} in logarithmic time. The following lemma encapsulates the goal of this subsection. Note that we are not only interested in the bag itself, but also in the intersection of the bag with of S and U (defined as the connected component of $G \setminus S$ containing π). While intersection with S can be trivially computed given the bag, we will need to trace the intersection with U inside the computation.

Lemma 29.6. *There exists an algorithm that, given access to the data structure, in $O(t^{O(1)} \cdot \log n)$ time finds a node i_0 of the tree decomposition such that $|U|/4 \leq |W_{i_0} \cap U| \leq |U|/2$ together with two subsets U_i, S_i of B_{i_0} such that $U_0 = U \cap B_{i_0}$ and $S_0 = S \cap B_{i_0}$.*

Proof. The algorithm keeps track of a node i of the tree decomposition together with a pair of subsets $(U_i, S_i) = (B_i \cap U, B_i \cap S)$ being the intersections of the bag associated to the current node with U and S, respectively. The algorithm starts with the root node r and two empty subsets, and iteratively traverses down the tree keeping an invariant that $CardU[i][(U_i, S_i)] \geq |U|/2$. Whenever we consider a join node i with two sons j_1, j_2, we choose to go down to the node where $CardU[i_t][(U_{j_t}, U_{j_t})]$ is larger among $t = 1, 2$. In this manner, at each step $CardU[i][(U_i, S_i)]$ can be decreased by at most 1 in case of a forget node, or can be at most halved in case of a join node. As $|U| \geq 36(k+t+2)$, it follows that the first node i_0 when the invariant $CardU[i][(U_i, S_i)] \geq |U|/2$ ceases to hold, satisfies $|U|/4 \leq CardU[i_0][(U_{i_0}, S_{i_0})] \leq |U|/2$, and therefore can be safely returned by the algorithm.

It remains to argue how sets (U_i, S_i) can be updated at each step of the traverse down the tree. Updating S_i is trivial as we store an explicit table remembering for each vertex whether it belongs to S. Therefore, now we focus on updating U.

The cases of introduce and join nodes are trivial. If i is an introduce node with son j, then clearly $U_j = U_i \cap B_j$. Similarly, if i is a join node with sons j_1, j_2, then $U_{j_1} = U_{j_2} = U_i$. We are left with the forget node.

Let i be a forget node with son j, and let $B_j = B_i \cup \{w\}$. We have that $U_j = U_i \cup \{w\}$ or $U_j = U_i$, depending whether $w \in U$ or not. This information can be read from the table $C[j]$ as follows:

- if $C[j][(U_i \cup \{w\}, S_j)] = \bot$, then $w \notin U$ and $U_j = U_i$;

- if $C[j][(U_i, S_j)] = \bot$, then $w \in U$ and $U_j = U_i \cup \{w\}$;
- otherwise, both $C[j][(U_i, S_j)]$ and $C[j][(U_i \cup \{w\}, S_j)]$ are not equal to $\bot$; this follows from the fact that at least one of them, corresponding to the correct choice whether $w \in U$ or $w \notin U$, must be not equal to $\bot$. Observe that in this case w is in a singleton equivalence class of $C[j][(U_i \cup \{w\}, S_j)]$, and the connected component of w in the extension of $U_i \cup \{w\}$ cannot contain the pin π. It follows that $w \notin U$ and we take $U_j = U_i$.

Computation at each step of the tree traversal takes $O(t^{O(1)})$ time. As the tree has logarithmic depth, the whole algorithm runs in $O(t^{O(1)} \cdot \log n)$ time. □

Dynamic programming for pushed separators

In this subsection we show how to construct dynamic programming tables for finding pushed separators. The implementation resembles that of table T_2, used for balanced S-separators.

In table T_4 we store entries for every node i of the tree decomposition, for every signature $\phi = (S_i, U_i)$ of B_i, and for every 4-tuple $\psi = (L, X, R, x)$, called again the *interface*, where

- (L, X, R) is a partition of U_i,
- x is an integer between 0 and $k + 1$.

Again, the intuition is that the interface encodes the interaction of a potential solution with the bag. Note that for every bag B_i we store at most $5^{|B_i|} \cdot (k + 2)$ entries.

We proceed to the formal definition of what is stored in table T_4. Let us fix a signature $\phi = (S_i, U_i)$ of B_i, and let $(S_i^{\text{ext}}, U_i^{\text{ext}})$ be its extension. For an interface $\psi = (L, X, R, x)$, we say that a terminal separation $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ in $G[U_i^{\text{ext}}]$ with terminals L, R is an *extension consistent* with interface $\psi = (L, X, R, x)$ if

- $L^{\text{ext}} \cap B_i = L$, $X^{\text{ext}} \cap B_i = X$ and $R^{\text{ext}} \cap B_i = R$;
- $|X^{\text{ext}} \cap W_i| = x$.

Then entry $T_4[i][\phi][\psi]$ contains the pair (r, X_0) where r is the maximum possible $|R^{\text{ext}} \cap W_i|$ among extensions consistent with ψ, and X_0 is the corresponding set $X^{\text{ext}} \cap W_i$ for which this maximum was attained, or $\bot$ if the signature ϕ is invalid or no consistent extension exists.

We now present how to compute entries of table T_4 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\bot$.

Case 1: Leaf node. If i is a leaf node then $T_4[i][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, 0)] = (0, \emptyset)$, and all the other entries are assigned $\bot$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and an interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let ϕ', ψ' be natural intersections of ϕ, ψ with B_j, respectively, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$ and $\psi' = (L \cap B_j, X \cap B_j, R \cap B_j, x)$. Let $T_4[j][\phi'][\psi'] = (r_j, X_0^j)$. We consider some sub-cases, depending on the alignment of v in ϕ and ψ. The cases with v belonging to L and R are symmetric, so we consider only the case for L.

Case 2.1: $v \in X$**.** Note that every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, every extension consistent with ψ' can be extended to an extension consistent with ψ by adding v to the extension of X. Hence, it follows that we can simply take $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 2.2: $v \in L$**.** Similarly as in the previous case, every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, if we are given an extension consistent with ψ', then we can add v to L and make an extension consistent with ψ if and only if v is not adjacent to any vertex of R; this follows from the fact that B_j separates v from W_j, so the only vertices from R^{ext} that v could be possibly adjacent to, lie in B_j. However, if v is adjacent to a vertex of R, then we can obviously put $(r_i, X_0^i) = \bot$ as there is no extension consistent with ψ: property that there is no edge between L and R is broken already in the bag. Otherwise, by the reasoning above we can put $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 2.3: $v \in B_i \setminus U_i$**.** Again, in this case we have one-to-one correspondence of extensions consistent with ψ and extensions consistent with ψ', so we may simply put $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i, and some interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let $\phi' = (S_j, U_j)$ be the only the extension of signature ϕ to B_j that has the same extension as ϕ; ϕ' can be deduced by looking up which signatures are found valid in table C in the same manner as in the forget step for computation of table C. We consider two cases depending on alignment of w in ϕ':

Case 3.1: $w \notin U_j$**.** If w is not in U_j, then it follows that we may put $(r_i, X_0^i) = T_4[j][\phi'][\psi']$: extensions consistent with ψ correspond one-to-one to extensions consistent with ψ'.

Case 3.2: $w \in U_j$**.** Assume that there exists some extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ consistent with ψ, and assume further that this extension is the one that maximizes $|R^{\text{ext}} \cap W_i|$. In this extension, vertex w is either in L^{ext}, X^{ext}, or in R^{ext}. Let us define the corresponding interfaces:

- $\psi_L = (L \cup \{w\}, X, R, x)$;
- $\psi_X = (L, X \cup \{w\}, R, x - 1)$;
- $\psi_R = (L, X, R \cup \{w\}, x)$.

If $x-1$ turns out to be negative, we do not consider ψ_X. For $t \in \{L, X, R\}$, let $(r_j, X_0^{j,t}) = T_4[j][\phi'][\psi_t]$. It follows that for at least one $\psi' \in \{\psi_L, \psi_X, \psi_R\}$ there must be an extension consistent with ψ': it is just the extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$. On the other hand, any extension consistent with any of interfaces ψ_L, ψ_X, ψ_R is also consistent with ψ. Hence, we may simply put $r_i = \max(r_L, r_X, r_R + 1)$, and define X_0^i as the corresponding $X_0^{j,t}$, with possibly w appended if $t = X$. Of course, in this maximum we do not consider the interfaces ψ_t for which $T_4[j][\phi'][\psi_t] = \bot$, and if $T_4[j][\phi'][\psi_t] = \bot$ for all $t \in \{L, X, R\}$, we put $(r_i, X_0^i) = \bot$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i, and an interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. Assume that there is some extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ consistent with ψ, and assume further that this extension is the one that maximizes $|R^{\text{ext}} \cap W_i|$. Define $r^p = |W_{j_p} \cap R|$ and $x^p = |W_{j_p} \cap X|$ for $p = 1, 2$; note that $r^1 + r^2 = r_i$ and $x^1 + x^2 = x$. It follows that in G_{j_1}, G_{j_2} there are some extensions consistent with (L, X, R, x^1) and (L, X, R, x^2), respectively—these are simply extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ intersected with V_i, V_j, respectively. On the other hand, if we have some extensions in G_{j_1}, G_{j_2} consistent with (L, X, R, x^1) and (L, X, R, x^2) for numbers x^p such that $x^1 + x^2 = x$, then the point-wise union of these extensions is an extension consistent with (L, X, R, x). It follows that in order to compute (r_i, X_0^i), we need to iterate through choices of x^p such that we have non-$\bot$ entries in $T_2[j_1][\phi_1][(L, X, R, x^1)] = (r_{j_1}^{x^1}, X_0^{j_1, x^1})$ and $T_2[j_2][\phi_2][(L, X, R, x^2)] = (r_{j_1}^{x^1}, X_0^{j_1, x^1})$, choose x^1, x^2 for which $r_{j_1}^{x^1} + r_{j_2}^{x^2}$ is maximum, and define $(r_i, X_0^i) = (r_{j_1}^{x^1} + r_{j_2}^{x^2}, X_0^{j_1, x^1} \cup X_0^{j_2, x^2})$. Of course, if for no choice of x^1, x^2 it is possible, we put $(r_i, X_0^i) = \bot$. Note that computing the union of the sets $X_0^{j_p, x^p}$ for $p = 1, 2$ takes $O(k)$ time as their sizes are bounded by k, and there is $O(t)$ possible choices of x^p to check.
Similarly as before, for every addition and removal of vertex v to and from S or marking and unmarking v as a pin, we can update table T_4 in $O(5^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_4 can be initialized in $O(5^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_4 must be performed after updating/initializing tables P and C.

Implementing the query findUSeparator

We now show how to combine Lemmata 29.5 and 29.6 with the construction of table T_4 to implement the query findUSeparator.

The algorithm performs as follows. First, using Lemma 29.6 we identify a node i_0 of the tree decomposition, together with disjoint subsets $(U_{i_0}, S_{i_0}) = (U \cap B_{i_0}, S \cap B_{i_0})$ of B_{i_0}, such that $|U|/4 \leq |W_{i_0} \cap U| \leq |U|/2$. Let $A_2 = W_{i_0}$ and $A_1 = V(G) \setminus V_{i_0}$. Consider separation $(A_1 \cap U, B_{i_0} \cap U, A_2 \cap U)$ of $G[U]$ and apply Lemma 29.5 to it. Let (T_L^0, X_B^0, T_R^0) be the partition of B_{i_0} and k_1^0, k_2^0 be the integers with $k_1^0 + k_2^0 + |X_B^0| \leq k+1$, whose existence is guaranteed by Lemma 29.5.

The algorithm now iterates through all possible partitions (T_L, X_B, T_R) of B_{i_0} and integers k_1, k_2 with $k_1 + k_2 + |X_B| \leq k+1$. We can clearly discard the partitions where there is an edge between T_L and T_R. For a partition (T_L, X_B, T_R), let G_1, G_2 be defined as in Lemma 29.5 for the graph $G[U]$. For a considered tuple $(T_L, X_B, T_R, k_1, k_2)$, we try to find:

(i) a separator of a right-pushed separation of order k_2 in G_2, and the corresponding cardinality of the right side;

(ii) a separator of a left-pushed separation of order k_1 in G_1, and the corresponding cardinality of the left side.

Goal *(i)* can be achieved simply by reading entries $T_4[i_0][(U_{i_0}, S_{i_0})][(T_L, X_B, T_R, k')]$ for $k' \leq k_2$, and taking the right-pushed separation with the largest right side. We are going to present how goal *(ii)* is achieved in the following paragraphs, but firstly let us show that achieving both of the goals is sufficient to answer the query.

Observe that if for some (T_L, X_B, T_R) and (k_1, k_2) we obtained both of the separators, denote them X_1, X_2, together with cardinalities of the corresponding sides, then using these cardinalities and precomputed $|U|$ we may check whether $X_1 \cup X_2 \cup X_B$ gives us a $\frac{8}{9}$-separation of $G[U]$. On the other hand, Lemma 29.5 asserts that when (T_L^0, X_B^0, T_R^0) and (k_1^0, k_2^0) are considered, we will find some pushed separations, and moreover any such two separations will yield a $\frac{8}{9}$-separation of $G[U]$. Note that this is indeed the case as the sides of the obtained separation have cardinalities at most

$$\frac{7}{8}|U| + \frac{(k+1)+(t+1)}{2} = \frac{8}{9}|U| + \frac{k+t+2}{2} - \frac{|U|}{72} \leq \frac{8}{9}|U|,$$

since $|U| \geq 36(k+t+2)$.

We are left with implementing goal (ii). Let G_1' be G_1 with terminal sets swapped; clearly, left-pushed separations in G_1 correspond to right-pushed separations in G_1'. We implement finding a right-pushed separations in G_1' as follows.

Let $P = (i_0, i_1, \ldots, i_h = r)$ be the path from i_0 to the root r of the tree decomposition. The algorithm traverses the path P, computing tables $D[i_t]$ for consecutive indexes $t = 1, 2, \ldots, t$. The table $D[i_t]$ is indexed by signatures ϕ and interfaces ψ in the same manner as T_4. Formally, for a fixed signature $\phi = (S_{i_t}, U_{i_t})$ of B_{i_t} with extension $(S_{i_t}^{\text{ext}}, U_{i_t}^{\text{ext}})$, we say that this signature is *valid with respect to* (S_{i_0}, U_{i_0}) if it is valid and moreover $(S_{i_0}, U_{i_0}) = (S_{i_t}^{\text{ext}} \cap B_{i_0}, U_{i_t}^{\text{ext}} \cap B_{i_0})$. For an interface ψ we say that separation $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ in $G[U_i^{\text{ext}} \setminus W_{i_0}]$ with terminals L, R is *consistent with* ψ *with respect to* (T_L, X_B, T_R), if it is consistent in the same sense as in table T_4, and moreover $(T_L, X_B, T_R) = (L^{\text{ext}} \cap B_{i_0}, X^{\text{ext}} \cap B_{i_0}, R^{\text{ext}} \cap B_{i_0})$. Then entry $T[i_t][\phi][\psi]$ contains the pair (r, X_0) where r is the maximum possible $|R^{\text{ext}} \cap W_i|$ among extensions consistent with ψ with respect to (T_L, X_B, T_R), and X_0 is the corresponding set $X^{\text{ext}} \cap W_i$ for which this maximum was attained, or $\perp$ if the signature ϕ is invalid with respect to (S_{i_0}, U_{i_0}) or no such consistent extension exists.

The tables $D[i_t]$ can be computed by traversing the path P using the same recurrential formulas as for table T_4. When computing the next $D[i_t]$, we use table $D[i_{t-1}]$ computed in the previous step and possible table T_4 from the second child of i_t. Moreover, as $D[i_0]$ we insert the dummy table $\mathit{Dummy}[\phi][\psi]$ defined as follows:

- $\mathit{Dummy}[(U_{i_0}, S_{i_0})][(T_R, X_B, T_L, 0)] = 0$;
- all the other entries are evaluated to $\bot$.

It is easy to observe that table *Dummy* exactly satisfies the definition of $D[i_0]$. It is also straightforward to check that the recurrential formulas used for computing T_4 can be used in the same manner to compute tables $D[i_t]$ for $t = 1, 2, \ldots, h$. The definition of D and the method of constructing it show, that the values $D[r][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, x)]$ for $x = 0, 1, \ldots, k$, correspond to exactly right-pushed separations with separators of size exactly x in the graph G_1': insertion of the dummy table removes A_2 from the graph and forces the separation to respect the terminals in B_{i_0}.

Let us conclude with a summary of the running time of the query. Algorithm of Lemma 29.6 uses $O(t^{O(1)} \cdot \log n)$ time. Then we iterate through at most $O(3^t \cdot k^2)$ tuples (T_L, X_B, T_R) and (k_1, k_2), and for each of them we spend $O(k)$ time on achieving goal (i) and $O(5^t \cdot k^{O(1)} \cdot \log n)$ time on achieving goal (ii). Hence, in total the running time is $O(15^t \cdot k^{O(1)} \cdot \log n)$.

Chapter 30

Concluding remarks

In this part we have presented an algorithm that gives a constant factor approximation (with a factor 5) of the treewidth of a graph, which runs in single exponential time in the treewidth and linear in the number of vertices of the input graph.

Consequences

A large number of computational results use the following overall scheme: first find a tree decomposition of bounded width, and then run a dynamic programming algorithm on it. Many of these results use the linear-time exact algorithm of Bodlaender [Bod96] for the first step. If we aim for algorithms whose running time dependency on treewidth is single exponential, however, then our algorithm is preferable over the exact algorithm of Bodlaender [Bod96]. Indeed, many classical problems like Dominating Set and Independent Set are easily solvable in time $O(c^k \cdot n)$ when a tree decomposition of width k is provided, see Telle and Proskurowski [TP97]. Furthermore, there have been developed algorithms for problems seemingly not admitting so robust solutions; the fundamental examples are Steiner Tree, Traveling Salesman and Feedback Vertex Set [BCKN13]. With the given approximation algorithm at hand we can prove that all these problems also claim $O(c^k \cdot n)$ running time even if the decomposition is not given to us explicitly, as we may find its constant factor approximation within the same complexity bound.

Improvements and open problems

The presented results are mainly of theoretical importance due to the large constant c at the base of the exponent. One immediate open problem is to obtain a constant factor approximation algorithm for treewidth with running time $O(c^k n)$, where c is a *small* constant.

Another open problem is to find more efficient *exact* fixed-parameter tractable algorithms for treewidth. Bodlaender's algorithm [Bod96] and the version of Reed and Perković both use $k^{O(k^3)} \cdot n$ time; the dominant term being a call to the dynamic programming algorithm of Bodlaender and Kloks [BK96]. In fact, no exact fixed-parameter tractable algorithm for treewidth is known whose running time as a function of the parameter k is asymptotically smaller than this; testing the treewidth by verifying the forbidden minors can be expected to be significantly slower. Thus, it would be very interesting to have an exact algorithm for testing if the treewidth of a given graph is at most k in $2^{o(k^3)} \cdot n^{O(1)}$ time.

Currently, the best approximation ratio for treewidth for algorithms whose running time is polynomial in n and single exponential in the treewidth is the 3-approximation algorithm from Chapter 26. What is the best approximation ratio for treewidth that can be obtained in this running time? Is it possible to give lower bounds?

It seems that the idea of treating the tree decomposition as a data structure on which logarithmic-time queries can be implemented, can be similarly applied to all the problems expressible in MSOL. Extending the results in this direction seems like a thrilling perspective for future work.

Part VII

Concluding remarks

30.1 Future work

In this thesis we have studied 5 different problems, or classes of problems, and their behaviour from a parameterized perspective. We have seen that problems interact with the parameterized complexity landscape in numerous ways, sometimes yielding surprising results. To demonstrate this, we have given original results within most of the research directions highlighted in Part I. We end with a selected set of open problems.

Faster algorithms

In Part II we presented a $2^{O(k \log \ell)} n^{O(1)}$ algorithm for Component Order Connectivity and a tight lower bound stating that no $2^{o(k \log \ell)} n^{O(1)}$ algorithm exists, assuming ETH.

Open problem 1. Can Component Order Connectivity be solved in time $f(\ell) 2^{O(k)} n^{O(1)}$?

Open problem 2. Can we prove that there is no algorithm solving Component Order Connectivity in time $2^{O(k+\ell)} n^{O(1)}$, under ETH?

In Part III we presented an algorithm for Threshold Editing with running time $2^{O(\sqrt{k} \log k)} + n^{O(1)}$ and a lower bound under ETH stating that there is no $2^{o(\sqrt{k})} n^{O(1)}$ time algorithm for this problem. The gap between the two yields an interesting question of where the true complexity of the problem lies.

Open problem 3. Can Threshold Editing be solved in $2^{O(\sqrt{k})} n^{O(1)}$ time?

Polynomial kernels

Another result presented in Part III is a $O(k^2)$ vertex kernel. There might very well be unexplored structure to exploit for this problem.

Open problem 4. Does Threshold Editing admit an $O(k^{2-\epsilon})$ vertex kernel for some $\epsilon > 0$?

In Part IV we obtained polynomial kernels for both $\mathcal{H}$-Free Edge Deletion and $\mathcal{H}$-Free Edge Editing on graphs of bounded maximum degree. We also sketch how one can prove that neither of the problems admit polynomial kernels on graphs of bounded degeneracy. This makes it natural to ask about the kernelization complexity of graph classes in between the two.

Open problem 5. Does $\mathcal{H}$-Free Edge Deletion or $\mathcal{H}$-Free Edge Editing admit a polynomial kernel on planar graphs?

Bandwidth

In Part V we gave polynomial time approximation algorithms for Bandwidth on caterpillars, trees and graphs of bounded treelength, where the approximation factors depends solely on b. For trees and graphs of bounded treelength this dependency on b is exponential. However, the exponential approximation factor for graphs of bounded treelength stems purely from the exponential approximation factor of the algorithm for trees. Hence, when improving the approximation factors it is natural to focus our attention towards trees.

Open problem 6. Can we prove that Bandwidth does not admit a constant factor approximation in $f(b)n^{O(1)}$ time on trees?

Open problem 7. Does Bandwidth admit a $b^{O(1)}$-approximation in polynomial time on trees? What if one allows $f(b)n^{O(1)}$ time?

Furthermore, it would be very interesting to see if the approximation results mentioned above can be generalized to general graphs.

Open problem 8. Does Bandwidth admit a fixed-parameter tractable approximation algorithm on general graphs?

In addition to the approximation algorithms, we provided a characterization for having low bandwidth for trees, as well as graphs of bounded treelength, via obstructions. Are these obstructions a characterization for general graphs?

Open problem 9. Does there exist a function f such that any graph with pathwidth at most c_1, local density at most c_2, and containing no S_{c_3} as a subgraph has bandwidth at most $f(c_1, c_2, c_3)$?

Treewidth

In Part VI we presented a 5-approximation algorithm for treewidth running in time $O(c^k n)$ time for some constant c. It is thus natural to ask if one can improve upon this. And for practical purposes, it is necessary to improve significantly.

Open problem 10. Can treewidth be constant factor approximated in $O(c^k n)$ time, where c is a *small* constant?

Both of the fastest known exact algorithms for treewidth [Bod96, PR00] runs in $k^{O(k^3)}n$ time. And there is no algorithm known with a better dependency on k.

Open problem 11. Is there an exact algorithm for testing if the treewidth of a given graph is at most k in $2^{o(k^3)}n^{O(1)}$ time?

Shallow decompositions

When trying to transfer techniques from computing tree decompositions to decompositions that are more shallow, or thinner, we encounter difficulties. In particular, inserting a separator as part of the decomposition have much stronger implications further into the computations. Because of this, it would be very interesting so see if similar results could be obtained for such decompositions.

Open problem 12. Is there a constant factor approximation algorithm for pathwidth with running time $O(c^k n)$, for some constant c?

Definition 30.1 (Treedepth)**.** For a connected graph G we say that a rooted tree (T, r) is a *treedepth decomposition* of G if $V(T) = V(G)$ and for every edge $uv \in E(G)$ it holds that u and v have an ancestor-descendant relationship in T, i.e. u lies on the path from v to r or vice versa. The *depth* of the decomposition is the maximum depth of all leaves in T. The *treedepth* of a connected graph G is the minimum depth over all treedepth decompositions of G. For a disconnected graph we define the treedepth to be the maximum treedepth over all its connected components.

Open problem 13. Is there a constant factor approximation algorithm for treedepth with running time $O(c^k n)$, for some constant c?

Bibliography

[ABDR12] I. Abraham, M. Babaioff, S. Dughmi, and T. Roughgarden. Combinatorial auctions with restricted complements. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, pages 3–16. ACM, 2012.

[ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[AF93] K. R. Abrahamson and M. R. Fellows. Finite automata, bounded treewidth and well-quasiordering. In N. Robertson and P. Seymour, editors, *Proceedings of the AMS Summer Workshop on Graph Minors, Graph Structure Theory, Contemporary Mathematics vol. 147*, pages 539–564. American Mathematical Society, 1993.

[AK10] F. N. Abu-Khzam. A kernelization algorithm for d-hitting set. *J. Comput. Syst. Sci.*, 76(7):524–531, 2010.

[ALS09] N. Alon, D. Lokshtanov, and S. Saurabh. Fast FAST. In *International Colloquium on Automata, Languages, and Programming*, pages 49–58. Springer, 2009.

[Ami10] E. Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010.

[APSZ81] S. Assmann, G. Peck, M. Syslo, and J. Zak. The bandwidth of caterpillars with hairs of length 1 and 2. *SIAM Journal on Algebraic Discrete Methods*, 2(4):387–393, 1981.

[APW12] P. Austrin, T. Pitassi, and Y. Wu. Inapproximability of treewidth, one-shot pebbling, and related layout problems. In A. Gupta, K. Jansen, J. D. P. Rolim, and R. A. Servedio, editors, *Proceedings APPROX 2012 and RANDOM 2012*, volume 7408 of *lncs*, pages 13–24. Springer, 2012.

[ASS14] N. R. Aravind, R. B. Sandeep, and N. Sivadasan. On polynomial kernelization of $\mathcal{H}$-free edge deletion. In *IPEC*, 2014.

[BAMSN13] W. Ben-Ameur, M.-A. Mohamed-Sidi, and J. Neto. The k-separator problem. In D.-Z. Du and G. Zhang, editors, *COCOON*, volume 7936 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2013.

[BBD06] P. Burzyn, F. Bonomo, and G. Durán. NP-completeness results for edge modification problems. *Discrete Appl. Math.*, 154(13):1824–1844, 2006.

[BBG$^+$92] K. S. Bagga, L. W. Beineke, W. Goddard, M. J. Lipman, and R. E. Pippert. A survey of integrity. *Discrete Applied Mathematics*, 37:13–28, 1992.

[BCKN13] H. L. Bodlaender, M. Cygan, S. Kratsch, and J. Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In *International Colloquium on Automata, Languages, and Programming*, pages 196–207. Springer, 2013.

[BDD$^+$16] 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, 2016.

[BDFH09] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009.

[BEPvR12] N. Bourgeois, B. Escoffier, V. T. Paschos, and J. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 62(1-2):382–415, 2012.

[BES87] C. A. Barefoot, R. Entringer, and H. Swart. Vulnerability in graphs—a comparative survey. *J. Combin. Math. Combin. Comput*, 1(38):13–22, 1987.

[BF05] H. L. Bodlaender and F. V. Fomin. Equitable colorings of bounded treewidth graphs. *Theoretical Computer Science*, 349(1):22–30, 2005.

[BFGR16] R. Belmonte, F. V. Fomin, P. A. Golovach, and M. Ramanujan. Metric dimension of bounded tree-length graphs. *arXiv preprint arXiv:1602.02610*, 2016.

[BFH94] H. L. Bodlaender, M. R. Fellows, and M. T. Hallett. Beyond NP-completeness for problems of bounded width: hardness for the W-hierarchy. In *STOC*, pages 449–458, 1994.

[BFPP14] I. Bliznets, F. V. Fomin, M. Pilipczuk, and M. Pilipczuk. A subexponential parameterized algorithm for proper interval completion. In *European Symposium on Algorithms*, pages 173–184. Springer, 2014.

[BFPP16] I. Bliznets, F. V. Fomin, M. Pilipczuk, and M. Pilipczuk. Subexponential parameterized algorithm for interval completion. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1131. SIAM, 2016.

[BG93] J. F. Buss and J. Goldsmith. Nondeterminism within p^*. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[BGHK95] 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.

[BH98] H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27(6):1725–1746, 1998.

[BHKK07] H.-J. Bockenhauer, J. Hromkovic, J. Kneis, and J. Kupke. The parameterized approximability of TSP with deadlines. *Theory of Computing Systems*, 41(3):431–444, 2007.

[BJK14] H. L. Bodlaender, B. M. Jansen, and S. Kratsch. Kernelization lower bounds by cross-composition. *SIAM Journal on Discrete Mathematics*, 28(1):277–305, 2014.

[BK96] 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.

[BLS99] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes. A Survey.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, USA, 1999.

[Bod88] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In T. Lepistö and A. Salomaa, editors, *Automata, Languages and Programming, 15th International Colloquium, ICALP88, Tampere, Finland, July 11-15, 1988, Proceedings*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 1988.

[Bod89] H. L. Bodlaender. Improved self-reduction algorithms for graphs with bounded treewidth. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 232–244. Springer, 1989.

[Bod93] H. L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 112–124. Springer, 1993.

[Bod96] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

[Bod98] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical computer science*, 209(1):1–45, 1998.

[Bor88] R. B. Borie. *Recursively Constructed Graph Families.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1988.

[Bra14] U. Brandes. Social network algorithmics. ISAAC, 2014.

[Cai96] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Inf. Process. Lett.*, 58(4):171–176, 1996.

[CC15] L. Cai and Y. Cai. Incompressibility of H-free edge modification problems. *Algorithmica*, 71(3):731–757, 2015.

[CCDG82] P. Z. Chinn, J. Chvátalová, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices—a survey. *Journal of Graph Theory*, 6(3):223–254, 1982.

[CDE$^+$08] V. Chepoi, F. Dragan, B. Estellon, M. Habib, and Y. Vaxès. Diameters, centers, and approximating trees of delta-hyperbolicgeodesic spaces and graphs. In *Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 59–68. ACM, 2008.

[CEF87] L. H. Clark, R. C. Entringer, and M. R. Fellows. Computational complexity of integrity. *J. Combin. Math. Combin. Comput*, 2:179–191, 1987.

[CFK$^+$15] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*, volume 4. Springer, 2015.

[CHKX06] J. Chen, X. Huang, I. A. Kanj, and G. Xia. Strong computational lower bounds via parameterized complexity. *Journal of Computer and System Sciences*, 72(8):1346–1367, 2006.

[CJL03] Z.-Z. Chen, T. Jiang, and G. Lin. Computing phylogenetic roots with bounded degrees and errors. *SIAM Journal on Computing*, 32(4):864–879, 2003.

[CKJ01] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[CKX10] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40):3736–3756, 2010.

[CL] Y. Chen and B. Lin. The constant inapproximability of the parameterized dominating set problem. In *To appear at FOCS 2016.*

[CM14] Y. Cao and D. Marx. Chordal Editing is fixed-parameter tractable. In *STACS*, volume 25 of *LIPIcs*, pages 214–225. Schloss Dagstuhl, 2014.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[CS89] F. R. K. Chung and P. D. Seymour. Graphs with small bandwidth and cutwidth. *Discrete Mathematics*, 75(1-3):113–119, 1989.

[CSTV93] R. F. Cohen, S. Sairam, R. Tamassia, and J. S. Vitter. Dynamic algorithms for optimization problems in bounded tree-width graphs. In G. Rinaldi and L. A. Wolsey, editors, *Proceedings of the 3rd Conference on Integer Programming and Combinatorial Optimization, IPCO'93*, pages 99–112, 1993.

[CZ98] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. part II: Optimal parallel algorithms. *Theoretical Computer Science*, 203(2):205–223, 1998.

[CZ00] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. part I: Sequential algorithms. *Algorithmica*, 27(3-4):212–226, 2000.

[Dam06] P. Damaschke. Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theoretical Computer Science*, 351(3):337–350, 2006.

[DDLS15] P. G. Drange, M. S. Dregi, D. Lokshtanov, and B. D. Sullivan. On the threshold of intractability. In *Algorithms-ESA 2015*, pages 411–423. Springer, 2015.

[DDS16] P. G. Drange, M. Dregi, and R. Sandeep. Compressing bounded degree graphs. In *Latin American Symposium on Theoretical Informatics*, pages 362–375. Springer, 2016.

[DDvH] P. G. Drange, M. Dregi, and P. van't Hof. On the computational complexity of vertex integrity and component order connectivity. *Algorithmica*, pages 1–22.

[DF95] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science*, 141(1):109–131, 1995.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized complexity.* Springer, 1999.

[DF13] R. G. Downey and M. R. Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

[DFHT05] E. D. Demaine, F. V. Fomin, M. Hajiaghayi, and D. M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and h-minor-free graphs. *Journal of the ACM (JACM)*, 52(6):866–893, 2005.

[DFMR08] R. G. Downey, M. R. Fellows, C. McCartin, and F. Rosamond. Parameterized approximation of dominating set problems. *Information Processing Letters*, 109(1):68–70, 2008.

[DFPV15] P. G. Drange, F. V. Fomin, M. Pilipczuk, and Y. Villanger. Exploring the subexponential complexity of completion problems. *ACM Transactions on Computation Theory (TOCT)*, 7(4):14, 2015.

[DFU11] C. Dubey, U. Feige, and W. Unger. Hardness results for approximating the bandwidth. *Journal of Computer and System Sciences*, 77(1):62–90, 2011.

[DG07] Y. Dourisboure and C. Gavoille. Tree-decompositions with bags of small diameter. *Discrete Mathematics*, 307(16):2008–2029, 2007.

[DHK05] E. D. Demaine, M. T. Hajiaghayi, and K.-i. Kawarabayashi. Algorithmic graph minor theory: Decomposition, approximation, and coloring. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 637–646. IEEE, 2005.

[Die05] R. Diestel. Graph theory. 2005. *Grad. Texts in Math*, 101, 2005.

[DKL14] F. F. Dragan, E. Köhler, and A. Leitert. Line-distortion, bandwidth and path-length of a graph. In *Scandinavian Workshop on Algorithm Theory*, pages 158–169. Springer, 2014.

[DL14] M. S. Dregi and D. Lokshtanov. Parameterized complexity of bandwidth on trees. In *International Colloquium on Automata, Languages, and Programming*, pages 405–416. Springer, 2014.

[DLL$^+$06] F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. In *Parameterized and Exact Computation*, pages 13–24. Springer, 2006.

[DLS09] M. Dom, D. Lokshtanov, and S. Saurabh. Incompressibility through colors and ids. In *International Colloquium on Automata, Languages, and Programming*, pages 378–389. Springer, 2009.

[DP15] P. G. Drange and M. Pilipczuk. A polynomial kernel for trivially perfect editing. In *ESA*, volume 9294 of *Lecture Notes in Computer Science*, pages 424–436. Springer, 2015.

[Dru15] A. Drucker. New limits to classical and quantum instance compression. *SIAM Journal on Computing*, 44(5):1443–1479, 2015.

[DRVS15] P. G. Drange, F. Reidl, F. S. Villaamil, and S. Sikdar. Fast biclustering by dual parameterization. In T. Husfeldt and I. A. Kanj, editors, *10th International Symposium on Parameterized and Exact Computation, IPEC 2015, September 16-18, 2015, Patras, Greece*, volume 43 of *LIPIcs*, pages 402–413. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[DV01] J. Dunagan and S. Vempala. On euclidean embeddings and bandwidth minimization. In *RANDOM-APPROX*, pages 229–240, 2001.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[EJT10] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 143–152. IEEE, 2010.

[EK72] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[EM15] J. Enright and K. Meeks. Deleting edges to restrict the size of an epidemic: a new application for treewidth. In *Combinatorial Optimization and Applications*, pages 574–585. Springer, 2015.

[EMC88] E. S. El-Mallah and C. J. Colbourn. The complexity of some edge deletion problems. *IEEE Transactions on Circuits and Systems*, 35(3):354–362, 1988.

[ER60] P. Erdös and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.

[EST94] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Information and Computation*, 113(1):50–79, 1994.

[Fei00] U. Feige. Approximating the bandwidth via volume respecting embeddings. *J. Comput. Syst. Sci.*, 60(3):510–539, 2000.

[FG06] J. Flum and M. Grohe. Parameterized complexity theory, volume XIV of texts in theoretical computer science. an eatcs series, 2006.

[FHL08] 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.

[FK10] F. V. Fomin and D. Kratsch. *Exact exponential algorithms.* Springer Science & Business Media, 2010.

[FL] M. Fellows and M. Langston. An analogue of the myhill-nerode theorem and its use in computing finite-basis characterizations. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 520–525. IEEE.

[FLM$^+$08] M. R. Fellows, D. Lokshtanov, N. Misra, F. A. Rosamond, and S. Saurabh. Graph layout problems parameterized by vertex cover. In *International Symposium on Algorithms and Computation*, pages 294–305. Springer, 2008.

[FLRS07] M. R. Fellows, M. A. Langston, F. A. Rosamond, and P. Shaw. Efficient parameterized preprocessing for cluster editing. In *FCT*. Springer, 2007.

[FMT09] T. Feder, H. Mannila, and E. Terzi. Approximating the minimum chain completion problem. *Information Processing Letters*, 109(17):980–985, 2009.

[FS89] M. Fellows and S. Stueckle. The immersion order, forbidden subgraphs and the complexity of network integrity. *J. Combin. Math. Combin. Comput*, 6:23–32, 1989.

[FS08] L. Fortnow and R. Santhanam. Infeasibility of instance compression and succinct PCPs for NP. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 133–142. ACM, 2008.

[FT09] U. Feige and K. Talwar. Approximating the bandwidth of caterpillars. *Algorithmica*, 55(1):190–204, 2009.

[FV13] F. V. Fomin and Y. Villanger. Subexponential parameterized algorithm for minimum fill-in. *SIAM Journal on Computing*, 42(6):2197–2216, 2013.

[GGHN09] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction and exact algorithms for clique cover. *ACM J Exp Algorithmics*, 13:2:2.2–2:2.15, 2009.

[GHI$^+$13] D. Gross, M. Heinig, L. Iswara, L. W. Kazmierczak, K. Luttrell, J. T. Saccoman, and C. Suffel. A survey of component order connectivity models of graph theoretic networks. *WSEAS Transactions on Mathematics*, 12:895–910, 2013.

[GHK$^+$11] P. A. Golovach, P. Heggernes, D. Kratsch, D. Lokshtanov, D. Meister, and S. Saurabh. Bandwidth on AT-free graphs. *Theor. Comput. Sci.*, 412(50):7001–7008, 2011.

[GHPP13] S. Guillemot, F. Havet, C. Paul, and A. Perez. On the (non-)existence of polynomial kernels for P_l-free edge modification problems. *Algorithmica*, 65(4):900–926, 2013.

[Gil11] D. Gildea. Grammar factorization by tree decomposition. *Computational Linguistics*, 37(1):231–248, 2011.

[GJ79a] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[GJ79b] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of NP-completeness, 1979.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM, 1974.

[GKK$^+$15] E. Ghosh, S. Kolay, M. Kumar, P. Misra, F. Panolan, A. Rai, and M. Ramanujan. Faster parameterized algorithms for deletion to split graphs. *Algorithmica*, 71(4):989–1006, 2015.

[GL81] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite.* Prentice Hall Professional Technical Reference, 1981.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[Gol04] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004.

[Guo07] J. Guo. Problem kernels for NP-complete edge deletion problems: split and related graphs. In *International Symposium on Algorithms and Computation*, pages 915–926. Springer, 2007.

[Gup00] A. Gupta. Improved bandwidth approximation for trees. In *SODA*, pages 788–793, 2000.

[Hag00] T. Hagerup. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica*, 27(3-4):292–315, 2000.

[Her14] T. Hertli. 3-SAT faster and simpler—Unique-SAT bounds for PPSZ hold in general. *SIAM Journal on Computing*, 43(2):718–729, 2014.

[HIS81] P. Hammer, T. Ibaraki, and B. Simeone. Threshold sequences. *SIAM Journal on Algebraic Discrete Methods*, 2(1):39–49, 1981.

[HKM09] P. Heggernes, D. Kratsch, and D. Meister. Bandwidth of bipartite permutation graphs in polynomial time. *J. Discrete Algorithms*, 7(4):533–544, 2009.

[HMM91] J. Haralambides, F. Makedon, and B. Monien. Bandwidth minimization: an approximation algorithm for caterpillars. *Mathematical Systems Theory*, 24(1):169–177, 1991.

[Hoc96] D. S. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

[HS81] P. L. Hammer and B. Simeone. The splittance of a graph. *Combinatorica*, 1(3):275–284, 1981.

[IP01] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62:367–375, 2001.

[IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63:512–530, 2001.

[Jan14] B. M. Jansen. Turing kernelization for finding long paths and cycles in restricted graph classes. In *European Symposium on Algorithms*, pages 579–591. Springer, 2014.

[JB11] B. Jansen and H. Bodlaender. Vertex cover kernelization revisited: Upper and lower bounds for a refined parameter. In *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, volume 9, pages 177–188, 2011.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KKM97] D. Kratsch, T. Kloks, and H. Müller. Measuring the vulnerability for classes of intersection graphs. *Discrete Applied Mathematics*, 77(3):259–270, 1997.

[KL16] M. Kumar and D. Lokshtanov. A $2\ell k$ kernel for ℓ-component order connectivity. *To appear at IPEC 2016*, 2016.

[Klo94] T. Kloks. *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media, 1994.

[KR08] S. Khot and O. Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

[KT06] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[KU12] C. Komusiewicz and J. Uhlmann. Cluster editing with locally bounded modifications. *Discrete Applied Mathematics*, 160(15):2259–2270, 2012.

[KV90] D. J. Kleitman and R. V. Vohra. Computing the bandwidth of interval graphs. *SIAM Journal on Discrete Mathematics*, 3(3):373–375, 1990.

[KvHK02] A. M. Koster, S. P. van Hoesel, and A. W. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3):170–180, 2002.

[KW13] S. Kratsch and M. Wahlström. Two edge modification problems without polynomial kernels. *Discrete Optimization*, 10(3):193–199, 2013.

[Lag96] J. Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. *Journal of Algorithms*, 20(1):20–44, 1996.

[Lee16] E. Lee. Partitioning a graph into small pieces with applications to path transversal. *CoRR*, abs/1607.05122, 2016.

[Lin15] B. Lin. The parameterized complexity of k-Biclique. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 605–615. SIAM, 2015.

[LMS11] D. Lokshtanov, D. Marx, and S. Saurabh. Slightly superexponential parameterized problems. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 760–776. SIAM, 2011.

[LNR$^+$14] D. Lokshtanov, N. Narayanaswamy, V. Raman, M. Ramanujan, and S. Saurabh. Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms (TALG)*, 11(2):15, 2014.

[LPPS14] D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 186–195. IEEE, 2014.

[LPRS16] D. Lokshtanov, F. Panolan, M. Ramanujan, and S. Saurabh. Lossy kernelization. *arXiv preprint arXiv:1604.04111*, 2016.

[LWG14] Y. Liu, J. Wang, and J. Guo. An overview of kernelization algorithms for graph modification problems. *Tsinghua Sci. Technol.*, 19(4):346–357, 2014.

[LWGC12] Y. Liu, J. Wang, J. Guo, and J. Chen. Complexity and parameterized algorithms for cograph editing. *Theor. Comput. Sci.*, 461:45–54, 2012.

[LWY$^+$15] Y. Liu, J. Wang, J. You, J. Chen, and Y. Cao. Edge deletion problems: Branching facilitated by modular decomposition. *Theoretical Computer Science*, 573:63–70, 2015.

[LY80] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.

[Man08] F. Mancini. *Graph modification problems related to graph classes.* PhD thesis, University of Bergen, 2008.

[Mar06] D. Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.

[Mar08a] D. Marx. Closest substring problems with small distances. *SIAM Journal on Computing*, 38(4):1382–1410, 2008.

[Mar08b] D. Marx. Parameterized complexity and approximation algorithms. *Comput. J.*, 51(1):60–78, 2008.

[Moh01] B. Mohar. Face covers and the genus problem for apex graphs. *Journal of Combinatorial Theory, Series B*, 82(1):102–117, 2001.

[Mon85] B. Monien. How to find long paths efficiently. *North-Holland Mathematics Studies*, 109:239–254, 1985.

[Mon86] B. Monien. The bandwidth minimization problem for caterpillars with hair length 3 is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 7(4):505–512, 1986.

[MP95] N. Mahadev and U. Peled. *Threshold graphs and related topics*, volume 56. North Holland, 1995.

[NG13] J. Nastos and Y. Gao. Familial groups in social networks. *Soc. Netw.*, 35(3):439–450, 2013.

[Nie06] R. Niedermeier. Invitation to fixed-parameter algorithms. 2006.

[NSS01] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113(1):109–128, 2001.

[OS06] S.-i. Oum and P. Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96(4):514–528, 2006.

[Oum08] S.-I. Oum. Approximating rank-width and clique-width quickly. *ACM Transactions on Algorithms (TALG)*, 5(1):10, 2008.

[Pap76] C. H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

[PR00] L. Perković and B. Reed. An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, 11(03):365–371, 2000.

[RD94] S. Ray and J. S. Deogun. Computational complexity of weighted integrity. *J. Combin. Math. Combin. Comput.*, 16:65–73, 1994.

[Ree92] B. A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 221–228. ACM, 1992.

[Rob86] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[RPBD12] P. Rinaudo, Y. Ponty, D. Barth, and A. Denise. Tree decomposition and parameterized algorithms for RNA structure-sequence alignment including tertiary interactions and pseudoknots. In *International Workshop on Algorithms in Bioinformatics*, pages 149–164. Springer, 2012.

[RS86] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[RS95] N. Robertson and P. D. Seymour. Graph minors. XIII. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

[RSST97] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. The four-colour theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2–44, 1997.

[Sav98] J. E. Savage. *Models of computation - exploring the power of computing.* Addison-Wesley, 1998.

[Sax80] J. B. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM Journal on Algebraic Discrete Methods*, 1(4):363–369, 1980.

[SB15] D. Schoch and U. Brandes. Stars, neighborhood inclusion, and network centrality. In *SIAM Workshop on Network Science*, 2015.

[Sha02] R. Sharan. *Graph modification problems and their applications to genomic research.* PhD thesis, Tel-Aviv University, 2002.

[Sko03] K. Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *Journal of Algorithms*, 47(1):40–59, 2003.

[SST04] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. *Discrete Appl. Math.*, 144(1-2):173–182, 2004.

[TP97] 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.

[Tra84] B. A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing*, 6(4):384–400, Oct 1984.

[Vaz13] V. V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[WS11] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[Xia16] M. Xiao. Linear kernels for separating a graph into components of bounded size. *CoRR*, abs/1608.05816, 2016.

[Yan81a] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Algebr. Discrete Methods*, 2(1):77–79, 1981.

[Yan81b] M. Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981.

[Yan97] J.-H. Yan. The bandwidth problem in cographs. *Tamsui Oxford Journal of Mathematical Sciences*, (13):31–36, 1997.

[Zeh15] M. Zehavi. Mixing color coding-related techniques. In *Algorithms-ESA 2015*, pages 1037–1049. Springer, 2015.