TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DIPLOMARBEIT

On Tree-Decompositions and their Algorithmic Implications for Bounded-Treewidth Graphs

Ausgeführt am Institut für
Diskrete Mathematik und Geometrie
der Technischen Universität Wien

unter der Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Michael Drmota

durch

Christina Satzinger, B.Sc.
Matrikelnummer: 0925839

Hoheneckerstraße 13
4523 Neuzeug

Datum

Unterschrift

Preface

Graphs naturally arise in many applications, such as network theory and scheduling. In many cases, the occurring problems can be formulated in terms of problems on corresponding abstract graphs, like finding special colourings or other subsets with specific properties. As a consequence, the development of efficient algorithms for various classes of graph problems is of great practical significance.

Unfortunately, a huge class of these graph problems appears to be NP-hard i.e. the problems do not admit general algorithms which are efficient for graphs with increasing size. Since large graph instances emerge in applications anyway, it seems quite natural to search for algorithms which, at least, tackle some common types of graphs.

For instance, many problems occur to be linear- or polynomial-time solvable on trees. Just consider the 2-colouring problem, which can be solved in linear time by traversing the tree in bottom-up order. Similar results can be observed for series-parallel graphs i.e. graphs obtained by series and parallel composition of edges.

The concepts of treewidth and tree-decomposition, which are presented throughout this thesis, serve as a generalization of the above graph classes. We will see that many $\mathcal{NP}$-hard problems allow for polynomial-time or sometimes even linear-time solutions if they get restricted to graphs of bounded treewidth. This is a quite powerful result since many practical graphs have relatively small treewidth!

In Chapter 1, we are going to introduce the necessary graph theoretical notions for treewidth and tree-decompositions. We start Section 1.1 by giving the actual definitions of treewidth and tree-decomposition. We continue by showing the most basic properties of tree-decompositions to give a deeper insight. In Section 1.2, we finally consider graphs of bounded treewidth. We relate this class to other common graph classes e.g. clique sums, chordal graphs and partial k-trees. We close the chapter with some remarks regarding the application in graph minor theory in Section 1.3. This chapter will, hopefully, equip the reader with an adequate perception of the underlying concepts.

Chapter 2 gives some more information for the subsequent part of this thesis, namely utilizing the properties of graphs of bounded treewidth for finding custom-built algorithms. We classify general graph problems and give a short introduction to fixed-parameter tractability in Section 2.1. Subsequently, we define some more notions in Section 2.2 which are of special interest for the later algorithmic considerations: nice tree-decompositions and terminal graphs. This finally concludes the introductory chapter.

In Chapter 3, we consider some problems that can efficiently be solved on graphs with given tree-decompositions. We not only introduce the l-colouring-problem, the k-disjoint-paths-problem, independent sets, dominating sets and vertex covers but give some ideas on how more algorithms of this kind could be found. We start with a very general dynamic programming approach in Section 3.1 and a (more or less detailed) consideration of each of the problems stated above. In Section 3.3, we finally take a look at a very general logical approach by Courcelle et al., which involves monadic second-order logic for graphs and gives some quite powerful results.

We continue with a discussion of the problem of actually determining treewidth and constructing tree-decompositions in Chapter 4. Although this problem is, in general, *NP*-hard, we argue that special sub-problems can be decided in linear or polynomial time. In Subsection 4.2.1, we introduce a polynomial-time algorithm by Arnborg et al. Then we consider a linear-time algorithm by Bodlaender in Subsection 4.2.3, which – despite its inefficiency for practical applications – finally shows that the problem variant is actually of linear complexity. We finish this discussion by briefly mentioning some common approximation methods and other useful bounds in Section 4.3.

We finally mention another interesting technique which is based on graph reductions in Chapter 5. This technique differs from the ones mentioned above by not evaluating an actual tree-decomposition and thus omitting some complexity issues discussed in Chapter 4. We intend to give a short motivation for these types of algorithms without considering all details.

This concludes the discussion of algorithmic methods for bounded-treewidth graphs and it remains to give some final remarks at the very end.

Acknowledgements

I thank my supervisor, Professor Dr. Michael Drmota, for introducing me to this fascinating subject and giving me the opportunity to write this master thesis. Furthermore, I want to thank him for his encouragement and support.

I would also like to thank my colleagues and friends who supported me during my studies. In particular, I would like to thank Clemens Müllner for his constant support and patience.

Finally, I want to thank my family, which encouraged and supported me throughout my entire life and education. Most of all, I thank my parents for always being there for me.

Contents

1 Graph Theoretical Preliminaries

Throughout this thesis, we assume that the reader is familiar with the most basic graph-theoretical notions. For general references, we recommend the book 'Graph Theory' by Reinhard Diestel [1]. There is, furthermore, a very nice compendium for this purpose in Miriam Heinz's thesis [2, Chapter 1] introducing the most basic notions.

At first, we would like to introduce the fundamental concept of tree-decompositions, as introduced by Roberson and Seymour in their work [3] on the graph minor theorem, and state a few of its properties, without claiming to be exhaustive. A lot of information about the structure of tree-decompositions can be found in Bodlaender's work, for instance in [4] and [5]. Some more information focusing on the graph minor theorem can also be found in [1, Chapter 12].

Subsequently, we are going to relate this notion to some other useful concepts. We consider clique-sums and chordal graphs as well as partial k-trees, which are used in papers such as [6]. We hope that this discussion will help getting a good impression of the main concepts.

In this chapter, we mainly follow the structure of Miriam Heinz in her thesis [2, Chapter 3].

1.1 Tree-decompositions and treewidth

In this section, we want to give a formal definition of tree-decompositions and treewidth. Furthermore, we will try to give some intuition for this abstract concept by stating some of its most important properties.

1.1.1 Definitions

We first provide the abstract definition of a tree-decomposition of a graph $G = (V, E)$.

Definition 1.1.1. A *tree-decomposition* of a graph $G = (V, E)$ is a pair $(T, \mathcal{X})$ of a tree T and a family of sets $\mathcal{X} = (X_t)_{t \in V(T)}$ such that $X_t \subseteq V$ and the following properties are satisfied:

(T1) $\bigcup_{t \in V(T)} X_t = V$.

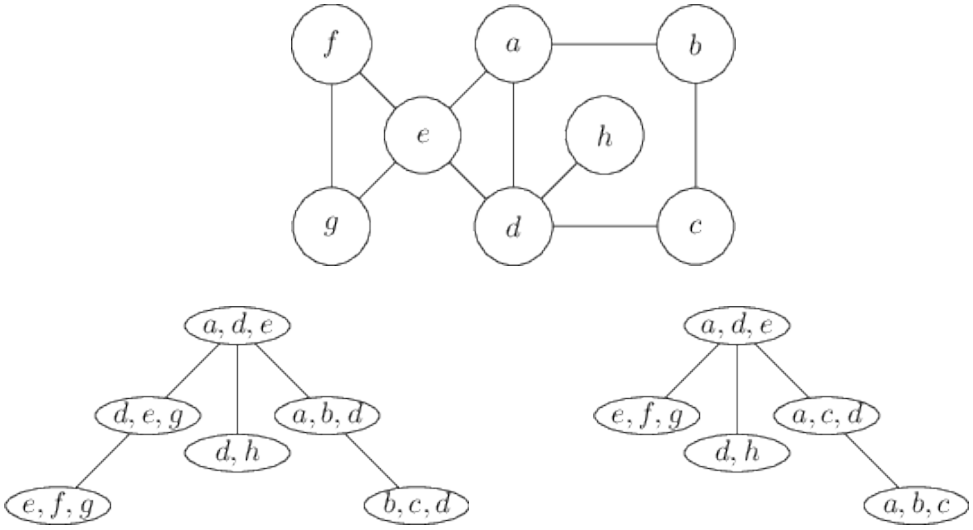

Figure 1.1: A graph G and two possible tree-decompositions

(T2) For all edges $uv \in E$, there exists a node $t \in V(T)$ such that $u \in X_t$ **and** $v \in X_t$.

(T3) Whenever $t_1, t_2, t_3 \in V(T)$ such that t_2 is contained in the unique path from t_1 to t_3 in T, there holds $X_{t_1} \cap X_{t_3} \subseteq X_{t_2}$.

The vertex sets X_t and also the induced subgraphs $G[X_t]$ are called the parts of the tree-decomposition $(T, \mathcal{X})$. To distinguish the vertices of G and T, we use the term *nodes* when talking about the vertices of T.

A tree-decomposition obviously describes a certain grouping of the vertices of G into its parts, where – by Property (T1) - all vertices belong to at least one part and – by Property (T2) – adjacent vertices share at least one part they belong to.

Obviously, there is no need for this construct to be unique! As an example, we provide a graph G and two possible tree-decompositions in Figure 1.1.

Property (T3) assures that any vertex contained in two arbitrary sets X_{t_1} and X_{t_3} of $\mathcal{X}$ is also contained in any further set X_{t_2} for an arbitrary node $t_2 \in V(T)$ that lies along the unique path $t_1 P t_3$ in T. Therefore this property can easily be reformulated to

(T3') For all $v \in V$, the induced subgraph $G[\{t \in T : v \in X_t\}]$ is connected.

This way, a tree-decomposition can also be seen as a union of induced subtrees $G[\{t \in T : v \in X_t\}], v \in V$. This is visualized in Figure 1.2.

We now properly define the concept of treewidth.

Definition 1.1.2. For a fixed tree-decomposition $(T, \mathcal{X})$ of a graph G, its *width* is defined

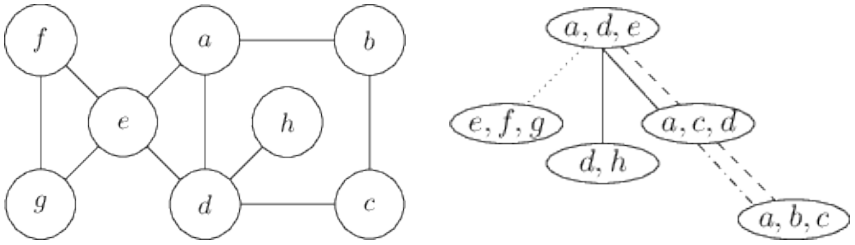

Figure 1.2: A tree-decomposition as a union of subtrees (each colour corresponds to one subtree)

by

$$\max_{t \in V(T)} |X_t| - 1$$

The treewidth $tw(G)$ of a graph G is defined as the minimal width of all tree-decompositions.

We can immediately give a range for the treewidth of a graph.

Lemma 1.1.3. *For each graph G there holds*

$$0 \leq tw(G) \leq |V(G)| - 1$$

Further $tw(G) \geq 1$ if there is at least one edge.

Proof. Every part of a tree-decomposition contains at least one and at most $|V(G)|$ vertices. By definition of treewidth, the first inequality holds. Whenever $|E(G)| > 0$, we can apply Property (T2) to see that there has to be at least one part containing both endpoints of this edge. So by definition, there holds $tw(G) \geq 1$. □

After this first simple considerations, we provide some important subclasses of graphs with bounded treewidth.

Example (trees). As the name 'treewidth' indicates, trees induce quite natural tree-decompositions of width 1. For a tree $T = (V, E)$, we define a tree-decomposition $(T, (X_t)_{t \in V})$ by choosing a root r, defining $X_r = \{r\}$, directing the edges from r and for every directed edge $st \in E$ defining $X_t = \{s, t\}$. This defines all the sets X_t and the above axioms hold. Therefore, we know $tw(T) \leq 1$ and $tw(T) = 1$ for non-trivial trees.

This result can obviously be generalized to forests. For each component tree C, we obtain a tree-decomposition $(T_C, \mathcal{X}_C)$ of width 1. By introducing a new node r and connecting it with each tree T_C, we get a new tree T. Using all the disjoint parts $\mathcal{X}_C$ and $X_r = \emptyset$, we obviously get a valid tree-decomposition of width 1 for forests.

Example (series-parallel graphs). Another subclass of historical importance are series-parallel graphs. Series-parallel graphs are obtained from K_2 by recursive duplication and subdivision of edges. Their name is motivated by the Kirchhoff rules for electrical networks, which allow to evaluate the resistance by a *series* and a *parallel* rule and can thus exactly be applied for series-parallel graphs.

K_2 obviously has treewidth 1 and duplicating edges does not increase the treewidth at all. Subdivision of an edge $uv \in E$ by a vertex w can easily be handled by adding a new part $\{u, v, w\}$ to the tree-decomposition. So series-parallel graphs have treewidth 2.

For further considerations, it is often useful to demand some additional properties. We introduce the following useful concept.

Definition 1.1.4. A tree-decomposition $(T, (X_i)_{i \in V(T)})$ of a graph G is called *reduced*, if it has width $tw(G)$ and

$$X_i \backslash X_j \neq \emptyset \text{ and } X_j \backslash X_i \neq \emptyset$$

for all $ij \in E(T)$.

The following lemma states that these particular demands are admissible i.e. we can still find such a tree-decomposition.

Lemma 1.1.5. *For all graphs G, there exists a reduced tree-decomposition $(T, (X_i)_{i \in V(T)})$ of width $tw(G)$.*

Proof. Choose an arbitrary tree-decomposition $(T, (X_i)_{i \in V(T)})$ of width $tw(G)$ with a minimal number of nodes.

If some $X_i \backslash X_j = \emptyset$, there holds $X_i \subseteq X_j$. We define a new tree T' obtained from T by contracting the edge ij to a single node k with vertex set $X_k = X_j$. Of course, the Properties (T1) and (T2) are not violated for G since the nodes in X_i, X_j are covered by X_k. Note that the subgraphs of T' induced by a single vertex v stay connected since for $v \in X_k$ either $v \in X_i \subseteq X_j = X_k$ or $v \in X_j \backslash X_i$ and j was already a leaf of the tree induced in T. This gives a new tree-decomposition which obviously has the same width but fewer nodes – a contradiction to minimality!

By contradiction, we see that $(T, (X_i)_{i \in V(T)})$ is reduced. □

1.1.2 Subgraphs and minors

From the treewidth of G, we can derive information about the treewidth of arbitrary subgraphs.

Lemma 1.1.6. *For an arbitrary subgraph H of a graph G, there holds $tw(H) \leq tw(G)$.*

Proof. If we take a tree-decomposition $(T, \mathcal{X})$ of G, the tree-decomposition $(T, \mathcal{X}')$ defined by the parts

$$X'_t = X_t \cap V(H)$$

is a valid tree-decomposition for H. All three axioms are passed on from G. □

We would like to mention that a similar result can be obtained for the treewidth of minors.

Lemma 1.1.7. *If H is a minor of a graph G, then there holds $tw(H) \leq tw(G)$.*

This result is only slightly more challenging and very important in graph minor theory. For reference, one can consider Miriam Heinz' thesis [2] or Diestel's book [1, Chapter 12].

Another property that can easily be observed is the following.

Lemma 1.1.8. *For the connected components $C_1, \ldots, C_n$ of a graph G, there holds $tw(G) = \max\{tw(C_i) : i = 1, \ldots, n\}$.*

Proof. Since all components are subgraphs of G, there holds $\max_{i=1}^{n} tw(C_i) \leq tw(G)$ by Lemma 1.1.6. From the tree-decompositions $(T^i, \mathcal{X}^i)$ of C_i, we can derive a tree-decomposition for G by gluing the trees T_i together at a common root node r with $X_r = \emptyset$. All the other parts are passed on from the components decompositions. Trivially, all axioms do hold because the components are not connected and use disjoint vertex labels. □

We can, therefore, always restrict to connected graphs when examining properties based on treewidth and tree-decompositions. For disconnected graphs, we simply consider each component separately.

1.1.3 Connectivity and separation properties

Tree-decompositions contain information about the connectivity properties of G. We now consider how tree-decompositions reflect the connectivity of G at a fixed node $t \in V(T)$.

Throughout this subsection, let $(T, (X_t)_{t \in V(T)})$ be a fixed tree-decomposition of G.

Definition 1.1.9. For any node $t \in V(T)$, the components of $T - t$ are called the *branches* of T at t.

Lemma 1.1.10. *Let $t \in V(T)$ be an arbitrary node. For each vertex $v \in V$, either $v \in X_t$ or v is contained only in parts of exactly one branch of T at t. This branch is denoted by $T_t(v)$.*

Proof. For $v \in X_t$, we are done. If $v \notin X_t$, it has to be contained in another part by Condition(T1). If it was contained in some vertex sets X_{t_1}, X_{t_2} of different branches, t would lie on the unique path from t_1 to t_2 in T and by Property (T3) we could conclude that $v \in X_t$. This is a contradiction, so there is just one such part. □

Lemma 1.1.11. *There holds:*

- *If $uv \in E$ and u and v are not in X_t, then $T_t(u) = T_t(v)$.*
- *More generally, if $u, v \in V$ are neither elements of X_t nor separated in G by X_t, then $T_t(u) = T_t(v)$.*

Proof. Since Property (T2) has to hold, we know that for each edge $uv \in E$ there exists a part $X_{t'}, t' \neq t$ containing both u and v. The branch containing the node t' contains both u and v in its part $X_{t'}$ and by uniqueness $T_t(u) = T_t(v)$. The second statement follows from the fact that we can iteratively apply the first statement for a path from u to v which does not meet X_t. □

This already suffices to see that it is not possible to find paths between vertices of different branches which do not meet X_t. More precisely, we can derive the following corollary from Lemma 1.1.10 and Lemma 1.1.11.

Corollary 1.1.12. *For an arbitrary node $t \in V(T)$ and branches $T_1, \dots, T_n$ of T at t, consider the subgraphs*

$$G_i := G\left[\bigcup_{l \in V(T_i)} X_l\right]$$

for $1 \leq i \leq n$. The subgraphs

$$G_1 - X_t, G_2 - X_t, \dots, G_n - X_t$$

neither have a vertex in common nor edges between them.

This means that the parts of the tree-decomposition are indeed somehow structured like a tree. We can even transfer some separation properties.

Lemma 1.1.13. *For any edge $t_1t_2 \in E(T)$, let T_1, T_2 be the components of t_1 and t_2 in the graph $T - t_1t_2$ and define the corresponding vertex sets*

$$U_1 := \bigcup_{t \in T_1} X_t, \text{ and } U_2 := \bigcup_{t \in T_2} X_t$$

There is no path from U_1 to U_2 that does not use vertices from $X_{t_1} \cap X_{t_2}$. If the tree-decomposition is reduced in the sense of Definition 1.1.4, the set $X_{t_1} \cap X_{t_2}$ separates U_1 and U_2 in G.

Proof. Since t_1 and t_2 are on each T_1T_2-path in T, Property (T3) implies that $U_1 \cap U_2 \subseteq X_{t_1} \cap X_{t_2}$. Therefore, any possible U_1U_2-path not using $X_{t_1} \cap X_{t_2}$ cannot use inner vertices from $U_1 \cap U_2$. If we assume the existence of such a path, it has to contain a direct edge u_1u_2 with $u_1 \in U_1 \backslash U_2$ and $u_2 \in U_2 \backslash U_1$ because $U_1 \cup U_2 = V$. Property (T2) yields a common part X_t with $u_1, u_2 \in X_t$.

But the corresponding node t is neither allowed to be in T_1 nor T_2 by the choice of u_1 and u_2, which yields a contradiction to the existence of t.

If the tree-decomposition is reduced, there are vertices $v_1 \in X_{t_1} \subseteq U_1, v_2 \in X_{t_2} \subseteq U_2$ which are not in $X_{t_1} \cap X_{t_2}$. Those are, of course, not connected anymore if we remove $X_{t_1} \cap X_{t_2}$ and, therefore, $X_{t_1} \cap X_{t_2}$ is a separator. □

Lemma 1.1.14. *Given a set $W \subseteq V$, there exists either $t \in T$ with $W \subseteq X_t$ or there exist $w_1, w_2 \in W$ and an edge $t_1t_2 \in E(T)$ such that $w_1, w_2 \notin X_{t_1} \cap X_{t_2}$ and they are separated by $X_{t_1} \cap X_{t_2}$ in G.*

Proof. Suppose there are no w_1, w_2, t_1, t_2 as desired, then for each edge t_1t_2 in T one of the sets U_1, U_2 (as defined in Lemma 1.1.13) contains W. We orient the edge t_1t_2 towards t_i with $W \subseteq U_i$. This defines an acyclic orientation of the edges of T and there is a maximal directed path ending at some node t.

Each $w \in W$ is contained in some part $X_{t'}$. If $t \neq t'$ we consider the edge e incident to t along the unique tt'-path. This edge has to be directed towards t because of the maximality and thus another node of the component containing t in $T - e$ has to contain w by definition of the orientation. Property (T3) implies $w \in X_t$ for arbitrary $w \in W$ i.e. $W \subseteq X_t$. □

We use the previous result to show that the treewidth does depend on the clique number of G.

Lemma 1.1.15. *The vertex set of any complete subgraph of G is contained in some part of a tree-decomposition $(T, \mathcal{X})$ of G. Thus there holds*

$$\omega(G) - 1 \leq tw(G)$$

Proof. Let W be a clique in G. Since all elements of W are pairwise adjacent, the second condition in Lemma 1.1.14 can not be valid. So we get $W \subseteq X_t$ for some node t.

Obviously $|W| \leq |X_t|$ for all complete subgraphs and all tree-decompositions. Therefore, the inequality follows directly. □

1.2 Graphs of treewidth at most k

In this section, we will consider graphs of bounded treewidth. Using some related concepts, we will give some kind of description for this very interesting subclass of graphs.

1.2.1 Clique sums

We noticed in the last section that the treewidth of G does depend on the graphs connectivity. We observed that for all edges tt' of T with $U_i \not\subseteq X_t \cap X_{t'}, i = 1, 2$ the set $X_t \cap X_{t'}$ separates the vertices of the two different components of $T - tt'$ due to Lemma 1.1.13. This may motivate the following considerations.

We consider graphs constructed using *k-cliques* i.e. subsets of k vertices of a graph such that their induced subgraph is isomorphic to K_k.

Definition 1.2.1. A *k-clique sum* of two graphs G_1 and G_2 containing k-cliques C_1 and C_2 respectively is the graph obtained by pairwise identifying the vertices (and edges) of C_1 and C_2, while possibly deleting some of the edges between these k new vertices. We say that this new graph is obtained by pasting G_1 and G_2 together along C_1 and C_2.

This process can be visualized by gluing two graphs together along the fixed k-cliques. One can show the following lemma

Lemma 1.2.2. *If the graphs G_1 and G_2 both have treewidth at most k, then so does any k-clique sum of G_1 and G_2.*

Proof. Let G be a k-clique sum obtained by pasting along k-cliques $C_1 \leq G_1$ and $C_2 \leq G_2$ without deleting edges. From the tree-decompositions $(T_1, \mathcal{X}_1)$ and $(T_2, \mathcal{X}_2)$ of G_1 and G_2, we construct a tree-decomposition $(T, \mathcal{X})$ for their clique sum.

Let $t_i \in V(T_i), i = 1, 2$ be the node such that $C_i \subseteq X_{t_i}$ according to Lemma 1.1.15. Now choose T to be the tree obtained by connecting the trees T_1 and T_2 with a new edge t_1t_2. Apart from identifying the nodes from C_1 and C_2 in the parts $\mathcal{X}_1$ and $\mathcal{X}_2$, we keep the labels.

It is easy to see that the Conditions (T1) and (T2) hold. Since only labels of the clique are shared and they are all contained in both nodes t_1 and t_2, Property (T3) holds too. Of course, the treewidth has not increased at all.

It remains to notice that we can use this tree-decomposition too when deleting possible edges – this just weakens Property (T2)! □

Using this fact we can derive the following result about the structure of graphs of fixed treewidth.

Lemma 1.2.3. *Let G be a graph of treewidth k. Then G can be obtained by recursively pasting together graphs of cardinality at most $k + 1$.*

Proof. We consider a tree-decomposition of G of width k. Every part $X_t, t \in V(T)$ consists of at most $k + 1$ vertices – these essentially give the component graphs $G_t = G[X_t], t \in T$ we are going to paste together!

For every edge tt' of T, we iteratively perform a clique sum of the (sub-)graphs G_t and $G_{t'}$ along $X_t \cap X_{t'}$. Therefore, we just add the mandatory edges to the component graphs G_i in the first place and do not remove them until the final clique sum is performed that involves a particular edge. Obviously the graph obtained in this process is G. □

We immediately derive another useful result.

Corollary 1.2.4. *Let G be a graph of treewidth k with at least $k+2$ vertices. Then G has a separator of size at most k i.e. a subset $S \subseteq V$ of size at most k such that $G - S$ is disconnected.*

Proof. Since G has at least $k+2$ vertices, it is obtained by gluing together at least two graphs G_1 and G_2 of cardinality at most $k+1$ along a clique C. We can assume that the graphs $G_i - C$ are non-empty because otherwise the pasting would not add anything and could be omitted anyway. By definition, C is a separator for G and, of course, its cardinality is at most k. □

1.2.2 Chordal graphs

At this point, we would like to mention *chordal graphs*. These graphs are not only an example for graphs with a very special structured tree-decompositions but they also help understanding treewidth for arbitrary graphs a little better.

Definition 1.2.5. A graph G is called *chordal* if every cycle C in G of length at least four contains a *chord*, i.e. an edge between two vertices of C that are not yet adjacent along C.

Remark. Every induced subgraph $G[W], W \subseteq V$ forming a cycle is therefore a triangle. Therefore, chordal graphs are sometimes called *triangulated graphs*.

Example. A obvious example for chordal graphs are complete graphs. By definition, all sets of three vertices induce a triangle.

There are some alternative characterizations that are sometimes quite useful. Especially, we would like to mention the following.

Definition 1.2.6. A vertex v of G is called *simplicial* if the neighbourhood $N_G(v)$ induces a complete subgraph of G.

Definition 1.2.7. A *perfect elimination ordering* is an ordering of the vertices $v_1, \ldots, v_n$ of a graph $G = (V, E)$ such that for $i = 1, \ldots, n$ holds

> The vertex v_i is simplicial in the induced subgraph $G[V \backslash \{v_1, \ldots, v_{i-1}\}]$.

A way to visualize this is that, when iteratively removing the vertices $v_1, \ldots, v_n$ from our graph G, they are always mutually adjacent. Graphs having a perfect elimination ordering are sometimes called *perfect elimination graphs*.

Theorem 1.2.8. *Given a graph $G = (V, E)$, the following statements are equivalent:*

1. *G is chordal.*
2. *G is a perfect elimination graph.*
3. *For each pair of vertices $v, w \in V$ every minimal vw-separator in G is a complete subgraph of G.*

We are not going to present a proof here since we are not going to use this result anyway. Nevertheless, the equivalence of chordal graphs and perfect elimination graphs will help linking this subsection to the following. A proof of Theorem 1.2.8 can, for instance, be found in [7].

We now show the following.

Lemma 1.2.9. *A graph $G = (V, E)$ is chordal if and only if it can be constructed recursively by pasting together complete graphs along complete subgraphs.*

Proof by Diestel [1]. It is quite easy to see, that graphs G obtained from chordal graphs G_1, G_2 by pasting along complete subgraphs are chordal: the intersection is complete, so any induced cycle in G either is completely contained in G_1 or in G_2 and thus a triangle by assumption. Therefore, the property of being chordal is recursively inherited from the original complete graphs.

Conversely, if G is chordal, we perform an induction on $|V|$. If G is complete, we just have one trivial component; otherwise we find two non-adjacent vertices $a, b \in G$ and take a minimum-size separator $X \subseteq V \backslash \{a, b\}$ of a and b i.e. every ab-path in G contains a vertex contained in X. The component of $G - X$ containing a is denoted by C and we define the subgraphs $G_1 = G[V(C) \cup X]$ and $G_2 = G - C$.

G arises from G_1 and G_2 by pasting along $G[X]$, where G_1 and G_2 are chordal (as induced subgraphs of the chordal graph G) and , by induction, both recursively constructed by pasting along complete graphs along complete subgraphs.

It remains to argue that the pasting along $G[X]$ is also along a complete subgraph. Suppose there are $s, t \in X$ which are non-adjacent. Since X is minimal, both s and t have a neighbour in the component C of a and we find a st-path with vertices in G_1. We denote the shortest such path by P_1 and, analogously, we find a shortest st-path P_2 in G_2. We now observe that P_1P_2 is a cycle in G which has no chord because the paths are minimal and s, t non-adjacent. This is a contradiction to the fact that G is chordal and, thus, $G[X]$ has to be complete.

So G is constructed by pasting together complete graphs along complete subgraphs too. □

We already mentioned at the beginning of this subsection that chordal graphs have a very nice structured tree-decomposition.

Corollary 1.2.10. *A graph $G = (V, E)$ is chordal if and only if there exists a tree-decomposition of G into complete parts i.e. every part induces a complete subgraph in G.*

Proof. If there is a tree-decomposition $(T, (X_t)_{t \in V(T)})$ with complete parts, then G can be obtained by pasting together the complete parts along the sets $X_{t_1} \cap X_{t_2}, t_1t_2 \in T$ as we did in Lemma 1.2.3: We apply induction on $|V|$. If there is only one part, we are complete and thus chordal; otherwise we choose a reduced tree-decomposition, take an edge $t_1t_2 \in T$ and split up G into the subgraphs G_1, G_2 induced by the components of t_1 and t_2 in T as we did in Lemma 1.1.13. Since we chose a reduced tree-decomposition, the graphs G_i are smaller than G and, thus, they are chordal by induction hypothesis. G is obtained from chordal graphs G_1, G_2 by pasting along $G_1 \cap G_2 \subseteq X_{t_1} \cap X_{t_2}$, which is complete because X_{t_1} and X_{t_2} are. Therefore, G is chordal by Lemma 1.2.9.

Conversely, if G is chordal, Lemma 1.2.9 gives a pasting structure for G. We can recursively define a tree-decomposition by starting with the trivial decompositions for the complete subgraphs and – whenever we paste together two graphs G_1, G_2 with tree-decompositions $(T_1, \mathcal{X}^1)$ and $(T_2, \mathcal{X}^2)$ along a complete subgraph – we define a valid tree-decomposition for the resulting graph. The complete subgraphs C_1, C_2 we paste along, are entirely contained in some parts X_s^1 and X_t^2 by Lemma 1.1.15. We just keep the vertex sets and define the tree T for G by connecting T_1, T_2 by adding an edge st. It is quite easy to see that this indeed gives a tree-decomposition for the original graph. □

We can use this result to compute or at least bound the treewidth of arbitrary graphs.

Corollary 1.2.11. *For a graph $G = (V, E)$ there holds*

$$tw(G) = \min\{\omega(H) - 1 | G \subseteq H; H \text{ chordal } \}$$

Proof. For each chordal supergraph H, we know that $tw(H) = \omega(H) - 1$ as immediate consequence of Corollary 1.2.10. Combining this with Lemma 1.1.6 we already know $tw(G) \leq tw(H) = \omega(H) - 1$.

Take a tree-decomposition $(T, \mathcal{X})$ of G with treewidth $tw(G)$ and consider the subgraphs G obtained by adding edges uv to G for all u, v that occur in a common part X_t. Considering $(T, \mathcal{X})$ as a tree-decomposition of H, we see that H has complete parts and treewidth at most $tw(G)$. By Corollary 1.2.10, H is chordal and $tw(H) = \omega(H) - 1$. So there holds $\omega(H) - 1 = tw(H) \leq tw(G)$ and we are done. □

As shown in Corollary 1.2.11, we can thus envision graphs with treewidth at most k as subgraphs of chordal graphs with clique number at most $k + 1$.

Corollary 1.2.12 (Folklore)**.** *G has treewidth at most k if and only if G is the subgraph of a chordal graph with maximum clique size at most $k + 1$.*

Or, in other words

Corollary 1.2.13 (Folklore)**.** *G has treewidth at most k if and only if G has a minimal triangulation with maximum clique size at most k.*

1.2.3 Partial k-trees

Another concept which is often used for algorithmic purposes are partial k-trees. For instance Arnborg and Proskurowski used the concept of partial k-trees in [6] to build linear-time algorithms for NP-complete problems on graphs. We will see this notion is equivalent to the concept of graphs of treewidth at most k.

Partial k-trees are subgraphs of so called k-trees.

Definition 1.2.14. k-trees are defined in a recursive manner:

- The complete graph K_k is a k-tree.
- Every graph obtained from a k-tree G by adding a new vertex v and edges from v to k mutually adjacent vertices of G is a k-tree.

Example. Obviously k-trees can be seen as a generalization of trees, which are 1-trees.

The recursive description immediately shows that k-trees have a *perfect elimination order* and thus are a special case of chordal trees, as mentioned in the previous subsection.

We now give an alternative proof for the equivalence of partial k-trees and graphs of treewidth at most k.

Theorem 1.2.15. *A graph $G = (V, E)$ is a partial k-tree if and only if G has treewidth at most k.*

Proof. Let G be a partial k-tree i.e. a subtree of a k-tree G'. We apply induction on $n := |V(G')|$. If $n \leq k+1$, we just have a complete graph and thus treewidth $tw(G') \leq k$. Otherwise there is a node v of degree k whose neighbourhood forms a k-clique. The graph $G' - v$ has treewidth at most k by the induction hypothesis and by Lemma 1.1.15 there is a part X_t containing $N(v)$. Expanding $(T, \mathcal{X})$ by adding a new node t' adjacent to t in T and defining $X_{t'} = N(v) \cup v$, we derive a tree-decomposition for G' of width at most k. Obviously $tw(G) \leq tw(G') \leq k$ holds.

For the second implication, let $(T, \mathcal{X})$ be a reduced tree-decomposition of width at most k for G. We perform induction on $n := |V|$. If $n \leq k + 1$ we obviously have a partial k-tree; otherwise we examine a leaf t of T and denote its unique neighbour in T by t'. Since we are reduced, there exists a vertex $v \in X_t \backslash X_{t'}$, which is – by definition – just contained in this one part X_t and, thus, has less than $|X_t| - 1$ neighbours. Now $G - v$ is a partial k-tree by induction. Adding v and less than $|X_t| - 1 \leq k$ edges preserves this property. Hence G is a partial k-tree. □

So these concepts are indeed equivalent and we have found an alternative, and possibly more intuitive, characterization of the concept of graphs with bounded treewidth. By the above construction, we can even obtain a bound for the number of nodes.

Corollary 1.2.16. *If $G = (V, E)$ has treewidth at most k, there exists a tree-decomposition $(T, \mathcal{X})$ with width at most k and $|V(T)| \leq |V| - k$.*

Furthermore, we can, for instance, observe the following.

Corollary 1.2.17. *Every graph of treewidth at most k contains a vertex of degree at most k.*

Proof. If each vertex had degree at least $k+1$, we obviously cannot find a perfect elimination ordering. □

Corollary 1.2.18. *Every graph of treewidth at most k has chromatic number at most $k+1$ i.e. its vertices can be coloured with $k+1$ colours such that adjacent vertices are in different colour classes.*

Proof. This is obviously true for graphs with just one vertex. The above statement follows by induction: we take away the node v of degree at most k, colour the remaining graph $G\backslash\{v\}$ of treewidth $\leq k$ and there is at least one unused colour in $N(v)$ we can use for v. □

1.2.4 Elimination orderings

During the last sections, we noticed that k-trees do have a perfect elimination ordering. We now want to find a similar result for partial k-trees and graphs of bounded treewidth.

Definition 1.2.19. An *elimination ordering* is a permutation $(v_i)_{i=1}^{|V|}$ of the vertices of a graph G. This corresponds to an elimination process for the graph defined by defining $G_0 := G$ and iteratively constructing new graphs G_i by:

1. removing the vertex v_i from G_{i-1}.
2. and making the neighbouring vertices $N_{G_{i-1}}(v_i)$ mutually adjacent.

Example. A perfect elimination ordering is just a special case of the above definition, where the second step of adding missing edges can be omitted: For perfect elimination orderings, by definition, the vertex v_i is always simplicial and its neighbours are thus already mutually adjacent!

Remark. As a matter of fact, the above definition of an elimination ordering even has some practical background. Consider a symmetric matrix $M = (m_{ij})_{i,j=1}^{n}$ and the corresponding graph $G_M = (V, E)$ with vertices $v_1, \ldots, v_n$ with $m_{ij} \neq 0$ if and only if the

corresponding vertices v_i and v_j are adjacent in G_M. When we perform Gauss elimination for the i-th row of such a symmetric matrix we get new elements

$$m_{jk} = m_{jk} - m_{ik} \cdot \frac{m_{ji}}{m_{ii}} = m_{jk} - m_{ik} \cdot \frac{m_{ij}}{m_{ii}}$$
$$m_{kj} = m_{kj} - m_{ij} \cdot \frac{m_{ki}}{m_{ii}} = m_{jk} - m_{ij} \cdot \frac{m_{ik}}{m_{ii}}$$

One can see that an element does not change if either m_{ij} or m_{ik} is zero. On the other hand, an element becomes non-zero if and only if m_{ij} or m_{ik} are non-zero. Translated into our graph, this means that we add an edge v_jv_k if the edges v_iv_j and v_iv_k had previously been present. All the entries m_{ik} and m_{ji} vanish, which means that the edges v_iv_j and v_iv_k are deleted.

We can also find some measure for such an elimination ordering.

Definition 1.2.20. The *width* of an elimination ordering $(v_i)_{i=1}^n$ of a graph $G = (V, E)$ is defined by

$$\max_{i=1,\dots,n} \left|N_{G_{i-1}}(v_i)\right|$$

i.e. it is the maximal number of neighbours a vertex v_i has at the time of its removal.

We now show:

Lemma 1.2.21. *G is a partial k-tree if and only if it has an elimination ordering of width at most k.*

Proof. For a partial k-tree G, we can find a supergraph H which is a k-tree. H has a perfect elimination ordering as stated before. We now follow this perfect elimination ordering as an elimination ordering of G. G is a subgraph of H, so a vertex can never have more than k neighbours at its time of removal and the width is at most k.

If, on the contrary, G has such an elimination ordering of width at most k, we can reconstruct the graph G by adding vertices in the inverse order of this elimination ordering and connecting them to the (previously added) neighbours $N_{G_{i-1}}(v_i)$ which are a $|N_{G_{i-1}}(v_i)|$-clique with $|N_{G_{i-1}}(v_i)| \leq k$. The result G of this construction is a partial k-tree because it is a k-tree with some removed edges . □

Using Theorem 1.2.15 we see

Corollary 1.2.22. *G is of treewidth at most k if and only if it has an elimination ordering of width at most k.*

So there is some kind of natural ordering for the vertices of partial k-trees and simultaneously graphs of treewidth at most k.

1.2.5 Brambles

In graph minor theory, another concept is very useful, which provides an upper bound for the treewidth. It is mentioned in [4] and – together with some more application – in Miriam Heinz's thesis [2, Chapter 4]. For more reference, see also [1, Chapter 11].

Definition 1.2.23. Two sets $W_1, W_2 \in V$ *touch* if they either intersect or a vertex in W_1 is adjacent to a vertex in W_2. A *bramble* $\mathcal{B}$ is a collection of mutually touching subsets of V. The *order* of a bramble $\mathcal{B}$ is the cardinality of a minimal set W intersecting all sets $B \in \mathcal{B}$.

The existence of a bramble of large order gives another equivalent concept to our bounded-treewidth graphs

Theorem 1.2.24 (Seymour and Thomas)**.** *A graph G has treewidth at least k if it has a bramble of order at least $k+1$.*

An immediate consequence is that the order of any bramble is an upper bound for the treewidth of G. The proof of Theorem 1.2.24 is not covered here but can be found in [2, Chapter 4] or [1, Chapter 11]. Anyway, it is nice to know that there exists some upper bounds too since previously we only had lower bounds as in Theorem 1.2.13.

1.3 Graph minor theory

As we already mentioned, the concepts of treewidth and tree-decompositions were originally introduced by Robertson and Seymour in [3] in the field of graph minor theory.

It is used to show the famous graph minor theorem:

Theorem 1.3.1 (Robertson and Seymour, 2004)**.** *The class of finite graphs is well-quasi-ordered by the minor relation i.e. the minor-relation induces a reflexive and transitive order, in which every infinite sequence $s_0, s_1, \ldots$ of elements contains elements s_i, s_j with $i < j$ and $s_i \leq s_j$.*

An essential step in their proof is to show that, for fixed k, graphs of treewidth less than k are well-quasi-ordered.

The graph minor theorem can also be interpreted in the following sense.

Definition 1.3.2. A graph G is a *excluded minor* of a class $\mathcal{G}$ of graphs, if no graph in $\mathcal{G}$ contains G as a minor.

Corollary 1.3.3. *Every minor-closed class of graphs can be described by a finite set of (pairwise incomparable) excluded minors.*

A famous example of this kind is the theorem of Kuratowski: The class of planar graphs, which is obviously minor-closed, is characterized by the excluded minors K_5 and $K_{3,3}$.

Note that this result also has some nice algorithmic implications. One could possibly try to use this set of excluded minors to test membership for the original class of graphs. Such problems were considered by Robertson and Seymour in their subsequent work [8].

We refer to [2] for a more detailed discussion of the usage of tree-decompositions in graph minor theory.

We, hereby, finish this mainly theoretical discussion of tree-decompositions and tree-width. At the end of Chapter 2, we will find tree-decompositions again but in a more applicable way.

2 Introduction

This chapter gives a general introduction to the more algorithmic part of this thesis.

We introduce the scope of this thesis by giving a brief overview of the graph theoretical problems we are going to tackle later on. Afterwards, we introduce the concept of fixed-parameter tractability in Section 2.1.

Subsequently, in Section 2.2, we introduce the important concepts of nice tree-decompositions and terminal graphs, which are of special interest for our considerations in later chapters.

2.1 General settings

2.1.1 Graph problems

In this subsection, we give a first impression of the kind of graph theoretical problems we are going to deal with throughout this chapter. These definitions are in particular important for later generalizations but the author decided to move them here to give a very first impression what this thesis is about. The definitions are taken from [9].

A basic formal notion in this context are *graph properties*.

Definition 2.1.1. A *graph property* is a function $\mathcal{P}$, which maps each graph (of a given class) to a boolean value i.e. *true* or *false* and does not depend on the actual representation/drawing i.e. isomorphic graphs are mapped to the same value.

We say that the property $\mathcal{P}$ holds for a graph G, if $\mathcal{P}(G) = \mathit{true}$.

Each graph property $\mathcal{P}$ induces a *decision problem*:

Given a graph G, does $\mathcal{P}$ hold for G?

We say that an algorithm decides a graph property $\mathcal{P}$ if it solves the corresponding decision problem.

An example for such a problem is the l-colourability decision problem i.e. deciding, whether a given graph G admits a function $f : V(G) \to \{1, 2, \ldots, l\}$ assigning one of l colours to each vertex, such that adjacent vertices obtain different colours. This problem is considered in Section 3.1.

Many graph properties of graphs are so called *construction properties*.

Definition 2.1.2. A *construction property* is a graph property $\mathcal{P}$ defined by a pair $(\mathcal{D}, \mathcal{Q})$ in the following way

$$\mathcal{P}(G) = \text{“there is an } S \in \mathcal{D}(G) \text{ with } \mathcal{Q}(G, S) = \mathit{true} \text{ ”}$$

where $\mathcal{D}$ is a function mapping a graph G to a corresponding *solution domain* i.e. a set $\mathcal{D}(G)$ depending on G and $\mathcal{Q}$ is an *extended* graph property for G and S i.e. a function mapping the pairs $(G, S), S \in \mathcal{D}(G)$ to boolean values.

An element $S \in \mathcal{D}(G)$ with $\mathcal{Q}(G, S) = \mathit{true}$ is called a solution for G.

The induced construction problem $\mathcal{P}$ is the problem of not only deciding $\mathcal{P}$ for a graph G but additionally finding a solution $S \in \mathcal{D}(G)$ if $\mathcal{P}(G)$ holds.

For the l-colourability construction problem, we could choose the sets

$$\mathcal{D}(G) = \{f : V(G) \to \{1, 2, \ldots, l\}\}$$

and let $\mathcal{Q}$ denote whether for fixed $f \in \mathcal{D}(G)$ the function f corresponds to a valid colouring for G.

Sometimes we also consider *optimization problems*. In contrast to graph properties, optimization problems implicitly contain some kind of valuation function for the solutions.

Definition 2.1.3. An *optimization property* is a function Φ defined on graphs by a quadruple $(\mathcal{D}, \mathcal{Q}, z, \mathit{opt})$ in the following way:

$$\Phi(G) = \mathit{opt}\{z(S) : S \in \mathcal{D}(G) \wedge \mathcal{Q}(G, S)\}$$

where $\mathit{opt} \in \{\min, \max\}$, $\mathcal{D}$ is a function mapping a graph G to a corresponding *solution domain* $\mathcal{D}(G)$ depending on G, $\mathcal{Q}$ is an *extended* graph property for G and S and z is a function mapping graphs $S \in \mathcal{D}(G)$ with $\mathcal{Q}(G, S) = \mathit{true}$ to $\mathbb{Z}$. The function Φ, therefore, takes values in $\mathbb{Z} \cup \{\mathit{false}\}$, where the value *false* is obtained in case of optimization over an empty set.

An element $S \in \mathcal{D}(G)$ with $\mathcal{Q}(G, S)$ and optimal value $z(S)$ is called a *solution* for Φ.

An optimization property corresponds to the *optimization problem* of evaluating the value Φ for a graph G. One can also consider the induced *constructive optimization problem* of not only finding the value $\Phi(G)$ but also a witness $S \in \mathcal{D}(G)$ where this value is actually obtained.

Of course, problems like finding the size of a maximum independent set of vertices are optimization problems and the problem of constructing a maximum-sized independent set is a constructive optimization problem.

These types of problems already cover all kinds of problems that we consider throughout this chapter.

2.1.2 Complexity and fixed-parameter tractability

This subsection is dedicated to the topic of complexity. We assume that the reader is familiar with the basic notions of complexity but we want to give a few additional remarks.

In basic complexity theory and analysis of algorithms, we consider time- and space-complexity of algorithms. Usually the runtime or used storage of an algorithm is given in $O/\Omega/\Theta$-notation for some given input size n. For instance, a time complexity (or runtime) $\mathcal{O}(f(n))$ tells us that the number of basic operations performed during the algorithm is bounded from above by $C \cdot f(n)$ for all inputs of lengths $n < n_0$ with some constants $n_0 \geq 0, C > 0$. We often say that such an algorithm takes $\mathcal{O}(f(n))$ time. Similarly the Ω-notation is used for an lower bound function and the Θ-notation expresses that the given function gives both a lower and an upper bound for different constants C_1 and C_2.

Of course, different algorithms for the very same problem possibly have different time complexity and usually we only bother about the most efficient one. Thus we also speak of complexity of a problem i.e. the complexity of the best (known) algorithm for a given problem. Due to these concepts, one can now classify the *hardness* of problems. For time complexity, the most common classes are:

- The class P: This is the class of all problems that can be solved by algorithms taking $\mathcal{O}(n^c)$ time, for some $c \geq 0$, on a deterministic Turing machine i.e. just by giving a sequence of basic computation steps. Such problems, like searching or sorting, are usually considered as efficiently solvable.
- The class *EXPTIME*: A much larger class of problems can, of course, be solved by algorithms taking exponential time (still on a deterministic Turing machine) in the input length e.g. traversing all subsets. Since the number of computation steps grows exponentially, these problems are in practice only solvable for very small input length.
- The class *NP*: This class lies somewhere in between P and *EXPTIME* and contains all problems that can be solved on a non-deterministic Turing machines in a polynomial number of computation steps. On non-deterministic Turing machines, it is possible to allow for even more complex computation steps e.g. trying many different approaches in parallel and waiting for at least one to work out. Therefore, it is generally believed that the class is strictly wider than P.

  Many graph theoretical problems, like colouring or finding maximum matchings, are *NP*-complete i.e. they are in *NP* and as such not easier than any other problem in *NP*. If one assumes that $P \neq NP$, those problems are not in P and they symptomatically lack in suitable algorithms working for larger input size.

NP-hard problems, i.e. problems which are *NP*-complete or even harder, like *EXPTIME*, still appear quite often in various applications. There is one very useful attempt to deal

with such *NP*-hard problems and it is called *parametrized complexity* [10].

In parametrized complexity problem variants are studied where additional parameters are fixed. Choosing these additional parameters is non-trivial but some might prove useful. The goal is to find a parameter such that the parametrized variant is *fixed-parameter tractable* i.e. all problems of input size $< n$ and with parameter $< k$ take $\mathcal{O}(f(k)n^c)$ time for a suitable function f and a constant c. Note that the dependence on k might even be exponential but the dependence on the input length n is polynomial!

For our graph problems, we are going to see that the treewidth $tw(G)$ works pretty well as additional parameter. Throughout this thesis, we will explore quite a few examples illustrating this fact. Generally the variable n is going to denote the number of graph vertices $|V|$, while we use the variable k for denoting our parameter: the graphs treewidth.

2.1.3 A common structure for algorithms on graphs of bounded treewidth

A very common scheme for fixed-parameter tractable graph problems exploiting tree-decompositions is the following:

1. First we compute a tree-decomposition of the input graph $G = (V, E)$.

   Although determining the treewidth of a graph is - in general - *NP*-complete as we will see in Chapter 4, some certain problem variants are fixed-parameter tractable.

   It is even possible, for fixed k, to find a tree-decompositions of width at most k in linear time with respect to the input size n, given that such a tree-decomposition exists. We consider this step in Chapter 4.

2. We apply a special kind of algorithm that traverses the nodes of the tree-decomposition from the leaves up to the root and evaluates some tables along the way. These tables correspond to some intermediate solutions. The root node, finally, should somehow yield a solution for the overall problem.

   Those kind of algorithms are, usually, fixed-parameter tractable and many algorithms even have linear time complexity in n. We consider such algorithms in Chapter 3.

This way, many *NP*-hard graph problems indeed become fixed-parameter tractable. Of course, the constants hidden in the $\mathcal{O}$-notation are usually quite large. They usually grow exponentially with respect to the graphs treewidth!

Nevertheless, the obtained algorithms are of practical significance and the main part of this thesis will follow this scheme. We will finally mention one other type of algorithm in the final Chapter 5.

2.2 Useful concepts for algorithmic considerations

As we already mentioned, the concepts of tree-decompositions and treewidth explored in Chapter 1 are very useful when they are applied to develop certain kinds of graph theoretical algorithms. Anyway, the abstract definition is quite difficult to handle.

Therefore concepts, such as the following *nice* tree-decompositions, were developed which provide a more specific structure. Both nice tree-decompositions and terminal graphs are very useful in the following chapters.

2.2.1 Nice tree-decompositions

The following notion definitely serves the purpose of simplifying the structure of a tree-decomposition. We mainly refer to [11] for this section.

Definition 2.2.1. A tree-decomposition $(T, \mathcal{X})$ is called *nice*, if the tree T is a rooted tree and each vertex of T is of one of the following four types:

- A leaf $t \in V(T)$ with $|X_t| = 1$ is called a *leaf node* of $(T, \mathcal{X})$.
- A vertex $t \in V(T)$ with exactly one child t' and $X_t = X_{t'} \cup \{v\}$ for some vertex $v \in V \backslash X_{t'}$ is called a *introduce node* of $(T, \mathcal{X})$.
- A vertex $t \in V(T)$ with exactly one child t' and $X_{t'} = X_t \cup \{v\}$ for some vertex $v \in V \backslash X_t$ is called a *forget node* of $(T, \mathcal{X})$.
- A vertex $t \in V(T)$ with exactly two children t_1, t_2 and $X_t = X_{t_1} = X_{t_2}$ is called a *join node* of $(T, \mathcal{X})$.

We again use the term *node* for all these types of vertices of the tree T.

This definition is quite practical because T provides some kind of order, in which the vertices can be traversed and, additionally, the possible situations occurring throughout this traversal are restricted to just four cases.

It is thus nice that we can always find tree-decompositions of this kind - just by introducing some more intermediate vertex steps and without increasing the treewidth. Our proof was motivated by the scheme given by Scheffler in [12] but works a little different because we use a slightly different definition of nice tree-decompositions.

Theorem 2.2.2. *Given a tree-decomposition $(T, \mathcal{X})$ of width k, we can always find a nice tree-decomposition $(T', \mathcal{X}')$ with the same width k and just $\mathcal{O}(k \cdot |V|)$ nodes. This transformation can be done in $\mathcal{O}(k^2 \cdot |V|)$ time.*

Proof. We obtain a nice tree-decomposition of width k by the following procedure. At first we choose an arbitrary root r for our tree T and orient it.

Next we traverse the tree from the root to the leaves, while iteratively removing undesirable cases. When we reach a node with $m > 2$ children, we recursively replace it by suitable join nodes i.e. in each step, we split up the m children into smaller sets C_1, C_2 and make the nodes in C_1 successors of the left child the nodes in and C_2 successors of the right child. Children having no children themselves do not need to be considered because they were redundant in the original tree-decomposition. Iteratively, we end up with a binary subtree.

For a node t having one child s and $X_t \subseteq X_s$ or $X_s \subseteq X_t$, we first omit the node with the minor vertex set, which was redundant in the previous decomposition anyway and continue with the remaining node. This step is necessary for the later estimation of the nodes! If t has one child s and the above conditions are excluded, we know

$$X_s \backslash X_t = \{v_1, \ldots, v_n\} \neq \emptyset \text{ and } X_t \backslash X_s = \{w_1, \ldots, w_m\} \neq \emptyset$$

We replace t by a series of introduce nodes for the vertices $v_1, \ldots, v_n$, followed by a series of forget nodes for the vertices $w_1, \ldots, w_m$. That way, we replace an edge $t - s$ by a series of $1 \leq n \leq k$ introduce and $1 \leq m \leq k$ forget nodes with vertex sets

$$\begin{aligned} X_t \supset X_t \backslash \{v_1\} \supset \ldots \supset X_t \backslash \{v_1, \ldots, v_{n-1}\} \supset \\ = X_t \cap X_s \subset X_s \backslash \{w_1, \ldots, w_{n-1}\} \subset \ldots \subset X_s \backslash \{w_1\} \subset X_s \end{aligned}$$

Similarly, we replace the nodes t without children and vertex set $X_t = \{v_1, \ldots, v_n\}$ by a series of n introduce nodes and a final leaf node with sets $X_t \supset X_t \backslash \{v_1\} \supset \ldots \supset \{v_n\}$.

We are now able to bound the number of nodes: In every forget node, we remove a vertex v. By Property (T3), these vertices are unique and there are at most $n = |V|$ forget nodes.

Introduce vertices are a bit more complex to estimate because a fixed vertex is possibly introduced in different subtrees. We inserted a series of at most k introduce nodes when dealing with nodes of degree 1, on the one hand, and when dealing with the leaves of T, on the other hand. The first type of introduce nodes can be bounded by kn because each such series is followed by at least one forget node. In our construction, the second type of construction does not appear after a join node. Since the intermediate introduce nodes are always followed by at least one forget node, the second type of series is always started at a forget node. Thanks to this correspondence, there are no more than kn introduce nodes of the second type either. The same argument bounds the number of leaf nodes by n.

The fact that the result is a subdivision of a binary tree with at most n leaves immediately tells us that there are at most $n - 1$ (join) nodes of degree 2 and the total number of nodes is, hence, $\mathcal{O}(k \cdot |V|)$.

At each node in this traversal, we need $\mathcal{O}(k)$ operations for the local changes. Therefore, the time complexity for the traversal is determined by the number of visited nodes and the total traversal takes $\mathcal{O}(k^2 \cdot |V|)$ time. □

Through nice tree-decompositions are not more powerful in general, they make the design of algorithms a lot easier. Typically, the tree T is traversed from the leafs to its root and - depending on the type of the actual node - partial solutions of the former nodes are combined in an adequate way to a new extended (partial) solution. The suitable method for doing this is usually pretty easy for leaf and forget nodes but more complex for introduce and join nodes. We take a look at this more formally in Chapter 3.

Finally, we want to do a quick reformulation of Lemma 1.1.13 for nice tree-decompositions that we will exhaustively use in the next chapter.

We know, by Lemma 1.1.13, that for an edge $ij \in E(T), j < i$ either $U_i \subset X_i \cap X_j$ for $i = 1$ or $i = 2$, or the set $X_i \cap X_j$ is a separator for the sets U_1, U_2 of our graph G. In this case, U_1 is exactly the set of all vertices in X_i and U_2 is the set of all vertices that do occur in X_j and later vertex sets. So, in particular

Lemma 2.2.3. *For a nice tree-decomposition $(T, (X_i)_{i \in V(T)})$ and an arbitrary node i with predecessor $parent(i)$, there are no edges between the vertices of $V_i \backslash (X_i \cap X_{parent(i)})$ and the vertices in $V \backslash V_i$.*

This tells us that, whenever we are done executing a particular node i, we do not have to bother about the vertices in $V_i \backslash X_i$ anymore in future adjacency tests. Therefore, it is possible to traverse the neighbours very efficiently!

2.2.2 Terminal graphs

One can consider nice tree-decompositions as certain algebraic expressions that generate a graph, see [4] or [11].

Definition 2.2.4. A *terminal graph* is a triple (V, E, X), where (V, E) is an undirected graph and $X \subseteq V$ is an ordered set of terminals i.e. a list of vertices (with a special role). If there are l terminals, we say our terminal graph is a *l-terminal graph*.

We sometimes speak of *terminal subgraphs*.

Definition 2.2.5. A l-terminal graph H is called (terminal) subgraph of $G = (V, E)$ if there exists a l-terminal graph H' with $G = H \oplus H'$, where $\oplus$ denotes the operation of first taking the disjoint union and then pairwise identifying the i-th terminals of the argument graphs. Possible multiple edges are omitted.

Note that the operation $\oplus$ works on l-terminal graphs but the result is an ordinary graph. We will reuse this operation in Chapter 3.

The notion of nice tree-decompositions $(T, (X_i)_{i \in V(T)})$ could also be stated in terms of terminal graphs. To each node i we associate a terminal graph $(V_i, E(V_i), X_i)$ induced by the vertex set

$$V_i = \bigcup \{X_j | j = i \text{ or } j \text{ lies below } i \text{ in our rooted tree } T\}$$

i.e. all vertices contained in the vertex sets of i and the nodes below i. The terminal vertices are the vertices in the vertex set of the current node. We use $i <_T j$ to express, that a node j lies below i in the tree T - in particular, the root is the unique minimal element in this tree-order.

The four types of nodes are translated into operations of arity $0, 1$ and 2 on terminal graphs.

Leaf node: The constant LEAF gives a 1-terminal graph $(\{v\}, \emptyset, (v))$ with one terminal vertex v. This obviously is the associated terminal graph induced by a leaf node v.

Introduce node: For each subset $S \subseteq \{1, \dots, |X|\}$, we define a unary operation INTRODUCE_S that adds a new vertex v to a terminal graph (V, E, X) which is adjacent to the i-th terminal node, for all $i \in S$. The vertex v is also added to the terminal list.

The result is, of course, the terminal graph induced by the vertex set $X \cup \{v\}$, given that the neighbours of v in (V, E) are given by S. This corresponds to a special introduce node introducing a new node v with exactly the neighbourhood S.

Forget node: For each node $1 \leq i \leq l$, we define a unary operation FORGET_i that makes the i-th terminal not terminal. This results, given a terminal graph (V, E, X), in the terminal graph $(V, E, X - \{x_i\})$ where $X = (x_i)_{i=1}^{l}$.

This corresponds to a special forget node, where the i-th element is deleted from the topmost vertex set and thus there is no connection to any possible new vertices anymore.

Join node: The binary operation JOIN takes two l-terminal graphs H_1, H_2 and yields the graph $H_1 \oplus H_2$ obtained by taking the disjoint union of H_1 and H_2 and identifying the terminals. The l nodes obtained by identification are exactly the terminals of the resulting terminal graph.

This exactly glues the two subgraphs together as it is intended for introduce nodes. All non-terminal nodes do not interfere with each other anyway by Lemma 1.1.12 and there is no need to identify them anyway.

Directly from the fact that each graph has a nice tree-decomposition, we can find a way to build a graph of treewidth at most k just by LEAF, JOIN, INTRODUCE_S and FORGET_i operations. On the contrary, this generation process also immediately gives a tree-decomposition. We thus observe the following.

Theorem 2.2.6. *A graph G has treewidth at most k if and only if $(V, E, \emptyset)$ can be formed by* LEAF, JOIN, INTRODUCE_S *and* FORGET_i *operations with $S \subset \{1, \dots, k\}$ and $1 \leq i \leq k + 1$.*

The later bounds for S and i immediately follow, since the maximum size of a vertex set is $k + 1$ by the definition of tree-decompositions.

2.3 Implementation details

Before designing an actual algorithm, one always has to think about some suitable data structures. An algorithm which involves graphs naturally needs some kind of structure storing the vertices, edges and their induced adjacency and incidence relations. Even simple time estimations on graphs, e.g. for adjacency tests, usually depend on this internal representation of the graph. It is, therefore, important to find a suitable representation.

We devote this short section to giving a clear overview, what kind of structures we intend to use to unify later complexity estimations.

2.3.1 A data structure for graphs

A very simple approach might be to just store the vertices and edges in lists. Anyway, this generally is not very efficient, because even for a simple adjacency test the entire edge list has to be traversed.

In practice, other representations for a graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ are used. Examples for such representations are:

adjacency matrices: We store a (symmetrical) matrix $G_M = (g_{ij})_{i,j=1}^{n}$ with $g_{ij} > 0$ if and only if v_i and v_j are adjacent. This way we can check adjacency of two vertices in constant time just by accessing the corresponding matrix element. To iterate over all neighbours one would need to traverse one column of the matrix, which can be done in $\mathcal{O}(n)$.

Obviously, one needs to store exactly n^2 values and the storage complexity is therefore $\Theta(n^2)$.

Adjacency lists: Another useful representation are adjacency lists $L_i = (n_{ij})_{j=1}^{n_i} \subset V^{n_i}, i = 1, \dots, n$ where the list L_i contains all the vertices adjacent to v_i. Of course, this gives an optimal time for traversing all neighbours, namely $\mathcal{O}(d(v_i))$, where $d(v_i)$ denotes the degree of v_i in G. On the contrary, for checking adjacency of two vertices v_i, v_j we either have to traverse the entire adjacency list L_i or L_j.

The storage needed for this representation is linear in $|E|$ because each edge contributes in exactly two adjacency list entries. Obviously this is $\mathcal{O}(n^2)$, but in general better - especially for sparse graphs!

Depending on the type of operations which need to be performed different representations might lead to the most efficient implementations.

We will mainly use adjacency matrices throughout the next chapter and - disregarding further remarks - we use this representation for giving complexity estimations for subparts of all our algorithms. Therefore, adjacency tests are performed in constant time, while traversing neighbours generally takes $\mathcal{O}(n)$ time. We do not bother because we usually only traverse the neighbours in the current vertex set X_i. This can obviously be

performed in $\mathcal{O}(k)$ time for graphs of treewidth at most k.

The storage complexity for graphs is thus $\mathcal{O}(n^2)$.

2.3.2 A data structure for tree-decompositions

We store nice tree-decompositions $(T, \mathcal{V})$ of graphs in linked list of type *Node* . Each *Node* n corresponds to some $t \in V(T)$ and has the following properties:

- *leftChild*: points to the entry corresponding to the left child of t, if existent
- *rightChild*: points to the entry corresponding to the right child of t, if existent
- *vertices*: an (ordered) list of all the indices of the vertices in X_t

We just need to save a pointer to *Node* n_r corresponding to the root node $r \in V(T)$ and then we can visit all nodes of our tree-decomposition by a depth-first- or breadth-first-search in linear time. Traversing the vertex list of any node, obviously takes $\mathcal{O}(k)$ additional time because the list has at most $k+1$ entries if the treewidth is at most k.

The storage complexity for this kind of structure is $\mathcal{O}(k|T|)$. Without loss of generality, we can assume that $|T|$ is in $\mathcal{O}(kn)$ by Theorem 2.2.2 and, therefore, we use $\mathcal{O}(k^2 n)$ storage.

We can distinguish the four types of nodes by checking the child nodes in constant time: for join nodes both children are defined, for leaf nodes there are no children and introduce and forget nodes have exactly one child. The later ones generally differ in the size of *vertices*; alternatively we can e.g. only use left children for introduce nodes and right nodes for forget nodes. Anyway, we assume these checks are implemented via some boolean functions **isLeaf**(*Node* n), **isJoin**(*Node* n), **isIntroduce**(*Node* n) and **isForget**(*Node* n) running in constant time.

Another useful operation is to check at a node $t \in V(T)$ whether a vertex v has a neighbour in the set X_t. Note that this operation can be performed in $\mathcal{O}(k)$ by traversing the list *vertices*.

These functions will occur in Chapter 3. For simplicity, we assume that the vertices of G have a fixed order which is followed in all lists *vertices*. This simplifies modifications in the building process, because we can detect the index of the modified element by just comparing the corresponding vertex lists.

This, finally, concludes the introductory part of this thesis. With Chapter 3, we finally start to construct some nice algorithms working with nice tree-decompositions.

3 Algorithms for Graphs with Known Decomposability

After introducing the necessary basics in Chapters 1 and 2, we are now able to start solving problems on graphs of bounded treewidth.

The knowledge of a tree-decompositions of a graph G makes a variety of (sometimes even *NP*-hard) problems on graphs *fixed-parameter tractable*. Using this approach, we can, assuming that the treewidth of our graph G is bounded by some fixed integer k, solve *NP*-complete problems such as l-colouring of vertices, maximum independent set and others in polynomial time.

We will, throughout this chapter, take a look at such algorithms and some general approaches that can be applied for a number of different graph problems.

3.1 Dynamic programming on graphs

In this section, we introduce a general technique for solving graph theoretical problems that was first mentioned in 1989 by Arnborg and Proskurowski in [6].

Of the approaches mentioned in this chapter, this one is maybe the most intuitive and informal one. Nevertheless, it involves a custom process for each problem and we will, therefore, spend most of this section giving concrete examples for such algorithms.

The original technique by Arnborg and Proskurowski was intended for partial k-trees but a similar scheme can be applied for graphs of bounded treewidth, as mentioned in [11].

3.1.1 General outline

The algorithm consists of the following steps:

1. Compute a nice tree-decompositions of G with width at most k (or equivalently a k-tree H containing G). As we will see in Chapter 4, this can be done in linear time.
2. Traverse the graph in the order given either by the tree T of our tree-decomposition or by a perfect elimination ordering for H. At each step, some kind of *partial solution* for a subtree of G is computed. This *partial solution* is some kind of table

storing certain *characteristics* of the subtrees induced by the set of already handled nodes.

For this computation, the *partial solution*s of previous steps are combined according to the local structure of G. Given a nice tree-decomposition, there is a fixed way of combining the *partial solution*s for each type of node. For partial k-trees, a similar update procedure is used.

The time spend in this traversal obviously depends on the update procedure. A reasonable update procedure should take at most polynomial time, giving a polynomial time complexity for the traversal. In many cases, the time-complexity of the update procedure is even constant in the number of vertices/nodes (e.g. $\mathcal{O}(k)$). This results in linear time complexity for the total traversal linear (with respect to the size of G).

3. From the characteristics given by the tables of the *partial solution* for the root node of T (or similarly the final k-clique in the elimination process of the k-tree H), one is then able to compute an answer or solution for the global problem.

The structure used by Arnborg and Proskurowski in [6] involves the concept of *branches* i.e. connected components induced by *descendants* of k-cliques (with respect to the order given by the elimination ordering for H). While iteratively removing vertices in the elimination ordering, the information about branches and their state information is updated.

In this thesis, we use a dynamic-programming approach involving nice tree-decompositions (see Section 2.2) as it is done in [11]. Given some upper bound k on the treewidth, we compute a tree-decomposition of width at most k and transform it into a nice tree-decomposition $(T = (I, E), (X_i)_{i \in I})$ – both steps take linear time.

To each node $i \in I$ we associate the set of vertices

$$V_i = \bigcup \{X_j | j = i \text{ or } j \text{ lies below } i \text{ in our rooted tree } T\}$$

i.e. all vertices contained in the vertex sets of i and the nodes j below i. The associated subgraphs are denoted by $G_i := G[V_i]$.

The tree T is traversed by a depth-first search and the nodes are executed in a bottom-up order i.e. from the leaf nodes to the root. At each node i the *characteristics* of the *partial solution*s for the direct descendants of i are *extended* to yield the *characteristics* for the subgraph G_i. At the end of the traversal, we obtain the *characteristics* of the *partial solution* for $G = G_r$ where r is the root node of T. From this *partial solution*, we derive a *solution* for our problem.

3.1.2 A detailed framework for graph properties

At this point, we need to get a bit more accurate with some notions we already started to use. These notions are defined in terms of terminal graphs.

If we want to find an algorithm deciding some graph property P, we have to define the following notions.

Solution: A *solution* for a graph property P is, formally, characterized by a binary relation sol_P that takes a graph G and a sufficient description of the solution s (e.g. in form of a string). There has to hold

$$P(G) \Leftrightarrow \exists s : sol_P(G, s)$$

Partial solution: A *partial solution* s for a property P, a graph G and a terminal subgraph H of G should describe the possible local behaviour of a solution for G on H. It usually is (at least some kind of) a restriction of the solution on G to the subgraph induced by H. Formally, we characterize this by a binary function $psol_P$ taking a terminal graph H and a description of the corresponding partial solution s.

Extension of a partial solution: An *extension* of a partial solution is usually defined quite naturally e.g. such that the restriction to the original set equals the original solution. Formally, it is described by a relation ex_P with four arguments: a graph G, a solution s, a terminal graph H and a partial solution s', such that

$$ex_P(G, s, H, s') \Rightarrow \exists H' : G = H \oplus H' \wedge sol_P(G, s) \wedge psol_P(H, s') \tag{3.1}$$

Also there should be partial solutions for all terminal subgraphs i.e. for all graphs G, solutions s and terminal graphs H, H'

$$(sol_P(G, s) \wedge G = H \oplus H') \Rightarrow \exists s' : psol_P(H, s') \wedge ex_P(G, s, H, s') \tag{3.2}$$

Characteristics of a partial solution: The *characteristics* of a partial solution should contain all necessary information to see if a partial solution can be extended. This can, formally, be characterized by a function ch_P defined on pairs of terminal subgraphs H and partial solutions s for H.

These characteristics induce equivalence classes of partial solutions for the graph H by identifying those which have the same characteristics. Since we just want to work with the characteristics, two partial solutions with same characteristics should be either both or none extensible i.e. for all terminal graphs H, H' and all (partial) solutions s, s'

$$\begin{aligned} &ch_P(H, s) = ch_P(H, s') \Rightarrow \\ &(\exists s_1 : ex_P(H \oplus H', s_1, H, s) \Leftrightarrow \exists s_2 : ex_P(H \oplus H', s_2, H, s')) \end{aligned} \tag{3.3}$$

Full set of characteristics: This set contains a list of all possible characteristics of partial solutions for a terminal graph H i.e.

$$full_P(H) = \{ch_P(H, s) | psol_P(H, s)\}$$

For a graph G_i induced by a node in a nice tree-decomposition, we often use the phrase *full set for* i.

While defining all these notions precisely, one also need to make sure that

- for each of the four types of nodes in a nice tree-decomposition, the full set of characteristics of the terminal subgraph G_i can be computed efficiently given the full sets for all children of i, **and**
- the problem can be decided efficiently given the full set for the root node.

Efficiently means here – depending on the application – usually either constant or polynomial time complexity with respect to n.

Generally speaking, the algorithmic structure of algorithm 1 is the following. If we

Algorithm 1 *Full set* c = **Traverse**(*Node* n): traversal algorithm for G

```
if isLeaf(n) then
    c = Leaf(n.vertices)
else if isJoin(n) then
    c1 = Traverse(n.leftChild)
    c2 = Traverse(n.rightChild)
    c = Join(n.vertices,c1,c2)
else if isIntroduce(n) then
    m = n.leftChild
    c1 = Traverse(m)
    i = index of node in X_t i.e. minimal index with n.vertices(i) ≠ m.vertices(i)
    c = Introduce(n.vertices,i,c1)
else if isForget(n) then
    m = n.rightChild
    c1 = Traverse(m)
    i = index of node in X_t i.e. minimal index with n.vertices(i) ≠ m.vertices(i)
    c = Forget(n.vertices,i,c1)
end if
```

execute **Traverse**(n) for *Node* n corresponding to the root node of T, the number of recursive calls of this function, of course, equals the size of T because each node is visited exactly once. At each of these nodes one of the functions **Leaf**(*Vertex list* v), **Join**(*Vertex list* v, *Index* i, *Full set* c), **Introduce**(*Vertex list* v, *Index* i, *Full set* c) and **Forget**(*Vertex list* v, *Index* i, *Full set* c) is called. If we denote the maximum complexity of these functions by $c(n)$, the total runtime is bounded by $\mathcal{O}(c(n) \cdot |T|)$ where $|T| = \mathcal{O}(k|V|)$.

3.1.3 Example: l-colouring of vertices

At first, we study l-colouring of vertices. Here the above concepts come in a very natural way. We assume that we are given a graph $G = (V, E)$ of treewidth at most k and we want to colour each vertex in V with one of l different colours such that no edge in E has both endpoints in the same colour class.

We consider the above notions. A solution for the l-colouring problem is obviously given by a function $f : V \to \{1, 2, \ldots, l\}$ such that

$$\forall uv \in E : f(u) \neq f(v).$$

A partial solution for a terminal graph $(V_i, E(V_i), X_i)$ is given by a solution for the graph $(V_i, E(V_i))$.

Such a partial solution f of G_i has an extension on the supergraph G if there exists a solution g for the l-colouring problem on G satisfying $g|_{V_i} = f$. The corresponding extension relation ex_P, by definition, satisfies the property stated in Equation (3.1). Furthermore, there always exist partial solutions given by restrictions as it is claimed in Equation (3.2).

We reconsider the result of Lemma 2.2.3. Applied to our colouring problem, we see that the colours of the vertices in $V_i \backslash X_i$ do not matter anymore because all adjacent edges were already handled. Reasonable characteristics for partial solutions f on G_i , therefore, are the restriction of f to the terminal vertices X_i. This way, Equation (3.3) is valid immediately.

With this definition, the full set for a node i contains all valid l-colourings for the subgraph induced by the vertex set X_i. We can obviously save these colourings at a node $i \in V(T)$ as an array of vectors

$$(s_j^i)_{j=1}^{|X_i|} \in \{1, \ldots, l\}^{|X_i|}, \text{ where } s_j^i \text{ denotes the colour-class of the } j\text{-th vertex in } X_i.$$

By definition of treewidth, there holds $|X_i| \leq k+1$ and, therefore, there are at most l^{k+1} possible l-colourings. The overall storage complexity at each node is thus $\mathcal{O}(kl^{k+1})$.

We need to consider the computation steps at all four types of nodes in a nice tree-decomposition.

Leaf node: At a leaf t with $X_t = \{v\}$, the computation is quite easy. The l possible colourings of the single node v can obviously be computed in $\mathcal{O}(l)$.

Algorithm 2 *Full set* c = **Leaf**(*Vertex list* X_t): computes all colourings for a leaf node t

```
for all colours j = 1, ..., l do
    add the unitary vector (j) to c
end for
```

Introduce node: At an introduce node t with $X_t = X_s \cup \{v\}$, a new vertex v is added, which is adjacent to at most k vertices in the previous vertex set X_s.

We just need to traverse all solutions for the previous node s (at most l^k colourings) and, for each, check which extensions are valid. This, obviously, can be done in $\mathcal{O}(kl^{k+1})$ operations by checking the colours of the adjacent vertices for each possible extension.

Algorithm 3 *Full set* c = **Introduce**(*Vertex list* X_t, *Index* i, *Full set* c_s): extends the colourings for introduce nodes

for all colourings $(t_1, \ldots, t_n) \in c_s$, colours $j = 1, \ldots, l$ **do**
  if for all $v \in X_t$ adjacent to the new vertex, $s_i \neq j$ holds **then**
    add the vector $(s_1, \ldots, s_{i-1}, j, s_i, \ldots, s_n)$ to c
  end if
end for

Forget node: For a forget node t, the computation is easy. Since each valid l-colouring of G_s can be restricted to a valid l-colouring of the graph G_t, we just have to restrict all previous characteristics by omitting the deleted vertex. This can be done in $\mathcal{O}(kl^{2k+1})$ by restricting each solution while checking for duplicates.

Algorithm 4 *Full set* c = **Forget**(*Vertex list* X_t, *Index* i, *Full set* c_s): restricts the colourings for forget nodes

for all colourings $(s_1, \ldots, s_n) \in c_s$ **do**
  if if all elements of c are different to $(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$ **then**
    add the vector $(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$ to c
  end if
end for

Join node: At a join node T, there are two previous nodes t_1, t_2 with $X_t = X_{t_1} = X_{t_2}$. If we just evaluate the intersection of the full sets for t_1 and t_2, the results give exactly the valid l-colourings for X_t and, therefore, the elements of the full set for t. This can be done in $\mathcal{O}(kl^{2k+2})$ by just checking all pairs in $X_{t_1} \times X_{t_2}$.

Algorithm 5 *Full set* c = **Join**(*Vertex list* X_t, *Full set* c_1, *Full set* c_2): restricts the colourings for join nodes

for all colourings $(s_1, \ldots, s_n) \in c_1$, $(s'_1, \ldots, s'_n) \in c_2$ **do**
  if $s_i = s'_i$, for all $i = 1, \ldots, n$ **then**
    add the vector $(s_1, \ldots, s_n)$ to c
  end if
end for

It is clear that the corresponding decision problem can be solved efficiently, given the full set for the root node because we just need to check whether the set for the root node is non-empty (positive instance) or empty (negative instance). We see that, although the complexity of some steps might be even exponential in k, the time is constant in $n := |V|$ and, therefore, the total traversal of all $\mathcal{O}(k|V|)$ nodes of T can be done in $\mathcal{O}(k^2 l^{2k+2} \cdot |V|)$ time i.e. we have a time complexity linear in $|V|$.

Additionally, one could also construct one (or even all) actual valid l-colouring by go-

ing back through all intermediate tables afterwards. At each step in the traversal, we have to choose one (or all) partial solutions that coincide with the previous one(s) at the intersection nodes. These could be saved by some additional book-keeping in the evaluation e.g. by saving pointers to the values in the previous table. Thanks to the fact that there are still at most l^{k+1} colourings in each full set, we can execute this traversal in linear time with respect to the number of vertices/nodes and, therefore, the construction problem has linear time complexity with respect to $|V|$.

This approach gives an algorithm for deciding if a graph is l-colorable, and even for the construction of such a colouring, that performs well in practice – even on graphs with many vertices – as long as the treewidth stays small.

Theorem 3.1.1. *The l-colouring problem for a graph $G = (V, E)$, an integer l and a nice tree-decomposition for G, can be solved in $\mathcal{O}(k^2 l^{2k+2} \cdot |V|)$ time i.e. in linear time complexity with respect to $|V|$.*

The exponents of l could be improved by choosing a more sophisticated implementation but still we see linear complexity.

It is also possible to evaluate the actual chromatic number (which is at most $k + 1$ by Corollary 1.2.18) by just slightly changing the $(k + 1)$-colouring algorithm if we apply the method introduced in Section 3.2.

Similar dynamic programming approaches can even be applied to other types of colouring problems. We would like to mention a discussion of the equitable colouring problem by Bodlaender and Fomin [13] and the article about precoloring-extension and the list-colouring problem by Jensen and Scheffler [14].

3.1.4 Example: The k-disjoint path problem

The following problem might seem a bit less intuitive but a similar approach works here as well. We want to sketch an algorithm for the k-disjoint-path problem i.e. the problem P of deciding, given a graph $G = (V, E)$ and some pairs of vertices $(v_j, w_j), 1 \leq j \leq k$, whether there exist k mutually disjoint paths from each v_j to w_j.

Again we provide the formal setting: A solution obviously should be a set of paths with the desired property.

A partial solution for G_i is a collection of disjoint paths that should correspond to the restriction of a solution to the graph G_i.

Therefore, we consider what happens when we restrict a solution to a subgraph. Due to the separation-properties of tree-decompositions, a restriction of a solution $(v_j P_j w_j)_{j=1}^k$ to a subgraph G_i consists of the following elements:

- If all vertices of P_j are in V_i, then there exists a full path $v_j P_j w_j$ with both $v_j, w_j \in G_i$.

- If both vertices v_j, w_j are in V_i but intermediate vertices are not, we get two paths $v_j P_j v'_j, w'_j P_j w_j$ with $v_j, w_j \in V_i$ and $v'_j, w'_j \in X_i$.
- If $v_j \in V_i$ but $w_j \notin V_i$, then we just get a path $v_j P_j v'_j$ with $v'_j \in X_i$.
- If, similarly, $v_j \notin V_i$ but $w_j \in V_i$, then we just get a path $w'_j P_j w_j$ with $w'_j \in X_i$.
- Additionally, we possibly get some paths $v'_j P_j w'_j$ with $v'_j, w'_j \in X_i$ which are subpaths of paths P_j of the previous three types.

In [12], a collection of such paths is called *plausible* solution and these plausible solutions exactly give the partial solutions for G_i. Without loss of generality, we assume that there are no isolated vertices $v' \in X_t$ in a plausible solution since removing them gives just another plausible solution.

A partial solution is called *feasible* if it is the restriction of some solution for the graph G. This is, of course, compatible with our axioms.

The characteristics of a partial solution L_i for G_i is a vector $(\phi(v))_{v \in X_i}$ with the following values:

- $\phi(v) = 0$ if the vertex v is unused by the current set of paths.
- $\phi(v) = 1$ if the vertex v already has maximal degree (i.e. degree 1 for the source and target vertices v_j, w_j and degree 2 for all other vertices).
- $\phi(v) = P_j$ if v is neither source nor target and there exists a path $v_j P_j v$ or a path $v P_j w_j$ in our partial solution.
- $\phi(v_j) = (w_j, 0), \phi(w_j) = (v_j, 0)$ if there exist paths $v_j P_j v$ and $w P_j w_j$ in our current set.
- $\phi(v_j) = (w_j, 1), \phi(w_j) = (v_j, 1)$ if there exists a path $v_j P_j w_j$.

Since the number of vertices in a set X_i is at most $k + 1$, the size of the full set for each i is bounded by some constant. We can compute the full sets in a bottom-up order such that the computation step at each node is constant with respect to $n := |V|$. A detailed description and full argumentation is given in [12] by Petra Scheffler.

3.2 Adaptation for optimization problems

Many graph problems P do involve some optimization process. For instance, the problem of finding the chromatic number, which was already mentioned above, where we ask for the *minimum* number l of colours such that a graph is l-colorable. Fortunately, we can reuse the general approach to tackle optimization as well (see [11]).

We define the concepts of solutions, partial solutions on terminal subgraphs and extensions without bothering about optimization. There are also some characteristics similar to the ones in the previous section.

The optimization is handled by an additional choice of property for an arbitrary partial solution s for a subgraph H. We not only consider the characteristics $ch_P(H, s)$ but also some kind of valuation $\nu_P(H, s)$ which rates the quality of the corresponding partial solution. This rating $\nu_P(H, s)$ should be an element of a totally ordered set – usually either an integer or a real number.

For each equivalence class of partial solutions, we just keep the best possible valuation $\nu_P(G_i, s)$ i.e. we basically ignore all other partial solutions in the same equivalence class having worse valuations. Of course, we need to make sure – with a reasonable definition of characteristics and valuation – that we do not lose any actual solutions here. If we look for small valuations, we – at least – need for all subgraphs H, H' and corresponding solutions s, s', t, t'

$$\begin{aligned} ch_P(H, s) = ch_P(H, s') \wedge \nu_P(H, s) < \nu_P(H, s') \wedge ex_P(H \oplus H', t, H, s) \\ \wedge ex_P(H \oplus H', t', H, s') \Rightarrow \nu_P(H \oplus H', t) < \nu_P(H \oplus H', t'). \end{aligned} \tag{3.4}$$

Looking for large valuations works similarly.

This way, we restrict the full sets for $i \in I$ to those characteristics representing only the best possible partial solution of each equivalence class.

Example (Chromatic number). In case of the chromatic number, we can just choose the number of used colours to rate partial solutions. Of course, we do not have to consider solutions that are identical on the sets X_i but use more colours on the nodes $V_i \backslash X_i$ – they will always result in a suboptimal solution for G! Therefore, it is sufficient to save the characteristics and the minimum of colours necessary to obtain these.

3.2.1 Example: Independent set

A common application is the problem of finding a maximum-size independent set $W \subseteq V$.

Definition 3.2.1. A set $W \subseteq V$ is called *independent* if each pair of vertices $w_1, w_2 \in W$ is non-adjacent i.e. $w_1 w_2 \notin E$.

This problem is, for instance, considered in [6] and [15].

The problem is obviously a special case of the more general maximum-weighted independent-set problem. Given a graph $G = (V, E)$ and some vertex weights $c(v), v \in V$, we want to find an independent set $W \subseteq V$ that maximizes $c(W) := \sum_{v \in W} c(v)$. We follow the approach presented by Bodlaender in [16].

A solution for this problem, of course, is a set $L \subseteq V$ of vertices of $G = (V, E)$ forming an independent set. A partial solution L_i for a terminal subgraph $(V_i, E(V_i), X_i)$ is defined as a solution for the graph G_i and, as such, $L_i \subseteq V_i$.

A solution L of vertices is an extension of a partial solution L_i on the subgraph G_i if and only if $L \cap V_i = L_i$. Of course, the conditions stated in Equation (3.1) and (3.2)

hold. For the later condition, we use that restrictions of solutions on subgraphs induced by vertex sets V_i are solutions for the graph $(V_i, E(V_i))$ because adjacency is inherited from the original graph.

Next, we have to think about reasonable characteristics and valuation of a partial solution L_i on G_i. Lemma 2.2.3 assures that no more edges interfere with the nodes in $V_i \backslash X_i$ and thus, apart from the nodes in $L_i \cap X_i$, only the sum of the overall vertex weights is important. Therefore, we choose $ch_P(G_i, L_i) = L_i \cap X_i \subseteq X_i$ and valuation $\nu_P(G_i, L_i) = c(L_i)$. We immediately see that Equation (3.4) is valid. For each possible characteristics $S \subseteq X_i$, we store the corresponding maximal valuation $c_i(S)$.

In particular, the full set of a node i consists of a table of at most 2^{k+1} different values. For each subset $S \subseteq X_i$, we save the maximum weight $c_i(S)$ of an independent set $W \subseteq V_i$ with characteristics S i.e. $W \cap X_i = S$. If there is no such independent set at all, we define $c_i(S) = -\infty$. The subsets could – for instance – be realized as binary strings of length $|X_i|$. This way, table lookups are possible in constant time. The storage complexity of this table is $\mathcal{O}(2^{k+1})$.

We describe the computation necessary in each of the four types of nodes of our nice tree-decomposition.

Leaf node: Since $X_t = \{v\}$ for some vertex v, we just evaluate $c(\emptyset) = 0$ and $c_i(\{v\}) = c(v)$ which can obviously be done in constant time.

Algorithm 6 *Full set* $c =$ **Leaf**(*Vertex list* X_t): evaluates the maximum weight for a leaf node

$c([0]) = 0$
$c([1]) = c(v)$, for the vertex $v \in X_t$

Introduce node: There is a unique child s of t such that G_t is obtained from G_s by adding one terminal vertex v. Note that v is only adjacent to vertices in X_s.

We distinguish the following cases:

case 1: Of course, for $S \subseteq X_s$ we still have the same maximum-weight independent set and $c_t(S) = c_s(S)$.

case 2: For the sets $S \cup \{v\}$ containing the new vertex v, we first consider the case that v is adjacent to any node in S. There obviously is no valid independent set and $c_t(S \cup \{v\}) = -\infty$.

case 3: Otherwise $S \cup \{v\}$ is, at least, an independent set. So $c_t(S \cup \{v\}) \geq c_s(S) + c(v)$ holds. But each maximum-weight independent set W for G_i with $W \cup X_i = S \cup \{v\}$ can be restricted to an independent set $W \backslash \{v\}$ for G_i and, thus, $c_s(S) \geq c_t(S \cup \{v\}) - c(v)$ as well.

We can implement this evaluation of the at most 2^{k+1} entries in a total time of $\mathcal{O}(k2^{k+1})$.

Algorithm 7 *Full set* c = **Introduce**(*Vertex list* X_t, *Index* i, *Full set* c_s): evaluates the maximum weight for an introduce node

Let v be the i-th vertex in X_t
for all $(s_1, \ldots, s_n) \in 2^{|X_t|-1}$ **do**
  $c([s_1, \ldots, s_{i-1}, 0, s_{i+1}, \ldots, s_n)) = c_s([s_1, \ldots, s_n])$
  if v is adjacent to some element $x_k \in X_s = X_t \backslash \{v\}$ with $s_k = 1$ **then**
    $c([s_1, \ldots, s_{i-1}, 1, s_{i+1}, \ldots, s_n]) = -\infty$
  else
    $c([s_1, \ldots, s_{i-1}, 1, s_{i+1}, \ldots, s_n]) = c_s([s_1, \ldots, s_n]) + c(v)$
  end if
end for

Forget node: Here, it is easy again. The unique child s of our node t was already handled and the only difference between the terminal graphs G_t and G_s is that $X_t = X_s \cup \{v\}$. When searching for the maximum-weight independent set W with $W \cap X_t = S$, we distinguish the cases $v \in W$ and $v \notin W$. In the first case, the maximum weight is given by $c_s(S \cup \{v\})$ and in the later one by $c_s(S)$. We just evaluate

$$c_t(S) = \max\{c_s(S), c_s(S \cup \{v\})\}$$

for all subsets S. This can, for each of the at most 2^k subsets, be done in $\mathcal{O}(k)$. Altogether, we spend $\mathcal{O}(k2^k)$ time at a forget node.

Algorithm 8 *Full set* c = **Forget**(*Vertex list* X_t, *Index* i, *Full set* c_s): evaluates the maximum weight for a forget node

for all $(s_1, \ldots, s_n) \in 2^{|X_t|}$ **do**
  $c([s_1, \ldots, s_n]) = \max\{c_s([s_1, \ldots, s_{i-1}, 0, \ldots, s_n]), c_s([s_1, \ldots, s_{i-1}, 1, \ldots, s_n])\}$
end for

Join node: For a node t with two children t_1, t_2, the parts of V_{t_1} and V_{t_2} do not interfere outside $X_t = X_{t_1} = X_{t_2}$ by the properties of tree-decompositions. For $S \subseteq X_t$, we just glue the independent sets together along S and obtain an independent set for G_t. Of course, there holds

$$c_t(S) \geq c_{t_1}(S) + c_{t_2}(S) - c(S).$$

On the other hand, a maximum-weight independent set W for G_t induces independent sets for both G_{t_1} and G_{t_2}. Thus, there holds

$$c_t(S) = c(W \cap V_{t_1}) + c(W \cap V_{t_2}) - c(S) \leq c_{t_1}(S) + c_{t_2}(S) - c(S)$$

and we just evaluate $c_t(S) = c_{t_1}(S) + c_{t_2}(S) - c(S)$, by table lookups and summation, in time $\mathcal{O}(k)$. Altogether, this takes at most $\mathcal{O}(k2^{k+1})$ time for a node t.

Algorithm 9 *Full set* $c =$ **Join**(*Vertex list* X_t, *Full set* c_1, *Full set* c_2): evaluates the maximum weight for a join node

for all $(s_1, \ldots, s_n) \in 2^{|X_t|}$ **do**
$c([s_1, \ldots, s_n)) = c_1([s_1, \ldots, s_n]) + c_2([s_1, \ldots, s_n]) - \sum_{x_j \in X_i : s_j = 1} c(x_j)$
end for

By our usual post-order traversal, we evaluate the table for the root node r in time $\mathcal{O}(k2^{k+1} \cdot |I|)$ i.e. in linear time with respect to the number of vertices $|V|$.

The problem of evaluating the maximum-weight of an independent set is solvable by evaluating the maximum value of all equivalence classes

$$\max_{S \subseteq X_r} c_r(S).$$

This can be done in $\mathcal{O}(2^{k+1})$ and, therefore, does not increase the overall time.

The construction variant of Weighted independent set could be solved from this table by additional book-keeping (e.g. by saving pointers to the table of the previous nodes during the evaluation). This does not increase the runtime as well and we obtain:

Theorem 3.2.2. *The maximum-weighted independent-set problem for a graph* $G = (V, E)$*, a list of vertex weights* $c(v), v \in V$ *and a nice tree-decomposition of* G*, can be solved in* $\mathcal{O}(k^2 2^{k+1} \cdot |V|)$ *time.*

Once more, the exponents could be improved by choosing a more sophisticated implementation.

3.2.2 Example: Vertex cover

We consider another simple optimization problem: A vertex cover is a subset $C \subseteq V$ of vertices such that for each edge $e \in E$ at least one end-vertex is in C. Usually, one considers the problem of finding a minimum-size vertex cover. We cover a more general version where fixed vertex weights $c(v), v \in V$ are given and search for a minimum-weight vertex cover i.e. $c(C) := \sum_{v \in C} c(v) \to \min$.

Solutions for the minimum-weight vertex-cover problem are subsets $L \subseteq V$, while partial solutions for a subgraph $(V_i, E(V_i), X_i)$ are solutions for the graph $(V_i, E(V_i))$ i.e. subsets $L_i \subseteq V_i$. Note that restrictions of solutions are solutions for the graph $(V_i, E(V_i))$ because all edges in $E(V_i)$ already need to be covered. A solution L is an extension of the partial solution L_i for G_i if $L \cap V_i = L_i$. Of course, the conditions in Equation (3.2) and (3.1) are satisfied by definition.

Next, we have to find suitable characteristics. We consider the adjacency lists of a graph $G = (V, E)$ to give an alternative characterization of vertex covers.

$C \subseteq V$ is a vertex cover if and only if there either holds $v \in C$ or $N(v) \subseteq C$ for each vertex $v \in V$.

Consider the terminal graph $G_i = (V_i, E_i, X_i)$: Vertices in $V_i \backslash X_i$ are not adjacent to vertices in $V \backslash V_i$ by Lemma 2.2.3. Therefore, vertices in $V_i \backslash X_i$ neither contain elements of $V \backslash V_i$ in their own adjacency lists, nor are they contained in adjacency list of vertices in $V \backslash V_i$. In conclusion, we do not need to store whether these are in C to be able to test the above criteria for vertices in $V \backslash V_i$. Thus it is sufficient to choose the vertices $C \cap X_i$ as characteristics for assuring Property (3.3).

The valuation of a partial solution L_i for G_i is given by the value $\nu_P(G_i, L_i) = c(L_i)$. Thanks to the above argument, Property (3.4) holds too. We denote the minimal valuation of a partial solution with characteristics S by $c_i(S)$. If there are no partial solutions with characteristics S, we set $c_i(S) = \infty$.

The full set for i contains the value $c_i(S)$ for each characteristics $S \subseteq V_i$. Since there are at most $k+1$ elements in X_i, a full set contains at most 2^{k+1} entries. We could, for instance, interpret the characteristics as binary strings and save the elements $c_i(S)$ as an array. This way, table lookups take $\mathcal{O}(k)$ time and the storage complexity is $\mathcal{O}(2^{k+1})$.

Now consider the evaluation steps for the nodes in a nice tree-decomposition. We possibly detect some similarities to the independent-set problem discussed in the previous subsection!

Leaf node: At a leaf node t, $X_t = \{v\}$ holds for some vertex $v \in V$. We just have to store the values $c_t(\emptyset) = 0$ and $c_t(\{v\}) = c(v)$ which can be done in constant time.

Algorithm 10 *Full set* $c=$ **Leaf**(*Vertex list* X_t): evaluates the minimum valuation for a leaf node

---

$c([0]) = 0$
$c([1]) = c(v)$, for the vertex $v \in X_t$

Introduce node: For an introduce node t with child s, there holds $X_t = X_s \cup \{v\}$ for some vertex $v \in V$. We could extend minimum-weight solutions for G_s with characteristics $S \subseteq V_s$ to solutions for G_t by adding v and increasing the weight by $c(v)$. This gives $c_t(S \cup \{v\}) \leq c_s(S) + c(v)$. On the contrary, a minimum-weight solution with characteristics $S \cup \{v\}$ induces a solution for G_s characterized by S with decreased valuation. Thus there holds

$$c_t(S \cup \{v\}) = c_s(S) + c(v).$$

If all neighbours of v in X_t are in some minimum-weight set S, S is another valid solution for G_t and $c_t(S) \leq c_s(S)$. Of course, we even have equality here. For $N(v) \not\subseteq S$ there is no valid vertex cover with characteristics S for G_t. In total,

$$c_t(S) = c_s(S), \text{ if } N(v) \subseteq S$$
$$c_t(S) = \infty, \text{ else}$$

Of course, each of these at most 2^{k+1} entries can be evaluated in $\mathcal{O}(k)$ by doing table lookup and testing the at most k neighbours of v in X_t. Altogether, this takes $\mathcal{O}(k2^{k+1})$ time.

Algorithm 11 *Full set* $c =$ **Introduce**(*Vertex list* X_t, *Index* i, *Full set* c_s): evaluates the minimum valuation for an introduce node

Let v be the i-th vertex of X_t
for all $(s_1, \ldots, s_n) \in 2^{|X_t|-1}$ **do**
  if there exists $x_k \in X_t \backslash \{v\}$ adjacent to v with $s_k = 0$ **then**
    $c([s_1, \ldots, s_{i-1}, 0, s_{i+1}, \ldots, s_n]) = \infty$
  else
    $c([s_1, \ldots, s_{i-1}, 0, s_{i+1}, \ldots, s_n]) = c_s([s_1, \ldots, s_n])$
  end if
  $c([s_1, \ldots, s_{i-1}, 1, s_{i+1}, \ldots, s_n]) = c_s([s_1, \ldots, s_n]) + c(v)$
end for

Forget node: For a forget node t with child s, there holds $X_s = X_t \cup \{v\}$ for some vertex $v \in V$. There is not much to do here because minimum-weight (partial) solutions for G_t are either induced by (partial) solutions for G_s containing v or not containing v. Of course, for $S \subseteq X_t$ there holds

$$c_t(S) = \min\{c_s(S), c_s(S \cup \{v\})\}.$$

We can evaluate these at most 2^k entries in $\mathcal{O}(k \cdot 2^k)$ by two table lookups each.

Algorithm 12 *Full set* $c =$ **Forget**(*Vertex list* X_t, *Index* i, *Full set* c_s): evaluates the minimum valuation for a forget node

for all $(s_1, \ldots, s_n) \in 2^{|X_t|}$ **do**
  $c([s_1, \ldots, s_n]) = \min\{c_s([s_1, \ldots, s_{i-1}, 0, s_i, \ldots, s_n]), c_s([s_1, \ldots, s_{i-1}, 1, s_i, \ldots, s_n])\}$
end for

Join node: For a join node t with children t_1, t_2 and $X_t = X_{t_1} = X_{t_2}$, the evaluation for $S \subseteq X_t$ is quite intuitive:

$$c_t(S) = c_{t_1}(S) + c_{t_2}(S) - c(S)$$

This equality holds because – if the characteristics S coincide – we can combine the maximum-weight solutions for G_{t_1} and G_{t_2} to one for G_i with valuation $c_{t_1}(S) + c_{t_2}(s) - c(S)$. On the contrary, a minimum-weight solution for G_t induces solutions for G_{t_1}, G_{t_2} with the same characteristics S by restriction. The evaluation works just like for the independent-set problem.

We can, of course, evaluate these at most 2^{k+1} values by table lookups and summing in $\mathcal{O}(k2^{k+1})$ time.

Algorithm 13 *Full set* $c =$ **Join**(*Vertex list* X_t, *Full set* c_1, *Full set* c_2): evaluates the minimum valuation for a join node

for all $(s_1, \ldots, s_n) \in 2^{|X_t|}$ **do**
$c([s_1, \ldots, s_n]) = c_1([s_1, \ldots, s_n]) + c_2([s_1, \ldots, s_n]) - \sum_{x_j \in X_i : s_j = 1} c(x_j)$
end for

By a simple post-order traversal, we evaluate the table for the root node r in time $\mathcal{O}(k2^{k+1} \cdot |I|)$ i.e. in linear time with respect to the number of vertices $|V|$.

We can solve the problem of evaluating the minimum weight of a vertex cover by evaluating the minimum valuation of all equivalence classes i.e.

$$\min_{S \subseteq X_r} c_r(S).$$

This can be done in $\mathcal{O}(2^{k+1})$ and, therefore, does not increase the overall time complexity.

The construction variant of the vertex-cover problem can be solved from this table by additional book-keeping, e.g. by saving pointers to the table of the previous nodes during the evaluation. This does not increase the runtime as well.

Theorem 3.2.3. *The minimum-weighted vertex-cover problem for a graph* $G = (V, E)$*, a list of vertex weights* $c(v), v \in V$ *and a nice tree-decomposition of* G*, can be solved in* $\mathcal{O}(k^2 2^{k+1} \cdot |V|)$ *time.*

The exponents could possibly be improved by choosing a better implementation.

3.2.3 Example: Dominating set

Another problem, which is often considered, is finding a minimum-size dominating set.

Definition 3.2.4. A set $W \subseteq V$ of a graph $G = (V, E)$ is called *dominating* if each vertex $v \in V$ is either in W or has a neighbour in W.

As for the maximum-size independent set, we consider the weighted version instead i.e. the problem of finding a minimum-weighted independent set given some additional vertex weights $c(v), v \in V$. Again, we just need to set all weights to 1 if we want the minimum-size dominating set. We follow the description given in [17].

Solutions are given by subsets $L \subseteq V$, while partial solutions for a subgraph $(V_i, E(V_i), X_i)$ are – as restrictions of solutions - given by subsets $L_i \subseteq V_i$. Note that, in this case, partial solutions are not necessarily solutions for the corresponding subgraphs!

A solution L is an extension of a partial solution L_i for G_i if $L \cap V_i = L_i$. Still, by definition, Properties (3.1) and (3.2) hold.

Next, we think of some reasonable characteristics: When doing the traversal for any subgraph G_i, the vertices in $V_i \backslash X_i$ already have to be covered completely because no further neighbours do occur. Therefore, we do not need to care about them in later steps of our algorithm. For the set X_i, we partition the vertices into three classes: the vertices in the dominating set $L_i \cap X_i$, the vertices that already have neighbours in the dominating set and the vertices we still need to cover.

We, therefore, choose the characteristics to be a 3-colouring of the vertices in X_i. Colour class X corresponds to the uncovered vertices, D covers all dominating vertices and C the dominated vertices. We save this as a vector of colours $s_j \in \{X, D, C\}$, where s_j denotes the colour of the j-th element $x_{ij} \in X_i$:

$$ch_P(G_i, L_i) := (s_j)_{j=1}^{|X_i|} \in \{X, D, C\}^{|X_i|}$$

Obviously, Property (3.3) holds!

The valuation of a partial solution L_i for G_i is given by the value $\nu_P(G_i, L_i) = c(L_i)$. Of course, Property (3.4) holds because of our choice of characteristics. For the equivalence class of some characteristics $S \in \{X, D, C\}^{|X_i|}$, we denote the maximum value of the valuation by $c_i(S)$. If there is no dominating set corresponding to these characteristics, we define $c_i(S) = \infty$.

The full set for i contains, for each such characteristics $S \in \{X, D, C\}^{|X_i|}$, the value $c_i(S)$. Of course, since there are at most $k + 1$ elements in X_i, there are at most 3^{k+1} entries and the storage complexity is $\mathcal{O}(3^{k+1})$.

We evaluate the tables in bottom-up order. We, additionally, observe recursively that the tables are monotonous in the following way: If some partial solutions s, s' differ only in components $s_i \neq s'_i$ with entries from $\{X, C\}$, then the partial solution with the least number of C-entries yields the smallest value.

Leaf node: For a leaf with $X_t = \{v\}$, there are only three entries $c_t(X) = 0$, $c_t(D) = c(v)$ and $c_t(C) = \infty$ which we evaluate in constant time. We are obviously monotonous here!

Algorithm 14 *Full set* $c =$ **Leaf**(*Vertex list* X_t): evaluates the table for a leaf node

$c(X) = 0, c(C) = \infty$
$c(D) = c(v)$, for the vertex $v \in X_t$

Introduce node: If, at a node t with child s, a vertex v is introduced as i-th terminal vertex, there holds

$$X_t = (x_{s,1}, \ldots, x_{s,i-1}, v, x_{s,i}, \ldots, x_{s,n}).$$

A (minimal-weight) partial solution $(s_j)_{x_j \in X_t \backslash \{v\}}$ for X_s can be extended to X_t in three possible ways. If we set $s_i = X$, we do not make any new choices yet and we do not increase the valuation.

Alternatively, we can choose $s_i = C$ and keep the valuation – this is only admissible if v has a dominating neighbour inside X_s! Of course, the new valuations do not need to be optimal. They are indeed optimal because each (minimum-weight) partial solution with characteristics $(s_j)_{x_j \in X_t}$ for X_t induces a partial solution for X_s. For $s_i \in \{X, C\}$, these are simply the restrictions and the valuation remains the same. So for these cases, we yield

$$c_t((s_j)_{x_j \in X_t}) = c_s((s_j)_{x_j \in X_t \setminus \{v\}}), \text{ if } s_i = X.$$
$$c_t((s_j)_{x_j \in X_t}) = c_s((s_j)_{x_j \in X_t \setminus \{v\}}), \text{ if } s_i = C \text{ and } v \text{ has a dominating neighbour.}$$
$$c_t((s_j)_{x_j \in X_t}) = \infty, \text{ if } s_i = C \text{ and } v \text{ has no dominating neighbour.}$$

Note that, due to these evaluations, the monotonicity is inherited from the previous node.

Another possibility is to make the new vertex dominating. This goes along with making all uncovered vertices in $N(v)$ dominated and increasing the valuation by $c(v)$. For $s_i = D$, we need to edit the values s_j corresponding to some $x_j \in N(v)$ with $s_j = C$ and no previous dominating neighbours as well. In this case, we can take the valuation of the induced partial solution $(s'_j)_{x_j \in X_s}$ which colours all neighbours of v with X instead. By monotonicity of the previous solution values this partial solution has the optimal valuation and we just add $c(v)$ to the current valuation:

$$c_t((s_j)_{x_j \in X_t}) = c_s((s'_j)_{x_j \in X_s}) + c(v), \text{ if } s_i = D.$$

Of course, each of these at most 3^{k+1} values can be evaluated by traversing all at most k neighbours and one table-lookup in $\mathcal{O}(k)$. Altogether, the time spent at node t is $\mathcal{O}(k3^{k+1})$.

Algorithm 15 *Full set* c = **Introduce**(*Vertex list* X_t, *Index* i, *Full set* c_s): evaluates the table for an introduce node

Let v be the i-th vertex of X_t
for all $(s_1, \ldots, s_n) \in \{X, D, C\}^{|X_t|}$ **do**
  $c([s_1, \ldots, s_{i-1}, X, s_{i+1}, \ldots, s_n]) = c_s([s_1, \ldots, s_n])$
  if there exists $x_k \in X_t \setminus \{v\}$ adjacent to v with $s_k = D$ **then**
    $c([s_1, \ldots, s_{i-1}, C, s_{i+1}, \ldots, s_n]) = c_s([s_1, \ldots, s_n])$
  else
    $c([s_1, \ldots, s_{i-1}, C, s_{i+1}, \ldots, s_n]) = \infty$
  end if
  Let s' be the solution obtained from s by $s'_k = X$ if $x_k \in X_j$ adjacent to v and $s_k = 0$
  $c([s_1, \ldots, s_{i-1}, D, s_{i+1}, \ldots, s_n]) = c_s([s'_1, \ldots, s'_n]) + c(v)$
end for

Forget node: For a forget node t with child s, it is is much easier. When making the vertex v non-terminal, it should either be coloured C or D. Both cases are actually admissible and we get the minimal valuation (for index i of v in X_t) by

$$c_t((s_j)_{x_j \in X_t}) = \min\{c_s((s_j)_{x_j \in X_t}, s_i = D), c_t((s_j)_{x_j \in X_t}, c_i = C)\}$$

This can be done by two table-lookups each in a total time $\mathcal{O}(k3^{k+1})$ and we observe that monotonicity is, again, preserved.

Algorithm 16 *Full set* $c =$ **Forget**(*Vertex list* X_t, *Index* i, *Full set* c_{prev}): evaluates the table for a forget node

for all $(s_1, \ldots, s_n) \in \{X, D, C\}^{|X_t|}$ **do**
$c([s_1, \ldots, s_n]) = \min(c_s([s_1, \ldots, s_{i-1}, D, s_i, \ldots, s_n]), c_s([s_1, \ldots, s_{i-1}, C, s_i, \ldots, s_n]))$
end for

Join node: It remains to consider the case of a join node t with two children t_1 and t_2 and $X_t = X_{t_1} = X_{t_2}$. Two (minimum-weight) partial solutions with characteristics $s^i, i = 1, 2$ for the child nodes could be extended simultaneously if they only differ in nodes $x_j \in X_t$ with $s^i_j \in \{X, C\}, i = 1, 2$ i.e. there are some nodes not covered by one colouring but, somehow, dominated in the other. Those values yield $s_j = C$ in their common extension. In these cases, there holds:

- $s_j \in \{D, X\} \Rightarrow s_j = s^1_j = s^2_j$
- $s_j = C \Rightarrow s^1_j, s^2_j \in \{C, X\} \wedge (s^1_j = C \vee s^2_j = C)$

If these properties hold, we say that s^1 and s^2 *divide* s. We observe that it is sufficient to restrict to the case $s^1_j \neq s^2_j$ if $s_j = C$ because we have monotonicity of the previous solution. If we have a minimal-weight partial solution for t, then its restrictions are, of course, of minimal weight too. So indeed,

$$c_t(s) = \min\{c_{t_1}(s^1) + c_{t_2}(s^2) - c(\{c(x_j) : x_j \in X_i, s_j = D\}) : s^1, s^2 \text{ divide } s\}.$$

But how can we compute these values efficiently? First, we bound the number of valid pairs of characteristics for t_1, t_2. For fixed k, there are at most

$$2^{|X_t|-k} \cdot \binom{|X_t|}{k}$$

colourings with exactly k values equal to C. For each such colouring, there are at most 2^k colourings that differ only in the C-values and could, therefore, contribute. Altogether we get at most

$$\sum_{k=0}^{|X_t|} 2^{|X_t|-k} \cdot \binom{|X_t|}{k} \cdot 2^k = 4^{|X_t|}$$

possible pairs that can contribute to any minimum. So the minimum can be evaluated in $\mathcal{O}(4^{k+1})$.

Algorithm 17 *Full set* $c =$ **Join**(*Vertex list* X_t, *Full set* c_{t_1}, *Full set* c_{t_2}): evaluates the table for a join node

```
for all k = 1, ..., |X_i| do
    for all s ∈ {D, C, X}^{|X_t|} with exactly s_{i_1} = s_{i_2} = ... = s_{i_k} = C do
        c[s] = ∞
        for all solutions s', s'' that divide s i.e. use some X-values instead of C do
            c[s] = min{c[s], c_{t_1}[s'] + c_{t_2}[s''] − Σ_{x_ti ∈ X_t, s_i = D} c(x_ti)}
        end for
    end for
end for
```

Altogether, this traversal can be done in time $\mathcal{O}(4^{k+1} \cdot |I|)$ i.e. with linear time with respect to the size of our graph G.

We can finally compute the valuation of the minimum-weight dominating set from the full set of the root node r by

$$\min\{c_r(s) : s \in \{D, C\}^{|X_r|}\}.$$

This just takes the minimum of all partial solutions without uncovered nodes. Of course, this can be done in $\mathcal{O}(3^{k+1})$ and, therefore, it does not increase the total runtime.

Again, with some additional book-keeping, we can even solve the construction variant in $\mathcal{O}(4^{k+1}k \cdot |V|)$ time. In conclusion, we yield:

Theorem 3.2.5. *The minimum-weighted dominating-set problem for a graph* $G = (V, E)$*, a list of vertex weights* $c(v), v \in V$ *and a nice tree-decomposition of* G*, can be solved in* $\mathcal{O}(4^{k+1}k \cdot |V|)$ *time.*

3.2.4 Additional remarks

There have been quite a few attempts to find a common framework for such dynamic programming approaches, which works for a large class of graph problems and helps deciding the existence of a linear-time algorithm (or even one with another polynomial bound for time complexity) at an earlier stage of the design process.

A quite general approach can be found in [18] by Bodlaender, where he works with monoids to deal with larger classes of graph decision problems on graphs of bounded treewidth. For some results, he additionally, assumes bounded degree.

In [11], Bodlaender mentions an approach dealing with automata theory. The purpose is to introduce an equivalence relation on k-terminal graphs which has only a finite number of equivalence classes. This helps bounding the number of different possible full sets and can, therefore, be used for bounding the overall runtime.

3.3 A logical approach

The dynamic programming approach, which was discussed above, seems quite natural for graphs with given tree-decomposition and is quite powerful. Nevertheless, we have seen that each problem requires a detailed discussion and it often involves more or less complex specific considerations.

There have been quite a few attempts to find a more general setting for linear-time algorithms working for even larger classes of problems. Additionally, it was desirable to find an approach which allows to decide whether a problem can be solved in linear time with respect to the number of vertices at an earlier stage in the design process.

A very successful approach of this kind was introduced by Courcelle [19]. He showed that each graph property which can be stated in *monadic second-order logic*(MSOL), an extended version of first-order logic allowing quantification over sets of vertices and sets of edges, can be solved in linear time for graphs of bounded treewidth.

His work was extended to *extended* monadic second-order logic, see [20], [21] and [22].

We want to give a short description of this method and the corresponding result, although a full consideration is out of scope of this thesis.

3.3.1 A language for graph problems

In this first section, we introduce a special language suited for graph problems. We intend to translate our graph problems into formulas and reduce the problem to a logical satisfiability problem. Our goal is to find a formulas Φ with the following property:

> The problem is solvable for a graph G if and only if $G \models \Phi$ i.e. the graph formula Φ is satisfied by our graph G.

Of course, we hope this new problem is solvable in polynomial (or linear) time for graphs of treewidth at most k.

The language we will use for this purpose, is a *monadic second-order* language. Monadic second-order logic is an extension of first-order logic i.e. there are object variables $x, y, \ldots$ and logical connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ as well as quantifiers for object variables $\forall x, \exists x$. Second-order logic, additionally, allows to consider sets of object variables $X, Y, \ldots$ and relate those via the element relation $\in$. Anyway, it not only allows quantification over such sets but quantification over functions and relations as well. In monadic second-order logic, we do not allow the later quantifications and use only quantification over elements and sets of elements.

We want to show and motivate how monadic second-order logic can be used to express graph properties and why this language is appropriate. Whenever we think of graphs, we express properties in terms of vertices and edges. Additionally, we often need sets of vertices or edges. Therefore, we try to use the following first-order language for graphs:

- We have two sorts of object variables: *vertex* variables $v, v_1, v_2, v_3, \ldots$ and *edge* variables $e, e_1, e_2, e_3, \ldots$.
- There is another kind of elements called *set* variables: We have sets of vertices $V, V_1, V_2, V_3, \ldots$ and sets of edges $E, E_1, E_2, E_3, \ldots$. Those can be interpreted as unary relation symbols for the object variables.
- We have a binary relation symbol $\in$ that corresponds to the element operation on objects and their corresponding set variable i.e. if v is a vertex variable and V a vertex set, then $v \in V$ if and only if v is an element of the corresponding set V. For an edge e and a set of edges E, we define $e \in E$ in a similar way.
- Additionally, there is one more relation symbol $Adj(e, v_i, v_j)$ of arity 3 which detects whether the first argument is an edge e from vertex v_i to vertex v_j. For directed graphs, this relation should be symmetric in the second and third component i.e. if there is an edge $e = v_i v_j \in E$, we assure both $Adj(e, v_i, v_j)$ and $Adj(e, v_j, v_i)$ are valid.

Note that for a fixed graph there are just finitely many variables of each sort and finitely many relations. Additionally, the adjacency relation corresponds the actual adjacencies in this concrete graph. The set variables V and E will, from now on, denote the set of all vertex and edge variables respectively for a fixed graph G.

The more formal framework (including all proper definitions of the semantics) is omitted here. We obtain a proper first-order language suitable for graphs. We have atomic formulas $v \in V_i$, $e \in E_i$ and $Adj(e, v_i, v_j)$, and we use the operators $\neg, \wedge, \vee, \leftarrow, \leftrightarrow$ and $\forall v, e$, $\exists v, e$ to obtain the set of all formulas. Furthermore, a graph G satisfies a formula Φ in this language if and only if G contains vertices and edges for each used vertex and edge variable, in a way that the adjacency relation Adj corresponds to actual adjacencies in G.

In this language we can already define the following notions:

- We can, obviously, use the abbreviations $\forall v \in V$, $\exists v \in V$, $\forall e \in E$ and $\exists e \in E$ instead of the the original quantifiers.
- We can express simple adjacency relations such as $v_i v_j \in E$ or $v_i \in N(v_j)$ by

  $$\exists e_k \in E : Adj(e_k, v_i, v_j) = true$$

- We can also define incidence $v_i \in e_j$ by $\exists v_k \in V : Adj(e_j, v_i, v_k) = true$.
- Of course, we can also express the binary subset relation $V_i \subseteq V$

  $$\forall v_j : v_j \in V_i \rightarrow v_j \in V$$

  For $E_i \subseteq E$ it works just the same way.
- We can define the set $N(v_i)$ by $\forall v_j : v_j \in N(v_i) \leftrightarrow v_i v_j \in E$.

- There is a simple formula for a set U being empty: $\forall v : \neg(v \in U)$. We use the abbreviation $U = \emptyset$.

Anyway, this first-order language is not as powerful as we would like our language to be. We are not even able to quantify over arbitrary subsets of vertices/edges – a sincere restriction! Just try to formulate the problems of finding independent or dominating sets of variable size.

A traditional second-order language, which additionally allows quantification over relations, would accomplish this goal. However, we finally want to find some polynomial algorithm to test for satisfiability of those formulas and this works much better for tighter extensions.

So we just introduce one additional feature: quantification over set variables i.e. $\forall V_i, \forall E_i$ and $\exists V_i, \exists E_i$. This is exactly the setting of *monadic second-order logic* (MSOL) for graphs and it allows to formulate a huge class of problems. As we did with the element relation, we use the subset relation to give a scope for set variables in quantifiers:

$$\forall V_i \subseteq V, \exists V_i \subseteq V, \forall E_i \subseteq E, \text{ and } \exists E_i \subseteq E.$$

Example (l-colouring). In this language, we can, for instance, give a formula whose satisfiability is equivalent to the l-colouring problem:

$$\begin{aligned}\exists V_1, \ldots, V_l : &(V_1 \subseteq V) \wedge (V_2 \subseteq V) \wedge \ldots \wedge (V_l \subseteq V) \wedge (V_1 \cup V_2 \cup \ldots \cup V_l = V) \\ &(V_1 \cap V_2 = \emptyset) \wedge \ldots \wedge (V_1 \cap V_l = \emptyset) \wedge \ldots \wedge (V_l \cap V_{l-1} = \emptyset) \\ &\wedge NotAdj(V_1) \wedge NotAdj(V_2) \wedge \ldots \wedge NotAdj(V_l) \\ &\text{where } NotAdj(V) := \forall v_1 \in V, v_2 \in V : \neg(v_1 v_2 \in E).\end{aligned}$$

Indeed, this language is quite useful to find a scheme detecting fixed-parameter tractability of our graph properties. Nevertheless, it is not sufficient to express cardinalities and one needs some further extension of monadic second-order logic. The first such extension was proposed by Courcelle [19] and it is called *counting monadic second-order logic*.

He adds a new type of unary relation symbol working on set variables:

$$card_{n,p}(U), \text{ for } U \subseteq V \text{ or } U \subseteq E, \text{ a number } p \geq 2 \text{ and an integer } 0 \leq n < p.$$

Those are defined to encode $|U| \equiv n \mod p$. Using these relations give additional types of atomic formulas $card_{n,p}(V_i)$ and $card_{n,p}(E_i)$.

Example. With this extension, we can, for instance, ask for an independent set of fixed size K.

$$\exists V_i \subseteq V : card_{K,|V|}(V_i) \wedge (\forall v_1 \in V_i, v_2 \in V_i : \neg(v_1 v_2 \in E)) \tag{3.5}$$

Similarly, we can detect vertex covers or dominating sets of fixed size K.

Example (k-disjoint path problem). Counting monadic second-order logic also gives the possibility to formulate the k-disjoint path problem for $(v_j, w_j), 1 \leq j \leq k$.

$$\begin{aligned}\exists V_1, V_2, \ldots, V_k \subseteq V : & Path(V_1, v_1, w_1) \wedge \ldots \wedge Path(V_k, v_k, w_k) \\ & V_1 \cap V_2 = \emptyset \wedge \cdots \wedge V_1 \cap V_k = \emptyset \wedge \ldots \wedge V_{k-1} \cap V_k = \emptyset\end{aligned}$$

where the formula $Path(P, s, t)$ should denote whether P is a path from s to t. This formula could be constructed recursively via

$$\begin{aligned}Path(\{v\}, s, t) &:= (v = s \wedge v = t) \\ Path(P, s, t) &:= s \in P \wedge card_{1,|V|}(N_G(s) \cap P) \\ &\quad \wedge (\forall v \in N_G(s) \cap P : Path(P \backslash \{s\}, v, t)\end{aligned}$$

As we already mentioned, Courcelle showed in [19] that each problem in (counting) monadic second-order logic can be decided in linear time for graphs of bounded treewidth.

3.3.2 Extended monadic second-order problems

The concept of counting monadic second-order logic was still not powerful enough to describe optimization problems which we already tackled in Section 3.1. Now we consider an approach by Arnborg, Lagergren and Seese [20], which deals with this problem in a slightly different way than Courcelles counting monadic second-order logic.

We define the class of *extended monadic second-order (EMS) problems* to be those problems for which we can find a monadic second-order formula $\Phi = \Phi[X_1, X_2, \ldots, X_n]$ with free set variables $X_1, X_2, \ldots, X_n$ such that the original problem is equivalent to satisfying Φ together with an *evaluation relation* i.e. a propositional formula ϕ built from a special kind of atoms $P_1, P_2, \ldots, P_m$.

The atoms are called *evaluation terms* and they are of type $P = 0$ or $P \leq 0$ with terms P built from rational numbers, arithmetic operators $+, -, \times$ and real-valued variables $|X_i|_j, i = 1, \ldots, n, j = 1, \ldots m$ and $Y_i, i = 1 \ldots, t$. The variable $|X_i|_j$ corresponds to the evaluation of some vertex/edge weight functions f_j on the set corresponding to the variable X_i i.e. there holds

$$|X_i|_j = \sum_{x \in X_i} f_j(x)$$

and the variables $Y_i, i = 1 \ldots, t$ denote possible further numbers given by the problem instance. We define

$$\phi := \phi[|X_1|_1, \ldots, |X_n|_l, Y_1, \ldots, Y_t].$$

For this formula containing fixed weight functions $f_1, \ldots, f_m$ there should, finally, hold:

> The given graph property holds for a graph G and some specification numbers $C_1, \ldots, C_t$ if and only if there exist some subsets of vertices or edges $A_1, \ldots, A_n$, such that $G \models \Phi[A_1, A_2, \ldots, A_n]$ and, additionally, there holds
>
> $$\phi[\sum_{a \in A_1} f_1(a), \ldots \sum_{a \in A_n} f_m(a), C_1, \ldots, C_t].$$

This class of problems also covers the counting monadic second-order problems because we can express cardinalities of sets by using the weight function $f_1(x) = 1, x \in V$. Additionally, we can formulate problems with custom weight functions given by the problem specification e.g. edge distances.

Arnborg et al. do not only cover the case of extended monadic second-order(EMS) problems but allow for extended *monadic second-order extremum problems*. These are stated by not only satisfying the above formulas Φ and ϕ, but simultaneously maximizing an additional evaluation term $F[\sum_{a \in A_1} f_1(a), \ldots, \sum_{a \in A_n} f_m(a), C_1, \ldots, C_t]$. Altogether, these problems are defined the following way:

> The given graph problem holds for a graph G and some specification numbers $C_1, \ldots, C_t$ if and only if there exist some subsets of vertices or edges $A_1, \ldots, A_n$ which maximize $F[|A_1|_1, \ldots, |A_n|_l, C_1, \ldots, C_t]$ within the constraints $\Phi[A_1, \ldots, A_n]$ and $\phi[|A_1|_1, \ldots, |A_n|_l, C_1, \ldots, C_t]$.

Arnborg et al. show in [20] that EMS extremum problems can be decided in polynomial time. If the optimized evaluation term is linear in all variables $|X_i|_j, i = 1, \ldots, n, j = 1, \ldots m$ and there are no further evaluation relations, i.e. $\phi = true$, then the corresponding problem can even be decided in linear time. These problems are called *linear* EMS extremum problems.

The class of EMS (extremum) problems is, indeed, very powerful. Of course, one can express the optimization problems considered in Section 3.1.

Example (Independent set). We take the formula with one set variable U

$$\Phi[U] := (\forall v_1 \in U, v_2 \in U : \neg(v_1 v_2 \in E))$$

which detects whether U is an independent set for G, $\phi = true$ and maximize the evaluation term $|U|_1$ given by the vertex weights of U. There are no further evaluation relations and, of course, the evaluation term is linear. So the problem is a linear EMS extremum problem.

Example (Vertex cover). The vertex-cover problem is a linear EMS extremum problem as well. We choose

$$\Phi[U] := (\forall e \in E : \exists v \in U : v \in e)$$

and maximize the single evaluation term $-|U|_1$ given by the vertex weights of U.

Example (Dominating set). To formalize the dominating-set problem, we define

$$\Phi[U] := (\forall v \in V : (v \in U \vee N(v) \cap U \neq \emptyset))$$

and maximize the evaluation term $-|U|_1$ given by the vertex weights of U. Hence the dominating-set problem is a linear EMS extremum problem too.

3.3.3 A proof sketch

By now, we have seen that many graph problems can be reduced to logical formulas which are in monadic second-order logic. Of course, this is more than just a nice way of notion. Thanks to the tree-like structure of graphs of bounded treewidth, one can construct a (finite) automaton accepting binary trees, which decides whether the formula is satisfiable by a structure representing a binary tree.

The argumentation uses four steps (see [20]):

1. We define a suitable formula Φ in monadic second-order logic for the given graph problem.
2. We transform the formula Φ in monadic second-order logic for graphs into a formula Ψ in another monadic second-order language suited for binary trees. This conversion takes linear time.
3. We construct a finite *tree* automaton, which decides for a given monadic second-order formula Ψ in this new language, whether there is a binary tree S satisfying Ψ, i.e. $S \models \Psi$ – again in linear time.
4. If our problem was an extended monadic second-order (extremum) problem, the automaton has to be modified in linear or polynomial time to give an actual solution that also satisfies Ψ **and** optimizes F.

Altogether, this allows to decide the problem in linear or polynomial time.

The first step was already covered in the last section. We want to give a short sketch what needs to be done in the remaining steps, without claiming to give a full proof. For a full proof, we refer to [20].

Binary trees In the second step, we reformulate our monadic second-order formula Φ. Instead of a monadic second-order formula based on the vertices and edges of our original graph, we obtain another monadic second-order formula on a binary tree constructed from the graphs (nice) tree decomposition $(T, (X_i)_{i \in T})$, which uses just unary predicates.

We start by adding, for each vertex $i \in V(T)$ and each vertex or edge t in the graph G_i, a new neighbour a_t of T. The obtained tree T' is assumed to have no vertices of degree 2, because a degree-2 vertex would correspond to a redundant node of our tree-decomposition.

Next, we define some suitable predicates P_V, P_E allowing to recognize the vertices and edges of G in T'. A similar thing is done for all other predicates necessary to transfer the properties of G to our the new language: We encode the adjacency relation and define additional relations to identify those vertices in T' originating from the same vertex of G (but different parts). In consequence, there is a natural way to transform formulas containing the original predicates into formulas containing these new unary predicates. Such formulas are satisfied by a tree T' if and only if the original formula was satisfied by G.

Finally, we introduce a new predicate P_c corresponding to a 2-colouring of T'. We recursively split up vertices of degree $d \geq 4$ into new ones with degrees $d-3$ and 3 belonging to the same colour class of P_c as the old vertex. In the obtained tree T'', there are only vertices of degree 1 and 3. Splitting up one vertex of degree 3 into one vertex of degree 1 and one vertex of degree 2, finally gives a binary tree. Thanks to the predicate P_c, we can again transform our formula into the new language for binary trees.

If this transformation is done properly, we find a new formula Φ'' in the monadic second-order formula for binary trees (with the predicates mentioned above) in a natural way, such that $G \models \Phi$ if and only if $T'' \models \Phi''$. One can even see that, during this process, we can somehow reformulate our evaluation terms and relations. Therefore, this process is also applicable for EMS-problems.

Tree automata In the third step, the monadic second-order formula for binary trees is decided by a special automaton.

Definition 3.3.1. A *tree automaton* is a quintuple $(S, \Sigma, \delta, s_0, A)$, where S is a set of states, Σ a finite alphabet, $\delta : S \times S \times \Sigma \to S$ a transition function, s_0 the initial state and A a set of accepting states.

Such a tree automaton executes a binary tree labelled with some elements of Σ by assigning states to its nodes. The labels are tuples of 0s and 1s – each entry corresponds to either an unary predicate or a free variable and encodes whether it is valid for the current vertex. The empty tree has state s_0 and, if the child vertices of a vertex v are assigned states $s_1, s_2 \in S$ and the vertex label is l, then the state for v is given by $\delta(s_1, s_2, l)$. Obviously, this execution can be performed in linear time in the number of vertices of the tree.

Before we start with the construction, we eliminate the object variables from our formula by replacing object variables with set variables of cardinality 1. This can be done quite easily and reduces the possibilities for atomic subformulas. For each atomic sub-formula ϕ, one needs to provide an appropriate tree automaton deciding whether there is a (labelled) binary tree satisfying ϕ (see [20, p. 328]).

Given a formula Φ, a tree automaton deciding Φ can be constructed recursively from automata recognizing the subformulas of Φ. Therefore, we first reduce the logical constants

in Φ to $\neg, \wedge$ and $\exists$ by doing a reformulation. We start with the automata for the atomic subformulas and proceed recursively. Let $(S_i, \Sigma, \delta_i, s_0^i, A_i)$ be automata for $\phi_i, i = 1, 2$.

- We obtain an automaton for $\neg\phi_1$ by $(S_1, \Sigma, \delta_1, s_0^1, S - A_1)$.
- An automaton for $\phi_1 \wedge \phi_2$ is given by $(S_1 \times S_2, \Sigma, \delta, (s_0^1, s_0^2), A_1 \times A_2)$ with $\delta((s^1, s^2), (t^1, t^2), a) = (\delta_1(s^1, t^1, a), \delta_2(s^2, t^2, a))$.
- We can build an automaton for $\exists x : \phi_1(x)$ by $(2^{S_1}, \Sigma, \delta_\exists, \{s_0^1\}, A_\exists)$ with $A_\exists = \{S : S \cap A_1 \neq \emptyset\}$ and, if the i-th bit of a represents the free variable variable x,

$$\delta_\exists(S_l, S_r, a) = \{\delta_1(s_l, s_r, b) : s_l \in S_l, s_r \in S_r; b \in \Sigma, a_j = b_j, j \neq i\}$$

These steps can be executed in linear time and, finally, we obtain a suitable automaton.

Extended MS-formulas For extended MS-formulas, some free set variables do remain. We denote the alphabet of our automaton by $\Sigma \times B$, where the elements of Σ correspond to the predicates and the elements of B to the free variables. We then build from the automaton $(S, \Sigma \times B, \delta, s_0, A)$ for the MS-formula Φ a slightly changed automaton. It is similar to the one for existential quantification: we have state set 2^S, language Σ and transition function

$$\delta' : S \times S \times \Sigma \to S : (s_1, s_2, \sigma) \mapsto \{\delta(e_1, e_2, \sigma, b) : e_1 \in s_1, e_2 \in s_2, b \in B\}.$$

Additionally, we add for each state $s \subset S$ and each element $e \in s$ a map m_e, which saves the values of the evaluations during the execution. These maps are sets of matrices and, in the end, we obtain the actual values of the evaluations in polynomial time from those matrices.

For EMS extremum problems, we do not need these maps. It is sufficient to use counters keeping track of the extremum of the objective function over all possible values in certain equivalence classes because of the linearity of the objective function. So for each step, we just need to check all possible combinations and store the maximal values for each equivalence class. This way, one actually obtains a linear-time algorithm.

We again refer to [20] for all details.

3.3.4 Final results

We already mentioned the result by Courcelle:

Theorem 3.3.2. *For each (counting) MS-problem P and each class K of universally bounded treewidth, deciding the problem P for $G \in K$ can be done in linear time if G is given together with a tree-decomposition.*

Arnborg et al. do not even use the result for counting MS-problems but give an even stronger result.

Theorem 3.3.3.
- *For each EMS-problem P and for each class K of graphs of universally bounded treewidth, the problem P for $G \in K$ (with suitable evaluation relations $f_1, \ldots, f_m$ and parameters $C_1, \ldots, C_t$) can be decided in polynomial time with respect to $|G|$ if a tree-decomposition of $G \in K$ is given.*
- *Additionally, EMS extremum problems P can be decided in polynomial time, for each class K of graphs of universally bounded treewidth, if a tree-decomposition of the corresponding graph $G \in K$ is given.*
- *If an EMS extremum problems P is linear and we charge arithmetic operation with constant cost, then P can even be decided in linear time, assuming that a tree-decomposition of the corresponding graph $G \in K$ is given.*

This result covers many graph problems such as the linear EMS-problems of finding vertex covers, dominating sets, independent sets, planar subgraphs, bipartite subgraphs, minimum maximal matching, cliques, maximum cuts and longest paths. An example for a non-linear EMS-problem is finding a partition into perfect matchings or the K-th shortest path for fixed K.

4 Determining Treewidth

In Chapter 3, we observed that considering treewidth and tree-decompositions of graphs is very useful for algorithmic purposes.

We devote this chapter to the actual treewidth computation and the construction of suitable tree-decompositions.

The problem of computing the treewidth of a graph, and – if possible – additionally some tree-decomposition of this width, was already considered by Robertson and Seymour in their original work on tree-decompositions and treewidth [3]. The complexity of their algorithm was stated to be $\mathcal{O}(n^{f(k)})$ i.e. polynomial in $n := |V|$. Their result was improved a few times thereafter.

4.1 Complexity of treewidth computations

Before the concepts of treewidth and tree-decompositions were introduced, a similar problem was considered by Arnborg, Corneil and Proskurowski in [23]: the problem of finding embeddings in a k-tree.

We recall the definition of partial k-trees as subgraphs of k-trees in Section 1.2.3. Since partial k-trees are exactly the graphs of treewidth at most k due to Theorem 1.2.15, the treewidth of a graph G can be computed by finding the minimal number k such that G is a partial k-tree. This number is denoted by $k_t(G)$ in the original paper.

Arnborg et al. showed in [23] that this problem is *NP*-complete.

Theorem 4.1.1 (Arnborg et al.)**.** *The problem of deciding for a graph G and an integer k, whether $k_t(G) \leq k$ is NP-complete.*

This is argued by providing a polynomial reduction to the minimum-cut linear-arrangement problem i.e. the problem of finding an arrangement

$$\tau = (v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{|V|})$$

of the vertices $V(G)$ such that, for each $1 \leq i < |V|$, there are at most k edges between the first i vertices $v_1, \ldots, v_i$ and the later vertices $v_{i+1}, \ldots, v_{|V|}$ of G. This problem has previously been shown to be *NP*-complete and thus evaluating the treewidth is *NP*-hard as well. *NP*-membership is quite easy to see because the arrangement naturally corresponds to an elimination ordering for G and we can, thereby, efficiently check whether the graph is a partial k-tree by counting the edges.

From these results, we conclude that there are – in general – no efficient algorithms for treewidth computations and, of course, constructing an tree-decomposition of optimal width is at least as difficult.

4.2 Exact algorithms

Fortunately, there are some problem variants which are fixed-parameter tractable i.e. can be decided in polynomial and even in linear time with respect to $n := |V|$. The most common variants are obtained by fixing an integer k within the problem specifications i.e. we ask:

> Given a graph $G = (V, E)$ and an integer k, does G have treewidth at most k? Often, we additionally ask for a tree-decomposition of width k.

For the cases $k = 1, 2, 3, 4$, linear-time algorithms had already been given when Bodlaender provided a linear-time algorithm for general k in [24].

This was the final result of a series of improvements which started with an algorithm by Arnborg, Corneil and Proskurowski in [23]. Arnborg at al. provided an $\mathcal{O}(n^{k+2})$-algorithm which we discuss in the following subsection. Furthermore, we want to give a short overview on further improvements and the result by Bodlaender.

Throughout this section, we only consider connected graphs. By Lemma 1.1.8, this is no essential restriction since

$$tw(G) = \max_{\text{components } C \text{ of } G} tw(C).$$

4.2.1 A polynomial algorithm

The first algorithm by Arnborg, Corneil and Proskurowski ([23]) was actually an algorithm for partial k-trees. We already mentioned in Section 4.1 that the treewidth can be evaluated by finding minimal embeddings in k-trees.

Arnborg and al. used the following result:

Lemma 4.2.1. *A connected graph G of size at least $k+2$ is a partial k-tree if and only if there exists a k-element separator C_i such that all induced subgraphs $C^i_j := G[C_i \cup C^j]$ with components $C^1, \ldots, C^j$ of $G - C_i$ are partial k-trees.*

The original proof used inductive arguments constructing those separators. We use the equivalence of partial k-trees and graphs of treewidth at most k from Theorem 1.2.15 and show the following variant:

Lemma 4.2.2. *A graph G (of size at least $k+2$) has treewidth at most k if and only if there exists a k-element separator C_i such that the graphs C^i_j, defined in Lemma 4.2.1, have treewidth at most k.*

Proof. We take a reduced tree-decomposition $(T, (X_t)_{t \in V(T)})$ of G having width $l \leq k$ and denote one part containing $l+1$ elements by $X_{t_1}, t_1 \in V(T)$. If $l < k$ we choose $k-l$ vertices of $V(G) \backslash X_{t_1}$ and add them to each vertex set X_t. Of course, this gives a new tree-decomposition $(T, (X_t)_{t \in V(T)})$ of width $|X_{t_1}| - 1 = k$.

t_1 has at least one neighbour t_2 in T and, because we are reduced, the set $S = X_{t_1} \cap X_{t_2}$ is a separator of size at most k for G by Lemma 1.1.13.

We extend this separator S by vertices of X_{t_1} until it has size exactly k. Of course, the obtained set C_i is still a separator because the remaining nodes in $X_{t_1} \backslash S$ are not connected to the component containing $X_{t_2} \backslash S$.

For each component C of $G - C_i$, the restriction of the original tree-decomposition gives a tree-decomposition for $G[C \cup C_i]$. However, C_i is entirely contained in the part X_{t_1} and adding edges inside X_{t_1} does not conflict with (T2). This allows us to use this tree-decomposition for the corresponding graph C_j^i as well. Therefore, $tw(C_j^i) \leq k$ and the given condition is necessary.

On the other hand, the tree-decompositions of the C_j^i each have some part containing the entire set C_i. If we connect the trees given by their tree-decompositions with a common root node r and define the vertex set $X_r := C_i$, we already get a tree-decomposition of width at most k for G, since all other vertices are disjoint. □

The algorithm given by Arnborg et al. simply considers all k-element vertex sets S of G and tests whether they are separators. For each separator C_i, we additionally compute the subgraphs $C_j^i, j = 1, \ldots, l$ induced by S and the vertices of one of the l components of $G - S$. Furthermore, we add edges to make the subgraph induced by C_i complete – this is necessary to make some "gluing" possible throughout the algorithm.

Note that it is possible to decide whether a set S is a separator and, simultaneously, compute the components C_j^i in time $\mathcal{O}(n^2)$ with a simple depth- or breath-first search algorithm. Evaluating all separators and resulting subgraphs, therefore, takes $\mathcal{O}(n^{k+2})$.

It remains to decide whether all graphs C_j^i, for some separator C_i, have treewidth at most k. Fortunately, there is a quite useful criteria which allows a bottom-up evaluation.

Lemma 4.2.3. *A subgraph C_j^i has treewidth at most k, if there exists a vertex $v \in V(C_j^i)$ and some k-vertex separators $C_m \neq C_i$ contained in $C_i \cup \{v\}$ such that some of the components $C_l^m - C_m$ with $C_l^m \subset C_j^i$ and treewidth of C_l^m at most k partition $C_j^i - C_i - \{v\}$.*

Proof. If C_j^i has treewidth at most k, there exists a reduced tree-decomposition of width k by Lemma 1.1.5. C_i is entirely contained in some vertex set X_t because C_i is complete. So either $X_t = C_i \cup \{v\}$ for some $v \in V(G)$ or $X_t = C_i$. First we handle the second case: We extend X_t by some vertex $v \in X_s \backslash X_t$ for some node s adjacent to t. Its existence is guaranteed since we use reduced tree-decompositions and we additionally yield, for all

neighbours s of t that

$$X_t \backslash X_s \neq \emptyset. \tag{4.1}$$

We know that the graphs G_k induced by the different branches of G at t, result in graphs $G_k - X_t$ which are disjoint and not connected by Lemma 1.1.12. Their vertices partition $G - X_t$ (by Property (T1)) and thus they are exactly the components of $G - X_t$. For $X_t = C_i \cup \{v\}$, the components of $G - C_i$ are obviously obtained from the components of $G - X_t$ by merging those components adjacent to v. Therefore, the components $G_k - X_t$ with vertices adjacent to v partition $C_j^i - C_i - \{v\}$.

It remains to show that each such component $G_k - X_t$ with vertices adjacent to v actually results from removing some separator $C_m \neq C_i$. In the branch of G at t which induces G_k there is a unique neighbour s of t. For its part X_s, we have that $X_t \backslash X_s \neq \emptyset$ by Property (4.1). Let $w \in X_t \backslash X_s$, then w is not adjacent to any node in the component $G_k - X_t$ due to Property (T2) and Property (T3). Obviously, $w \neq v$ since v does not share this property. Since $G_k - X_t$ has no outgoing edges to vertices outside $X_t \backslash \{w\}$, it appears as some component $C_l^m - C_m$ of $G - C_m$ for $C_m \cup \{w\} = C_i \cup \{v\}, v \neq w$. his shows that the above condition is necessary.

If such sets C_l^m with the property stated above exist, we take a tree-decompositions of those C_l^m. We root each one of them at the node n_l^m containing the complete subgraph $C_m - \{v\}$ and we glue them together at a common root r node with set $X_r = C_i \cup \{v\}$. We just have to check Property (T3) for the vertices in $C_i \cup \{v\}$ to see that this is a valid tree-decomposition for C_j^i because having a partition of $C_j^i - C_i - \{v\}$ implies all others. All the nodes n_l^m contain v and the root node glues together the subtrees containing $\{v\}$. If some node n_l^m does not contain a vertex $w \in C_i$, then C_l^m cannot contain it either because $C_l^m - C_m \leq C_j^i - C_i - \{v\}$. Again, the subtree containing w is connected. This gluing process thus gives a tree-decomposition for C_j^i of width k. □

The graphs C_l^m in the previous lemma are strictly smaller than C_j^i and we can use this to do a bottom-up evaluation. This gives a procedure **testTreewidth**(*integer* k)stated in Algorithm 18.

We already noticed that finding separators can be done in $\mathcal{O}(n^{k+2})$ and so does the first loop.

Bucket-sorting is linear in the number of graphs C_j^i. We can at least bound the number of graphs C_j^i by $(n-k) \cdot n^{k+1}$ because there are at most n^k separators and each one has at most $n - k$ resulting components.

In the second loop, testing all vertices $v \in C_j^i$ adds a factor $\mathcal{O}(n)$ to these costs. Checking the exit condition does not increase the loops time complexity. The additional effort for evaluating the union in the innermost loop is $\mathcal{O}(n)$ because – for a proper implementation we can access the (at most k) possible separators C_m in constant time each and find the subgraphs C_l^m in $\mathcal{O}(n)$ for each separator. The last estimation works because the total

Algorithm 18 testTreewidth(k): tests whether the input graph G has treewidth at most k

for all k-element vertex sets S **do**
  if S is a separator **then**
    Save $C_i = S$ in an array $(C_i)_{i \in I}$ and the resulting subgraphs as list $(C_j^i)_{j \in J_i}$.
  end if
end for
Bucket-sort the graphs C_j^i by increasing size
Store the answer *true* for all C_j^i with size $k+1$
for all C_j^i with increasing order **do**
  for all $v \in V(C_j^j)$ **do**
    $U = \emptyset$
    for all C_l^m in $(C_j^i - C_i) \cup C_m$ with $C_m \subset C_i \cup \{v\}$ and answer *true* **do**
      $U = U \cup C_l^m$
    end for
    if U contains $C_j^i - C_i$ **then**
      set answer for C_j^i to *true* and continue with next C_j^i
    end if
  end for
  if Answer for C_j^i not yet set **then**
    Set answer for C_j^i to *false*
  end if
  if for this i all C_j^i have answer *true* **then**
    witness found, return *true*
  end if
end for
no witness found; return *false*

number of vertices in $\bigcup_j (C_l^m - C_m)$ is bounded by n. Checking the union again does not increase the time complexity and altogether we find that the time complexity for this algorithms is $\mathcal{O}(n^{k+2})$.

We already argued above that this algorithm is correct. Therefore, the above method indeed gives a $\mathcal{O}(n^{k+2})$ algorithm for testing whether the treewidth is at most k. In the original work [23], there is, of course, no construction of an actual tree-decomposition but the option of finding an embedding into a k-tree along the way. Anyway, we modified the proofs for the two lemmata above to indicate that the algorithm can be modified to construct tree-decompositions. One would just need to start with the construction for small C_j^i and glue those together along the way.

4.2.2 A two-step algorithm

The time complexity of treewidth computations has, afterwards, been reduced by a new kind of two-step algorithm.

First, a tree-decomposition of width bounded by some constant depending on the fixed value k is evaluated. In the original version by Robertson and Seymour in [8], the treewidth bound was $4k$ and the evaluation could be executed in $\mathcal{O}(n^2)$ time.

This first step was improved a few times: Lagergren [25] and Reed [26] gave sequential algorithms in $\mathcal{O}(n \log^2(n))$ and $\mathcal{O}(n \log(n))$, respectively – all based upon finding some kind of *balanced* separators. Those algorithms can even be efficiently parallelized to use sub-linear time. We skip the details of those algorithms here and instead sketch another approach for the first step by Bodlaender in Subsection 4.2.3.

The second step of the algorithm was, at first, an abstract one using graph minor theory. This abstract algorithm uses the *obstruction* set of forbidden minors for the minor-closed class of graphs of treewidth at most k and checks whether the graph contains one of these minors. For minor-testing, there exist linear-time algorithms developed with a dynamic-programming-approach (like in Chapter 3).

Later, it was discovered that the second step can be done without the use of graph minors. Lagergren and Arnborg [27] and, independently, Bodlaender and Kloks [28] gave explicit algorithms for the second step which take a tree-decomposition of treewidth linear in k and decide whether the graphs treewidth is at most k. Bodlaender and Kloks also gave instructions, how to actually find tree-decompositions of width k.

Since this algorithm is actually used by the linear-time algorithm introduced in Subsection 4.2.3, we choose to give a short characterization of the algorithm by Bodlaender and Kloks in [28].

The main result is the following one:

Theorem 4.2.4 (Bodlaender and Kloks, 1996)**.** *For all k and l, there exists a linear-time algorithm that – given a graph $G = (V, E)$ together with a tree-decomposition $(T, \mathcal{X})$ of*

treewidth at most l – determines whether the treewidth of G is at most k and, if so, finds a tree-decomposition of G with treewidth at most k.

Bodlaender and Kloks start by turning the tree-decomposition of width at most l into an equivalent nice tree-decomposition $(T, (X_i)_{i \in V(t)})$ of same treewidth. Subsequently, a dynamic-programming algorithm similar to the ones in Chapter 3 is executed.

As solutions, we define any tree-decomposition for G. Partial solutions for subgraphs G_i induced by a vertex set V_i are, again, solutions for G_i i.e. tree-decompositions for G_i. Extensions are defined in a natural way i.e. such that the restriction to G_i equals the original partial solution.

Defining suitable characteristics is a bit more challenging. We start with any partial tree-decomposition $Y = (T^Y, \mathcal{S}^Y)$ for a subgraph G_i of G. Whenever we want to extend it, we are interested in the distribution of the nodes containing vertices of X_i. Those nodes have to contain all potential neighbours of new vertices in $V(G) \backslash V_i$ to archive Property (T2) and, therefore, some of them might need to be extended by additional vertices.

Definition 4.2.5. The pair $(T^Y, \{S \cap X_i : S \in \mathcal{S}^Y\})$ for X_i is called *restriction* of the partial tree-decomposition $Y = (T^Y, \mathcal{S}^Y)$.

It is quite obvious that the restriction is a tree-decomposition for the subgraph $G[X_i]$. Nevertheless, it might contain quite many nodes and we want to restrict to the essential structure – just as we did for the reduced tree-decompositions mentioned in Chapter 1. Here we recursively remove those leaves which are already fully contained in the vertex set of their parent node and thus not contain any new vertex. Additionally, we replace subsequent nodes of degree 2 by single nodes. This just gives another tree-decomposition of $G[X_i]$.

Definition 4.2.6. The *trunk* of a partial tree-decomposition $Y = (T^Y, \mathcal{S}^Y)$ is the tree $\mathcal{T}$ obtained from T^Y by recursively removing leaves which do not contain any vertex v not contained in any other vertex set of $\mathcal{S}^Y$. Additionally, we remove possible chains of vertices with degree 2 by replacing them with a single edge e.

Each edge $e \in \mathcal{T}$ corresponds to some path $v_0 v_1 \dots v_n$ in T^Y containing the inner vertices of degree 2 which were removed along e in the construction of the trunk. We consider the sequence of corresponding parts of the restriction

$$S_0^Y \cap X_i, S_1^Y \cap X_i, \dots, S_n^Y \cap X_i$$

and remove possible subsequent identical vertex sets. The remaining list

$$Z_e := (S_{t_0}^Y \cap X_i, S_{t_1}^Y \cap X_i, \dots, S_{t_p}^Y \cap X_i)$$

with $0 = t_0 < t_1 < \dots < t_p = n$ is called *interval model* for the edge e.

The pair $(\mathcal{T}, (Z_e)_{e \in E(\mathcal{T})}$ is called *tree model* for Y.

These tree models define our equivalence classes of partial tree-decompositions.

The trunk of any partial tree-decomposition of width k has a very nice property: Since we removed leaves not containing any new vertex and we just use nodes of X_i, there are at most $k+1$ leaves. There are no degree-2 vertices and, for trees, there holds $|E| = |V| - 1$. This, immediately, yields that

$$2 \cdot |E| = 2 \cdot |V| - 2 = \sum_{v \in V} d(v) \geq l + 3 \cdot (|V| - l)$$

where $l \leq k+1$ denotes the actual number of leaves, and thus $|V| \leq 2l-2 \leq 2k$. So, both the number of trunk vertices and the number of trunk edges are linear in k. Bodlaender and Kloks additionally showed that the number of subsets in each set Z_e can be bounded by $2k+3$ and, consequently, the number of possible tree models is only dependent on k. Hence, there is only a constant number of equivalence classes with respect to the number of nodes in the original graph G.

To finally select *good* partial solutions, we need to keep track of the sizes of the vertex sets in the partial tree-decomposition for G_i. Intuitively, in each equivalence class, those tree-decompositions with smaller vertex sets are better since they result in extensions with smaller vertex sets. The goal of minimizing the set sizes is archived through an additional valuation component (similar to the algorithms for optimization problems in Section 3.2).

Definition 4.2.7. Let $Y = (T^Y, \mathcal{S}^Y)$ be a partial tree-decomposition with tree model $(\mathcal{T}, (Z_e)_{e \in E(\mathcal{T})})$. For each edge e and interval model $Z_e = (S^Y_{t_0} \cap X_i, S^Y_{t_1} \cap X_i, \ldots, S^Y_{t_p} \cap X_i)$ we define $[y_e] := [y_e^1, \ldots, y_e^p]$ with (for $0 \leq i \leq p-1$)

$$y_e^{i+1} = (|S_{t_i}|, |S_{t_i+1}|, \ldots, |S_{t_{i+1}-1}|)$$

This array of lists is too long to actually use it. It might even contain duplicates. This is why Bodlaender and Kloks use some nice variant of the lists defined above. They replace each list $[y_e]$ by some new list $\tau[y_e]$, which is obtained by taking each list $y_e^i = (a_i)_{i=0}^n$ separately and recursively applying the following operation:

> If there is any sub-sequence $(a_i, \ldots, a_j), i < j+1$ with either $a_i \leq a_k \leq a_j$ for all $i < k < j$ or $a_i \geq a_k \geq a_j$ for all $i < k < j$, replace it by (a_i, a_j).

Since replacing those sub-sequences does not influence the order of the remaining elements, this uniquely defines a new list $\tau[y_e]$, named the *typical list* for e. We not only replace possible subsequent duplicates; the above ascending and descending subsequences e.g. also cover the case of adjacent nodes with some subset relation for the corresponding vertex sets. They also define a very nice relation on such integer lists.

Definition 4.2.8. We use the notation $[y^*] \in E([y])$ for some arrays of integer lists $[y] = (y_1, y_2, \ldots, y_n), [y^*] = (y_1^*, y_2^*, \ldots, y_n^*)$, if each integer list y_i^* is obtained from y_i by subsequently taking each element at least once i.e. if $y_i = (x_1^i, x_2^i, \ldots, x_n^i)$, $y^* *_i$ is of form $(x_1, x_1, \ldots, x_1, x_2, \ldots, x_2, \ldots x_n, \ldots, x_n)$.

Using this notation, we define a relation $<_L$ on such integer lists by $[x] <_L [y]$ if and only if there exist $[x^*] \in E([x]), [y^*] \in E([y])$ of equal length n such that for all $1 \leq i \leq n$ there holds $x_i^* \leq y_i^*$.

Bodlaender and Kloks observe that this relation on the lists $\tau[y_e]$ for partial solutions with the same tree model indeed rates its quality. For reference, see [28, Chapter 3].

A dynamic programming algorithm finally computes the tables containing the best valuation of each equivalence classes in bottom-up order:

- At leaf nodes, there is just one tree-decomposition where each leaf contains a new vertex.
- At join nodes, we somehow glue together pairs of partial solutions while, of course, checking whether the set sizes stay at most $k + 1$.
- At forget nodes, we need to remove the corresponding vertex from each set in Z_e for all e and re-compute the tree model as well as the typical list.
- At introduce nodes, they exploit that there are only constantly many equivalence classes for the new node and they check the restriction for each one.

We end up with the corresponding table for the root node of the tree-decomposition. Of course, if this table is empty, then the treewidth is $> k$; otherwise one can compute some tree-decomposition of width at most k from this table. We skip all details here and refer to the original paper by Bodlaender and Kloks [28].

4.2.3 A linear time algorithm for graphs of bounded treewidth

As we observed in the previous subsection, the second step was known to be tractable in linear time soon and, for a long time, the first step remained the crucial one for complexity issues. Bodlaender finally managed to improve this step to take linear time with respect to the number of vertices. This immediately gave a linear-time algorithm for the combined decision and construction variant on general graphs.

While previous algorithms had been based on well-chosen separators, Bodlaender observed that bounded-treewidth graphs only have few vertices with large degree. We want to give a rough sketch on the main ideas of his algorithm. The full argumentation can be found in [24].

For his argumentation, he uses some fixed integer value d depending on the treewidth k.

$$d := 2k^3 \cdot (k+1) \cdot (4k^2 + 12k + 16)$$

Definition 4.2.9. A vertex of degree at most d is called *low-degree vertex*, while vertices with degree at least d are called high-degree vertices.

A vertex v is called friendly if it is a low-degree vertex and adjacent to another low-degree vertex.

We start with a simple observation which is obtained using a special kind of tree-decomposition called *smooth* tree-decomposition: For smoothed tree-decompositions, we demand $|X_i| = k + 1$ for all nodes i and $|X_i \cap X_j| = k$ for all edges ij. This demand is admissible since we can recursively apply the following operations without violating the Axioms (T1), (T2) and (T3).

- If $X_j \subseteq X_i$, we contract the edge ij to some new node k with $X_k = X_i$.
- If $X_j \not\subseteq X_i$ but $|X_i| < k + 1$, we add a vertex $v \in X_j \backslash X_i$ to X_i increasing its size.
- If $|X_i| = |X_j| = k+1$ but $|X_i - X_j| > 1$, we subdivide the edge ij by introducing a new node k with vertex set $X_k = X_i \backslash \{v\} \cup \{w\}$ where $v \in X_i \backslash X_j$ and $w \in X_j \backslash X_i$. This decreases the number of overlaps by 1.

We now show the following lemma:

Lemma 4.2.10. *If $G = (V, E)$ has treewidth at most k, then $|E| \leq k|V| - \frac{1}{2}k(k + 1)$.*

Proof. We use a smooth tree-decomposition for G; If there is just one bag, there holds $n = k$ and $|E| \leq \dfrac{n(n-1)}{2} = kn - \dfrac{1}{2}k(k + 1)$. If there are at least two parts, we consider the graph induced by removing one leaf i of the tree-decomposition. Since our tree-decomposition is smooth, there is a unique vertex $v \in X_i$ which is not contained in any other vertex set. This vertex is connected to at most k neighbours in X_i by Property (T2). Applying the induction hypothesis for $G' = G - \{v\}$, we yield

$$|E| \leq |E(G')| + k \leq k|V(G')| - \frac{1}{2}k(k + 1) + k = k|V(G)| - \frac{1}{2}k(k + 1). \qquad \square$$

We see that a violation of the above inequality suffices to see that the treewidth is strictly greater than k. Additionally, we derive a nice bound for the number of high-degree vertices.

Lemma 4.2.11. *There are fewer than $\dfrac{2k}{d} \cdot |V|$ high-degree vertices in a graph $G = (V, E)$ with treewidth k.*

Proof. If there are l high-degree vertices, there are at least $l \cdot \frac{d}{2}$ edges. Using Lemma 4.2.10, we yield

$$l \cdot d \leq 2|E| \leq 2k|V| - k(k + 1) < 2k|V|. \qquad \square$$

Bodlaender finally obtains the following result.

Theorem 4.2.12. *For every graph $G = (V, E)$ with treewidth at most k, at least one of the following properties holds:*

1. *G contains at least $|V|/(4k^2 + 12k + 16)$ friendly vertices.*

2. *The improved graph $G' = (V, E')$, i.e. the graph obtained from G by adding edges between vertices with at least $k+1$ common neighbours of degree at most k in G, contains at least $|V|/(8k^2 + 24k + 32)$ simplicial vertices having degree at most k in G.*

We omit this slightly technical proof here and refer to [24, Section 4]. The structure of the algorithm is a recursive one. For small graphs, the algorithm uses an arbitrary algorithm – it does not influence the complexity anyway. In each of the above cases, a reasonable modified graph G' is computed and the algorithm is called recursively for G'. Afterwards, the tree-decomposition for G' needs to be modified to yield another one for G.

In the first case, Bodlaender exploits the following fact.

Lemma 4.2.13. *If there are n_f friendly vertices in G, any maximal matching of G contains at least $\frac{n_f}{2d}$ edges.*

Proof. For some maximal matching M, any friendly vertex v must either be an endpoint of an edge in M or – by maximality – the friendly (low-degree) neighbour of v is endpoint of an edge in M. For each edge in M, at most $2d$ friendly vertices are incident or adjacent to a friendly incident vertex. This gives the correspondence $2d|M| \geq n_f$. □

The algorithm computes a maximal matching using a greedy $\mathcal{O}(|V| + |E|) = O(|V|)$-algorithm. Afterwards, all edges in M are contracted and we yield a graph $G' = (V', E')$ and some map $f_M : V \to V'$ which maps vertices in G to their resulting vertices after the contractions. One easily observes the following lemma.

Lemma 4.2.14. *For a graph G with some maximum matching M and a graph G', as defined above, there holds:*

> *If $(T, (X_i)_{i \in V(T)})$ is a tree-decomposition of G of width k, then the pair $(T, (Y_i)_{i \in V(T)})$ with $Y_i = \{v \in V : f_M(v) \in X_i\}$ is a tree-decomposition for G' of width at most $2k + 1$.*

We then recursively compute a tree-decomposition for G' and extend it to G in a natural way. This yields a tree-decomposition for G of width at most $2k + 1$. Finally, we use the algorithm by Bodlaender and Kloks stated in the previous subsection to test whether the treewidth is at most k. Of course, since the treewidth of G and G' correspond, we also detect if G has treewidth at least $k + 1$.

In the second case of Theorem 4.2.12, we have many friendly vertices. We compute the improved graph mentioned in Theorem 4.2.12 i.e. we make vertices with at least $k + 1$ common neighbours of degree at most k in G adjacent. There holds:

Lemma 4.2.15. *Any tree-decomposition of G of width k is also a tree-decomposition for the improved graph of G, and vice versa.*

Proof. If $(T, (X_i)_{i \in V(T)})$ is a reduced tree-decomposition for G and v, w share $k+1$ neighbours N, we apply Lemma 1.1.14 for the set $W = \{v, w\}$.

Assume there is some edge $t_1t_2 \in E(T)$ such that $v, w \notin X_{t_1} \cap X_{t_2}$ and the sets X_{t_1}, X_{t_2} were separated by $X_{t_1} \cap X_{t_2}$ in G. The separating set has to contain all $k+1$ vertices in N to separate v and w which implies $|X_{t_1}| = |X_{t_2}| = k+1$. Of course, this is a contradiction to the fact that we used a reduced tree-decomposition and thus there has to exist some part containing the set $\{v, w\}$. Making v, w adjacent does, therefore, not increase the treewidth.

The second implication is trivial since G is a subgraph of G'. □

The algorithms for graphs with less friendly vertices starts by computing the improved graph of G. In the next step the simplicial vertices stated in Theorem 4.2.12 are considered.

Definition 4.2.16. A vertex $v \in V$ which is simplicial in the improved graph of G is called *I-simplicial*.

Of course, the improved graph is not allowed to have simplicial vertices of degree at least $k+1$ since such vertices would result in a $k+2$ clique. If the treewidth is at most k, there obviously are no cliques of size $k+2$ and thus the restriction for the degree of I-simplicial vertices is natural. The algorithms first checks this condition and the lower bound on the number of such vertices stated in Theorem 4.2.12.

If all these tests are negative, we remove the I-simplicial vertices from the improved graph and, recursively, apply the algorithm for the new graph. We notice the following fact.

Lemma 4.2.17. *If $(T, (X_i)_{i \in V(T)})$ is a tree-decomposition of the graph G' obtained by removing the I-simplicial vertices, then for all I-simplicial vertices v there exists an node $i \in V(T)$ with $N_G(v) \subseteq X_i$.*

Proof. Since each such vertex v is simplicial, their neighbourhood $N_G(v)$ forms a clique in the improved graph. However, Bodlaender states that I-simplicial vertices can not be adjacent in G and thus their neighbourhood does not contain any other I-simplicial node. Therefore, all neighbours of v have to be contained in some vertex set X_i of this tree-decomposition. □

For each tree-decomposition of the graph G' without those simplicial vertices, we still have some part X_i with $N_{G'}(v) \subseteq X_i$ for each I-simplicial vertex v. By appending one new node adjacent to i in the tree T with vertex set $N_G(v) \cup \{v\}$ we already find a tree-decomposition of width k for the original graph. If the treewidth of the subgraph G' of G is larger than k, then so is the treewidth of G and the algorithm returns *false*.

Implementing the ideas from above in some suitable way, Bodlaender actually yields the stated linear-time algorithm for the second step.

Theorem 4.2.18. *For all $k \in \mathbb{N}$, there exists a linear-time algorithm finding a tree-decomposition of width at most k or recognizing that the treewidth is at least $k+1$.*

The non-recursive steps can easily be observed to be linear. For many friendly vertices, we observe that the matching is sufficiently large and thus the size of the obtained graph can be bounded from above. For fewer friendly vertices, Bodlaender shows that there are sufficiently many I-simplicial vertices and again the size of the obtained graph is bounded from above. Both bounds are sufficiently tight to yield a linear runtime for this recursive algorithm.

Bodlaender also estimates the constant factor hidden in the $\mathcal{O}$-notation: After all this effort it is, sadly, still exponential in k^3.

There is one more improvement of this algorithm by Perković and Reed in [29], which, in case the algorithm returns that the treewidth is $\geq k$, also returns a subgraph G' with treewidth $\geq k$ along with a tree-decomposition of G' of width $\leq 2k$. We do not consider this improvement here.

4.3 Approximating treewidth

Since exact algorithms for treewidth and tree-decompositions are still time-consuming, it is, additionally, important to consider approximation algorithms which do not guarantee to give an optimal width for their tree-decomposition but still give reasonable results for the corresponding applications.

We just want to state a few results which were mentioned by Bodlander and Koster in [16, Chapter 4].

4.3.1 Approximation algorithms

We observed in Section 1.2.2 that the treewidth is bounded from below by $\omega(G) - 1$, where $\omega(G)$ denotes the graphs maximum clique size.

To compute the treewidth for chordal graphs, one would possibly start by computing a perfect elimination ordering $v_1, \ldots, v_n$ and then recursively building a tree-decomposition by adding simplicial vertices.

Since in each step the vertex v_i is simplicial in $G[V \backslash \{v_1, \ldots, v_{i-1}\}]$, its neighbourhood $N(v_i)$ induces a clique. By Lemma 1.1.15, this clique is contained in some vertex set X_i of any tree-decomposition for $G[V \backslash \{v_1, \ldots, v_{i-1}, v_i\}]$ and by adding one new part adjacent to X_i with vertex set $X_j = N(v_i) \cup \{v_i\}$, we yield a valid tree-decomposition for $G[V \backslash \{v_1, \ldots, v_{i-1}\}]$. That way we, recursively, get a tree-decomposition of optimal width for G. Obviously, the width corresponds to the maximum degree of any vertex v_i at the time of its removal.

Of course, this does not work for arbitrary graphs. Anyway, we can try to construct a chordal supergraph G' of G by *chordalization* (or triangulation) and exploit that

$$tw(G) = \min\{\omega(H) - 1 | G \subseteq H; H \text{ chordal } \}$$

as stated in Corollary 1.2.11.

According to [16, Chapter 4], many heuristics actually exploit this very simple fact. We just need to choose an arbitrary elimination ordering for G. There are some different approaches to do this: One could e.g. chordalize a graph in a greedy way by repeatedly selecting vertices having a neighbourhood which is already nearly complete and adding those edges for chordalization (called *greedy fill-in*). Another popular example is *minimum degree fill-in*, where only vertices of minimum degree are selected for elimination.

Other algorithms searching for possibly suboptimal solutions use the concept of separators. A nice survey on all these methods is given in [30].

Other approximation algorithms even give some guaranty for the quality of the obtained solution. For instance, there is an early algorithm by Bodlaender and al. in [31], which outputs a tree-decomposition of width $\mathcal{O}(k \log(k))$ and takes polynomial time. There are other $\log(k)$-approximation algorithms i.e. algorithms guaranteeing treewidth $\mathcal{O}(k \log(k))$ in polynomial time as well.

It would, of course, seem desirable to find algorithms with a constant approximation factor. However, it is not known whether there are such algorithms running in polynomial time both in n and the treewidth k ([16]).

Additionally, people have tried various other attempts which are common for known NP-problems: There are some heuristics which locally improve solutions by replacing large vertex bags while preserving the axioms. Of course, these can be used to improve solutions found by any other algorithm as well as for starting with the trivial tree-decomposition consisting of just one bag.

Other attempts use meta-heuristics e.g. simulated annealing of genetic algorithms. We do not consider those approaches here.

4.3.2 Lower bounds

Another way to tackle our problem is to find some good lower bounds. Of course, this does not give a feasible tree-decomposition but it might still give a good estimate. This could prove useful e.g. to decide which instances are actually to complex to deal with.

Additionally, good lower bounds are used in another common approach for NP-problems, namely branch-and-bound algorithms.

Most common methods for lower bounds are based on (minimal or average) degrees. Since those degree values do not appear to be closed under subgraphs or minors, they are often substantially improved by evaluating these bounds for all (or at least many)

subgraphs or minors and combining those results. Another approach for finding a lower bound might be the concepts of brambles which was briefly mentioned in Section 1.2.5 and seems useful for close-to-planar graphs.

A survey on approaches for lower bounds is given in [32].

5 An Alternative Approach: Reduction Algorithms

Throughout this section, we present one more interesting method to solve decision and optimization properties on graphs of bounded treewidth. It is based on graph reduction i.e. we try to replace graphs of type $H \oplus K$ by a smaller graph $H' \oplus K$ without changing the answer to our problems. This method is of special interest since it does not use an actual tree-decomposition and, therefore, skips the complexity issues of treewidth-computations discussed in Chapter 4.

As the treewidth concept itself, the idea for this approach was taken from algorithms working for special graph classes. A reduction algorithm for some series-parallel graphs could, for instance, reduce the original graph to a single edge by iteratively deleting parallel edges and removing series of subsequent vertices.

A first algebraic theory for graph reductions was given by Arnborg et al. In [33], they show that for special kinds of graph properties (including MS-definable problems as considered in Section 3.3) a $\mathcal{O}(n)$-time decision algorithm using more than linear space exists. Each such algorithm is characterized by defining a suitable set of reduction rules. This chapter deals with such reduction systems and indicates how to find them.

We basically follow the considerations of Bodlaender and Antwerpen-de Fluiter in [9]. We first give a general definition of graph reductions and, afterwards, consider the case of bounded treewidth graphs.

Throughout this section, we use a slightly different representation of graphs $G = (V, E)$: For each vertex $v \in V$, we save an *adjacency* list i.e. a doubly-linked list of all incident edges and a record containing a pointer to the first and last edge of this list. This way, edges are represented twice – we link those with some more pointers. In total, we store $2 \cdot |E|$ edges and additionally $2 \cdot |E| + 2 \cdot |V|$ pointers which still gives linear space complexity. We call such a representation an *adjacency list representation* of G.

5.1 Reduction systems

In this section, we want to give a short insight on graph reductions. We, again, use the concept of terminal graphs as introduced in Section 2.2.2.

Definition 5.1.1. A *reduction rule* r is an ordered pair of l-terminal graphs H_1,H_2 for some $l \geq 0$. Such a reduction rule can be *applied* to a graph G in the following sense:

If G contains a l-terminal subgraph G_1 isomorphic to H_1 and $G = G_1 \oplus G_3$, then we replace G_1 by a graph G_2 isomorphic to H_2 and yield a new graph $G' = G_2 \oplus G_3$. Such an application is called a *reduction*, and it is denoted by $G \xrightarrow{r} G'$.

Given a set of reduction rules $\mathcal{R}$, we denote the application of an $r \in \mathcal{R}$ by $G \xrightarrow{\mathcal{R}} G'$. A graph G is *irreducible* for $\mathcal{R}$ if no further reduction is possible with any reduction rule in $\mathcal{R}$.

5.1.1 Reduction systems for graph properties

We usually want reduction rules to have some nice properties with respect to a fixed graph property P.

Definition 5.1.2. We define the following properties:

- A set of reduction rules $\mathcal{R}$ is called *safe* for P if, whenever $G \xrightarrow{\mathcal{R}} G'$, the equivalence $P(G) \Leftrightarrow P(G')$ holds.
- A set of reductions rules $\mathcal{R}$ is called *complete* for P if the set of irreducible graphs for which P holds is finite.
- A set of reductions rules $\mathcal{R}$ is called *decreasing* for P if, whenever $G \xrightarrow{\mathcal{R}} G'$, we have $|G| > |G'|$ i.e. G' has less vertices.

A *reduction system* for P is a pair $(\mathcal{R}, \mathcal{I})$ where $\mathcal{R}$ is a finite, safe, complete and decreasing set of reduction rules for P and $\mathcal{I}$ is the (finite) set of irreducible graphs for which P holds.

A reduction system characterizes P:

> P holds if and only if there is a sequence of reduction rules in $\mathcal{R}$ which reduces G to one of the graphs in $\mathcal{I}$.

This immediately gives an algorithm of deciding P. This algorithm consists of performing at most $n := |V|$ reductions because the reduction rules are decreasing. It remains to find an efficient way to decide, whether a rule can be applied to a give graph. In [9], a method called *bounded adjacency-list method* is used. They define so called *special* reduction systems, for which this decision algorithm for P takes $\mathcal{O}(n)$ time and uses $\mathcal{O}(n)$ space for connected graphs.

Definition 5.1.3. A reduction system $(\mathcal{R}, \mathcal{I})$ is called *special* if there are integers $n_{min} < \max\{|V(H_1)| : (H_1, H_2) \in \mathcal{R}\} < d$ satisfying

- For each rule $(H_1, H_2) \in \mathcal{R}$, the graphs H_1, H_2 are connected and *open* i.e. there are no edges between their terminals.
- Each connected graph G (given by an adjacency-list representation) which satisfies $P(G)$ and $|V(G)| \geq n_{min}$, G contains a *d-discoverable* match, i.e. a connected and open l-terminal graph G_1 with the following properties:

- The maximum degree of any vertex in G_1 is at most d.
- There is a l-terminal graph G_2 with $G = G_1 \oplus G_2$.
- G_1 contains a non-terminal vertex v such that for all vertices $w \in V(G_1)$ there exists a vw-walk $v = u_1 u_2 \dots u_n = w$ in G_1 in which incident edges $u_{i-1}u_i$, $u_i u_{i+1}$ have distance at most d in the adjacency list of u_i, for all $1 < i < n$.

Due to this definition, there always exist some walks from any inner vertex to arbitrary other vertices of G_1 (via v) which admit this restriction on the distance in the adjacency lists. We can, obviously, restrict to walks where each edge occurs at most twice; otherwise an edge would be used twice in the same direction and we can omit the intermediate part.

Consequently, we can find d-discoverable subgraphs by starting at v and exploring walks using edges at most twice. Of course, this yields the following result:

Lemma 5.1.4. *If $v \in V$ is a vertex of G and a non-terminal vertex of of some d-discoverable match G_1, then the match can be found from v in time depending only on d and the size of G_1, but not on the size of G.*

Therefore, we note that inner vertices in G_1 do not only need to have degree $\leq d$ in G_1 but also in G. One can thus find a linear reduction algorithm based on the following steps.

1. $S = \{v \in V(G) | deg(v) \leq d\}$
2. if $S \neq \emptyset$ take $v \in S$ else return $G \in \mathcal{I}$
3. If v is inner vertex of a d-discoverable match G_1 to a rule in $\mathcal{R}$, apply the rule to G else remove v from S
4. continue with step 2

Since both essential steps can be executed in constant time, the runtime depends only on the number of iteration steps and thus on the number of vertices removed from S. Initially, there are $\mathcal{O}(n)$ vertices; when applying a rule we possibly have to add some of the terminal vertices as their degree decreases but for each reduction step their number is $\mathcal{O}(1)$. Altogether the execution takes $\mathcal{O}(n)$ time.

This seems pretty promising but we have to keep in mind that the constant hidden in the $\mathcal{O}$-notation is dependent on the number of reduction rules.

5.1.2 Reduction systems for construction properties

For construction properties, we not only need a reduction system but algorithms to provide the actual solutions. We again characterize construction problems by pairs $(\mathcal{D}, \mathcal{Q})$ in the following way

$\mathcal{P}(G) =$ "there is an $S \in \mathcal{D}(G)$ with $\mathcal{Q}(G, S) = \mathit{true}$ "

where $\mathcal{D}$ is a function mapping a graph G to a corresponding *solution domain* i.e. a set of solutions $\mathcal{D}(G)$ depending on G and $\mathcal{Q}$ is an *extended* graph property for G and S i.e. a function mapping the pairs $(G, S), S \in \mathcal{D}(G)$ to boolean values (see Section 2.1.1).

If we not only provide a (special) reduction system but

- an algorithm $\mathcal{A}_{\mathcal{R}}$ which, given a reduction rule $r = (H_1, H_2)$, terminal graphs G_i isomorphic to $H_i, i = 1, 2$ and a solution $S \in D(G_2 \oplus H)$ with $Q(G_2 \oplus H, S)$, computes a solution $S' \in D(G_1 \oplus H)$ with $Q(G_1 \oplus H, S')$, and
- an algorithm $\mathcal{A}_{\mathcal{I}}$ which, given a graph G isomorphic to $I \in \mathcal{I}$ computes a solution $S \in D(G)$,

we could extend the above decision algorithm to a construction algorithm by undoing the reduction afterwards. If the algorithms above use only constant time this gives a $\mathcal{O}(n)$ time and space construction algorithm.

5.1.3 Reduction systems for optimization problems

For optimization problems induced by a function Φ, we have to use extended reduction rules that use an additional integer counter. These ruled are called *reduction-counter rules*.

Definition 5.1.5. A *reduction-counter rule* is a pair (r, i), where r is a reduction rule and i an integer.

A match for (r, i) in a graph G is defined as a match for r in G.

An application of a match (r, i) to a graph G together with an integer counter *cnt* is an operation which applies r to G an additionally replaces *cnt* by *cnt*+i.

The integer i keeps track of the cost/gain of the reduction i.e. for safe reduction rules (r, i) we have $\Phi(G) = \Phi(G') + i$ for the corresponding optimization property Φ. The notations and remaining definitions work as for graph properties.

Definition 5.1.6. A *reduction-counter system* for a graph optimization problem Φ is a triple $(\mathcal{R}, \mathcal{I}, \phi)$, where $\mathcal{R}$ is a finite set of reduction-counter rules which are safe, complete and decreasing for Φ, $\mathcal{I}$ is the set of irreducible graphs with $\Phi(G) \neq false$ and ϕ is a function mapping graphs $G \in \mathcal{I}$ to their (optimal) valuation $\phi(G)$.

We do not want to go into more details here and refer to [9] for the formal definition. However, one obtains an algorithm similar to the algorithm by the induced graph property by just additionally summing up the counter values during the reductions and, at the end, adding the value $\phi(G)$ of the obtained irreducible graph G,

If necessary, a construction of the solutions for the optimization problem can be obtained in a similar way as it was done for graph problems. The full details can be found in [9].

In all these cases and even without a full consideration, a (linear) algorithm can be obtained from a (special) reduction system in a quite intuitive way. We just need to find a (special) reduction system suitable for our problem and this can actually be done for a quite huge class of graph and optimization properties on graphs of bounded treewidth as we will see in the next section.

5.2 Obtaining reduction systems for graphs of bounded treewidth

In this section, we sketch how special reduction systems can be found for a large number of graph problems.

5.2.1 Graph properties

For a graph property P, we start by defining a relation $\sim_{P,k}$ on k-terminal graphs,

$$G \sim_{P,k} H \Leftrightarrow (\forall K : P(G \oplus K) \leftrightarrow P(H \oplus K)). \tag{5.1}$$

Definition 5.2.1. A graph property P is of *finite index* if for every $k \geq 0$, the number of equivalence classes of $\sim_{P,k}$ is finite.

Such finite index properties can actually be solved in linear time on graphs of bounded treewidth by introducing some special reduction systems.

We just consider *refinements* of the above relation i.e. related relations where each equivalence class is a subset of one of the equivalence classes of $\sim_{P,k}$. Of course, if those are already of finite index, so is $\sim_{P,k}$. Similarly, one can show:

Lemma 5.2.2. *If some graph properties P_1 and P_2 are of finite index, then the graph properties Q_1 and Q_2 defined by $Q_1(G) = P_1(G) \wedge P_2(G)$ and $Q_2(G) = P_1(G) \vee P_2(G)$ are also of finite index.*

Example (Graphs with bounded treewidth). For $G = G_1 \oplus G_2$, a tree-decomposition of G can be obtained from tree-decompositions of G_1 and G_2 by gluing the trees together at a common root node with vertex set $X_r = \emptyset$ and keeping all other sets while identifying the necessary vertices. Thus, we have $tw(G) \leq \max(tw(G_1), tw(G_2)$. Anyway, if the original tree-decompositions were of minimal treewidth, so is the result, because both G_1 and G_2 can be interpreted as subgraphs. Therefore, $tw(G) = \max(tw(G_1), tw(G_2)$ and the relations are given by

$$G \sim_{TW_k,l} H \Leftrightarrow (\forall K : tw(G \oplus K) \leq k \leftrightarrow tw(H \oplus K) \leq k) \tag{5.2}$$

$$\Leftrightarrow (\forall K : (tw(G) \leq k \wedge tw(K) \leq k) \leftrightarrow (tw(H) \leq k \wedge tw(K) \leq k)). \tag{5.3}$$

Of course, there are just two equivalence classes: the graphs with treewidth $\leq k$ and those with treewidth $> k$, and $TW_k(G)$ is of finite index.

Given a finite-index graph property P, we define the property P_k by

$$P_k(G) = P(G) \wedge TW_k(G) \tag{5.4}$$

By Lemma 5.2.2, this property is again of finite index.

Finally, one obtains the following result.

Theorem 5.2.3. *Let P be a graph property and suppose that P is of finite index. For each $k \geq 1$ there exists a special reduction system for P_k. If P is efficiently decidable i.e. there is a known algorithm deciding P and there is a finite, efficiently decidable refinement $\sim_l$ of $\sim_{P,l}$ for each $l \geq 0$, then such a special reduction system can be constructed efficiently.*

Proof sketch. For each $l \leq 2(k+1)$ and every equivalence class C_l of $\sim_{P_k,l}$, we check whether C_l contains open and connected l-terminal graphs with treewidth at most k and, if possible, we choose a representative $H^l_{C_l} \in C_l$. Consider a fixed value $n_{min} > |H^l_C|$ for all C and l.

One can show that there exist integers d and n_{max} with $2(n_{min} - 1) \leq n_{max} \leq d$ and a constant $c > 0$ such that in each connected graph with treewidth at most k and with $n := |V(G)| > n_{min}$, there are at least $\lceil cn \rceil$ d-discoverable open and connected terminal graphs H with at most $2(k+1)$ terminals and $n_{min} \leq |V(H)| \leq n_{max}$. As done in [9], we use [34] as a reference.

For each l with $0 \leq l \leq 2(k+1)$ and all open l-terminal graphs H with $n_{min} \leq |H| \leq n_{max}$ and treewidth at most k we find a graph H' with $H \sim_l H'$ (e.g. the representative) and add a rule (H, H'). These are only finitely many because the number of graphs with restricted number of vertices is finite. Furthermore, we choose

$$\mathcal{I} = \{G | G \text{ is irreducible } \wedge G \text{ is connected } \}.$$

By definition, the reduction system is safe and decreasing. One can also see that we indeed get a special reduction system. The second condition follows from the above choice of d and n_{max}. This also shows that the system is complete.

We can efficiently find such a system by just trying all possible values for n_{min} until a suitable value is found because the above process works and is efficient. □

Of course, the number of rules in such reductions systems can be quite huge and the time complexity of the corresponding reduction algorithm is highly dependent on their number. On the other hand, we notice that we do not even need an actual tree-decomposition to perform the above construction. That is, why a more efficient parallel variant of the obtained reduction algorithm sometimes gives efficient algorithms.

A parallel setting for the above principle is used by Bodlaender and Hagerup in [34] and their algorithms sometimes even take $\mathcal{O}(\log(n))$ time. Even with parallelization, the most efficient known algorithms for the construction of tree-decompositions still take $\mathcal{O}(\log^2(n))$ time (see [34]).

5.2.2 Construction properties

For construction properties, it is not sufficient to just consider equal satisfiability when defining the equivalence classes. We also need to make sure, that the obtained solutions correspond.

For this purpose, we consider solutions to be *t-vertex-edges-tuples* i.e. elements of the Cartesian product of t sets, which are either V,E, $\mathcal{P}(V)$ or $\mathcal{P}(E)$. This also gives a natural way to restrict solutions to subsets: we just restrict the corresponding domain! One defines a compatibility for solutions, which expresses whether solutions can be glued together.

Definition 5.2.4. Let D be a vertex-edge-tuple, G and H l-terminal graphs and S_G and S_H be some partial solutions for G or H respectively. If there is a solution for $G \oplus H$ whose restriction to G and H equals S_G and S_H respectively, then (G, S_G) and (H, S_H) are called $\oplus$-*compatible* and we denote $S = S_G \oplus S_H$.

Similarly, for l-terminal graphs G_1, G_2 and corresponding solutions S_1, S_2 the pairs (G_1, S_1) and (G_2, S_2) are *compatible*, if for each l-terminal graph H and solution S_H for H the pair (G_1, S_1) is $\oplus$-compatible with (H, S_H) if and only if (G_2, S_2) is $\oplus$-compatible with (H, S_H).

The equivalence relation $\sim_{Q,l}$ for l-terminal graphs G_1, G_2 and a construction property Q is, then, defined the following way:

$$\begin{aligned}(G_1, S_1) \sim_{Q,l} (G_2, S_2) \Leftrightarrow &(G_1, S_1), (G_2, S_2) \text{ are compatible and for all } l\text{-terminal} \\ &\text{graphs } H \text{ and corresponding solutions } S \text{ we have} \\ &Q(G_1 \oplus H, S_1 \oplus S) \equiv Q(G_2 \oplus H, S_2 \oplus S).\end{aligned}$$

Again, we work with refinements and finally obtain:

Theorem 5.2.5. *Let P be a construction property defined by (D, Q) and suppose D is a vertex-edge-tuple. If $\sim_{Q,l}$ has finitely many equivalence classes for each $l \geq 0$, then for each $k \geq 1$, there exists a special constructive reduction system $(\mathcal{R}, \mathcal{I}, \mathcal{A}_R, \mathcal{A}_I)$ for P_k defined by $(D, \mathcal{Q}_k)$ with $Q_k(G, S) = Q(G, S) \wedge TW_K(G)$.*

We omit the case of optimization problems here. Reduction systems for optimization problems as well as a study of the parallel variant of the actual reduction algorithm is given in [9].

With the mentioned parallel variants, the reduction algorithm takes $\mathcal{O}(\log(n) \cdot \log^*(n))$ or even $\mathcal{O}(\log(n))$ time (depending on the type of parallelization) with $\mathcal{O}(n)$ operations and space.

Conclusion

We want to conclude by summarizing the observations made throughout this thesis.

The discussed approaches indicate that the treewidth parameter k is indeed a suitable parameter for parametrized complexity. It not only works well for dynamic-programming approaches, as discussed in Chapter 3, but it even allows for other methods such as reduction algorithms which are based on fundamentally different ideas, as observed in Chapter 5. The parameter k, therefore, seems to reflect the essential structure which determines the actual complexity of the corresponding graph – at least for some important computational purposes.

Nevertheless, some constants hidden in the $\mathcal{O}$-notation of the mentioned and related algorithms are still quite huge. There is still a need for improvements and heuristics with better performance for treewidth computations for arbitrary, and also for special types of graphs. However, the diversity of approaches for algorithms based on tree-decompositions allows for a variety of such improvements. This makes this field an interesting and wide area for further research.

The author hopes that this thesis raises the readers interest in this area, just as it did raise her own.

Bibliography

[1] Reinhard Diestel.
Graph theory, volume 173 of *Graduate Texts in Mathematics*.
Springer, Heidelberg, fourth edition, 2010.

[2] Miriam Heinz.
Tree-decomposition - graph minor theory and algorithmic implications.
Diplomarbeit, Vienna University of Technology, 2013.

[3] Neil Robertson and P. D. Seymour.
Graph minors. II. Algorithmic aspects of tree-width.
J. Algorithms, 7(3):309–322, 1986.

[4] Hans L. Bodlaender.
Treewidth: structure and algorithms.
In *Structural information and communication complexity*, volume 4474 of *Lecture Notes in Comput. Sci.*, pages 11–25. Springer, Berlin, 2007.

[5] Hans L. Bodlaender.
A partial k-arboretum of graphs with bounded treewidth.
Theoret. Comput. Sci., 209(1-2):1–45, 1998.

[6] Stefan Arnborg and Andrzej Proskurowski.
Linear time algorithms for NP-hard problems restricted to partial k-trees.
Discrete Appl. Math., 23(1):11–24, 1989.

[7] Donald J Rose.
Triangulated graphs and the elimination process.
Journal of Mathematical Analysis and Applications, 32(3):597 – 609, 1970.

[8] Neil Robertson and P. D. Seymour.
Graph minors. XIII. The disjoint paths problem.
J. Combin. Theory Ser. B, 63(1):65–110, 1995.

[9] Hans L. Bodlaender and Babette van Antwerpen-de Fluiter.
Reduction algorithms for graphs of small treewidth.
Inform. and Comput., 167(2):86–119, 2001.

[10] R. Rod G. Downey and M.R. Fellows.
Parameterized Complexity.

[11] Hans L. Bodlaender.
Treewidth: algorithmic techniques and results.
In *Mathematical foundations of computer science 1997 (Bratislava)*, volume 1295 of *Lecture Notes in Comput. Sci.*, pages 19–36. Springer, Berlin, 1997.

[12] Petra Scheffler.
A practical linear time algorithm for disjoint paths in graphs with bounded tree

width.
Fachbereich Mathematik - Report, 396, 1994.

[13] Hans L. Bodlaender and Fedor V. Fomin.
Equitable colorings of bounded treewidth graphs.
Theoret. Comput. Sci., 349(1):22–30, 2005.

[14] Klaus Jansen and Petra Scheffler.
Generalized coloring for tree-like graphs.
Discrete Appl. Math., 75(2):135–155, 1997.

[15] H. L. Bodlaender.
A tourist guide through treewidth.
Acta Cybernet., 11(1-2):1–21, 1993.

[16] Hans L. Bodlaender and Arie M. C. A. Koster.
Combinatorial optimization on graphs of bounded treewidth.
Comput. J., 51(3):255–269, May 2008.

[17] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier.
Fixed parameter algorithms for dominating set and related problems on planar graphs.
Algorithmica, 33(4):461–493, 2002.

[18] Hans L. Bodlaender.
Dynamic programming on graphs with bounded treewidth.
In *Automata, languages and programming (Tampere, 1988)*, volume 317 of *Lecture Notes in Comput. Sci.*, pages 105–118. Springer, Berlin, 1988.

[19] Bruno Courcelle.
The monadic second-order logic of graphs. I. Recognizable sets of finite graphs.
Inform. and Comput., 85(1):12–75, 1990.

[20] Stefan Arnborg, Jens Lagergren, and Detlef Seese.
Easy problems for tree-decomposable graphs.
J. Algorithms, 12(2):308–340, 1991.

[21] Richard B. Borie, R. Gary Parker, and Craig A. Tovey.
Deterministic decomposition of recursive graph classes.
SIAM J. Discrete Math., 4(4):481–501, 1991.

[22] B. Courcelle and M. Mosbah.
Monadic second-order evaluations on tree-decomposable graphs.
Theoret. Comput. Sci., 109(1-2):49–82, 1993.
International Workshop on Computing by Graph Transformation (Bordeaux, 1991).

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

[24] Hans L. Bodlaender.
A linear-time algorithm for finding tree-decompositions of small treewidth.
SIAM J. Comput., 25(6):1305–1317, 1996.

[25] Jens Lagergren.
Efficient parallel algorithms for graphs of bounded tree-width.

J. Algorithms, 20(1):20–44, 1996.

[26] Bruce A. Reed.
Finding approximate separators and computing tree width quickly.
In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 221–228, New York, NY, USA, 1992. ACM.

[27] Jens Lagergren and Stefan Arnborg.
Finding minimal forbidden minors using a finite congruence.
In *Automata, languages and programming (Madrid, 1991)*, volume 510 of *Lecture Notes in Comput. Sci.*, pages 532–543. Springer, Berlin, 1991.

[28] Hans L. Bodlaender and Ton Kloks.
Efficient and constructive algorithms for the pathwidth and treewidth of graphs.
J. Algorithms, 21(2):358–402, 1996.

[29] Ljubomir Perković and Bruce Reed.
An improved algorithm for finding tree decompositions of small width.
Internat. J. Found. Comput. Sci., 11(3):365–371, 2000.
Selected papers from the Workshop on Theoretical Aspects of Computer Science (WG 99), Part 1 (Ascona).

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

[31] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks.
Approximating treewidth, pathwidth, frontsize, and shortest elimination tree.
J. Algorithms, 18(2):238–255, 1995.

[32] Hans L. Bodlaender and Arie M. C. A. Koster.
Treewidth computations II. Lower bounds.
Inform. and Comput., 209(7):1103–1119, 2011.

[33] Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese.
An algebraic theory of graph reduction.
J. Assoc. Comput. Mach., 40(5):1134–1164, 1993.

[34] Hans L. Bodlaender and Torben Hagerup.
Parallel algorithms with optimal speedup for bounded treewidth.
SIAM J. Comput., 27(6):1725–1746 (electronic), 1998.