Treewidth and Dynamic Programming 10

10.1 Basic Facts About Treewidth

10.1.1 Introduction

One of the major outgrowths from the beautiful work of Robertson and Seymour, for example [582, 588, 589], was the focus on a new parameter, which measures how "tree-like" a graph is in a precisely defined sense. We could then lift parametric results from trees to graphs that are "tree-like".

Concepts equivalent to tree decomposition were subsequently, and apparently independently, discovered in other contexts such as the artificial intelligence literature. As we see in the next section, this parametric method of "tree-like" data analysis also enables a uniform approach to linear-time algorithms on a wide class of graphs. Furthermore, this parameter is also the basis of modern algorithmic developments like "bidimensionality" and "protrusions" we will meet later in this book. Finally, the focus on *topological* graph theory had a profound impact on combinatorics.

Whilst determining whether a graph has treewidth k is FPT, the algorithms to do this are impractical. Because of this, it was thought that treewidth would have no impact on practical algorithmics, besides a classification tool. Yet there is a growing use of treewidth in practice, particularly via heuristics and where the certificate of the treewidth is either given or close to the surface. For example, Koster, van Hoesel and Kolen [458] applied this to constraint satisfaction problems such as frequency assignment problems from practical data. Inference problems and Bayesian belief networks come readily equipped with a tree structure, and admit analysis by treewidth-based algorithms. (See, for example, S.L. Lauritzen and D.J. Spiegelhalter [482] or Alber, Dorn, and Niedermeier [23].)

We remark that algorithms for (truly) small treewidth are of interest, and such graphs arise quite regularly to depict natural phenomena. For example, Thorup [645] showed that the flow control graph of go-to free programs like *Pascal*. Yamagucki, Aoki, and Mamitsuka [667] computed the treewidth of 9712 chemical compounds from the LIGAND database and found that all but one had treewidth at most 3 and the remainder had treewidth 4.

R.G. Downey, M.R. Fellows, *Fundamentals of Parameterized Complexity*,
Texts in Computer Science, DOI 10.1007/978-1-4471-5559-1_10,

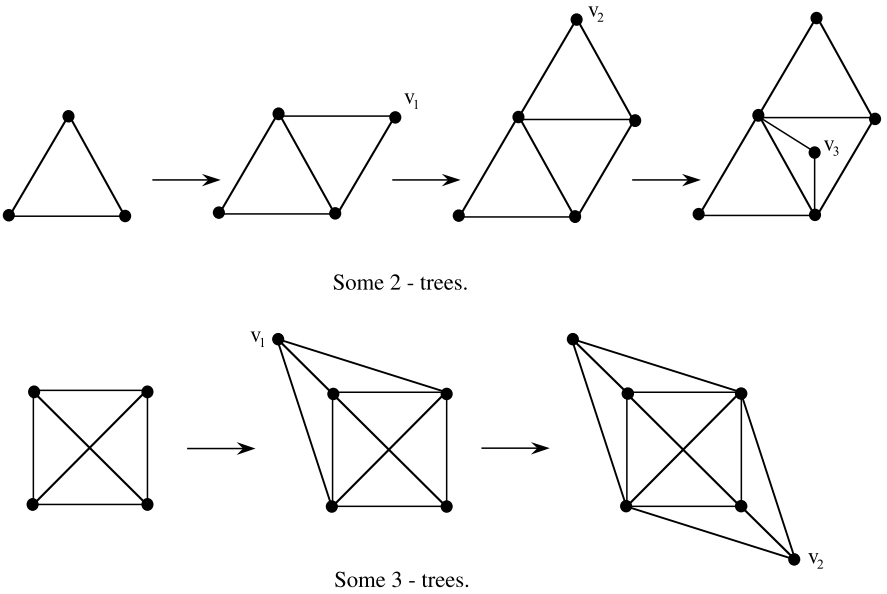

Fig. 10.1 Examples of 2- and 3-trees

10.1.2 The Basic Definitions

The simplest definition of treewidth for *graphs* comes from *partial k-trees.*

Definition 10.1.1 (k-tree)
(a) The class of k-trees is the smallest class obeying (i) and (ii) below.
  (i) K_{k+1}, the complete graph on $k+1$ vertices, is a k-tree.
  (ii) If G is a k-tree and H is a subgraph of G isomorphic to K_k, then the graph G' constructed from G by first adding a new vertex v to G and then adding edges to make $H \cup \{v\}$ a copy of K_{k+1}, is a k-tree.
(b) If G_1 is a subgraph of a k-tree, then G_1 is called a *partial k-tree*.

In Fig. 10.1 we give examples of 2- and 3-trees.

Definition 10.1.2 (Treewidth) We say that a graph G has *treewidth* k if k is least such that G is a partial k tree.

For algorithmic purposes, Definition 10.1.2 is not the most useful. An equivalent characterization of treewidth is provided below.

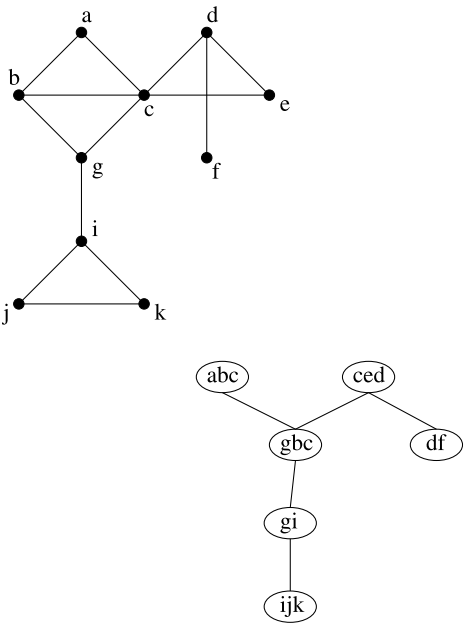

Fig. 10.2 Example of tree decomposition of width 2

Definition 10.1.3 (Tree decomposition)

(a) A *tree decomposition* of a graph $G = (V, E)$ is a tree $\mathcal{T}$ together with a collection of subsets T_x (called *bags*) of V labeled by the vertices x of $\mathcal{T}$ such that $\bigcup_{x \in \mathcal{T}} T_x = V$ and the following connectivity properties (1) and (2) hold:
   (1) For every edge uv of G there is some x such that $\{u, v\} \subseteq T_x$.
   (2) (Interpolation property) If y is a vertex on the unique path in $\mathcal{T}$ from x to z then $T_x \cap T_z \subseteq T_y$.

(b) The *width* of a tree decomposition is the maximum value of $|T_x| - 1$ taken over all the vertices x of the tree $\mathcal{T}$ of the decomposition.

We remark that Definition 10.1.3 of treewidth is not only algorithmically more useful, but easily allows for generalization to multigraphs, hypergraphs, matroids, and the like.

If a tree decomposition of a graph G gives a path, then we say that the tree decomposition is a *path decomposition*, and use *pathwidth* in place of treewidth.

Figure 10.2 gives an example of a tree decomposition of width 2.

The axioms above imply that certain connectivity conditions will occur on the graphs.

Lemma 10.1.1 *Let $\{T_x : x \in \mathcal{T}\}$ be a tree decomposition of G.*
(i) *Let $x \in V(G)$. Then the collection of $y \in \mathcal{T}$ with $x \in T_y$ forms a subtree of T.*
(ii) *Suppose that C is a clique of G. Then there is some $x \in \mathcal{T}$ with $C \subseteq T_x$.*

We leave this proof as an exercise (Exercise 10.2.1 below). The following theorem and its proof demonstrates that treewidth and pathwidth, the two basic representations of width, are identical.

Theorem 10.1.1 (Arnborg, Gavril [341], Rose, Tarjan, and Lueker [594]) *A graph has treewidth k iff G has a tree decomposition of width k.*

Proof Let G be a graph with a tree decomposition $\{T_x : x \in \mathcal{T}\}$ of width k. Thus $\max_{x \in \mathcal{T}} |T_x| = k+1$. We prove that G is a partial k-tree, and in fact G is a subgraph of a k-tree $K(G)$ in which every $\leq k$ element subset of a T_x is part of a k-clique. We use induction on trees $\mathcal{T}$. Let x be a leaf of $\mathcal{T}$. Let $\mathcal{T}'$ be the result of removing x from $\mathcal{T}$, and let G' be a graph corresponding to $\mathcal{T}'$. By hypothesis, G' is a subgraph of a k-tree $K(G')$ with the desired properties. Let y be attached to x in $\mathcal{T}$. Let $T_x \cap T_y = T'_x$ and $T_x - T'_x = T''_x$. If $T''_x = \emptyset$, then $T_x \subseteq T_y$ and hence $G = G'$. So suppose that $T''_x \neq \emptyset$. By property 2. of Definition 10.1.3, for all $z \neq x$, $T''_x \cap T_z = \emptyset$. It follows that T'_x is a proper subset of T_x, and hence as the width is k, T'_x is a subset of at most k elements of T_y. By hypothesis, this means that T'_x is a subset of part of a k-clique C of $K(G')$. The idea is now to simply add T''_x one node at a time. We will use Algorithm 10.1.1.

Algorithm 10.1.1
1. Initially we let $K = K(G')$.
   While $T''_x \neq \emptyset$ we repeat the following, steps 2–5.
2. Pick a vertex v of T''_x.
3. Add v to K and also add all edges between vertices of C and v.
4. Let $T''_x \leftarrow T''_x - \{v\}$.
5. If there is any vertex $c \in C - T_x$, let $C \leftarrow [C - \{c\}] \cup \{v\}$.

By induction on this algorithm, one can see that after each iteration, C is a k-clique of K. Each step emulates the definition of a k-tree. Hence as $K(G')$ is a k-tree, so too is K_{fin} the result of Algorithm 10.1.1 applied to $K(G')$. By induction, we also can see that all of T''_x is added together with all possible edges that can exist between vertices in $T_x = T'_x \cup T''_x$. Thus K_{fin} contains G and has the desired properties.

Suppose, however, that G is a partial k-tree. Since it suffices to consider k-trees, we will first suppose that G is a k-tree. We prove by induction on $|V(G)|$ that G will have a tree decomposition of width k. We use the stronger hypothesis that if G' is a k-tree with $< |V(G)|$ vertices, then G' has a tree decomposition $\{T_x : x \in \mathcal{T}\}$ where $|T_x| \leq k+1$ for all $x \in \mathcal{T}$, and each k-clique of G' occurs in some T_x. The result is clear if $G = K_{k+1}$. So if G has more than $k+1$ nodes, let v be the last node added in the formation of G. (That is, v is a *simplicial* node.) Let $N(v)$ be

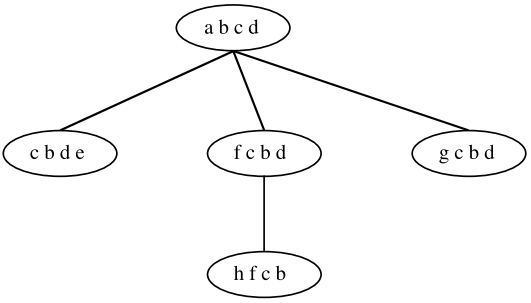

Fig. 10.3 Example of tree decomposition of width 3

the neighbors of v in G. Let G' be the result of the removal of v and the k edges vu with $u \in N(v)$. Then we can apply the induction hypothesis to G' and obtain a tree decomposition $\{T'_x : x \in \mathcal{T}'\}$ of G' with the desired properties. Now $N(v)$ is a k-clique and hence, by hypothesis, there is some $y \in T'$ with $N(v) \subseteq T'_y$. Create a new tree $\mathcal{T}$ from $\mathcal{T}'$ by attaching a new leaf $\ell = \ell(v)$ at y. Let $T_\ell = N(v) \cup \{v\}$. By hypothesis, $\mathcal{T}$ will be the relevant tree decomposition of G. □

Definition 10.1.4 (Bounded treewidth) We say that a class $\mathcal{C}$ of graphs has *bounded treewidth* if there is some k such that for all $G \in \mathcal{C}$, G has treewidth $\leq k$.

10.2 Exercises

Exercise 10.2.1 Prove Lemma 10.1.1.

Exercise 10.2.2 (Per W. Merkle) An ordering $I = \langle I_1, \ldots, I_n \rangle$ of a set of subsets of V is called an *acyclic hypergraph* iff for all $j \leq n$, there is a $q < j$ such that $I_j \cap (\bigcup_{i<j} I_i) \subseteq I_q$. (This is also called the *running intersection property* per Lauritzen and Spiegelhalter [482].) Given a finite graph, we say that an acyclic hypergraph I is an acyclic hypergraph decomposition of G iff $V = V(G)$ and for each edge e of G there is some i such that the vertices of e are in I_i.

Prove that G has a tree decomposition of width w iff G has an acyclic hypergraph decomposition I such that for all $i \leq n$, $|I_i| \leq w$.

(Hint: Prove that one can construct the tree decomposition from I to have the I_i as the actual bags, and conversely.)

Exercise 10.2.3 Apply the algorithm implicit in Theorem 10.3 to embed the graph described by the tree decomposition of Fig. 10.3 into a 3-tree.

Exercise 10.2.4 Prove that if a graph G has a size k vertex cover, then it must also have treewidth $\leq k$.

Exercise 10.2.5 A *Halin graph* is constructed as follows: take a tree without vertices of degree 2. Then, embed it in the plane, and add a cycle through its leaves, in order (i.e., the result remains planar). Prove that a Halin graph must have treewidth 3.

Exercise 10.2.6 A graph is called *outerplanar* iff there exists a plane embedding G of the graph such that G has a cycle C with the property that every edge of G is either a member of C or a chord of C. Prove that if G is outerplanar then it has treewidth 2.

10.3 Algorithms for Graphs of Bounded Treewidth

Historically, algorithm authors often tried to get around classical intractability by looking at their particular problems represented on special classes of graphs. They often discovered that intractable problems became tractable if the problems were restricted to, say, "outerplanar" graphs. Such restriction is not purely an academic exercise since, in many practical situations, the graphs that arise do not, in fact, demonstrate the full pathology of the class of all graphs. (Consider, for instance, a TRAVELING SALESMAN concerned with a given city. The graph will almost certainly be of small maximum clique size.) The table (from Van Leeuwen [652]) below, lists some families of graphs that have been studied and have treewidth.

Families of graphs	Bound on treewidth
Trees	1
Almost trees (k)	$k+1$
Partial k-trees	k
Bandwidth k	k
Cutwidth k	k
Planar of radius k	$3k$
Series parallel	2
Outerplanar	2
Halin	3
k-Outerplanar	$3k-1$
Chordal with maximum clique size k	$k-1$
Undirected path with maximum clique size k	$k-1$
Directed path with maximum clique size k	$k-1$
Interval with maximum clique size k	$k-1$
Proper interval with maximum clique size k	$k-1$
Circular arc with maximum clique size k	$2k-1$
Proper circular arc with maximum clique size k	$2k-2$

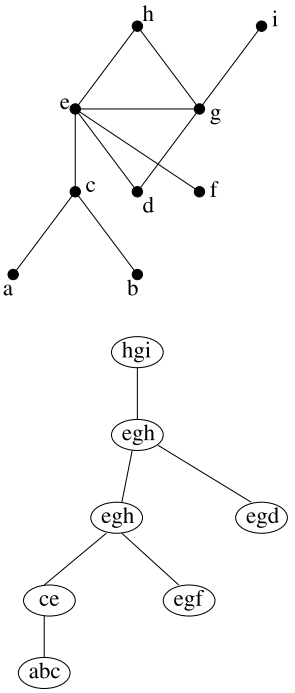

Fig. 10.4 A tree decomposition

We present this table only for background information. It is not really important if the reader is unfamiliar with some of the graph families above. The reader is invited to verify a few of these treewidth bounds in the Exercises in Sect. 10. We consider the cases of CUTWIDTH and BANDWIDTH in some detail in Sect. 12.7.

Important, however, is the fact that treewidth can be represented by these graph families. Demonstration of tractability for graphs of bounded treewidth will demonstrate tractability for all these families. *Furthermore, and perhaps even more importantly, as we see in the next section, tree decomposition enables general automata-theoretic methods for demonstrating tractability of a wide class of properties for graphs of bounded treewidth.*

We now sketch how to run algorithms on graphs of bounded treewidth. This can be viewed as "*dynamic programming*", a very valuable algorithmic technique. A good introduction to this technique is given by Bodlaender [70]; helpful introductions to other algorithmic aspects of treewidth are also provided by Bodlaender [73, 75].

We can apply the dynamic programming technique of graphing bounded treewidth to refine algorithms for the INDEPENDENT SET. For general graphs G, the problem is NP-complete and we also think it is fixed parameter intractable, as we will see that it is $W[1]$ complete.

In the case of graphs of bounded treewidth, we can give a linear-time algorithm. So suppose that we have a tree decomposition of G. Consider the one in Fig. 10.4.

We will use *tables* to grow the independent set up the tree, starting at the leaves of the decomposition. Notice that once a vertex leaves the bags, it will never come back, and hence we do not really need to keep track of its effect. Thus we can focus on working with tables that correspond to all the subsets of the current bag. We need only consider the size of independent sets I growing relative to the information flow across the boundary of the current bag.

Starting at the leaf corresponding to the triangle abc, we could generate the subset table below. Here, the box with the entry ab denotes the subset $\{a, b\}$, meaning that the independent set should contain both a and b, and would have cardinality 2 entered in the cell below. The box with bc corresponds to subset $\{b, c\}$, and since $\{b, c\}$ is *not* an independent set, no cardinality is denoted by a line in the cell below.

Ø	a	b	c	ab	ac	bc	abc
0	1	1	1	2	–	–	–

The subset table for the next box would have only four columns since there are only four subsets of $\{c, d\}$ and we consider maximal independent sets I containing the specified subsets.

Ø	c	e	ce
2	1	3	–

The first entry has value 2 since this means that neither of c or e is included in this independent set I, and we take the maximum entry from the previous table compatible with $\{c, e\} \cap I = \emptyset$, namely 2 for the $\{a, b\}$ column.

The second entry indicates that $c \in I$ and $e \notin I$, that is, that the intersection of the bag $\{c, e\}$ with I is the singleton set $\{c\}$. The maximum entry in the second row of the previous table, over compatible columns, yields an entry of 1.

The third entry corresponds to $I \cap \{c, e\} = \{e\}$. This is compatible with $\{a, b\} \subseteq I$. In particular, e is not adjacent to a or b, yielding an entry of $2 + 1 = 3$.

The fourth entry corresponds to $I \cap \{c, e\} = \{c, e\}$, but this is impossible since c and e are adjacent, hence the line entry.

The rest of the tables are filled in similarly. With pointers you can also keep track of the relevant independent set.

The next table is for a join node of the tree decomposition. As above, each column corresponds to the intersection of I with the bag $\{e, g, h\}$. Now we must check compatibility with respect to the *two* tables, corresponding to the bags $\{c, e\}$ and $\{e, g, f\}$. The entry is the maximum sum from the two tables, over compatible columns, and numerically adjusting for either repetitions, or for adding new vertices to I.

The final tree of tables corresponding to the tree decomposition is shown in Fig. 10.5.

The maximum value of 4 for the column corresponding to $I \cap \{e, g, h\} = \{h\}$ can be obtained by noting that this intersection information is jointly compatible with the column corresponding to $\emptyset$ in the table for the bag $\{c, e\}$, with value 2, and with

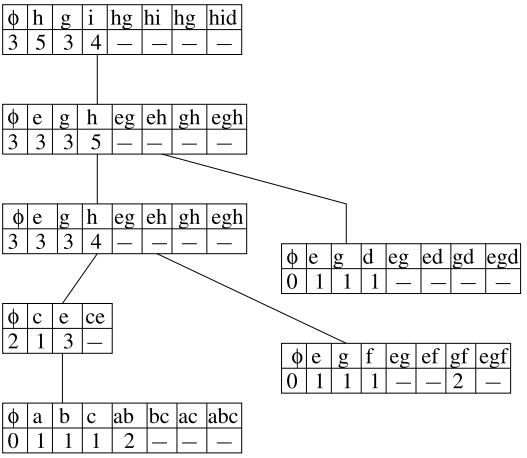

Fig. 10.5 Dynamic programming

the column corresponding to $\{f\}$ in the table for the bag $\{e, g, f\}$, with value 1. These are disjoint, so no repetitions are involved, and the new vertex h joins I, yielding $2 + 1 + 1 = 4$.

Now the data structures above are the most naive implementation possible, and using many techniques we can speed things up. Later we will discuss one method using matrices and distance products as data structures. We do this in Sect. 16.4, when we consider dynamic programming on graphs of bounded branchwidth, which is a similar parameter to treewidth. Other sophisticated techniques are mentioned in Chap. 16.

A sly feature of dynamic programming by this method of growing the independent set using coordinated tables, is that we need a tree decomposition for the graph G. It turns out that for a fixed t there is a *linear-time algorithm* that determines if G has width t, and then finds the tree decomposition of G—should the graph actually have one. However, the algorithm has really terrible constants and there is no actually feasible tree decomposition for this problem. For more on this we refer the reader to Bodlaender [73], and his web site. John Fouhy [329] in his M.Sc. Thesis looked at computational experiments for treewidth heuristics. (See www.mcs.vuw.ac.nz/~downey/students.html.) Chapter 11 reports on many other treewidth heuristics. There is much work to be done in this field.

10.4 Exercises

Exercise 10.4.1 Construct a tree decomposition for the graph of treewidth 2 in Fig. 10.6.

Exercise 10.4.2 Using dynamic programming on the graph of Exercise 10.4.1, compute

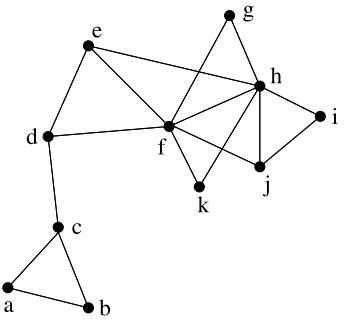

Fig. 10.6 A graph of treewidth 2

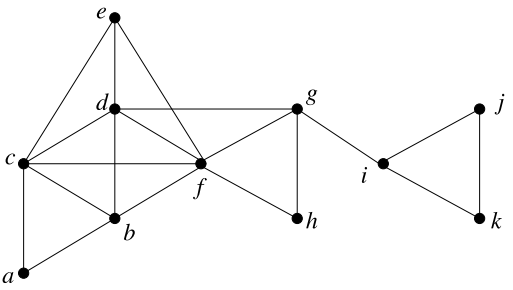

Fig. 10.7 A graph of treewidth 3

(i) A maximum independent set.
(ii) A minimum dominating set.
(iii) A minimum vertex cover.

Exercise 10.4.3 Construct a tree decomposition for the graph of treewidth 3 in Fig. 10.7.

Exercise 10.4.4 Using dynamic programming on the graph of Exercise 10.4.3, compute
(i) A maximum independent set.
(ii) A minimum dominating set.
(iii) A minimum vertex cover.

Exercise 10.4.5 A graph G is called *series parallel* if it can be generated by the following operations. Take a loop or an edge and interpolate a vertex v. Take an edge e and create a parallel edge (i.e., with the same vertices) e'. Prove that if G is series parallel, then G has treewidth 2.

Exercise 10.4.6 A graph G is said to be *almost tree* (k) iff it has at most k cycles. Prove that if G is almost tree (k), then G has treewidth $\leq k+1$.

Treewidth methods can be used to replace the *ad hoc* combinatorial methods previously employed by many authors and which usually entailed heroic case analysis. The quintessential example of this phenomenon will be Courcelle's theorem (The-

orem 13.1.1), which demonstrates for graphs of bounded treewidth, a linear-time algorithm for recognition of any property of graphs statable in a very powerful fragment of second-order arithmetic (e.g., HAMILTONICITY). All of these arguments are based on the fact that the following problem is fixed parameter tractable.

TREEWIDTH
Input: A graph G.
Parameter: A positive integer k.
Question: Does G have treewidth k?

Theorem 10.4.1 (Bodlaender's Theorem, Bodlaender [74]) TREEWIDTH *is strongly* FPT. *In fact, for each k, there is a linear-time algorithm for recognition of graphs of treewidth k. Furthermore, given a graph G of treewidth k, the algorithm will produce a tree decomposition of G of width k in linear time.*

Bodlaender's theorem is proven in Bodlaender [74], via Bodlaender and Kloks [94]. Modulo the latter result, we give a complete proof of Bodlaender's theorem in the next section.

10.5 Bodlaender's Theorem

Our proof that there is a linear-time FPT algorithm which
(i) determines if a graph G has treewidth k and
(ii) produces a tree decomposition of width k if the answer to (i) is "yes", relies upon the following theorem.

Theorem 10.5.1 (Bodlaender and Kloks [94])
1. *There is an $O(|G|)$* FPT *algorithm which has parameter k and p with $k \leq p$ which, given a treewidth p tree decomposition of G determines if G has treewidth k.*
2. *Furthermore, if the treewidth of G is k, there is a linear-time algorithm which will output a treewidth k tree decomposition for G.*

There are several different approaches to proving Theorem 10.5.1(1). We can deduce it by previewing later chapters of this book. By Robertson and Seymour [589], as we see in Chap. 17, there is a finite set of graphs $H_1, \ldots, H_q$ such that a graph G of treewidth p has treewidth k iff H_i is not a minor of G for $i \in \{1, \ldots, q\}$.[1] Actually, since we know a bound on the treewidth of the H_i, it is possible to calculate the set $H_1, \ldots, H_q$, as we will see in Chap. 19. We will soon see (after Chap. 12 on Automata and Bounded Treewidth), that the minor relation is definable in monadic second-order logic (see Courcelle's theorem (Theorem 13.1.1)), we can construct an

[1] We say a graph H is a minor of a graph G if H can be obtained from G by deletion and contractions of edges. We refer to Definition 17.1.7.

automaton which runs in linear time and accepts only parse trees of width k from the input of (parse trees in linear time derived from) treewidth p tree decompositions.[2] The algorithm of Bodlaender and Kloks, which produces the tree decomposition if there is one, is concrete and elementary, but too involved to easily fit in the book. (We will give some further notes on the development of Bodlaender's theorem in the Historical Notes at the end of this chapter.)

The main idea of the proof of Bodlaender's theorem is that given G, we first partition the vertices into vertices of *low* and of *high* degree (soon to be described in Definition 10.5.1). There are then two cases.

Case 1. There are many vertices of low degree adjacent to each other. In this case, it is shown that a maximal matching contains *many* edges. One then *contracts*[3] all edges in a maximal matching and generates a new derived graph G'. Now, recursively get a treewidth k decomposition for G', or get a certificate that the treewidth of G' and therefore G exceeds k. In the former case it is possible to generate a treewidth $2k+1$ decomposition for G from that of G' using the Bodlaender–Kloks Algorithm.

Case 2. There are only few vertices of low degree adjacent to another vertex of low degree. In this case, we will *enrich* the graph by adding certain new edges to form a new graph G' of the same treewidth as G (if it is t). This new graph G' will have many vertices which are *simplicial* in the sense that their neighbors form cliques and can be removed to form G''. Given a tree decomposition of G'', one can easily get one for G.

It will be easy to see that the algorithm runs in linear time.

Let us first consider some details of the algorithm parameters.

Lemma 10.5.1 (Folklore) *Suppose that G has treewidth k. Let $\{X_i : i \in \mathcal{T}\}$ be a tree decomposition for G. Then the following properties hold*:

(i) $|E| < k|V| - \frac{1}{2}k(k+1)$.

(ii) *If $W \subseteq V$ is a clique, for some $i \in \mathcal{T}$, $W \subseteq X_i$.*

(iii) *If W_1 and W_2 induce a complete bipartite subgraph of G, then for some $i \in \mathcal{T}$, $W_1 \subseteq X_i$.*

(iv) *Suppose that $v, w \in X_i$ and $v \neq w$. Then $\{X_i : i \in \mathcal{T}\}$ is a tree decomposition for $G + vw$.*

(v) *Suppose that x and y are vertices of G with at least $k+1$ common neighbors. Then the treewidth of $G + xy$ is k and $\{X_i : i \in \mathcal{T}\}$ is also a tree decomposition for $G + xy$.*

[2] The reader might notice an apparent circularity in all of this. Is not Bodlaender's theorem actually *used* in the proof of Courcelle's theorem? The proof of Courcelle's theorem actually demonstrates that, for a fixed p, monadic second-order statements have linear-time recognition from a given treewidth p decomposition. This is independent of Bodlaender's theorem, which can now be used to add that "hence monadic second-order statements have linear-time recognition algorithms".

[3] Recall that a contraction of an edge uv in a graph G is the operation of identifying u with v. We refer to Fig. 17.2.

Proof See Exercise 10.6.1. □

A tree decomposition $\{X_i : i \in \mathcal{T}\}$ of G is called *smooth* iff for all $i \in \mathcal{T}$, X_i has $k+1$ elements and for all nodes p, q of $\mathcal{T}$ with p the child of q, $|X_p \cap X_q| = k$. Part (i) of the following lemma is well known and has a proof similar to the proof of Theorem 12.7.1 (see Exercise 10.6.2).

Lemma 10.5.2

(i) *Any tree decomposition can be converted into a smooth one for the same graph by applying the following operations to nodes p, q in $\mathcal{T}$ with p the child of q or q the child of p.*
   (a) *If $X_p \subseteq X_q$, contract pq let the bag be X_q.*
   (b) *If $X_p \nsubseteq X_q$, but $|X_q| < k+1$, take any vertex from $X_p - X_q$ and add it to X_q.*
   (c) *If $|X_p| = |X_q| = k+1$ and $|X_p - X_q| > 1$, interpolate new bags between X_p and X_q.*

(ii) (Bodlaender) *If $\{X_i : i \in \mathcal{T}\}$ is a smooth tree decomposition of G of width k, then $\mathcal{T}$ has exactly $|V(G)| - k$ nodes.*

Proof See Exercise 10.6.2. □

Algorithm Parameters Choose rationals $0 < c_1, c_2 < 1$ such that

$$c_2 = \frac{1}{4k^2 + 12k + 16} - \frac{c_1 k^2 (k+1)}{2} > 0.$$

Also, define

$$d = \max\left\{k^2 + 4k + 4, \left\lceil \left(\frac{2k}{c_1}\right) \right\rceil\right\}.$$

Definition 10.5.1 (High, low, and friendly degrees of a vertex; per Bodlaender [74]) If the degree of a vertex v exceeds d, then we say that v has high degree; if it is less than or equal to d, it has low degree; and finally if v has both low degree and is also adjacent to one of low degree, we say that it is *friendly*.

Lemma 10.5.3

(i) *If G has treewidth k, then it contains fewer than $c_1|V|$ high-degree vertices.*

(ii) *If there are f friendly vertices in G, then any maximal matching contains at least $\frac{f}{2d}$ edges.*

Proof (i) is clear. (ii) Let M be a maximal matching. Note that any friendly vertex must either be an endpoint of an edge of M or adjacent to an endpoint of an edge of M. Now, associated with each edge e of M there can be at most $2d$ friendly vertices which are either endpoints of e or adjacent a low-degree friendly endpoint of e. Hence, $M > \frac{f}{2d}$. □

Definition 10.5.2 (M-derived graph)
1. Let M be a maximal matching of G. The *M-derived graph* is the one obtained by contracting all the edges of M.
2. The *M-derivation function* is defined via $g_M(v) = v$ if v is not a vertex of an edge of M, and $g_M(v) = g_M(w)$ is the vertex v' resulting from the contraction of vw of M.

We need one further lemma regarding algorithm parameters for the M-derived graph of G. Its proof is left to the reader (Exercise 10.6.3).

Lemma 10.5.4
1. *If $\{X_i : i \in \mathcal{T}\}$ is a treewidth k decomposition of G', where G' denotes the M-derived graph of G, then $\{Y_i : i \in \mathcal{T}\}$ defined via $Y_i = \{v \in V : g_M(v) \in X_i\}$ is a treewidth $\leq 2k + 1$ decomposition of G.*
2. *The treewidth of G' is less than or equal to that of G.*

For a graph G, we define the *improved graph* $\widehat{G}$ of G to be that obtained by adding an edge uv for all pairs u, v of V with at least $k + 1$ common neighbors in G.

Now we get to the key theorem. We need this preliminary definition.

Definition 10.5.3 ($\mathcal{T}$-simplicial vertex)
1. We say that a vertex v of G is *simplicial in G* if its set of neighbors form a clique in G.
2. A vertex w is called *$\mathcal{T}$-simplicial vertex* if it is simplicial in the improved graph $\widehat{G}$ of G, it is of low degree and it is not friendly in G.

We also need this decisive lemma, upon which Bodlaender's theorem and linear-time FPT algorithm are based.

Lemma 10.5.5 *Let $\{X'_i : i \in \mathcal{T}'\}$ be a tree decomposition of the graph G' obtained by removing all $\mathcal{T}$-simplicial vertices (and adjacent edges) from the improved graph $\widehat{G}$ of G. Then, for all $\mathcal{T}$-simplicial vertices v, there is some node i_v such that the set of neighbors of v in G all lie in X'_{i_v}.*

Proof See Exercise 10.6.4. □

We are now prepared to follow the central argument of Bodlaender's theorem into the actions of his algorithm.

Theorem 10.5.2 (Bodlaender [74]) *For every graph G, one of the following holds*:
(i) *G contains at least $\frac{|V|}{4k^2+12K+16}$ friendly vertices.*
(ii) *The improved graph $\widehat{G}$ of G contains at least $c_2|V|$ $\mathcal{T}$-simplicial vertices.*
(iii) *The treewidth of G exceeds k.*

Before we verify Theorem 10.5.2, we can now state Bodlaender's algorithm.

Algorithm 10.5.1 (Bodlaender's algorithm) For graphs larger than some fixed constant (which is computable), the following algorithm determines if a graph G has treewidth $\leq k$ and, if so, produces a treewidth k tree decomposition for G.

Step 1. Check if $|E| \leq k|V| - \frac{1}{2}k(k+1)$. If the answer is *no*, output that the treewidth of G exceeds k (Lemma 10.5.1(i)). If the answer is *yes*, go to step 2.

Step 2. There are two cases, depending on the number f of friendly vertices.

Case 1. $f \geq \frac{|V|}{4k^2+12K+16}$.

Action:

1. Find a maximal matching M in G.
2. Compute the derived graph (G', E') by contracting edges of M.
3. Recursively apply the algorithm to G'. If the treewidth of G' exceeds k, stop and output that the treewidth of G exceeds k (Lemma 10.5.4(ii)). Otherwise, the recursive call yields a treewidth k tree decomposition $\{X'_i : i \in \mathcal{T}'\}$ for G'. Now apply Lemma 10.5.4(i) to get a treewidth $2k+1$ tree decomposition $\{Y_i : i \in \mathcal{T}'\}$ of G.
4. Now apply the Bodlaender–Kloks algorithm to, for instance, the treewidth $2k+1$ tree decomposition of G, to either return that G has treewidth exceeding k or return a treewidth k tree decomposition of G.

Case 2. $f < \frac{|V|}{4k^2+12K+16}$.

Action:

1. Compute the improved graph $\widehat{G}$ of G. If there is a $\mathcal{T}$-simplicial vertex of degree $\geq k+1$, then stop and output that the treewidth of G exceeds k since $\widehat{G}$ contains a $k+2$-clique.
2. Otherwise, put all $\mathcal{T}$-simplicial vertices into some set S. Now, compute the graph G' obtained by deleting all $\mathcal{T}$-simplicial vertices and adjacent edges from G.
3. If $|S| < c_2|V|$, then stop and output that the treewidth of G exceeds k (Theorem 10.5.2(ii)).
4. Otherwise, recursively apply the algorithm to G'. If the treewidth of G' is greater than G, then the treewidth of G is greater than k, as G' is a subgraph of G. Otherwise, the algorithm will return a treewidth k tree decomposition $\{X'_i : i \in \mathcal{T}'\}$ for G'.
5. In that case, for all $v \in S$, find an $i_v \in \mathcal{T}$ such that the neighbors $N_G(v)$ of v in G all lie in bag X'_{i_v} (Lemma 10.5.5). We then add a new node j_v to $\mathcal{T}$ adjacent to i_v in $\mathcal{T}$ with $X_{j_v} = \{v\} \cup N_G(v)$.

The fact that the algorithm is correct is immediate by the lemmata preceding it. Note that we either recursively apply the algorithm on a graph with $(1 - \frac{1}{2d(4k^2+12k+16)})|V|$ vertices or on a graph of $(1-c_2)|V|$ vertices. Since both $1-c_2$ and $1 - \frac{1}{2d(4k^2+12k+16)}$ are greater than or equal to 0 and less than 1, it follows that the algorithm must run in linear time, given that the called nonrecursive parts have linear-time implementations, which they do.

Proving Bodlaender's Theorem 10.5.2 remains to be done. We do this in the following four lemmas. We say that a vertex v is *localized* with respect to a tree decomposition $\{X_i : i \in \mathcal{T}\}$ if it has low degree, it is not friendly, and for some bag X_i, all of the neighbors of v are in X_i.

Lemma 10.5.6 *Suppose that $\{X_i : i \in \mathcal{T}\}$ is a smooth tree decomposition of G of width k. Then,* (i) *and* (ii) *below hold.*

(i) *To each leaf i of $\mathcal{T}$ one can associate a vertex v_i which is either friendly or localized, and furthermore for all $j \neq i$, $v_i \notin X_j$.*

(ii) *To each path P of the form $i_0, \ldots, i_{k^2+3k+2}$ in $\mathcal{T}$ with i_j of degree 2 for $j \in \{0, \ldots, k^2+3k+2\}$, there is a vertex $v_P \in \bigcup_{j=0}^{k^2+3k+2} X_{i_j}$ such that v_P is either friendly or localized, and for all q not on the path P, $v_P \notin X_q$.*

Proof (i) Let i be a leaf of $\mathcal{T}$, and j a neighbor. Let v be the unique vertex in $X_i - X_j$. Note that the only neighbors of v are in X_i so it has low degree. If all the neighbors of v have high degree, then it is localized. Otherwise it is friendly.

(ii) $|\bigcup_{j=0}^{k^2+3k+2} X_{i_j}| = k^2 + 4k + 4$ (by smoothness) and hence $\leq d$. Hence, all the vertices in $(\bigcup_{j=0}^{k^2+3k+2} X_{i_j}) - (X_{i_0} \cup X_{i_{k^2+3k+2}})$ are of low degree. Suppose that all of them are unfriendly, and hence only adjacent to high-degree vertices in $X_{i_0} \cup X_{i_{k^2+3k+2}}$. Let $w_1, \ldots, w_r$ list the high-degree vertices in X_{i_0}. For $s \in \{1, \ldots, r\}$, we know by smoothness that there is a number n_s such that $w_s \in X_{i_0}, \ldots, X_{i_{n_s}}$. We may suppose that $n_1 \leq n_2 \leq \cdots \leq n_r$. If some vertex v belongs to exactly one X_{i_j} on the path P, it must be of low degree and hence localized. If some low-degree vertex belongs to only bags from $X_{i_{n_j}+1}, \ldots, X_{i_{n_j+1}}$, then it is localized since all of its neighbors must belong to $X_{i_{n_j+1}}$. All vertices in $\bigcup_{j=0}^{k^2+3k+2} X_{i_j}$ not of these two types must belong to at least one of the distinguished bags

$$X_{i_0}, X_{i_{n_1}}, \ldots, X_{i_{n_r}}, X_{i_{k^2+3k+2}}.$$

This is only a total of $k^2 + 4k + 3$ vertices, and the result follows. □

Now we demonstrate that we can nontrivially apply the lemma. We define a *leaf-path collection* to be a set of leaves of $\mathcal{T}$ plus a set of paths of $\mathcal{T}$ of length at least $k^2 + 3k + 4$. All the nodes of this leaf-path collection, except the endpoints, have degree 2 and do not belong to any other path in the collection.

Lemma 10.5.7

(i) *If $\mathcal{T}$ is any tree with n nodes then it must have a leaf-path collection of size $\geq \frac{n}{2k^2+6k+8}$ (i.e., it contains that many leaves and paths).*

(ii) *Consequently, if $\{X_i : i \in \mathcal{T}\}$ is a smooth tree decomposition of width k, then G has at least $\frac{|V|}{2k^2+6k+8} - 1$ vertices that are localized or friendly.*

Proof (i) This is a calculation. Let n_3 be the number of nodes in $\mathcal{T}$ of degree ≥ 3, n_1 the number of leaves, and n_2 the number of nodes of degree 2. We note that all nodes of degree 2 belong to $< n_1 + n_3$ connected components of the forest obtained from $\mathcal{T}$ by removing all nodes of degree ≥ 3. Each such component has at most $k^2 + 3k + 3$ nodes if it does not contribute to some leaf-path collection. Therefore, there are fewer than $(n_1 + n_3)(k^2 + 3k + 3)$ nodes of degree 2 not on a path in the leaf-path collection $\mathcal{C}$ of maximum size. The number of paths in $\mathcal{C}$ is at least:

$$\frac{n_2 - (n_1 + n_3)(k^2 + 3k + 3)}{k^2 + 3k + 4}.$$

Thus, the size of leaf-path collection $\mathcal{C}$ is at least:

$$\frac{n_2 - (n_1 + n_3)(k^2 + 3k + 3)}{k^2 + 3k + 4} + n_1 \geq \frac{\frac{1}{2}n}{k^2 + 3k + 4},$$

giving part (i). For (ii), by Lemma 10.5.2(ii), $\mathcal{T}$ has $|V| - k$ many nodes and, hence, an application of (i) and Lemma 10.5.6 yields the result. □

To fully prove Bodlaender's theorem, we need one more definition and two more lemmas.

Definition 10.5.4 (Important high-degree vertices) We say that a set of high-degree vertices $Y \subseteq V$ is *important* with respect to the decomposition $\{X_i : i \in \mathcal{T}\}$, if $Y \subseteq X_i$ for some $i \in \mathcal{T}$ and for any other set Y' of high-degree vertices with $Y' \subseteq X_j$ for some j, Y is not a proper subset of Y'.

Lemma 10.5.8 *Let $\{X_i : i \in \mathcal{T}\}$ be a decomposition of G of width k. The number of important sets is at most the number of high-degree vertices of G.*

Proof Let H be the set of high-degree vertices of G. Then, $\{X_i \cap H : i \in \mathcal{T}\}$ is a tree decomposition of the subgraph induced by H. Each important set Y is a set of the form $X_i \cap H$ not contained in any other $X_j \cap H$. Repeatedly contract the edges ii' with $X_i \cap H \subseteq X_{i'} \cap H$, forming a new node containing all the vertices of $X_{i'}$. Clearly, the resulting tree decomposition will have size at most H and will contain all the important sets of G. □

Fix a tree decomposition $\{X_i : i \in \mathcal{T}\}$ of G. Since localized vertices are not friendly, there is a function f, called a *localized to important function*, which associates each localized vertex v with an important set Y such that $N_G(v) \subseteq Y$.

Lemma 10.5.9 *Let f be such a localized to important function for a smooth decomposition $\{X_i : i \in \mathcal{T}\}$ of width k for G. Then, for any important set Y, there are at most $\frac{1}{2}k^2(k+1)$ localized vertices in $f^{-1}(Y)$ which are not $\mathcal{T}$-simplicial.*

Proof To each non-$\mathcal{T}$-simplicial localized vertex v, associate a pair of its neighbors that are nonadjacent in the improved graph. Notice that we cannot so associate more than k vertices with the same pair—lest the graph have $\geq k+1$ common low-degree vertices, which would necessarily have an edge between them in the improved graph. By simply counting, we see that the number of $\mathcal{T}$-simplicial vertices with $f(v) = Y$ is at most $\frac{1}{2}|Y|(|Y|-1)$, which is $\leq \frac{1}{2}k^2(k+1)$. □

Proof of Bodlaender's theorem, completed To complete the proof of Theorem 10.5.2, suppose that G contains fewer than $\frac{|V|}{4k^2+12k+16}$ friendly vertices and that the treewidth of G is at most k. By Lemma 10.5.3(i), there are at most $c_1|V|$ high-degree vertices in G. Therefore, by Lemma 10.5.8, the number of important sets for any smooth tree decomposition $\{X_i : i \in \mathcal{T}\}$ of G of width k is $\leq c_1|V|$. Therefore, by Lemma 10.5.9, at most $\frac{1}{2}k^2(k+1)(c_1|V|-1)$ localized vertices are not $\mathcal{T}$-simplicial. Using Lemma 10.5.7(ii), we see that

$$\frac{|V|}{2k^2+6k+8} - 1 - \frac{|V|}{4k^2+12k+16} - \frac{1}{2}k^2(k+1)(c_1|V|-1)c_2|V|$$

vertices are $\mathcal{T}$-simplicial, which establishes Theorem 10.5.2. □

10.6 Exercises

Exercise 10.6.1 Prove Lemma 10.5.1.
(Hint: For (i) consider a k-tree, for (ii) and (iii) use the interpolation property in the definition of tree decomposition, (iv) is easy, and for (v), argue that x and y must occur in some bag together since otherwise we could apply (iii) and then (ii) to get a $k+2$-clique in a treewidth k graph.)

Exercise 10.6.2 Prove Lemma 10.5.2.
(Hint: For (ii), use induction on $|V(\mathcal{T})|$.)

Exercise 10.6.3 Prove Lemma 10.5.4.

Exercise 10.6.4 Prove Lemma 10.5.5.
(Hint: Use Lemma 10.5.1.)

10.7 Historical Notes

The material of this section is drawn from the 1966 paper wherein Bodlaender [74] presented his theorem. The problem of deciding if a graph has treewidth k is NP-complete if k is allowed to vary, was discussed a decade later in Arnborg, Corneil, and Proskurowski [44]. In 1987 [44], those authors also showed that there was an $O(n^{k+2})$ for treewidth k graphs. Robertson and Seymour [588] gave the first FPT (it was $O(n^2)$) algorithm in 1995. Their algorithm was based upon the minor well-quasi-ordering theorem, and thus was highly nonconstructive and nonelementary

and had huge constants. Algorithms for recognition of bounded treewidth graphs have been improved, approximated, and parallelized since 1987 in the work of, for instance, Lagergren [475], Reed [575], Perkovíc and B. Reed [558], Fellows and Langston [297], and Matousek and Thomas [519].

Linear-time algorithms had been demonstrated in 1991 for recognition of treewidth k graphs $k \leq 3$ (e.g., Matousek and Thomas [519]). For a general k, linear-time recognition of treewidth k graphs from among treewidth $m \leq k$ graphs given by treewidth m decompositions is due independently to Lagergren and Arnborg, Abrahamson and Fellows, Bodlaender and Kloks [94] (where it first appears in print), although it follows also by Courcelle's theorem as we noted earlier. Bodlaender and Kloks gave the explicit algorithm which allows us to actually derive the relevant tree decomposition needed for Bodlaender's theorem. Their algorithm and Bodlaender's theorem are both double exponential in k, which is the best known constant. It would be very interesting if this could be reduced to a single exponential.

In the next chapter, we discuss heuristics for treewidth to optimize algorithm running time while finding tree decompositions.

10.8 Summary

This chapter introduces the reader to the key concepts of treewidth and tree decomposition. It is shown how to run dynamic programming on tree decompositions. Bodlaender's algorithm is proven.