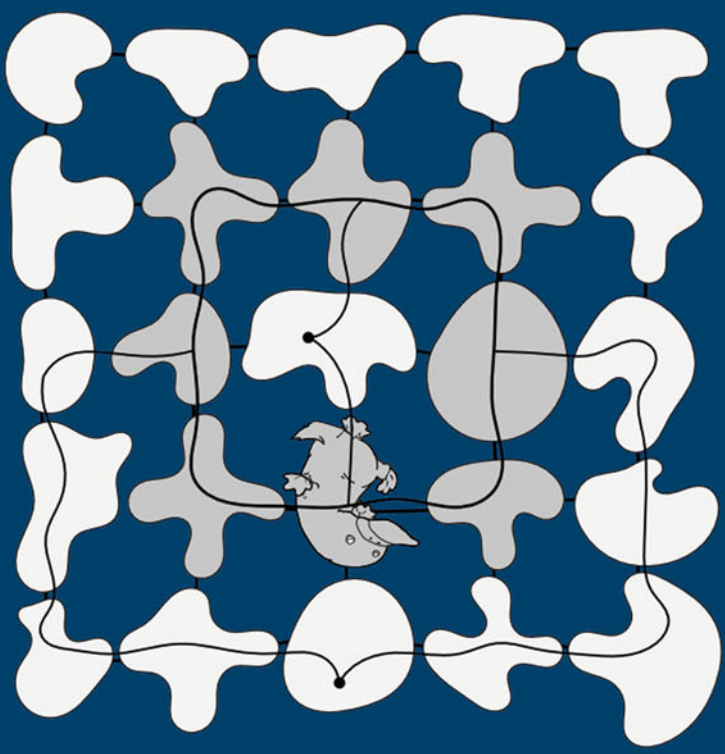
Marek Cygan · Fedor V. Fomin
Łukasz Kowalik · Daniel Lokshtanov
Dániel Marx · Marcin Pilipczuk
Michał Pilipczuk · Saket Saurabh

Parameterized Algorithms



Springer

Chapter 2
Kernelization

Kernelization is a systematic approach to study polynomial-time preprocessing algorithms. It is an important tool in the design of parameterized algorithms. In this chapter we explain basic kernelization techniques such as crown decomposition, the expansion lemma, the sunflower lemma, and linear programming. We illustrate these techniques by obtaining kernels for Vertex Cover, Feedback Arc Set in Tournaments, Edge Clique Cover, Maximum Satisfiability, *and* d-Hitting Set.

Preprocessing (data reduction or kernelization) is used universally in almost every practical computer implementation that aims to deal with an NP-hard problem. The goal of a preprocessing subroutine is to solve efficiently the "easy parts" of a problem instance and reduce it (shrink it) to its computationally difficult "core" structure (the *problem kernel* of the instance). In other words, the idea of this method is to reduce (but not necessarily solve) the given problem instance to an equivalent "smaller sized" instance in time polynomial in the input size. A slower exact algorithm can then be run on this smaller instance.

How can we measure the effectiveness of such a preprocessing subroutine? Suppose we define a useful preprocessing algorithm as one that runs in polynomial time and replaces an instance I with an equivalent instance that is at least one bit smaller. Then the existence of such an algorithm for an NP-hard problem would imply P$=$ NP, making it unlikely that such an algorithm can be found. For a long time, there was no other suggestion for a formal definition of useful preprocessing, leaving the mathematical analysis of polynomial-time preprocessing algorithms largely neglected. But in the language of parameterized complexity, we can formulate a definition of useful preprocessing by demanding that large instances with a small parameter should be shrunk, while instances that are small compared to their parameter

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_2

2.2.1 Vertex Cover

Let G be a graph and $S \subseteq V(G)$. The set S is called a *vertex cover* if for every edge of G at least one of its endpoints is in S. In other words, the graph $G - S$ contains no edges and thus $V(G) \setminus S$ is an *independent set*. In the Vertex Cover problem, we are given a graph G and a positive integer k as input, and the objective is to check whether there exists a vertex cover of size at most k.

The first reduction rule is based on the following simple observation. For a given instance (G, k) of Vertex Cover, if the graph G has an isolated vertex, then this vertex does not cover any edge and thus its removal does not change the solution. This shows that the following rule is safe.

Reduction VC.1. If G contains an isolated vertex v, delete v from G. The new instance is $(G - v, k)$.

The second rule is based on the following natural observation:

> If G contains a vertex v of degree more than k, then v should be in every vertex cover of size at most k.

Indeed, this is because if v is not in a vertex cover, then we need at least $k + 1$ vertices to cover edges incident to v. Thus our second rule is the following.

Reduction VC.2. If there is a vertex v of degree at least $k + 1$, then delete v (and its incident edges) from G and decrement the parameter k by 1. The new instance is $(G - v, k - 1)$.

Observe that exhaustive application of reductions VC.1 and VC.2 completely removes the vertices of degree 0 and degree at least $k + 1$. The next step is the following observation.

> If a graph has maximum degree d, then a set of k vertices can cover at most kd edges.

This leads us to the following lemma.

Lemma 2.3. *If (G, k) is a yes-instance and none of the reduction rules VC.1, VC.2 is applicable to G, then $|V(G)| \leq k^2 + k$ and $|E(G)| \leq k^2$.*

Proof. Because we cannot apply Reductions VC.1 anymore on G, G has no isolated vertices. Thus for every vertex cover S of G, every vertex of $G - S$ should be adjacent to some vertex from S. Since we cannot apply Reductions VC.2, every vertex of G has degree at most k. It follows that

$|V(G - S)| \leq k|S|$ and hence $|V(G)| \leq (k+1)|S|$. Since (G, k) is a yes-instance, there is a vertex cover S of size at most k, so $|V(G)| \leq (k+1)k$. Also every edge of G is covered by some vertex from a vertex cover and every vertex can cover at most k edges. Hence if G has more than k^2 edges, this is again a no-instance. □

Lemma 2.3 allows us to claim the final reduction rule that explicitly bounds the size of the kernel.

Reduction VC.3. Let (G, k) be an input instance such that Reductions VC.1 and VC.2 are not applicable to (G, k). If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

Finally, we remark that all reduction rules are trivially applicable in linear time. Thus, we obtain the following theorem.

Theorem 2.4. Vertex Cover *admits a kernel with* $\mathcal{O}(k^2)$ *vertices and* $\mathcal{O}(k^2)$ *edges.*

2.2.2 Feedback Arc Set in Tournaments

In this section we discuss a kernel for the Feedback Arc Set in Tournaments problem. A *tournament* is a directed graph T such that for every pair of vertices $u, v \in V(T)$, exactly one of (u, v) or (v, u) is a directed edge (also often called an *arc*) of T. A set of edges A of a directed graph G is called a *feedback arc set* if every directed cycle of G contains an edge from A. In other words, the removal of A from G turns it into a directed acyclic graph. Very often, acyclic tournaments are called *transitive* (note that then $E(G)$ is a transitive relation). In the Feedback Arc Set in Tournaments problem we are given a tournament T and a nonnegative integer k. The objective is to decide whether T has a feedback arc set of size at most k.

For tournaments, the deletion of edges results in directed graphs which are not tournaments anymore. Because of that, it is much more convenient to use the characterization of a feedback arc set in terms of "reversing edges". We start with the following well-known result about *topological orderings* of directed acyclic graphs.

Lemma 2.5. *A directed graph G is acyclic if and only if it is possible to order its vertices in such a way such that for every directed edge (u, v), we have $u < v$.*

We leave the proof of Lemma 2.5 as an exercise; see Exercise 2.1. Given a directed graph G and a subset $F \subseteq E(G)$ of edges, we define $G \circledast F$ to be the directed graph obtained from G by reversing all the edges of F. That is, if $\mathrm{rev}(F) = \{(u, v) \ : \ (v, u) \in F\}$, then for $G \circledast F$ the vertex set is $V(G)$

and the edge set $E(G \circledast F) = (E(G) \cup \mathrm{rev}(F)) \setminus F$. Lemma 2.5 implies the following.

Observation 2.6. Let G be a directed graph and let F be a subset of edges of G. If $G \circledast F$ is a directed acyclic graph then F is a feedback arc set of G.

The following lemma shows that, in some sense, the opposite direction of the statement in Observation 2.6 is also true. However, the minimality condition in Lemma 2.7 is essential, see Exercise 2.2.

Lemma 2.7. *Let G be a directed graph and F be a subset of $E(G)$. Then F is an inclusion-wise minimal feedback arc set of G if and only if F is an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph.*

Proof. We first prove the forward direction of the lemma. Let F be an inclusion-wise minimal feedback arc set of G. Assume to the contrary that $G \circledast F$ has a directed cycle C. Then C cannot contain only edges of $E(G) \setminus F$, as that would contradict the fact that F is a feedback arc set. Let $f_1, f_2, \cdots, f_\ell$ be the edges of $C \cap \mathrm{rev}(F)$ in the order of their appearance on the cycle C, and let $e_i \in F$ be the edge f_i reversed. Since F is inclusion-wise minimal, for every e_i, there exists a directed cycle C_i in G such that $F \cap C_i = \{e_i\}$. Now consider the following closed walk W in G: we follow the cycle C, but whenever we are to traverse an edge $f_i \in \mathrm{rev}(F)$ (which is not present in G), we instead traverse the path $C_i - e_i$. By definition, W is a closed walk in G and, furthermore, note that W does not contain any edge of F. This contradicts the fact that F is a feedback arc set of G.

The minimality follows from Observation 2.6. That is, every set of edges F such that $G \circledast F$ is acyclic is also a feedback arc set of G, and thus, if F is not a minimal set such that $G \circledast F$ is acyclic, then it will contradict the fact that F is a minimal feedback arc set.

For the other direction, let F be an inclusion-wise minimal set of edges such that $G \circledast F$ is an acyclic directed graph. By Observation 2.6, F is a feedback arc set of G. Moreover, F is an inclusion-wise minimal feedback arc set, because if a proper subset F' of F is an inclusion-wise minimal feedback arc set of G, then by the already proved implication of the lemma, $G \circledast F'$ is an acyclic directed graph, a contradiction with the minimality of F. □

We are ready to give a kernel for Feedback Arc Set in Tournaments.

Theorem 2.8. Feedback Arc Set in Tournaments *admits a kernel with at most $k^2 + 2k$ vertices.*

Proof. Lemma 2.7 implies that a tournament T has a feedback arc set of size at most k if and only if it can be turned into an acyclic tournament by reversing directions of at most k edges. We will use this characterization for the kernel.

In what follows by a *triangle* we mean a directed cycle of length three. We give two simple reduction rules.

Reduction FAST.1. If an edge e is contained in at least $k+1$ triangles, then reverse e and reduce k by 1.

Reduction FAST.2. If a vertex v is not contained in any triangle, then delete v from T.

The rules follow similar guidelines as in the case of Vertex Cover. In Reduction FAST.1, we greedily take into a solution an edge that participates in $k+1$ otherwise disjoint forbidden structures (here, triangles). In Reduction FAST.2, we discard vertices that do not participate in any forbidden structure, and should be irrelevant to the problem.

However, a formal proof of the safeness of Reduction FAST.2 is not immediate: we need to verify that deleting v and its incident edges does not make make a yes-instance out of a no-instance.

Note that after applying any of the two rules, the resulting graph is again a tournament. The first rule is safe because if we do not reverse e, we have to reverse at least one edge from each of $k+1$ triangles containing e. Thus e belongs to every feedback arc set of size at most k.

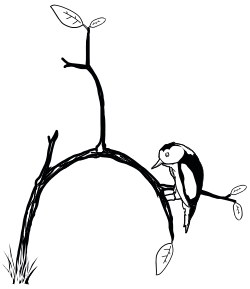
Let us now prove the safeness of the second rule. Let $X = N^+(v)$ be the set of heads of directed edges with tail v and let $Y = N^-(v)$ be the set of tails of directed edges with head v. Because T is a tournament, X and Y is a partition of $V(T) \setminus \{v\}$. Since v is not a part of any triangle in T, we have that there is no edge from X to Y (with head in Y and tail in X). Consequently, for any feedback arc set A_1 of tournament $T[X]$ and any feedback arc set A_2 of tournament $T[Y]$, the set $A_1 \cup A_2$ is a feedback arc set of T. As the reverse implication is trivial (for any feedback arc set A in T, $A \cap E(T[X])$ is a feedback arc set of $T[X]$, and $A \cap E(T[Y])$ is a feedback arc set of $T[Y]$), we have that (T, k) is a yes-instance if and only if $(T - v, k)$ is.

Finally, we show that every reduced yes-instance T, an instance on which none of the presented reduction rules are applicable, has at most $k(k+2)$ vertices. Let A be a feedback arc set of a reduced instance T of size at most k. For every edge $e \in A$, aside from the two endpoints of e, there are at most k vertices that are in triangles containing e — otherwise we would be able to apply Reduction FAST.1. Since every triangle in T contains an edge of A and every vertex of T is in a triangle, we have that T has at most $k(k+2)$ vertices.

Thus, given (T, k) we apply our reduction rules exhaustively and obtain an equivalent instance (T', k'). If T' has more than $k'^2 + k'$ vertices, then the algorithm returns that (T, k) is a no-instance, otherwise we get the desired kernel. This completes the proof of the theorem. □

Chapter 3
Bounded search trees

In this chapter we introduce a variant of exhaustive search, namely the method of bounded search trees. This is one of the most commonly used tools in the design of fixed-parameter algorithms. We illustrate this technique with algorithms for two different parameterizations of Vertex Cover*, as well as for the problems (undirected)* Feedback Vertex Set *and* Closest String.



Bounded search trees, or simply *branching*, is one of the simplest and most commonly used techniques in parameterized complexity that originates in the general idea of backtracking. The algorithm tries to build a feasible solution to the problem by making a sequence of decisions on its shape, such as whether to include some vertex into the solution or not. Whenever considering one such step, the algorithm investigates many possibilities for the decision, thus effectively *branching* into a number of subproblems that are solved one by one. In this manner the execution of a branching algorithm can be viewed as a *search tree*, which is traversed by the algorithm up to the point when a solution is discovered in one of the leaves. In order to justify the correctness of a branching algorithm, one needs to argue that in case of a yes-instance some sequence of decisions captured by the algorithm leads to a feasible solution. If the total size of the search tree is bounded by a function of the parameter alone, and every step takes polynomial time, then such a branching algorithm runs in FPT time. This is indeed the case for many natural backtracking algorithms.

More precisely, let I be an instance of a minimization problem (such as Vertex Cover). We associate a measure $\mu(I)$ with the instance I, which, in the case of FPT algorithms, is usually a function of k alone. In a branch step we generate from I simpler instances $I_1, \ldots, I_\ell$ ($\ell \geq 2$) of the same problem such that the following hold.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_3

nodes of a search tree. In Section 3.3 we give a branching algorithm for Feedback Vertex Set in undirected graphs. Section 3.4 presents an algorithm for a different parameterization of Vertex Cover and shows how this algorithm implies algorithms for other parameterized problems such as Odd Cycle Transversal and Almost 2-SAT. Finally, in Section 3.5 we apply this technique to a non-graph problem, namely Closest String.

3.1 Vertex Cover

As the first example of branching, we use the strategy on Vertex Cover. In Chapter 2 (Lemma 2.23), we gave a kernelization algorithm which in time $\mathcal{O}(n\sqrt{m})$ constructs a kernel on at most $2k$ vertices. Kernelization can be easily combined with a brute-force algorithm to solve Vertex Cover in time $\mathcal{O}(n\sqrt{m}+4^k k^{\mathcal{O}(1)})$. Indeed, there are at most $2^{2k} = 4^k$ subsets of size at most k in a $2k$-vertex graph. Thus, by enumerating all vertex subsets of size at most k in the kernel and checking whether any of these subsets forms a vertex cover, we can solve the problem in time $\mathcal{O}(n\sqrt{m} + 4^k k^{\mathcal{O}(1)})$. We can easily obtain a better algorithm by branching. Actually, this algorithm was already presented in Chapter 1 under the cover of the Bar Fight Prevention problem.

Let (G, k) be a Vertex Cover instance. Our algorithm is based on the following two simple observations.

- For a vertex v, any vertex cover must contain either v or *all* of its neighbors $N(v)$.
- Vertex Cover becomes trivial (in particular, can be solved optimally in polynomial time) when the maximum degree of a graph is at most 1.

We now describe our recursive branching algorithm. Given an instance (G, k), we first find a vertex $v \in V(G)$ of maximum degree in G. If v is of degree 1, then every connected component of G is an isolated vertex or an edge, and the instance has a trivial solution. Otherwise, $|N(v)| \geq 2$ and we recursively branch on two cases by considering

either v, or $N(v)$ in the vertex cover.

In the branch where v is in the vertex cover, we can delete v and reduce the parameter by 1. In the second branch, we add $N(v)$ to the vertex cover, delete $N[v]$ from the graph and decrease k by $|N(v)| \geq 2$.

The running time of the algorithm is bounded by

(the number of nodes in the search tree) × (time taken at each node).

Clearly, the time taken at each node is bounded by $n^{\mathcal{O}(1)}$. Thus, if $\tau(k)$ is the number of nodes in the search tree, then the total time used by the algorithm is at most $\tau(k)n^{\mathcal{O}(1)}$.

In fact, in every search tree $\mathcal{T}$ that corresponds to a run of a branching algorithm, every internal node of $\mathcal{T}$ has at least two children. Thus, if $\mathcal{T}$ has ℓ leaves, then the number of nodes in the search tree is at most $2\ell - 1$. Hence, to bound the running time of a branching algorithm, it is sufficient to bound the number of leaves in the corresponding search tree.

In our case, the tree $\mathcal{T}$ is the search tree of the algorithm when run with parameter k. Below its root, it has two subtrees: one for the same algorithm run with parameter $k-1$, and one recursive call with parameter at most $k-2$. The same pattern occurs deeper in $\mathcal{T}$. This means that if we define a function $T(k)$ using the recursive formula

$$T(i) = \begin{cases} T(i-1) + T(i-2) & \text{if } i \geq 2, \\ 1 & \text{otherwise,} \end{cases}$$

then the number of leaves of $\mathcal{T}$ is bounded by $T(k)$.

Using induction on k, we prove that $T(k)$ is bounded by 1.6181^k. Clearly, this is true for $k = 0$ and $k = 1$, so let us proceed for $k \geq 2$:

$$\begin{aligned} T(k) &= T(k-1) + T(k-2) \leq 1.6181^{k-1} + 1.6181^{k-2} \\ &\leq 1.6181^{k-2}(1.6181 + 1) \leq 1.6181^{k-2}(1.6181)^2 \leq 1.6181^k. \end{aligned}$$

This proves that the number of leaves is bounded by 1.6181^k. Combined with kernelization, we arrive at an algorithm solving Vertex Cover in time $\mathcal{O}(n\sqrt{m} + 1.6181^k k^{\mathcal{O}(1)})$.

A natural question is how did we know that 1.6181^k is a solution to the above recurrence. Suppose that we are looking for an upper bound on function $T(k)$ of the form $T(k) \leq c \cdot \lambda^k$, where $c > 0$, $\lambda > 1$ are some constants. Clearly, we can set constant c so that the initial conditions in the definition of $T(k)$ are satisfied. Then, we are left with proving, using induction, that this bound holds for every k. This boils down to proving that

$$c \cdot \lambda^k \geq c \cdot \lambda^{k-1} + c \cdot \lambda^{k-2}, \tag{3.1}$$

since then we will have

$$T(k) = T(k-1) + T(k-2) \leq c \cdot \lambda^{k-1} + c \cdot \lambda^{k-2} \leq c \cdot \lambda^k.$$

Observe that (3.1) is equivalent to $\lambda^2 \geq \lambda + 1$, so it makes sense to look for the lowest possible value of λ for which this inequality is satisfied; this is actually the one for which equality holds. By solving equation $\lambda^2 = \lambda + 1$ for $\lambda > 1$, we find that $\lambda = \frac{1+\sqrt{5}}{2} < 1.6181$, so for this value of λ the inductive proof works.

The running time of the above algorithm can be easily improved using the following argument, whose proof we leave as Exercise 3.1.

Proposition 3.1. Vertex Cover *can be solved optimally in polynomial time when the maximum degree of a graph is at most* 2.

Thus, we branch only on the vertices of degree at least 3, which immediately brings us to the following upper bound on the number of leaves in a search tree:

$$T(k) = \begin{cases} T(k-1) + T(k-3) & \text{if } k \geq 3, \\ 1 & \text{otherwise.} \end{cases}$$

Again, an upper bound of the form $c \cdot \lambda^k$ for the above recursive function can be obtained by finding the largest root of the polynomial equation $\lambda^3 = \lambda^2 + 1$. Using standard mathematical techniques (and/or symbolic algebra packages) the root is estimated to be at most 1.4656. Combined with kernelization, this gives us the following theorem.

Theorem 3.2. Vertex Cover *can be solved in time* $\mathcal{O}(n\sqrt{m} + 1.4656^k k^{\mathcal{O}(1)})$.

Can we apply a similar strategy for graphs of vertex degree at most 3? Well, this becomes more complicated as Vertex Cover is NP-hard on this class of graphs. But there are more involved branching strategies, and there are faster branching algorithms than the one given in Theorem 3.2.

3.2 How to solve recursive relations

For algorithms based on the bounded search tree technique, we need to bound the number of nodes in the search tree to obtain an upper bound on the running time of the algorithm. For this, recurrence relations are used. The most common case in parameterized branching algorithms is when we use linear recurrences with constant coefficients. There exists a standard technique to bound the number of nodes in the search tree for this case. If the algorithm solves a problem of size n with parameter k and calls itself recursively on problems with decreased parameters $k - d_1, k - d_2, \ldots, k - d_p$, then $(d_1, d_2, \ldots, d_p)$ is called the *branching vector* of this recursion. For example, we used a branching vector $(1, 2)$ to obtain the first algorithm for Vertex Cover in the previous section, and a branching vector $(1, 3)$ for the second one. For a branching vector $(d_1, d_2, \ldots, d_p)$, the upper bound $T(k)$

The answer to the first question is "it is good": up to a polynomial factor, the estimation is tight. The second question is much more difficult, since the actual way a branching procedure explores the search space may be more complex than our estimation of its behavior using recursive formulas. If, say, a branching algorithm uses several ways of branching into subproblems (so-called branching rules) that correspond to different branching vectors, and/or is combined with local reduction rules, then so far we do not know how to estimate the running time better than by using the branching number corresponding to the worst branching vector. However, the delicate interplay between different branching rules and reduction rules may lead to a much smaller tree than what follows from our imprecise estimations.

3.3 Feedback Vertex Set

For a given graph G and a set $X \subseteq V(G)$, we say that X is a *feedback vertex set* of G if $G-X$ is an acyclic graph (i.e., a forest). In the Feedback Vertex Set problem, we are given an undirected graph G and a nonnegative integer k, and the objective is to determine whether there exists a feedback vertex set of size at most k in G. In this section, we give a branching algorithm solving Feedback Vertex Set in time $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$.

It is more convenient for us to consider this problem in the more general setting of *multigraphs*, where the input graph G may contain multiple edges and loops. We note that both a double edge and a loop are cycles. We also use the convention that a loop at a vertex v contributes 2 to the degree of v.

We start with some simple reduction rules that clean up the graph. At any point, we use the lowest-numbered applicable rule. We first deal with the multigraph parts of G. Observe that any vertex with a loop needs to be contained in any solution set X.

Reduction FVS.1. If there is a loop at a vertex v, delete v from the graph and decrease k by 1.

Moreover, notice that the multiplicity of a multiple edge does not influence the set of feasible solutions to the instance (G, k).

Reduction FVS.2. If there is an edge of multiplicity larger than 2, reduce its multiplicity to 2.

We now reduce vertices of low degree. Any vertex of degree at most 1 does not participate in any cycle in G, so it can be deleted.

Reduction FVS.3. If there is a vertex v of degree at most 1, delete v.

Concerning vertices of degree 2, observe that, instead of including into the solution any such vertex, we may as well include one of its neighbors. This leads us to the following reduction.

Reduction FVS.4. If there is a vertex v of degree 2, delete v and connect its two neighbors by a new edge.

Two remarks are in place. First, a vertex v in Reduction FVS.4 cannot have a loop, as otherwise Reduction FVS.1 should be triggered on v instead. This ensures that the neighbors of v are distinct from v, hence the rule may be applied and the safeness argument works. Second, it is possible that v is incident to a double edge: in this case, the reduction rule deletes v and adds a loop to a sole "double" neighbor of v. Observe that in this case Reduction FVS.1 will trigger subsequently on this neighbor.

We remark that after exhaustively applying these four reduction rules, the resulting graph G

(P1) contains no loops,
(P2) has only single and double edges, and
(P3) has minimum vertex degree at least 3.

Moreover, all rules are trivially applicable in polynomial time. From now on we assume that in the input instance (G, k), graph G satisfies properties (P1)–(P3).

We remark that for the algorithm in this section, we do not need properties (P1) and (P2). However, we will need these properties later for the kernelization algorithm in Section 9.1.

Finally, we need to add a rule that stops the algorithm if we already exceeded our budget.

Reduction FVS.5. If $k < 0$, terminate the algorithm and conclude that (G, k) is a no-instance.

The intuition behind the algorithm we are going to present is as follows. Observe that if X is a feedback vertex set of G, then $G - X$ is a forest. However, $G - X$ has at most $|V(G)| - |X| - 1$ edges and thus $G - X$ cannot have "many" vertices of high degree. Thus, if we pick some $f(k)$ vertices with the highest degrees in the graph, then every solution of size at most k must contain one of these high-degree vertices. In what follows we make this intuition work.

Let $(v_1, v_2, \ldots, v_n)$ be a descending ordering of $V(G)$ according to vertex degrees, i.e., $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$. Let $V_{3k} = \{v_1, \ldots, v_{3k}\}$. Let us recall that the minimum vertex degree of G is at least 3. Our algorithm for Feedback Vertex Set is based on the following lemma.

Lemma 3.3. *Every feedback vertex set in G of size at most k contains at least one vertex of V_{3k}.*

Proof. To prove this lemma we need the following simple claim.

Claim 3.4. *For every feedback vertex set X of G,*

$$\sum_{v \in X} (d(v) - 1) \geq |E(G)| - |V(G)| + 1.$$

Proof. Graph $F = G - X$ is a forest and thus the number of edges in F is at most $|V(G)| - |X| - 1$. Every edge of $E(G) \setminus E(F)$ is incident to a vertex of X. Hence

$$\sum_{v \in X} d(v) + |V(G)| - |X| - 1 \geq |E(G)|.$$

⌟

Targeting a contradiction, let us assume that there is a feedback vertex set X of size at most k such that $X \cap V_{3k} = \emptyset$. By the choice of V_{3k}, for every $v \in X$, $d(v)$ is at most the minimum of vertex degrees from V_{3k}. Because $|X| \leq k$, by Claim 3.4 we have that

$$\sum_{i=1}^{3k} (d(v_i) - 1) \geq 3 \cdot \Big(\sum_{v \in X} (d(v) - 1) \Big) \geq 3 \cdot (|E(G)| - |V(G)| + 1).$$

In addition, we have that $X \subseteq V(G) \setminus V_{3k}$, and hence

$$\sum_{i > 3k} (d(v_i) - 1) \geq \sum_{v \in X} (d(v) - 1) \geq (|E(G)| - |V(G)| + 1).$$

Therefore,

$$\sum_{i=1}^{n} (d(v_i) - 1) \geq 4 \cdot (|E(G)| - |V(G)| + 1).$$

However, observe that $\sum_{i=1}^{n} d(v_i) = 2|E(G)|$: every edge is counted twice, once for each of its endpoints. Thus we obtain

$$4 \cdot (|E(G)| - |V(G)| + 1) \leq \sum_{i=1}^{n} (d(v_i) - 1) = 2|E(G)| - |V(G)|,$$

which implies that $2|E(G)| < 3|V(G)|$. However, this contradicts the fact that every vertex of G is of degree at least 3. □

We use Lemma 3.3 to obtain the following algorithm for Feedback Vertex Set.

Theorem 3.5. *There exists an algorithm for* Feedback Vertex Set *running in time* $(3k)^k \cdot n^{\mathcal{O}(1)}$.

Proof. Given an undirected graph G and an integer $k \geq 0$, the algorithm works as follows. It first applies Reductions FVS.1, FVS.2, FVS.3, FVS.4,

and FVS.5 exhaustively. As the result, we either already conclude that we are dealing with a no-instance, or obtain an equivalent instance (G', k') such that G' has minimum degree at least 3 and $k' \leq k$. If G' is empty, then we conclude that we are dealing with a yes-instance, as $k' \geq 0$ and an empty set is a feasible solution. Otherwise, let $V_{3k'}$ be the set of $3k'$ vertices of G' with largest degrees. By Lemma 3.3, every solution X to the Feedback Vertex Set instance (G', k') contains at least one vertex from $V_{3k'}$. Therefore, we branch on the choice of one of these vertices, and for every vertex $v \in V_{3k'}$, we recursively apply the algorithm to solve the Feedback Vertex Set instance $(G' - v, k' - 1)$. If one of these branches returns a solution X', then clearly $X' \cup \{v\}$ is a feedback vertex set of size at most k' for G'. Else, we return that the given instance is a no-instance.

At every recursive call we decrease the parameter by 1, and thus the height of the search tree does not exceed k'. At every step we branch in at most $3k'$ subproblems. Hence the number of nodes in the search tree does not exceed $(3k')^{k'} \leq (3k)^k$. This concludes the proof. □

3.4 Vertex Cover Above LP

Recall the integer linear programming formulation of Vertex Cover and its relaxation LPVC(G):

$$\begin{array}{lll} \min & \sum_{v \in V(G)} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every } uv \in E(G), \\ & 0 \leq x_v \leq 1 & \text{for every } v \in V(G). \end{array}$$

These programs were discussed in Section 2.2.1. If the minimum value of LPVC(G) is $\mathrm{vc}^*(G)$, then the size of a minimum vertex cover is at least $\mathrm{vc}^*(G)$. This leads to the following parameterization of Vertex Cover, which we call Vertex Cover Above LP: Given a graph G and an integer k, we ask for a vertex cover of G of size at most k, but instead of seeking an FPT algorithm parameterized by k as for Vertex Cover, the parameter now is $k - \mathrm{vc}^*(G)$. In other words, the goal of this section is to design an algorithm for Vertex Cover on an n-vertex graph G with running time $f(k - \mathrm{vc}^*(G)) \cdot n^{\mathcal{O}(1)}$ for some computable function f.

The parameterization by $k - \mathrm{vc}^*(G)$ falls into a more general theme of *above guarantee parameterization*, where, instead of parameterizing purely by the solution size k, we look for some (computable in polynomial time) lower bound ℓ for the solution size, and use a more refined parameter $k - \ell$, the *excess* above the lower bound. Such a parameterization makes perfect sense in problems where the solution size is often quite large and, consequently, FPT algorithms in parameterization by k may not be very efficient. In the Vertex Cover problem, the result of Section 2.5 — a kernel with at

most $2k$ vertices — can be on one hand seen as a very good result, but on the other hand can be an evidence that the solution size parameterization for Vertex Cover may not be the most meaningful one, as the parameter can be quite large. A more striking example is the Maximum Satisfiability problem, studied in Section 2.3.2: here an instance with at least $2k$ clauses is trivially a yes-instance. In Section *9.2 we study an above guarantee parameterization of a variant of Maximum Satisfiability, namely Max-Er-SAT. In Exercise 9.3 we also ask for an FPT algorithm for Maximum Satisfiability parameterized by $k - m/2$, where m is the number of clauses in the input formula. The goal of this section is to study the above guarantee parameterization of Vertex Cover, where the lower bound is the cost of an optimum solution to an LP relaxation.

Before we describe the algorithm, we fix some notation. By *optimum solution* $\mathbf{x} = (x_v)_{v \in V(G)}$ to LPVC(G), we mean a feasible solution with $1 \geq x_v \geq 0$ for all $v \in V(G)$ that minimizes the objective function (sometimes called the *cost*) $\mathbf{w}(\mathbf{x}) = \sum_{v \in V(G)} x_v$. By Proposition 2.24, for any graph G there exists an optimum half-integral solution of LPVC(G), i.e., a solution with $x_v \in \{0, \frac{1}{2}, 1\}$ for all $v \in V(G)$, and such a solution can be found in polynomial time.

Let $\mathrm{vc}(G)$ denote the size of a minimum vertex cover of G. Clearly, $\mathrm{vc}(G) \geq \mathrm{vc}^*(G)$. For a half-integral solution $\mathbf{x} = (x_v)_{v \in V(G)}$ and $i \in \{0, \frac{1}{2}, 1\}$, we define $V_i^{\mathbf{x}} = \{v \in V \ : \ x_v = i\}$. We also say that $\mathbf{x} = \{x_v\}_{v \in V(G)}$ is *all-$\frac{1}{2}$-solution* if $x_v = \frac{1}{2}$ for every $v \in V(G)$. Because the all-$\frac{1}{2}$-solution is a feasible solution, we have that $\mathrm{vc}^*(G) \leq \frac{|V(G)|}{2}$. Furthermore, we define the *measure* of an instance (G, k) to be our parameter of interest $\mu(G, k) = k - \mathrm{vc}^*(G)$.

Recall that in Section 2.5 we have developed Reduction VC.4 for Vertex Cover. This reduction, if restricted to half-integral solutions, can be stated as follows: for an optimum half-integral LPVC(G) solution $\mathbf{x}$, we (a) conclude that the input instance (G, k) is a no-instance if $\mathbf{w}(\mathbf{x}) > k$; and (b) delete $V_0^{\mathbf{x}} \cup V_1^{\mathbf{x}}$ and decrease k by $|V_1^{\mathbf{x}}|$ otherwise. As we are now dealing with measure $\mu(G, k) = k - \mathrm{vc}^*(G)$, we need to understand how this parameter changes under Reduction VC.4.

Lemma 3.6. *Assume an instance (G', k') is created from an instance (G, k) by applying Reduction VC.4 to a half-integral optimum solution $\mathbf{x}$. Then $\mathrm{vc}^*(G) - \mathrm{vc}^*(G') = \mathrm{vc}(G) - \mathrm{vc}(G') = |V_1^{\mathbf{x}}| = k - k'$. In particular, $\mu(G', k') = \mu(G, k)$.*

We remark that, using Exercise 2.25, the statement of Lemma 3.6 is true for any optimum solution $\mathbf{x}$, not only a half-integral one (see Exercise 3.19). However, the proof for a half-integral solution is slightly simpler, and, thanks to Proposition 2.24, we may work only with half-integral solutions.

Proof (of Lemma 3.6). Observe that every edge of G incident to $V_0^{\mathbf{x}}$ has its second endpoint in $V_1^{\mathbf{x}}$. Hence, we have the following:

- For every vertex cover Y of G', $Y \cup V_1^{\mathbf{x}}$ is a vertex cover of G and, consequently, $\mathrm{vc}(G) \leq \mathrm{vc}(G') + |V_1^{\mathbf{x}}|$.
- For every feasible solution $\mathbf{y}'$ to LPVC(G'), if we define a vector $\mathbf{y} = (y_v)_{v \in V(G)}$ as $y_v = y'_v$ for every $v \in V(G')$ and $y_v = x_v$ for every $v \in V(G) \setminus V(G')$, then we obtain a feasible solution to LPVC(G) of cost $\mathbf{w}(\mathbf{y}') + |V_1^{\mathbf{x}}|$; consequently, $\mathrm{vc}^*(G) \leq \mathrm{vc}^*(G') + |V_1^{\mathbf{x}}|$.

Now it suffices to prove the reversed versions of the two inequalities obtained above. Theorem 2.19 ensures that $\mathrm{vc}(G') \leq \mathrm{vc}(G) - |V_1^{\mathbf{x}}|$. Moreover, since $\mathbf{x}$ restricted to $V(G')$ is a feasible solution to LPVC(G') of cost $\mathrm{vc}^*(G) - |V_1^{\mathbf{x}}|$, we have $\mathrm{vc}^*(G') \leq \mathrm{vc}^*(G) - |V_1^{\mathbf{x}}|$. □

In Theorem 2.21 we simply solved LPVC(G), and applied Reduction VC.4 to obtain a kernel with at most $2k$ vertices. In this section we would like to do something slightly stronger: to apply Reduction VC.4 as long as there exists some half-integral optimum solution $\mathbf{x}$ that is not the all-$\frac{1}{2}$-solution. Luckily, by a simple self-reduction trick, we can always detect such solutions.

Lemma 3.7. *Given a graph G, one can in $\mathcal{O}(mn^{3/2})$ time find an optimum solution to LPVC(G) which is not the all-$\frac{1}{2}$-solution, or correctly conclude that the all-$\frac{1}{2}$-solution is the unique optimum solution to LPVC(G). Moreover, the returned optimum solution is half-integral.*

Proof. First, use Proposition 2.24 to solve LPVC(G), obtaining an optimum half-integral solution $\mathbf{x}$. If $\mathbf{x}$ is not the all-$\frac{1}{2}$-solution, then return $\mathbf{x}$. Otherwise, proceed as follows.

For every $v \in V(G)$, use Proposition 2.24 again to solve LPVC($G - v$), obtaining an optimum half-integral solution $\mathbf{x}^v$. Define a vector $\mathbf{x}^{v,\circ}$ as $\mathbf{x}^v$, extended with a value $x_v = 1$. Note that $\mathbf{x}^{v,\circ}$ is a feasible solution to LPVC(G) of cost $\mathbf{w}(\mathbf{x}^{v,\circ}) = \mathbf{w}(\mathbf{x}^v) + 1 = \mathrm{vc}^*(G - v) + 1$. Thus, if for some $v \in V(G)$ we have $\mathbf{w}(\mathbf{x}^v) = \mathrm{vc}^*(G - v) \leq \mathrm{vc}^*(G) - 1$, then $\mathbf{x}^{v,\circ}$ is an optimum solution for LPVC(G). Moreover, $\mathbf{x}^{v,\circ}$ is half-integral, but is not equal to the all-$\frac{1}{2}$-solution due to the value at vertex v. Hence, we may return $\mathbf{x}^{v,\circ}$.

We are left with the case

$$\mathrm{vc}^*(G - v) > \mathrm{vc}^*(G) - 1 \quad \text{for every } v \in V(G). \tag{3.4}$$

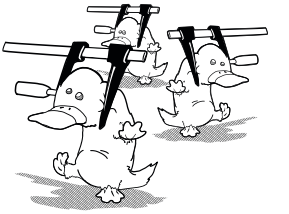
We claim that in this case the all-$\frac{1}{2}$-solution is the unique solution to LPVC(G); note that such a claim would conclude the proof of the lemma, as the computation time used so far is bounded by $\mathcal{O}(mn^{3/2})$ ($n+1$ applications of Proposition 2.24). Observe that, due to Proposition 2.24, both $2\,\mathrm{vc}^*(G-v)$ and $2\,\mathrm{vc}^*(G)$ are integers and, consequently, we obtain the following strengthening of (3.4):

$$\mathrm{vc}^*(G - v) \geq \mathrm{vc}^*(G) - \frac{1}{2} \quad \text{for every } v \in V(G). \tag{3.5}$$

By contradiction, let $\mathbf{x}$ be an optimum solution to LPVC(G) that is not the all-$\frac{1}{2}$-solution. As the all-$\frac{1}{2}$-solution is an optimum solution to LPVC(G),

Chapter 4
Iterative compression

In this chapter we introduce iterative compression, a simple yet very useful technique for designing fixed-parameter tractable algorithms. Using this technique we obtain FPT *algorithms for* Feedback Vertex Set in Tournaments, Feedback Vertex Set *and* Odd Cycle Transversal.



In 2004, Reed, Smith and Vetta [397] presented the first fixed-parameter tractable algorithm for Odd Cycle Transversal, running in time $3^k n^{\mathcal{O}(1)}$. This result is important not only because of the significance of the problem, but also because the proposed approach turned out to be a novel and generic technique, applicable in many other situations. Based on this new technique, called nowadays *iterative compression*, a number of FPT algorithms for several important problems have been obtained. Besides Odd Cycle Transversal, examples include Directed Feedback Vertex Set and Almost 2-SAT.

Typically, iterative compression algorithms are designed for parameterized minimization problems, where the goal is to find a small set of vertices or edges of the graph, whose removal makes the graph admit some global property. The upper bound on the size of this set is the parameter k. The main idea is to employ a so-called *compression routine*. A compression routine is an algorithm that, given a problem instance and a corresponding solution, either calculates a smaller solution or proves that the given solution is of the minimum size. Using a compression routine, one finds an optimal solution to the problem by iteratively building up the structure of the instance and compressing intermediate solutions.

The main point of the iterative compression technique is that if the compression routine runs in FPT time, then so does the whole algorithm. The strength of iterative compression is that it allows us to see the problem from a different viewpoint: The compression routine has not only the problem

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_4

instance as input, but also a solution, which carries valuable structural information about the instance. Therefore, constructing a compression routine may be simpler than designing a direct FPT algorithm for the original problem.

While embedding the compression routine into the iteration framework is usually straightforward, finding the compression routine itself is not. Therefore, the art of iterative compression typically lies in the design of the compression routine. In this chapter we design algorithms for Feedback Vertex Set in Tournaments, Feedback Vertex Set and Odd Cycle Transversal using the method of iterative compression. An important case of Almost 2-SAT is covered in the exercises. This technique will be also applied in Chapter 7 to solve Planar Vertex Deletion, and in Chapter 8 for Directed Feedback Vertex Set.

4.1 Illustration of the basic technique

The technique described in this section is based on the following strategy.

> Solution compression: First, apply some simple trick so that you can assume that a slightly too large solution is available. Then exploit the structure it imposes on the input graph to construct an optimal solution.

As a simple example of this approach, let us try to apply it to our favourite example problem Vertex Cover. The algorithm we are going to obtain now is much worse than the one obtained in the previous chapter, but it serves well for the illustration. Assume we are given a Vertex Cover instance (G, k). We use the well-known 2-approximation algorithm to obtain an approximate vertex cover Z. If $|Z| > 2k$, then we can clearly conclude that (G, k) is a no-instance, so assume otherwise. We are now going to exploit the structure that Z imposes on the graph G: Z is small, so we can afford some branching on Z, while at the same time $G - Z$ is edgeless.

The branching step is as follows: we branch in all possible ways an optimal solution X can intersect Z. Let $X_Z \subseteq Z$ be one such guess. We are searching now for a vertex cover X of size at most k, such that $X \cap Z = X_Z$. Let $W = Z \setminus X_Z$. If there is an edge in $G[W]$, then we can clearly conclude that our guess of X_Z was wrong. Otherwise, note that any vertex cover X satisfying $X \cap Z = X_Z$ needs to include $X_Z \cup N_G(W)$. Furthermore, since $G - Z$ is an independent set, $X_Z \cup N_G(W)$ is actually a vertex cover of G. Consequently, if for some choice of $X_Z \subseteq Z$ we have $|X_Z \cup N_G(W)| \leq k$, then we return a solution $X := X_Z \cup N_G(W)$, and otherwise we conclude that (G, k) is a no-instance. Thus, we obtain an algorithm solving Vertex Cover in time $2^{|Z|} n^{\mathcal{O}(1)} \leq 4^k n^{\mathcal{O}(1)}$.

Let us go back to the point when we have guessed the set X_Z and defined $W = Z \setminus X_Z$. In the previous paragraph we have in fact argued that the following Disjoint Vertex Cover problem is polynomial-time solvable: Does $G - X_Z$ contain a vertex cover of size at most $k - |X_Z|$ that is disjoint from W? Note here that W is a vertex cover of $G - X_Z$, giving us a lot of insight into the structure of $G - X_Z$.

In our example, the final dependency of the running time on k is much worse than the one obtained by branching algorithms in the previous chapter. One of the reasons is that we started with a large set Z, of size at most $2k$. Fortunately, there is an easy and very versatile way to obtain a set Z of size $k + 1$. This is exactly the main trick of *iterative compression*.

As the name suggests, in iterative compression we apply the compression step iteratively. To exemplify this idea on Vertex Cover, let us take an arbitrary ordering $(v_1, v_2, \ldots, v_n)$ of G. For $i \in \{1, \ldots, k\}$, we denote by G_i the subgraph of G induced by the first i vertices. For $i = k$, we can take the vertex set of G_i as a vertex cover X in G_i of size k. We proceed iteratively. Suppose that for some $i > k$, we have constructed a vertex cover X_i of G_i of size at most k. Then in graph G_{i+1}, set $Z_{i+1} := X_i \cup \{v_{i+1}\}$ is a vertex cover of size at most $k + 1$. If actually $|Z_{i+1}| \leq k$ then we are done: we can simply put $X_{i+1} = Z_{i+1}$ and proceed to the next iteration. Otherwise we have $|Z_{i+1}| = k + 1$, and we need to compress the too large solution. By applying the branching algorithm described above, i.e., solving $2^{|Z_{i+1}|} = 2^{k+1}$ instances of Disjoint Vertex Cover, in time $2^{k+1}n^{\mathcal{O}(1)}$, we can either find a vertex cover X_{i+1} of G_i of size at most k, or conclude that no such cover exists. If G_i does not admit a vertex of size at most k, then of course neither does G, and we may terminate the whole iteration and provide a negative answer to the problem. If X_{i+1} has been found, however, then we may proceed further to the graph G_{i+2} and so on. To conclude, observe that $G_n = G$, so at the last iteration we obtain a solution for the input Vertex Cover instance in time $2^k n^{\mathcal{O}(1)}$.

Combined with other ideas, this simple strategy becomes a powerful technique which can be used to solve different parameterized problems. Let us sketch how this method can be applied to a graph problem. The central idea here is to design an FPT algorithm which for a given $(k + 1)$-sized solution for a problem either compresses it to a solution of size at most k or proves that there is no solution of size at most k. This is known as the compression step of the algorithm. The method adopted usually is to begin with a subgraph that trivially admits a k-sized solution and then expand it iteratively. In any iteration, we try to find a compressed k-sized solution for the instance corresponding to the current subgraph. If we find such a solution, then by adding a vertex or an edge we obtain a solution to the next instance, but this solution can be too large by 1. To this solution we apply the compression step. We stop when we either obtain a solution of size at most k for the entire graph, or if some intermediate instance turns out to be incompressible. In order to stop in the case when some intermediate instance turns out to

be incompressible, the problem must have the property that the optimum solution size in any intermediate instance is at most the optimum solution size in the whole graph.

4.1.1 A few generic steps

We start with explaining the generic steps of iterative compression that are common to most of the problems. We focus here on vertex subset problems; the steps for an edge subset problem are analogous. Suppose that we want to solve $(*)$-Compression, where the wildcard $(*)$ can be replaced by the name of the problem we are trying to solve. In $(*)$-Compression, as input we are given an instance of the problem $(*)$ with a solution of size $k+1$ and a positive integer k. The objective is to either find a solution of size at most k or conclude that no solution of size at most k exists. For example, for $(*)$ =Feedback Vertex Set, we are given a graph G and a feedback vertex set X of size $k+1$. The task is to decide whether G has a feedback vertex set of size at most k.

The first observation that holds for all the problems in this section is the following:

If there exists an algorithm solving $(*)$-Compression in time $f(k) \cdot n^c$, then there exists an algorithm solving problem $(*)$ in time $\mathcal{O}(f(k) \cdot n^{c+1})$.

We already explained how to prove such an observation for Vertex Cover. We will repeat it once again for Feedback Vertex Set in Tournaments and skip the proofs of this observation for Feedback Vertex Set and Odd Cycle Transversal.

Now we need to compress a solution Z of size $k+1$. In all our examples we follow the same strategy as we did for Vertex Cover. That is, for every $i \in \{0, \ldots, k\}$ and every subset X_Z of Z of size i, we solve the following Disjoint-$(*)$ problem: Either find a solution X to $(*)$ in $G - X_Z$ such that $|X| \leq k - i$, where X and $W := Z \setminus X_Z$ are disjoint, or conclude that this is impossible. Note that in the disjoint variant we have that $|W| = k - i + 1$, so again the size of the solution W is one larger than the allowed budget; the difference now is that taking vertices from W is explicitly forbidden. We use the following observation, which follows from simple branching.

If there exists an algorithm solving Disjoint-$(*)$ in time $g(k) \cdot n^{\mathcal{O}(1)}$, then there exists an algorithm solving $(*)$-Compression in time

$$\sum_{i=0}^{k} \binom{k+1}{i} g(k-i) n^{\mathcal{O}(1)}.$$

In particular, if $g(k) = \alpha^k$, then $(*)$-Compression can be solved in time $(1+\alpha)^k n^{\mathcal{O}(1)}$.

Thus, the crucial part of the algorithms based on iterative compression lies in solving the disjoint version of the corresponding problem. We will provide the detailed proof of the above observation for Feedback Vertex Set in Tournaments. We will not repeat these arguments for Feedback Vertex Set and Odd Cycle Transversal, and explain only algorithms solving Disjoint-$(*)$ for these problems.

Finally, let us point out that when designing algorithms for the compression and disjoint versions of a problem, one cannot focus only on the decision version, where the task is just to determine whether a solution exists. This is because constructing an actual solution is needed to perform the next step of the iteration. This issue is almost never a real problem: either the algorithm actually finds the solution on the way, or one can use the standard method of self-reducibility to query a decision algorithm multiple times in order to reconstruct the solution. However, the reader should be aware of the caveat, especially when trying to estimate the precise polynomial factor of the running time of the algorithm.

4.2 Feedback Vertex Set in Tournaments

In this section we design an FPT algorithm for Feedback Vertex Set in Tournaments (FVST) using the methodology of iterative compression. In this problem, the input consists of a tournament T and a positive integer k, and the objective is to decide whether there exists a vertex set $X \subseteq V(T)$ of size at most k such that $T - X$ is a directed acyclic graph (equivalently, a transitive tournament). We call the solution X a *directed feedback vertex set*. Let us note that Feedback Vertex Set in Tournaments is a special case of Directed Feedback Vertex Set, where the input directed graph is restricted to being a tournament.

In what follows, using the example of Feedback Vertex Set in Tournaments we describe the steps that are common to most of the applications of iterative compression.

We first define the compression version of the problem, called Feedback Vertex Set in Tournaments Compression. In this problem, the input consists of a tournament T, a directed feedback vertex set Z of T of size $k+1$, and a positive integer k, and the objective is either to find a directed

feedback vertex set of T of size at most k, or to conclude that no such set exists.

Suppose we can solve Feedback Vertex Set in Tournaments Compression in time $f(k) \cdot n^{\mathcal{O}(1)}$. Given this, we show how to solve the original problem in $f(k) \cdot n^{\mathcal{O}(1)}$ time. We take an arbitrary ordering $(v_1, v_2, \ldots, v_n)$ of $V(T)$ and for every $i \in \{1, \ldots, n\}$ we define $V_i = \{v_1, \ldots, v_i\}$ and $T_i = T[V_i]$. Notice that

- V_k is a directed feedback vertex set of size k of T_k.
- If X is a directed feedback vertex set of T_i, then $X \cup \{v_{i+1}\}$ is a directed feedback vertex set of T_{i+1}.
- If T_i does not admit a directed feedback vertex set of size at most k, then neither does T.

These three facts together with an $f(k) \cdot n^c$-time algorithm for Feedback Vertex Set in Tournaments Compression imply an $f(k) \cdot n^{c+1}$-time algorithm for Feedback Vertex Set in Tournaments as follows. In tournament T_k set V_k is a directed feedback vertex set of size k. Suppose that for $i \geq k$ we have constructed a directed feedback vertex set X_i of T_i of size at most k. Then in T_{i+1}, set $Z_{i+1} := X_i \cup \{v_{i+1}\}$ is a directed feedback vertex set of size at most $k+1$. If actually $|Z_{i+1}| \leq k$, then we may proceed to the next iteration with $X_{i+1} = Z_{i+1}$. Otherwise we have $|Z_{i+1}| = k+1$. Then in time $f(k) \cdot n^c$ we can either construct a directed feedback vertex set X_{i+1} in T_{i+1} of size k, or deduce that (T, k) is a no-instance of Feedback Vertex Set in Tournaments. By iterating at most n times, we obtain the following lemma.

Lemma 4.1. *The existence of an algorithm solving* Feedback Vertex Set in Tournaments Compression *in time* $f(k) \cdot n^c$ *implies that* Feedback Vertex Set in Tournaments *can be solved in time* $\mathcal{O}(f(k) \cdot n^{c+1})$.

In all applications of iterative compression one proves a lemma similar to Lemma 4.1. Hence, the bulk of the work goes into solving the compression version of the problem. We now discuss how to solve the compression problem by reducing it to a bounded number of instances of the following disjoint version of the problem: Disjoint Feedback Vertex Set in Tournaments. In this problem, the input consists of a tournament T together with a directed feedback vertex set W and the objective is either to find a directed feedback vertex set $X \subseteq V(T) \setminus W$ of size at most k, or to conclude that no such set exists.

Let Z be a directed feedback vertex set of size $k+1$ in a tournament T. To decide whether T contains a directed feedback vertex set X of size k (i.e., to solve a Feedback Vertex Set in Tournaments Compression instance (G, Z, k)), we do the following. We guess the intersection of X with Z, that is, we guess the set $X_Z := X \cap Z$, delete X_Z from T and reduce parameter k by $|X_Z|$. For each guess of X_Z, we set $W := Z \setminus X_Z$ and solve Disjoint Feedback Vertex Set in Tournaments on the instance $(T - X_Z, W, k -$

$|X_Z|$). If for some guess X_Z we find a directed feedback vertex set X' of $T-X_Z$ of size at most $k-|X_Z|$ that is disjoint from W, then we output $X := X' \cup X_Z$. Otherwise, we conclude that the given instance of the compression problem is a no-instance. The number of all guesses is bounded by $\sum_{i=0}^{k} \binom{k+1}{i}$. So to obtain an FPT algorithm for Feedback Vertex Set in Tournaments Compression, it is sufficient to solve Disjoint Feedback Vertex Set in Tournaments in FPT time. This leads to the following lemma.

Lemma 4.2. *If there exists an algorithm solving* Disjoint Feedback Vertex Set in Tournaments *in time* $g(k) \cdot n^{\mathcal{O}(1)}$*, then there exists an algorithm solving* Feedback Vertex Set in Tournaments Compression *in time* $\sum_{i=0}^{k} \binom{k+1}{i} g(k-i) \cdot n^{\mathcal{O}(1)}$*. In particular, if* $g(k) = \alpha^k$ *for some fixed constant* α*, then the algorithm runs in time* $(\alpha+1)^k \cdot n^{\mathcal{O}(1)}$*.*

By Lemmas 4.1 and 4.2, we have that

> Solving Feedback Vertex Set in Tournaments boils down to solving the disjoint variant of the problem.

There is nothing very particular about Feedback Vertex Set in Tournaments in Lemma 4.2. In many graph modification problems, for which the corresponding disjoint version can be solved in time $g(k) \cdot n^{\mathcal{O}(1)}$, one can obtain an algorithm for the original problem by proving a lemma analogous to Lemma 4.2. We need only the following two properties: (i) a way to deduce that if an intermediate instance is a no-instance, then the input instance is a no-instance as well; and (ii) a way to enhance a computed solution X_i of an intermediate instance G_i with a new vertex or edge to obtain a slightly larger solution Z_{i+1} of the next intermediate instance G_{i+1}.

4.2.1 Solving Disjoint Feedback Vertex Set in Tournaments *in polynomial time*

Our next step is to show that Disjoint Feedback Vertex Set in Tournaments can be solved in polynomial time. As we already discussed, together with Lemmas 4.1 and 4.2, this shows that Feedback Vertex Set in Tournaments can be solved in time $2^k n^{\mathcal{O}(1)}$.

For our proof we need the following simple facts about tournaments, which are left as an exercise.

Lemma 4.3. *Let* T *be a tournament. Then*

1. T *has a directed cycle if and only if* T *has a directed triangle;*

2. *If T is acyclic then it has unique topological ordering. That is, there exists a unique ordering $\prec$ of the vertices of T such that for every directed edge (u, v), we have $u \prec v$ (that is, u appears before v in the ordering $\prec$).*

We now describe the algorithm for the disjoint version of the problem.

Let (T, W, k) be an instance of Disjoint Feedback Vertex Set in Tournaments and let $A = V(T) \setminus W$. Recall that we have that $|W| = k + 1$. Clearly, we can assume that both $T[W]$ and $T[A]$ induce transitive tournaments, since otherwise (T, W, k) is a no-instance. By Lemma 4.3, in order to solve Disjoint Feedback Vertex Set in Tournaments it is sufficient to find a set of at most k vertices from A intersecting all the directed triangles in T. This observation gives us the following simple reduction.

Reduction FVST.1. If T contains a directed triangle x, y, z with exactly one vertex from A, say z, then delete z and reduce the parameter by 1. The new instance is $(T - \{z\}, W, k - 1)$.

Reduction FVST.1 simply says that all directed triangles in T with exactly one vertex in A can be eliminated by picking the corresponding vertex in A. Safeness of Reduction FVST.1 follows from the fact that we are not allowed to select any vertex from W.

Given an instance (T, W, k), we first apply Reduction FVST.1 exhaustively. So from now onwards assume that Reduction FVST.1 is no longer applicable. Since tournaments $T[W]$ and $T[A]$ are acyclic, by Lemma 4.3 we know that they both have unique topological orderings. Let the topological orderings of $T[W]$ and $T[A]$ be denoted by $\sigma = (w_1, \ldots, w_q)$ and ρ, respectively. Suppose X is a desired solution; then $T[W \cup (A \setminus X)]$ is a transitive tournament with the unique ordering such that when we restrict this ordering to W, we obtain σ, and when we restrict it to $A \setminus X$, we get restriction of ρ to $A \setminus X$. Since the ordering of σ is preserved in the ordering of $T[W \cup (A \setminus X)]$, our modified goal now is as follows.

> Insert a maximum-sized subset of A into ordering $\sigma = (w_1, \ldots, w_q)$.

Every vertex $v \in A$ has a "natural position" in σ, say $p[v]$, defined as follows. Since Reduction FVST.1 is no longer applicable, for all $v \in A$, we have that $T[W \cup \{v\}]$ is acyclic and the position of v in σ is its position in the unique topological ordering of $T[W \cup \{v\}]$. Thus, there exists an integer $p[v]$ such that for $i < p[v]$, there is an arc from w_i to v and for all $i \geq p[v]$ there is an arc from v to w_i. Thus we get that

$$(v, w_i) \in E(T) \iff i \geq p[v]\,. \tag{4.1}$$

Observe that $p[v]$ is defined for all $v \in A$, it is unique, and $p[v] \in \{1, \ldots, q+1\}$.

We now construct an ordering π of A as follows: u is before v in π if and only if $p[u] < p[v]$ or $p[u] = p[v]$ and u is before v in the ordering ρ. That

is, in the ordering π we take iteratively the sets $\{v \in A \ : \ p[v] = i\}$ for $i = 1, 2, \ldots, q+1$ and, within each set, we order the vertices according to ρ, the topological order of $T[A]$.

The main observation now is as follows:

1. In the transitive tournament $T - X$, the topological ordering of $T[A \setminus X]$ needs to be a restriction of π, because $T[W]$ remains in $T - X$ and σ is the topological ordering of $T[W]$.
2. On the other hand, the topological ordering of $T[A \setminus X]$ needs to be a restriction of ρ, the topological ordering of $T[A]$.

Consequently, it suffices to search for the longest common subsequence of π and ρ.

We next prove a lemma which formalizes the intuition suggested above.

Lemma 4.4. *Let $B \subseteq A$. Then $T[W \cup B]$ is acyclic if and only if the vertices of B form a common subsequence of ρ and π.*

Proof. By $\rho|_B$ and $\pi|_B$ we denote restrictions of ρ and π to B, respectively.

For the forward direction, suppose that $T[W \cup B]$ is acyclic. We show that $\rho|_B = \pi|_B$ and hence vertices of B form a common subsequence of ρ and π. Targeting a contradiction, assume that there exist $x, y \in B$ such that x appears before y in $\rho|_B$ and y appears before x in $\pi|_B$. Then $(x, y) \in E(T)$, and $p[y] \leq p[x]$. Moreover, if it was that $p[x] = p[y]$, then the order of x and y in π would be determined by ρ. Thus we conclude that $p[y] < p[x]$. By (4.1), we have that $(y, w_{p[y]}) \in E(T)$ and $(w_{p[y]}, x) \in E(T)$. Because of the directed edges (x, y), $(y, w_{p[y]})$, and $(w_{p[y]}, x)$, we have that $\{x, y, w_{p[y]}\}$ induces a directed triangle in $T[W \cup B]$, a contradiction.

Now we show the reverse direction of the proof. Assume that the vertices of B form a common subsequence of ρ and π. In particular, this means that $\rho|_B = \pi|_B$. To show that $T[W \cup B]$ is acyclic, by Lemma 4.3 it is sufficient to show that $T[W \cup B]$ contains no directed triangles. Since $T[W]$ and $T[A]$ are acyclic and there are no directed triangles with exactly two vertices in W (Reduction FVST.1), it follows that there can only be directed triangles with exactly two vertices in B. Since $\rho|_B = \pi|_B$, for all $x, y \in B$ with $(x, y) \in E(T)$, we have that $p[x] \leq p[y]$. Then by (4.1), there is no $w_i \in W$ with $(y, w_i) \in E(T)$ and $(w_i, x) \in E(T)$. Hence there is no directed triangle in $T[W \cup B]$, and thus it is acyclic. This completes the proof of the lemma. □

We need the following known fact about the longest common subsequence problem, whose proof we leave as Exercise 4.2.

Lemma 4.5. *A longest common subsequence of two sequences with p and q elements can be found in time $\mathcal{O}(pq)$.*

We are ready to describe the algorithm now.

Lemma 4.6. Disjoint Feedback Vertex Set in Tournaments *is solvable in polynomial time.*

Proof. Let (T, W, k) be an instance of the problem. We use Reduction FVST.1 exhaustively. Let R be the set of vertices deleted by Reduction FVST.1, and let (T', W, k') be the reduced instance.

By Lemma 4.4, the optimal way to make T' acyclic by vertex deletions is exactly the same as that to make sequences ρ and π equal by vertex deletions. Thus, to find an optimal vertex deletion set, we can find a longest common subsequence of ρ and π, which can be done in polynomial time by Lemma 4.5. Let B be the vertices of a longest common subsequence of ρ and π and let $X := R \cup (V(T') \setminus B)$. If $|X| > k$; then (T, W, k) is a no-instance. Otherwise, X is the desired directed feedback vertex set of size at most k. □

Lemmas 4.1 and 4.2 combined with Lemma 4.6 give the following result.

Theorem 4.7. Feedback Vertex Set in Tournaments *can be solved in time* $2^k n^{\mathcal{O}(1)}$.

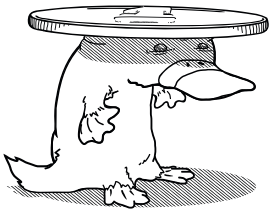
4.3 Feedback Vertex Set

In this section we give an algorithm for the Feedback Vertex Set problem on undirected graphs using the method of iterative compression. Recall that X is a feedback vertex set of an undirected graph G if $G - X$ is a forest. We give only the algorithms for the disjoint version of the problem. As was discussed in Section 4.1.1, the existence of an algorithm with running time $\alpha^k n^{\mathcal{O}(1)}$ for the disjoint variant problem would yield that Feedback Vertex Set is solvable in time $(1 + \alpha)^k n^{\mathcal{O}(1)}$.

We start by defining Disjoint Feedback Vertex Set. In this problem, as input we are given an undirected graph G, integer k and a feedback vertex set W in G of size $k + 1$. The objective is to find a feedback vertex set $X \subseteq V(G) \setminus W$ of size at most k, or correctly conclude that no such feedback vertex set exists. We first give an algorithm for Disjoint Feedback Vertex Set running in time $4^k n^{\mathcal{O}(1)}$, and then we provide a faster algorithm with running time $\varphi^{2k} n^{\mathcal{O}(1)}$, where $\varphi = \frac{1+\sqrt{5}}{2} < 1.6181$ denotes the golden ratio.

Chapter 5
Randomized methods in parameterized algorithms

In this chapter, we study how the powerful paradigm of randomization can help in designing FPT *algorithms. We introduce the general techniques of color coding, divide and color, and chromatic coding, and show the use of these techniques on the problems* Longest Path *and* d-Clustering. *We discuss also how these randomized algorithms can be derandomized using standard techniques.*



We give a few commonly used approaches to design randomized FPT algorithms.

In Section 5.1, we start with a simple example of an algorithm for Feedback Vertex Set. Here, we essentially observe that, after applying a few easy reduction rules, a large fraction of the edges of the graph needs to be adjacent to the solution vertices — and, consequently, taking into the solution a randomly chosen endpoint of a randomly chosen edge leads to a good success probability.

Then, in Section 5.2, we move to the very successful technique of *color coding*, where one randomly colors a universe (e.g., the vertex set of the input graph) with a carefully chosen number of colors and argue that, with sufficient probability, a solution we are looking for is somehow "properly colored". The problem-dependant notion of "properly colored" helps us resolve the problem if we only look for properly colored solutions. For resolving the colored version of the problem, we use basic tools learned in the previous chapters, such as bounded search trees. In this section we start with the classic example of a color coding algorithm for Longest Path. Then, we show how a random partition (i.e., a random coloring with two colors) can be used to highlight a solution, using an example of the Subgraph Isomorphism problem in bounded degree graphs. Moreover, we present the so-called *divide and color* technique for Longest Path, where a recursive random partitioning scheme allows us to obtain a better running time than the simple color coding approach. We conclude this section with a more advanced example of a *chromatic coding* approach for d-Clustering.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_5

In the area of polynomial-time algorithms, quite often we develop an algorithm that does something useful (e.g., solves the problem) with probability at least p, where $1/p$ is bounded polynomially in the input size. If this is the case, then we can repeat the algorithm a polynomial number of times, obtaining a constant error probability. In the FPT world, the same principle applies; however, now the threshold of "useful probability" becomes $1/(f(k)n^{\mathcal{O}(1)})$. That is, if we develop an algorithm that runs in FPT time and solves the problem with probability at least $1/(f(k)n^{\mathcal{O}(1)})$, then repeating the algorithm $f(k)n^{\mathcal{O}(1)}$ times gives us constant error probability, while still maintaining an FPT running time bound. This is the goal of most of the algorithms in this chapter.

5.1 A simple randomized algorithm for Feedback Vertex Set

In this section, we design a simple randomized algorithm for Feedback Vertex Set. As discussed in Section 3.3, when tackling with Feedback Vertex Set, it is more convenient to work with multigraphs, not only simple graphs. Recall that both a double edge and a loop are cycles, and we use the convention that a loop at a vertex v contributes *2* to the degree of v.

The following crucial lemma observes that, once we apply to the input instance basic reduction rules that deal with low-degree vertices, many edges of the graph are incident to the solution vertices.

Lemma 5.1. *Let G be a multigraph on n vertices, with minimum degree at least 3. Then, for every feedback vertex set X of G, more than half of the edges of G have at least one endpoint in X.*

Proof. Let $H = G - X$. Since every edge in $E(G) \setminus E(H)$ is incident to a vertex in X, the claim of the lemma is equivalent to $|E(G) \setminus E(H)| > |E(H)|$. However, since H is a forest, $|V(H)| > |E(H)|$ and it suffices to show that $|E(G) \setminus E(H)| > |V(H)|$.

Let J denote the set of edges with one endpoint in X and the other in $V(H)$. Let $V_{\leq 1}$, V_2 and $V_{\geq 3}$ denote the set of vertices in $V(H)$ such that they have degree at most 1, exactly 2, and at least 3 in H, respectively. (Note that we use here the degree of a vertex in the forest H, not in the input graph G.) Since G has minimum degree at least 3, every vertex in $V_{\leq 1}$ contributes at least two distinct edges to J. Similarly, each vertex in V_2 contributes at least one edge to J. As H is a forest, we have also that $|V_{\geq 3}| < |V_{\leq 1}|$. Putting all these bounds together, we obtain:

$$\begin{aligned}|E(G) \setminus E(H)| &\geq |J| \\ &\geq 2|V_{\leq 1}| + |V_2| > |V_{\leq 1}| + |V_2| + |V_{\geq 3}| \\ &= |V(H)|.\end{aligned}$$

This concludes the proof of the lemma. □

Recall that in Section 3.3, we have developed the simple reduction rules FVS.1–FVS.5 that reduce all vertices of degree at most 2 in the input graph. Lemma 5.1 says that, once such a reduction has been performed, a majority of the edges have at least one endpoint in a solution. Hence, if we pick an edge uniformly at random, and then independently pick its random endpoint, with probability at least $1/4$ we would pick a vertex from the solution. By iterating this process, we obtain an algorithm that solves the input Feedback Vertex Set instance in polynomial time, with probability at least 4^{-k}.

Theorem 5.2. *There exists a polynomial-time randomized algorithm that, given a* Feedback Vertex Set *instance* (G, k)*, either reports a failure or finds a feedback vertex set in* G *of size at most* k*. Moreover, if the algorithm is given a yes-instance, it returns a solution with probability at least* 4^{-k}*.*

Proof. We describe the algorithm as a recursive procedure that, given a graph G and an integer k, aims at a feedback vertex set of G of size at most k.

We first apply exhaustively Reductions FVS.1–FVS.5 to the Feedback Vertex Set instance (G, k). If the reductions conclude that we are dealing with a no-instance, then we report a failure. Otherwise, let (G', k') be the reduced instance. Note that $0 \leq k' \leq k$ and G' has minimum degree at least 3. Let X_0 be the set of vertices deleted due to Reduction FVS.1. Note that the vertices of X_0 have been qualified as mandatory vertices for a feedback vertex set in G, that is, there exists a feedback vertex set in G of minimum possible size that contains X_0, and for any feedback vertex set X' of G', $X' \cup X_0$ is a feedback vertex set of G. Moreover, $|X_0| = k - k'$.

If G' is an empty graph, then we return X_0; note that in this case $|X_0| \leq k$ as $k' \geq 0$, and X_0 is a feedback vertex set of G. Otherwise, we pick an edge e of G' uniformly at random (i.e., each edge is chosen with probability $1/|E(G')|$), and choose one endpoint of e independently and uniformly at random. Let v be the chosen endpoint. We recurse on $(G' - v, k' - 1)$. If the recursive step returns a failure, then we return a failure as well. If the recursive step returns a feedback vertex set X', then we return $X := X' \cup \{v\} \cup X_0$.

Note that in the second case the set X' is a feedback vertex set of $G' - v$ of size at most $k' - 1$. First, observe that the size bound on X' implies $|X| \leq k$. Second, we infer that $X' \cup \{v\}$ is a feedback vertex set of G' and, consequently, X is a feedback vertex set of G. Hence, the algorithm always reports a failure or returns a feedback vertex set of G of size at most k. It remains to argue about the probability bound; we prove it by induction on k.

Assume that there exists a feedback vertex set X of G of size at most k. By the analysis of the reduction rules of Section 3.3, Reduction FVS.5 is

not triggered, the instance (G', k') is computed and there exists a feedback vertex set X' of G' of size at most k'. If G' is empty, then the algorithm returns X_0 deterministically. Otherwise, since G' has minimum degree at least 3, Lemma 5.1 implies that, with probability larger than $1/2$, the edge e has at least one endpoint in X'. Consequently, with probability larger than $1/4$, the chosen vertex v belongs to X', and $X' \setminus \{v\}$ is a feedback vertex set of size at most $k' - 1$ in the graph $G' - v$. By the inductive hypothesis, the recursive call finds a feedback vertex set of $G' - v$ of size at most $k' - 1$ (not necessarily the set $X' \setminus \{v\}$) with probability at least $4^{-(k'-1)}$. Hence, a feedback vertex set of G of size at most k is found with probability at least $\frac{1}{4} \cdot 4^{-(k'-1)} = 4^{-k'} \geq 4^{-k}$. This concludes the inductive step, and finishes the proof of the theorem. □

As discussed at the beginning of this section, we can repeat the algorithm of Theorem 5.2 independently 4^k times to obtain a constant error probability.

Corollary 5.3. *There exists a randomized algorithm that, given a* Feedback Vertex Set *instance* (G, k)*, in time* $4^k n^{\mathcal{O}(1)}$ *either reports a failure or finds a feedback vertex set in* G *of size at most* k*. Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

5.2 Color coding

The technique of color coding was introduced by Alon, Yuster and Zwick to handle the problem of detecting a small subgraph in a large input graph. More formally, given a k-vertex "pattern" graph H and an n-vertex input graph G, the goal is to find a subgraph of G isomorphic to H. A brute-force approach solves this problem in time roughly $\mathcal{O}(n^k)$; the color coding technique approach allows us to obtain an FPT running time bound of $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ in the case when H is a forest or, more generally, when H is of constant treewidth (for more on treewidth, we refer you to Chapter 7). We remark here that, as we discuss in Chapter 13, such an improvement is most likely not possible in general, as the case of H being a k-vertex clique is conjectured to be hard.

The idea behind the color coding technique is to randomly color the entire graph with a set of colors with the number of colors chosen in a way that, if the smaller graph does exist in this graph as a subgraph, then with high probability it will be colored in a way that we can find it *efficiently*. In what follows we mostly focus on a simplest, but very instructive case of H being a k-vertex path.

We remark that most algorithms based on the color coding technique can be derandomized using splitters and similar pseudorandom objects. We discuss these methods in Section 5.6.

5.2.1 A color coding algorithm for Longest Path

In the Longest Path problem, we are given a graph G and a positive integer k as input, and the objective is to test whether there exists a simple path on k vertices in the graph G. Note that this corresponds to the aforementioned "pattern" graph search problem with H being a path on k vertices (henceforth called a *k-path* for brevity).

Observe that finding a *walk* on k vertices in a directed graph is a simple task; the hardness of the Longest Path problem lies in the requirement that we look for a *simple path*. A direct approach involves keeping track of the vertices already visited, requiring an $\binom{n}{k}$ factor in the running time bound. The color coding technique is exactly the trick to avoid such a dependency.

> Color the vertices uniformly at random from $\{1, \ldots, k\}$, and find a path on k vertices, if it exists, whose all colors are pairwise distinct.

The essence of the color coding technique is the observation that, if we color some universe with k colors uniformly at random, then a given k-element subset is colored with distinct colors with sufficient probability.

Lemma 5.4. *Let U be a set of size n, and let $X \subseteq U$ be a subset of size k. Let $\chi : U \to [k]$ be a coloring of the elements of U, chosen uniformly at random (i.e., each element of U is colored with one of k colors uniformly and independently at random). Then the probability that the elements of X are colored with pairwise distinct colors is at least e^{-k}.*

Proof. There are k^n possible colorings χ, and $k!k^{n-k}$ of them are injective on X. The lemma follows from the well-known inequality $k! > (k/e)^k$. □

Hence, the color coding step reduces the case of finding a k-path in a graph to finding a *colorful* k-path in a vertex-colored graph. (In what follows, a path is called *colorful* if all vertices of the path are colored with pairwise distinct colors.) Observe that this step replaces the $\binom{n}{k}$ factor, needed to keep track of the used vertices in a brute-force approach, with a much better 2^k factor, needed to keep track of used *colors*. As the next lemma shows, in the case of finding a colorful path, the algorithm is relatively simple.

Lemma 5.5. *Let G be a directed or an undirected graph, and let $\chi : V(G) \to [k]$ be a coloring of its vertices with k colors. There exists a deterministic algorithm that checks in time $2^k n^{\mathcal{O}(1)}$ whether G contains a colorful path on k vertices and, if this is the case, returns one such path.*

Proof. Let $V_1, \ldots, V_k$ be a partitioning of $V(G)$ such that all vertices in V_i are colored i. We apply dynamic programming: for a *nonempty* subset S of $\{1, \ldots, k\}$ and a vertex $u \in \bigcup_{i \in S} V_i$, we define the Boolean value $\mathrm{PATH}(S, u)$

to be equal to true if there is a colorful path with vertices colored with all colors from S and with an endpoint in u. For $|S| = 1$, note that $\mathrm{PATH}(S, u)$ is true for any $u \in V(G)$ if and only if $S = \{\chi(u)\}$. For $|S| > 1$, the following recurrence holds:

$$\mathrm{PATH}(S, u) = \begin{cases} \bigvee\{\mathrm{PATH}(S \setminus \{\chi(u)\}, v) \ : \ uv \in E(G)\} & \text{if } \chi(u) \in S \\ \text{False} & \text{otherwise.} \end{cases}$$

Indeed, if there is a colorful path ending at u using all colors from S, then there has to be colorful path ending at a neighbor v of u and using all colors from $S \setminus \{\chi(u)\}$.

Clearly, all values of PATH can be computed in time $2^k n^{\mathcal{O}(1)}$ by applying the above recurrence, and, moreover, there exists a colorful k-path in G if and only if $\mathrm{PATH}([k], v)$ is true for some vertex $v \in V(G)$. Furthermore, a colorful path can be retrieved using the standard technique of backlinks in dynamic programming. □

We now combine Lemmas 5.4 and 5.5 to obtain the main result of this section.

Theorem 5.6. *There exists a randomized algorithm that, given a* Longest Path *instance* (G, k)*, in time* $(2e)^k n^{\mathcal{O}(1)}$ *either reports a failure or finds a path on* k *vertices in* G*. Moreover, if the algorithm is given a yes-instance, it returns a solution with a constant probability.*

Proof. We show an algorithm that runs in time $2^k n^{\mathcal{O}(1)}$, and, given a yes-instance, returns a solution with probability at least e^{-k}. Clearly, by repeating the algorithm independently e^k times, we obtain the running time bound and success probability guarantee promised by the theorem statement.

Given an input instance (G, k), we uniformly at random color the vertices of $V(G)$ with colors $[k]$. That is, every vertex is colored independently with one color from the set $[k]$ with uniform probability. Denote the obtained coloring by $\chi : V(G) \to [k]$. We run the algorithm of Lemma 5.5 on the graph G with coloring χ. If it returns a colorful path, then we return this path as a simple path in G. Otherwise, we report failure.

Clearly, any path returned by the algorithm is a k-path in G. It remains to bound the probability of finding a path in the case (G, k) is a yes-instance.

To this end, suppose G has a path P on k vertices. By Lemma 5.4, P becomes a colorful path in the coloring χ with probability at least e^{-k}. If this is the case, the algorithm of Lemma 5.5 finds a colorful path (not necessarily P itself), and the algorithm returns a k-path in G. This concludes the proof of the theorem. □

We have proved that the probability that χ is successful is at least 2^{-dk}. Hence, to obtain a Monte Carlo algorithm with false negatives we repeat the above procedure 2^{dk} times, and obtain the following result:

Theorem 5.7. *There exist Monte Carlo algorithms with false negatives that solve the input* Subgraph Isomorphism *instance* (G, H) *in time* $2^{dk}k!n^{\mathcal{O}(1)}$ *and in time* $2^{dk}k^{\mathcal{O}(d \log d)}n^{\mathcal{O}(1)}$. *Here,* $|V(G)| = n$, $|V(H)| = k$, *and the maximum degree of* G *is bounded by* d.

*5.4 A divide and color algorithm for Longest Path

We now improve the running time bound of the algorithm for Longest Path, by using a different choice of coloring. The idea is inspired by the "divide and conquer" approach: one of the basic algorithmic techniques to design polynomial-time algorithms. In other words, we will see a technique that is an amalgamation of divide and conquer and color coding. Recall that for color coding we used k colors to make the subgraph we are seeking colorful. A natural question is: Do we always need k colors, or in some cases can we reduce the number of colors and hence the randomness used in the algorithm? Some problems can naturally be decomposed into disjoint pieces and solutions to individual pieces can be combined together to get the complete solution. The idea of *divide and color* is to use randomization to separate the pieces. For example, in the case of graph problems, we will randomly partition all vertices (or edges) of a graph into the left and the right side, and show that the structure we were looking for has been conveniently split between the sides. We solve the problem recursively on a graph induced on left and right sides separately and then combine them to get the structure we are searching for.

Thus, the workflow of the algorithm is a bit different from the one in the previous color coding example. We perform the partition step at every node of the recursion tree, solving recursively the same problem in subinstances. The leaves of the recursion tree correspond to trivial instances with $k = \mathcal{O}(1)$, where no involved work is needed. On the other hand, the work needed to glue the information obtained from subinstances to obtain a solution for the current instance can be seen as a variant of dynamic programming.

The basic idea of the divide and color technique for the Longest Path problem is to randomly assign each vertex of the graph G to either a set L (left) or another set R (right) with equal probability and thus obtain a partitioning of the vertices of the input graph. By doing this, we hope that there is a k-path P such that the first $\lceil \frac{k}{2} \rceil$ vertices of P are in L and the last $\lfloor \frac{k}{2} \rfloor$ vertices of P are in R. Observe that for a fixed k-path

Simple-Randomized-Paths(X, ℓ)
Input: A subset $X \subseteq V(G)$ and an integer ℓ, $1 \leq \ell \leq k$
Output: $\widehat{D}_{X,\ell}$

1. If $\ell = 1$, then return $\widehat{D}_{X,\ell}[v, v] = \top$ for all $v \in X$, $\bot$ otherwise.
2. Uniformly at random partition X into L and R.
3. $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$:=Simple-Randomized-Paths$(L, \lceil \frac{\ell}{2} \rceil)$
4. $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$:=Simple-Randomized-Paths$(R, \lfloor \frac{\ell}{2} \rfloor)$
5. Return $\widehat{D}_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$

Fig. 5.1: A simple algorithm to find a path on k vertices

this happens with probability exactly 2^{-k}. After the partitioning we recurse on the two induced graphs $G[L]$ and $G[R]$.

However, the naive implementation of this scheme gives a success probability worse than $2^{-\mathcal{O}(k)}$, and hence cannot be used to obtain an algorithm with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ and constant success probability. We need two additional ideas to make this scheme work. First, we need to define the recursion step in a way that it considers all possible pairs of endpoints for the path. We present first an algorithm implementing this idea. Then we improve the algorithm further by observing that we can do better than selecting a single random partition in each recursion step and then repeating the whole algorithm several times to increase the probability of success. Instead, in each recursive step, we create several random partitions and make several recursive calls.

In order to combine the subpaths obtained from $G[L]$ and $G[R]$, we need more information than just one k-vertex path from each side. For a given subset $X \subseteq V(G)$, a number $\ell \in \{1, \ldots, k\}$ and a pair of vertices u and v of X let $D_{X,\ell}[u, v]$ denote a Boolean value equal to true if and only if there exists an ℓ-vertex path from u to v in $G[X]$. Note that for a fixed subset X, we can consider $D_{X,\ell}$ as an $|X| \times |X|$ matrix. In what follows, we describe a procedure Simple-Randomized-Paths(X, ℓ) that computes a matrix $\widehat{D}_{X,\ell}$. The relation between $\widehat{D}_{X,\ell}$ and $D_{X,\ell}$ is the following. If $\widehat{D}_{X,\ell}[u, v]$ is true for some $u, v \in X$, then so is $D_{X,\ell}[u, v]$. The crux of the method will be in ensuring that if $D_{X,\ell}[u, v]$ is true for some fixed u and v, then $\widehat{D}_{X,\ell}[u, v]$ is also true with sufficiently high probability. Thus, we will get a one-sided error Monte Carlo algorithm.

Given a partition (L, R) of X, an $|L| \times |L|$ matrix A, and an $|R| \times |R|$ matrix B, we define $A \bowtie B$ as an $|X| \times |X|$ Boolean matrix D. For every pair of vertices $u, v \in X$ we have $D[u, v]$ equal to true if and only if $u \in L$, $v \in R$

and there is an edge $xy \in E(G[X])$ such that $x \in L$, $y \in R$ and both $A[u,x]$ and $B[y,v]$ are set to true. Then, $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ is true if and only if there exists an ℓ-path P in $G[X]$ from u to v, whose first $\lceil \frac{\ell}{2} \rceil$ vertices belong to L and the remaining $\lfloor \frac{\ell}{2} \rfloor$ vertices belong to R. In particular, $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ implies $D_{X,\ell}[u,v]$. The main observation is that if $D_{X,\ell}[u,v]$ is true, then one can hope that the random choice of L and R partitions the vertices of the path P correctly, i.e., so that $(D_{L,\lceil \frac{\ell}{2} \rceil} \bowtie D_{R,\lfloor \frac{\ell}{2} \rfloor})[u,v]$ is also true with some sufficiently large probability. Thus, we compute the matrices $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$ and $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$ recursively by invoking Simple-Randomized-Paths$(L, \lceil \frac{\ell}{2} \rceil)$ and Simple-Randomized-Paths$(R, \lfloor \frac{\ell}{2} \rfloor)$, respectively, and then return $\widehat{D}_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$. For the base case in this recurrence, we take $\ell = 1$ and compute $\widehat{D}_{X,1} = D_{X,1}$ directly from the definition.

Let us analyze the probability that, if $D_{X,\ell}[u,v]$ is true for some $u, v \in V(G)$, then $\widehat{D}_{X,\ell}[u,v]$ is also true. For fixed ℓ, by p_ℓ denote the infinimum of this probability for all graphs G, all sets $X \subseteq V(G)$, and all vertices $u, v \in X$. If $\ell = 1$, then $\widehat{D}_{X,\ell} = D_{X,\ell}$ and, consequently, $p_1 = 1$. Otherwise, let P be any ℓ-path in $G[X]$ with endpoints u and v; we now analyze how the path P can be "detected" by the algorithm. Let x be the $\lceil \frac{\ell}{2} \rceil$-th vertex on P (counting from u), and let y be the successor of x on P. Moreover, let P_L be the subpath of P between u and x, inclusive, and let P_R be the subpath of P between y and v, inclusive. Note that, P_L has $\lceil \frac{\ell}{2} \rceil$ vertices, P_R has $\lfloor \frac{\ell}{2} \rfloor$ vertices and, as $\ell > 1$, $\lfloor \frac{\ell}{2} \rfloor \leq \lceil \frac{\ell}{2} \rceil < \ell$.

Observe that, with probability $2^{-\ell}$, all vertices of P_L are assigned to L and all vertices of P_R are assigned to R. Moreover, if this is the case, then both $D_{L,\lceil \frac{\ell}{2} \rceil}[u,x]$ and $D_{R,\lfloor \frac{\ell}{2} \rfloor}[y,v]$ are true, with the paths P_L and P_R being the respective witnesses. Consequently, $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}[u,x]$ and $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}[y,v]$ are both true with probability at least $p_{\lceil \frac{\ell}{2} \rceil} p_{\lfloor \frac{\ell}{2} \rfloor}$. If this is the case, $\widehat{D}_{X,\ell}[u,v]$ is set to true by the definition of the $\bowtie$ product. Hence, we obtain the following recursive bound:

$$p_\ell \geq 2^{-\ell} p_{\lceil \frac{\ell}{2} \rceil} p_{\lfloor \frac{\ell}{2} \rfloor}. \tag{5.1}$$

By solving the recurrence (5.1), we obtain $p_\ell = 2^{-\mathcal{O}(\ell \log \ell)}$. To see this, observe that, in the search for the path P for the value $D_{X,\ell}[u,v]$, there are $\ell - 1$ partitions that the algorithm Simple-Randomized-Paths needs to find correctly: one partition of all ℓ vertices, two partitions of roughly $\ell/2$ vertices, four partitions of roughly $\ell/4$ vertices, etc. That is, for each $i = 0, 1, \ldots, \lfloor \log \ell \rfloor - 1$, the algorithm needs to make, roughly 2^i times, a correct partition of roughly $k/2^i$ vertices. Consequently, the success probability of the algorithm is given by

$$p_\ell \sim \prod_{i=0}^{\log \ell - 1} \left(\frac{1}{2^{\ell/2^i}} \right)^{2^i} = 2^{-\mathcal{O}(\ell \log \ell)}.$$

Faster-Randomized-Paths(X, ℓ)
Input: A subset $X \subseteq V(G)$ and an integer ℓ, $1 \leq \ell \leq k$
Output: $D_{X,\ell}$

1. If $\ell = 1$, then return $D_{X,\ell}[v, v] = \top$ for all $v \in X$, $\bot$ otherwise.
2. Set $\widehat{D}_{X,\ell}[u, v]$ to false for any $u, v \in X$.
3. Repeat the following $2^\ell \log(4k)$ times
   (a) Uniformly at random partition X into L and R.
   (b) $\widehat{D}_{L,\lceil \frac{\ell}{2} \rceil}$:=Faster-Randomized-Paths$(L, \lceil \frac{\ell}{2} \rceil)$
   (c) $\widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$:=Faster-Randomized-Paths$(R, \lfloor \frac{\ell}{2} \rfloor)$
   (d) Compute $\widehat{D}'_{X,\ell} := \widehat{D}_{L,\lceil \frac{\ell}{2} \rceil} \bowtie \widehat{D}_{R,\lfloor \frac{\ell}{2} \rfloor}$ and update
   $\widehat{D}_{X,\ell}[u, v] := \widehat{D}_{X,\ell}[u, v] \vee \widehat{D}'_{X,\ell}[u, v]$ for every $u, v \in V(G)$.
4. Return $\widehat{D}_{X,\ell}$.

Fig. 5.2: A faster algorithm to find a path on k-vertices

Thus, to achieve a constant success probability, we need to run the algorithm Simple-Randomized-Paths$(V(G), k)$ $2^{\mathcal{O}(k \log k)}$ number of times, obtaining a significantly worse running time bound than the one of Theorem 5.6.

Let us now try to improve the running time of our algorithm. The main idea is that instead of repeating the entire process $2^{\mathcal{O}(k \log k)}$ times, we split the repetitions between the recursive calls. More precisely, we choose some function $f(\ell, k)$ and, in each recursive call, we try not only one random partition of $V(G)$ into L and R, but $f(\ell, k)$ partitions, each chosen independently at random. Recall that ℓ denotes the argument of the recursive call (the length of paths we are currently looking for), whereas k denotes the argument of the root call to Faster-Randomized-Paths (the length of a path we are looking for in the entire algorithm). In this manner, we increase the running time of the algorithm, but at the same time we increase the success probability. Let us now analyze what function $f(\ell, k)$ guarantees constant success probability.

Recall that a fixed partition of the vertices into L and R suits our needs for recursion (i.e., correctly partitions the vertex set of one fixed ℓ-path) with probability $2^{-\ell}$. Hence, if we try $f(\ell, k)$ partitions chosen independently at random, the probability that none of the $f(\ell, k)$ partitions suits our needs is at most

$$\left(1 - \frac{1}{2^\ell}\right)^{f(\ell,k)}. \tag{5.2}$$

Fix a k-vertex path P witnessing that $D_{V(G),k}[u, v]$ is true for some fixed $u, v \in V(G)$. The main observation is that, although the entire recursion tree is huge — as we shall later see, it will be of size roughly 4^k — we do not need to be successful in all nodes of the recursion tree. First, we need to guess

correctly at least one partition in the root node of the recursion tree. Then, we may limit ourselves to the two recursive calls made for that one particular correct partition at the root node, and insist that in these two recursive calls we guess at least one correct partition of the corresponding subpaths of length $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$. If we proceed with such a reasoning up to the leaves of the recursion, we infer that we need to guess correctly at least one partition at exactly $k-1$ nodes. Consequently, we obtain constant success probability if the value of the expression (5.2) is at most $\frac{1}{2(k-1)}$ for every $1 \leq \ell \leq k$. To achieve this bound, observe that we can choose $f(\ell, k) = 2^\ell \log(4k)$. Indeed, since $1 + x \leq e^x$ for every real x,

$$\left(1 - \frac{1}{2^\ell}\right)^{2^\ell \log(4k)} \leq \left(e^{-\frac{1}{2^\ell}}\right)^{2^\ell \log(4k)} = \frac{1}{e^{\log(4k)}} \leq \frac{1}{2k}.$$

Since the probability space in the algorithm is quite complex, some readers may find the argumentation in the previous paragraph too informal. For sake of completeness, we provide a formal argumentation in the next lemma.

Lemma 5.8. *If (G, k) is a yes-instance and $f(\ell, k) = 2^\ell \log(4k)$ (i.e., the value of the expression (5.2) is at most $\frac{1}{2(k-1)}$ for every $1 \leq \ell \leq k$), then with probability at least $\frac{1}{2}$ there exists a pair of vertices $u, v \in V(G)$ with $\widehat{D}_{V(G),k}[u, v]$ equal to true.*

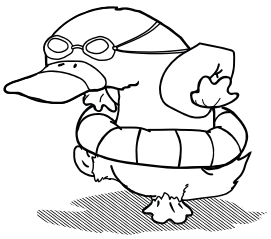
Proof. Consider a set $X \subseteq V(G)$, vertices $u, v \in V(G)$, an integer $1 \leq \ell \leq k$, and an ℓ-vertex path P in $G[X]$ with endpoints u and v. Clearly, P witnesses that $D_{X,\ell}[u, v] = 1$. Let p_ℓ^k denote the infinimum, for fixed k and ℓ, over all choices of G, X, u, and v such that $D_{X,\ell}[u, v] = 1$, of the probability that a call to Faster-Randomized-Paths(X, ℓ) on instance (G, k) returns $\widehat{D}_{X,\ell}[u, v] = 1$. Let $s = \frac{1}{2(k-1)}$. We prove by induction on ℓ that for every $1 \leq \ell \leq k$ we have $p_\ell^k \geq 1 - (\ell - 1)s$; note that the lemma follows from such a claim for $\ell = k$. In the base case, observe that $p_1^k = 1$ as $\widehat{D}_{X,1} = D_{X,1}$ for every $X \subseteq V(G)$.

For the induction step, consider a path P in $G[X]$, with $\ell > 1$ vertices and endpoints u and v. Observe that at the call Faster-Randomized-Paths(X, ℓ) to set $\widehat{D}_{X,\ell}[u, v] = 1$ it suffices to (a) at least once guess a correct partition for the path P; and (b) for the first such correct partition (L, R) compute $\widehat{D}_{L,\lceil \ell/2 \rceil}[u, x] = 1$ and $\widehat{D}_{R,\lfloor \ell/2 \rfloor}[y, v] = 1$, where x is the $\lceil \frac{\ell}{2} \rceil$-th vertex on P (counting from u), and y is the successor of x on P. Consequently,

$$\begin{aligned} p_\ell^k &\geq (1-s) p_{\lceil \ell/2 \rceil}^k p_{\lfloor \ell/2 \rfloor}^k \\ &\geq (1-s) \cdot (1 - (\lceil \ell/2 \rceil - 1)s) \cdot (1 - (\lfloor \ell/2 \rfloor - 1)s) \\ &\geq 1 - s - (\lceil \ell/2 \rceil - 1)s - (\lfloor \ell/2 \rfloor - 1)s \\ &= 1 - (\ell - 1)s. \end{aligned}$$

Chapter 6
Miscellaneous

In this chapter, we gather a few algorithmic tools that we feel are important, but do not fit into any of the previous chapters. First, we discuss exponential-time dynamic-programming algorithms that consider all subsets of a certain set when defining the subproblems. Second, we introduce the Integer Linear Programming Feasibility *problem, formulate the classical results on how to solve it in the case of a bounded number of variables, and show an example of its application in fixed-parameter algorithms. Finally, we discuss the algorithmic implications of the Robertson-Seymour theorem on graph minors.*



The chapter consists of three independent sections, each tackling a different tool in parameterized algorithms. Since a full exposition of, say, the algorithmic applications of the graph minors project would constitute a large volume in itself, we have decided to give only a glimpse of the topic in each of the three sections.

In Section 6.1, we present a common theme in many exponential-time algorithms: a dynamic-programming algorithm, where the subproblems are defined by considering all subsets of a certain set of elements (and hence the number of subproblems and the running time are exponential in the number of elements). As a first example, we give a simple $2^{|U|}(|U|+|\mathcal{F}|)^{\mathcal{O}(1)}$-time dynamic-programming algorithm for Set Cover with a family of sets $\mathcal{F}$ over a universe U. Then, we obtain a $3^{|K|}n^{\mathcal{O}(1)}$ algorithm for Steiner Tree, where K is the set of terminal vertices. Using additional ideas, we will improve the running time of this algorithm in Section 10.1.2.

Section 6.2 introduces yet another tool to design fixed-parameter algorithms, namely integer linear programs. In turns out that many NP-hard problems can be expressed in the language of Integer Linear Programming. For example, in Section 2.5 we have seen how a Vertex Cover instance can be encoded as an Integer Linear Programming instance.

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_6

In 1983, Lenstra showed that Integer Linear Programming is fixed-parameter tractable when parameterized by the dimension of the space, i.e., the number of variables. (Naturally, Lenstra did not use the terminology of fixed-parameter tractability, as it was introduced much later.) Thus, Lenstra's result gives us a very general tool for proving fixed-parameter tractability of various problems. Note that a similar phenomenon happens in the world of polynomial-time algorithms, where a large number of tractable problems, including the problems of finding a shortest path or a maximum flow, can be represented in the language of Linear Programming. In Section 6.2, we state the fastest known algorithm for Integer Linear Programming, and exemplify its usage on the Imbalance problem.

In Section 6.3, we move to the theory of graph minors. The Graph Minors project of Robertson and Seymour, developed in the last three decades, resulted not only in achieving its main goal — proving Wagner's conjecture, asserting that the class of all graphs is well-quasi-ordered by the minor relation — but also flourished with other tools, techniques and insights that turned out to have plenty of algorithmic implications. In this chapter, we restrict ourselves only to the implications of the Robertson-Seymour theorem in parameterized complexity, and we show how theorems from Graph Minors immediately imply fixed-parameter tractability of such problems as detecting the Euler genus of a graph. Robertson-Seymour theory also gives us pretext to discuss the notion of *nonuniform* fixed-parameter algorithms. We remark here that the next chapter, Chapter 7, is entirely devoted to the algorithmic usage of the notion of treewidth, a different offspring of the Graph Minors project.

6.1 Dynamic programming over subsets

In this section, we give two examples of dynamic-programming algorithms over families of sets. Our examples are Set Cover and Steiner Tree.

6.1.1 Set Cover

Let $\mathcal{F}$ be a family of sets over a universe U. For a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ and a subset $U' \subseteq U$, we say that $\mathcal{F}'$ *covers* U' if every element of U' belongs to some set of $\mathcal{F}'$, that is, $U' \subseteq \bigcup \mathcal{F}'$. In the Set Cover problem, we are given a family of sets $\mathcal{F}$ over a universe U and a positive integer k, and the task is to check whether there exists a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\mathcal{F}'$ covers U. We give an algorithm for Set Cover that runs in time $2^{|U|}(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$. In fact, this algorithm does not use the value of k, and finds the minimum possible cardinality of a family $\mathcal{F}' \subseteq \mathcal{F}$ that covers U.

Theorem 6.1. *Given a* Set Cover *instance* $(U, \mathcal{F}, k)$*, the minimum possible size of a subfamily* $\mathcal{F}' \subseteq \mathcal{F}$ *that covers* U *can be found in time* $2^{|U|}(|U| + |\mathcal{F}|)^{\mathcal{O}(1)}$.

Proof. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\}$. We define the dynamic-programming table as follows: for every subset $X \subseteq U$ and for every integer $0 \leq j \leq |\mathcal{F}|$, we define $T[X, j]$ as the minimum possible size of a subset $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_j\}$ that covers X. (Henceforth, we call such a family $\mathcal{F}'$ a *valid candidate* for the entry $T[X, j]$.) If no such subset $\mathcal{F}'$ exists (i.e., if $X \not\subseteq \bigcup_{i=1}^{j} F_i$), then $T[X, j] = +\infty$.

In our dynamic-programming algorithm, we compute all $2^{|U|}(|\mathcal{F}| + 1)$ values $T[X, j]$. To achieve this goal, we need to show (a) base cases, in our case values $T[X, j]$ for $j = 0$; (b) recursive computations, in our case how to compute the value $T[X, j]$ knowing values $T[X', j']$ for $j' < j$.

For the base case, observe that $T[\emptyset, 0] = 0$ while $T[X, 0] = +\infty$ for $X \neq \emptyset$.

For the recursive computations, let $X \subseteq U$ and $0 < j \leq |\mathcal{F}|$; we show that

$$T[X, j] = \min(T[X, j-1], 1 + T[X \setminus F_j, j-1]). \tag{6.1}$$

We prove (6.1) by showing inequalities in both directions. In one direction, let $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_j\}$ be a family of minimum size that covers X. We distinguish two cases. If $F_j \notin \mathcal{F}'$, then note that $\mathcal{F}'$ is also a valid candidate for the entry $T[X, j-1]$ (i.e., $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_{j-1}\}$ and $\mathcal{F}'$ covers X). If $F_j \in \mathcal{F}'$, then $\mathcal{F}' \setminus \{F_j\}$ is a valid candidate for the entry $T[X \setminus F_j, j-1]$. In the other direction, observe that any valid candidate $\mathcal{F}'$ for the entry $T[X, j-1]$ is also a valid candidate for $T[X, j]$ and, moreover, for every valid candidate $\mathcal{F}'$ for $T[X \setminus F_j, j-1]$, the family $\mathcal{F}' \cup \{F_j\}$ is a valid candidate for $T[X, j]$. This finishes the proof of (6.1).

By using (6.1), we compute all values $T[X, j]$ for $X \subseteq U$ and $0 \leq j \leq |\mathcal{F}|$ within the promised time bound. Finally, observe that $T[U, |\mathcal{F}|]$ is the answer we are looking for: the minimum size of a family $\mathcal{F}' \subseteq \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\} = \mathcal{F}$ that covers U. □

We remark that, although the dynamic-programming algorithm of Theorem 6.1 is very simple, we suspect that the exponential dependency on $|U|$, that is, the term $2^{|U|}$, is optimal. However, there is no known reduction that supports this claim with the Strong Exponential Time Hypothesis (discussed in Chapter 14).

6.1.2 Steiner Tree

Let G be an undirected graph on n vertices and $K \subseteq V(G)$ be a set of *terminals*. A *Steiner tree* for K in G is a connected subgraph H of G containing K, that is, $K \subseteq V(H)$. As we will always look for a Steiner tree of minimum

possible size or weight, without loss of generality, we may assume that we focus only on subgraphs H of G that are trees. The vertices of $V(H) \setminus K$ are called *Steiner vertices* of H. In the (weighted) Steiner Tree problem, we are given an undirected graph G, a weight function $\mathbf{w}\colon E(G) \to \mathbb{R}_{>0}$ and a subset of terminals $K \subseteq V(G)$, and the goal is to find a Steiner tree H for K in G whose weight $\mathbf{w}(H) = \sum_{e \in E(H)} \mathbf{w}(e)$ is minimized. Observe that if the graph G is unweighted (i.e., $\mathbf{w}(e) = 1$ for every $e \in E(G)$), then we in fact optimize the number of edges of H, and we may equivalently optimize the number of Steiner vertices of H.

For a pair of vertices $u, v \in V(G)$, by $\mathrm{dist}(u, v)$ we denote the cost of a shortest path between u and v in G (i.e., a path of minimum total weight). Let us remind the reader that, for any two vertices u, v, the value $\mathrm{dist}(u, v)$ is computable in polynomial time, say by making use of Dijkstra's algorithm.

The goal of this section is to design a dynamic-programming algorithm for Steiner Tree with running time $3^{|K|} n^{\mathcal{O}(1)}$, where $n = |V(G)|$.

We first perform some *preprocessing steps*. First, assume $|K| > 1$, as otherwise the input instance is trivial. Second, without loss of generality, we may assume that G is connected: a Steiner tree for K exists in G only if all terminals of K belong to the same connected component of G and, if this is the case, then we may focus only on this particular connected component. This assumption ensures that, whenever we talk about some minimum weight Steiner tree or a shortest path, such a tree or path exists in G (i.e., we do not minimize over an empty set). Third, we may assume that each terminal in K is of degree exactly 1 in G and its sole neighbor is not a terminal. To achieve this property, for every terminal $t \in K$, we attach a new neighbor t' of degree 1, that is, we create a new vertex t' and an edge tt' of some fixed weight, say 1. Observe that, if $|K| > 1$, then the Steiner trees in the original graph are in one-to-one correspondence with the Steiner trees in the modified graphs.

We start with defining a table for dynamic programming. For every nonempty subset $D \subseteq K$ and every vertex $v \in V(G) \setminus K$, let $T[D, v]$ be the minimum possible weight of a Steiner tree for $D \cup \{v\}$ in G.

The intuitive idea is as follows: for every subset of terminals D, and for every vertex $v \in V(G) \setminus K$, we consider the possibility that in the optimal Steiner tree H for K, there is a subtree of H that contains D and is attached to the rest of the tree H through the vertex v. For $|D| > 1$, such a subtree decomposes into two smaller subtrees rooted at some vertex u (possibly $u = v$), and a shortest path between v and u. We are able to build such subtrees for larger and larger sets D through the dynamic-programming algorithm, filling up the table $T[D, v]$.

The base case for computing the values $T[D, v]$ is where $|D| = 1$. Observe that, if $D = \{t\}$, then a Steiner tree of minimum weight for $D \cup \{v\} = \{v, t\}$

is a shortest path between v and t in the graph G. Consequently, we can fill in $T[\{t\}, v] = \mathrm{dist}(t, v)$ for every $t \in K$ and $v \in V(G) \setminus K$.

In the next lemma, we show a recursive formula for computing the values $T[D, v]$ for larger sets D.

Lemma 6.2. *For every $D \subseteq K$ of size at least 2, and every $v \in V(G) \setminus K$, the following holds*

$$T[D, v] = \min_{\substack{u \in V(G) \setminus K \\ \emptyset \neq D' \subsetneq D}} \left\{ T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u) \right\}. \tag{6.2}$$

Proof. We prove (6.2) by showing inequalities in both directions.

In one direction, fix $u \in V(G)$ and $\emptyset \neq D' \subsetneq D$. Let H_1 be the tree witnessing the value $T[D', u]$, that is, H_1 is a Steiner tree for $D' \cup \{u\}$ in G of minimum possible weight. Similarly, define H_2 for the value $T[D \setminus D', u]$. Moreover, let P be a shortest path between v and u in G. Observe that $H_1 \cup H_2 \cup P$ is a connected subgraph of G that contains $D \cup \{v\}$ and is of weight

$$\mathbf{w}(H_1 \cup H_2 \cup P) \leq \mathbf{w}(H_1) + \mathbf{w}(H_2) + \mathbf{w}(P) = T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u).$$

Thus

$$T[D, v] \leq \min_{\substack{u \in V(G) \setminus K \\ \emptyset \neq D' \subsetneq D}} \left\{ T[D', u] + T[D \setminus D', u] + \mathrm{dist}(v, u) \right\}.$$

In the opposite direction, let H be a Steiner tree for $D \cup \{v\}$ in G of minimum possible weight. Let us root the tree H in the vertex v, and let u_0 be the vertex of H that has at least two children and, among such vertices, is closest to the root. An existence of such a vertex follows from the assumptions that $|D| \geq 2$ and that every terminal vertex is of degree 1. Moreover, since every terminal of K is of degree 1 in G, we have $u_0 \notin K$. Let u_1 be an arbitrarily chosen child of u_0 in the tree H. We decompose H into the following three edge-disjoint subgraphs:

1. P is the path between u_0 and v in H;
2. H_1 is the subtree of H rooted at u_1, together with the edge u_0u_1;
3. H_2 consists of the remaining edges of H, that is, the entire subtree of H rooted at u_0, except for the descendants of u_1 (that are contained in H_1). See Fig. 6.1.

Let $D' = V(H_1) \cap K$ be the terminals in the tree H_1. Since every terminal is of degree 1 in G, we have $D \setminus D' = V(H_2) \cap K$. Observe that, as H is of minimum possible weight, $D' \neq \emptyset$, as otherwise $H \setminus H_1$ is a Steiner tree for $D \cup \{v\}$ in G. Similarly, we have $D' \subsetneq D$ as otherwise $H \setminus H_2$ is a Steiner tree for $D \cup \{v\}$ in G. Furthermore, note that from the optimality of H it follows that $\mathbf{w}(H_1) = T[D', u_0]$, $\mathbf{w}(H_2) = T[D \setminus D', u_0]$ and, moreover, P is a shortest path between u_0 and v. Consequently,

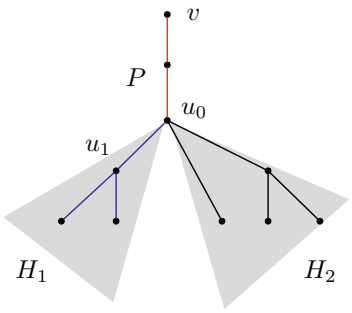


Fig. 6.1: Decomposition of H

$$\begin{aligned} T[D,v] = \mathbf{w}(H) &= T[D',u_0] + T[D \setminus D',u_0] + \operatorname{dist}(v,u_0) \\ &\geq \min_{\substack{u \in V(G) \setminus K \\ \emptyset \neq D' \subsetneq D}} \left\{ T[D',u] + T[D \setminus D',u] + \operatorname{dist}(v,u) \right\}. \end{aligned}$$

This finishes the proof of the lemma. □

With the insight of Lemma 6.2, we can now prove the main result of this section.

Theorem 6.3. Steiner Tree *can be solved in time* $3^{|K|} n^{\mathcal{O}(1)}$.

Proof. Let (G, w, K) be an instance of Steiner Tree after the preprocessing steps have been performed. We compute all values of $T[D, v]$ in the increasing order of the cardinality of the set D. As discussed earlier, in the base case we have $T[\{t\}, v] = \operatorname{dist}(t, v)$ for every $t \in K$ and $v \in V(G) \setminus K$. For larger sets D, we compute $T[D, v]$ using (6.2); note that in this formula we use values of $T[D', u]$ and $T[D \setminus D', u]$, and both D' and $D \setminus D'$ are proper subsets of D. In this manner, a fixed value $T[D, v]$ can be computed in time $2^{|D|} n^{\mathcal{O}(1)}$. Consequently, all values $T[D, v]$ are computed in time

$$\sum_{v \in V(G) \setminus K} \sum_{D \subseteq K} 2^{|D|} n^{\mathcal{O}(1)} \leq n \sum_{j=2}^{|K|} \binom{|K|}{j} 2^j n^{\mathcal{O}(1)} = 3^{|K|} n^{\mathcal{O}(1)}.$$

Finally, observe that, if the preprocessing steps have been performed, then any Steiner tree for K in $V(G)$ needs to contain at least one Steiner point and, consequently, the minimum possible weight of such a Steiner tree equals $\min_{v \in V(G) \setminus K} T[K, v]$. □

We will see in Section 10.1.2 how the result of Theorem 6.3 can be improved.

6.2 Integer Linear Programming

In this section, we take a closer look at parameterized complexity of Integer Linear Programming.

We start with some definitions. In the Integer Linear Programming Feasibility problem, the input consists of p variables $x_1, x_2, \ldots, x_p$, and a set of m inequalities of the following form:

$$\begin{array}{ccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + \ldots + & a_{1,p}x_p & \leq & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + \ldots + & a_{2,p}x_p & \leq & b_2 \\ \vdots & & \vdots & \ddots & \vdots & \vdots & \\ a_{m,1}x_1 & + & a_{m,2}x_2 & + \ldots + & a_{m,p}x_p & \leq & b_m \end{array}$$

where every coefficient $a_{i,j}$ and b_i is required to be an integer. The task is to check whether one can choose *integer* values for every variable x_i so that all inequalities are satisfiable.

Equivalently, one may look at an Integer Linear Programming Feasibility instance as a matrix $A \in \mathbb{Z}^{m \times p}$ and a vector $b \in \mathbb{Z}^m$; the task is to check whether there exists a vector $x \in \mathbb{Z}^p$ such that $Ax \leq b$. We assume that an input of Integer Linear Programming Feasibility is given in binary and thus the size of the input is the number of bits in its binary representation.

In typical applications, when we want to solve a concrete algorithmic problem by formulating it as Integer Linear Programming Feasibility, we may assume that the absolute values of the variables in the solution are polynomially bounded by the size n of the instance we want to solve. Therefore, if the Integer Linear Programming Feasibility instance has p variables, then we may solve it in time $n^{\mathcal{O}(p)}$ by brute force. The main technical tool that we use in this section is the fact that Integer Linear Programming Feasibility is fixed-parameter tractable parameterized by the number of variables. This opens up the possibility of obtaining FPT results by translating the problem into an instance of Integer Linear Programming Feasibility with bounded number of variables.

Theorem 6.4 ([280],[319],[215]). *An* Integer Linear Programming Feasibility *instance of size L with p variables can be solved using $\mathcal{O}(p^{2.5p+o(p)} \cdot L)$ arithmetic operations and space polynomial in L.*

In other words, Theorem 6.4 says that Integer Linear Programming Feasibility is fixed-parameter tractable when parameterized by p, with a relatively good (slightly super-exponential) dependence on the parameter and linear dependence on the input size.

In some applications, it is more convenient to work with an *optimization* version of the Integer Linear Programming Feasibility problem,

namely Integer Linear Programming. Although this problem has been already briefly discussed in Section 2.5, let us now recall its definition. In the Integer Linear Programming problem, apart from the standard input for Integer Linear Programming Feasibility (i.e., a matrix $A \in \mathbb{Z}^{m \times p}$ and a vector $b \in \mathbb{Z}^m$) we are given a vector $c \in \mathbb{Z}^p$, and our goal is to find a vector $x \in \mathbb{Z}^p$ satisfying all the aforementioned inequalities (i.e., $Ax \leq b$) that *minimizes* the *objective function* $c \cdot x$ (the scalar product of c and x).

Using binary search, it is easy to derive an algorithm for Integer Linear Programming using Theorem 6.4.

Theorem 6.5. *An* Integer Linear Programming *instance of size L with p variables can be solved using*

$$\mathcal{O}(p^{2.5p+o(p)} \cdot (L + \log M_x) \log(M_x M_c))$$

arithmetic operations and space polynomial in $L + \log M_x$, where M_x is an upper bound on the absolute value a variable can take in a solution, and M_c is the largest absolute value of a coefficient in the vector c.

Proof. Observe that the absolute value of the objective function is at most pM_xM_c as long as the variables have absolute values at most M_x. We perform a binary search to find the minimum value of the objective function. That is, for a fixed integer threshold $-pM_xM_c \leq t \leq pM_xM_c$, we add an inequality $cx \leq t$ to the system $Ax \leq b$ and apply the algorithm of Theorem 6.4 to the obtained Integer Linear Programming Feasibility instance. The instance has size $\mathcal{O}(L + p \log(pM_xM_c)) = \mathcal{O}(p(L + \log M_x))$, and hence each application of Theorem 6.4 runs in time $\mathcal{O}(p^{2.5p+o(p)} \cdot (L + \log M_x))$. Consequently, we are able to find an optimum value t_0 of the objective function within the promised bound on the running time. Moreover, any solution to the Integer Linear Programming Feasibility instance consisting of the system $Ax \leq b$ with an additional inequality $cx \leq t_0$ is an optimal solution to the input Integer Linear Programming instance. □

6.2.1 The example of Imbalance

Now we exemplify the usage of Theorem 6.5 on the Imbalance problem.

In order to define the problem itself, we need to introduce some notation. Let G be an n-vertex undirected graph. An *ordering* of $V(G)$ is any bijective function $\pi \colon V(G) \to \{1, 2, \ldots, n\}$. For $v \in V(G)$, we define $L_\pi(v) = \{u \in N(v) \ : \ \pi(u) < \pi(v)\}$ and $R_\pi(v) = \{u \in N(v) \ : \ \pi(u) > \pi(v)\} = N(v) \setminus L_\pi(v)$. Thus $L_\pi(v)$ is the set of vertices preceding v, and $R_\pi(v)$ is the set of vertices succeeding v in π. The *imbalance at vertex v* is defined as $\iota_\pi(v) = ||L_\pi(v)| - |R_\pi(v)||$, and the *imbalance of the ordering π* equals $\iota(\pi) = \sum_{v \in V(G)} \iota_\pi(v)$.

Chapter 7
Treewidth

The treewidth of a graph is one of the most frequently used tools in parameterized algorithms. Intuitively, treewidth measures how well the structure of a graph can be captured by a tree-like structural decomposition. When the treewidth of a graph is small, or equivalently the graph admits a good tree decomposition, then many problems intractable on general graphs become efficiently solvable. In this chapter we introduce treewidth and present its main applications in parameterized complexity. We explain how good tree decompositions can be exploited to design fast dynamic-programming algorithms. We also show how treewidth can be used as a tool in more advanced techniques, like shifting strategies, bidimensionality, or the irrelevant vertex approach.

In Section 6.3, we gave a brief overview on how the deep theory of Graph Minors of Robertson and Seymour can be used to obtain nonconstructive FPT algorithms. One of the tools defined by Robertson and Seymour in their work was the treewidth of a graph. Very roughly, treewidth captures how similar a graph is to a tree. There are many ways to define "tree-likeness" of a graph; for example, one could measure the number of cycles, or the number of vertices needed to be removed in order to make the graph acyclic. However, it appears that the approach most useful from algorithmic and graph theoretical perspectives, is to view tree-likeness of a graph G as the existence of a structural decomposition of G into pieces of bounded size that are connected in a tree-like fashion. This intuitive concept is formalized via the notions of a *tree decomposition* and the *treewidth* of a graph; the latter is a quantitative measure of how good a tree decomposition we can possibly obtain.

Treewidth is a fundamental tool used in various graph algorithms. In parameterized complexity, the following win/win approach is commonly ex-

M. Cygan et al., *Parameterized Algorithms*,
DOI 10.1007/978-3-319-21275-3_7

ploited. Given some computational problem, let us try to construct a good tree decomposition of the input graph. If we succeed, then we can use dynamic programming to solve the problem efficiently. On the other hand, if we fail and the treewidth is large, then there is a reason for this outcome. This reason has the form of a combinatorial obstacle embedded in the graph that forbids us to decompose it expeditiously. However, the existence of such an obstacle can also be used algorithmically. For example, for some problems like Vertex Cover or Feedback Vertex Set, we can immediately conclude that we are dealing with a no-instance in case the treewidth is large. For other problems, like Longest Path, large treewidth implies that we are working with a yes-instance. In more complicated cases, one can examine the structure of the obstacle in the hope of finding a so-called irrelevant vertex or edge, whose removal does not change the answer to the problem. Thus, regardless of whether the initial construction of a good tree decomposition succeeded or failed, we win: we solve the problem by dynamic programming, or we are able to immediately provide the answer, or we can simplify the problem and restart the algorithm.

We start the chapter by slowly introducing treewidth and tree decompositions, and simultaneously showing connections to the idea of dynamic programming on the structure of a graph. In Section 7.1 we build the intuition by explaining, on a working example of Weighted Independent Set, how dynamic-programming procedures can be designed on trees and on subgraphs of grids. These examples bring us naturally to the definitions of path decompositions and pathwidth, and of tree decompositions and treewidth; these topics are discussed in Section 7.2. In Section 7.3 we provide the full framework of dynamic programming on tree decompositions. We consider carefully three exemplary problems: Weighted Independent Set, Dominating Set, and Steiner Tree; the examples of Dominating Set and Steiner Tree will be developed further in Chapter 11.

Section 7.4 is devoted to connections between graphs of small treewidth and monadic second-order logic on graphs. In particular, we discuss a powerful meta-theorem of Courcelle, which establishes the tractability of decision problems definable in Monadic Second-Order logic on graphs of bounded treewidth. Furthermore, we also give an extension of Courcelle's theorem to optimization problems.

In Section 7.5 we present a more combinatorial point of view on pathwidth and treewidth, by providing connections between these graph parameters, various search games on graphs, and classes of interval and chordal graphs. While these discussions are not directly relevant to the remaining part of this chapter, we think that they give an insight into the nature of pathwidth and treewidth that is invaluable when working with them.

In Section 7.6 we address the question of how to compute a reasonably good tree decomposition of a graph. More precisely, we present an approximation algorithm that, given an n-vertex graph G and a parameter k, works in time $\mathcal{O}(8^k k^2 \cdot n^2)$ and either constructs a tree decomposition of G of width at

most $4k + 4$, or reports correctly that the treewidth of G is more than k. Application of this algorithm is usually the first step of the classic win/win framework described in the beginning of this section.

We then move to using treewidth and tree decompositions as tools in more involved techniques. Section 7.7 is devoted to applications of treewidth for designing FPT algorithms on planar graphs. Most of these applications are based on deep structural results about obstacles to admitting good tree decompositions. More precisely, in Section 7.7.1 we introduce the so-called Excluded Grid Theorem and some of its variants, which states that a graph of large treewidth contains a large grid as minor. In the case of planar graphs, the theorem gives a very tight relation between the treewidth and the size of the largest grid minor that can be found in a graph. This fact is then exploited in Section 7.7.2, where we introduce the powerful framework of *bidimensionality*, using which one can derive parameterized algorithms on planar graphs with subexponential parametric dependence of the running time. In Section 7.7.3 we discuss the parameterized variant of the shifting technique; a reader familiar with basic results on approximation algorithms may have seen this method from a different angle. We apply the technique to give fixed-parameter tractable algorithms on planar graphs for Subgraph Isomorphism and Minimum Bisection.

In Section *7.8 we give an FPT algorithm for a problem where a more advanced version of the treewidth win/win approach is implemented. More precisely, we provide an FPT algorithm for Planar Vertex Deletion, the problem of deleting at most k vertices to obtain a planar graph. The crux of this algorithm is to design an *irrelevant vertex rule*: to prove that if a large grid minor can be found in the graph, one can identify a vertex that can be safely deleted without changing the answer to the problem. This technique is very powerful, but also requires attention to many technical details. For this reason, some technical steps of the correctness proof are omitted.

The last part of this chapter, Section 7.9, gives an overview of other graph parameters related to treewidth, such as branchwidth and rankwidth.

7.1 Trees, narrow grids, and dynamic programming

Imagine that you want to have a party and invite some of your colleagues from work to come to your place. When preparing the list of invitations, you would like to maximize the *total fun factor* of the invited people. However, from experience you know that there is not much fun when your direct boss is also invited. As you want everybody at the party to have fun, you would rather avoid such a situation for any of the invited colleagues.

We model this problem as follows. Assume that job relationships in your company are represented by a rooted tree T. Vertices of the tree represent your colleagues, and each $v \in V(T)$ is assigned a nonnegative weight $\mathbf{w}(v)$

that represents the amount of contributed fun for a particular person. The task is to find the maximum weight of an independent set in T, that is, of a set of pairwise nonadjacent vertices. We shall call this problem Weighted Independent Set.

This problem can be easily solved on trees by making use of dynamic programming. As usual, we solve a large number of subproblems that depend on each other. The answer to the problem shall be the value of a single, topmost subproblem. Assume that r is the root of the tree T (which corresponds to the superior of all the employees, probably the CEO). For a vertex v of T, let T_v be the subtree of T rooted at v. For the vertex v we define the following two values:

- Let $A[v]$ be the maximum possible weight of an independent set in T_v.
- Let $B[v]$ be the maximum possible weight of an independent set in T_v that does not contain v.

Clearly, the answer to the whole problem is the value of $A[r]$.

Values of $A[v]$ and $B[v]$ for leaves of T are equal to $\mathbf{w}(v)$ and 0, respectively. For other vertices, the values are calculated in a bottom-up order. Assume that $v_1, \ldots, v_q$ are the children of v. Then we can use the following recursive formulas:

$$B[v] = \sum_{i=1}^{q} A[v_i]$$

and

$$A[v] = \max\left\{B[v], \mathbf{w}(v) + \sum_{i=1}^{q} B[v_i]\right\}.$$

Intuitively, the correctness of these formulas can be explained as follows. We know that $B[v]$ stores the maximum possible weight of an independent set in T_v that does not contain v and, thus, the independent set we are seeking is contained in $T_{v_1}, \ldots, T_{v_q}$. Furthermore, since T_v is a tree, there is no edge between vertices of two distinct subtrees among $T_{v_1}, \ldots, T_{v_q}$. This in turn implies that the maximum possible weight of an independent set of T_v that does not contain v is the sum of maximum possible weights of an independent set of T_{v_i}, $i \in \{1, \ldots, q\}$. The formula for $A[v]$ is justified by the fact that an independent set in T_v of the maximum possible weight either contains v, which is taken care of by the term $\mathbf{w}(v) + \sum_{i=1}^{q} B[v_i]$, or does not contain v, which is taken care of by the term $B[v]$. Therefore, what remains to do is to calculate the values of $A[v]$ and $B[v]$ in a bottom-up manner in the tree T, and finally read the answer from $A[r]$. This procedure can be clearly implemented in linear time. Let us remark that with a slight modification of the algorithm, using a standard method of remembering the origin of computed values as backlinks, within the same running time one can find not only the maximum possible weight, but also the corresponding independent set.

Let us now try to solve Weighted Independent Set on a different class of graphs. The problem with trees is that they are inherently "thin", so let us

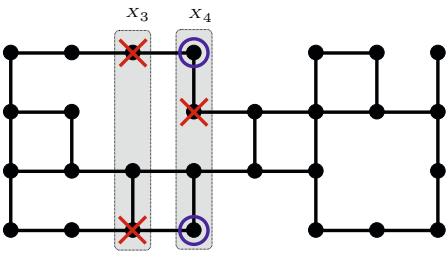


Fig. 7.1: A subgraph of a 4×8 grid, with the third and the fourth columns highlighted. When computing $c[4, Y]$ for the forbidden set $Y = \{(2, 4)\}$ (crossed out in the fourth column), one of the sets S we consider is $S = \{(1, 4), (4, 4)\}$, depicted with blue circles. Then, when looking at the previous column we need to forbid picking neighbors $(1, 3)$ and $(4, 3)$ (crossed out in the third column), since this would violate the independence constraint

try to look at graphs that are "thicker" in nature, like grids. Since the class of grids is not very broad, let us rather focus on subgraphs of grids. More precisely, assume that we are given a graph G that is a subgraph of a $k \times N$ grid. The vertex set of a $k \times N$ grid consists of all pairs of the form (i, j) for $1 \leq i \leq k$ and $1 \leq j \leq N$, and two pairs (i_1, j_1) and (i_2, j_2) are adjacent if and only if $|i_1 - i_2| + |j_1 - j_2| = 1$. Graph G is a subgraph of an $k \times N$ grid, which means that some vertices and edges of the grid can be missing in G. Figure 7.1 presents an example of a subgraph of a 4×8 grid. In our considerations, we will think of the number of rows k as quite small (say, $k = 10$), while N, the number of columns, can be very large (say, $N = 10^6$). Of course, since we are trying to solve the Weighted Independent Set problem, every vertex $v \in V(G)$ is assigned its weight $\mathbf{w}(v)$.

Let X_j be the j-th column of G, that is, $X_j = V(G) \cap \{(i, j) : 1 \leq i \leq k\}$. Moreover, for $1 \leq j \leq N$, let $G_j = G[X_1 \cup X_2 \cup \ldots \cup X_j]$ be the graph induced by the first j columns. We would like to run a dynamic programming-algorithm that sweeps the grid from left to right, column by column. In the case of trees, we recognized two possible situations that were handled differently in the two dynamic-programming tables: either the root of the subtree was allowed to be picked into an independent set, or it was forbidden. Mimicking this idea, let us define the following function $c[j, Y]$ that we shall compute in the algorithm. For $Y \subseteq X_j$, we take the following definition:

$$c[j, Y] = \text{maximum possible weight of an independent set in } G_j - Y.$$

In other words, we are examining graph G_j, and we look for the best possible independent set that avoids picking vertices from Y. We now move on to explaining how the values of $c[j, Y]$ will be computed.

For $j = 1$, the situation is very simple. For every $Y \subseteq X_1$, we iterate through all possible subsets $S \subseteq X_1 \setminus Y$, and for each of them we check whether it is an independent set. Then $c[1, Y]$ is the largest weight among the candidates (independent sets) that passed this test. Since for each $Y \subseteq X_1$ we iterate through all possible subsets of $X_1 \setminus Y$, in total, for all the Ys, the number of checks is bounded by

$$\sum_{\ell=0}^{|X_1|} \binom{|X_1|}{\ell} 2^{|X_1|-\ell} = 3^{|X_1|} \leq 3^k. \tag{7.1}$$

Here, factor $\binom{|X_1|}{\ell}$ comes from the choice of Y (the sum iterates over $\ell = |Y|$), while factor $2^{|X_1|-\ell}$ represents the number of choices for $S \subseteq X_1 \setminus Y$. Since every check can be implemented in $k^{\mathcal{O}(1)}$ time (assuming that for every vertex we store a list of its neighbors), the total time spent on computing values $c[1, \cdot]$ is $3^k \cdot k^{\mathcal{O}(1)}$.

We now show how to compute the values of $c[j, \cdot]$ depending on the precomputed values of $c[j-1, \cdot]$, for $j > 1$. Let us look at one value $c[j, Y]$ for some $Y \subseteq X_j$. Similarly, for $j = 1$, we should iterate through all the possible ways a maximum weight independent set intersects column X_j. This intersection should be independent, of course, and moreover if we pick some $v \in X_j \setminus Y$ to the independent set, then this choice forbids choosing its neighbor in the previous column X_{j-1} (providing this neighbor exists). But for column X_{j-1} we have precomputed answers for *all* possible combinations of forbidden vertices. Hence, we can easily read from the precomputed values what is the best possible weight of an extension of the considered intersection with X_j to the previous columns. All in all, we arrive at the following recursive formula:

$$c[j, Y] = \max_{\substack{S \subseteq X_i \setminus Y \\ S \text{ is independent}}} \Big\{ \mathbf{w}(S) + c[j-1, N(S) \cap X_{j-1}] \Big\}; \tag{7.2}$$

here $\mathbf{w}(S) = \sum_{v \in S} \mathbf{w}(v)$. Again, when applying (7.2) for every $Y \subseteq X_i$ we iterate through all the subsets of $X_j \setminus Y$. Therefore, as in (7.1) we obtain that the total number of sets S checked, for all the sets Y, is at most 3^k. Each S is processed in $k^{\mathcal{O}(1)}$ time, so the total time spent on computing values $c[j, \cdot]$ is $3^k \cdot k^{\mathcal{O}(1)}$.

To wrap up, we first compute the values $c[1, \cdot]$, then iteratively compute values $c[j, \cdot]$ for $j \in \{2, 3, \ldots, N\}$ using (7.2), and conclude by observing that the answer to the problem is equal to $c[N, \emptyset]$. As argued, each iteration takes time $3^k \cdot k^{\mathcal{O}(1)}$, so the whole algorithm runs in time $3^k \cdot k^{\mathcal{O}(1)} \cdot N$.

Let us now step back and look at the second algorithm that we designed. Basically, the only property of G that we really used is that $V(G)$ can be partitioned into a sequence of small subsets (columns, in our case), such that edges of G can connect only two consecutive subsets. In other words, G has a "linear structure of separators", such that each separator separates the part

lying on the left of it from the part on the right. In the algorithm we actually used the fact that the columns are disjoint, but this was not that crucial. The main point was that only two consecutive columns can interact.

Let us take a closer look at the choice of the definition of the table $c[j, Y]$. Let I be an independent set in a graph G that is a subgraph of a $k \times N$ grid. For fixed column number j, we can look at the index Y in the cell $c[j, Y]$ as the succinct representation of the interaction between $I \cap \bigcup_{a=1}^{j} X_a$ and $I \cap \bigcup_{a=j+1}^{N} X_a$. More precisely, assume that $Y = X_j \cap N(I \cap X_{j+1})$ and, consequently, the size of $C = I \cap \bigcup_{a=1}^{j} X_a$ is one of the candidates for the value $c[j, Y]$. Furthermore, let C' be any other candidate for the value $c[j, Y]$, that is, let C' be any independent set in $G_j \setminus Y$. Observe that then $(I \setminus C) \cup C'$ is also an independent set in G. In other words, when going from the column j to the column $j + 1$, the only information we need to remember is the set Y of vertices "reserved as potential neighbors", and, besides the set Y, we do not care how exactly the solution looked so far.

If we now try to lift these ideas to general graphs, then the obvious thing to do is to try to merge the algorithms for trees and for subgraphs of grids into one. As $k \times N$ grids are just "fat" paths, we should define the notion of trees of "fatness" k. This is exactly the idea behind the notion of treewidth. In the next section we shall first define parameter *pathwidth*, which encapsulates in a more general manner the concepts that we used for the algorithm on subgraphs of grids. From pathwidth there will be just one step to the definition of *treewidth*, which also encompasses the case of trees, and which is the main parameter we shall be working with in this chapter.

7.2 Path and tree decompositions

We have gathered already enough intuition so that we are ready to introduce formally the main notions of this chapter, namely path and tree decompositions.

Path decompositions. A *path decomposition* of a graph G is a sequence $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of *bags*, where $X_i \subseteq V(G)$ for each $i \in \{1, 2, \ldots, r\}$, such that the following conditions hold:

(P1) $\bigcup_{i=1}^{r} X_i = V(G)$. In other words, every vertex of G is in at least one bag.

(P2) For every $uv \in E(G)$, there exists $\ell \in \{1, 2, \ldots, r\}$ such that the bag X_ℓ contains both u and v.

(P3) For every $u \in V(G)$, if $u \in X_i \cap X_k$ for some $i \leq k$, then $u \in X_j$ also for each j such that $i \leq j \leq k$. In other words, the indices of the bags containing u form an interval in $\{1, 2, \ldots, r\}$.

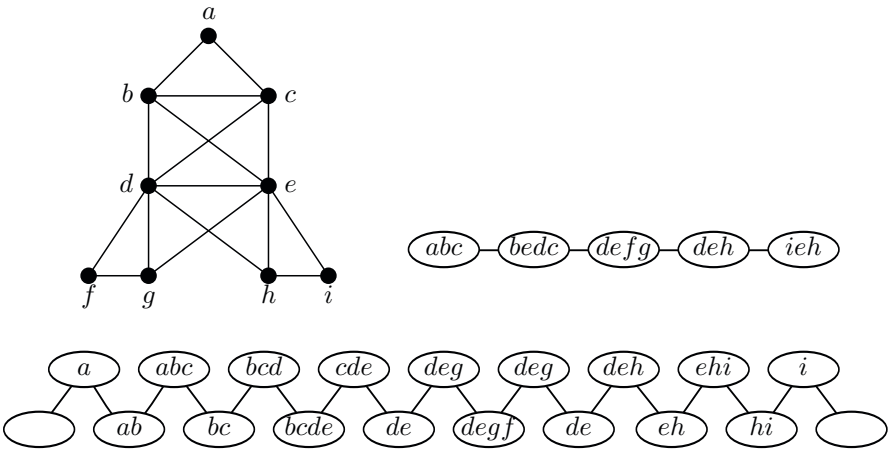


Fig. 7.2: A graph, its path and nice path decompositions

See Fig. 7.2 for example of a path decomposition. The *width* of a path decomposition $(X_1, X_2, \ldots, X_r)$ is $\max_{1 \le i \le r} |X_i| - 1$. The *pathwidth* of a graph G, denoted by $\mathrm{pw}(G)$, is the minimum possible width of a path decomposition of G. The reason for subtracting 1 in the definition of the width of the path decomposition is to ensure that the pathwidth of a path with at least one edge is 1, not 2. Similarly, we subtract in the definition of treewidth to ensure that the treewidth of a tree is 1.

We can also interpret the bags of a path decomposition as nodes of a path and two consecutive bags correspond to two adjacent nodes of the path. This interpretation will become handy when we introduce tree decomposition.

For us the most crucial property of path decompositions is that they define a sequence of separators in the graph. In the following, we will say that (A, B) is a *separation* of a graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. Then $A \cap B$ is a *separator* of this separation, and $|A \cap B|$ is the *order* of the separation. Note that any path in G that begins in A and ends in B must contain at least one vertex of the separator $A \cap B$. Also, for a subset $A \subseteq V(G)$ we define the *border* of A, denoted by $\partial(A)$, as the set of those vertices of A that have a neighbor in $V(G) \setminus A$. Note that $(A, (V(G) \setminus A) \cup \partial(A))$ is a separation with separator $\partial(A)$. Let us remark that by Lemma 7.1, each of the bags X_i separates vertices in the bags before i with the vertices of the bags following after i.

Lemma 7.1. *Let $(X_1, X_2, \ldots, X_r)$ be a path decomposition of a graph G. Then for every $j \in \{1, \ldots, r-1\}$ it holds that $\partial(\bigcup_{i=1}^{j} X_i) \subseteq X_j \cap X_{j+1}$. In other words, $(\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$ is a separation of G with separator $X_j \cap X_{j+1}$.*

Proof. Let us fix j and let $(A, B) = (\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$. We first show that $\partial(\bigcup_{i=1}^{j} X_i) = \partial(A) \subseteq X_j \cap X_{j+1}$. Targeting a contradiction, let us assume that there is a $u \in \partial(A)$ such that $u \notin X_j \cap X_{j+1}$. This means that there is an edge $uv \in E(G)$ such that $u \in A$, $v \notin A$ but also $u \notin X_j \cap X_{j+1}$. Let i be the largest index such that $u \in X_i$ and k be the smallest index such that $v \in X_k$. Since $u \in A$ and $u \notin X_j \cap X_{j+1}$, (P3) implies that $i \leq j$. Since $v \notin A$, we have also $k \geq j+1$. Therefore $i < k$. On the other hand, by (P2) there should be a bag X_ℓ containing both u and v. We obtain that $\ell \leq i < k \leq \ell$, which is a contradiction. The fact that $A \cap B = X_j \cap X_{j+1}$ follows immediately from (P3). □

Note that we can always assume that no two consecutive bags of a path decomposition are equal, since removing one of such bags does not violate any property of a path decomposition. Thus, a path decomposition $(X_1, X_2, \ldots, X_r)$ of width p naturally defines a sequence of separations $(\bigcup_{i=1}^{j} X_i, \bigcup_{i=j+1}^{r} X_i)$. Each of these separations has order at most p, because the intersection of two different sets of size at most $p+1$ has size at most p.

We now introduce sort of a "canonical" form of a path decomposition, which will be useful in the dynamic-programming algorithms presented in later sections. A path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of a graph G is *nice* if

- $X_1 = X_r = \emptyset$, and
- for every $i \in \{1, 2, \ldots, r-1\}$ there is either a vertex $v \notin X_i$ such that $X_{i+1} = X_i \cup \{v\}$, or there is a vertex $w \in X_i$ such that $X_{i+1} = X_i \setminus \{w\}$.

See Fig. 7.2 for example.

Bags of the form $X_{i+1} = X_i \cup \{v\}$ shall be called *introduce bags* (or *introduce nodes*, if we view the path decomposition as a path rather than a sequence). Similarly, bags of the form $X_{i+1} = X_i \setminus \{w\}$ shall be called *forget bags* (*forget nodes*). We will also say that X_{i+1} *introduces* v or *forgets* w. Let us note that because of (P3), every vertex of G gets introduced and becomes forgotten exactly once in a nice path decomposition, and hence we have that r, the total number of bags, is exactly equal to $2|V(G)|+1$. It turns out that every path decomposition can be turned into a nice path decomposition of at most the same width.

Lemma 7.2. *If a graph G admits a path decomposition of width at most p, then it also admits a nice path decomposition of width at most p. Moreover, given a path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ of G of width at most p, one can in time $\mathcal{O}(p^2 \cdot \max(r, |V(G)|))$ compute a nice path decomposition of G of width at most p.*

The reader is asked to prove Lemma 7.2 in Exercise 7.1.

Tree decompositions. A tree decomposition is a generalization of a path decomposition. Formally, a *tree decomposition* of a graph G is a pair $\mathcal{T} =$

$(T, \{X_t\}_{t \in V(T)})$, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

(T1) $\bigcup_{t \in V(T)} X_t = V(G)$. In other words, every vertex of G is in at least one bag.
(T2) For every $uv \in E(G)$, there exists a node t of T such that bag X_t contains both u and v.
(T3) For every $u \in V(G)$, the set $T_u = \{t \in V(T) \ : \ u \in X_t\}$, i.e., the set of nodes whose corresponding bags contain u, induces a connected subtree of T.

The *width* of tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ equals $\max_{t \in V(T)} |X_t| - 1$, that is, the maximum size of its bag minus 1. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum possible width of a tree decomposition of G.

To distinguish between the vertices of the decomposition tree T and the vertices of the graph G, we will refer to the vertices of T as *nodes*. As we mentioned before, path decompositions correspond exactly to tree decompositions with the additional requirement that T has to be a path.

Let us give several examples of how small or large can be the treewidth of particular graph classes. Forests and trees are of treewidth at most 1 and cycles have treewidth 2, see Exercise 7.8. The treewidth of an outerplanar graph, which is a graph that can be drawn in the plane in such manner that all its vertices are on one face, is at most 2, see Exercise 7.12. On the other hand, the treewidth of planar graphs can be arbitrarily large. For example, as we will see later, the treewidth of a $t \times t$ grid is t. Interestingly, for every planar graph H, there is a constant c_H, such that for every graph G excluding H as a minor, the treewidth of G does not exceed c_H, see Exercise 7.36.

While planar graphs can have arbitrarily large treewidths, as we will see later, still the treewidth of an n-vertex planar graph is sublinear, more precisely $\mathcal{O}(\sqrt{n})$. The treewidth of an n-vertex clique K_n is $n-1$ and of a complete bipartite graph $K_{n,m}$ is $\min\{m, n\} - 1$. Expander graphs serve as an example of a sparse graph class with treewidth $\Omega(n)$, see Exercise 7.34.

In Lemma 7.1, we have proved that for every pair of adjacent nodes of a path decomposition, the intersection of the corresponding bags is a separator that separates the left part of the decomposition from the right part. The following lemma establishes a similar separation property of bags of a tree decomposition. Its proof is similar to the proof of Lemma 7.1 and is left to the reader as Exercise 7.5.

Lemma 7.3. *Let $(T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of a graph G and let ab be an edge of T. The forest $T - ab$ obtained from T by deleting edge ab consists of two connected components T_a (containing a) and T_b (containing b). Let $A = \bigcup_{t \in V(T_a)} X_t$ and $B = \bigcup_{t \in V(T_b)} X_t$. Then $\partial(A), \partial(B) \subseteq X_a \cap X_b$. Equivalently, (A, B) is a separation of G with separator $X_a \cap X_b$.*

Again, note that we can always assume that the bags corresponding to two adjacent nodes in a tree decomposition are not the same, since in such situation we could contract the edge between them, keeping the same bag in the node resulting from the contraction. Thus, if $(T, \{X_t\}_{t \in V(T)})$ has width t, then each of the separations given by Lemma 7.3 has order at most t.

Similarly to nice path decompositions, we can also define nice tree decompositions of graphs. It will be convenient to think of nice tree decompositions as rooted trees. That is, for a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ we distinguish one vertex r of T which will be the root of T. This introduces natural parent-child and ancestor-descendant relations in the tree T. We will say that such a rooted tree decomposition $(T, \{X_t\}_{t \in V(T)})$ is *nice* if the following conditions are satisfied:

- $X_r = \emptyset$ and $X_\ell = \emptyset$ for every leaf ℓ of T. In other words, all the leaves as well as the root contain empty bags.
- Every non-leaf node of T is of one of the following three types:
  - **Introduce node**: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$; we say that v is *introduced* at t.
  - **Forget node**: a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$; we say that w is *forgotten* at t.
  - **Join node**: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

At first glance the condition that the root and the leaves contain empty bags might seem unnatural. As we will see later, this property helps to streamline designing dynamic-programming algorithms on tree decompositions, which is the primary motivation for introducing nice tree decompositions. Note also that, by property (T3) of a tree decomposition, every vertex of $V(G)$ is forgotten only once, but may be introduced several times.

Also, note that we have assumed that the bags at a join node are equal to the bags of the children, sacrificing the previous observation that any separation induced by an edge of the tree T is of order at most t. The increase of the size of the separators from t to $t + 1$ has negligible effect on the asymptotic running times of the algorithms, while nice tree decompositions turn out to be very convenient for describing the details of the algorithms.

The following result is an analogue of Lemma 7.2 for nice tree decompositions. The reader is asked to prove it in Exercise 7.2.

Lemma 7.4. *If a graph G admits a tree decomposition of width at most k, then it also admits a nice tree decomposition of width at most k. Moreover, given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k, one can in time $\mathcal{O}(k^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree decomposition of G of width at most k that has at most $\mathcal{O}(k|V(G)|)$ nodes.*

Due to Lemma 7.4, we will assume that all the nice tree decompositions used by our algorithms have $\mathcal{O}(k|V(G)|)$ nodes.

Note that in a general setting a good path/tree decomposition of an input graph is not known in advance. Hence, in the dynamic-programming algorithms we will always assume that such a decomposition is provided on the input together with the graph. For this reason, we will need to address separately how to compute its path/tree decomposition of optimum or near-optimum width, so that an efficient dynamic-programming algorithm can be employed on it. In this book we shall not answer this question for path decompositions and pathwidth. However, in Section 7.6 we present algorithms for computing tree decompositions of (approximately) optimum width.

7.3 Dynamic programming on graphs of bounded treewidth

In this section we give examples of dynamic-programming-based algorithms on graphs of bounded treewidth.

7.3.1 Weighted Independent Set

In Section 7.1 we gave two dynamic-programming routines for the Weighted Independent Set problem. The first of them worked on trees, and the second of them worked on subgraphs of grids. Essentially, in both cases the main idea was to define subproblems for parts of graphs separated by small separators. In the case of trees, every vertex of a tree separates the subtree rooted at it from the rest of the graph. Thus, choices made by the solution in the subtree are independent of what happens outside it. The algorithm for subgraphs of grids exploited the same principle: every column of the grid separates the part of the graph on the left of it from the part on the right.

It seems natural to combine these ideas for designing algorithms for graphs of bounded treewidth. Let us focus on our running example of the Weighted Independent Set problem, and let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of the input n-vertex graph G that has width at most k. By applying Lemma 7.4 we can assume that $\mathcal{T}$ is a nice tree decomposition. Recall that then T is rooted at some node r. For a node t of T, let V_t be the union of all the bags present in the subtree of T rooted at t, including X_t. Provided that $t \neq r$ we can apply Lemma 7.3 to the edge of T between t and its parent, and infer that $\partial(V_t) \subseteq X_t$. The same conclusion is trivial when $t = r$, since then $V_r = V(G)$ and $\partial(V_r) = \emptyset$. This exactly formalizes the intuition that the subgraph induced by V_t can communicate with the rest of the graph only via bag X_t, which is of small size.

Extending our intuition from the algorithms of Section 7.1, we would like to define subproblems depending on the interaction between the solution and

bag X_t. Consider the following: Let I_1, I_2 be two independent sets of G such that $I_1 \cap X_t = I_2 \cap X_t$. Let us add the weights of vertices of I_1 and I_2 that are contained in V_t, and suppose that it turned out that $\mathbf{w}(I_1 \cap V_t) > \mathbf{w}(I_2 \cap V_t)$. Observe that then solution I_2 is suboptimal for the following reason. We can obtain a solution I_2' from I_2 by replacing $I_2 \cap V_t$ with $I_1 \cap V_t$. The fact that X_t separates $V_t \setminus X_t$ from the rest of the graph, and that I_1 and I_2 have the same intersection with X_t, implies that I_2' is still an independent set. On the other hand, $\mathbf{w}(I_1 \cap V_t) > \mathbf{w}(I_2 \cap V_t)$ implies that $\mathbf{w}(I_2') > \mathbf{w}(I_2)$.

Therefore, among independent sets I satisfying $I \cap X_t = S$ for some fixed S, all the maximum-weight solutions have exactly the same weight of the part contained in V_t. This weight corresponds to the maximum possible value for the following subproblem: given $S \subseteq X_t$, we look for a maximum-weight extension $\widehat{S} \supseteq S$ such that $\widehat{S} \subseteq V_t$, $\widehat{S} \cap X_t = S$, and $\widehat{S}$ is independent. Indeed, the argument from the previous paragraph shows that for any solution I with $I \cap X_t = S$, the part of the solution contained in V_t can be safely replaced with the best possible partial solution $\widehat{S}$ — this replacement preserves independence and can only increase the weight. Observe that the number of subproblems is small: for every node t, we have only $2^{|X_t|}$ subproblems. Also, we do not need to remember $\widehat{S}$ explicitly; remembering its weight will suffice.

Hence, we now mimic the bottom-up dynamic programming that we performed for trees. For every node t and every $S \subseteq X_t$, define the following value:

$$c[t, S] = \text{maximum possible weight of a set } \widehat{S} \text{ such that } S \subseteq \widehat{S} \subseteq V_t,\ \widehat{S} \cap X_t = S, \text{ and } \widehat{S} \text{ is independent.}$$

If no such set $\widehat{S}$ exists, then we put $c[t, S] = -\infty$; note that this happens if and only if S is not independent itself. Also $c[r, \emptyset]$ is exactly the maximum weight of an independent set in G; this is due to the fact that $V_r = V(G)$ and $X_r = \emptyset$.

The reader can now observe that this definition of function $c[\cdot, \cdot]$ differs from the one used in Section 7.1. While there we were just *forbidding* vertices from some set Y from being used, now we fix *exactly* how a solution is supposed to interact with a bag. Usually, fixing the exact interaction of the solution with a bag is a more generic approach to designing dynamic-programming algorithms on tree decompositions. However, it must be admitted that tweaking the definition slightly (e.g., by relaxing the exactness condition to allow or forbid usage of a subset of vertices) often leads to a simpler description. This is actually the case also in the example of Weighted Independent Set. In Exercise 7.17 the reader is asked to work out the details of a dynamic program with the definition of a state mimicking the one used in Section 7.1.

We now move on to presenting how the values of $c[\cdot, \cdot]$ are computed. Thanks to the definition of a nice tree decomposition, there are only a few simple ways in which a bag at some node can relate to the bags of the children

of this node. Therefore, if we are fortunate we can compute the values at each node t based on the values computed for the children of t. This will be done by giving recursive formulas for $c[t, S]$. The case when t is a leaf corresponds to the base case of the recurrence, whereas values for $c[\cdot, \cdot]$ for a non-leaf node t depend on the values of $c[\cdot, \cdot]$ for the children of t. By applying the formulas in a bottom-up manner on T we will finally compute $c[r, \emptyset]$, which is the value we are looking for.

Leaf node. If t is a leaf node, then we have only one value $c[t, \emptyset] = 0$.

Introduce node. Suppose t is an introduce node with child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Let S be any subset of X_t. If S is not independent, then we can immediately put $c[t, S] = -\infty$; hence assume otherwise. Then we claim that the following formula holds:

$$c[t, S] = \begin{cases} c[t', S] & \text{if } v \notin S; \\ c[t', S \setminus \{v\}] + \mathbf{w}(v) & \text{otherwise.} \end{cases} \tag{7.3}$$

To prove formally that this formula holds, consider first the case when $v \notin S$. Then the families of sets $\widehat{S}$ considered in the definitions of $c[t, S]$ and of $c[t', S]$ are equal, which immediately implies that $c[t, S] = c[t', S]$.

Consider the case when $v \in S$, and let $\widehat{S}$ be a set for which the maximum is attained in the definition of $c[t, S]$. Then it follows that $\widehat{S} \setminus \{v\}$ is one of the sets considered in the definition of $c[t', S \setminus \{v\}]$, which implies that $c[t', S \setminus \{v\}] \geq \mathbf{w}(\widehat{S} \setminus \{v\}) = \mathbf{w}(\widehat{S}) - \mathbf{w}(v) = c[t, S] - \mathbf{w}(v)$. Consequently,

$$c[t, S] \leq c[t', S \setminus \{v\}] + \mathbf{w}(v). \tag{7.4}$$

On the other hand, let $\widehat{S}'$ be a set for which the maximum is attained in the definition of $c[t', S \setminus \{v\}]$. Since we assumed that S is independent, we have that v does not have any neighbor in $S \setminus \{v\} = \widehat{S}' \cap X_{t'}$. Moreover, by Lemma 7.3, v does not have any neighbors in $V_{t'} \setminus X_{t'}$, which is a superset of $\widehat{S}' \setminus X_{t'}$. We conclude that v does not have any neighbors in $\widehat{S}'$, which means that $\widehat{S}' \cup \{v\}$ is an independent set. Since this set intersects with X_t exactly at S, it is considered in the definition of $c[t, S]$ and we have that

$$c[t, S] \geq \mathbf{w}(\widehat{S}' \cup \{v\}) = \mathbf{w}(\widehat{S}') + \mathbf{w}(v) = c[t', S \setminus \{v\}] + \mathbf{w}(v). \tag{7.5}$$

Concluding, (7.4) and (7.5) together prove (7.3) for the case $v \in S$.

Forget node. Suppose t is a forget node with child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Let S be any subset of X_t; again we assume that S is independent, since otherwise we put $c[t, S] = -\infty$. We claim that the following formula holds:

$$c[t, S] = \max\Big\{c[t', S], c[t', S \cup \{w\}]\Big\}. \tag{7.6}$$

We now give a formal proof of this formula. Let $\widehat{S}$ be a set for which the maximum is attained in the definition of $c[t, S]$. If $w \notin \widehat{S}$, then $\widehat{S}$ is one of the sets considered in the definition of $c[t', S]$, and hence $c[t', S] \geq \mathbf{w}(\widehat{S}) = c[t, S]$. However, if $w \in \widehat{S}$ then $\widehat{S}$ is one of the sets considered in the definition of $c[t', S \cup \{w\}]$, and then $c[t', S \cup \{w\}] \geq \mathbf{w}(\widehat{S}) = c[t, S]$. As exactly one of these alternatives happens, we can infer that the following holds always:

$$c[t, S] \leq \max\Big\{c[t', S], c[t', S \cup \{w\}]\Big\}. \tag{7.7}$$

On the other hand, observe that each set that is considered in the definition of $c[t', S]$ is also considered in the definition of $c[t, S]$, and the same holds also for $c[t', S \cup \{w\}]$. This means that $c[t, S] \geq c[t', S]$ and $c[t, S] \geq c[t', S \cup \{w\}]$. These two inequalities prove that

$$c[t, S] \geq \max\Big\{c[t', S], c[t', S \cup \{w\}]\Big\}. \tag{7.8}$$

The combination of (7.7) and (7.8) proves that formula (7.6) indeed holds.

Join node. Finally, suppose that t is a join node with children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$. Let S be any subset of X_t; as before, we can assume that S is independent. The claimed recursive formula is as follows:

$$c[t, S] = c[t_1, S] + c[t_2, S] - \mathbf{w}(S). \tag{7.9}$$

We now prove formally that this formula holds. First take $\widehat{S}$ to be a set for which the maximum is attained in the definition of $c[t, S]$. Let $\widehat{S}_1 = \widehat{S} \cap V_{t_1}$ and $\widehat{S}_2 = \widehat{S} \cap V_{t_2}$. Observe that $\widehat{S}_1$ is independent and $\widehat{S}_1 \cap X_{t_1} = S$, so this set is considered in the definition of $c[t_1, S]$. Consequently $c[t_1, S] \geq \mathbf{w}(\widehat{S}_1)$, and analogously $c[t_2, S] \geq \mathbf{w}(\widehat{S}_2)$. Since $\widehat{S}_1 \cap \widehat{S}_2 = S$, we obtain the following:

$$c[t, S] = \mathbf{w}(\widehat{S}) = \mathbf{w}(\widehat{S}_1) + \mathbf{w}(\widehat{S}_2) - \mathbf{w}(S) \leq c[t_1, S] + c[t_2, S] - \mathbf{w}(S). \tag{7.10}$$

On the other hand, let $\widehat{S}'_1$ be a set for which the maximum is attained in the definition of $c[t_1, S]$, and similarly define $\widehat{S}'_2$ for $c[t_2, S]$. By Lemma 7.3 we have that there is no edge between vertices of $V_{t_1} \setminus X_t$ and $V_{t_2} \setminus X_t$, which implies that the set $\widehat{S}' := \widehat{S}'_1 \cup \widehat{S}'_2$ is independent. Moreover $\widehat{S}' \cap X_t = S$, which implies that $\widehat{S}'$ is one of the sets considered in the definition of $c[t, S]$. Consequently,

$$c[t, S] \geq \mathbf{w}(\widehat{S}') = \mathbf{w}(\widehat{S}'_1) + \mathbf{w}(\widehat{S}'_2) - \mathbf{w}(S) = c[t_1, S] + c[t_2, S] - \mathbf{w}(S). \tag{7.11}$$

From (7.10) and (7.11) we infer that (7.9) indeed holds.

This concludes the description and the proof of correctness of the recursive formulas for computing the values of $c[\cdot, \cdot]$. Let us now wrap up the whole algorithm and estimate the running time. Recall that we are working on a tree

decomposition of width at most k, which means that $|X_t| \leq k+1$ for every node t. Thus at node t we compute $2^{|X_t|} \leq 2^{k+1}$ values of $c[t,S]$. However, we have to be careful when estimating the time needed for computing each of these values.

Of course, we could just say that using the recursive formulas and any graph representation we can compute each value $c[t,S]$ in $n^{\mathcal{O}(1)}$ time, but then we would end up with an algorithm running in time $2^k \cdot n^{\mathcal{O}(1)}$. It is easy to see that all the operations needed to compute one value can be performed in $k^{\mathcal{O}(1)}$ time, apart from checking adjacency of a pair of vertices. We need it, for example, to verify that a set S is independent. However, a straightforward implementation of an adjacency check runs in $\mathcal{O}(n)$ time, which would add an additional $\mathcal{O}(n)$ factor to the running time of the algorithm. Nonetheless, as G is a graph of treewidth at most k, it is possible to construct a data structure in time $k^{\mathcal{O}(1)}n$ that allows performing adjacency queries in time $\mathcal{O}(k)$. The reader is asked to construct such a data structure in Exercise 7.16.

Wrapping up, for every node t it takes time $2^k \cdot k^{\mathcal{O}(1)}$ to compute all the values $c[t,S]$. Since we can assume that the number of nodes of the given tree decompositions is $\mathcal{O}(kn)$ (see Lemma 7.4), the total running time of the algorithm is $2^k \cdot k^{\mathcal{O}(1)} \cdot n$. Hence, we obtain the following theorem.

Theorem 7.5. *Let G be an n-vertex graph with weights on vertices given together with its tree decomposition of width at most k. Then the* Weighted Independent Set *problem in G is solvable in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

Again, using the standard technique of backlinks, i.e., memorizing for every cell of table $c[\cdot,\cdot]$ how its value was obtained, we can reconstruct the solution (i.e., an independent set with the maximum possible weight) within the same asymptotic running time. The same will hold also for all the other dynamic-programming algorithms given in this section.

Since a graph has a vertex cover of size at most ℓ if and only if it has an independent set of size at least $n-\ell$, we immediately obtain the following corollary.

Corollary 7.6. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve the* Vertex Cover *problem in G in time $2^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

The algorithm of Theorem 7.5 seems actually quite simple, so it is very natural to ask if its running time could be improved. We will see in Chapter 14, Theorem 14.38, that under some reasonable assumptions the upper bound of Theorem 7.5 is tight.

The reader might wonder now why we gave all the formal details of this dynamic-programming algorithm, even though most of them were straightforward. Our point is that despite the naturalness of many dynamic-programming algorithms on tree decompositions, proving their correctness formally needs a lot of attention and care with regard to the details. We tried to show which

implications need to be given in order to obtain a complete and formal proof, and what kind of arguments can be used along the way.

> Most often, proving correctness boils down to showing two inequalities for each type of node: one relating an optimum solution for the node to some solutions for its children, and the second showing the reverse correspondence. It is usual that a precise definition of a state of the dynamic program together with function c denoting its value already suggests natural recursive formulas for c. Proving correctness of these formulas is usually a straightforward and tedious task, even though it can be technically challenging.

For this reason, in the next dynamic-programming routines we usually only give a precise definition of a state, function c, and the recursive formulas for computing c. We resort to presenting a short rationale for why the formulas are correct, leaving the full double-implication proof to the reader. We suggest that the reader always performs such double-implication proofs for his or her dynamic-programming routines, even though all the formal details might not always appear in the final write-up of the algorithm. Performing such formal proofs can highlight the key arguments needed in the correctness proof, and is the best way of uncovering possible problems and mistakes.

Another issue that could be raised is why we should not go further and also estimate the polynomial factor depending on k in the running time of the algorithm, which is now stated just as $k^{\mathcal{O}(1)}$. We refrain from this, since the actual value of this factor depends on the following:

- How fast we can implement all the low-level details of computing formulas for $c[t, S]$, e.g., iteration through subsets of vertices of a bag. This in particular depends on how we organize the structure of G and $\mathcal{T}$ in the memory.
- How fast we can access the exponential-size memory needed for storing the dynamic-programming table $c[\cdot, \cdot]$.

Answers to these questions depend on low-level details of the implementation and on the precise definition of the assumed model of RAM computations. While optimization of the $k^{\mathcal{O}(1)}$ factor is an important and a nontrivial problem in practice, this issue is beyond the theoretical scope of this book. For this reason, we adopt a pragmatic policy of not stating such polynomial factors explicitly for dynamic-programming routines. In the bibliographic notes we provide some further references to the literature that considers this issue.

7.3.2 Dominating Set

Our next example is the Dominating Set problem. Recall that a set of vertices D is a dominating set in graph G if $V(G) = N[D]$. The goal is to provide a dynamic-programming algorithm on a tree decomposition that determines the minimum possible size of a dominating set of the input graph G. Again, n will denote the number of vertices of the input graph G, and k is an upper bound on the width of the given tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G.

In this algorithm we will use a refined variant of nice tree decompositions. In the algorithm of the previous section, for every node t we were essentially interested in the best partial solutions in the graph $G[V_t]$. Thus, whenever a vertex v was introduced in some node t, we simultaneously introduced also all the edges connecting it to other vertices of X_t. We will now add a new type of a node called an *introduce edge node*. This modification enables us to add edges one by one, which often helps in simplifying the description of the algorithm.

Formally, in this extended version of a nice tree decomposition we have both introduce vertex nodes and introduce edge nodes. Introduce vertex nodes correspond to introduce nodes in the standard sense, while introduce edge nodes are defined as follows.

- **Introduce edge node**: a node t, labeled with an edge $uv \in E(G)$ such that $u, v \in X_t$, and with exactly one child t' such that $X_t = X_{t'}$. We say that edge uv is *introduced* at t.

We additionally require that every edge of $E(G)$ is introduced exactly once in the whole decomposition. Leaf nodes, forget nodes and join nodes are defined just as previously. Given a standard nice tree decomposition, it can be easily transformed to this variant as follows. Observe that condition (T3) implies that, in a nice tree decomposition, for every vertex $v \in V(G)$, there exists a unique highest node $t(v)$ such that $v \in X_{t(v)}$; moreover, the parent of $X_{t(v)}$ is a forget node that forgets v. Consider an edge $uv \in E(G)$, and observe that (T2) implies that $t(v)$ is an ancestor of $t(u)$ or $t(u)$ is an ancestor of $t(v)$. Without loss of generality assume the former, and observe that we may insert the introduce edge bag that introduces uv between $t(u)$ and its parent (which forgets u). This transformation, for every edge $uv \in E(G)$, can be easily implemented in time $k^{\mathcal{O}(1)}n$ by a single top-down transversal of the tree decomposition. Moreover, the obtained tree decomposition still has $\mathcal{O}(kn)$ nodes, since a graph of treewidth at most k has at most kn edges (see Exercise 7.15).

With each node t of the tree decomposition we associate a subgraph G_t of G defined as follows:

$$G_t = \Big(V_t, E_t = \{e \ : \ e \text{ is introduced in the subtree rooted at } t\}\Big).$$

While in the previous section the subproblems for t were defined on the graph $G[V_t]$, now we will define subproblems for the graph G_t.

We are ready to define the subproblems formally. For Weighted Independent Set, we were computing partial solutions according to how a maximum weight independent set intersects a bag of the tree decomposition. For domination the situation is more complicated. Here we have to distinguish not only if a vertex is in the dominating set or not, but also if it is dominated. A *coloring* of bag X_t is a mapping $f\colon X_t \to \{0, \hat{0}, 1\}$ assigning three different colors to vertices of the bag.

- **Black**, represented by 1. The meaning is that all black vertices have to be contained in the partial solution in G_t.
- **White**, represented by 0. The meaning is that all white vertices are not contained in the partial solution and must be dominated by it.
- **Grey**, represented by $\hat{0}$. The meaning is that all grey vertices are not contained in the partial solution, but do not have to be dominated by it.

The reason why we need to distinguish between white and grey vertices is that some vertices of a bag can be dominated by vertices or via edges which are not introduced so far. Therefore, we also need to consider subproblems where some vertices of the bag are not required to be dominated, since such subproblems can be essential for constructing the optimum solution. Let us stress the fact that we do not forbid grey vertices to be dominated — we just do not care whether they are dominated or not.

For a node t, there are $3^{|X_t|}$ colorings of X_t; these colorings form the space of states at node t. For a coloring f of X_t, we denote by $c[t, f]$ the minimum size of a set $D \subseteq V_t$ such that

- $D \cap X_t = f^{-1}(1)$, which is the set of vertices of X_t colored black.
- Every vertex of $V_t \setminus f^{-1}(\hat{0})$ either is in D or is adjacent in G_t to a vertex of D. That is, D dominates all vertices of V_t in graph G_t, except possibly some grey vertices in X_t.

We call such a set D a *minimum compatible set* for t and f. If no minimum compatible set for t and f exists, we put $c[t, f] = +\infty$. Note that the size of a minimum dominating set in G is exactly the value of $c[r, \emptyset]$ where r is the root of the tree decomposition. This is because we have $G = G_r$ and $X_r = \emptyset$, which means that for X_r we have only one coloring: the empty function.

It will be convenient to use the following notation. For a subset $X \subseteq V(G)$, consider a coloring $f : X \to \{0, \hat{0}, 1\}$. For a vertex $v \in V(G)$ and a color $\alpha \in \{0, \hat{0}, 1\}$ we define a new coloring $f_{v\to\alpha} : X \cup \{v\} \to \{0, \hat{0}, 1\}$ as follows:

$$f_{v\to\alpha}(x) = \begin{cases} f(x) & \text{when } x \neq v, \\ \alpha & \text{when } x = v. \end{cases}$$

For a coloring f of X and $Y \subseteq X$, we use $f|_Y$ to denote the restriction of f to Y.

We now proceed to present the recursive formulas for the values of c. As we mentioned in the previous section, we give only the formulas and short arguments for their correctness, leaving the full proof to the reader.

Leaf node. For a leaf node t we have that $X_t = \emptyset$. Hence there is only one, empty coloring, and we have $c[t, \emptyset] = 0$.

Introduce vertex node. Let t be an introduce node with a child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Since this node does not introduce edges to G_t, the computation will be very simple — v is isolated in G_t. Hence, we just need to be sure that we do not introduce an isolated white vertex, since then we have $c[t, f] = +\infty$. That is, for every coloring f of X_t we can put

$$c[t, f] = \begin{cases} +\infty & \text{when } f(v) = 0, \\ c[t', f|_{X_{t'}}] & \text{when } f(v) = \hat{0}, \\ 1 + c[t', f|_{X_{t'}}] & \text{when } f(v) = 1. \end{cases}$$

Introduce edge node. Let t be an introduce edge node labeled with an edge uv and let t' be the child of t. Let f be a coloring of X_t. Then sets D compatible for t and f should be almost exactly the sets that are compatible for t' and f, apart from the fact that the edge uv can additionally help in domination. That is, if f colors u black and v white, then when taking the precomputed solution for t' we can relax the color of v from white to grey — in the solution for t' we do not need to require any domination constraint on v, since v will get dominated by u anyways. The same conclusion can be drawn when u is colored white and v is colored black. Therefore, we have the following formulas:

$$c[t, f] = \begin{cases} c[t', f_{v \to \hat{0}}] & \text{when } (f(u), f(v)) = (1, 0), \\ c[t', f_{u \to \hat{0}}] & \text{when } (f(u), f(v)) = (0, 1), \\ c[t', f] & \text{otherwise.} \end{cases}$$

Forget node. Let t be a forget node with a child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Note that the definition of compatible sets for t and f requires that vertex w be dominated, so every set D compatible for t and f is also compatible for t' and $f_{w \to 1}$ (if $w \in D$) or $f_{w \to 0}$ (if $w \notin D$). On the other hand, every set compatible for t' and any of these two colorings is also compatible for t and f. This justifies the following recursive formula:

$$c[t, f] = \min \Big\{ c[t', f_{w \to 1}], c[t', f_{w \to 0}] \Big\}.$$

Join node. Let t be a join node with children t_1 and t_2. Recall that $X_t = X_{t_1} = X_{t_2}$. We say that colorings f_1 of X_{t_1} and f_2 of X_{t_2} are *consistent* with a coloring f of X_t if for every $v \in X_t$ the following conditions hold

(*i*) $f(v) = 1$ if and only if $f_1(v) = f_2(v) = 1$,
(*ii*) $f(v) = 0$ if and only if $(f_1(v), f_2(v)) \in \{(\hat{0}, 0), (0, \hat{0})\}$,
(*iii*) $f(v) = \hat{0}$ if and only if $f_1(v) = f_2(v) = \hat{0}$.

On one hand, if D is a compatible set for f and t, then $D_1 := D \cap V_{t_1}$ and $D_2 := D \cap V_{t_2}$ are compatible sets for t_1 and f_1, and t_2 and f_2, for some colorings f_1, f_2 that are consistent with f. Namely, for every vertex v that is white in f we make it white either in f_1 or in f_2, depending on whether it is dominated by D_1 in G_{t_1} or by D_2 in G_{t_2} (if v is dominated by both D_1 and D_2, then both options are correct). On the other hand, if D_1 is compatible for t_1 and f_1 and D_2 is compatible for t_2 and f_2, for some colorings f_1, f_2 that are consistent with f, then it is easy to see that $D := D_1 \cup D_2$ is compatible for t and f. Since for such D_1, D_2 we have that $D \cap X_t = D_1 \cap X_{t_1} = D_2 \cap X_{t_2} = f^{-1}(1)$, it follows that $|D| = |D_1| + |D_2| - |f^{-1}(1)|$. Consequently, we can infer the following recursive formula:

$$c[t, f] = \min_{f_1, f_2} \Big\{ c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(1)| \Big\}, \tag{7.12}$$

where the minimum is taken over all colorings f_1, f_2 consistent with f.

This finishes the description of the recursive formulas for the values of c. Let us analyze the running time of the algorithm. Clearly, the time needed to process each leaf node, introduce vertex/edge node or forget node is $3^k \cdot k^{\mathcal{O}(1)}$, providing that we again use the data structure for adjacency queries of Exercise 7.16. However, computing the values of c in a join node is more time consuming. The computation can be implemented as follows. Note that if a pair f_1, f_2 is consistent with f, then for every $v \in X_t$ we have

$$(f(v), f_1(v), f_2(v)) \in \{(1, 1, 1), (0, 0, \hat{0}), (0, \hat{0}, 0), (\hat{0}, \hat{0}, \hat{0})\}.$$

It follows that there are exactly $4^{|X_t|}$ triples of colorings (f, f_1, f_2) such that f_1 and f_2 are consistent with f, since for every vertex v we have four possibilities for $(f(v), f_1(v), f_2(v))$. We iterate through all these triples, and for each triple (f, f_1, f_2) we include the contribution from f_1, f_2 to the value of $c[t, f]$ according to (7.12). In other words, we first put $c[t, f] = +\infty$ for all colorings f of X_t, and then for every considered triple (f, f_1, f_2) we replace the current value of $c[t, f]$ with $c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(1)|$ in case the latter value is smaller. As $|X_t| \leq k + 1$, it follows that the algorithm spends $4^k \cdot k^{\mathcal{O}(1)}$ time for every join node. Since we assume that the number of nodes in a nice tree decomposition is $\mathcal{O}(kn)$ (see Exercise 7.15), we derive the following theorem.

Theorem 7.7. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve the* Dominating Set *problem in G in time $4^k \cdot k^{\mathcal{O}(1)} \cdot n$.*

In Chapter 11, we show how one can use more clever techniques in the computations for the join node in order to reduce the exponential dependence on the treewidth from 4^k to 3^k.

7.3.3 Steiner Tree

Our last example is the Steiner Tree problem. We are given an undirected graph G and a set of vertices $K \subseteq V(G)$, called *terminals*. The goal is to find a subtree H of G of the minimum possible size (that is, with the minimum possible number of edges) that connects all the terminals. Again, assume that $n = |V(G)|$ and that we are given a nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k. We will use the same variant of a nice tree decomposition as in Section 7.3.2, that is, with introduce edge nodes.

While the exponential dependence on the treewidth in the algorithms we discussed so far is single-exponential, for Steiner Tree the situation will be different. In what follows, we give an algorithm with running time $k^{\mathcal{O}(k)} \cdot n$. While a single-exponential algorithm for Steiner Tree actually exists, it requires more advanced techniques that will be discussed in Chapter 11.

In order to make the description of the algorithm simpler, we will make one small adjustment to the given decomposition. Let us pick an arbitrary terminal $u^\star \in K$ and let us add it to every bag of the decomposition $\mathcal{T}$. Then the width of $\mathcal{T}$ increases by at most 1, and bags at the root and at all the leaves are equal to $\{u^\star\}$. The idea behind this simple tweak is to make sure that every bag of $\mathcal{T}$ contains at least one terminal. This will be helpful in the definition of the state of the dynamic program.

Let H be a Steiner tree connecting K and let t be a node of $\mathcal{T}$. The part of H contained in G_t is a forest F with several connected components, see Fig. 7.3. Note that this part is never empty, because X_t contains at least one terminal. Observe that, since H is connected and X_t contains a terminal, each connected component of F intersects X_t. Moreover, every terminal from $K \cap V_t$ should belong to some connected component of F. We try to encode all this information by keeping, for each subset $X \subseteq X_t$ and each partition $\mathcal{P}$ of X, the minimum size of a forest F in G_t such that

(a) $K \cap V_t \subseteq V(F)$, i.e., F spans all terminals from V_t,
(b) $V(F) \cap X_t = X$, and
(c) the intersections of X_t with vertex sets of connected components of F form exactly the partition $\mathcal{P}$ of X.

When we introduce a new vertex or join partial solution (at join nodes), the connected components of partial solutions could merge and thus we need to keep track of the updated partition into connected components.

More precisely, we introduce the following function. For a bag X_t, a set $X \subseteq X_t$ (a set of vertices touched by a Steiner tree), and a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X, the value $c[t, X, \mathcal{P}]$ is the minimum possible number of edges of a forest F in G_t such that:

- F has exactly q connected components that can be ordered as $C_1, \ldots, C_q$ so that $P_s = V(C_s) \cap X_t$ for each $s \in \{1, \ldots, q\}$. Thus the partition $\mathcal{P}$ corresponds to connected components of F.

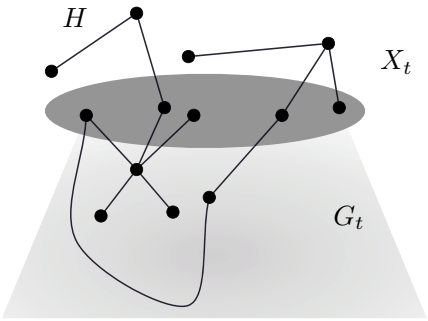


Fig. 7.3: Steiner tree H intersecting bag X_t and graph G_t

- $X_t \cap V(F) = X$. That is, vertices of $X_t \setminus X$ are untouched by F.
- Every terminal vertex from $K \cap V_t$ is in $V(F)$.

A forest F conforming to this definition will be called *compatible* for $(t, X, \mathcal{P})$. If no compatible forest F exists, we put $c[t, X, \mathcal{P}] = +\infty$.

Note that the size of an optimum Steiner tree is exactly $c[r, \{u^\star\}, \{\{u^\star\}\}]$, where r is the root of the decomposition $\mathcal{T}$. This is because we have that $X_r = \{u^\star\}$. We now provide recursive formulas to compute the values of c.

Leaf node. If t is a leaf node, then $X_t = \{u^\star\}$. Since $u^\star \in K$, we have $c[t, \emptyset, \emptyset] = +\infty$ and $c[t, \{u^\star\}, \{\{u^\star\}\}] = 0$.

Introduce vertex node. Suppose that t is an introduce vertex node with a child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. Recall that we have *not introduced* any edges adjacent to v so far, so v is isolated in G_t. Hence, for every set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X we do the following. If v is a terminal, then it has to be in X. Moreover, if v is in X, then $\{v\}$ should be a block of $\mathcal{P}$, that is, v should be in its own connected component. If any of these conditions is not satisfied, we put $c[t, X, \mathcal{P}] = +\infty$. Otherwise we have the following recursive formula:

$$c[t, X, \mathcal{P}] = \begin{cases} c[t', X \setminus \{v\}, \mathcal{P} \setminus \{\{v\}\}] & \text{if } v \in X, \\ c[t', X, \mathcal{P}] & \text{otherwise.} \end{cases}$$

Introduce edge node. Suppose that t is an introduce edge node that introduces an edge uv, and let t' be the child of t. For every set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X we consider three cases. If $u \notin X$ or $v \notin X$, then we cannot include uv into the tree under construction, as one of its endpoints has been already determined not to be touched by the tree. Hence in this case $c[t, X, \mathcal{P}] = c[t', X, \mathcal{P}]$. The same happens if u and v are both in X, but are not in the same block of $\mathcal{P}$. Assume then that u and v are

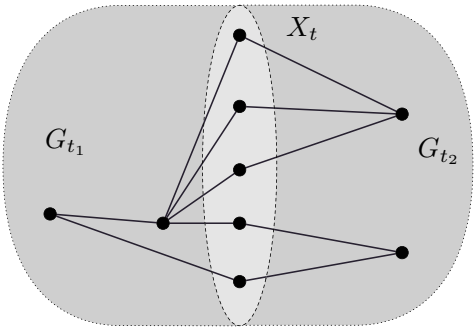


Fig. 7.4: Undesirable merge of partial solutions in a join node

both in X and they actually are in the same block of $\mathcal{P}$. Then the edge uv either has been picked to the solution, or has not been picked. If not, then we should look at the same partition $\mathcal{P}$ at t', and otherwise the block of u and v in $\mathcal{P}$ should have been obtained from merging two smaller blocks, one containing u and the second containing v. Hence

$$c[t, X, \mathcal{P}] = \min \Big\{ \min_{\mathcal{P}'} c[t', X, \mathcal{P}'] + 1, c[t', X, \mathcal{P}] \Big\},$$

where in the inner minimum we consider all partitions $\mathcal{P}'$ of X in which u and v are in separate blocks (as otherwise adding uv would create a cycle) such that after merging the blocks of u and v we obtain the partition $\mathcal{P}$.

Forget node. Suppose that t is a forget node with a child t' such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_{t'}$. Consider any set $X \subseteq X_t$ and partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X. The solution for X and $\mathcal{P}$ might either use the vertex w, in which case it should be added to one of existing blocks of $\mathcal{P}$, or not use it, in which case we should simply look on the same partition of the same X. Hence

$$c[t, X, \mathcal{P}] = \min \Big\{ \min_{\mathcal{P}'} c[t', X \cup \{w\}, \mathcal{P}'], c[t', X, \mathcal{P}] \Big\},$$

where the inner minimum is taken over all partitions $\mathcal{P}'$ of $X \cup \{w\}$ that are obtained from $\mathcal{P}$ by adding w to one of the existing blocks.

Join node. Suppose t is a join node with children t_1 and t_2. Recall that then $X_t = X_{t_1} = X_{t_2}$. In essence, in the computation for t we need to encode merging two partial solutions: one originating from G_{t_1} and the second originating from G_{t_2}. When merging two partial solutions, however, we have to be careful because such a merge can create cycles, see Fig. 7.4.

To avoid cycles while merging, we introduce an auxiliary structure.[1] For a partition $\mathcal{P}$ of X let $G_{\mathcal{P}}$ be a forest with a vertex set X, such that the set of connected components in $G_{\mathcal{P}}$ corresponds exactly to $\mathcal{P}$. In other words, for each block of $\mathcal{P}$ there is a tree in $G_{\mathcal{P}}$ with the same vertex set. We say that a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_q\}$ of X is an *acyclic merge* of partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ if the merge of two forests $G_{\mathcal{P}_1}$ and $G_{\mathcal{P}_2}$ (treated as a multigraph) is a forest whose family of connected components is exactly $\mathcal{P}$.

Thus we have the following formula:

$$c[t, X, \mathcal{P}] = \min_{\mathcal{P}_1, \mathcal{P}_2} c[t_1, X, \mathcal{P}_1] + c[t_2, X, \mathcal{P}_2],$$

where in the minimum we consider all pairs of partitions $\mathcal{P}_1, \mathcal{P}_2$ such that $\mathcal{P}$ is an acyclic merge of them.

This concludes the description of the recursive formulas for the values of c. We proceed to estimate the running time. Recall that every bag of the decomposition has size at most $k+2$. Hence, the number of states per node is at most $2^{k+2} \cdot (k+2)^{k+2} = k^{\mathcal{O}(k)}$, since for a node t there are $2^{|X_t|}$ subsets $X \subseteq X_t$ and at most $|X|^{|X|}$ partitions of X. The computation of a value for every state requires considering at most all the pairs of states for some other nodes, which means that each value can be computed in time $(k^{\mathcal{O}(k)})^2 = k^{\mathcal{O}(k)}$. Thus, up to a factor polynomial in k, which is anyhow dominated by the $\mathcal{O}$-notation in the exponent, for every node the running time of computing the values of c is $k^{\mathcal{O}(k)}$. We conclude with the following theorem.

Theorem 7.8. *Let G be an n-vertex graph, let $K \subseteq V(G)$ be a given set of terminals, and assume that G is given together with its tree decomposition of width at most k. Then one can find the minimum possible number of edges of a Steiner tree connecting K in time $k^{\mathcal{O}(k)} \cdot n$.*

Algorithms similar to the dynamic programming of Theorem 7.8 can be used to solve many problems in time $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$. Essentially, such a running time appears for problems with connectivity requirements, since then it is natural to keep in the dynamic programming state a partition of a subset of the bag. Since the number of such partitions is at most $k^{\mathcal{O}(k)}$, this factor appears naturally in the running time.

For convenience, we now state formally the most prominent problems that can be solved in *single-exponential time* when parameterized by treewidth (i.e., in time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$), and those that can be solved in *slightly super-exponential time* (i.e., $k^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$). We leave designing the remaining algorithms from the following theorems as two exercises: Exercise 7.18 and Exercise 7.19.

[1] One could avoid the cycle detection by relaxing the definition of $c[t, X, \mathcal{P}]$ and dropping the assumption that F is a forest. However, as in other problems, like Feedback Vertex Set, cycle checking is essential, we show a common solution here.

Let us recall that the MaxCut problem asks for a partition of $V(G)$ into sets A and B such that the number of edges between A and B is maximized. The q-Coloring problem asks whether G can be properly colored using q colors, while in Chromatic Number the question is to find the minimum possible number of colors needed to properly color G. Longest Path and Longest Cycle ask for the existence of a path/cycle on at least ℓ vertices in G, for a given integer ℓ. Similarly, in Cycle Packing the question is whether one can find ℓ vertex-disjoint cycles in G. Problems Connected Vertex Cover, Connected Dominating Set, Connected Feedback Vertex Set differ from their standard (non-connected) variants by additionally requiring that the solution vertex cover/dominating set/feedback vertex set induces a connected graph in G.

Theorem 7.9. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then in G one can solve*

- Vertex Cover *and* Independent Set *in time* $2^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- Dominating Set *in time* $4^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- Odd Cycle Transversal *in time* $3^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- MaxCut *in time* $2^k \cdot k^{\mathcal{O}(1)} \cdot n$,
- q-Coloring *in time* $q^k \cdot k^{\mathcal{O}(1)} \cdot n$.

Theorem 7.10. *Let G be an n-vertex graph given together with its tree decomposition of width at most k. Then one can solve each of the following problems in G in time $k^{\mathcal{O}(k)} \cdot n$:*

- Steiner Tree,
- Feedback Vertex Set,
- Hamiltonian Path *and* Longest Path,
- Hamiltonian Cycle *and* Longest Cycle,
- Chromatic Number,
- Cycle Packing,
- Connected Vertex Cover,
- Connected Dominating Set,
- Connected Feedback Vertex Set.

Let us also note that the dependence on k for many of the problems listed in Theorem 7.10 will be improved to single-exponential in Chapter 11. Also, the running time of the algorithm for Dominating Set will be improved from $4^k \cdot k^{\mathcal{O}(1)} \cdot n$ to $3^k \cdot k^{\mathcal{O}(1)} \cdot n$.

As the reader probably observed, dynamic-programming algorithms on graphs of bounded treewidth are very similar to each other.

The main challenge for most of the problems is to understand what information to store at nodes of the tree decomposition. Obtaining formulas for forget, introduce and join nodes can be a tedious task, but is usually straightforward once a precise definition of a state is established.

It is also worth giving an example of a problem which cannot be solved efficiently on graphs of small treewidth. In the Steiner Forest problem we are given a graph G and a set of pairs $(s_1, t_1), \ldots, (s_p, t_p)$. The task is to find a minimum subgraph F of G such that each of the pairs is connected in F. Steiner Tree is a special case of Steiner Forest with $s_1 = \cdots = s_p$. The intuition behind why a standard dynamic programming approach on graphs of constant treewidth does not work for this problem is as follows. Suppose that we have a set of vertices $S = \{s_i\}_{1 \leq i \leq p}$ separated from vertices $T = \{t_i\}_{1 \leq i \leq p}$ by a bag of tree decomposition of size 2. Since all paths from S to T must pass through the bag of size 2, the final solution F contains at most two connected components. Any partial solution divides the vertices of S into two parts based on which vertex in the separating bag they are connected to. Thus it seems that to keep track of all partial solutions for the problem, we have to compute all possible ways the set S can be partitioned into two subsets corresponding to connected components of F, which is 2^p. Since p does not depend on the treewidth of G, we are in trouble. In fact, this intuition is supported by the fact that Steiner Forest is NP-hard on graphs of treewidth at most 3, see [26]. In Section 13.6, we also give examples of problems which are W[1]-hard parameterized by the treewidth of the input graph.

Finding faster dynamic-programming strategies can be an interesting challenge and we will discuss several nontrivial examples of such algorithms in Chapter 11. However, if one is not particularly interested in obtaining the best possible running time, then there exist meta-techniques for designing dynamic-programming algorithms on tree decompositions. These tools we discuss in the next section.

7.4 Treewidth and monadic second-order logic

As we have seen in the previous sections, many optimization problems are fixed-parameter tractable when parameterized by the treewidth. Algorithms for this parameterization are in the vast majority derived using the paradigm of dynamic programming. One needs to understand how to succinctly represent necessary information about a subtree of the decomposition in a dynamic-programming table. If the size of this table turns out to be bounded by a function of the treewidth only, possibly with some polynomial

factors in the total graph size, then there is hope that a bottom-up dynamic-programming procedure can compute the complete answer to the problem. In other words, the final outcome can be obtained by consecutively assembling information about the behavior of the problem in larger and larger subtrees of the decomposition.

The standard approach to formalizing this concept is via *tree automata*. This leads to a deep theory linking logic on graphs, tree automata, and treewidth. In this section, we touch only the surface of this subject by highlighting the most important results, namely Courcelle's theorem and its optimization variant.

Intuitively, Courcelle's theorem provides a unified description of properties of a problem that make it amenable to dynamic programming over a tree decomposition. This description comes via a form of a logical formalism called *Monadic Second-Order logic on graphs*. Slightly more precisely, the theorem and its variants state that problems expressible in this formalism are always fixed-parameter tractable when parameterized by treewidth. Before we proceed to stating Courcelle's theorem formally, we need to understand first how Monadic Second-Order logic on graphs works.

7.4.1 Monadic second-order logic on graphs

$\mathbf{MSO}_2$ for dummies. The logic we are about to introduce is called $\mathbf{MSO}_2$. Instead of providing immediately the formal description of this logic, we first give an example of an $\mathbf{MSO}_2$ formula in order to work out the main concepts. Consider the following formula $\mathbf{conn}(X)$, which verifies that a subset X of vertices of a graph $G = (V, E)$ induces a connected subgraph.

$$\begin{aligned}\mathbf{conn}(X) = \quad & \forall_{Y \subseteq V}[(\exists_{u \in X}\, u \in Y \wedge \exists_{v \in X}\, v \notin Y) \\ & \Rightarrow (\exists_{e \in E}\, \exists_{u \in X}\, \exists_{v \in X}\, \mathbf{inc}(u, e) \wedge \mathbf{inc}(v, e) \wedge u \in Y \wedge v \notin Y)].\end{aligned}$$

Now, we rewrite this formula in English.

For every subset of vertices Y, if X contains both a vertex from Y and a vertex outside of Y, then there exists an edge e whose endpoints u, v both belong to X, but one of them is in Y and the other is outside of Y.

One can easily see that this condition is equivalent to the connectivity of $G[X]$: the vertex set of G cannot be partitioned into Y and $V(G) \setminus Y$ in such a manner that X is partitioned nontrivially and no edge of $G[X]$ crosses the partition.

As we see on this example, $\mathbf{MSO}_2$ is a formal language of expressing properties of graphs and objects inside these graphs, such as vertices, edges, or subsets of them. A formula φ of $\mathbf{MSO}_2$ is nothing else but a string over some mysterious symbols, which we shall decode in the next few paragraphs.

One may think that a formula defines a *program* that can be run on an input graph, similarly as, say, a C++ program can be run on some text input.[2] A C++ program is just a sequence of instructions following some syntax, and an $\mathbf{MSO}_2$ formula is just a sequence of symbols constructed using a specified set of rules. A C++ program can be run on multiple different inputs, and may provide different results of the computation. Similarly, an $\mathbf{MSO}_2$ formula may be evaluated in different graphs, and it can give different outcomes. More precisely, an $\mathbf{MSO}_2$ formula can be *true* in a graph, or *false*. The result of an application of a formula to a graph will be called the *evaluation* of the formula in the graph.

Similarly to C++ programs, $\mathbf{MSO}_2$ formulas have *variables* which represent different objects in the graph. Generally, we shall have four *types* of variables: variables for single vertices, for single edges, for subsets of vertices, and for subsets of edges; the last type was not used in formula $\mathbf{conn}(X)$. At each point of the process of evaluation of the formula, every variable is *evaluated* to some object of appropriate type.

Note that a formula can have "parameters": variables that are given from "outside", whose properties we verify in the graph. In the $\mathbf{conn}(X)$ example such a parameter is X, the vertex subset whose connectivity is being tested. Such variables will be called *free variables* of the formula. Note that in order to properly evaluate the formula in a graph, we need to be given the evaluation of these variables. Most often, we will assume that the input graph is *equipped* with evaluation of all the free variables of the considered $\mathbf{MSO}_2$ formula, which means that these evaluations are provided together with the graph.

If we already have some variables in the formula, we can test their mutual interaction. As we have seen in the $\mathbf{conn}(X)$ example, we can for instance check whether some vertex u belongs to some vertex subset Y ($u \in Y$), or whether an edge e is incident to a vertex u ($\mathbf{inc}(u, e)$). These checks can be combined using standard Boolean operators such as $\neg$ (negation, logical NOT), $\wedge$ (conjunction, logical AND), $\vee$ (disjunction, logical OR), $\Rightarrow$ (implication).

The crucial concept that makes $\mathbf{MSO}_2$ useful for expressing graph properties are *quantifiers*. They can be seen as counterparts of *loops* in standard programming languages. We have two types of quantifiers, $\forall$ and $\exists$. Each quantifier is applied to some *subformula* ψ, which in the programming language analogy is just a block of code bound by the loop. Moreover, every quantifier introduces a new variable over which it iterates. This variable can be then used in the subformula.

Quantifier $\forall$ is called the *universal quantifier*. Suppose we write a formula $\forall_{v \in V} \psi$, where ψ is some subformula that uses variable v. This formula should be then read as "For every vertex v in the graph, ψ holds." In other words, quantifier $\forall_{v \in V}$ iterates through all possible evaluations of variable v to a vertex of the graph, and for each of them it is checked whether ψ is indeed

[2] For a reader familiar with the paradigm of *functional programming* (languages like *Lisp* or *Haskell*), an analogy with any functional language would be more appropriate.

true. If this is the case for *every* evaluation of v, then the whole formula $\forall_{v \in V} \psi$ is true; otherwise it is false.

Quantifier $\exists$, called the *existential quantifier*, works sort of similarly. Formula $\exists_{v \in V} \psi$ should be read as "There exists a vertex v in the graph, such that ψ holds." This means that $\exists_{v \in V}$ iterates through all possible evaluations of variable v to a vertex of the graph, and verifies whether there is at least one for which ψ is true.

Of course, here we just showed examples of quantification over variables for single vertices, but we can also quantify over variables for single edges (e.g., $\forall_{e \in E}/\exists_{e \in E}$), vertex subsets (e.g., $\forall_{X \subseteq V}/\exists_{X \subseteq V}$), or edge subsets (e.g., $\forall_{C \subseteq E}/\exists_{C \subseteq E}$). Standard Boolean operators can be also used to combine larger formulas; see for instance our use of the implication in formula $\mathbf{conn}(X)$.

We hope that the reader already understands the basic idea of $\mathbf{MSO}_2$ as a (programming) language for expressing graph properties. We now proceed to explaining formally the syntax of $\mathbf{MSO}_2$ (how formulas can be constructed), and the semantics (how formulas are evaluated). Fortunately, they are much simpler than for C++.

Syntax and semantics of $\mathbf{MSO}_2$. Formulas of $\mathbf{MSO}_2$ can use four types of variables: for single vertices, single edges, subsets of vertices, and subsets of edges. The subscript 2 in $\mathbf{MSO}_2$ exactly signifies that quantification over edge subsets is also allowed. If we forbid this type of quantification, we arrive at a weaker logic $\mathbf{MSO}_1$. The vertex/edge subset variables are called *monadic* variables.

Every formula φ of $\mathbf{MSO}_2$ can have free variables, which often will be written in parentheses besides the formula. More precisely, whenever we write a formula of $\mathbf{MSO}_2$, we should always keep in mind what variables are assumed to be existent in the context in which this formula will be used. The sequence of these variables is called the *signature* over which the formula is written;[3] following our programming language analogy, this is the environment in which the formula is being defined. Then variables from the signature can be used in φ as free variables. The signature will be denoted by Σ. Note that for every variable in the signature we need to know what is its type.

In order to evaluate a formula φ over signature Σ in a graph G, we need to know how the variables of Σ are evaluated in G. By Σ^G we will denote the sequence of evaluations of variables from Σ. Evaluation of a single variable x will be denoted by x^G. Graph G and Σ^G together shall be called the *structure* in which φ is being evaluated. If φ is true in structure $\langle G, \Sigma^G \rangle$, then we shall denote it by

$$\langle G, \Sigma^G \rangle \models \varphi,$$

[3] A reader familiar with the foundations of logic in computer science will certainly see that we are slightly tweaking some definitions so that they fit to our small example. We do it in order to simplify the description of $\mathbf{MSO}_2$ while maintaining basic compliance with the general literature. In the bibliographic notes we provide pointers to more rigorous introductions to logic in computer science, where the notions of *signature* and *structure* are introduced properly.

which should be read as "Structure $\langle G, \Sigma^G \rangle$ is a model for φ."

Formulas of $\mathbf{MSO}_2$ are constructed inductively from smaller subformulas. We first describe the smallest building blocks, called *atomic formulas*.

- If $u \in \Sigma$ is a vertex (edge) variable and $X \in \Sigma$ is a vertex (edge) set variable, then we can write formula $u \in X$. The semantics is standard: the formula is true if and only if $u^G \in X^G$.
- If $u \in \Sigma$ is a vertex variable and $e \in \Sigma$ is an edge variable, then we can write formula $\mathbf{inc}(u, e)$. The semantics is that the formula is true if and only if u^G is an endpoint of e^G.
- For any two variables $x, y \in \Sigma$ of the same type, we can write formula $x = y$. This formula is true in the structure if and only if $x^G = y^G$.

Now that we know the basic building blocks, we can start to create larger formulas. As described before, we can use standard Boolean operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$ working as follows. Suppose that φ_1, φ_2 are two formulas over the same signature Σ. Then we can write the following formulas, also over Σ

- Formula $\neg\varphi_1$, where $\langle G, \Sigma^G \rangle \models \neg\varphi_1$ if and only if $\langle G, \Sigma^G \rangle \not\models \varphi_1$.
- Formula $\varphi_1 \wedge \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \wedge \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ and $\langle G, \Sigma^G \rangle \models \varphi_2$.
- Formula $\varphi_1 \vee \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \vee \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ or $\langle G, \Sigma^G \rangle \models \varphi_2$.
- Formula $\varphi_1 \Rightarrow \varphi_2$, where $\langle G, \Sigma^G \rangle \models \varphi_1 \Rightarrow \varphi_2$ if and only if $\langle G, \Sigma^G \rangle \models \varphi_1$ implies that $\langle G, \Sigma^G \rangle \models \varphi_2$.

Finally, we can use quantifiers. For concreteness, suppose we have a formula ψ over signature Σ' that contains some vertex variable v. Let $\Sigma = \Sigma' \setminus \{v\}$. Then we can write the following formulas over Σ:

- Formula $\varphi_\forall = \forall_{v \in V} \psi$. Then $\langle G, \Sigma^G \rangle \models \varphi_\forall$ if and only if *for every* vertex $v^G \in V(G)$, it holds that $\langle G, \Sigma^G, v^G \rangle \models \psi$.
- Formula $\varphi_\exists = \exists_{v \in V} \psi$. Then $\langle G, \Sigma^G \rangle \models \varphi_\exists$ if and only if there *exists* a vertex $v^G \in V(G)$ such that $\langle G, \Sigma^G, v^G \rangle \models \psi$.

Similarly, we can perform quantification over variables for single edges ($\forall_{e \in E}/\exists_{e \in E}$), vertex subsets ($\forall_{X \subseteq V}/\exists_{X \subseteq V}$), and edge subsets ($\forall_{C \subseteq E}/\exists_{C \subseteq E}$). The semantics is defined analogously.

Observe that in formula $\mathbf{conn}(X)$ we used a couple of notation "hacks" that simplified the formula, but were formally not compliant to the syntax described above. We namely allow some shorthands to streamline writing formulas. Firstly, we allow simple shortcuts in the quantifiers. For instance, $\exists_{v \in X} \psi$ is equivalent to $\exists_{v \in V} (v \in X) \wedge \psi$ and $\forall_{v \in X} \psi$ is equivalent to $\forall_{v \in V} (v \in X) \Rightarrow \psi$. We can also merge a number of similar quantifiers into one, e.g., $\exists_{X_1, X_2 \subseteq V}$ is the same as $\exists_{X_1 \subseteq V} \exists_{X_2 \subseteq V}$. Another construct that we can use is the subset relation $X \subseteq Y$: it can be expressed as $\forall_{v \in V} (v \in X) \Rightarrow (v \in Y)$, and similarly for edge subsets. We can also express the adjacency relation between two vertex variables: $\mathbf{adj}(u, v) = (u \neq v) \wedge (\exists_{e \in E} \mathbf{inc}(u, e) \wedge \mathbf{inc}(v, e))$.

Finally, we use $x \neq y$ for $\neg(x = y)$ and $x \notin X$ for $\neg(x \in X)$. The reader is encouraged to use his or her own shorthands whenever it is beneficial.

Examples. Let us now provide two more complicated examples of graph properties expressible in $\mathbf{MSO}_2$. We have already seen how to express that a subset of vertices induces a connected graph. Let us now look at 3-colorability. To express this property, we need to quantify the existence of three vertex subsets X_1, X_2, X_3 which form a partition of V, and where each of them is an independent set.

$$\begin{aligned}\textbf{3colorability} = \quad & \exists_{X_1,X_2,X_3 \subseteq V}\, \textbf{partition}(X_1, X_2, X_3) \wedge \\ & \textbf{indp}(X_1) \wedge \textbf{indp}(X_2) \wedge \textbf{indp}(X_3).\end{aligned}$$

Here, **partition** and **indp** are two auxiliary subformulas. Formula **partition** has three vertex subset variables X_1, X_2, X_3 and verifies that (X_1, X_2, X_3) is a partition of the vertex set V. Formula **indp** verifies that a given subset of vertices is independent.

$$\begin{aligned}\textbf{partition}(X_1, X_2, X_3) = \quad & \forall_{v \in V}\, [(v \in X_1 \wedge v \notin X_2 \wedge v \notin X_3) \\ & \vee (v \notin X_1 \wedge v \in X_2 \wedge v \notin X_3) \\ & \vee (v \notin X_1 \wedge v \notin X_2 \wedge v \in X_3)];\end{aligned}$$

$$\textbf{indp}(X) = \quad \forall_{u,v \in X} \neg \textbf{adj}(u, v).$$

Second, let us look at Hamiltonicity: we would like to write a formula that is true in a graph G if and only if G admits a Hamiltonian cycle. For this, let us quantify existentially a subset of edges C that is supposed to comprise the edges of the Hamiltonian cycle we look for. Then we need to verify that (a) C induces a connected graph, and (b) every vertex of V is adjacent to exactly two different edges of C.

$$\textbf{hamiltonicity} = \quad \exists_{C \subseteq E}\, \textbf{connE}(C) \wedge \forall_{v \in V}\, \textbf{deg2}(v, C).$$

Here, $\textbf{connE}(C)$ is an auxiliary formula that checks whether the graph (V, C) is connected (using similar ideas as for $\textbf{conn}(X)$), and $\textbf{deg2}(v, C)$ verifies that vertex v has exactly two adjacent edges belonging to C:

$$\begin{aligned}\textbf{connE}(C) = \quad & \forall_{Y \subseteq V} \quad [(\exists_{u \in V}\, u \in Y \wedge \exists_{v \in V}\, v \notin Y) \\ & \Rightarrow (\exists_{e \in C}\, \exists_{u \in Y}\, \exists_{v \notin Y}\, \textbf{inc}(u, e) \wedge \textbf{inc}(v, e))];\end{aligned}$$

$$\begin{aligned}\textbf{deg2}(v, C) = \quad & \exists_{e_1,e_2 \in C}\, [(e_1 \neq e_2) \wedge \textbf{inc}(v, e_1) \wedge \textbf{inc}(v, e_2) \wedge \\ & (\forall_{e_3 \in C}\, \textbf{inc}(v, e_3) \Rightarrow (e_1 = e_3 \vee e_2 = e_3))].\end{aligned}$$

7.4.2 Courcelle's theorem

In the following, for a formula φ by $||\varphi||$ we denote the length of the encoding of φ as a string.

Theorem 7.11 (Courcelle's theorem, [98]). *Assume that φ is a formula of $\mathbf{MSO}_2$ and G is an n-vertex graph equipped with evaluation of all the free variables of φ. Suppose, moreover, that a tree decomposition of G of width t is provided. Then there exists an algorithm that verifies whether φ is satisfied in G in time $f(||\varphi||, t) \cdot n$, for some computable function f.*

The proof of Courcelle's theorem is beyond the scope of this book, and we refer to other sources for a comprehensive presentation. As we will see later, the requirement that G be given together with its tree decomposition is not necessary, since an optimal tree decomposition of G can be computed within the same complexity bounds. Algorithms computing treewidth will be discussed in the next section and in the bibliographic notes.

Recall that in the previous section we constructed formulas **3colorability** and **hamiltonicity** that are satisfied in G if and only if G is 3-colorable or has a Hamiltonian cycle, respectively. If we now apply Courcelle's theorem to these constant-size formulas, we immediately obtain as a corollary that testing these two properties of graphs is fixed-parameter tractable when parameterized by treewidth.

Let us now focus on the Vertex Cover problem: given a graph G and integer k, we would like to verify whether G admits a vertex cover of size at most k. The natural way of expressing this property in $\mathbf{MSO}_2$ is to quantify existentially k vertex variables, representing vertices of the vertex cover, and then verify that every edge of G has one of the quantified vertices as an endpoint. However, observe that the length of such a formula depends linearly on k. This means that a direct application of Courcelle's theorem gives only an $f(k,t) \cdot n$ algorithm, and not an $f(t) \cdot n$ algorithm as was the case for the dynamic-programming routine of Corollary 7.6. Note that the existence of an $f(k,t) \cdot n$ algorithm is only a very weak conclusion, because as we already have seen in Section 3.1, even the simplest branching algorithm for Vertex Cover runs in time $\mathcal{O}(2^k \cdot (n+m))$.

Therefore, we would rather have the following optimization variant of the theorem. Formula φ has some free monadic (vertex or edge) variables $X_1, X_2, \ldots, X_p$, which correspond to the sets we seek in the graph. In the Vertex Cover example we would have one vertex subset variable X that represents the vertex cover. Formula φ verifies that the variables $X_1, X_2, \ldots, X_p$ satisfy all the requested properties; for instance, that X indeed covers every edge of the graph. Then the problem is to find an evaluation of variables $X_1, X_2, \ldots, X_p$ that minimizes/maximizes the value of some arithmetic expression $\alpha(|X_1|, |X_2|, \ldots, |X_p|)$ depending on the cardinalities of these sets, subject to $\varphi(X_1, X_2, \ldots, X_p)$ being true. We will focus on α

being an *affine function*, that is, $\alpha(x_1, x_2, \ldots, x_p) = a_0 + \sum_{i=1}^{p} a_i x_i$ for some $a_0, a_1, \ldots, a_p \in \mathbb{R}$.

The following theorem states that such an optimization version of Courcelle's theorem indeed holds.

Theorem 7.12 ([19]). *Let φ be an $\mathbf{MSO}_2$ formula with p free monadic variables $X_1, X_2, \ldots, X_p$, and let $\alpha(x_1, x_2, \ldots, x_p)$ be an affine function. Assume that we are given an n-vertex graph G together with its tree decomposition of width t, and suppose G is equipped with evaluation of all the free variables of φ apart from $X_1, X_2, \ldots, X_p$. Then there exists an algorithm that in $f(||\varphi||, t) \cdot n$ finds the minimum and maximum value of $\alpha(|X_1|, |X_2|, \ldots, |X_p|)$ for sets $X_1, X_2, \ldots, X_p$ for which $\varphi(X_1, X_2, \ldots, X_p)$ is true, where f is some computable function.*

To conclude our Vertex Cover example, we can now write a simple constant-length formula $\mathbf{vcover}(X)$ that verifies that X is a vertex cover of G: $\mathbf{vcover}(X) = \forall_{e \in E} \exists_{x \in X} \mathbf{inc}(x, e)$. Then we can apply Theorem 7.12 to **vcover** and $\alpha(|X|) = |X|$, and infer that finding the minimum cardinality of a vertex cover can be done in $f(t) \cdot n$ time, for some function f.

Note that both in Theorem 7.11 and in Theorem 7.12 we allow the formula to have some additional free variables, whose evaluation is provided together with the graph. This feature can be very useful whenever in the considered problem the graph comes together with some predefined objects, e.g., terminals in the Steiner Tree problem. Observe that we can easily write an $\mathbf{MSO}_2$ formula $\mathbf{Steiner}(K, F)$ for a vertex set variable K and edge set variable F, which is true if and only if the edges from F form a Steiner tree connecting K. Then we can apply Theorem 7.12 to minimize the cardinality of F subject to $\mathbf{Steiner}(K, F)$ being true, where the vertex subset K is given together with the input graph. Thus we can basically re-prove Theorem 7.8, however without any explicit bound on the running time.

To conclude, let us deliberate briefly on the function f in the bound on the running time of algorithms provided by Theorems 7.11 and 7.12. Unfortunately, it can be proved that this function has to be nonelementary; in simple words, it cannot by bounded by a folded c times exponential function for any constant c. Generally, the main reason why the running time must be so high is the possibility of having alternating sequences of quantifiers in the formula φ. Slightly more precisely, we can define the *quantifier alternation* of a formula φ to be the maximum length of an alternating sequence of nested quantifiers in φ, i.e., $\forall \exists \forall \exists \ldots$ (we omit some technicalities in this definition). Then it can be argued that formulas of quantifier alternation at most q give rise to algorithms with at most c-times exponential function f, where c depends linearly on q. However, tracing the exact bound on f even for simple formulas φ is generally very hard, and depends on the actual proof of the theorem that is used. This exact bound is also likely to be much higher than optimal. For this reason, Courcelle's theorem and its variants should be regarded primarily as classification tools, whereas design-

ing efficient dynamic-programming routines on tree decompositions requires "getting your hands dirty" and constructing the algorithm explicitly.

7.5 Graph searching, interval and chordal graphs

In this section we discuss alternative interpretations of treewidth and pathwidth, connected to the concept of *graph searching*, and to the classes of *interval* and *chordal* graphs. We also provide some tools for certifying that treewidth and pathwidth of a graph is at least/at most some value. While these results are not directly related to the algorithmic topics discussed in the book, they provide combinatorial intuition about the considered graph parameters that can be helpful when working with them.

Alternative characterizations of pathwidth. We start with the concept of graph searching. Suppose that G is a graph representing a network of tunnels where an agile and omniscient fugitive with unbounded speed is hiding. The network is being searched by a team of searchers, whose goal is to find the fugitive. A *move* of the search team can be one of the following:

- *Placement*: We can take a searcher from the pool of free searchers and place her on some vertex v.
- *Removal*: We can remove a searcher from a vertex v and put her back to the pool of free searchers.

Initially all the searchers are free. A *search program* is a sequence of moves of searchers. While the searchers perform their program, the fugitive, who is not visible to them, can move between the nodes of the network at unbounded speed. She cannot, however, pass through the vertices occupied by the searchers, and is caught if a searcher is placed on a vertex she currently occupies. Searchers win if they launch a search program that guarantees catching the fugitive. The fugitive wins if she can escape the searchers indefinitely.

Another interpretation of the same game is that searchers are cleaning a network of tunnels contaminated with a poisonous gas. Initially all the edges (tunnels) are contaminated, and an edge becomes clear if both its endpoints are occupied by searchers. The gas, however, spreads immediately between edges through vertices that are not occupied by any searcher. If a cleaned edge becomes contaminated in this manner, we say that it is *recontaminated*. The goal of the searchers is to clean the whole network using a search program.

These interpretations can be easily seen to be equivalent,[4] and they lead to the following definition of a graph parameter. The *node search number* of a graph G is the minimum number of searchers required to clean G from gas

[4] Strictly speaking, there is a boring corner case of nonempty edgeless graph, where no gas is present but one needs a searcher to visit one by one all vertices to catch the fugitive. In what follows we ignore this corner case, as all further results consider only a positive number of searchers.

or, equivalently, to catch an invisible fugitive in G. For concreteness, from now on we will work with the gas cleaning interpretation.

An important question in graph searching is the following: if k searchers can clean a graph, can they do it in a *monotone* way, i.e., in such a way that no recontamination occurs? The answer is given by the following theorem of LaPaugh; its proof lies beyond the scope of this book.

Theorem 7.13 ([315]). *For any graph G, if k searchers can clear G, then k searchers can clear G in such a manner that no edge gets recontaminated.*

Let us now proceed to the class of interval graphs. A graph G is an *interval graph* if and only if one can associate with each vertex $v \in V(G)$ a closed interval $I_v = [l_v, r_v]$, $l_v \leq r_v$, on the real line, such that for all $u, v \in V(G)$, $u \neq v$, we have that $uv \in E(G)$ if and only if $I_u \cap I_v \neq \emptyset$. The family of intervals $\mathcal{I} = \{I_v\}_{v \in V}$ is called an *interval representation* or an *interval model* of G. By applying simple transformations to the interval representation at hand, it is easy to see that every interval graph on n vertices has an interval representation in which the left endpoints are distinct integers $1, 2, \ldots, n$. We will call such a representation *canonical*.

Recall that a graph G' is a *supergraph* of a graph G if $V(G) \subseteq V(G')$ and $E(G) \subseteq E(G')$. We define the *interval width* of a graph G as the minimum over all interval supergraphs G' of G of the maximum clique size in G'. That is,

$$\text{interval-width}(G) = \min \{\omega(G') \ : \ G \subseteq G' \wedge G' \text{ is an interval graph}\}.$$

Here, $\omega(G')$ denotes the maximum size of a clique in G'. In other words, the interval width of G is at most k if and only if there is an interval supergraph of G with the maximum clique size at most k. Note that in this definition we can assume that $V(G') = V(G)$, since the class of interval graphs is closed under taking induced subgraphs.

We are ready to state equivalence of all the introduced interpretations of pathwidth.

Theorem 7.14. *For any graph G and any $k \geq 0$, the following conditions are equivalent:*

(i) *The node search number of G is at most $k+1$.*
(ii) *The interval width of G is at most $k+1$.*
(iii) *The pathwidth of G is at most k.*

Proof. $(i) \Rightarrow (ii)$. Without loss of generality, we can remove from G all isolated vertices. Assume that there exists a search program that cleans G using at most $k+1$ searchers. By Theorem 7.13, we can assume that this search is monotone, i.e., no edge becomes recontaminated. Suppose the program uses p moves; we will naturally index them with $1, 2, \ldots, p$. Let $x_1, x_2, \ldots, x_p$ be the sequence of vertices on which searchers are placed/from which searchers

are removed in consecutive moves. Since G has no isolated vertices, each vertex of G has to be occupied at some point (to clear the edges incident to it) and hence it appears in the sequence $x_1, x_2, \ldots, x_p$. Also, without loss of generality we assume that each placed searcher is eventually removed, since at the end we can always remove all the searchers.

We now claim that we can assume that for every vertex $v \in V(G)$, a searcher is placed on v exactly once and removed from v exactly once. Let us look at the first move j when an edge incident to v is cleaned. Note that it does not necessarily hold that $x_j = v$, but vertex v has to be occupied after the j-th move is performed. Let $j_1 \leq j$ be the latest move when a searcher is placed at v before or at move j; in particular $x_{j_1} = v$. Let $j_2 > j$ be the first move when a searcher is removed from v after move j; in particular $x_{j_2} = v$. Since j is the first move when an edge incident to v is cleaned, we have that before j_1 all the edges incident to v are contaminated. Therefore, we can remove from the search program any moves at vertex v that are before j_1, since they actually do not clean anything. On the other hand, observe that at move j_2 all the edges incident to v must be cleaned, since otherwise a recontamination would occur. These edges stay clean to the end of the search, by the monotonicity of our search program. Therefore, we can remove from the search program all the moves at v that occur after j_2, since they actually do not clean anything.

For every vertex v, we define $\ell(v)$ as the first move when a searcher is placed on v, and $r(v)$ as the move when a searcher is removed from v. Now with each vertex v we associate an interval $I_v = [\ell(v), r(v) - 1]$; note that integers contained in I_v are exactly moves after which v is occupied. The intersection graph G_I of intervals $\mathcal{I} = \{I_v\}_{v \in V}$ is, of course, an interval graph. This graph is also a supergraph of G for the following reason. Since every edge uv is cleared, there is always a move j_{uv} after which both u and v are occupied by searchers. Then $j_{uv} \in I_u \cap I_v$, which means that I_u and I_v have nonempty intersection. Finally, the maximum size of a clique in G_I is the maximum number of intervals intersecting in one point, which is at most the number of searchers used in the search.

$(ii) \Rightarrow (iii)$. Let G_I be an interval supergraph of G with the maximum clique size at most $k+1$, and let $\mathcal{I} = \{I_v\}_{v \in V}$ be the canonical representation of G_I. For $i \in \{1, \ldots, n\}$, we define X_i as the set of vertices v of G whose corresponding intervals I_v contain i. All intervals associated with vertices of X_i pairwise intersect at i, which means that X_i is a clique in G. Consequently $|X_i| \leq k + 1$. We claim that $(X_1, \ldots, X_n)$ is a path decomposition of G. Indeed, property (P1) holds trivially. For property (P2), because G_I is a supergraph of G, for every edge $uv \in E(G)$ we have $I_u \cap I_v \neq \emptyset$. Since left endpoints of intervals of $\mathcal{I}$ are $1, 2, \ldots, n$, there exists also some $i \in \{1, 2, \ldots, n\}$ that belongs to $I_u \cap I_v$. Consequently X_i contains both u and v. For property (P3), from the definition of sets X_i it follows that, for every vertex v, the sets containing v form an interval in $\{1, 2, \ldots, n\}$.

$(iii) \Rightarrow (i)$. Let $\mathcal{P} = (X_1, X_2, \ldots, X_r)$ be a path decomposition of G of width at most k. We define the following search program for $k+1$ searchers. First, we place searchers on X_1. Then, iteratively for $i = 2, 3, \ldots, r$, we move searchers from X_{i-1} to X_i by first removing all the searchers from $X_{i-1} \setminus X_i$, and then placing searchers on $X_i \setminus X_{i-1}$. Note that in this manner the searchers are all the time placed on the vertices of $X_{i-1} \cap X_i$. Moreover, since the sizes of bags X_i are upper bounded by $k+1$, we use at most $k+1$ searchers at a time. We now prove that this search program cleans the graph.

By property (P2), for every edge $uv \in E(G)$ there exists a bag X_i such that $\{u, v\} \subseteq X_i$. Hence, every edge gets cleaned at some point. It remains to argue that no recontamination happens during implementation of this search program. For the sake of contradiction, suppose that recontamination happens for the first time when a searcher is removed from some vertex $u \in X_{i-1} \setminus X_i$, for some $i \geq 2$. This means that u has to be incident to some edge uv that is contaminated at this point. Again by the property (P2), there exists some bag X_j such that $\{u, v\} \in X_j$. However, since $u \in X_{i-1}$ and $u \notin X_i$, then by property (P3) u can only appear in bags with indices $1, 2, \ldots, i-1$. Hence $j \leq i-1$, which means that edge uv has been actually already cleaned before, when searchers were placed on the whole bag X_j. Since we assumed that we consider the first time when a recontamination happens, we infer that edge uv must have remained clean up to this point, a contradiction. □

Alternative characterizations of treewidth. A similar set of equivalent characterizations is known for the treewidth. The role of interval graphs is now played by chordal graphs. Let us recall that a graph is *chordal* if it does not contain an induced cycle of length more than 3, i.e., every cycle of length more than 3 has a chord. Sometimes chordal graphs are called triangulated. It is easy to prove that interval graphs are chordal, see Exercise 7.27.

We define the *chordal width* of a graph G as the minimum over all chordal supergraphs G' of G of the maximum clique size in G'. That is,

$$\text{chordal-width}(G) = \min \{\omega(G') \ : \ G \subseteq G' \wedge G' \text{ is a chordal graph}\}.$$

As we will see soon, the chordal width is equal to treewidth (up to a ± 1 summand), but first we need to introduce the notion of graph searching related to the treewidth.

The rules of the search game for treewidth are very similar to the node searching game for pathwidth. Again, a team of searchers, called in this setting *cops*, tries to capture in a graph an agile and omniscient fugitive, called in this setting a *robber*. The main difference is that the cops actually know where the robber is localized. At the beginning of the game cops place themselves at some vertices, and then the robber chooses a vertex to start her escape. The game is played in rounds. Let us imagine that cops are equipped with helicopters. At the beginning of each round, some subset of cops take off

in their helicopters, declaring where they will land at the end of the round. While the cops are in the air, the robber may run freely in the graph; however, she cannot pass through vertices that are occupied by cops that are not airborne. After the robber runs to her new location, being always a vertex, the airborne cops land on pre-declared vertices. As before, the cops win if they have a procedure to capture the robber by landing on her location, and the robber wins if she can avoid cops indefinitely.

Intuitively, the main difference is that if cops are placed at some set $S \subseteq V(G)$, then they know in which connected component of $G - S$ the robber resides. Hence, they may concentrate the chase in this component. In the node search game the searchers lack this knowledge, and hence the search program must be oblivious to the location of the fugitive. It is easy to see that when cops do not see the robber, then this reduces to an equivalent variant of the node search game. However, if they see the robber, they can gain a lot. For example, two cops can catch a visible robber on any tree, but catching an invisible fugitive on an n-vertex tree can require $\log_3 n$ searchers; see Exercise 7.10.

Finally, let us introduce the third alternative characterization of treewidth, which is useful in showing lower bounds on this graph parameter. We say that two subsets A and B of $V(G)$ *touch* if either they have a vertex in common, or there is an edge with one endpoint in A and second in B. A *bramble* is the family of pairwise touching connected vertex sets in G. A subset $C \subseteq V(G)$ *covers* a bramble $\mathcal{B}$ if it intersects every element of $\mathcal{B}$. The least number of vertices covering bramble $\mathcal{B}$ is the *order* of $\mathcal{B}$.

It is not difficult to see that if G has a bramble $\mathcal{B}$ of order $k+1$, then k cops cannot catch the robber. Indeed, during the chase the robber maintains the invariant that there is always some $X \in \mathcal{B}$ that is free of cops, and in which she resides. This can be clearly maintained at the beginning of the chase, since for every set of initial position of cops there is an element X of $\mathcal{B}$ that will not contain any cops, which the robber can choose for the start. During every round, the robber examines the positions of all the cops after landing. Again, there must be some $Y \in \mathcal{B}$ that will be disjoint from the set of these positions. If $X = Y$ then the robber does not need to move, and otherwise she runs through X (which is connected and free from cops at this point) to a vertex shared with Y or to an edge connecting X and Y. In this manner the robber can get through to the set Y, which is free from the cops both at this point and after landing.

This reasoning shows that the maximum order of a bramble in a graph G provides a lower bound on the number of cops needed in the search game on G, and thus on the treewidth of G. The following deep result of Seymour and Thomas shows that this lower bound is in fact tight. We state this theorem without a proof.

Theorem 7.15 ([414]). *For every $k \geq 0$ and graph G, the treewidth of G is at least k if and only if G contains a bramble of order at least $k+1$.*

Now we have gathered three alternative characterizations of treewidth, and we can state the analogue of Theorem 7.14 for this graph parameter. The result is given without a proof because of its technicality, but the main steps are discussed in Exercise 7.28.

Theorem 7.16. *For any graph G and any $k \geq 0$, the following conditions are equivalent:*

(i) *The treewidth of G is at most k.*
(ii) *The chordal width of G is at most $k+1$.*
(iii) *$k+1$ cops can catch a visible robber on G.*
(iv) *There is no bramble of order larger than $k+1$ in G.*

7.6 Computing treewidth

In all the applications of the treewidth we discussed in this chapter, we were assuming that a tree decomposition of small width is given as part of the input. A natural question is how fast we can find such a decomposition. Unfortunately, it turns out that it is NP-hard to compute the treewidth of a given graph, so we need to resort to FPT or approximation algorithms.

In order to make our considerations more precise, let us consider the Treewidth problem defined as follows: we are given a graph G and an integer k, and the task is to determine whether the treewidth of G is at most k. There is a "simple" argument why Treewidth is fixed-parameter tractable: It follows from the results of the Graph Minors theory of Robertson and Seymour, briefly surveyed in Chapter 6. More precisely, it is easy to see that treewidth is a minor-monotone parameter in the following sense: for every graph G and its minor H, it holds that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ (see Exercise 7.7). Hence if we define $\mathcal{G}_k$ to be the class of graphs of treewidth at most k, then $\mathcal{G}_k$ is closed under taking minors. By Corollary 6.11, there exists a set of forbidden minors $\mathrm{Forb}(\mathcal{G}_k)$, which depends on k only, such that a graph has treewidth k if and only if it does not contain any graph from $\mathrm{Forb}(\mathcal{G}_k)$ as a minor. Hence, we can apply the algorithm of Theorem 6.12 to each $H \in \mathrm{Forb}(\mathcal{G}_k)$ separately, verifying in $f(H) \cdot |V(G)|^3$ time whether it is contained in G as a minor. Since $\mathrm{Forb}(\mathcal{G}_k)$ depends only on k, both in terms of its cardinality and the maximum size of a member, this algorithm runs in $g(k) \cdot |V(G)|^3$ for some function g. Similarly to the algorithms discussed in Section 6.3, this gives only a nonuniform FPT algorithm — Corollary 6.11 does not provide us any means of constructing the family $\mathcal{G}_k$, or even bounding its size.

Alternatively, one could approach the Treewidth problem using the graph searching game that we discussed in Section 7.5. This direction quite easily leads to an algorithm with running time $n^{\mathcal{O}(k)}$; the reader is asked to construct it in Exercise 7.26.

It is, nonetheless, possible to obtain a uniform, constructive algorithm for the Treewidth problem. However, this requires much more involved techniques and technical effort. In the following theorem we state the celebrated algorithm of Bodlaender that shows that Treewidth is FPT.

Theorem 7.17 ([45]). *There exists an algorithm that, given an n-vertex graph G and integer k, runs in time $k^{\mathcal{O}(k^3)} \cdot n$ and either constructs a tree decomposition of G of width at most k, or concludes that* $\mathrm{tw}(G) > k$.

Generally, in the literature there is a variety of algorithms for computing good tree decompositions of graphs. The algorithm of Theorem 7.17 is the fastest known *exact* parameterized algorithm, meaning that it computes the exact value of the treewidth. Most other works follow the direction of *approximating* treewidth. In other words, we relax the requirement that the returned decomposition has width at most k by allowing the algorithm to output a decomposition with a slightly larger width. In exchange we would like to obtain better parameter dependence than $k^{\mathcal{O}(k^3)}$, which would be a dominating factor in most applications.

In this section we shall present the classical approximation algorithm for treewidth that originates in the work of Robertson and Seymour. More precisely, we prove the following result.

Theorem 7.18. *There exists an algorithm that, given an n-vertex graph G and integer k, runs in time $\mathcal{O}(8^k k^2 \cdot n^2)$ and either constructs a tree decomposition of G of width at most $4k+4$, or concludes that* $\mathrm{tw}(G) > k$.

We remark that the running time of algorithm of Theorem 7.17 depends linearly on the size of the graph, whereas in Theorem 7.18 the dependence is quadratic. Although in this book we usually do not investigate precisely the polynomial factors in the running time bounds, we would like to emphasize that this particular difference is important in applications, as the running time of most dynamic-programming algorithms on tree decompositions depends linearly on the graph size, and, consequently, the n^2 factor coming from Theorem 7.18 is a bottleneck. To cope with this issue, very recently a 5-approximation algorithm running in time $2^{\mathcal{O}(k)} \cdot n$ has been developed (see also the bibliographic notes at the end of the chapter); however, the techniques needed to obtain such a result are beyond the scope of this book.

The algorithm of Theorem 7.18 is important not only because it provides an efficient way of finding a reasonable tree decomposition of a graph, but also because the general strategy employed in this algorithm has been reused multiple times in approximation algorithms for other structural graph parameters. The crux of this approach can be summarized as follows.

- It is easy to see that every n-vertex tree T contains a vertex v such that every connected component of $T - v$ has at most $\frac{n}{2}$ vertices.

A similar fact can be proved for graphs of bounded treewidth. More precisely, graphs of treewidth at most k have balanced separators of size at most $k+1$: it is possible to remove $k+1$ vertices from the graph so that every connected component left is "significantly" smaller than the original graph.

- The algorithm decomposes the graph recursively. At each step we try to decompose some part of a graph that has only ck vertices on its boundary, for some constant c.
- The crucial step is to find a small separator of the currently processed part H with the following property: the separator splits H into two pieces H_1, H_2 so that the boundary of H gets partitioned evenly between H_1 and H_2. If for some $\alpha > 1$, each of H_1 and H_2 contains only, say, $\frac{ck}{\alpha}$ vertices of $\partial(H)$, then we have $|\partial(H_1)|, |\partial(H_2)| \leq \frac{ck}{\alpha} + (k+1)$; the summand $k+1$ comes from the separator. If $\frac{ck}{\alpha} + (k+1) \leq ck$, then we can recurse into pieces H_1 and H_2.

Before we proceed to the proof of Theorem 7.18, we need to introduce some auxiliary results about balanced separators and separations in graphs of small treewidth.

7.6.1 Balanced separators and separations

From now on we will work with the notions of separators and separations in graphs; we have briefly discussed these concepts in Section 7.2. Let us recall the main points and develop further these definitions. A pair of vertex subsets (A, B) is a *separation* in graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. The *separator* of this separation is $A \cap B$, and the *order* of separation (A, B) is equal to $|A \cap B|$.

We say that (A, B) *separates* two sets X, Y if $X \subseteq A$ and $Y \subseteq B$. Note that then every X-Y path, i.e., a path between a vertex from X and a vertex from Y, has to pass through at least one vertex of $A \cap B$. Given this observation, we may say that a set $C \subseteq V(G)$ *separates* two vertex sets X and Y if every X-Y path contains a vertex of C. Note that in this definition we allow C to contain vertices from X or Y. Given C that separates X and Y, it is easy to construct a separation (A, B) that separates X and Y and has C as the separator.

For two vertex sets X and Y, by $\mu(X, Y)$ we denote the minimum size of a separator separating X and Y, or, equivalently, the minimum order of a separation separating X and Y. Whenever the graph we refer to is not clear from the context, we put it as a subscript of the μ symbol. By the classic theorem of Menger, $\mu(X, Y)$ is equal to the maximum number of vertex-disjoint X-Y paths. The value of $\mu(X, Y)$ can be computed in polynomial time by

any algorithm for computing the maximum flow in a graph. Moreover, within the same running time we can obtain both a separation of order $\mu(X,Y)$ separating X and Y, as well as a family of $\mu(X,Y)$ vertex-disjoint X-Y paths (see also Theorems 8.2, 8.4 and 8.5).

We now move to our first auxiliary result, which states that graphs of treewidth at most k have balanced separators of size at most $k+1$. We prove a slightly more general statement of this fact. Assume that $\mathbf{w}\colon V(G) \to \mathbb{R}_{\geq 0}$ is a nonnegative weight function on the vertices of G. For a set $X \subseteq V(G)$, let us define $\mathbf{w}(X) = \sum_{u \in X} \mathbf{w}(u)$. Let $\alpha \in (0,1)$ be any constant. We say that a set $X \subseteq V(G)$ is an *α-balanced separator* in G if for every connected component D of $G - X$, it holds that $\mathbf{w}(D) \leq \alpha \cdot \mathbf{w}(V(G))$. Informally speaking, a balanced separator breaks the graph into pieces whose weights constitute only a constant fraction of the original weight of the graph.

Like trees, graphs of "small" treewidth have "small" balanced separators. This property is heavily exploited in numerous algorithmic applications of treewidth.

If we put uniform weights on vertices, then balanced separators split the graph more or less evenly with respect to the cardinalities of the obtained pieces. Recall, however, that in the presented strategy for the algorithm of Theorem 7.18 we would like to split evenly some smaller subset of vertices, which is the boundary of the currently processed part of the graph. Therefore, in our applications we will put weights 1 on some vertices of the graph, and 0 on the other. Then balanced separators split evenly the set of vertices that are assigned weight 1. In other words, what we need is a weighted version of balanced separators.

We now prove the fact that graphs of small treewidth admit small balanced separators.

Lemma 7.19. *Assume G is a graph of treewidth at most k, and consider a nonnegative weight function $\mathbf{w}\colon V(G) \to \mathbb{R}_{\geq 0}$ on the vertices of G. Then in G there exists a $\frac{1}{2}$-balanced separator X of size at most $k+1$.*

Proof. Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of G of width at most k. We prove that one of the bags of $\mathcal{T}$ is a $\frac{1}{2}$-balanced separator we are looking for. We start by rooting T at an arbitrarily chosen node r. For a node t of T, let V_t be the set of vertices of G contained in the bags of the subtree T_t of T rooted at node t (including t itself). Let us select a node t of T such that:

- $\mathbf{w}(V_t) \geq \frac{1}{2}\mathbf{w}(V(G))$, and subject to
- t is at the maximum distance in T from r.

Observe that such a vertex t exists because root r satisfies the first condition.

We claim that X_t is a $\frac{1}{2}$-balanced separator in G. Let $t_1, \ldots, t_p$ be the children of t in T (possibly $p = 0$). Observe that every connected component D of $V(G) \setminus X_t$ is entirely contained either in $V(G) \setminus V_t$, or in $V_{t_i} \setminus X_t$ for some $i \in \{1, 2, \ldots, p\}$. Since $\mathbf{w}(V_t) \geq \frac{1}{2}\mathbf{w}(V(G))$, then $\mathbf{w}(V(G) \setminus V_t) \leq \frac{1}{2}\mathbf{w}(V(G))$. On the other hand, by the choice of t we have that $\mathbf{w}(V_{t_i}) < \frac{1}{2}\mathbf{w}(V(G))$ for every $i \in \{1, \ldots, p\}$. Consequently, in both cases D is entirely contained in a set whose weight is at most $\frac{1}{2}\mathbf{w}(V(G))$, which means that $\mathbf{w}(D) \leq \frac{1}{2}\mathbf{w}(V(G))$. □

The proof of Lemma 7.19 shows that if a tree decomposition of G of width at most k is given, one can find such a $\frac{1}{2}$-balanced separator in polynomial time. Moreover, the found separator is one of the bags of the given decomposition. However, in the remainder of this section we will only need the existential statement.

Recall that in the sketched strategy we were interested in splitting the currently processed subgraph into two even parts. However, after deleting the vertices of the separator given by Lemma 7.19 the graph can have more than two connected components. For this reason, we would like to group these pieces (components) into two roughly equal halves. More formally, we say that a separation (A, B) of G is an *α-balanced separation* if $\mathbf{w}(A \setminus B) \leq \alpha \cdot \mathbf{w}(V(G))$ and $\mathbf{w}(B \setminus A) \leq \alpha \cdot \mathbf{w}(V(G))$. The next lemma shows that we can focus on balanced separations instead of separators, at a cost of relaxing $\frac{1}{2}$ to $\frac{2}{3}$.

Lemma 7.20. *Assume G is a graph of treewidth at most k, and consider a nonnegative weight function $\mathbf{w} \colon V(G) \to \mathbb{R}_{\geq 0}$ on the vertices of G. Then in G there exists a $\frac{2}{3}$-balanced separation (A, B) of order at most $k + 1$.*

Proof. Let X be a $\frac{1}{2}$-balanced separator in G of size at most $k + 1$, given by Lemma 7.19. Let $D_1, D_2, \ldots, D_p$ be the vertex sets of connected components of $G - X$. For $i \in \{1, 2, \ldots, p\}$, let $a_i = \mathbf{w}(D_i)$; recall that we know that $a_i \leq \frac{1}{2}\mathbf{w}(V(G))$. By reordering the components if necessary, assume that $a_1 \geq a_2 \geq \ldots \geq a_p$.

Let q be the smallest index such that $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$, or $q = p$ if no such index exists. We claim that $\sum_{i=1}^{q} a_i \leq \frac{2}{3}\mathbf{w}(V(G))$. This is clearly true if $\sum_{i=1}^{p} a_i < \frac{1}{3}\mathbf{w}(V(G))$. Also, if $q = 1$, then $\sum_{i=1}^{q} a_i = a_1 \leq \frac{1}{2}\mathbf{w}(V(G))$ and the claim holds in this situation as well. Therefore, assume that $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$ and $q > 1$. By the minimality of q, we have that $\sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G))$. Hence $a_q \leq a_{q-1} \leq \sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G))$, so $\sum_{i=1}^{q} a_i = a_q + \sum_{i=1}^{q-1} a_i < \frac{1}{3}\mathbf{w}(V(G)) + \frac{1}{3}\mathbf{w}(V(G)) = \frac{2}{3}\mathbf{w}(V(G))$. This proves the claim.

We now define $A = X \cup \bigcup_{i=1}^{q} D_i$ and $B = X \cup \bigcup_{i=q+1}^{p} D_i$. Clearly (A, B) is a separation of G with X being the separator, so it has order at most $k+1$. By the claim from the last paragraph, we have that $\mathbf{w}(A \setminus B) = \sum_{i=1}^{q} a_i \leq \frac{2}{3}\mathbf{w}(V(G))$. Moreover, either we have that $p = q$ and $B \setminus A = \emptyset$, or $p < q$ and then $\sum_{i=1}^{q} a_i \geq \frac{1}{3}\mathbf{w}(V(G))$. In the first case $\mathbf{w}(B \setminus A) = 0$, and in the latter case we have that $\mathbf{w}(B \setminus A) = \sum_{i=q+1}^{p} a_i \leq \mathbf{w}(V(G)) - \frac{1}{3}\mathbf{w}(V(G)) =$