Chapter 10

Extending the Limits of Tractability

Although we started the book by studying a number of techniques for solving problems efficiently, we've been looking for a while at classes of problems—NP-complete and PSPACE-complete problems—for which no efficient solution is believed to exist. And because of the insights we've gained this way, we've implicitly arrived at a two-pronged approach to dealing with new computational problems we encounter: We try for a while to develop an efficient algorithm; and if this fails, we then try to prove it NP-complete (or even PSPACE-complete). Assuming one of the two approaches works out, you end up either with a solution to the problem (an algorithm), or a potent "reason" for its difficulty: It is as hard as many of the famous problems in computer science.

Unfortunately, this strategy will only get you so far. If there is a problem that people really want your help in solving, they won't be particularly satisfied with the resolution that it's NP-hard[1] and that they should give up on it. They'll still want a solution that's as good as possible, even if it's not the exact, or optimal, answer. For example, in the Independent Set Problem, even if we can't find the largest independent set in a graph, it's still natural to want to compute for as much time as we have available, and output as large an independent set as we can find.

The next few topics in the book will be focused on different aspects of this notion. In Chapters 11 and 12, we'll look at algorithms that provide approximate answers with guaranteed error bounds in polynomial time; we'll also consider *local search heuristics* that are often very effective in practice,

[1] We use the term *NP-hard* to mean "at least as hard as an NP-complete problem." We avoid referring to optimization problems as NP-complete, since technically this term applies only to decision problems.

even when we are not able to establish any provable guarantees about their behavior.

But to start, we explore some situations in which one can exactly solve instances of NP-complete problems with reasonable efficiency. How do these situations arise? The point is to recall the basic message of NP-completeness: the worst-case instances of these problems are very difficult and not likely to be solvable in polynomial time. On a *particular* instance, however, it's possible that we are not really in the "worst case"—maybe, in fact, the instance we're looking at has some special structure that makes our task easier. Thus the crux of this chapter is to look at situations in which it is possible to quantify some precise senses in which an instance may be easier than the worst case, and to take advantage of these situations when they occur.

We'll look at this principle in several concrete settings. First we'll consider the Vertex Cover Problem, in which there are two natural "size" parameters for a problem instance: the size of the graph, and the size of the vertex cover being sought. The NP-completeness of Vertex Cover suggests that we will have to be exponential in (at least) one of these parameters; but judiciously choosing which one can have an enormous effect on the running time.

Next we'll explore the idea that many NP-complete graph problems become polynomial-time solvable if we require the input to be a tree. This is a concrete illustration of the way in which an input with "special structure" can help us avoid many of the difficulties that can make the worst case intractable. Armed with this insight, one can generalize the notion of a tree to a more general class of graphs—those with small *tree-width*—and show that many NP-complete problems are tractable on this more general class as well.

Having said this, we should stress that our basic point remains the same as it has always been: *Exponential algorithms scale very badly*. The current chapter represents ways of staving off this problem that can be effective in various settings, but there is clearly no way around it in the fully general case. This will motivate our focus on approximation algorithms and local search in subsequent chapters.

10.1 Finding Small Vertex Covers

Let us briefly recall the Vertex Cover Problem, which we saw in Chapter 8 when we covered NP-completeness. Given a graph $G = (V, E)$ and an integer k, we would like to find a vertex cover of size at most k—that is, a set of nodes $S \subseteq V$ of size $|S| \leq k$, such that every edge $e \in E$ has at least one end in S.

Like many NP-complete decision problems, Vertex Cover comes with two parameters: n, the number of nodes in the graph, and k, the allowable size of a vertex cover. This means that the range of possible running-time bounds is much richer, since it involves the interplay between these two parameters.

The Problem

Let's consider this interaction between the parameters n and k more closely. First of all, we notice that if k is a fixed constant (e.g., $k = 2$ or $k = 3$), then we can solve Vertex Cover in polynomial time: We simply try all subsets of V of size k, and see whether any of them constitute a vertex cover. There are $\binom{n}{k}$ subsets, and each takes time $O(kn)$ to check whether it is a vertex cover, for a total time of $O(kn\binom{n}{k}) = O(kn^{k+1})$. So from this we see that the intractability of Vertex Cover only sets in for real once k grows as a function of n.

However, even for moderately small values of k, a running time of $O(kn^{k+1})$ is quite impractical. For example, if $n = 1{,}000$ and $k = 10$, then on a computer executing a million high-level instructions per second, it would take at least 10^{24} seconds to decide if G has a k-node vertex cover—which is several orders of magnitude larger than the age of the universe. And this is for a small value of k, where the problem was supposed to be more tractable! It's natural to start asking whether we can do something that is practically viable when k is a small constant.

It turns out that a much better algorithm can be developed, with a running-time bound of $O(2^k \cdot kn)$. There are two things worth noticing about this. First, plugging in $n = 1{,}000$ and $k = 10$, we see that our computer should be able to execute the algorithm in a few seconds. Second, we see that as k grows, the running time is still increasing very rapidly; it's simply that the exponential dependence on k has been moved out of the exponent on n and into a separate function. From a practical point of view, this is much more appealing.

Designing the Algorithm

As a first observation, we notice that if a graph has a small vertex cover, then it cannot have very many edges. Recall that the *degree* of a node is the number of edges that are incident to it.

(10.1) *If $G = (V, E)$ has n nodes, the maximum degree of any node is at most d, and there is a vertex cover of size at most k, then G has at most kd edges.*

Proof. Let S be a vertex cover in G of size $k' \le k$. Every edge in G has at least one end in S; but each node in S can cover at most d edges. Thus there can be at most $k'd \le kd$ edges in G. ■

Since the degree of any node in a graph can be at most $n - 1$, we have the following simple consequence of (10.1).

(10.2) *If $G = (V, E)$ has n nodes and a vertex cover of size k, then G has at most $k(n-1) \leq kn$ edges.*

So, as a first step in our algorithm, we can check if G contains more than kn edges; if it does, then we know that the answer to the decision problem—Is there a vertex cover of size at most k?—is no. Having done this, we will assume that G contains at most kn edges.

The idea behind the algorithm is conceptually very clean. We begin by considering any edge $e = (u, v)$ in G. In any k-node vertex cover S of G, one of u or v must belong to S. Suppose that u belongs to such a vertex cover S. Then if we delete u and all its incident edges, it must be possible to cover the remaining edges by at most $k-1$ nodes. That is, defining $G-\{u\}$ to be the graph obtained by deleting u and all its incident edges, there must be a vertex cover of size at most $k-1$ in $G-\{u\}$. Similarly, if v belongs to S, this would imply there is a vertex cover of size at most $k-1$ in $G-\{v\}$.

Here is a concrete way to formulate this idea.

(10.3) *Let $e = (u, v)$ be any edge of G. The graph G has a vertex cover of size at most k if and only if at least one of the graphs $G-\{u\}$ and $G-\{v\}$ has a vertex cover of size at most $k-1$.*

Proof. First, suppose G has a vertex cover S of size at most k. Then S contains at least one of u or v; suppose that it contains u. The set $S-\{u\}$ must cover all edges that have neither end equal to u. Therefore $S-\{u\}$ is a vertex cover of size at most $k-1$ for the graph $G-\{u\}$.

Conversely, suppose that one of $G-\{u\}$ and $G-\{v\}$ has a vertex cover of size at most $k-1$—suppose in particular that $G-\{u\}$ has such a vertex cover T. Then the set $T \cup \{u\}$ covers all edges in G, so it is a vertex cover for G of size at most k. ■

Statement (10.3) directly establishes the correctness of the following recursive algorithm for deciding whether G has a k-node vertex cover.

```
To search for a $k$-node vertex cover in $G$:
  If $G$ contains no edges, then the empty set is a vertex cover
  If $G$ contains$> k\ |V|$ edges, then it has no $k$-node vertex cover
  Else let $e = (u, v)$ be an edge of $G$
    Recursively check if either of $G-\{u\}$ or $G-\{v\}$
                has a vertex cover of size $k-1$
    If neither of them does, then $G$ has no $k$-node vertex cover
    Else, one of them (say, $G-\{u\}$) has a $(k-1)$-node vertex cover $T$
      In this case, $T \cup \{u\}$ is a $k$-node vertex cover of $G$
    Endif
  Endif
```

Analyzing the Algorithm

Now we bound the running time of this algorithm. Intuitively, we are searching a "tree of possibilities"; we can picture the recursive execution of the algorithm as giving rise to a tree, in which each node corresponds to a different recursive call. A node corresponding to a recursive call with parameter k has, as children, two nodes corresponding to recursive calls with parameter $k-1$. Thus the tree has a total of at most 2^{k+1} nodes. In each recursive call, we spend $O(kn)$ time.

Thus, we can prove the following.

(10.4) *The running time of the Vertex Cover Algorithm on an n-node graph, with parameter k, is $O(2^k \cdot kn)$.*

We could also prove this by a recurrence as follows. If $T(n,k)$ denotes the running time on an n-node graph with parameter k, then $T(\cdot,\cdot)$ satisfies the following recurrence, for some absolute constant c:

$$\begin{aligned} T(n,1) &\le cn, \\ T(n,k) &\le 2T(n,k-1) + ckn. \end{aligned}$$

By induction on $k \ge 1$, it is easy to prove that $T(n,k) \le c \cdot 2^k kn$. Indeed, if this is true for $k-1$, then

$$\begin{aligned} T(n,k) &\le 2T(n-1,k-1) + ckn \\ &\le 2c \cdot 2^{k-1}(k-1)n + ckn \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn \\ &\le c \cdot 2^k kn. \end{aligned}$$

In summary, this algorithm is a powerful improvement on the simple brute-force approach. However, no exponential algorithm can scale well for very long, and that includes this one. Suppose we want to know whether there is a vertex cover with at most 40 nodes, rather than 10; then, on the same machine as before, our algorithm will take a significant number of years to terminate.

10.2 Solving NP-Hard Problems on Trees

In Section 10.1 we designed an algorithm for the Vertex Cover Problem that works well when the size of the desired vertex cover is not too large. We saw that finding a relatively small vertex cover is much easier than the Vertex Cover Problem in its full generality.

Here we consider special cases of NP-complete graph problems with a different flavor—not when the natural "size" parameters are small, but when the input graph is structurally "simple." Perhaps the simplest types of graphs are *trees*. Recall that an undirected graph is a tree if it is connected and has no cycles. Not only are trees structurally easy to understand, but it has been found that many NP-hard graph problems can be solved efficiently when the underlying graph is a tree. At a qualitative level, the reason for this is the following: If we consider a subtree of the input rooted at some node v, the solution to the problem restricted to this subtree only "interacts" with the rest of the tree through v. Thus, by considering the different ways in which v might figure in the overall solution, we can essentially decouple the problem in v's subtree from the problem in the rest of the tree.

It takes some amount of effort to make this general approach precise and to turn it into an efficient algorithm. Here we will see how to do this for variants of the Independent Set Problem; however, it is important to keep in mind that this principle is quite general, and we could equally well have considered many other NP-complete graph problems on trees.

First we will see that the Independent Set Problem itself can be solved by a greedy algorithm on a tree. Then we will consider the generalization called the *Maximum-Weight Independent Set Problem*, in which nodes have weight, and we seek an independent set of maximum weight. We'll see that the Maximum-Weight Independent Set Problem can be solved on trees via dynamic programming, using a fairly direct implementation of the intuition described above.

A Greedy Algorithm for Independent Set on Trees

The starting point of our greedy algorithm on a tree is to consider the way a solution looks from the perspective of a single edge; this is a variant on an idea from Section 10.1. Specifically, consider an edge $e = (u, v)$ in G. In any independent set S of G, at most one of u or v can belong to S. We'd like to find an edge e for which we can greedily decide which of the two ends to place in our independent set.

For this we exploit a crucial property of trees: Every tree has at least one *leaf*—a node of degree 1. Consider a leaf v, and let (u, v) be the unique edge incident to v. How might we "greedily" evaluate the relative benefits of including u or v in our independent set S? If we include v, the only other node that is directly "blocked" from joining the independent set is u. If we include u, it blocks not only v but all the other nodes joined to u as well. So if we're trying to maximize the size of the independent set, it seems that including v should be better than, or at least as good as, including u.

(10.5) *If $T = (V, E)$ is a tree and v is a leaf of the tree, then there exists a maximum-size independent set that contains v.*

Proof. Consider a maximum-size independent set S, and let $e = (u, v)$ be the unique edge incident to node v. Clearly, at least one of u or v is in S; for if neither is present, then we could add v to S, thereby increasing its size. Now, if $v \in S$, then we are done; and if $u \in S$, then we can obtain another independent set S' of the same size by deleting u from S and inserting v. ■

We will use (10.5) repeatedly to identify and delete nodes that can be placed in the independent set. As we do this deletion, the tree T may become disconnected. So, to handle things more cleanly, we actually describe our algorithm for the more general case in which the underlying graph is a *forest*—a graph in which each connected component is a tree. We can view the problem of finding a maximum-size independent set for a forest as really being the same as the problem for trees: an optimal solution for a forest is simply the union of optimal solutions for each tree component, and we can still use (10.5) to think about the problem in any component.

Specifically, suppose we have a forest F; then (10.5) allows us to make our first decision in the following greedy way. Consider again an edge $e = (u, v)$, where v is a leaf. We will include node v in our independent set S, and not include node u. Given this decision, we can delete the node v (since it's already been included) and the node u (since it cannot be included) and obtain a smaller forest. We continue recursively on this smaller forest to get a solution.

```
To find a maximum-size independent set in a forest $F$:
  Let $S$ be the independent set to be constructed (initially empty)
  While $F$ has at least one edge
     Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
     Add $v$ to $S$
     Delete from $F$ nodes $u$ and $v$, and all edges incident to them
  Endwhile
  Return $S$
```

(10.6) *The above algorithm finds a maximum-size independent set in forests (and hence in trees as well).*

Although (10.5) was a very simple fact, it really represents an application of one of the design principles for greedy algorithms that we saw in Chapter 4: an *exchange argument*. In particular, the crux of our Independent Set Algorithm is the observation that any solution not containing a particular leaf can be "transformed" into a solution that is just as good and contains the leaf.

To implement this algorithm so it runs quickly, we need to maintain the current forest F in a way that allows us to find an edge incident to a leaf efficiently. It is not hard to implement this algorithm in linear time: We need to maintain the forest in a way that allows us to do so on one iteration of the `While` loop in time proportional to the number of edges deleted when u and v are removed.

The Greedy Algorithm on More General Graphs The greedy algorithm specified above is not guaranteed to work on general graphs, because we cannot be guaranteed to find a leaf in every iteration. However, (10.5) *does* apply to any graph: if we have an arbitrary graph G with an edge (u, v) such that u is the only neighbor of v, then it's always safe to put v in the independent set, delete u and v, and iterate on the smaller graph.

So if, by repeatedly deleting degree-1 nodes and their neighbors, we're able to eliminate the entire graph, then we're guaranteed to have found an independent set of maximum size—even if the original graph was not a tree. And even if we don't manage to eliminate the whole graph, we may still succeed in running a few iterations of the algorithm in succession, thereby shrinking the size of the graph and making other approaches more tractable. Thus our greedy algorithm is a useful heuristic to try "opportunistically" on arbitrary graphs, in the hope of making progress toward finding a large independent set.

Maximum-Weight Independent Set on Trees

Next we turn to the more complex problem of finding a maximum-weight independent set. As before, we assume that our graph is a tree $T = (V, E)$. Now we also have a positive *weight* w_v associated with each node $v \in V$. The *Maximum-Weight Independent Set Problem* is to find an independent set S in the graph $T = (V, E)$ so that the total weight $\sum_{v \in S} w_v$ is as large as possible.

First we try the idea we used before to build a greedy solution for the case without weights. Consider an edge $e = (u, v)$, such that v is a leaf. Including v blocks fewer nodes from entering the independent set; so, if the weight of v is at least as large as the weight of u, then we can indeed make a greedy decision just as we did in the case without weights. However, if $w_v < w_u$, we face a dilemma: We acquire more weight by including u, but we retain more options down the road if we include v. There seems to be no easy way to resolve this locally, without considering the rest of the graph. However, there is still something we can say. If node u has many neighbors $v_1, v_2, \ldots$ that are leaves, then we should make the same decision for all of them: Once we decide not to include u in the independent set, we may as well go ahead and include all its adjacent leaves. So for the subtree consisting of u and its adjacent leaves, we really have only two "reasonable" solutions to consider: including u, or including all the leaves.

We will use these ideas to design a polynomial-time algorithm using dynamic programming. As we recall, dynamic programming allows us to record a few different solutions, build these up through a sequence of subproblems, and thereby decide only at the end which of these possibilities will be used in the overall solution.

The first issue to decide for a dynamic programming algorithm is what our subproblems will be. For Maximum-Weight Independent Set, we will construct subproblems by *rooting* the tree T at an arbitrary node r; recall that this is the operation of "orienting" all the tree's edges away from r. Specifically, for any node $u \neq r$, the parent $p(u)$ of u is the node adjacent to u along the path from the root r. The other neighbors of u are its children, and we will use *children*(u) to denote the set of children of u. The node u and all its descendants form a subtree T_u whose root is u.

We will base our subproblems on these subtrees T_u. The tree T_r is our original problem. If $u \neq r$ is a leaf, then T_u consists of a single node. For a node u all of whose children are leaves, we observe that T_u is the kind of subtree discussed above.

To solve the problem by dynamic programming, we will start at the leaves and gradually work our way up the tree. For a node u, we want to solve the subproblem associated with the tree T_u after we have solved the subproblems for all its children. To get a maximum-weight independent set S for the tree T_u, we will consider two cases: Either we include the node u in S or we do not. If we include u, then we cannot include any of its children; if we do not include u, then we have the freedom to include or omit these children. This suggests that we should define two subproblems for each subtree T_u: the subproblem $\text{OPT}_{in}(u)$ will denote the maximum weight of an independent set of T_u that includes u, and the subproblem $\text{OPT}_{out}(u)$ will denote the maximum weight of an independent set of T_u that does not include u.

Now that we have our subproblems, it is not hard to see how to compute these values recursively. For a leaf $u \neq r$, we have $\text{OPT}_{out}(u) = 0$ and $\text{OPT}_{in}(u) = w_u$. For all other nodes u, we get the following recurrence that defines $\text{OPT}_{out}(u)$ and $\text{OPT}_{in}(u)$ using the values for u's children.

(10.7) *For a node u that has children, the following recurrence defines the values of the subproblems:*

- $\text{OPT}_{in}(u) = w_u + \sum_{v \in children(u)} \text{OPT}_{out}(v)$
- $\text{OPT}_{out}(u) = \sum_{v \in children(u)} \max(\text{OPT}_{out}(v), \text{OPT}_{in}(v))$.

Using this recurrence, we get a dynamic programming algorithm by building up the optimal solutions over larger and larger subtrees. We define arrays $M_{out}[u]$ and $M_{in}[u]$, which hold the values $\text{OPT}_{out}(u)$ and $\text{OPT}_{in}(u)$, respectively. For building up solutions, we need to process all the children of a node before we process the node itself; in the terminology of tree traversal, we visit the nodes in *post-order*.

```
To find a maximum-weight independent set of a tree $T$:
    Root the tree at a node $r$
    For all nodes $u$ of $T$ in post-order
        If $u$ is a leaf then set the values:
            $M_{out}[u] = 0$
            $M_{in}[u] = w_u$
        Else set the values:
            $M_{out}[u] = \sum_{v \in children(u)} \max(M_{out}[u], M_{in}[u])$
            $M_{in}[u] = w_u + \sum_{v \in children(u)} M_{out}[u]$.
        Endif
    Endfor
    Return $\max(M_{out}[r], M_{in}[r])$
```

This gives us the value of the maximum-weight independent set. Now, as is standard in the dynamic programming algorithms we've seen before, it's easy to recover an actual independent set of maximum weight by recording the decision we make for each node, and then tracing back through these decisions to determine which nodes should be included. Thus we have

(10.8) *The above algorithm finds a maximum-weight independent set in trees in linear time.*

10.3 Coloring a Set of Circular Arcs

Some years back, when telecommunications companies began focusing intensively on a technology known as *wavelength-division multiplexing*, researchers at these companies developed a deep interest in a previously obscure algorithmic question: the problem of coloring a set of circular arcs.

After explaining how the connection came about, we'll develop an algorithm for this problem. The algorithm is a more complex variation on the theme of Section 10.2: We approach a computationally hard problem using dynamic programming, building up solutions over a set of subproblems that only "interact" with each other on very small pieces of the input. Having to worry about only this very limited interaction serves to control the complexity of the algorithm.

The Problem

Let's start with some background on how network routing issues led to the question of circular-arc coloring. Wavelength-division multiplexing (WDM) is a methodology that allows multiple communication streams to share a single portion of fiber-optic cable, provided that the streams are transmitted on this cable using different wavelengths. Let's model the underlying communication network as a graph $G = (V, E)$, with each *communication stream* consisting of a path P_i in G; we imagine data flowing along this stream from one endpoint of P_i to the other. If the paths P_i and P_j share some edge in G, it is still possible to send data along these two streams simultaneously as long as they are routed using different *wavelengths*. So our goal is the following: Given a set of k available wavelengths (labeled $1, 2, \ldots, k$), we wish to assign a wavelength to each stream P_i in such a way that each pair of streams that share an edge in the graph are assigned different wavelengths. We'll refer to this as an instance of the *Path Coloring Problem*, and we'll call a solution to this instance—a legal assignment of wavelengths to paths—a *k-coloring*.

This is a natural problem that we could consider as it stands; but from the point of view of the fiber-optic routing context, it is useful to make one further simplification. Many applications of WDM take place on networks G that are extremely simple in structure, and so it is natural to restrict the instances of Path Coloring by making some assumptions about this underlying network structure. In fact, one of the most important special cases in practice is also one of the simplest: when the underlying network is simply a ring; that is, it can be modeled using a graph G that is a cycle on n nodes.

This is the case we will focus on here: We are given a graph $G = (V, E)$ that is a cycle on n nodes, and we are given a set of paths $P_1, \ldots, P_m$ on this cycle. The goal, as above, is to assign one of k given wavelengths to each path

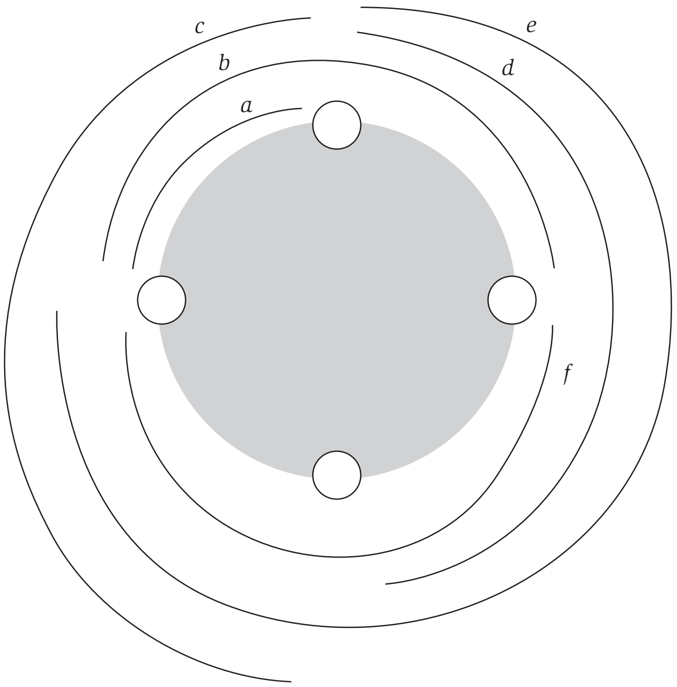

Figure 10.1 An instance of the Circular-Arc Coloring Problem with six arcs (a, b, c, d, e, f) on a four-node cycle.

P_i so that overlapping paths receive different wavelengths. We will refer to this as a *valid* assignment of wavelengths to the paths. Figure 10.1 shows a sample instance of this problem. In this instance, there is a valid assignment using $k = 3$ wavelengths, by assigning wavelength 1 to the paths a and e, wavelength 2 to the paths b and f, and wavelength 3 to the paths c and d. From the figure, we see that the underlying cycle network can be viewed as a circle, and the paths as arcs on this circle; hence we will refer to this special case of Path Coloring as the *Circular-Arc Coloring Problem*.

The Complexity of Circular-Arc Coloring It's not hard to see that Circular-Arc Coloring can be directly reduced to Graph Coloring. Given an instance of Circular-Arc Coloring, we define a graph H that has a node z_i for each path P_i, and we connect nodes z_i and z_j in H if the paths P_i and P_j share an edge in G. Now, routing all streams using k wavelengths is simply the problem of coloring H using at most k colors. (In fact, this problem is yet another application of graph coloring in which the abstract "colors," since they encode different wavelengths of light, are actually colors.)

Note that this doesn't imply that Circular-Arc Coloring is NP-complete—all we've done is to reduce it *to* a known NP-complete problem, which doesn't tell us anything about its difficulty. For Path Coloring on general graphs, in fact, it is easy to reduce from Graph Coloring to Path Coloring, thereby establishing that Path Coloring is NP-complete. However, this straightforward reduction does not work when the underlying graph is as simple as a cycle. So what is the complexity of Circular-Arc Coloring?

It turns out that Circular-Arc Coloring can be shown to be NP-complete using a very complicated reduction. This is bad news for people working with optical networks, since it means that optimal wavelength assignment is unlikely to be efficiently solvable. But, in fact, the known reductions that show Circular-Arc Coloring is NP-complete all have the following interesting property: The hard instances of Circular-Arc Coloring that they produce all involve a set of available wavelengths that is quite large. So, in particular, these reductions don't show that the Circular-Arc Coloring is hard in the case when the number of wavelengths is small; they leave open the possibility that for every fixed, *constant* number of wavelengths k, it is possible to solve the wavelength assignment problem in time polynomial in n (the size of the cycle) and m (the number of paths). In other words, we could hope for a running time of the form we saw for Vertex Cover in Section 10.1: $O(f(k) \cdot p(n, m))$, where $f(\cdot)$ may be a rapidly growing function but $p(\cdot, \cdot)$ is a polynomial.

Such a running time would be appealing (assuming $f(\cdot)$ does not grow too outrageously), since it would make wavelength assignment potentially feasible when the number of wavelengths is small. One way to appreciate the challenge in obtaining such a running time is to note the following analogy: The general Graph Coloring Problem is already hard for three colors. So if Circular-Arc Coloring were tractable for each fixed number of wavelengths (i.e., colors) k, it would show that it's a special case of Graph Coloring with a qualitatively different complexity.

The goal of this section is to design an algorithm with this type of running time, $O(f(k) \cdot p(n, m))$. As suggested at the beginning of the section, the algorithm itself builds on the intuition we developed in Section 10.2 when solving Maximum-Weight Independent Set on trees. There the difficult search inherent in finding a maximum-weight independent set was made tractable by the fact that for each node v in a tree T, the problems in the components of $T - \{v\}$ became completely decoupled once we decided whether or not to include v in the independent set. This is a specific example of the general principle of fixing a small set of decisions, and thereby separating the problem into smaller subproblems that can be dealt with independently.

The analogous idea here will be to choose a particular point on the cycle and decide how to color the arcs that cross over this point; fixing these degrees of freedom allows us to define a series of smaller and smaller subproblems on the remaining arcs.

Designing the Algorithm

Let's pin down some notation we're going to use. We have a graph G that is a cycle on n nodes; we denote the nodes by $v_1, v_2, \ldots, v_n$, and there is an edge (v_i, v_{i+1}) for each i, and also an edge (v_n, v_1). We have a set of paths $P_1, P_2, \ldots, P_m$ in G, we have a set of k available colors; we want to color the paths so that if P_i and P_j share an edge, they receive different colors.

A Simple Special Case: Interval Coloring In order to build up to an algorithm for Circular-Arc Coloring, we first briefly consider an easier coloring problem: the problem of coloring intervals on a line. This can be viewed as a special case of Circular-Arc Coloring in which the arcs lie only in one hemisphere; we will see that once we do not have difficulties from arcs "wrapping around," the problem becomes much simpler. So in this special case, we are given a set of intervals, and we must label each one with a number in such a way that any two overlapping intervals receive different labels.

We have actually seen exactly this problem before: It is the Interval Partitioning (or Interval Coloring) Problem for which we gave an optimal greedy algorithm at the end of Section 4.1. In addition to showing that there is an efficient, optimal algorithm for coloring intervals, our analysis in that earlier section revealed a lot about the structure of the problem. Specifically, if we define the *depth* of a set of intervals to be the maximum number that pass over any single point, then our greedy algorithm from Chapter 4 showed that the minimum number of colors needed is always equal to the depth. Note that the number of colors required is clearly at least the depth, since intervals containing a common point need different colors; the key here is that one never needs a number of colors that is greater than the depth.

It is interesting that this exact relationship between the number of colors and the depth does not hold for collections of arcs on a circle. In Figure 10.2, for example, we see a collection of circular arcs that has depth 2 but needs three colors. This is a basic reflection of the fact that in trying to color a collection of circular arcs, one encounters "long-range" obstacles that render the problem much more complex than the coloring problem for intervals on a line. Despite this, we will see that thinking about the simpler problem of coloring intervals will be useful in designing our algorithm for Circular-Arc Coloring.

Transforming to an Interval Coloring Problem We now return to the Circular-Arc Coloring Problem. For now, we will consider a special case of the problem in which, for each edge e of the cycle, there are exactly k paths that contain e. We will call this the *uniform-depth* case. It turns out that al-

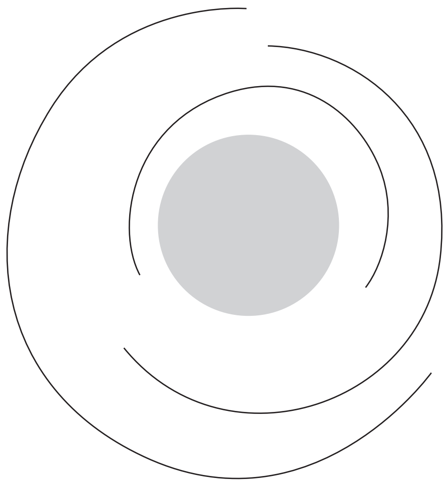

Figure 10.2 A collection of circular arcs needing three colors, even though at most two arcs pass over any point of the circle.

though this special case may seem fairly restricted, it contains essentially the whole complexity of the problem; once we have an algorithm for the uniform-depth case, it will be easy to translate this to an algorithm for the problem in general.

The first step in designing an algorithm will be to transform the instance into a modified form of Interval Coloring: We "cut" the cycle by slicing through the edge (v_n, v_1), and then "unroll" the cycle into a path G'. This process is illustrated in Figure 10.3. The sliced-and-unrolled graph G' has the same nodes as G, plus two extra ones where the slicing occurred: a node v_0 adjacent to v_1 (and no other nodes), and a node v_{n+1} adjacent to v_n (and no other nodes). Also, the set of paths has changed slightly. Suppose that $P_1, P_2, \ldots, P_k$ are the paths that contained the edge (v_n, v_1) in G. Each of these paths P_i has now been sliced into two, one that we'll label P_i' (starting at v_0) and one that we'll label P_i'' (ending at v_{n+1}).

Now this is an instance of Interval Coloring, and it has depth k. Thus, following our discussion above about the relation between depth and colors, we see that the intervals

$$P_1', P_2', \ldots, P_k', P_{k+1}, \ldots, P_m, P_1'', P_2'', \ldots, P_k''$$

can be colored using k colors. So are we done? Can we just translate this solution into a solution for the paths on G?

In fact, this is not so easy; the problem is that our interval coloring may well not have given the paths P_i' and P_i'' the same color. Since these are two

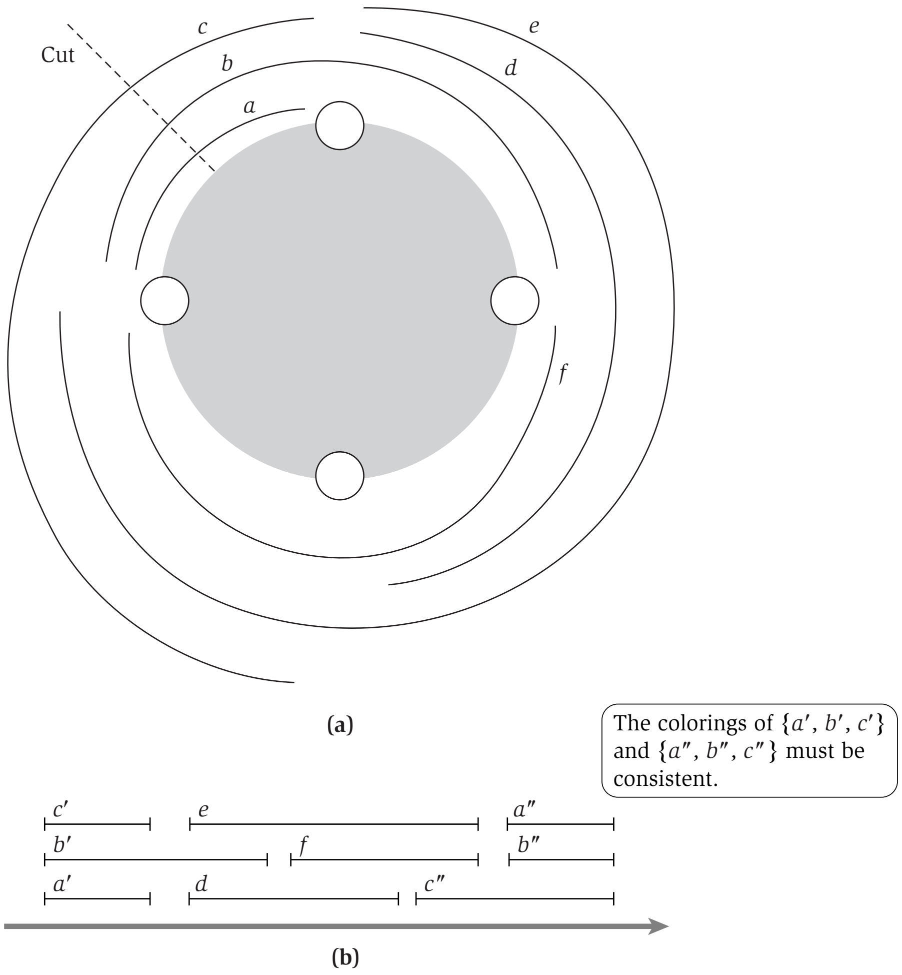
Cut
c
b
a
e
d
f
(a)
The colorings of $\{a', b', c'\}$ and $\{a'', b'', c''\}$ must be consistent.
c′
e
a″
b′
f
b″
a′
d
c″
(b)

Figure 10.3 (a) Cutting through the cycle in an instance of Circular-Arc Coloring, and then unrolling it so it becomes, in (b), a collection of intervals on a line.

pieces of the same path P_i on G, it's not clear how to take the differing colors of P_i' and P_i'' and infer from this how to color P_i on G. For example, having sliced open the cycle in Figure 10.3(a), we get the set of intervals pictured in Figure 10.3(b). Suppose we compute a coloring so that the intervals in the first row get the color 1, those in the second row get the color 2, and those in the third row get the color 3. Then we don't have an obvious way to figure out a color for a and c.

This suggests a way to formalize the relationship between the instance of Circular-Arc Coloring in G and the instance of Interval Coloring in G'.

(10.9) *The paths in G can be k-colored if and only if the paths in G' can be k-colored subject to the additional restriction that P_i' and P_i'' receive the same color, for each $i = 1, 2, \ldots, k$.*

Proof. If the paths in G can be k-colored, then we simply use these as the colors in G', assigning each of P_i' and P_i'' the color of P_i. In the resulting coloring, no two paths with the same color have an edge in common.

Conversely, suppose the paths in G' can be k-colored subject to the additional restriction that P_i' and P_i'' receive the same color, for each $i = 1, 2, \ldots, k$. Then we assign path P_i (for $i \le k$) the common color of P_i' and P_i''; and we assign path P_j (for $j > k$) the color that P_j gets in G'. Again, under this coloring, no two paths with the same color have an edge in common. ■

We've now transformed our problem into a search for a coloring of the paths in G' subject to the condition in (10.9): The paths P_i' and P_i'' (for $1 \le i \le k$) should get the same color.

Before proceeding, we introduce some further terminology that makes it easier to talk about algorithms for this problem. First, since the names of the colors are arbitrary, we can assume that path P_i' is assigned the color i for each $i = 1, 2, \ldots, k$. Now, for each edge $e_i = (v_i, v_{i+1})$, we let S_i denote the set of paths that contain this edge. A k-coloring of just the paths in S_i has a very simple structure: it is simply a way of assigning exactly one of the colors $\{1, 2, \ldots, k\}$ to each of the k paths in S_i. We will think of such a k-coloring as a one-to-one function $f : S_i \to \{1, 2, \ldots, k\}$.

Here's the crucial definition: We say that a k-coloring f of S_i and a k-coloring g of S_j are *consistent* if there is a single k-coloring of all the paths that is equal to f on S_i, and also equal to g on S_j. In other words, the k-colorings f and g on restricted parts of the instance could both arise from a single k-coloring of the whole instance. We can state our problem in terms of consistency as follows: If f' denotes the k-coloring of S_0 that assigns color i to P_i', and f'' denotes the k-coloring of S_n that assigns color i to P_i'', then we need to decide whether f' and f'' are consistent.

Searching for an Acceptable Interval Coloring It is not clear how to decide the consistency of f' and f'' directly. Instead, we adopt a dynamic programming approach by building up the solution through a series of subproblems.

The subproblems are as follows: For each set S_i, working in order over $i = 0, 1, 2, \ldots, n$, we will compute the set F_i of all k-colorings on S_i that are consistent with f'. Once we have computed F_n, we need only check whether it contains f'' in order to answer our overall question: whether f' and f'' are consistent.

To start the algorithm, we define $F_0 = \{f'\}$: Since f' determines a color for every interval in S_0, clearly no other k-coloring of S_0 can be consistent with it. Now suppose we have computed $F_0, F_1, \ldots, F_i$; we show how to compute F_{i+1} from F_i.

Recall that S_i consists of the paths containing the edge $e_i = (v_i, v_{i+1})$, and S_{i+1} consists of the paths containing the next consecutive edge $e_{i+1} = (v_{i+1}, v_{i+2})$. The paths in S_i and S_{i+1} can be divided into three types:

- Those that contain both e_i and e_{i+1}. These lie in both S_i and S_{i+1}.
- Those that end at node v_{i+1}. These lie in S_i but not S_{i+1}.
- Those that begin at node v_{i+1}. These lie in S_{i+1} but not S_i.

Now, for any coloring $f \in F_i$, we say that a coloring g of S_{i+1} is an *extension* of f if all the paths in $S_i \cap S_{i+1}$ have the same colors with respect to f and g. It is easy to check that if g is an extension of f, and f is consistent with f', then so is g. On the other hand, suppose some coloring g of S_{i+1} is consistent with f'; in other words, there is a coloring h of all paths that is equal to f' on S_0 and is equal to g on S_{i+1}. Then, if we consider the colors assigned by h to paths in S_i, we get a coloring $f \in F_i$, and g is an extension of f.

This proves the following fact.

(10.10) *The set F_{i+1} is equal to the set of all extensions of k-colorings in F_i.*

So, in order to compute F_{i+1}, we simply need to list all extensions of all colorings in F_i. For each $f \in F_i$, this means that we want a list of all colorings g of S_{i+1} that agree with f on $S_i \cap S_{i+1}$. To do this, we simply list all possible ways of assigning the colors of $S_i - S_{i+1}$ (with respect to f) to the paths in $S_{i+1} - S_i$. Merging these lists for all $f \in F_i$ then gives us F_{i+1}.

Thus the overall algorithm is as follows.

```
To determine whether $f'$ and $f''$ are consistent:
  Define $F_0 = \{f'\}$
  For $i = 1, 2, \ldots, n$
    For each $f \in F_i$
      Add all extensions of $f$ to $F_{i+1}$
    Endfor
  Endfor
  Check whether $f''$ is in $F_n$
```

Figure 10.4 shows the results of executing this algorithm on the example of Figure 10.3. As with all the dynamic programming algorithms we have seen in this book, the actual coloring can be computed by tracing back through the steps that built up the sets $F_1, F_2, \ldots, F_n$.

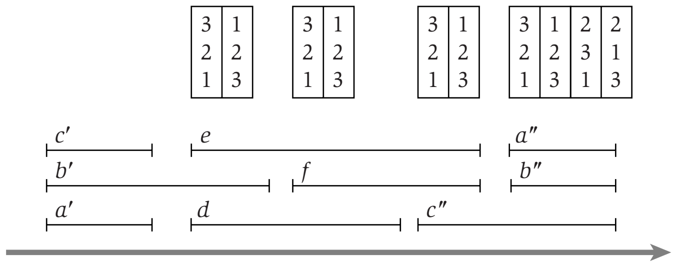

Figure 10.4 The execution of the coloring algorithm. The initial coloring f' assigns color 1 to a', color 2 to b', and color 3 to c'. Above each edge e_i (for $i > 0$) is a table representing the set of all consistent colorings in F_i: Each coloring is represented by one of the columns in the table. Since the coloring $f''(a'') = 1$, $f''(b'') = 2$, and $f''(c'') = 3$ appears in the final table, there is a solution to this instance.

We will discuss the running time of this algorithm in a moment. First, however, we show how to remove the assumption that the input instance has uniform depth.

Removing the Uniform-Depth Assumption Recall that the algorithm we just designed assumes that for each edge e, exactly k paths contain e. In general, each edge may carry a different number of paths, up to a maximum of k. (If there were an edge contained in $k + 1$ paths, then all these paths would need a different color, and so we could immediately conclude that the input instance is not colorable with k colors.)

It is not hard to modify the algorithm directly to handle the general case, but it is also easy to reduce the general case to the uniform-depth case. For each edge e_i that carries only $k_i < k$ paths, we add $k - k_i$ paths that consist only of the single edge e_i. We now have a uniform-depth instance, and we claim

(10.11) *The original instance can be colored with k colors if and only if the modified instance (obtained by adding single-edge paths) can be colored with k colors.*

Proof. Clearly, if the modified instance has a k-coloring, then we can use this same k-coloring for the original instance (simply ignoring the colors it assigns to the single-edge paths that we added). Conversely, suppose the original instance has a k-coloring f. Then we can construct a k-coloring of the modified instance by starting with f and considering the extra single-edge paths one at a time, assigning any free color to each of these paths as we consider them. ■

Analyzing the Algorithm

Finally, we bound the running time of the algorithm. This is dominated by the time to compute the sets $F_1, F_2, \ldots, F_n$. To build one of these sets F_{i+1}, we need to consider each coloring $f \in F_i$, and list all permutations of the colors that f assigns to paths in $S_i - S_{i+1}$. Since S_i has k paths, the number of colorings in F_i is at most $k!$. Listing all permutations of the colors that f assigns to $S_i - S_{i+1}$ also involves enumerating a set of size $\ell!$, where $\ell \leq k$ is the size of $S_i - S_{i+1}$.

Thus the total time to compute F_{i+1} from one F_i has the form $O(f(k))$ for a function $f(\cdot)$ that depends only on k. Over the n iterations of the outer loop to compute $F_1, F_2, \ldots, F_n$, this gives a total running time of $O(f(k) \cdot n)$, as desired.

This concludes the description and analysis of the algorithm. We summarize its properties in the following statement.

(10.12) *The algorithm described in this section correctly determines whether a collection of paths on an n-node cycle can be colored with k colors, and its running time is $O(f(k) \cdot n)$ for a function $f(\cdot)$ that depends only on k.*

Looking back on it, then, we see that the running time of the algorithm came from the intuition we described at the beginning of the section: For each i, the subproblems based on computing F_i and F_{i+1} fit together along the "narrow" interface consisting of the paths in just S_i and S_{i+1}, each of which has size at most k. Thus the time needed to go from one to the other could be made to depend only on k, and not on the size of the cycle G or on the number of paths.

* 10.4 Tree Decompositions of Graphs

In the previous two sections, we've seen how particular NP-hard problems (specifically, Maximum-Weight Independent Set and Graph Coloring) can be solved when the input has a restricted structure. When you find yourself in this situation—able to solve an NP-complete problem in a reasonably natural special case—it's worth asking why the approach doesn't work in general. As we discussed in Sections 10.2 and 10.3, our algorithms in both cases were taking advantage of a particular kind of structure: the fact that the input could be broken down into subproblems with very limited interaction.

For example, to solve Maximum-Weight Independent Set on a tree, we took advantage of a special property of (rooted) trees: Once we decide whether or not to include a node u in the independent set, the subproblems in each subtree become completely separated; we can solve each as though the others did not exist. We don't encounter such a nice situation in general graphs, where there might not be a node that "breaks the communication" between subproblems in the rest of the graph. Rather, for the Independent Set Problem in general graphs, decisions we make in one place seem to have complex repercussions all across the graph.

So we can ask a weaker version of our question instead: For how general a class of graphs can we use this notion of "limited interaction"—recursively chopping up the input using small sets of nodes—to design efficient algorithms for a problem like Maximum-Weight Independent Set?

In fact, there is a natural and rich class of graphs that supports this type of algorithm; they are essentially "generalized trees," and for reasons that will become clear shortly, we will refer to them as *graphs of bounded tree-width*. Just as with trees, many NP-complete problems are tractable on graphs of bounded tree-width; and the class of graphs of bounded tree-width turns out to have considerable practical value, since it includes many real-world networks on which NP-complete graph problems arise. So, in a sense, this type of graph serves as a nice example of finding the "right" special case of a problem that simultaneously allows for efficient algorithms and also includes graphs that arise in practice.

In this section, we define tree-width and give the general approach for solving problems on graphs of bounded tree-width. In the next section, we discuss how to tell whether a given graph has bounded tree-width.

Defining Tree-Width

We now give a precise definition for this class of graphs that is designed to generalize trees. The definition is motivated by two considerations. First, we want to find graphs that we can decompose into disconnected pieces by removing a small number of nodes; this allows us to implement dynamic programming algorithms of the type we discussed earlier. Second, we want to make precise the intuition conveyed by "tree-like" drawings of graphs as in Figure 10.5(b).

We want to claim that the graph G pictured in this figure is decomposable in a tree-like way, along the lines that we've been considering. If we were to encounter G as it is drawn in Figure 10.5(a), it might not be immediately clear why this is so. In the drawing in Figure 10.5(b), however, we see that G is really composed of ten interlocking triangles; and seven of the ten triangles have the property that if we delete them, then the remainder of G falls apart into disconnected pieces that recursively have this interlocking-triangle structure. The other three triangles are attached at the extremities, and deleting them is sort of like deleting the leaves of a tree.

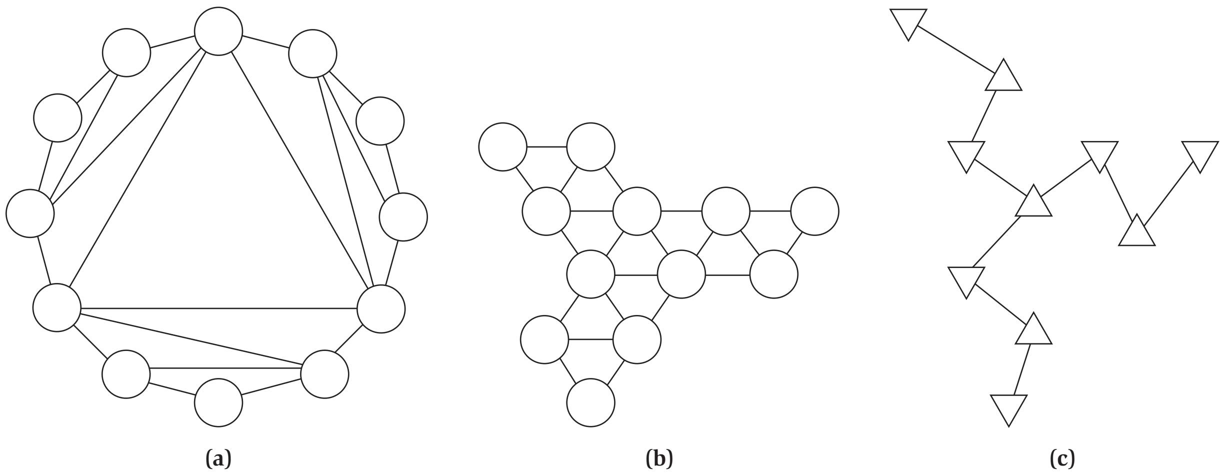

Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

So G is tree-like if we view it not as being composed of twelve nodes, as we usually would, but instead as being composed of ten triangles. Although G clearly contains many cycles, it seems, intuitively, to lack cycles when viewed at the level of these ten triangles; and based on this, it inherits many of the nice decomposition properties of a tree.

We will want to represent the tree-like structure of these triangles by having each triangle correspond to a node in a tree, as shown in Figure 10.5(c). Intuitively, the tree in this figure corresponds to this graph, with each node of the tree representing one of the triangles. Notice, however, that the same nodes of the graph occur in multiple triangles, even in triangles that are not adjacent in the tree structure; and there are edges between nodes in triangles very far away in the tree-structure—for example, the central triangle has edges to nodes in every other triangle. How can we make the correspondence between the tree and the graph precise? We do this by introducing the idea of a *tree decomposition* of a graph G, so named because we will seek to decompose G according to a tree-like pattern.

Formally, a tree decomposition of $G = (V, E)$ consists of a tree T (on a different node set from G), and a subset $V_t \subseteq V$ associated with each node t of T. (We will call these subsets V_t the "pieces" of the tree decomposition.) We will sometimes write this as the ordered pair $(T, \{V_t : t \in T\})$. The tree T and the collection of pieces $\{V_t : t \in T\}$ must satisfy the following three properties.

- *(Node Coverage)* Every node of G belongs to at least one piece V_t.
- *(Edge Coverage)* For every edge e of G, there is some piece V_t containing both ends of e.
- *(Coherence)* Let t_1, t_2, and t_3 be three nodes of T such that t_2 lies on the path from t_1 to t_3. Then, if a node v of G belongs to both V_{t_1} and V_{t_3}, it also belongs to V_{t_2}.

It's worth checking that the tree in Figure 10.5(c) is a tree decomposition of the graph using the ten triangles as the pieces.

Next consider the case when the graph G is a tree. We can build a tree decomposition of it as follows. The decomposition tree T has a node t_v for each node v of G, and a node t_e for each edge e of G. The tree T has an edge (t_v, t_e) when v is an end of e. Finally, if v is a node, then we define the piece $V_{t_v} = \{v\}$; and if $e = (u, v)$ is an edge, then we define the piece $V_{t_e} = \{u, v\}$. One can now check that the three properties in the definition of a tree decomposition are satisfied.

Properties of a Tree Decomposition

If we consider the definition more closely, we see that the Node Coverage and Edge Coverage Properties simply ensure that the collection of pieces corresponds to the graph G in a minimal way. The crux of the definition is in the Coherence Property. While it is not obvious from its statement that Coherence leads to tree-like separation properties, in fact it does so quite naturally. Trees have two nice separation properties, closely related to each other, that get used all the time. One says that if we delete an edge e from a tree, it falls apart into exactly two connected components. The other says that if we delete a node t from a tree, then this is like deleting all the incident edges, and so the tree falls apart into a number of components equal to the degree of t. The Coherence Property is designed to guarantee that separations of T, of both these types, correspond naturally to separations of G as well.

If T' is a subgraph of T, we use $G_{T'}$ to denote the subgraph of G induced by the nodes in all pieces associated with nodes of T', that is, the set $\cup_{t \in T'} V_t$.

First consider deleting a node t of T.

(10.13) *Suppose that $T - t$ has components $T_1, \ldots, T_d$. Then the subgraphs*

$$G_{T_1} - V_t, G_{T_2} - V_t, \ldots, G_{T_d} - V_t$$

have no nodes in common, and there are no edges between them.

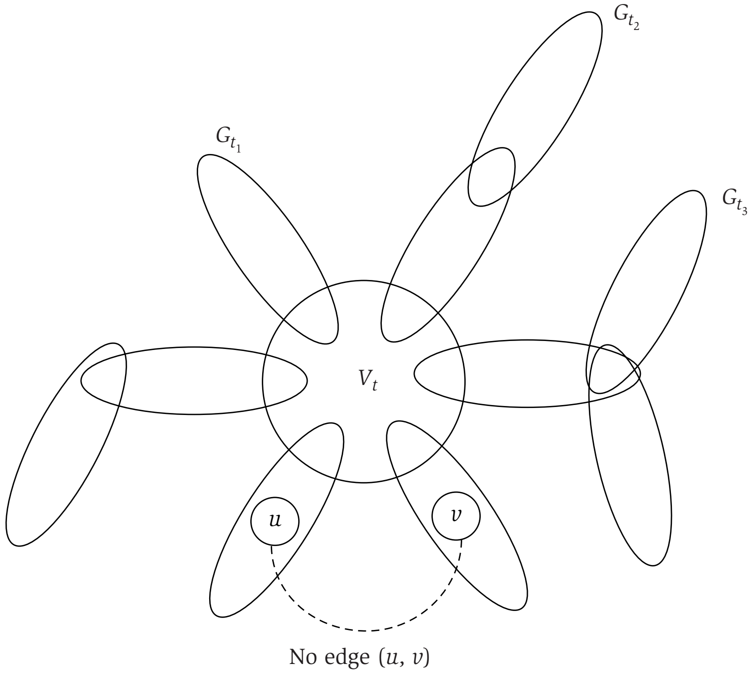

Figure 10.6 Separations of the tree T translate to separations of the graph G.

Proof. We refer to Figure 10.6 for a general view of what the separation looks like. We first prove that the subgraphs $G_{T_i} - V_t$ do not share any nodes. Indeed, any such node v would need to belong to both $G_{T_i} - V_t$ and $G_{T_j} - V_t$ for some $i \neq j$, and so such a node v belongs to some piece V_x with $x \in T_i$, and to some piece V_y with $y \in T_j$. Since t lies on the x-y path in T, it follows from the Coherence Property that v lies in V_t and hence belongs to neither $G_{T_i} - V_t$ nor $G_{T_j} - V_t$.

Next we must show that there is no edge $e = (u, v)$ in G with one end u in subgraph $G_{T_i} - V_t$ and the other end v in $G_{T_j} - V_t$ for some $j \neq i$. If there were such an edge, then by the Edge Coverage Property, there would need to be some piece V_x containing both u and v. The node x cannot be in both the subgraphs T_i and T_j. Suppose by symmetry $x \notin T_i$. Node u is in the subgraph G_{T_i}, so u must be in a set V_y for some y in T_i. Then the node u belongs to both V_x and V_y, and since t lies on the x-y path in T, it follows that u also belongs to V_t, and so it does not lie in $G_{T_i} - V_t$ as required. ■

Proving the edge separation property is analogous. If we delete an edge (x, y) from T, then T falls apart into two components: X, containing x, and Y,

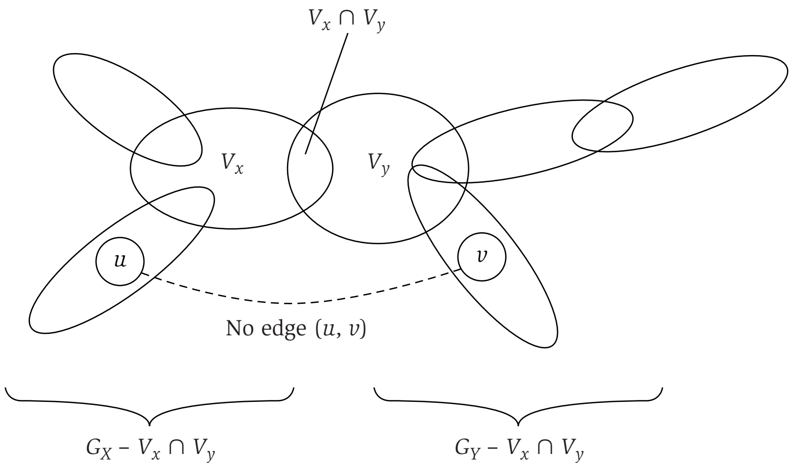

Figure 10.7 Deleting an edge of the tree T translates to separation of the graph G.

containing y. Let's establish the corresponding way in which G is separated by this operation.

(10.14) *Let X and Y be the two components of T after the deletion of the edge (x, y). Then deleting the set $V_x \cap V_y$ from V disconnects G into the two subgraphs $G_X - (V_x \cap V_y)$ and $G_Y - (V_x \cap V_y)$ More precisely, these two subgraphs do not share any nodes, and there is no edge with one end in each of them.*

Proof. We refer to Figure 10.7 for a general view of what the separation looks like. The proof of this property is analogous to the proof of (10.13). One first proves that the two subgraphs $G_X - (V_x \cap V_y)$ and $G_Y - (V_x \cap V_y)$ do not share any nodes, by showing that a node v that belongs to both G_X and G_Y must belong to both V_x and to V_y, and hence it does not lie in either $G_Y - (V_x \cap V_y)$ or $G_X - (V_x \cap V_y)$.

Now we must show that there is no edge $e = (u, v)$ in G with one end u in $G_X - (V_x \cap V_y)$ and the other end v in $G_Y - (V_x \cap V_y)$. If there were such an edge, then by the Edge Coverage Property, there would need to be some piece V_z containing both u and v. Suppose by symmetry that $z \in X$. Node v also belongs to some piece V_w for $w \in Y$. Since x and y lie on the w-z path in T, it follows that V belongs to V_x and V_y. Hence $v \in V_x \cap V_y$, and so it does not lie in $G_Y - (V_x \cap V_y)$ as required. ■

So tree decompositions are useful in that the separation properties of T carry over to G. At this point, one might think that the key question is: Which graphs have tree decompositions? But this is not the point, for if we think about it, we see that of course every graph has a tree decomposition. Given any G, we can let T be a tree consisting of a single node t, and let the single piece V_t be equal to the entire node set of G. This easily satisfies the three properties required by the definition; and such a tree decomposition is no more useful to us than the original graph.

The crucial point, therefore, is to look for a tree decomposition in which all the pieces are *small*. This is really what we're trying to carry over from trees, by requiring that the deletion of a very small set of nodes breaks apart the graph into disconnected subgraphs. So we define the *width* of a tree decomposition $(T, \{V_t\})$ to be one less than the maximum size of any piece V_t:

$$\text{width}(T, \{V_t\}) = \max_t |V_t| - 1.$$

We then define the *tree-width* of G to be the minimum width of any tree decomposition of G. Due to the Edge Coverage Property, all tree decompositions must have pieces with at least two nodes, and hence have tree-width at least 1. Recall that our tree decomposition for a tree G has tree-width 1, as the sets V_t each have either one or two nodes. The somewhat puzzling "−1" in this definition is so that trees turn out to have tree-width 1, rather than 2. Also, all graphs with a nontrivial tree decomposition of tree-width w have separators of size w, since if (x, y) is an edge of the tree, then, by (10.14), deleting $V_x \cap V_y$ separates G into two components.

Thus we can talk about the set of all graphs of tree-width 1, the set of all graphs of tree-width 2, and so forth. The following fact establishes that trees are the only graphs with tree-width 1, and hence our definitions here indeed generalize the notion of a tree. The proof also provides a good way for us to exercise some of the basic properties of tree decompositions. We also observe that the graph in Figure 10.5 is thus, according to the notion of tree-width, a member of the next "simplest" class of graphs after trees: It is a graph of tree-width 2.

(10.15) *A connected graph G has tree-width* 1 *if and only if it is a tree.*

Proof. We have already seen that if G is a tree, then we can build a tree decomposition of tree-width 1 for G.

To prove the converse, we first establish the following useful fact: If H is a subgraph of G, then the tree-width of H is at most the tree-width of G. This is simply because, given a tree decomposition $(T, \{V_t\})$ of G, we can define a tree decomposition of H by keeping the same underlying tree T and replacing each piece V_t with $V_t \cap H$. It is easy to check that the required three properties still hold. (The fact that certain pieces may now be equal to the empty set does not pose a problem.)

Now suppose by way of contradiction that G is a connected graph of tree-width 1 that is not a tree. Since G is not a tree, it has a subgraph consisting of a simple cycle C. By our argument from the previous paragraph, it is now enough for us to argue that the graph C does not have tree-width 1. Indeed, suppose it had a tree decomposition $(T, \{V_t\})$ in which each piece had size at most 2. Choose any two edges (u, v) and (u', v') of C; by the Edge Coverage Property, there are pieces V_t and $V_{t'}$ containing them. Now, on the path in T from t to t' there must be an edge (x, y) such that the pieces V_x and V_y are unequal. It follows that $|V_x \cap V_y| \le 1$. We now invoke (10.14): Defining X and Y to be the components of $T - (x, y)$ containing x and y, respectively, we see that deleting $V_x \cap V_y$ separates C into $C_X - (V_x \cap V_y)$ and $C_Y - (V_x \cap V_y)$. Neither of these two subgraphs can be empty, since one contains $\{u, v\} - (V_x \cap V_y)$ and the other contains $\{u', v'\} - (V_x \cap V_y)$. But it is not possible to disconnect a cycle into two nonempty subgraphs by deleting a single node, and so this yields a contradiction. ■

When we use tree decompositions in the context of dynamic programming algorithms, we would like, for the sake of efficiency, that they not have too many pieces. Here is a simple way to do this. If we are given a tree decomposition $(T, \{V_t\})$ of a graph G, and we see an edge (x, y) of T such that $V_x \subseteq V_y$, then we can contract the edge (x, y) (folding the piece V_x into the piece V_y) and obtain a tree decomposition of G based on a smaller tree. Repeating this process as often as necessary, we end up with a *nonredundant tree decomposition*: There is no edge (x, y) of the underlying tree such that $V_x \subseteq V_y$.

Once we've reached such a tree decomposition, we can be sure that it does not have too many pieces:

(10.16) *Any nonredundant tree decomposition of an n-node graph has at most n pieces.*

Proof. We prove this by induction on n, the case $n = 1$ being clear. Let's consider the case in which $n > 1$. Given a nonredundant tree decomposition $(T, \{V_t\})$ of an n-node graph, we first identify a leaf t of T. By the nonredundancy condition, there must be at least one node in V_t that does not appear in the neighboring piece, and hence (by the Coherence Property) does not appear in any other piece. Let U be the set of all such nodes in V_t. We now observe that by deleting t from T, and removing V_t from the collection of pieces, we obtain a nonredundant tree decomposition of $G - U$. By our inductive hypothesis, this tree decomposition has at most $n - |U| \le n - 1$ pieces, and so $(T, \{V_t\})$ has at most n pieces. ■

While (10.16) is very useful for making sure one has a small tree decomposition, it is often easier in the course of analyzing a graph to start by building a redundant tree decomposition, and only later "condensing" it down to a nonredundant one. For example, our tree decomposition for a graph G that is a tree built a redundant tree decomposition; it would not have been as simple to directly describe a nonredundant one.

Having thus laid the groundwork, we now turn to the algorithmic uses of tree decompositions.

Dynamic Programming over a Tree Decomposition

We began by claiming that the Maximum-Weight Independent Set could be solved efficiently on any graph for which the tree-width was bounded. Now it's time to deliver on this promise. Specifically, we will develop an algorithm that closely follows the linear-time algorithm for trees. Given an n-node graph with an associated tree decomposition of width w, it will run in time $O(f(w) \cdot n)$, where $f(\cdot)$ is an exponential function that depends only on the width w, not on the number of nodes n. And, as in the case of trees, although we are focusing on Maximum-Weight Independent Set, the approach here is useful for many NP-hard problems.

So, in a very concrete sense, the complexity of the problem has been pushed off of the size of the graph and into the tree-width, which may be much smaller. As we mentioned earlier, large networks in the real world often have very small tree-width; and often this is not coincidental, but a consequence of the structured or modular way in which they are designed. So, if we encounter a 1,000-node network with a tree decomposition of width 4, the approach discussed here takes a problem that would have been hopelessly intractable and makes it potentially quite manageable.

Of course, this is all somewhat reminiscent of the Vertex Cover Algorithm from Section 10.1. There we pushed the exponential complexity into the parameter k, the size of the vertex cover being sought. Here we did not have an obvious parameter other than n lying around, so we were forced to invent a fairly nonobvious one: the tree-width.

To design the algorithm, we recall what we did for the case of a tree T. After rooting T, we built the independent set by working our way up from the leaves. At each internal node u, we enumerated the possibilities for what to do with u—include it or not include it—since once this decision was fixed, the problems for the different subtrees below u became independent.

The generalization for a graph G with a tree decomposition $(T, \{V_t\})$ of width w looks very similar. We root the tree T and build the independent set by considering the pieces V_t from the leaves upward. At an internal node t

of T, we confront the following basic question: The optimal independent set intersects the piece V_t in some subset U, but we don't know which set U it is. So we enumerate all the possibilities for this subset U—that is, all possibilities for which nodes to include from V_t and which to leave out. Since V_t may have size up to $w+1$, this may be 2^{w+1} possibilities to consider. But we now can exploit two key facts: first, that the quantity 2^{w+1} is a lot more reasonable than 2^n when w is much smaller than n; and second, that once we fix a particular one of these 2^{w+1} possibilities—once we've decided which nodes in the piece V_t to include—the separation properties (10.13) and (10.14) ensure that the problems in the different subtrees of T below t can be solved independently. So, while we settle for doing brute-force search at the level of a *single* piece, we have an algorithm that is quite efficient at the global level when the individual pieces are small.

Defining the Subproblems More precisely, we root the tree T at a node r. For any node t, let T_t denote the subtree rooted at t. Recall that G_{T_t} denotes the subgraph of G induced by the nodes in all pieces associated with nodes of T_t; for notational simplicity, we will also write this subgraph as G_t. For a subset U of V, we use $w(U)$ to denote the total weight of nodes in U; that is, $w(U) = \sum_{u \in U} w_u$.

We define a set of subproblems for each subtree T_t, one corresponding to each possible subset U of V_t that may represent the intersection of the optimal solution with V_t. Thus, for each independent set $U \subseteq V_t$, we write $f_t(U)$ to denote the maximum weight of an independent set S in G_t, subject to the requirement that $S \cap V_t = U$. The quantity $f_t(U)$ is undefined if U is not an independent set, since in this case we know that U cannot represent the intersection of the optimal solution with V_t.

There are at most 2^{w+1} subproblems associated with each node t of T, since this is the maximum possible number of independent subsets of V_t. By (10.16), we can assume we are working with a tree decomposition that has at most n pieces, and hence there are a total of at most $2^{w+1}n$ subproblems overall. Clearly, if we have the solutions to all these subproblems, we can determine the maximum weight of an independent set in G by looking at the subproblems associated with the root r: We simply take the maximum, over all independent sets $U \subseteq V_r$, of $f_r(U)$.

Building Up Solutions Now we must show how to build up the solutions to these sub-problems via a recurrence. It's easy to get started: When t is a leaf, $f_t(U)$ is equal to $w(U)$ for each independent set $U \subseteq V_t$.

Now suppose that t has children $t_1, \ldots, t_d$, and we have already determined the values of $f_{t_i}(W)$ for each child t_i and each independent set $W \subseteq V_{t_i}$. How do we determine the value of $f_t(U)$ for an independent set $U \subseteq V_t$?

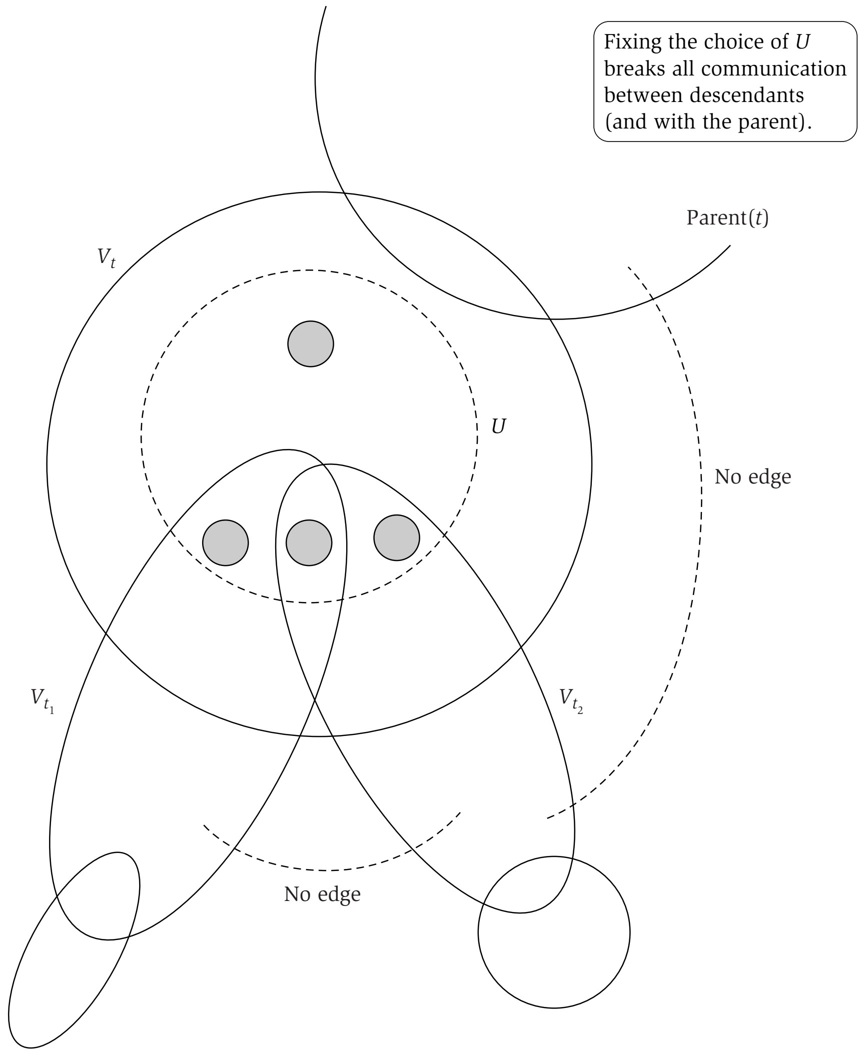
Fixing the choice of U breaks all communication between descendants (and with the parent).
Parent(t)
V_t
U
No edge
V_{t_1}
V_{t_2}
No edge

Figure 10.8 The subproblem $f_t(U)$ in the subgraph G_t. In the optimal solution to this subproblem, we consider independent sets S_i in the descendant subgraphs G_{t_i}, subject to the constraint that $S_i \cap V_t = U \cap V_{t_i}$.

Let S be the maximum-weight independent set in G_t subject to the requirement that $S \cap V_t = U$; that is, $w(S) = f_t(U)$. The key is to understand how this set S looks when intersected with each of the subgraphs G_{t_i}, as suggested in Figure 10.8. We let S_i denote the intersection of S with the nodes of G_{t_i}.

(10.17) *S_i is a maximum-weight independent set of G_{t_i}, subject to the constraint that $S_i \cap V_t = U \cap V_{t_i}$.*

Proof. Suppose there were an independent set S'_i of G_{t_i} with the property that $S'_i \cap V_t = U \cap V_{t_i}$ and $w(S'_i) > w(S_i)$. Then consider the set $S' = (S - S_i) \cup S'_i$. Clearly $w(S') > w(S)$. Also, it is easy to check that $S' \cap V_t = U$.

We claim that S' is an independent set in G; this will contradict our choice of S as the maximum-weight independent set in G_t subject to $S \cap V_t = U$. For suppose S' is not independent, and let $e = (u, v)$ be an edge with both ends in S'. It cannot be that u and v both belong to S, or that they both belong to S'_i, since these are both independent sets. Thus we must have $u \in S - S'_i$ and $v \in S'_i - S$, from which it follows that u is not a node of G_{t_i} while $v \in G_{t_i} - (V_t \cap V_{t_i})$. But then, by (10.14), there cannot be an edge joining u and v. ■

Statement (10.17) is exactly what we need to design a recurrence relation for our subproblems. It says that the information needed to compute $f_t(U)$ is implicit in the values already computed for the subtrees. Specifically, for each child t_i, we need simply determine the value of the maximum-weight independent set S_i of G_{t_i}, subject to the constraint that $S_i \cap V_t = U \cap V_{t_i}$. This constraint does not completely determine what $S_i \cap V_{t_i}$ should be; rather, it says that it can be any independent set $U_i \subseteq V_{t_i}$ such that $U_i \cap V_t = U \cap V_{t_i}$. Thus the weight of the optimal S_i is equal to

$$\max\{f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}.$$

Finally, the value of $f_t(U)$ is simply $w(U)$ plus these maxima added over the d children of t—except that to avoid overcounting the nodes in U, we exclude them from the contribution of the children. Thus we have

(10.18) *The value of $f_t(U)$ is given by the following recurrence:*

$$f_t(U) = w(U) + \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U_i \cap U) :$$
$$U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}.$$

The overall algorithm now just builds up the values of all the subproblems from the leaves of T upward.

```
To find a maximum-weight independent set of $G$,
given a tree decomposition $(T, \{V_t\})$ of $G$:
   Modify the tree decomposition if necessary so it is nonredundant
   Root $T$ at a node $r$
   For each node $t$ of $T$ in post-order
       If $t$ is a leaf then
```

```
            For each independent set $U$ of $V_t$
                $f_t(U) = w(U)$
        Else
            For each independent set $U$ of $V_t$
                $f_t(U)$ is determined by the recurrence in (10.18)
        Endif
    Endfor
    Return max $\{f_r(U) : U \subseteq V_r$ is independent$\}$.
```

An actual independent set of maximum weight can be found, as usual, by tracing back through the execution.

We can determine the time required for computing $f_t(U)$ as follows: For each of the d children t_i, and for each independent set U_i in V_{t_i}, we spend time $O(w)$ checking if $U_i \cap V_t = U \cap V_{t_i}$, to determine whether it should be considered in the computation of (10.18).

This is a total time of $O(2^{w+1}wd)$ for $f_t(U)$; since there are at most 2^{w+1} sets U associated with t, the total time spent on node t is $O(4^{w+1}wd)$. Finally, we sum this over all nodes t to get the total running time. We observe that the sum, over all nodes t, of the number of children of t is $O(n)$, since each node is counted as a child once. Thus the total running time is $O(4^{w+1}wn)$.

* 10.5 Constructing a Tree Decomposition

In the previous section, we introduced the notion of tree decompositions and tree-width, and we discussed a canonical example of how to solve an NP-hard problem on graphs of bounded tree-width.

The Problem

There is still a crucial missing piece in our algorithmic use of tree-width, however. Thus far, we have simply provided an algorithm for Maximum-Weight Independent Set on a graph G, *provided we have been given a low-width tree decomposition of* G. What if we simply encounter G "in the wild," and no one has been kind enough to hand us a good tree decomposition of it? Can we compute one on our own, and then proceed with the dynamic programming algorithm?

The answer is basically yes, with some caveats. First we must warn that, given a graph G, it is NP-hard to determine its tree-width. However, the situation for us is not actually so bad, because we are only interested here in graphs for which the tree-width is a small constant. And, in this case, we will describe an algorithm with the following guarantee: Given a graph G of tree-width less than w, it will produce a tree decomposition of G of width less than $4w$ in time $O(f(w) \cdot mn)$, where m and n are the number of edges and nodes of G, and $f(\cdot)$ is a function that depends only on w. So, essentially, when the tree-width is small, there's a reasonably fast way to produce a tree decomposition whose width is almost as small as possible.

Designing and Analyzing the Algorithm

An Obstacle to Low Tree-Width The first step in designing an algorithm for this problem is to work out a reasonable "obstacle" to a graph G having low tree-width. In other words, as we try to construct a tree decomposition of low width for $G = (V, E)$, might there be some "local" structure we could discover that will tell us the tree-width must in fact be large?

The following idea turns out to provide us with such an obstacle. First, given two sets $Y, Z \subseteq V$ of the same size, we say they are *separable* if some strictly smaller set can completely disconnect them—specifically, if there is a set $S \subseteq V$ such that $|S| < |Y| = |Z|$ and there is no path from $Y - S$ to $Z - S$ in $G - S$. (In this definition, Y and Z need not be disjoint.) Next we say that a set X of nodes in G is *w-linked* if $|X| \geq w$ and X does not contain separable subsets Y and Z, such that $|Y| = |Z| \leq w$.

For later algorithmic use of w-linked sets, we make note of the following fact.

(10.19) *Let $G = (V, E)$ have m edges, let X be a set of k nodes in G, and let $w \leq k$ be a given parameter. Then we can determine whether X is w-linked in time $O(f(k) \cdot m)$, where $f(\cdot)$ depends only on k. Moreover, if X is not w-linked, we can return a proof of this in the form of sets $Y, Z \subseteq X$ and $S \subseteq V$ such that $|S| < |Y| = |Z| \leq w$ and there is no path from $Y - S$ to $Z - S$ in $G - S$.*

Proof. We are trying to decide whether X contains separable subsets Y and Z such that $|Y| = |Z| \leq w$. We can first enumerate all pairs of sufficiently small subsets Y and Z; since X only has 2^k subsets, there are at most 4^k such pairs.

Now, for each pair of subsets Y, Z, we must determine whether they are separable. Let $\ell = |Y| = |Z| \leq w$. But this is exactly the Max-Flow Min-Cut Theorem when we have an undirected graph with capacities on the nodes: Y and Z are separable if and only there do not exist ℓ node-disjoint paths, each with one end in Y and the other in Z. (See Exercise 13 in Chapter 7 for the version of maximum flows with capacities on the nodes.) We can determine whether such paths exist using an algorithm for flow with (unit) capacities on the nodes; this takes time $O(\ell m)$. ■

One should imagine a w-linked set as being highly self-entwined—it has no two small parts that can be easily split off from each other. At the same time, a tree decomposition cuts up a graph using very small separators; and so it is intuitively reasonable that these two structures should be in opposition to each other.

(10.20) *If G contains a $(w+1)$-linked set of size at least $3w$, then G has tree-width at least w.*

Proof. Suppose, by way of contradiction, that G has a $(w+1)$-linked set X of size at least $3w$, and it also has a tree decomposition $(T, \{V_t\})$ of width less than w; in other words, each piece V_t has size at most w. We may further assume that $(T, \{V_t\})$ is nonredundant.

The idea of the proof is to find a piece V_t that is "centered" with respect to X, so that when some part of V_t is deleted from G, one small subset of X is separated from another. Since V_t has size at most w, this will contradict our assumption that X is $(w+1)$-linked.

So how do we find this piece V_t? We first root the tree T at a node r; using the same notation as before, we let T_t denote the subtree rooted at a node t, and write G_t for G_{T_t}. Now let t be a node that is as far from the root r as possible, subject to the condition that G_t contains more than $2w$ nodes of X.

Clearly, t is not a leaf (or else G_t could contain at most w nodes of X); so let $t_1, \ldots, t_d$ be the children of t. Note that since each t_i is farther than t from the root, each subgraph G_{t_i} contains at most $2w$ nodes of X. If there is a child t_i so that G_{t_i} contains at least w nodes of X, then we can define Y to be w nodes of X belonging to G_{t_i}, and Z to be w nodes of X belonging to $G - G_{t_i}$. Since $(T, \{V_t\})$ is nonredundant, $S = V_{t_i} \cap V_t$ has size at most $w - 1$; but by (10.14), deleting S disconnects $Y - S$ from $Z - S$. This contradicts our assumption that X is $(w+1)$-linked.

So we consider the case in which there is no child t_i such that G_{t_i} contains at least w nodes of X; Figure 10.9 suggests the structure of the argument in this case. We begin with the node set of G_{t_1}, combine it with G_{t_2}, then G_{t_3}, and so forth, until we first obtain a set of nodes containing more than w members of X. This will clearly happen by the time we get to G_{t_d}, since G_t contains more than $2w$ nodes of X, and at most w of them can belong to V_t. So suppose our process of combining $G_{t_1}, G_{t_2}, \ldots$ first yields more than w members of X once we reach index $i \le d$. Let W denote the set of nodes in the subgraphs $G_{t_1}, G_{t_2}, \ldots, G_{t_i}$. By our stopping condition, we have $|W \cap X| > w$. But since G_{t_i} contains fewer than w nodes of X, we also have $|W \cap X| < 2w$. Hence we can define Y to be $w+1$ nodes of X belonging to W, and Z to be $w+1$ nodes of X belonging to $V - W$. By (10.13), the piece V_t is now a set of size at most w whose deletion disconnects $Y - V_t$ from $Z - V_t$. Again this contradicts our assumption that X is $(w+1)$-linked, completing the proof. ■

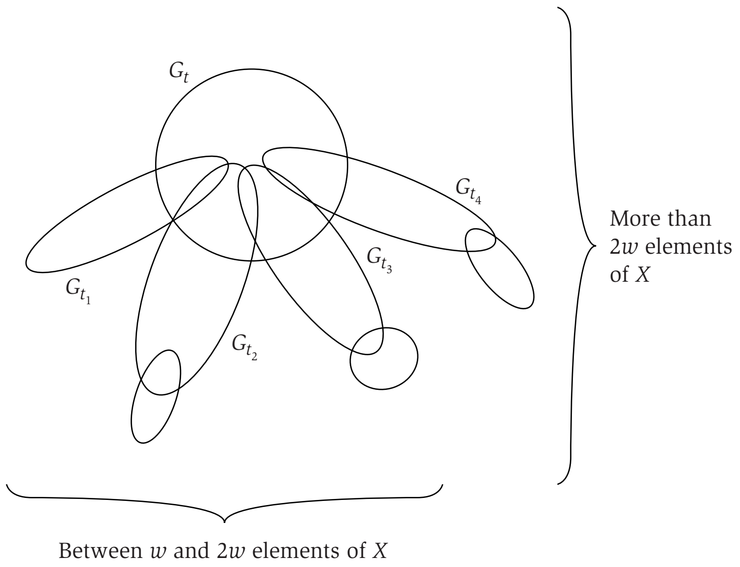
G_t
G_{t_1}
G_{t_2}
G_{t_3}
G_{t_4}
More than $2w$ elements of X
Between w and $2w$ elements of X

Figure 10.9 The final step in the proof of (10.20).

An Algorithm to Search for a Low-Width Tree Decomposition Building on these ideas, we now give a greedy algorithm for constructing a tree decomposition of low width. The algorithm will not precisely determine the tree-width of the input graph $G = (V, E)$; rather, given a parameter w, either it will produce a tree decomposition of width less than $4w$, or it will discover a $(w+1)$-linked set of size at least $3w$. In the latter case, this constitutes a proof that the tree-width of G is at least w, by (10.20); so our algorithm is essentially capable of narrowing down the true tree-width of G to within a factor of 4. As discussed earlier, the running time will have the form $O(f(w) \cdot mn)$, where m and n are the number of edges and nodes of G, and $f(\cdot)$ depends only on w.

Having worked with tree decompositions for a little while now, one can start imagining what might be involved in constructing one for an arbitrary input graph G. The process is depicted at a high level in Figure 10.10. Our goal is to make G fall apart into tree-like portions; we begin the decomposition by placing the first piece V_t anywhere. Now, hopefully, $G - V_t$ consists of several disconnected components; we recursively move into each of these components, placing a piece in each so that it partially overlaps the piece V_t that we've already defined. We hope that these new pieces cause the graph to break up further, and we thus continue in this way, pushing forward with small sets while the graph breaks apart in front of us. The key to making this algorithm work is to argue the following: If at some point we get stuck, and our

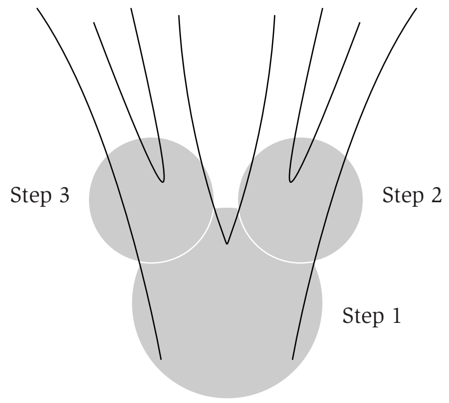

Figure 10.10 A schematic view of the first three steps in the construction of a tree decomposition. As each step produces a new piece, the goal is to break up the remainder of the graph into disconnected components in which the algorithm can continue iteratively.

small sets don't cause the graph to break up any further, then we can extract a large $(w+1)$-linked set that proves the tree-width was in fact large.

Given how vague this intuition is, the actual algorithm follows it more closely than you might expect. We start by assuming that there is no $(w+1)$-linked set of size at least $3w$; our algorithm will produce a tree decomposition provided this holds true, and otherwise we can stop with a proof that the tree-width of G is at least w. We grow the underlying tree T of the decomposition, and the pieces V_t, in a greedy fashion. At every intermediate stage of the algorithm, we will maintain the property that we have a *partial tree decomposition*: by this we mean that if $U \subseteq V$ denotes the set of nodes of G that belong to at least one of the pieces already constructed, then our current tree T and pieces V_t should form a tree decomposition of the subgraph of G induced on U. We define the width of a partial tree decomposition, by analogy with our definition for the width of a tree decomposition, to be one less than the maximum piece size. This means that in order to achieve our goal of having a width of less than $4w$, it is enough to make sure that all pieces have size at most $4w$.

If C is a connected component of $G-U$, we say that $u \in U$ is a *neighbor* of C if there is some node $v \in C$ with an edge to u. The key behind the algorithm is not to simply maintain a partial tree decomposition of width less than $4w$, but also to make sure the following invariant is enforced the whole time:

$(*)$ At any stage in the execution of the algorithm, each component C of $G-U$ has at most $3w$ neighbors, and there is a single piece V_t that contains all of them.

Why is this invariant so useful? It's useful because it will let us add a new node s to T and grow a new piece V_s in the component C, with the confidence that s can be a leaf hanging off t in the larger partial tree decomposition. Moreover, $(*)$ requires there be at most $3w$ neighbors, while we are trying to produce a tree decomposition of width less than $4w$; this extra w gives our new piece "room" to expand by a little as it moves into C.

Specifically, we now describe how to add a new node and a new piece so that we still have a partial tree decomposition, the invariant $(*)$ is still maintained, and the set U has grown strictly larger. In this way, we make at least one node's worth of progress, and so the algorithm will terminate in at most n iterations with a tree decomposition of the whole graph G.

Let C be any component of $G-U$, let X be the set of neighbors of U, and let V_t be a piece that, as guaranteed by $(*)$, contains all of X. We know, again by $(*)$, that X contains at most $3w$ nodes. If X in fact contains strictly fewer than $3w$ nodes, we can make progress right away: For any node $v \in C$ we define a new piece $V_s = X \cup \{v\}$, making s a leaf of t. Since all the edges from v into U have their ends in X, it is easy to confirm that we still have a partial tree decomposition obeying $(*)$, and U has grown.

Thus, let's suppose that X has exactly $3w$ nodes. In this case, it is less clear how to proceed; for example, if we try to create a new piece by arbitrarily adding a node $v \in C$ to X, we may end up with a component of $C-\{v\}$ (which may be all of $C-\{v\}$) whose neighbor set includes all $3w+1$ nodes of $X \cup \{v\}$, and this would violate $(*)$.

There's no simple way around this; for one thing, G may not actually have a low-width tree decomposition. So this is precisely the place where it makes sense to ask whether X poses a genuine obstacle to the tree decomposition or not: we test whether X is a $(w+1)$-linked set. By (10.19), we can determine the answer to this in time $O(f(w) \cdot m)$, since $|X| = 3w$. If it turns out that X is $(w+1)$-linked, then we are all done; we can halt with the conclusion that G has tree-width at least w, which was one acceptable outcome of the algorithm. On the other hand, if X is not $(w+1)$-linked, then we end up with $Y, Z \subseteq X$ and $S \subseteq V$ such that $|S| < |Y| = |Z| \leq w+1$ and there is no path from $Y-S$ to $Z-S$ in $G-S$. The sets Y, Z, and S will now provide us with a means to extend the partial tree decomposition.

Let S' consist of the nodes of S that lie in $Y \cup Z \cup C$. The situation is now as pictured in Figure 10.11. We observe that $S' \cap C$ is not empty: Y and Z each have edges into C, and so if $S' \cap C$ were empty, there would be a path from $Y-S$ to $Z-S$ in $G-S$ that started in Y, jumped immediately into C, traveled through C, and finally jumped back into Z. Also, $|S'| \leq |S| \leq w$.

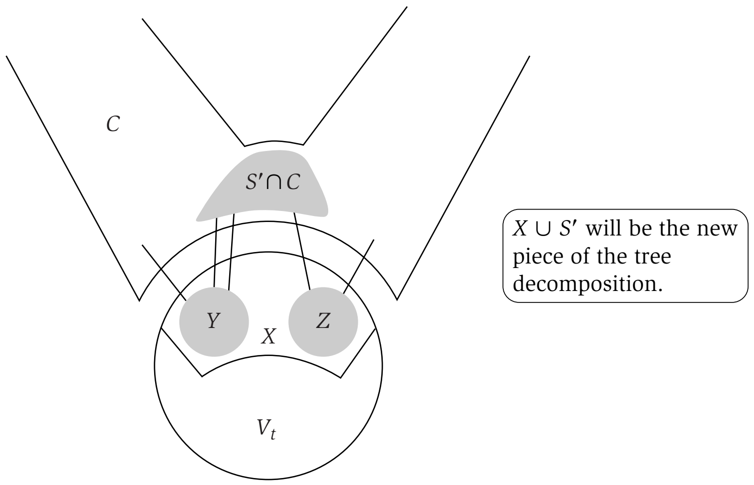

Figure 10.11 Adding a new piece to the partial tree decomposition.

We define a new piece $V_s = X \cup S'$, making s a leaf of t. All the edges from S' into U have their ends in X, and $|X \cup S'| \le 3w + w = 4w$, so we still have a partial tree decomposition. Moreover, the set of nodes covered by our partial tree decomposition has grown, since $S' \cap C$ is not empty. So we will be done if we can show that the invariant $(*)$ still holds. This brings us exactly the intuition we tried to capture when discussing Figure 10.10: As we add the new piece $X \cup S'$, we are hoping that the component C breaks up into further components in a nice way.

Concretely, our partial tree decomposition now covers $U \cup S'$; and where we previously had a component C of $G-U$, we now may have several components $C' \subseteq C$ of $G-(U \cup S')$. Each of these components C' has all its neighbors in $X \cup S'$; but we must additionally make sure there are at most $3w$ such neighbors, so that the invariant $(*)$ continues to hold. So consider one of these components C'. We claim that all its neighbors in $X \cup S'$ actually belong to one of the two subsets $(X-Z) \cup S'$ or $(X-Y) \cup S'$, and each of these sets has size at most $|X| \le 3w$. For, if this did not hold, then C' would have a neighbor in both $Y-S$ and $Z-S$, and hence there would be a path, through C', from $Y-S$ to $Z-S$ in $G-S$. But we have already argued that there cannot be such a path. This establishes that $(*)$ still holds after the addition of the new piece and completes the argument that the algorithm works correctly.

Finally, what is the running time of the algorithm? The time to add a new piece to the partial tree decomposition is dominated by the time required to check whether X is $(w+1)$-linked, which is $O(f(w) \cdot m)$. We do this for at most n iterations, since we increase the number of nodes of G that we cover in each iteration. So the total running time is $O(f(w) \cdot mn)$.

We summarize the properties of our tree decomposition algorithm as follows.

(10.21) *Given a graph G and a parameter w, the tree decomposition algorithm in this section does one of the following two things:*

- *it produces a tree decomposition of width less than $4w$, or*
- *it reports (correctly) that G does not have tree-width less than w.*

The running time of the algorithm is $O(f(w) \cdot mn)$, for a function $f(\cdot)$ that depends only on w.

Solved Exercises

Solved Exercise 1

As we've seen, 3-SAT is often used to model complex planning and decision-making problems in artificial intelligence: the variables represent binary decisions to be made, and the clauses represent constraints on these decisions. Systems that work with instances of 3-SAT often need to represent situations in which some decisions have been made while others are still undetermined, and for this purpose it is useful to introduce the notion of a *partial assignment* of truth values to variables.

Concretely, given a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, we say that a *partial assignment* for X is an assignment of the value 0, 1, or ? to each x_i; in other words, it is a function $\rho : X \rightarrow \{0, 1, ?\}$. We say that a variable x_i is *determined* by the partial assignment if it receives the value 0 or 1, and *undetermined* if it receives the value ?. We can think of a partial assignment as choosing a truth value of 0 or 1 for each of its determined variables, and leaving the truth value of each undetermined variable up in the air.

Now, given a collection of clauses $C_1, \ldots, C_m$, each a disjunction of three distinct terms, we may be interested in whether a partial assignment is sufficient to "force" the collection of clauses to be satisfied, regardless of how we set the undetermined variables. Similarly, we may be interested in whether there exists a partial assignment with only a few determined variables that can force the collection of clauses to be satisfied; this small set of determined variables can be viewed as highly "influential," since their outcomes alone can be enough to force the satisfaction of the clauses.

For example, suppose we are given clauses

$$(x_1 \vee \overline{x_2} \vee \overline{x_4}), (x_2 \vee \overline{x_3} \vee x_4), (\overline{x_2} \vee \overline{x_3} \vee x_5), (x_1 \vee x_3 \vee \overline{x_6}).$$

Then the partial assignment that sets x_1 to 1, sets x_3 to 0, and sets all other variables to ? has only two determined variables, but it forces the collection of clauses to be satisfied: No matter how we set the remaining four variables, the clauses will be satisfied.

Here's a way to formalize this. Recall that a *truth assignment* for X is an assignment of the value 0 or 1 to each x_i; in other words, it must select a truth value for *every* variable and not leave any variables undetermined. We say that a truth assignment ν is *consistent* with a partial assignment ρ if each variable that is determined in ρ has the same truth value in both ρ and ν. (In other words, if $\rho(x_i) \neq ?$, then $\rho(x_i) = \nu(x_i)$.) Finally, we say that a partial assignment ρ *forces* the collection of clauses $C_1, \ldots, C_m$ if, for every truth assignment ν that is consistent with ρ, it is the case that ν satisfies $C_1, \ldots, C_m$. (We will also call ρ a *forcing partial assignment*.)

Motivated by the issues raised above, here's the question. We are given a collection of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, a parameter $b < n$, and a collection of clauses $C_1, \ldots, C_m$ over the variables, where each clause is a disjunction of three distinct terms. We want to decide whether there exists a forcing partial assignment ρ for X, such that at most b variables are determined by ρ. Give an algorithm that solves this problem with a running time of the form $O(f(b) \cdot p(n, m))$, where $p(\cdot)$ is a polynomial function, and $f(\cdot)$ is an arbitrary function that depends only on b, not on n or m.

Solution Intuitively, a forcing partial assignment must "hit" each clause in at least one place, since otherwise it wouldn't be able to ensure the truth value. Although this seems natural, it's not actually part of the definition (the definition just talks about truth assignments that are consistent with the partial assignment), so we begin by formalizing and proving this intuition.

(10.22) *A partial assignment ρ forces all clauses if and only if, for each clause C_i, at least one of the variables in C_i is determined by ρ in a way that satisfies C_i.*

Proof. Clearly, if ρ determines at least one variable in each C_i in a way that satisfies it, then no matter how we construct a full truth assignment for the remaining variables, all the clauses are already satisfied. Thus any truth assignment consistent with ρ satisfies all clauses.

Now, for the converse, suppose there is a clause C_i such that ρ does not determine any of the variables in C_i in a way that satisfies C_i. We want to show that ρ is not forcing, which, according to the definition, requires us to exhibit a consistent truth assignment that does not satisfy all clauses. So consider the following truth assignment ν: ν agrees with ρ on all determined variables, it assigns an arbitrary truth value to each undetermined variable not appearing in C_i, and it sets each undetermined variable in C_i in a way that fails to satisfy it. We observe that ν sets each of the variables in C_i so as not to satisfy it, and hence ν is not a satisfying assignment. But ν is consistent with ρ, and so it follows that ρ is not a forcing partial assignment. ■

In view of (10.22), we have a problem that is very much like the search for small vertex covers at the beginning of the chapter. There we needed to find a set of nodes that covered all edges, and we were limited to choosing at most k nodes. Here we need to find a set of variables that covers all clauses (and with the right true/false values), and we're limited to choosing at most b variables.

So let's try an analogue of the approach we used for finding a small vertex cover. We pick an arbitrary clause C_ℓ, containing x_i, x_j, and x_k (each possibly negated). We know from (10.22) that any forcing assignment ρ must set one of these three variables the way it appears in C_ℓ, and so we can try all three of these possibilities. Suppose we set x_i the way it appears in C_ℓ; we can then eliminate from the instance all clauses (including C_ℓ) that are satisfied by this assignment to x_i, and consider trying to satisfy what's left. We call this smaller set of clauses the instance *reduced by the assignment to* x_i. We can do the same for x_j and x_k. Since ρ must determine one of these three variables the way they appear in C_ℓ, and then still satisfy what's left, we have justified the following analogue of (10.3). (To make the terminology a bit easier to discuss, we say that the *size* of a partial assignment is the number of variables it determines.)

(10.23) *There exists a forcing assignment of size at most b if and only if there is a forcing assignment of size at most* $b-1$ *on at least one of the instances reduced by the assignment to* x_i, x_j, *or* x_k.

We therefore have the following algorithm. (It relies on the boundary cases in which there are no clauses (when by definition we can declare success) and in which there are clauses but $b=0$ (in which case we declare failure).

```
To search for a forcing partial assignment of size at most $b$:
  If there are no clauses, then by definition we have
     a forcing assignment
  Else if $b=0$ then by (10.22) there is no forcing assignment
  Else let $C_\ell$ be an arbitrary clause containing variables $x_i$, $x_j$, $x_k$
    For each of $x_i$, $x_j$, $x_k$:
      Set $x_i$ the way it appears in $C_\ell$
      Reduce the instance by this assignment
      Recursively check for a forcing assignment of size at
        most $b-1$ on this reduced instance
    Endfor
    If any of these recursive calls (say for $x_i$) returns a
      forcing assignment $\rho'$ of size most $b-1$ then
        Combining $\rho'$ with the assignment to $x_i$ is the desired answer
    Else (none of these recursive calls succeeds)
      There is no forcing assignment of size at most $b$
    Endif
  Endif
```

To bound the running time, we consider the tree of possibilities being searched, just as in the algorithm for finding a vertex cover. Each recursive call gives rise to three children in this tree, and this goes on to a depth of at most b. Thus the tree has at most $1+3+3^2+\cdots+3^b \le 3^{b+1}$ nodes, and at each node we spend at most $O(m+n)$ time to produce the reduced instances. Thus the total running time is $O(3^b(m+n))$.

Exercises

1. In Exercise 5 of Chapter 8, we claimed that the Hitting Set Problem was NP-complete. To recap the definitions, consider a set $A=\{a_1,\ldots,a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. We say that a set $H \subseteq A$ is a *hitting set* for the collection $B_1, B_2, \ldots, B_m$ if H contains at least one element from each B_i—that is, if $H \cap B_i$ is not empty for each i. (So H "hits" all the sets B_i.)

   Now suppose we are given an instance of this problem, and we'd like to determine whether there is a hitting set for the collection of size at most k. Furthermore suppose that each set B_i has at most c elements, for a constant c. Give an algorithm that solves this problem with a running time of the form $O(f(c,k)\cdot p(n,m))$, where $p(\cdot)$ is a polynomial function, and $f(\cdot)$ is an arbitrary function that depends only on c and k, not on n or m.

2. The difficulty in 3-SAT comes from the fact that there are 2^n possible assignments to the input variables $x_1, x_2, \ldots, x_n$, and there's no apparent way to search this space in polynomial time. This intuitive picture, however, might create the misleading impression that the fastest algorithms for 3-SAT actually require time 2^n. In fact, though it's somewhat counterintuitive when you first hear it, there are algorithms for 3-SAT that run in significantly less than 2^n time in the worst case; in other words, they

determine whether there's a satisfying assignment in less time than it would take to enumerate all possible settings of the variables.

Here we'll develop one such algorithm, which solves instances of 3-SAT in $O(p(n) \cdot (\sqrt{3})^n)$ time for some polynomial $p(n)$. Note that the main term in this running time is $(\sqrt{3})^n$, which is bounded by 1.74^n.

(a) For a truth assignment Φ for the variables $x_1, x_2, \ldots, x_n$, we use $\Phi(x_i)$ to denote the value assigned by Φ to x_i. (This can be either 0 or 1.) If Φ and Φ' are each truth assignments, we define the *distance* between Φ and Φ' to be the number of variables x_i for which they assign different values, and we denote this distance by $d(\Phi, \Phi')$. In other words, $d(\Phi, \Phi') = |\{i : \Phi(x_i) \neq \Phi'(x_i)\}|$.

A basic building block for our algorithm will be the ability to answer the following kind of question: Given a truth assignment Φ and a distance d, we'd like to know whether there exists a satisfying assignment Φ' such that the distance from Φ to Φ' is at most d. Consider the following algorithm, Explore(Φ, d), that attempts to answer this question.

```
Explore($\Phi$,$d$):
  If $\Phi$ is a satisfying assignment then return "yes"
  Else if $d$ = 0 then return "no"
  Else
    Let $C_i$ be a clause that is not satisfied by $\Phi$
      (i.e., all three terms in $C_i$ evaluate to false)
    Let $\Phi_1$ denote the assignment obtained from $\Phi$ by
      taking the variable that occurs in the first term of
      clause $C_i$ and inverting its assigned value
    Define $\Phi_2$ and $\Phi_3$ analogously in terms of the
      second and third terms of the clause $C_i$
    Recursively invoke:
      Explore($\Phi_1$,$d-1$)
      Explore($\Phi_2$,$d-1$)
      Explore($\Phi_3$,$d-1$)
    If any of these three calls returns "yes"
      then return "yes"
    Else return "no"
```

Prove that Explore(Φ, d) returns "yes" if and only if there exists a satisfying assignment Φ' such that the distance from Φ to Φ' is at most d. Also, give an analysis of the running time of Explore(Φ, d) as a function of n and d.

(b) Clearly any two assignments Φ and Φ' have distance at most n from each other, so one way to solve the given instance of 3-SAT would be to pick an arbitrary starting assignment Φ and then run `Explore(`Φ, n`)`. However, this will not give us the running time we want.

Instead, we will need to make several calls to `Explore`, from different starting points Φ, and search each time out to more limited distances. Describe how to do this in such a way that you can solve the instance of 3-SAT in a running time of only $O(p(n) \cdot (\sqrt{3})^n)$.

3. Suppose we are given a directed graph $G = (V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$, and we want to decide whether G has a Hamiltonian path from v_1 to v_n. (That is, is there a path in G that goes from v_1 to v_n, passing through every other vertex exactly once?)

Since the Hamiltonian Path Problem is NP-complete, we do not expect that there is a polynomial-time solution for this problem. However, this does not mean that all nonpolynomial-time algorithms are equally "bad." For example, here's the simplest brute-force approach: For each permutation of the vertices, see if it forms a Hamiltonian path from v_1 to v_n. This takes time roughly proportional to $n!$, which is about 3×10^{17} when $n = 20$.

Show that the Hamiltonian Path Problem can in fact be solved in time $O(2^n \cdot p(n))$, where $p(n)$ is a polynomial function of n. This is a much better algorithm for moderate values of n; for example, 2^n is only about a million when $n = 20$.

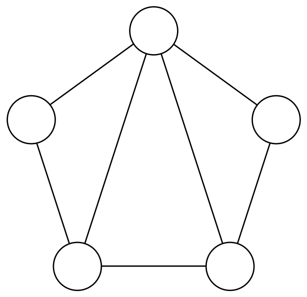
Figure 10.12 A triangulated cycle graph: The edges form the boundary of a convex polygon together with a set of line segments that divide its interior into triangles.

4. We say that a graph $G = (V, E)$ is a *triangulated cycle graph* if it consists of the vertices and edges of a triangulated convex n-gon in the plane—in other words, if it can be drawn in the plane as follows.

The vertices are all placed on the boundary of a convex set in the plane (we may assume on the boundary of a circle), with each pair of consecutive vertices on the circle joined by an edge. The remaining edges are then drawn as straight line segments through the interior of the circle, with no pair of edges crossing in the interior. We require the drawing to have the following property. If we let S denote the set of all points in the plane that lie on vertices or edges of the drawing, then each bounded component of the plane after deleting S is bordered by exactly three edges. (This is the sense in which the graph is a "triangulation.")

A triangulated cycle graph is pictured in Figure 10.12.

Prove that every triangulated cycle graph has a tree decomposition of width at most 2, and describe an efficient algorithm to construct such a decomposition.

5. The *Minimum-Cost Dominating Set Problem* is specified by an undirected graph $G = (V, E)$ and costs $c(v)$ on the nodes $v \in V$. A subset $S \subset V$ is said to be a *dominating set* if all nodes $u \in V - S$ have an edge (u, v) to a node v in S. (Note the difference between dominating sets and vertex covers: in a dominating set, it is fine to have an edge (u, v) with neither u nor v in the set S as long as both u and v have neighbors in S.)

(a) Give a polynomial-time algorithm for the Dominating Set Problem for the special case in which G is a tree.

(b) Give a polynomial-time algorithm for the Dominating Set Problem for the special case in which G has tree-width 2, and we are also given a tree decomposition of G with width 2.

6. The Node-Disjoint Paths Problem is given by an undirected graph G and k pairs of nodes (s_i, t_i) for $i = 1, \ldots, k$. The problem is to decide whether there are node-disjoint paths P_i so that path P_i connects s_i to t_i. Give a polynomial-time algorithm for the Node-Disjoint Paths Problem for the special case in which G has tree-width 2, and we are also given a tree decomposition T of G with width 2.

7. The *chromatic number* of a graph G is the minimum k such that it has a k-coloring. As we saw in Chapter 8, it is NP-complete for $k \geq 3$ to decide whether a given input graph has chromatic number $\leq k$.

(a) Show that for every natural number $w \geq 1$, there is a number $k(w)$ so that the following holds. If G is a graph of tree-width at most w, then G has chromatic number at most $k(w)$. (The point is that $k(w)$ depends only on w, not on the number of nodes in G.)

(b) Given an undirected n-node graph $G = (V, E)$ of tree-width at most w, show how to compute the chromatic number of G in time $O(f(w) \cdot p(n))$, where $p(\cdot)$ is a polynomial but $f(\cdot)$ can be an arbitrary function.

8. Consider the class of 3-SAT instances in which each of the n variables occurs—counting positive and negated appearances combined—in exactly three clauses. Show that any such instance of 3-SAT is in fact satisfiable, and that a satisfying assignment can be found in polynomial time.

9. Give a polynomial-time algorithm for the following problem. We are given a binary tree $T = (V, E)$ with an even number of nodes, and a nonnegative weight on each edge. We wish to find a partition of the nodes V into two sets of *equal* size so that the weight of the cut between the two sets is as large as possible (i.e., the total weight of edges with one end in each set is as large as possible). Note that the restriction that the graph is a tree is crucial here, but the assumption that the tree is binary is not. The problem is NP-hard in general graphs.

Notes and Further Reading

The first topic in this chapter, on how to avoid a running time of $O(kn^{k+1})$ for Vertex Cover, is an example of the general theme of *parameterized complexity*: for problems with two such "size parameters" n and k, one generally prefers running times of the form $O(f(k) \cdot p(n))$, where $p(\cdot)$ is a polynomial, rather than running times of the form $O(n^k)$. A body of work has grown up around this issue, including a methodology for identifying NP-complete problems that are unlikely to allow for such improved running times. This area is covered in the book by Downey and Fellows (1999).

The problem of coloring a collection of circular arcs was shown to be NP-complete by Garey, Johnson, Miller, and Papadimitriou (1980). They also described how the algorithm presented in this chapter follows directly from a construction due to Tucker (1975). Both Interval Coloring and Circular-Arc Coloring belong to the following class of problems: Take a collection of geometric objects (such as intervals or arcs), define a graph by joining pairs of objects that intersect, and study the problem of coloring this graph. The book on graph coloring by Jensen and Toft (1995) includes descriptions of a number of other problems in this style.

The importance of tree decompositions and tree-width was brought into prominence largely through the work of Robertson and Seymour (1990). The algorithm for constructing a tree decomposition described in Section 10.5 is due to Diestel et al. (1999). Further discussion of tree-width and its role in both algorithms and graph theory can be found in the survey by Reed (1997) and the book by Diestel (2000). Tree-width has also come to play an important role in inference algorithms for probabilistic models in machine learning (Jordan 1998).

Notes on the Exercises Exercise 2 is based on a result of Uwe Schöning; and Exercise 8 is based on a problem we learned from Amit Kumar.