21 Minimum Spanning Trees

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, the designer can use an arrangement of $n - 1$ wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

To model this wiring problem, use a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, a weight $w(u, v)$ specifies the cost (amount of wire needed) to connect u and v. The goal is to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph G. We call the problem of determining the tree T the ***minimum-spanning-tree problem***.[1] Figure 21.1 shows an example of a connected graph and a minimum spanning tree.

This chapter studies two ways to solve the minimum-spanning-tree problem. Kruskal's algorithm and Prim's algorithm both run in $O(E \lg V)$ time. Prim's algorithm achieves this bound by using a binary heap as a priority queue. By using Fibonacci heaps instead (see page 478), Prim's algorithm runs in $O(E + V \lg V)$ time. This bound is better than $O(E \lg V)$ whenever $|E|$ grows asymptotically faster than $|V|$.

---

[1] The phrase "minimum spanning tree" is a shortened form of the phrase "minimum-weight spanning tree." There is no point in minimizing the number of edges in T, since all spanning trees have exactly $|V| - 1$ edges by Theorem B.2 on page 1169.

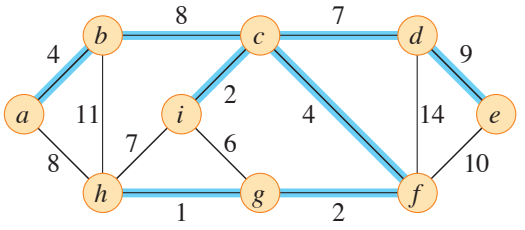

Figure 21.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the blue edges form a minimum spanning tree. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

The two algorithms are greedy algorithms, as described in Chapter 15. Each step of a greedy algorithm must make one of several possible choices. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy does not generally guarantee that it always finds globally optimal solutions to problems. For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight. Although you can read this chapter independently of Chapter 15, the greedy methods presented here are a classic application of the theoretical notions introduced there.

Section 21.1 introduces a "generic" minimum-spanning-tree method that grows a spanning tree by adding one edge at a time. Section 21.2 gives two algorithms that implement the generic method. The first algorithm, due to Kruskal, is similar to the connected-components algorithm from Section 19.1. The second, due to Prim, resembles Dijkstra's shortest-paths algorithm (Section 22.3).

Because a tree is a type of graph, in order to be precise we must define a tree in terms of not just its edges, but its vertices as well. Because this chapter focuses on trees in terms of their edges, we'll implicitly understand that the vertices of a tree T are those that some edge of T is incident on.

21.1 Growing a minimum spanning tree

The input to the minumum-spanning-tree problem is a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. The goal is to find a minimum spanning tree for G. The two algorithms considered in this chapter use a greedy approach to the problem, although they differ in how they apply this approach.

This greedy strategy is captured by the procedure GENERIC-MST on the facing page, which grows the minimum spanning tree one edge at a time. The generic method manages a set A of edges, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Each step determines an edge (u, v) that the procedure can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for A, since it can be added safely to A while maintaining the invariant.

This generic algorithm uses the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A belong to a minimum spanning tree, and the loop must terminate by the time it has considered all edges. Therefore, the set A returned in line 5 must be a minimum spanning tree.

The tricky part is, of course, finding a safe edge in line 3. One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree T such that $A \subseteq T$. Within the **while** loop body, A must be a proper subset of T, and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and (u, v) is safe for A.

The remainder of this section provides a rule (Theorem 21.1) for recognizing safe edges. The next section describes two algorithms that use this rule to find safe edges efficiently.

We first need some definitions. A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. Figure 21.2 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints belongs to S and the other belongs to $V - S$. A cut ***respects*** a set A of edges if no edge in A crosses the cut. An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. There can be more than one light edge crossing a cut in the case of ties. More generally, we say that an edge is a ***light edge*** satisfying a given property if its weight is the minimum of any edge satisfying the property.

The following theorem gives the rule for recognizing safe edges.

Theorem 21.1

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum

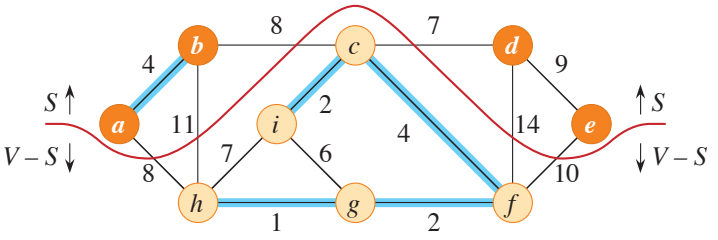

Figure 21.2 A cut $(S, V - S)$ of the graph from Figure 21.1. Orange vertices belong to the set S, and tan vertices belong to $V - S$. The edges crossing the cut are those connecting tan vertices with orange vertices. The edge (d, c) is the unique light edge crossing the cut. Blue edges form a subset A of the edges. The cut $(S, V - S)$ respects A, since no edge of A crosses the cut.

spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Proof Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We'll construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A.

The edge (u, v) forms a cycle with the edges on the simple path p from u to v in T, as Figure 21.3 illustrates. Since u and v are on opposite sides of the cut $(S, V - S)$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.

We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \le w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) . \end{aligned}$$

But T is a minimum spanning tree, so that $w(T) \le w(T')$, and thus, T' must be a minimum spanning tree as well.

It remains to show that (u, v) is actually a safe edge for A. We have $A \subseteq T'$, since $A \subseteq T$ and $(x, y) \notin A$, and thus, $A \cup \{(u, v)\} \subseteq T'$. Consequently, since T' is a minimum spanning tree, (u, v) is safe for A. ■

Theorem 21.1 provides insight into how the GENERIC-MST method works on a connected graph $G = (V, E)$. As the method proceeds, the set A is always acyclic, since it is a subset of a minimum spanning tree and a tree may not contain a cycle.

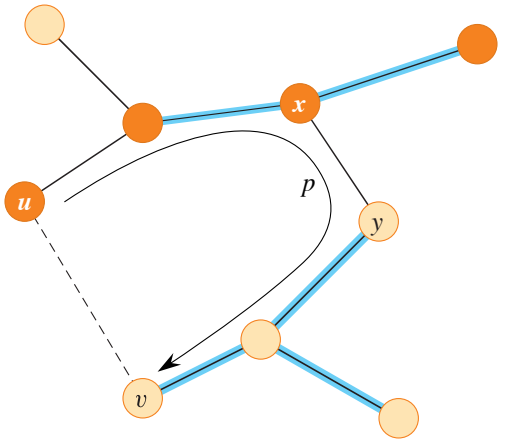

Figure 21.3 The proof of Theorem 21.1. Orange vertices belong to S, and tan vertices belong to $V - S$. Only edges in the minimum spanning tree T are shown, along with edge (u, v), which does not lie in T. The edges in A are blue, and (u, v) is a light edge crossing the cut $(S, V - S)$. The edge (x, y) is an edge on the unique simple path p from u to v in T. To form a minimum spanning tree T' that contains (u, v), remove the edge (x, y) from T and add the edge (u, v).

At any point in the execution, the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree. (Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains $|V|$ trees, one for each vertex.) Moreover, any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

The **while** loop in lines 2–4 of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration. Initially, when $A = \emptyset$, there are $|V|$ trees in G_A, and each iteration reduces that number by 1. When the forest contains only a single tree, the method terminates.

The two algorithms in Section 21.2 use the following corollary to Theorem 21.1.

Corollary 21.2

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Proof The cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A. ■

Exercises

21.1-1

Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

21.1-2

Professor Sabatier conjectures the following converse of Theorem 21.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

21.1-3

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

21.1-4

Give a simple example of a connected graph such that the set of edges $\{(u, v) :$ there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

21.1-5

Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

21.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

21.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

21.1-8
Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

21.1-9
Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

21.1-10
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges in T decreases. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) , \\ w(x, y) - k & \text{if } (u, v) = (x, y) . \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w'.

★ ***21.1-11***
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges *not* in T decreases. Give an algorithm for finding the minimum spanning tree in the modified graph.

21.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section elaborate on the generic method. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST. In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a lowest-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a lowest-weight edge connecting the tree to a vertex not in the tree. Both algorithms assume that the input graph is connected and represented by adjacency lists.

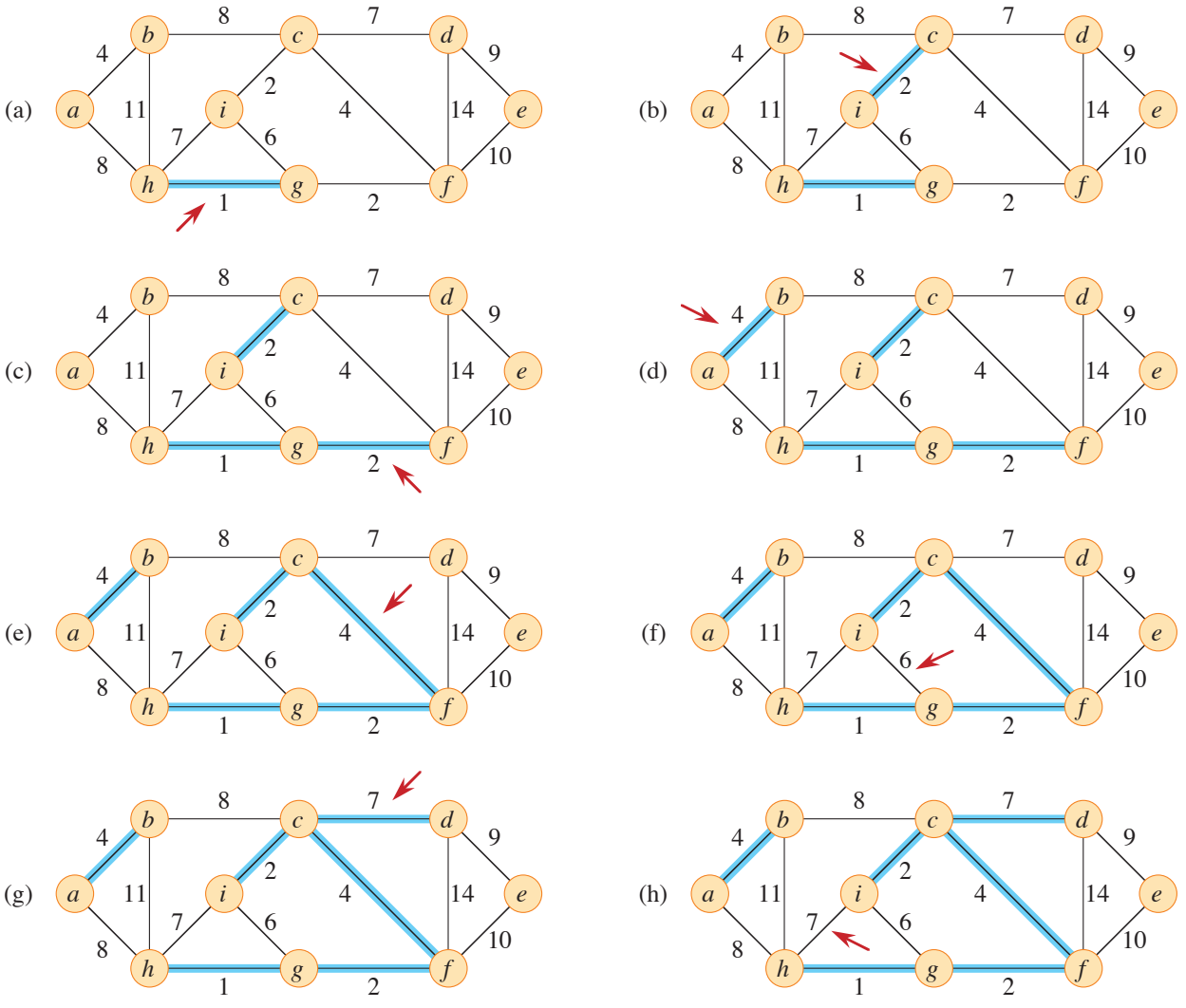

Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's algorithm

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) with the lowest weight. Let C_1 and C_2 denote the two trees that are connected by (u, v). Since (u, v) must be a light edge connecting C_1 to some other tree, Corollary 21.2 implies

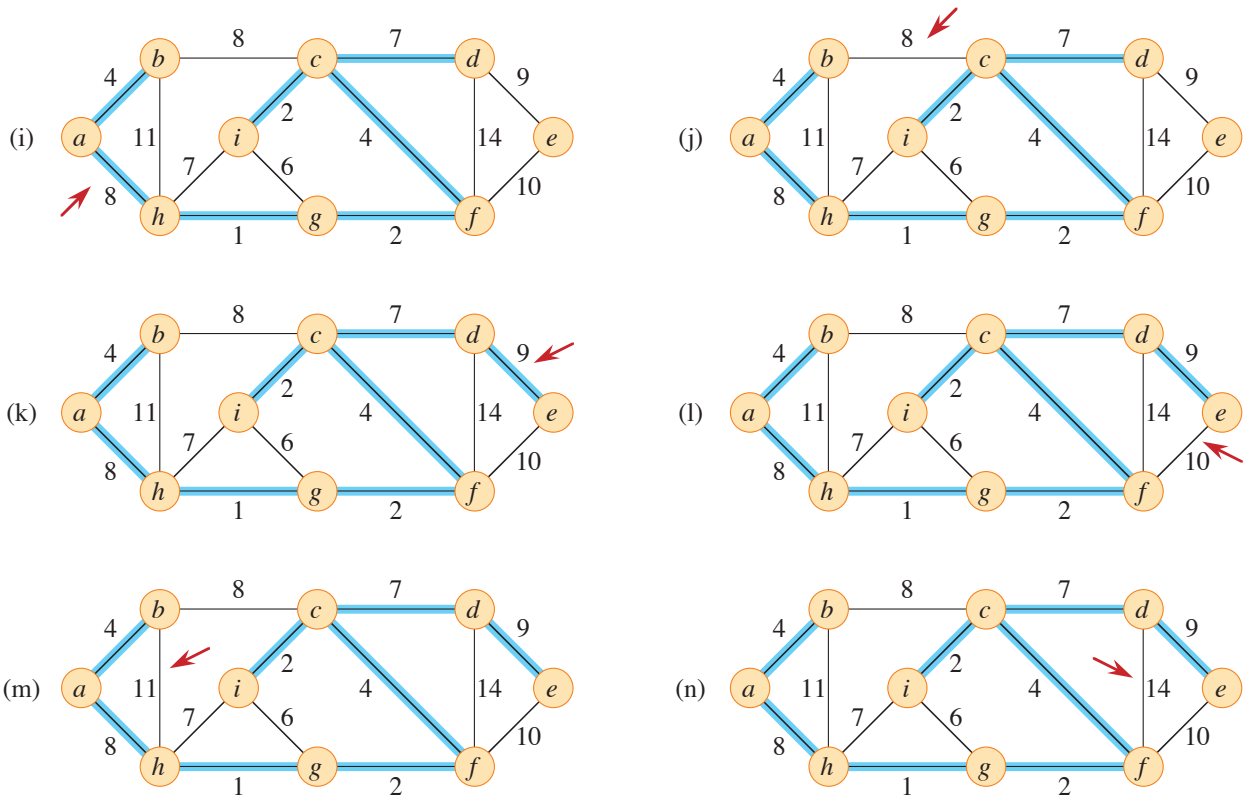

Figure 21.4, continued Further steps in the execution of Kruskal's algorithm.

that (u, v) is a safe edge for C_1. Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge with the lowest possible weight.

Like the algorithm to compute connected components from Section 19.1, the procedure MST-KRUSKAL on the following page uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, to determine whether two vertices u and v belong to the same tree, just test whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Figure 21.4 shows how Kruskal's algorithm works. Lines 1–3 initialize the set A to the empty set and create $|V|$ trees, one containing each vertex. The **for** loop in lines 6–9 examines edges in order of weight, from lowest to highest. The loop checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is ignored. Otherwise, the two vertices belong to different

```
MST-KRUSKAL(G, w)
 1  A = ∅
 2  for each vertex v ∈ G.V
 3       MAKE-SET(v)
 4  create a single list of the edges in G.E
 5  sort the list of edges into monotonically increasing order by weight w
 6  for each edge (u, v) taken from the sorted list in order
 7       if FIND-SET(u) ≠ FIND-SET(v)
 8            A = A ∪ {(u, v)}
 9            UNION(u, v)
10  return A
```

trees. In this case, line 8 adds the edge (u, v) to A, and line 9 merges the vertices in the two trees.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the specific implementation of the disjoint-set data structure. Let's assume that it uses the disjoint-set-forest implementation of Section 19.3 with the union-by-rank and path-compression heuristics, since that is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, creating a single list of edges in line 4 takes $O(V + E)$ time (which is $O(E)$ because G is connected), and the time to sort the edges in line 5 is $O(E \lg E)$. (We'll account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 6–9 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these disjoint-set operations take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 19.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\,\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section 21.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we'll see in Section 22.3. Prim's algorithm has the property that the edges in the set A always form a single tree. As Figure 21.5 shows, the tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. Each step adds to the tree A

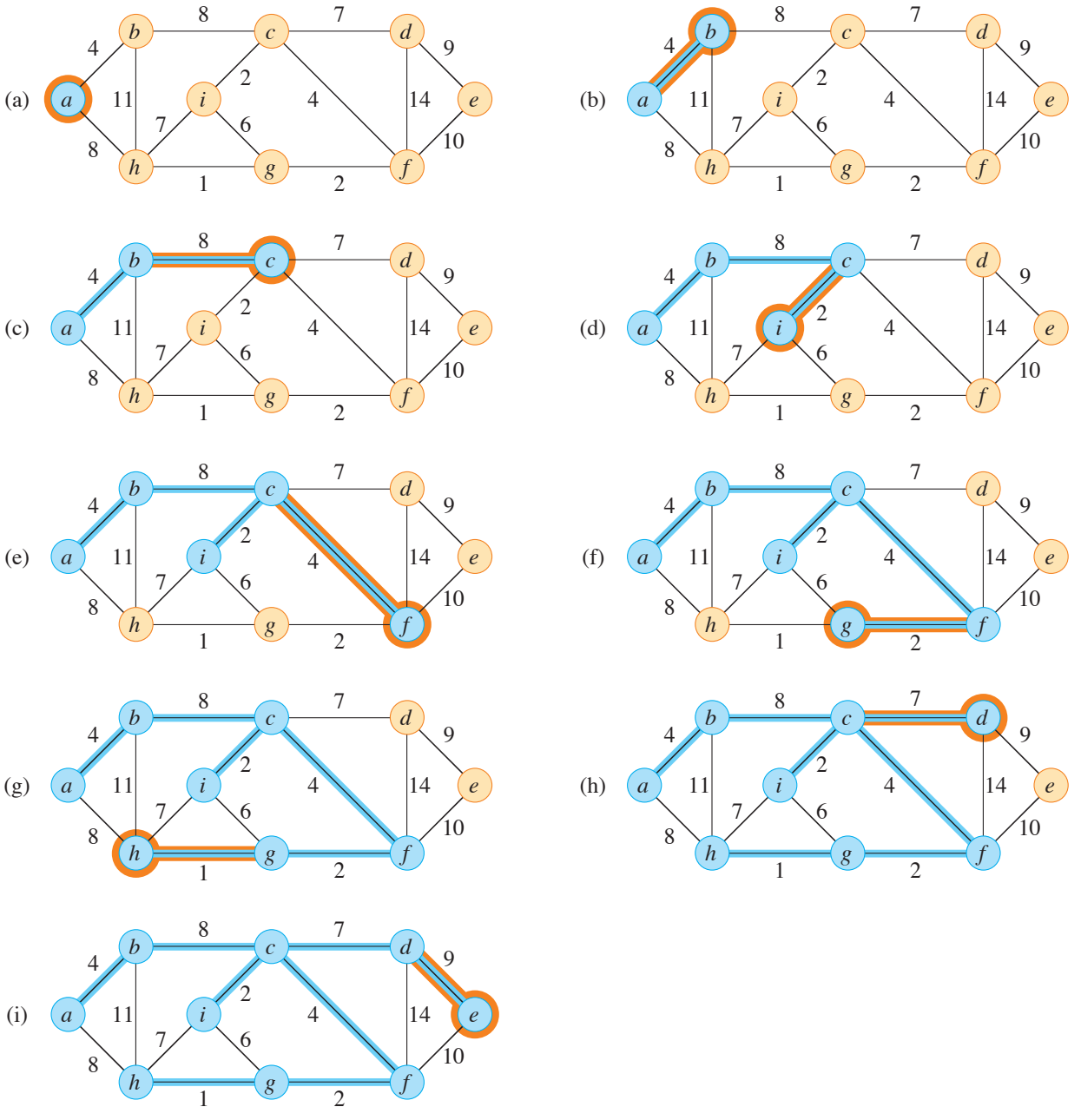

Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

a light edge that connects A to an isolated vertex—one on which no edge of A is incident. By Corollary 21.2, this rule adds only edges that are safe for A. Therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

In the procedure MST-PRIM below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. In order to efficiently select a new edge to add into tree A, the algorithm maintains a min-priority queue Q of all vertices that are *not* in the tree, based on a *key* attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree, where by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree. The algorithm implicitly maintains the set A from GENERIC-MST as

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\} \ ,$$

where we interpret the vertices in Q as forming a set. When the algorithm terminates, the min-priority queue Q is empty, and thus the minimum spanning tree A for G is

$$A = \{(v, v.\pi) : v \in V - \{r\}\} \ .$$

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

Figure 21.5 shows how Prim's algorithm works. Lines 1–7 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 to make it the first vertex processed), set the parent of each vertex to NIL, and insert each vertex into the min-priority queue Q. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 8–14,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq \text{NIL}$, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Line 9 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to lines 4–7). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding the edge $(u, u.\pi)$ to A. The **for** loop of lines 10–14 updates the key and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant. Whenever line 13 updates $v.key$, line 14 calls DECREASE-KEY to inform the min-priority queue that v's key has changed.

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap (see Chapter 6), including a way to map between vertices and their corresponding heap elements. The BUILD-MIN-HEAP procedure can perform lines 5–7 in $O(V)$ time. In fact, there is no need to call BUILD-MIN-HEAP. You can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 10–14 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, the test for membership in Q in line 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE-KEY in line 14 takes $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

You can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap (see page 478). If a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and each INSERT and DECREASE-KEY operation takes only $O(1)$ amortized time. Therefore, by using a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Exercises

21.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

21.2-2

Give a simple implementation of Prim's algorithm that runs in $O(V^2)$ time when the graph $G = (V, E)$ is represented as an adjacency matrix.

21.2-3

For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

21.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

21.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

21.2-6

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

★ ***21.2-7***
Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

★ ***21.2-8***
Suppose that a graph G has a minimum spanning tree already computed. How quickly can you update the minimum spanning tree upon adding a new vertex and incident edges to G?

Problems

21-1 Second-best minimum spanning tree
Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T' such that $w(T') = \min\{w(T'') : T'' \in \mathcal{T} - \{T\}\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $(T - \{(u, v)\}) \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

21-2 Minimum spanning tree in sparse graphs
For a very sparse connected graph $G = (V, E)$, it is possible to further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with a Fibonacci heap by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, for each vertex u, choose the minimum-weight edge (u, v)

incident on u, and put (u, v) into the minimum spanning tree under construction. Then, contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, first identify sets of vertices that are united into the same new vertex. Then create the graph that would have resulted from contracting these edges one at a time, but do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph might be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, initialize the two attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. Use the *orig* attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The c attribute holds the weight of an edge, and as edges are contracted, it is updated according to the above scheme for choosing edge weights. The procedure MST-REDUCE on the facing page takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree T.

a. Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call MST-PRIM(G', c', r), where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.

b. Argue that $|G'.V| \le |V|/2$.

c. Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (*Hint:* Use simple data structures.)

d. Suppose that you run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T. Argue that the overall running time of the k phases is $O(kE)$.

e. Suppose that after running k phases of MST-REDUCE, as in part (d), you run Prim's algorithm by calling MST-PRIM(G', c', r), where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg\lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.

f. For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

```
MST-REDUCE(G, T)
 1  for each vertex v ∈ G.V
 2      v.mark = FALSE
 3      MAKE-SET(v)
 4  for each vertex u ∈ G.V
 5      if u.mark == FALSE
 6          choose v ∈ G.Adj[u] such that (u, v).c is minimized
 7          UNION(u, v)
 8          T = T ∪ {(u, v).orig}
 9          u.mark = TRUE
10          v.mark = TRUE
11  G'.V = {FIND-SET(v) : v ∈ G.V}
12  G'.E = ∅
13  for each edge (x, y) ∈ G.E
14      u = FIND-SET(x)
15      v = FIND-SET(y)
16      if u ≠ v
17          if (u, v) ∉ G'.E
18              G'.E = G'.E ∪ {(u, v)}
19              (u, v).orig' = (x, y).orig
20              (u, v).c' = (x, y).c
21          elseif (x, y).c < (u, v).c'
22              (u, v).orig' = (x, y).orig
23              (u, v).c' = (x, y).c
24  construct adjacency lists G'.Adj for G'
25  return G' and T
```

21-3 Alternative minimum-spanning-tree algorithms

Consider the three algorithms MAYBE-MST-A, MAYBE-MST-B, and MAYBE-MST-C on the next page. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not necessarily a minimum spanning tree. Also describe the most efficient implementation of each algorithm, regardless of whether it computes a minimum spanning tree.

21-4 Bottleneck spanning tree

A ***bottleneck spanning tree*** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. The value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.

```
MAYBE-MST-A(G, w)
1  sort the edges into monotonically decreasing order of edge weights w
2  T = E
3  for each edge e, taken in monotonically decreasing order by weight
4      if T − {e} is a connected graph
5          T = T − {e}
6  return T

MAYBE-MST-B(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      if T ∪ {e} has no cycles
4          T = T ∪ {e}
5  return T

MAYBE-MST-C(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      T = T ∪ {e}
4      if T has a cycle c
5          let e′ be a maximum-weight edge on c
6          T = T − {e′}
7  return T
```

a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, you will show how to find a bottleneck spanning tree in linear time.

b. Give a linear-time algorithm that, given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.

c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (*Hint:* You might want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 21-2.)

Chapter notes

Tarjan [429] surveys the minimum-spanning-tree problem and provides excellent advanced material. Graham and Hell [198] compiled a history of the minimum-spanning-tree problem.

Tarjan attributes the first minimum-spanning-tree algorithm to a 1926 paper by O. Borůvka. Borůvka's algorithm consists of running $O(\lg V)$ iterations of the procedure MST-REDUCE described in Problem 21-2. Kruskal's algorithm was reported by Kruskal [272] in 1956. The algorithm commonly known as Prim's algorithm was indeed invented by Prim [367], but it was also invented earlier by V. Jarník in 1930.

When $|E| = \Omega(V \lg V)$, Prim's algorithm, implemented with a Fibonacci heap, runs in $O(E)$ time. For sparser graphs, using a combination of the ideas from Prim's algorithm, Kruskal's algorithm, and Borůvka's algorithm, together with advanced data structures, Fredman and Tarjan [156] give an algorithm that runs in $O(E \lg^* V)$ time. Gabow, Galil, Spencer, and Tarjan [165] improved this algorithm to run in $O(E \lg\lg^* V)$ time. Chazelle [83] gives an algorithm that runs in $O(E\, \hat{\alpha}(E, V))$ time, where $\hat{\alpha}(E, V)$ is the functional inverse of Ackermann's function. (See the chapter notes for Chapter 19 for a brief discussion of Ackermann's function and its inverse.) Unlike previous minimum-spanning-tree algorithms, Chazelle's algorithm does not follow the greedy method. Pettie and Ramachandran [356] give an algorithm based on precomputed "MST decision trees" that also runs in $O(E\, \hat{\alpha}(E, V))$ time.

A related problem is ***spanning-tree verification***: given a graph $G = (V, E)$ and a tree $T \subseteq E$, determine whether T is a minimum spanning tree of G. King [254] gives a linear-time algorithm to verify a spanning tree, building on earlier work of Komlós [269] and Dixon, Rauch, and Tarjan [120].

The above algorithms are all deterministic and fall into the comparison-based model described in Chapter 8. Karger, Klein, and Tarjan [243] give a randomized minimum-spanning-tree algorithm that runs in $O(V + E)$ expected time. This algorithm uses recursion in a manner similar to the linear-time selection algorithm in Section 9.3: a recursive call on an auxiliary problem identifies a subset of the edges E' that cannot be in any minimum spanning tree. Another recursive call on $E - E'$ then finds the minimum spanning tree. The algorithm also uses ideas from Borůvka's algorithm and King's algorithm for spanning-tree verification.

Fredman and Willard [158] showed how to find a minimum spanning tree in $O(V + E)$ time using a deterministic algorithm that is not comparison based. Their algorithm assumes that the data are b-bit integers and that the computer memory consists of addressable b-bit words.