

### Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section 23.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we shall see in Section 24.3. Prim's algorithm has the property that the edges in the set  $A$  always form a single tree. As Figure 23.5 shows, the tree starts from an arbitrary root vertex  $r$  and grows until the tree spans all the vertices in  $V$ . Each step adds to the tree  $A$  a light edge that connects  $A$  to an isolated vertex—one on which no edge of  $A$  is incident. By Corollary 23.2, 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 order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in  $A$ . In the pseudocode below, the connected graph  $G$  and the root  $r$  of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are *not* in the tree reside in a min-priority queue  $Q$  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; 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

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

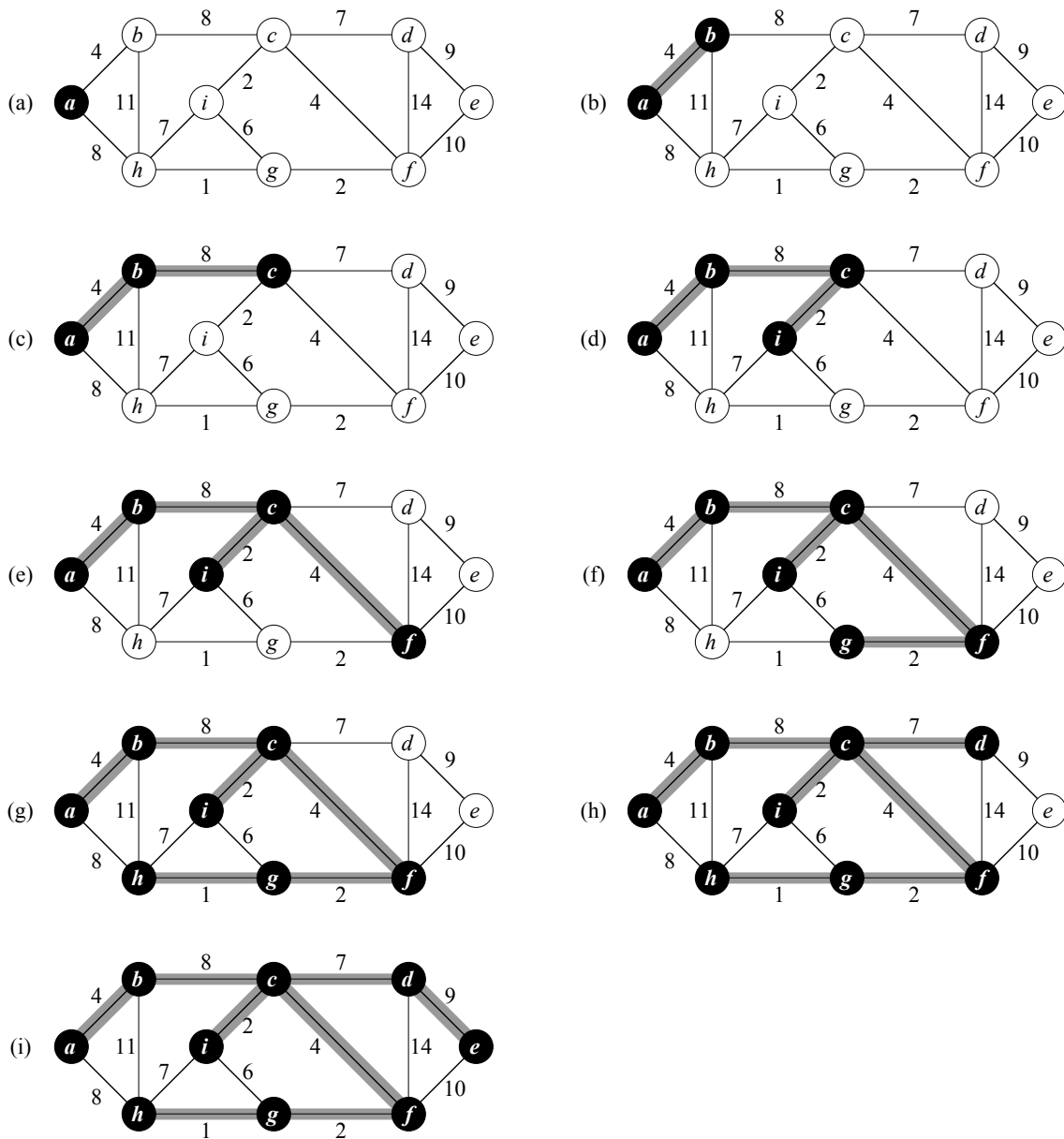
When the algorithm terminates, the min-priority queue  $Q$  is empty; the minimum spanning tree  $A$  for  $G$  is thus

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

MST-PRIM( $G, w, r$ )

```

1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```



**Figure 23.5** The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is *a*. Shaded edges are in the tree being grown, and black vertices are in 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. In the second step, 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.

Figure 23.5 shows how Prim's algorithm works. Lines 1–5 set the key of each vertex to  $\infty$  (except for the root  $r$ , whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the min-priority queue  $Q$  to contain all the vertices. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 6–11,

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  $v.\text{key} < \infty$  and  $v.\text{key}$  is the weight of a light edge  $(v, v.\pi)$  connecting  $v$  to some vertex already placed into the minimum spanning tree.

Line 7 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 line 4). Removing  $u$  from the set  $Q$  adds it to the set  $V - Q$  of vertices in the tree, thus adding  $(u, u.\pi)$  to  $A$ . The **for** loop of lines 8–11 updates the *key* and  $\pi$  attributes of every vertex  $v$  adjacent to  $u$  but not in the tree, thereby maintaining the third part of the loop invariant.

The running time of Prim's algorithm depends on how we implement the min-priority queue  $Q$ . If we implement  $Q$  as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in  $O(V)$  time. 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 8–11 executes  $O(E)$  times altogether, since the sum of the lengths of all adjacency lists is  $2|E|$ . Within the **for** loop, we can implement the test for membership in  $Q$  in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in  $Q$ , and updating the bit when the vertex is removed from  $Q$ . The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in  $O(\lg V)$  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.

We can improve the asymptotic running time of Prim's algorithm by using Fibonacci heaps. Chapter 19 shows that if a Fibonacci heap holds  $|V|$  elements, an EXTRACT-MIN operation takes  $O(\lg V)$  amortized time and a DECREASE-KEY operation (to implement line 11) takes  $O(1)$  amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue  $Q$ , the running time of Prim's algorithm improves to  $O(E + V \lg V)$ .