24.1-9

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'.

24.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section are elaborations of the generic algorithm. 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. The safe edge added to A is always a least-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 least-weight edge connecting the tree to a vertex not in the tree.

Kruskal's algorithm

Kruskal's algorithm is based directly on the generic minimum-spanning-tree algorithm given in Section 24.1. It 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) of least 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 24.2 implies that (u, v) is a safe edge for C_1. Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Our implementation of Kruskal's algorithm is like the algorithm to compute connected components from Section 22.1. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). The combining of trees is accomplished by the UNION procedure.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u, v) ∈ E, in order by nondecreasing weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

Kruskal's algorithm works as shown in Figure 24.4. Lines 1–3 initialize the set A to the empty set and create $|V|$ trees, one containing each vertex. The edges in E are sorted into order by nondecreasing weight in line 4. The **for** loop in lines 5–8 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 discarded. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure. We shall assume the disjoint-set-forest implementation of Section 22.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initialization takes time $O(V)$, and the time to sort the edges in line 4 is $O(E \lg E)$. There are $O(E)$ operations on the disjoint-set forest, which in total take $O(E\ \alpha(E, V))$ time, where α is the functional inverse of Ackermann's function defined in Section 22.4. Since $\alpha(E, V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$.

Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree algorithm from Section 24.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph. (See Section 25.2.) Prim's algorithm has the property that the edges in the set A always form a single tree. As is illustrated in Figure 24.5, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. At each step, a light edge connecting a vertex in A to a vertex in $V - A$ is added to the tree. By Corollary 24.2, 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 is "greedy" since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

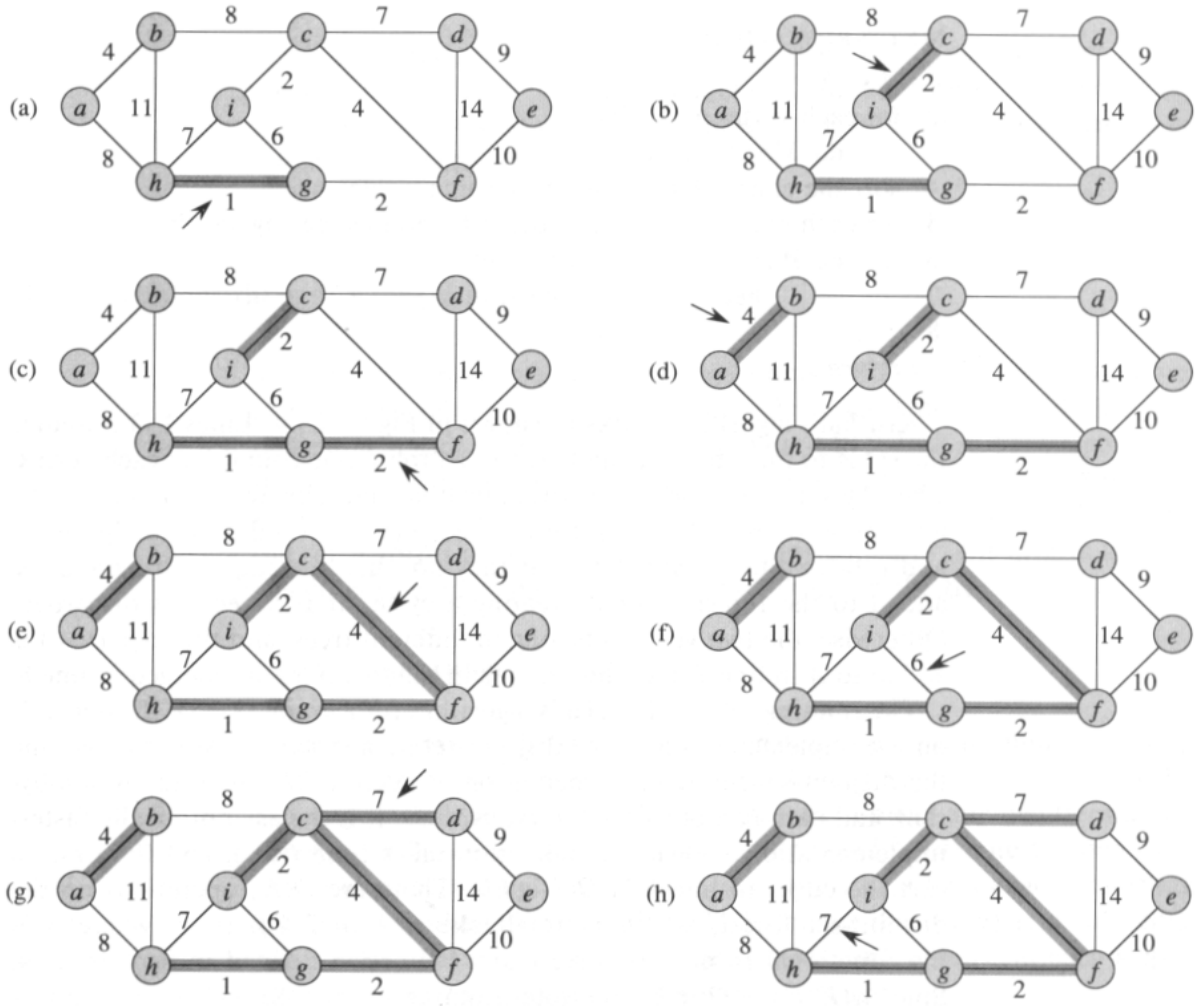
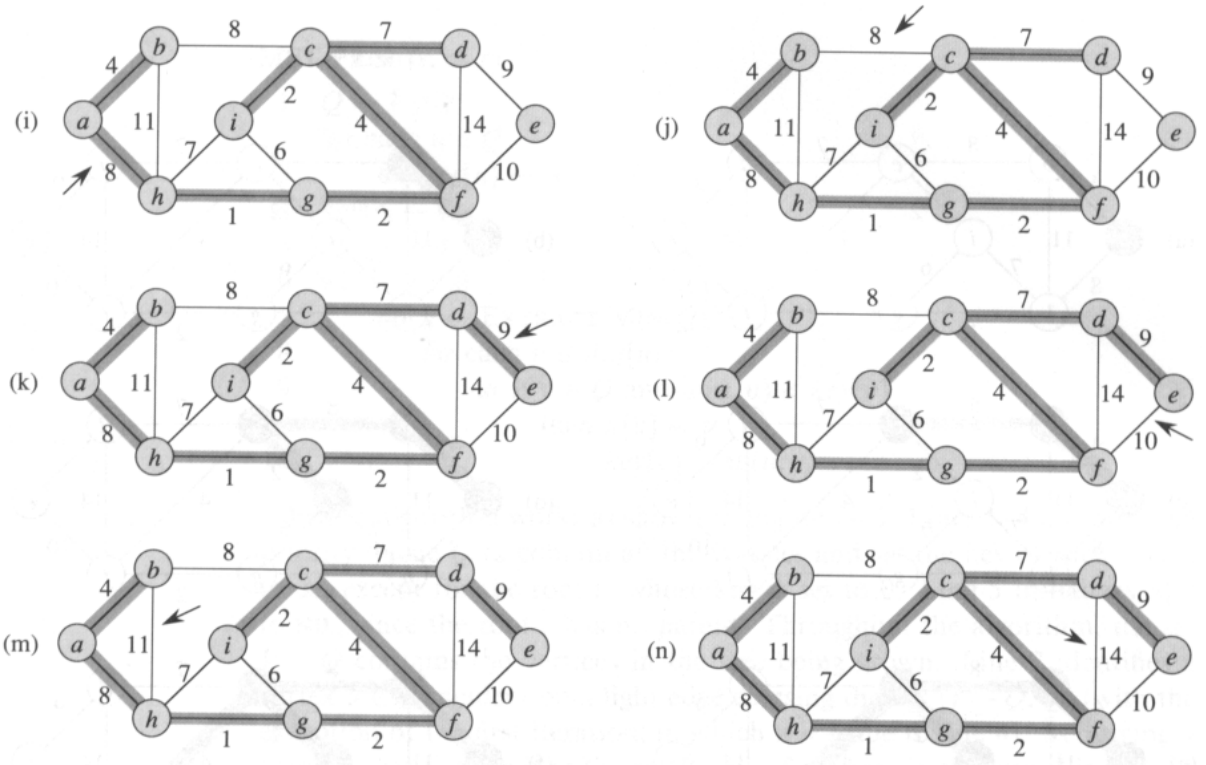


Figure 24.4 The execution of Kruskal's algorithm on the graph from Figure 24.1. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.



The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added 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 priority queue Q based on a *key* field. For each vertex v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $key[v] = \infty$ if there is no such edge. The field $\pi[v]$ names the "parent" of v in the tree. During the algorithm, the set A from GENERIC-MST is kept implicitly as

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

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

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

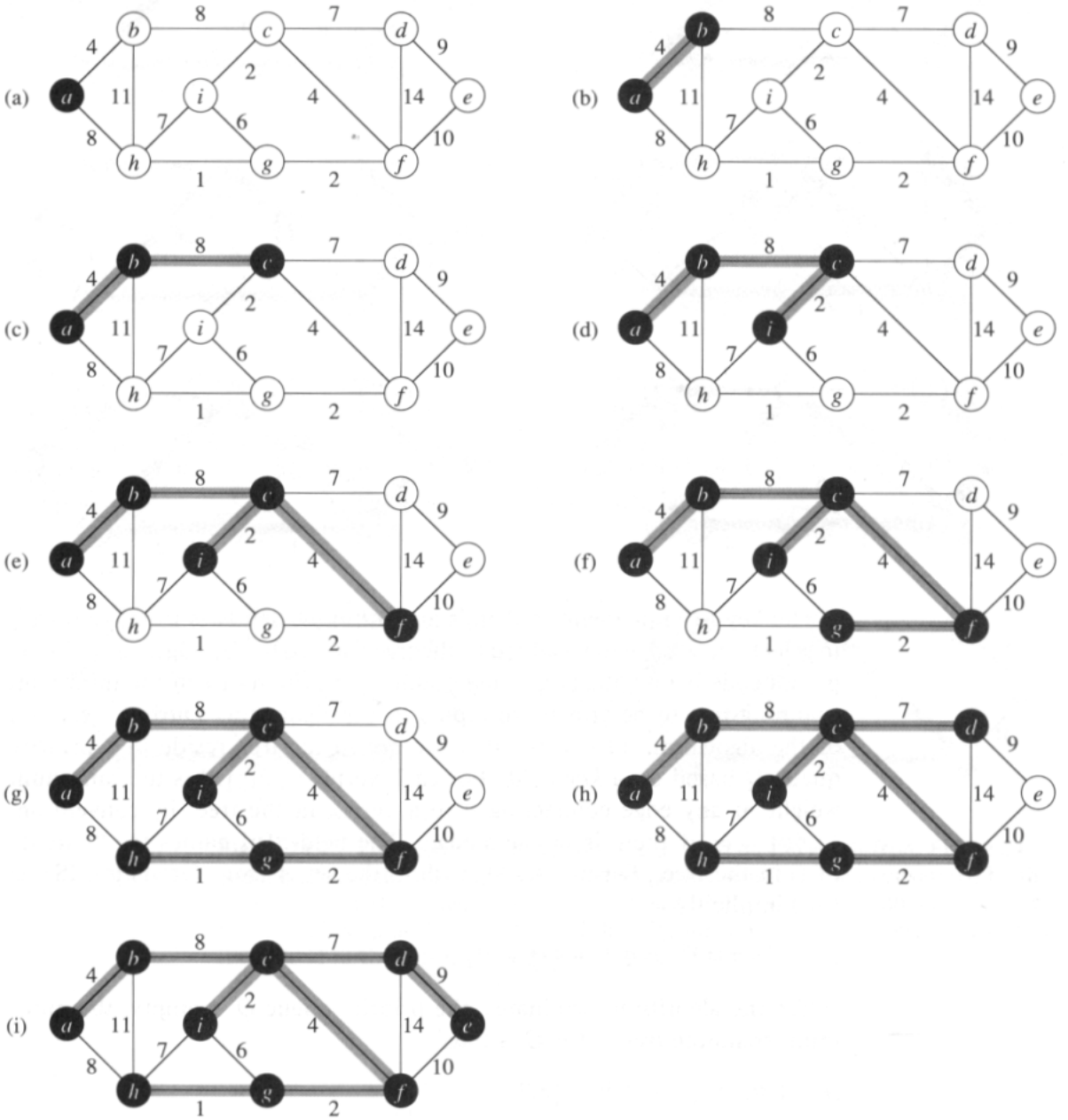


Figure 24.5 The execution of Prim's algorithm on the graph from Figure 24.1. The root vertex is a. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing 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.

```
MST-PRIM(G, w, r)
 1  Q ← V[G]
 2  for each u ∈ Q
 3       do key[u] ← ∞
 4  key[r] ← 0
 5  π[r] ← NIL
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Prim's algorithm works as shown in Figure 24.5. Lines 1–4 initialize the priority queue Q to contain all the vertices and set the key of each vertex to ∞, except for the root r, whose key is set to 0. Line 5 initializes $\pi[r]$ to NIL, since the root r has no parent. Throughout the algorithm, the set $V - Q$ contains the vertices in the tree being grown. Line 7 identifies a vertex $u \in Q$ incident on a light edge crossing 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. Lines 8–11 update the *key* and π fields of every vertex v adjacent to u but not in the tree. The updating maintains the invariants that $key[v] = w(v, \pi[v])$ and that $(v, \pi[v])$ is a light edge connecting v to some vertex in the tree.

The performance of Prim's algorithm depends on how we implement the priority queue Q. If Q is implemented as a binary heap (see Chapter 7), we can use the BUILD-HEAP procedure to perform the initialization in lines 1–4 in $O(V)$ time. The loop is executed $|V|$ 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 is executed $O(E)$ 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 9 can be implemented 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 heap, which can be implemented in a binary heap 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.

The asymptotic running time of Prim's algorithm can be improved, however, by using Fibonacci heaps. Chapter 21 shows that if $|V|$ elements are organized into a Fibonacci heap, we can perform an EXTRACT-MIN operation in $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time. Therefore, if we use a Fibonacci

heap to implement the priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Exercises

24.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how ties are broken when the edges are sorted into order. 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.

24.2-2

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

24.2-3

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

24.2-4

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?

24.2-5

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?

24.2-6

Describe an efficient algorithm that, given an undirected graph G, determines a spanning tree of G whose largest edge weight is minimum over all spanning trees of G.

24.2-7 ⋆

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?

24.2-8 ⋆

Suppose that a graph G has a minimum spanning tree already computed. How quickly can the minimum spanning tree be updated if a new vertex and incident edges are added to G?