Chapter 15 Minimum Spanning Trees

Saguaros, Saguaro National Monument, 1941. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Contents

Suppose the remote mountain country of Vectoria has been given a major grant to install a large Wi-Fi the center of each of its mountain villages. The grant pays for the installation and maintenance of the towers, but it doesn't pay the cost for running the cables that would be connecting all the towers to the Internet. Communication cables can run from the main Internet access point to a village tower and cables can also run between pairs of towers. The challenge is to interconnect all the towers and the Internet access point as cheaply as possible.

We can model this problem using a graph, G, where each vertex in G is the location of a Wi-Fi the Internet access point, and an edge in G is a possible cable we could run between two such vertices. Each edge in G could then be given a weight that is equal to the cost of running the cable that that edge represents. Thus, we are interested in finding a connected acyclic subgraph of G that includes all the vertices of G and has minimum total cost. That is, using the language of graph theory, we are interested in finding a minimum spanning tree of G, which is the topic we explore in this chapter. (See Figure 15.1.)

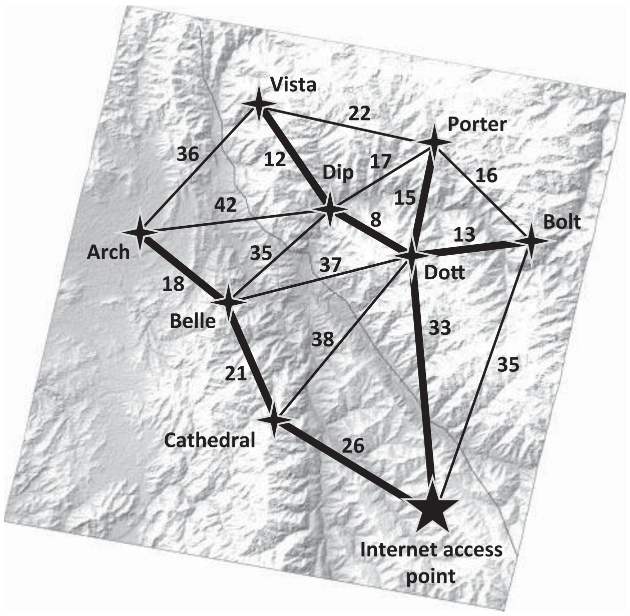


Figure 15.1: Connecting Wi-Fi Vectoria as cheaply as possible. Each edge is labeled with the cost, in millions of Vectoria dollars, of running a cable between its two endpoints. The edges that belong to the cheapest way to connect all the villages and the Internet access point are shown in bold. Note that this spanning tree is not the same as a shortest-path tree rooted at the Internet access point. For example, the shortest-path distance to Bolt is 35, but this edge is not included, since we can connect Bolt to Dott with an edge of cost 13. Background image is a Kashmir elevation map, 2005. U.S. government image. Credit: NASA.

15.1 Properties of Minimum Spanning Trees

As in the tower-connecting problem in Vectoria mentioned above, suppose we wish to connect all the computers in a new office building using the least amount of cable or suppose we have an undirected computer network in which each connection between two routers has a cost for usage and we want to connect all our routers at the minimum cost possible. As in the example of connecting villages in Vectoria, we can model these problems using a weighted graph, G, whose vertices represent the computers or routers, and whose edges represent all the possible pairs (u, v) of computers, where the weight $w((v, u))$ of edge (v, u) is equal to the amount of cable or network cost needed to connect computer v to computer u. Rather than computing a shortest-path tree from some particular vertex v, we are interested instead in finding a (free) tree T that contains all the vertices of G and has the minimum total weight over all such trees.

Given a weighted undirected graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum of the weights of the edges of T, that is,

$$w(T) = \sum_{e \in T} w(e).$$

Recall that a tree such as this, which contains every vertex of a connected graph G is a ***spanning tree***. Computing a spanning tree T with smallest total weight is the problem of constructing a ***minimum spanning tree*** (or ***MST***).

All the well-known efficient algorithms for finding minimum spanning trees are applications of the ***greedy method***. As was discussed in Section 10, we apply the greedy method by iteratively choosing objects to join a growing collection, by incrementally picking an object that minimizes or maximizes the change in some objective function. Thus, algorithms for constructing minimum spanning trees based on this approach must define some kind of greedy ordering and some kind of objective function to optimize with each greedy step. The details for these definitions differ among the various well-known MST algorithms, however.

The first MST algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in order of their weights. The second algorithm we discuss is the Prim-Jarník algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm. We conclude this chapter by discussing a third algorithm, due to Barůvka, which applies the greedy approach in a parallel way.

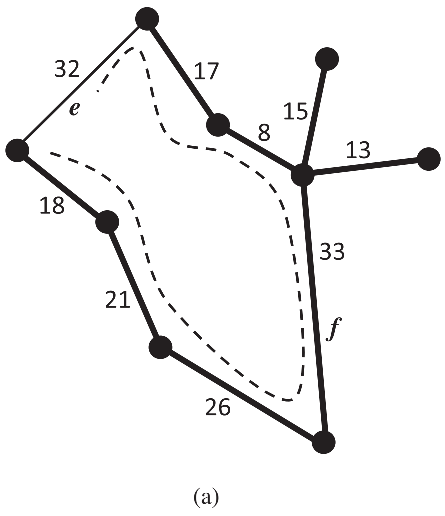
In order to simplify the description of the algorithms, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u, z).

Crucial Facts about Minimum Spanning Trees

Before we discuss the details of efficient MST algorithms, however, let us give some crucial facts about minimum spanning trees that form the basis of these algorithms.

Lemma 15.1: *Let G be a weighted connected graph, and let T be a minimum spanning tree for T. If e is an edge of G that is not in T, then the weight of e is at least as great as any edge in the cycle created by adding e to T.*

Proof: Since T is a spanning tree (that is, a connected acyclic subgraph of G that contains all the vertices of G), adding e to T creates a cycle, C. Suppose, for the sake of contradiction, that there is an edge, f, whose weight is more than e's weight, that is, $w(e) < w(f)$. Then we can remove f from T and replace it with e, and this will result in a spanning tree, T', whose total weight is less than the total weight of T. But the existence of such a tree, T', would contradict the fact that T is a ***minimum*** spanning tree. So no such edge, f, can exist. (See Figure 15.2.) ■



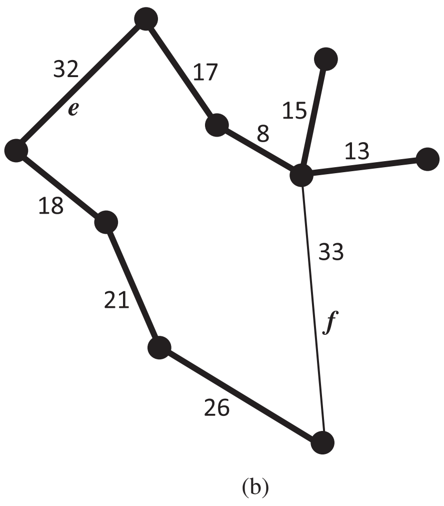


Figure 15.2: Any nontree edge must have weight that is at least as great as every edge in the cycle created by that edge and a minimum spanning tree. (a) Suppose a nontree edge, e, has lower weight than an edge, f, in the cycle created by e and a minimum spanning tree (shown with bold edges). (b) Then we could replace f by e and get a spanning tree with lower total weight, which would contradict the fact that we started with a ***minimum*** spanning tree.

In addition, all the MST algorithms we discuss in this chapter are based crucially on the following fact. (See Figure 15.3.)

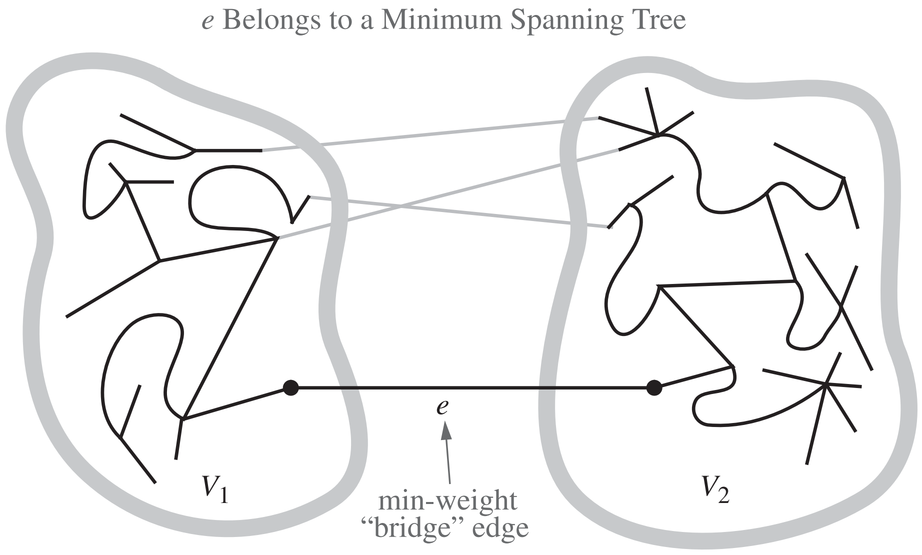


Figure 15.3: An illustration of a crucial fact about minimum spanning trees.

Theorem 15.2: *Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. There is a minimum spanning tree T that has e as one of its edges.*

Proof: The proof is similar to that of Lemma 15.1. Let T be a minimum spanning tree of G. If T does not contain edge e, the addition of e to T must create a cycle. Therefore, there is some edge, f, of this cycle that has one endpoint in V_1 and the other in V_2. Moreover, by the choice of e, $w(e) \leq w(f)$. If we remove f from $T \cup \{e\}$, we obtain a spanning tree whose total weight is no more than before. Since T was a minimum spanning tree, this new tree must also be a minimum spanning tree. ■

In fact, if the weights in G are distinct, then the minimum spanning tree is unique; we leave the justification of this less crucial fact as an exercise (C-15.3).

Also, note that Theorem 15.2 remains valid even if the graph G contains negative-weight edges or negative-weight cycles, unlike the algorithms we presented for shortest paths.

15.2 Kruskal's Algorithm

The reason Theorem 15.2 is so important is that it can be used as the basis for building a minimum spanning tree. In Kruskal's algorithm, it is used to build the minimum spanning tree in clusters. Initially, each vertex is in its own cluster all by itself. The algorithm then considers each edge in turn, ordered by increasing weight. If an edge e connects two different clusters, then e is added to the set of edges of the minimum spanning tree, and the two clusters connected by e are merged into a single cluster. If, on the other hand, e connects two vertices that are already in the same cluster, then e is discarded. Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the minimum spanning tree.

We give pseudocode for Kruskal's method for solving the MST problem in Algorithm 15.4, and we illustrate this algorithm in Figures 15.5, 15.6, and 15.7.

```
Algorithm KruskalMST($G$):
  Input: A simple connected weighted graph $G$ with $n$ vertices and $m$ edges
  Output: A minimum spanning tree $T$ for $G$
  for each vertex $v$ in $G$ do
      Define an elementary cluster $C(v) \leftarrow \{v\}$.
  Let $Q$ be a priority queue storing the edges in $G$, using edge weights as keys
  $T \leftarrow \emptyset$      // $T$ will ultimately contain the edges of the MST
  while $T$ has fewer than $n - 1$ edges do
      $(u, v) \leftarrow Q$.removeMin()
      Let $C(v)$ be the cluster containing $v$
      Let $C(u)$ be the cluster containing $u$
      if $C(v) \neq C(u)$ then
          Add edge $(v, u)$ to $T$
          Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$
  return tree $T$
```

Algorithm 15.4: Kruskal's algorithm for the MST problem.

As mentioned before, the correctness of Kruskal's algorithm follows from the crucial fact about minimum spanning trees from Theorem 15.2. Each time Kruskal's algorithm adds an edge (v, u) to the minimum spanning tree T, we can define a partitioning of the set of vertices V (as in the theorem) by letting V_1 be the cluster containing v and letting V_2 contain the rest of the vertices in V. This clearly defines a disjoint partitioning of the vertices of V and, more importantly, since we are extracting edges from Q in order by their weights, e must be a minimum-weight edge with one vertex in V_1 and the other in V_2. Thus, Kruskal's algorithm always adds a valid minimum spanning tree edge.

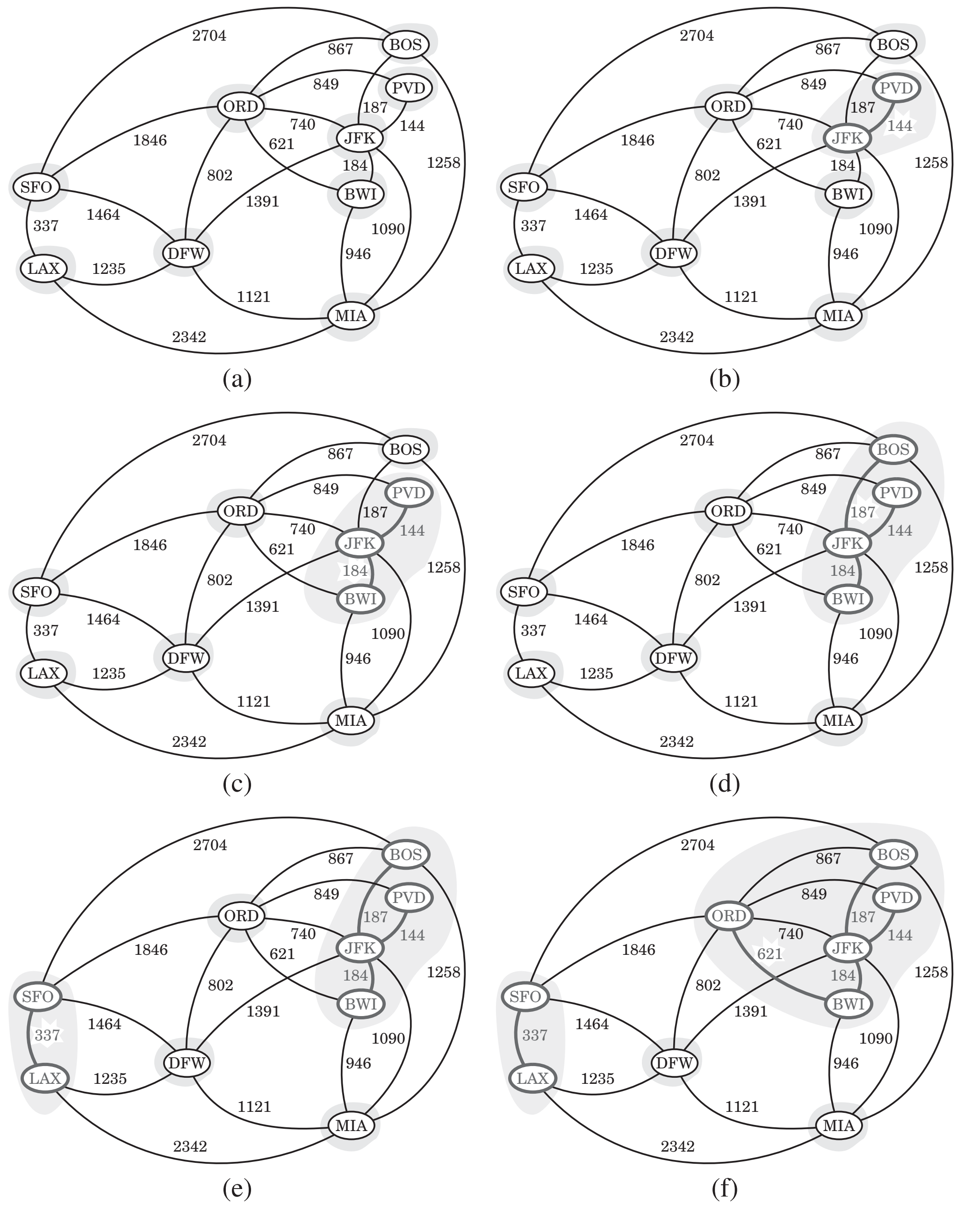


Figure 15.5: Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions, and we highlight the edge being considered in each iteration (continued in Figure 15.6).

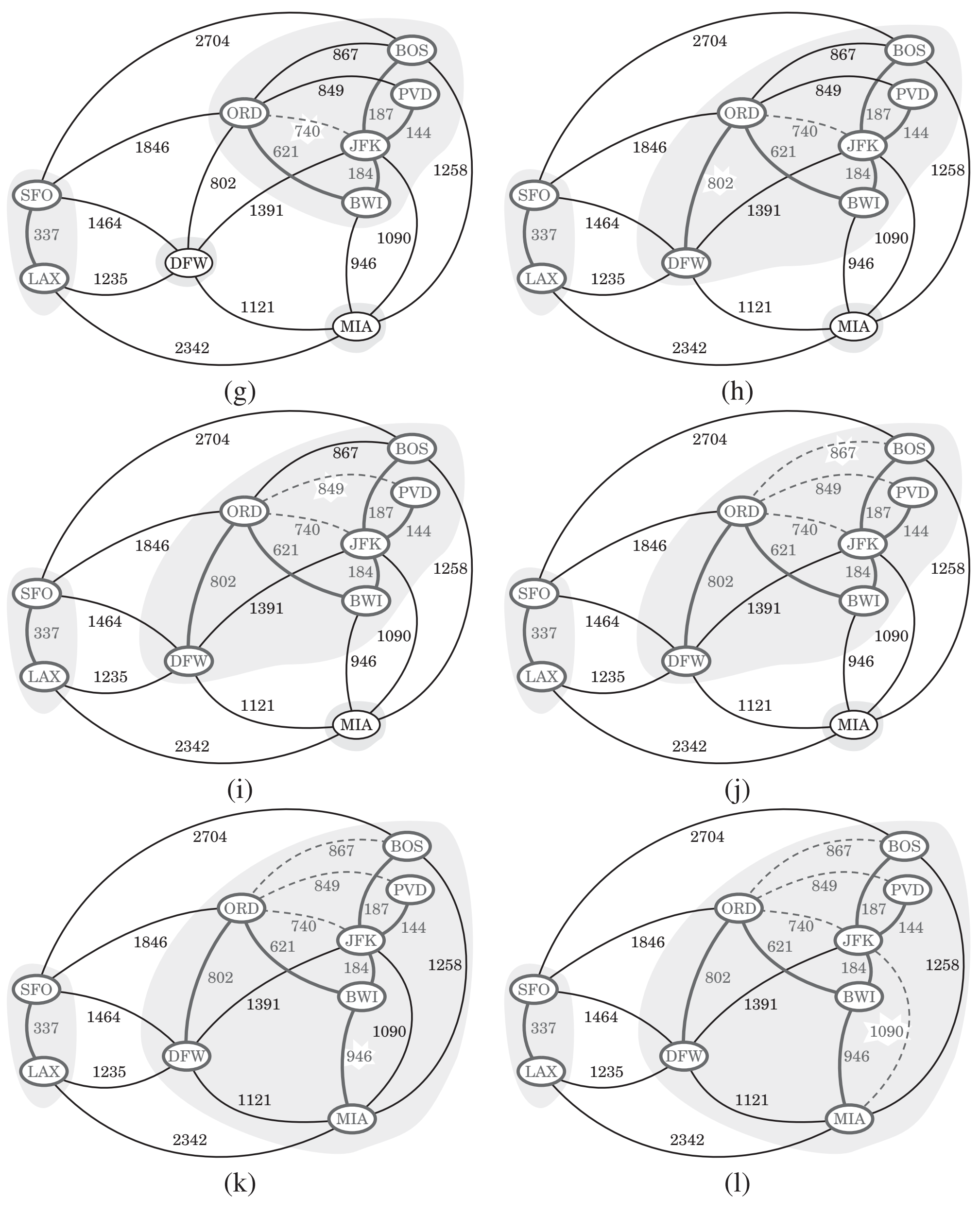


Figure 15.6: An example of an execution of Kruskal's MST algorithm (continued). Rejected edges are shown dashed (continued in Figure 15.7).

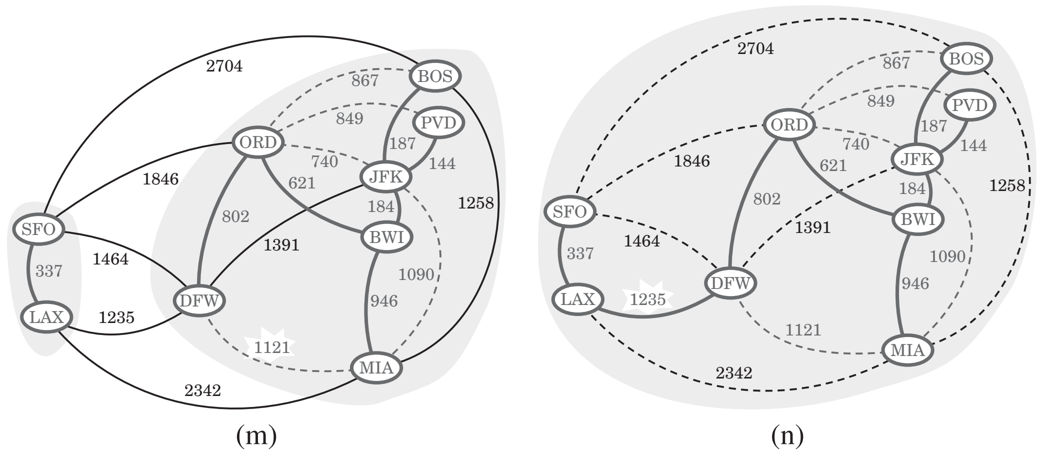


Figure 15.7: Example of an execution of Kruskal's MST algorithm (continued from Figures 15.5 and 15.6). The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm.

Analyzing Kruskal's Algorithm

Suppose G is a connected, weighted, undirected graph with n vertices and m edges. We assume that the edge weights can be compared in constant time. Because of the high level of the description we gave for Kruskal's algorithm in Algorithm 15.4, analyzing its running time requires that we give more details on its implementation. Specifically, we should indicate the data structures used and how they are implemented.

We implement the priority queue Q using a heap. Thus, we can initialize Q in $O(m \log m)$ time by repeated insertions, or in $O(m)$ time using bottom-up heap construction (see Section 5.4). In addition, at each iteration of the **while** loop, we can remove a minimum-weight edge in $O(\log m)$ time, which actually is $O(\log n)$, since G is simple.

We can use a list-based implementation of a partition (Section 7.1) for the clusters. Namely, let us represent each cluster C with an unordered linked list of vertices, storing, with each vertex v, a reference to its cluster $C(v)$. With this representation, testing whether $C(u) \neq C(v)$ takes $O(1)$ time. When we need to merge two clusters, $C(u)$ and $C(v)$, we move the elements of the ***smaller*** cluster into the larger one and update the cluster references of the vertices in the smaller cluster. Since we can simply add the elements of the smaller cluster at the end of the list for the larger cluster, merging two clusters takes time proportional to the size of the smaller cluster. That is, merging clusters $C(u)$ and $C(v)$ takes $O(\min\{|C(u)|, |C(v)|\})$ time. There are other, more efficient, methods for merging clusters (see Section 7.1), but this simple approach will be sufficient.

Lemma 15.3: *Consider an execution of Kruskal's algorithm on a graph with n vertices, where clusters are represented with sequences and with cluster references at each vertex. The total time spent merging clusters is $O(n \log n)$.*

Proof: We observe that each time a vertex is moved to a new cluster, the size of the cluster containing the vertex at least doubles. Let $t(v)$ be the number of times that vertex v is moved to a new cluster. Since the maximum cluster size is n,

$$t(v) \leq \log n.$$

The total time spent merging clusters in Kruskal's algorithm can be obtained by summing up the work done on each vertex, which is proportional to

$$\sum_{v \in G} t(v) \leq n \log n.$$

■

Using Lemma 15.3 and arguments similar to those used in the analysis of Dijkstra's algorithm, we conclude that the total running time of Kruskal's algorithm is $O((n+m)\log n)$, which can be simplified as $O(m \log n)$ since G is simple and connected.

Theorem 15.4: *Given a simple connected weighted graph G with n vertices and m edges, Kruskal's algorithm constructs a minimum spanning tree for G in $O(m \log n)$ time.*

An Alternative Implementation

In some applications, we may be given the edges in sorted order by their weights. In such cases, we can implement Kruskal's algorithm faster than the analysis given above. Specifically, we can implement the priority queue, Q, in this case, simply as an ordered list. This approach allows us to perform all the removeMin operations in constant time.

Then, instead of using a simple list-based partition data structure, we can use the tree-based union-find structure given in Chapter 7. This implies that the sequence of $O(m)$ union-find operations runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is the slow-growing inverse of the Ackermann function. Thus, we have the following.

Theorem 15.5: *Given a simple connected weighted graph G with n vertices and m edges, with the edges ordered by their weights, we can implement Kruskal's algorithm to construct a minimum spanning tree for G in $O(m\,\alpha(n))$ time.*

15.3 The Prim-Jarník Algorithm

In the Prim-Jarník algorithm, we grow a minimum spanning tree from a single cluster starting from some "root" vertex v. The main idea is similar to that of Dijkstra's algorithm. We begin with some vertex v, defining the initial "cloud" of vertices C. Then, in each iteration, we choose a minimum-weight edge $e = (v, u)$, connecting a vertex v in the cloud C to a vertex u outside of C. The vertex u is then brought into the cloud C and the process is repeated until a spanning tree is formed. Again, the crucial fact about minimum spanning trees from Theorem 15.2 is used here, for by always choosing the smallest-weight edge joining a vertex inside C to one outside C, we are assured of always adding a valid edge to the MST.

To efficiently implement this approach, we can take another cue from Dijkstra's algorithm. We maintain a label $D[u]$ for each vertex u outside the cloud C, so that $D[u]$ stores the weight of the best current edge for joining u to the cloud C. These labels allow us to reduce the number of edges that we must consider in deciding which vertex is next to join the cloud. We give the pseudocode in Algorithm 15.8.

```
Algorithm PrimJarníkMST($G$):
    Input: A weighted connected graph $G$ with $n$ vertices and $m$ edges
    Output: A minimum spanning tree $T$ for $G$
    Pick any vertex $v$ of $G$
    $D[v] \leftarrow 0$
    for each vertex $u \neq v$ do
        $D[u] \leftarrow +\infty$
    Initialize $T \leftarrow \emptyset$.
    Initialize a priority queue $Q$ with an item $((u, \text{null}), D[u])$ for each vertex $u$,
    where $(u, \text{null})$ is the element and $D[u]$ is the key.
    while $Q$ is not empty do
        $(u, e) \leftarrow Q$.removeMin()
        Add vertex $u$ and edge $e$ to $T$.
        for each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ do
            // perform the relaxation procedure on edge $(u, z)$
            if $w((u, z)) < D[z]$ then
                $D[z] \leftarrow w((u, z))$
                Change to $(z, (u, z))$ the element of vertex $z$ in $Q$.
                Change to $D[z]$ the key of vertex $z$ in $Q$.
    return the tree $T$
```

Algorithm 15.8: The Prim-Jarník algorithm for the MST problem.

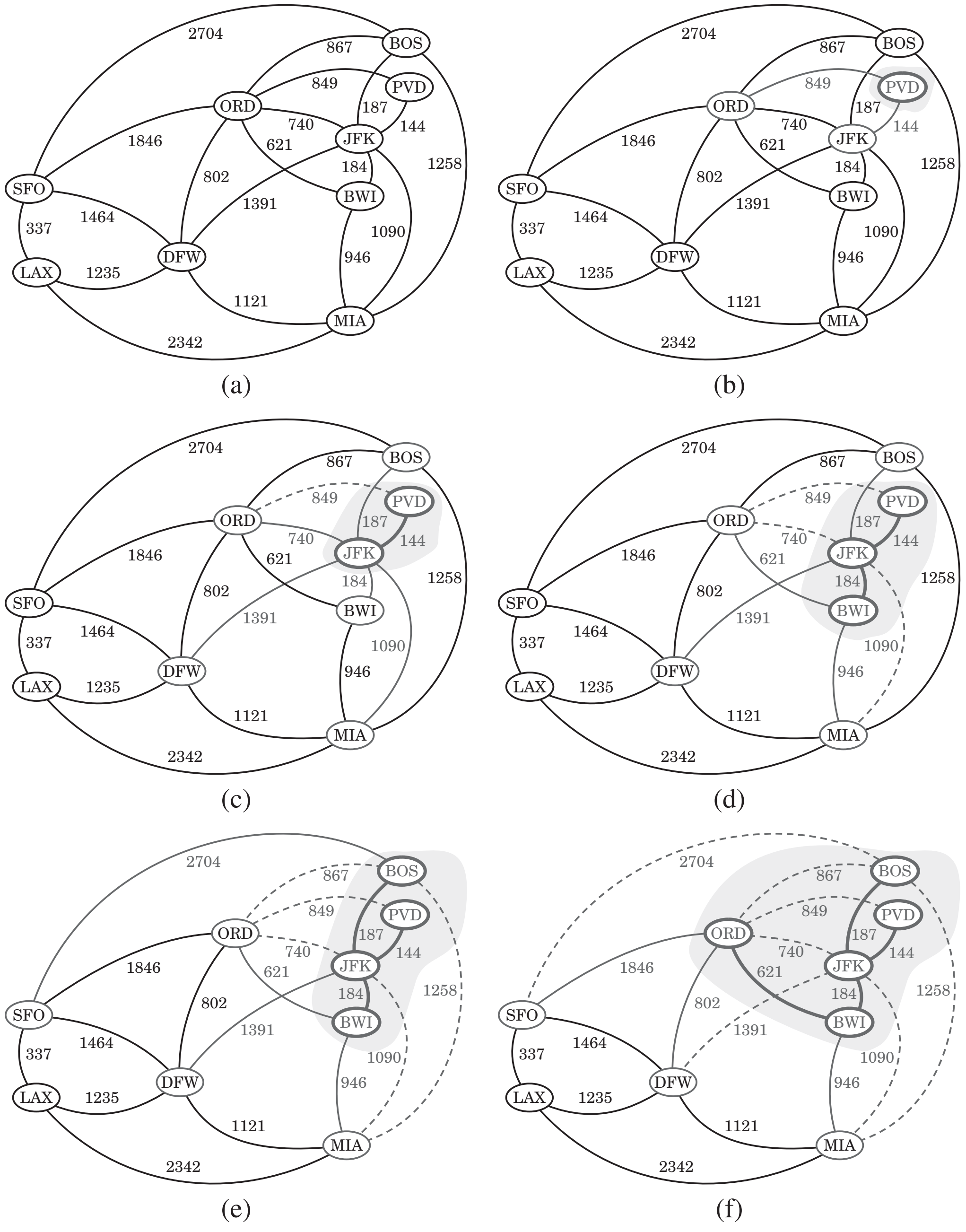


Figure 15.9: Visualizing the Prim-Jarník algorithm (continued in Figure 15.10).

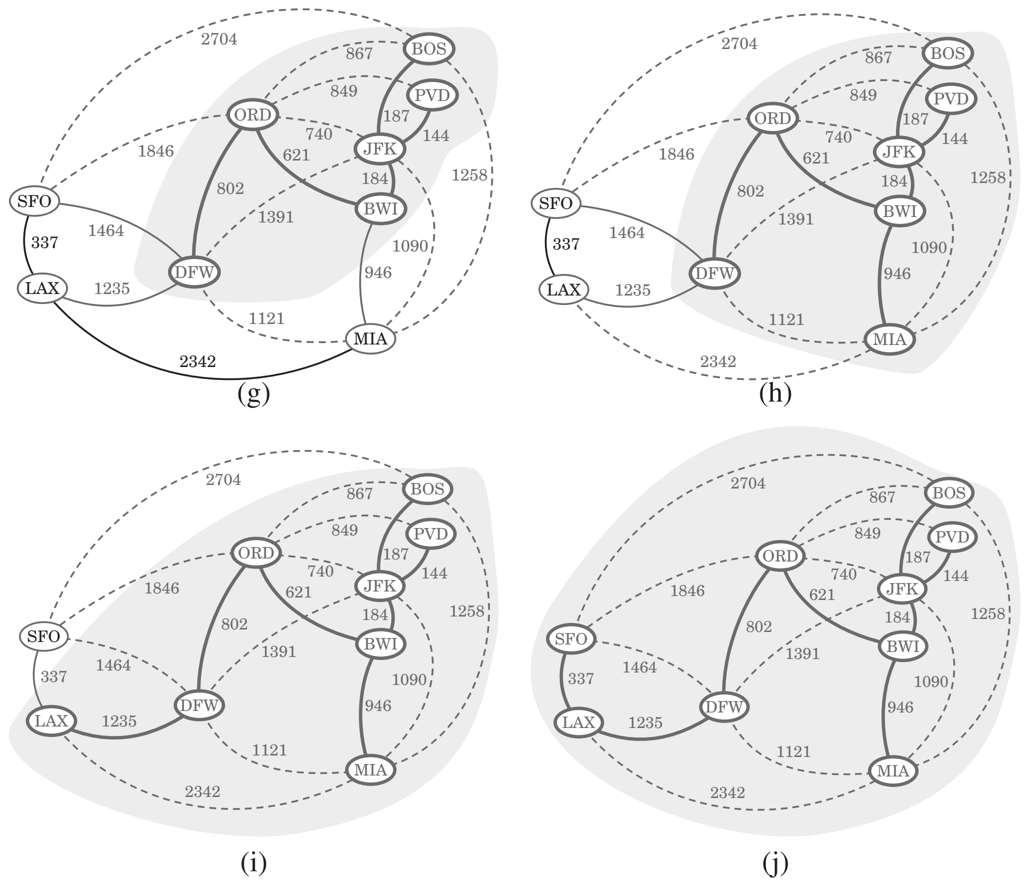


Figure 15.10: Visualizing the Prim-Jarník algorithm (continued from Figure 15.9).

Analyzing the Prim-Jarník Algorithm

Let n and m denote the number of vertices and edges of the input graph G, respectively. The implementation issues for the Prim-Jarník algorithm are similar to those for Dijkstra's algorithm. If we implement the priority queue Q as a heap that supports the locator-based priority queue methods (see Section 5.5), we can extract the vertex u in each iteration in $O(\log n)$ time.

In addition, we can update each $D[z]$ value in $O(\log n)$ time, as well, which is a computation considered at most once for each edge (u, z). The other steps in each iteration can be implemented in constant time. Thus, the total running time is $O((n + m) \log n)$, which is $O(m \log n)$. Hence, we can summarize as follows:

Theorem 15.6: *Given a simple connected weighted graph G with n vertices and m edges, the Prim-Jarník algorithm constructs a minimum spanning tree for G in $O(m \log n)$ time.*

We illustrate the Prim-Jarník algorithm in Figures 15.9 and 15.10.

15.4 Barůvka's Algorithm

Each of the two minimum spanning tree algorithms we have described previously has achieved its efficient running time by utilizing a priority queue Q, which could be implemented using a heap (or even a more sophisticated data structure). This usage should seem natural, for minimum spanning tree algorithms involve applications of the greedy method—and, in this case, the greedy method must explicitly be optimizing certain priorities among the vertices of the graph in question. It may be a bit surprising, but as we show in this section, we can actually design an efficient minimum spanning tree algorithm without using a priority queue. Moreover, what may be even more surprising is that the insight behind this simplification comes from the oldest known minimum spanning tree algorithm—the algorithm of Barůvka.

We present a pseudocode description of Barůvka's minimum spanning tree algorithm in Algorithm 15.11, and we illustrate this algorithm in Figure 15.12.

```
Algorithm BarůvkaMST($G$):
    Input: A weighted connected graph $G = (V, E)$ with unique edge weights
    Output: The minimum spanning tree $T$ for $G$.
    Let $T$ be a subgraph of $G$ initially containing just the vertices in $V$
    while $T$ has fewer than $|V| - 1$ edges do   // $T$ is not yet an MST
        for each connected component, $C_i$, of $T$ do
            // Perform the MST edge addition procedure for cluster $C_i$
            Let $e = (v, u)$ be a smallest-weight edge in $E$ with $v \in C_i$ and $u \notin C_i$
            Add $e$ to $T$ (unless $e$ is already in $T$)
    return $T$
```

Algorithm 15.11: Pseudo-code for Barůvka's algorithm.

Implementing Barůvka's algorithm to be efficient for a graph with n vertices and m edges is quite simple, requiring only that we be able to do the following:

- Maintain the forest T subject to edge insertion, which we can easily support in $O(1)$ time each using an adjacency list for T
- Traverse the forest T to identify connected components (clusters), which we can easily do in $O(n)$ time using a depth-first search of T
- Mark vertices with the name of the cluster they belong to, which we can do with an extra instance variable for each vertex
- Identify a smallest-weight edge in E incident upon a cluster C_i, which we can do by scanning the adjacency lists in G for the vertices in C_i.

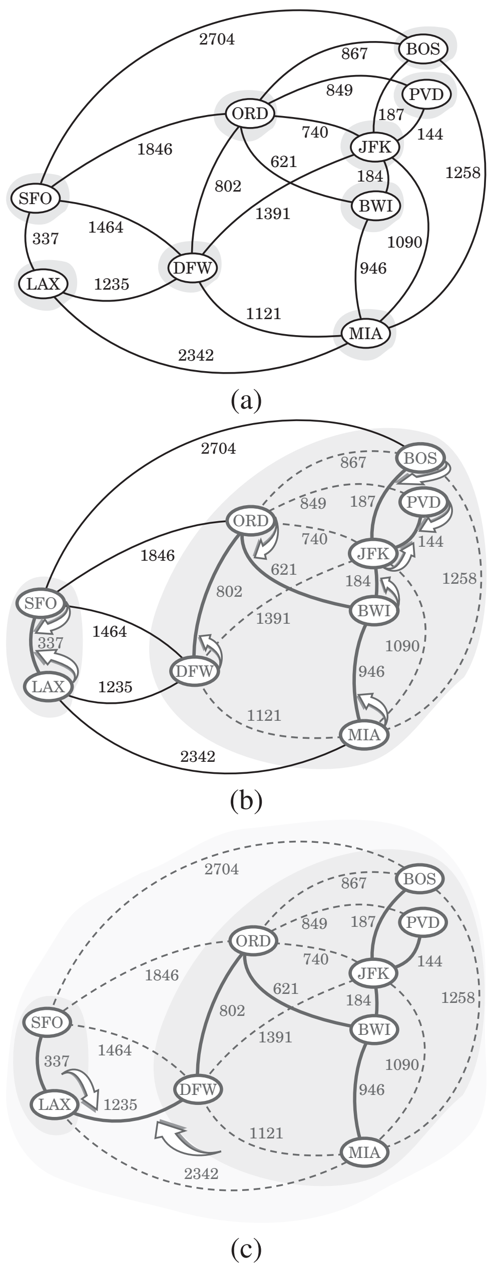


Figure 15.12: Example of an execution of Barůvka's algorithm. We show clusters as shaded regions. We highlight the edge chosen by each cluster with an arrow and we draw each such MST edge as a thick line. Edges determined not to be in the MST are shown dashed.

Why Is This Algorithm Correct?

Like Kruskal's algorithm, Barůvka's algorithm builds the minimum spanning tree by growing a number of clusters of vertices in a series of rounds, not just one cluster, as was done by the Prim-Jarník algorithm. But in Barůvka's algorithm, the clusters are grown by applying the crucial fact about minimum spanning trees from Theorem 15.2 to each cluster simultaneously. This approach adds many edges in each round.

In each iteration of Barůvka's algorithm, we choose the smallest-weight edge coming out of each connected component C_i of the current set T of minimum spanning tree edges. In each case, this edge is a valid choice, for if we consider a partitioning of V into the vertices in C_i and those outside of C_i, then the chosen edge e for C_i satisfies the condition of the crucial fact about minimum spanning trees from Theorem 15.2 for guaranteeing that e belongs to a minimum spanning tree. Moreover, because of our assumption that edge weights are unique, there will be just one such edge per cluster, and if two clusters are joined by the same minimum-weight edge, then they will both choose that edge.

Incidentally, if the edge weights in G are not initially unique, we can impose uniqueness by numbering the vertices 1 to n and taking the new weight of each edge $e = (u, v)$, written so that $u < v$, to be $(w(e), u, v)$, where $w(e)$ is the old weight of e and we use a lexicographic comparison rule for the new weights.

Analyzing Barůvka's Algorithm

Let us analyze the running time of Barůvka's algorithm (Algorithm 15.11). We can implement each round performing the searches to find the minimum-weight edge going out of each cluster by an exhaustive search through the adjacency lists of each vertex in each cluster. Thus, the total running time spent in searching for minimum-weight edges can be made to be $O(m)$, for it involves examining each edge (v, u) in G twice: once for v and once for u (since vertices are labeled with the number of the cluster they are in). The remaining computations in the main while-loop involve relabeling all the vertices, which takes $O(n)$ time, and traversing all the edges in T, which takes $O(n)$ time. Thus, each round in Barůvka's algorithm takes $O(m)$ time (since $n \leq m$). In each round of the algorithm, we choose one edge coming out of each cluster, and we then merge each new connected component of T into a new cluster. Thus, each old cluster of T must merge with at least one other old cluster of T. That is, in each round of Barůvka's algorithm, the total number of clusters is reduced by half. Therefore, the total number of rounds is $O(\log n)$; hence, the total running time of Barůvka's algorithm is $O(m \log n)$. We summarize:

Theorem 15.7: *Barůvka's algorithm computes a minimum spanning tree for a connected weighted graph G with n vertices and m edges in $O(m \log n)$ time.*

15.5 Exercises

Reinforcement

R-15.1 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Kruskal's algorithm on this graph. (Note that there is only one minimum spanning tree for this graph.)

R-15.2 Repeat the previous problem for the Prim-Jarník algorithm.

R-15.3 Repeat the previous problem for Barůvka's algorithm.

R-15.4 Describe the meaning of the graphical conventions used in Figures 15.5 through 15.7, illustrating Kruskal's algorithm. What do thick lines and dashed lines signify?

R-15.5 Repeat Exercise R-15.4 for Figures 15.9 and 15.10, illustrating the Prim-Jarník algorithm.

R-15.6 Repeat Exercise R-15.4 for Figure 15.12, illustrating Barůvka's algorithm.

R-15.7 Give an alternative pseudocode description of Kruskal's algorithm that makes explicit use of the union and find operations.

R-15.8 Why do all the MST algorithms discussed in this chapter still work correctly even if the graph has negative-weight edges, and even negative-weight cycles?

R-15.9 Give an example of weighted, connected, undirected graph, G, such that the minimum spanning tree for G is different from every shortest-path tree rooted at a vertex of G.

R-15.10 Let G be a weighted, connected, undirected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in the minimum spanning tree for G such that e has one endpoint in V_1 and the other in V_2. Give an example that shows that e is not necessarily the smallest-weight edge that has one endpoint in V_1 and the other in V_2.

Creativity

C-15.1 Suppose G is a weighted, connected, undirected graph and e is a smallest-weight edge in G. Show that there is a minimum spanning tree of G that contains e.

C-15.2 Suppose G is a weighted, connected, undirected, simple graph and e is a largest-weight edge in G. Prove or disprove the claim that there is no minimum spanning tree of G that contains e.

C-15.3 Suppose G is an undirected, connected, weighted graph such that the edges in G have distinct edge weights. Show that the minimum spanning tree for G is unique.

C-15.4 Suppose G is an undirected, connected, weighted graph such that the edges in G have distinct positive edge weights. Show that the minimum spanning tree for G is unchanged even if we square all the edge weights in G, that is, G has the same set of edges in its minimum spanning tree even if we replace the weight, $w(e)$, of each edge e in G, with $w(e)^2$.

C-15.5 Suppose G is an undirected, connected, weighted graph such that the edges in G have distinct edge weights, which may be positive or negative. Show that the minimum spanning tree for G may be changed if we square all the edge weights in G, that is, G may have a different set of edges in its minimum spanning tree if we replace the weight, $w(e)$, of each edge e in G, with $w(e)^2$.

C-15.6 Show how to modify the Prim-Jarník algorithm to run in $O(n^2)$ time.

C-15.7 Show how to modify Barůvka's algorithm so that it runs in $O(n^2)$ time.

Hint: Consider contracting the representation of the graph so that each connected component "cluster" is reduced to a single "super" vertex, with self-loops and parallel edges removed with each iteration. Then characterize the running time using a recurrence relation involving only n and show that this running time is $O(n^2)$.

C-15.8 Suppose you are given a weighted graph, G, with n vertices and m edges, such that the weight of each edge in G is a real number chosen independently at random from the interval $[0, 1]$. Show that the expected running time of the Prim-Jarník algorithm on G is $O(n \log^2 n + m)$, assuming the priority queue, Q, is implemented with a heap, as described in the chapter.

Hint: Think about first bounding the expected number of times the Prim-Jarník algorithm would change the value of $D[v]$ for any given vertex, v, in G.

C-15.9 Suppose G is a weighted, connected, undirected graph with each edge having a unique integer weight, which may be either positive or negative. Let G' be the same graph as G, but with each edge, e, in G' having weight that is 1 greater than e's weight in G. Show that G and G' have the same minimum spanning tree.

C-15.10 Suppose Joseph Kruskal had an evil twin, named Peter, who designed an algorithm that takes the exact opposite approach from his brother's algorithm for finding an MST in an undirected, weighted, connected graph, G. Also, for the sake of simplicity, suppose the edges of G have distinct weights. Peter's algorithm is as follows: consider the edges of G by decreasing weights. For each edge, e, in this order, if the removal of e from G would not disconnect G, then remove e from G. Peter claims that the graph that remains at the end of his algorithm is a minimum spanning tree. Prove or disprove Peter's claim.

C-15.11 Suppose Vojtěch Jarník had an evil twin, named Stanislaw, who designed a divide-and-conquer algorithm for finding minimum spanning trees. Suppose G is an undirected, connected, weighted graph, and, for the sake of simplicity, let us further suppose that the weights of the edges in G are distinct. Stanislaw's algorithm, MinTree(G), is as follows: If G is a single vertex, then it just returns, outputting nothing. Otherwise, it divides the set of vertices of G into two sets, V_1 and V_2, of equal size (plus or minus one vertex). Let e be the minimum-weight edge in G that connects V_1 and V_2. Output e as belonging to the minimum spanning tree. Let G_1 be the subgraph of G induced by V_1 (that is, G_1 consists of the vertices in V_1 plus all the edges of G that connect pairs of vertices in V_1). Similarly, let G_2 be the subgraph of G induced by V_2. The algorithm then recursively calls MinTree(G_1) and MinTree(G_2). Stanislaw claims that the edges output by his algorithm are exactly the edges of the minimum spanning tree of G. Prove or disprove Stanislaw's claim.

Applications

A-15.1 Suppose you are a manager in the IT department for the government of a corrupt dictator, who has a collection of computers that need to be connected together to create a communication network for his spies. You are given a weighted graph, G, such that each vertex in G is one of these computers and each edge in G is a pair of computers that could be connected with a communication line. It is your job to decide how to connect the computers. Suppose now that the CIA has approached you and is willing to pay you various amounts of money for you to choose some of these edges to belong to this network (presumably so that they can spy on the dictator). Thus, for you, the weight of each edge in G is the amount of money, in U.S. dollars, that the CIA will pay you for using that edge in the communication network. Describe an efficient algorithm, therefore, for finding a ***maximum spanning tree*** in G, which would maximize the money you can get from the CIA for connecting the dictator's computers in a spanning tree. What is the running time of your algorithm?

A-15.2 Suppose you are given a diagram of a telephone network, which is a graph G whose vertices represent switching centers, and whose edges represent communication lines between two centers. The edges are marked by their bandwidth, that is, the maximum speed, in bits per second, that information can be transmitted along that communication line. The bandwidth of a path in G is the bandwidth of its lowest-bandwidth edge. Give an algorithm that, given a diagram and two switching centers a and b, will output the maximum bandwidth of a path between a and b. What is the running time of your algorithm?

A-15.3 Suppose NASA wants to link n stations spread out over the solar system using ***free-space optical communication***, which is a technology that involves shooting lasers through space between pairs of stations to facilitate communication. Naturally, the energy needed to connect two stations in this way depends on the distance between them, with some connections requiring more energy than others. Therefore, each pair of stations has a different known energy that is needed to allow this pair of stations to communicate. NASA wants to connect all these stations together using the minimum amount of energy possible. Describe an algorithm for constructing such a communication network in $O(n^2)$ time.

A-15.4 Imagine that you just joined a company, GT&T, which set up its computer network a year ago for linking together its n offices spread across the globe. You have reviewed the work done at that time, and you note that they modeled their network as a connected, undirected graph, G, with n vertices, one for each office, and m edges, one for each possible connection. Furthermore, you note that they gave a weight, $w(e)$, for each edge in G that was equal to the annual rent that it costs to use that edge for communication purposes, and then they computed a minimum spanning tree, T, for G, to decide which of the m edges in G to lease. Suppose now that it is time renew the leases for connecting the vertices in G and you notice that the rent for one of the connections not used in T has gone down. That is, the weight, $w(e)$, of an edge in G that is not in T has been reduced. Describe an $O(n+m)$-time algorithm to update T to find a new minimum spanning, T', for G given the change in weight for the edge e.

A-15.5 Consider a scenario where you have a set of n players in a multiplayer game that we would like to connect to form a team. You are given a connected, weighted, undirected graph, G, with n vertices and m edges, which represents the n players and the m possible connections that can be made to link them together in a communicating team (that is, a connected subgraph of G). In this case, the weight on each edge, e, in G is an integer in the range $[1, n^2]$, which represents the amount of game points required to use the edge e for communication. Describe how to find a minimum spanning tree for G and thereby form this team with minimum cost, using an algorithm that runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is the slow-growing inverse of the Ackermann function.

A-15.6 Suppose you have n rooms that you would like to connect in a communication network in one of the dormitories of Flash University. You have modeled the problem using a connected, undirected graph, G, where each of the n vertices in G is a room and each of the m edges in G is a possible connection that you can form by running a cable between the rooms represented by the end vertices of that edge. In this case, however, there are only two kinds of cables that you may possibly use, a 12-foot cable, which costs \$10 and is sufficient to connect some pairs of rooms, and a 50-foot cable, which costs \$30 and can be used to connect pairs of rooms that are farther apart. Describe an algorithm for finding a minimum-cost spanning tree for G in $O(n + m)$ time.

A-15.7 Suppose, for a data approximation application, you are given a set of n points in the plane and you would like to partition these points into two sets, A and B, such that each point in A is as close or closer to another point in A than it is to any point in B, and vice versa. Describe an efficient algorithm for doing this partitioning.

Chapter Notes

The first known minimum spanning tree algorithm is due to Barůvka [21], and was published in 1926. The Prim-Jarník algorithm was first published in Czech by Jarník [111] in 1930 and in English in 1957 by Prim [174]. Kruskal published his minimum spanning tree algorithm in 1956 [137]. The reader interested in further study of the history of the minimum spanning tree problem is referred to the paper by Graham and Hell [92]. The current asymptotically fastest minimum spanning tree algorithm is a randomized method of Karger, Klein, and Tarjan [119] that runs in $O(m)$ expected time.

Incidentally, the running time for the Prim-Jarník algorithm can be improved to be $O(n \log n + m)$ by implementing the queue Q with either of two more sophisticated data structures, the "Fibonacci Heap" [76] or the "Relaxed Heap" [58]. The reader interested in these implementations is referred to the papers that describe the implementation of these structures, and how they can be applied to minimum spanning tree problems.