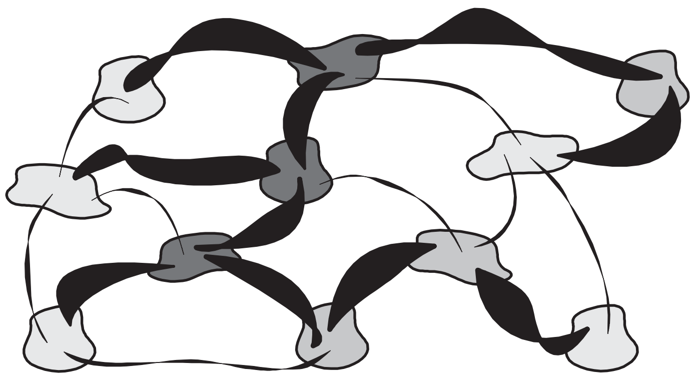
Chapter 7 Weighted Graphs



Contents

As we saw in the previous chapter, the breadth-first search strategy can be used to find a shortest path from some starting vertex to every other vertex in a connected graph. This approach makes sense in cases where each edge is as good as any other, but there are many situations where this approach is not appropriate.

For example, we might be using a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers. In this case, it is probably not appropriate for all the edges to be equal to each other, for some connections in a computer network are typically much faster than others (for example, some edges might represent slow phone-line connections while others might represent high-speed, fiber-optic connections). Likewise, we might want to use a graph to represent the roads between cities, and we might be interested in finding the fastest way to travel cross-country. In this case, it is again probably not appropriate for all the edges to be equal to each other, for some intercity distances will likely be much larger than others. Thus, it is natural to consider graphs whose edges are not weighted equally.

In this chapter, we study weighted graphs. A ***weighted graph*** is a graph that has a numeric label $w(e)$ associated with each edge e, called the ***weight*** of edge e. Edge weights can be integers, rational numbers, or real numbers, which represent a concept such as distance, connection costs, or affinity. We show an example of a weighted graph in Figure 7.1.

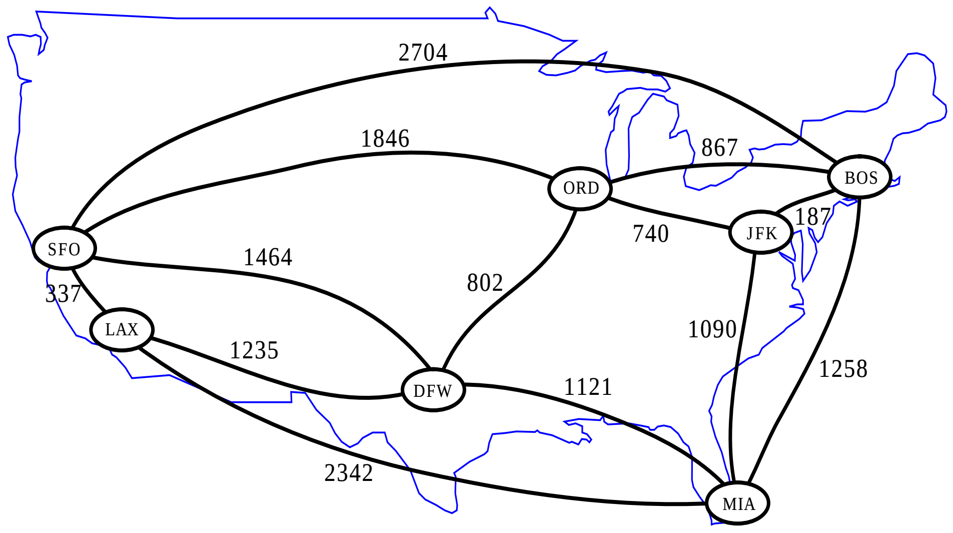


Figure 7.1: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum weight path in the graph from JFK to LAX.

7.1 Single-Source Shortest Paths

Let G be a weighted graph. The ***length*** (or ***weight***) of a path P is the sum of the weights of the edges of P. That is, if P consists of edges $e_0, e_1, \ldots, e_{k-1}$ then the length of P, denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(e_i).$$

The ***distance*** from a vertex v to a vertex u in G, denoted $d(v,u)$, is the length of a minimum length path (also called ***shortest path***) from v to u, if such a path exists.

People often use the convention that $d(v,u) = +\infty$ if there is no path at all from v to u in G. Even if there is a path from v to u in G, the distance from v to u may not be defined, however, if there is a cycle in G whose total weight is negative. For example, suppose vertices in G represent cities, and the weights of edges in G represent how much money it costs to go from one city to another. If someone were willing to actually pay us to go from say JFK to ORD, then the "cost" of the edge (JFK,ORD) would be negative. If someone else were willing to pay us to go from ORD to JFK, then there would be a negative-weight cycle in G and distances would no longer be defined. That is, anyone can now build a path (with cycles) in G from any city A to another city B that first goes to JFK and then cycles as many times as he or she likes from JFK to ORD and back, before going on to B. The existence of such paths allows us to build arbitrarily low negative-cost paths (and in this case make a fortune in the process). But distances cannot be arbitrarily low negative numbers. Thus, any time we use edge weights to represent distances, we must be careful not to introduce any negative-weight cycles.

Suppose we are given a weighted graph G, and we are asked to find a shortest path from some vertex v to each other vertex in G, viewing the weights on the edges as distances. In this section, we explore efficient ways of finding all such ***single-source shortest paths***, if they exist.

The first algorithm we discuss is for the simple, yet common, case when all the edge weights in G are nonnegative (that is, $w(e) \geq 0$ for each edge e of G); hence, we know in advance that there are no negative-weight cycles in G. Recall that the special case of computing a shortest path when all weights are 1 was solved with the BFS traversal algorithm presented in Section 6.3.3.

There is an interesting approach for solving this ***single-source*** problem based on the ***greedy method*** design pattern (Section 5.1). Recall that in this pattern we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration. This paradigm can often be used in situations where we are trying to optimize some cost function over a collection of objects. We can add objects to our collection, one at a time, always picking the next one that optimizes the function from among those yet to be chosen.

7.1.1 Dijkstra's Algorithm

The main idea in applying the greedy method pattern to the single-source shortest-path problem is to perform a "weighted" breadth-first search starting at v. In particular, we can use the greedy method to develop an algorithm that iteratively grows a "cloud" of vertices out of v, with the vertices entering the cloud in order of their distances from v. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to v. The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from v to every other vertex of G. This approach is a simple, but nevertheless powerful, example of the greedy method design pattern.

A Greedy Method for Finding Shortest Paths

Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra's algorithm***. When applied to other graph problems, however, the greedy method may not necessarily find the best solution (such as in the so-called ***traveling salesman problem***, in which we wish to find the shortest path that visits all the vertices in a graph exactly once). Nevertheless, there are a number of situations in which the greedy method allows us to compute the best solution. In this chapter, we discuss two such situations: computing shortest paths and constructing minimum spanning trees.

In order to simplify the description of Dijkstra's algorithm, 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). We leave the description of Dijkstra's algorithm so that it works for a weighted directed graph as an exercise (R-7.2).

In Dijkstra's algorithm, the cost function we are trying to optimize in our application of the greedy method is also the function that we are trying to compute—the shortest path distance. This may at first seem like circular reasoning until we realize that we can actually implement this approach by using a "bootstrapping" trick, consisting of using an approximation to the distance function we are trying to compute, which in the end will be equal to the true distance.

Edge Relaxation

Let us define a label $D[u]$ for each vertex u of G, which we use to approximate the distance in G from v to u. The meaning of these labels is that $D[u]$ will always store the length of the best path we have found ***so far*** from v to u. Initially, $D[v] = 0$ and $D[u] = +\infty$ for each $u \neq v$, and we define the set C, which is our "***cloud***" of vertices, to initially be the empty set $\emptyset$. At each iteration of the algorithm, we select a vertex u not in C with smallest $D[u]$ label, and we pull u into C. In the very first iteration we will, of course, pull v into C. Once a new vertex u is pulled into C, we then update the label $D[z]$ of each vertex z that is adjacent to u and is outside of

C, to reflect the fact that there may be a new and better way to get to z via u. This update operation is known as a ***relaxation*** procedure, for it takes an old estimate and checks if it can be improved to get closer to its true value. (A metaphor for why we call this a relaxation comes from a spring that is stretched out and then "relaxed" back to its true resting shape.) In the case of Dijkstra's algorithm, the relaxation is performed for an edge (u,z), such that we have computed a new value of $D[u]$ and wish to see if there is a better value for $D[z]$ using the edge (u,z). The specific edge relaxation operation is as follows:

Edge Relaxation:

```
if $D[u] + w((u,z)) < D[z]$ then
    $D[z] \leftarrow D[u] + w((u,z))$.
```

Note that if the newly discovered path to z is no better than the old way, then we do not change $D[z]$.

The Details of Dijkstra's Algorithm

We give the pseudo-code for Dijkstra's algorithm in Algorithm 7.2. Note that we use a priority queue Q to store the vertices outside of the cloud C.

```
Algorithm DijkstraShortestPaths($G, v$):
   Input: A simple undirected weighted graph $G$ with nonnegative edge weights,
      and a distinguished vertex $v$ of $G$
   Output: A label $D[u]$, for each vertex $u$ of $G$, such that $D[u]$ is the distance from
      $v$ to $u$ in $G$
   $D[v] \leftarrow 0$
   for each vertex $u \neq v$ of $\vec{G}$ do
      $D[u] \leftarrow +\infty$
   Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
   while $Q$ is not empty do
      {pull a new vertex $u$ into the cloud}
      $u \leftarrow Q$.removeMin()
      for each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ do
         {perform the relaxation procedure on edge $(u,z)$}
         if $D[u] + w((u,z)) < D[z]$ then
            $D[z] \leftarrow D[u] + w((u,z))$
            Change to $D[z]$ the key of vertex $z$ in $Q$.
   return the label $D[u]$ of each vertex $u$
```

Algorithm 7.2: Dijkstra's algorithm for the single-source shortest path problem for a graph G, starting from a vertex v.

We illustrate several iterations of Dijkstra's algorithm in Figures 7.3 and 7.4.

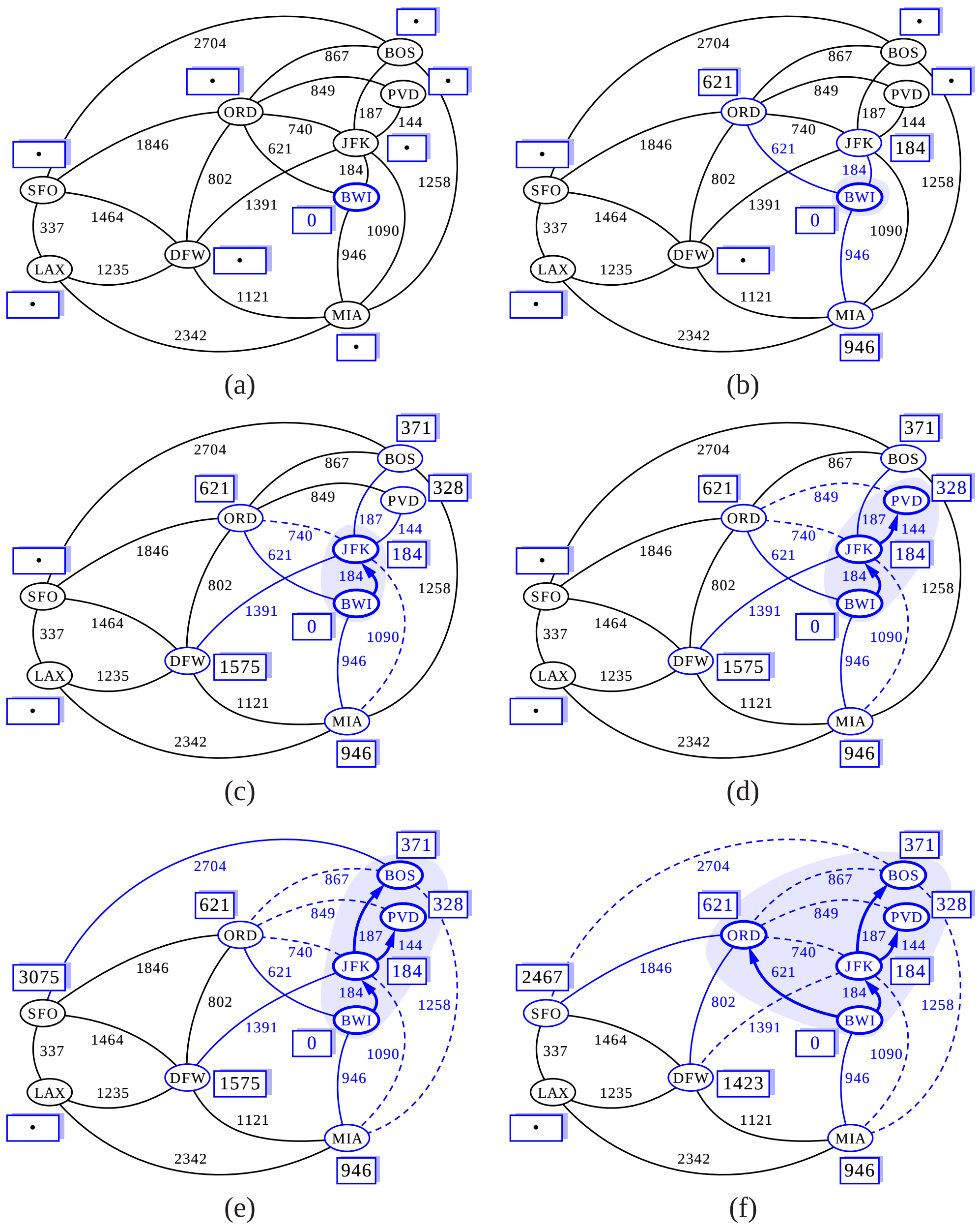


Figure 7.3: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex u stores the label $D[u]$. The symbol ● is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick arrows, and for each vertex u outside the "cloud" we show the current best edge for pulling in u with a solid line. (Continued in Figure 7.4.)

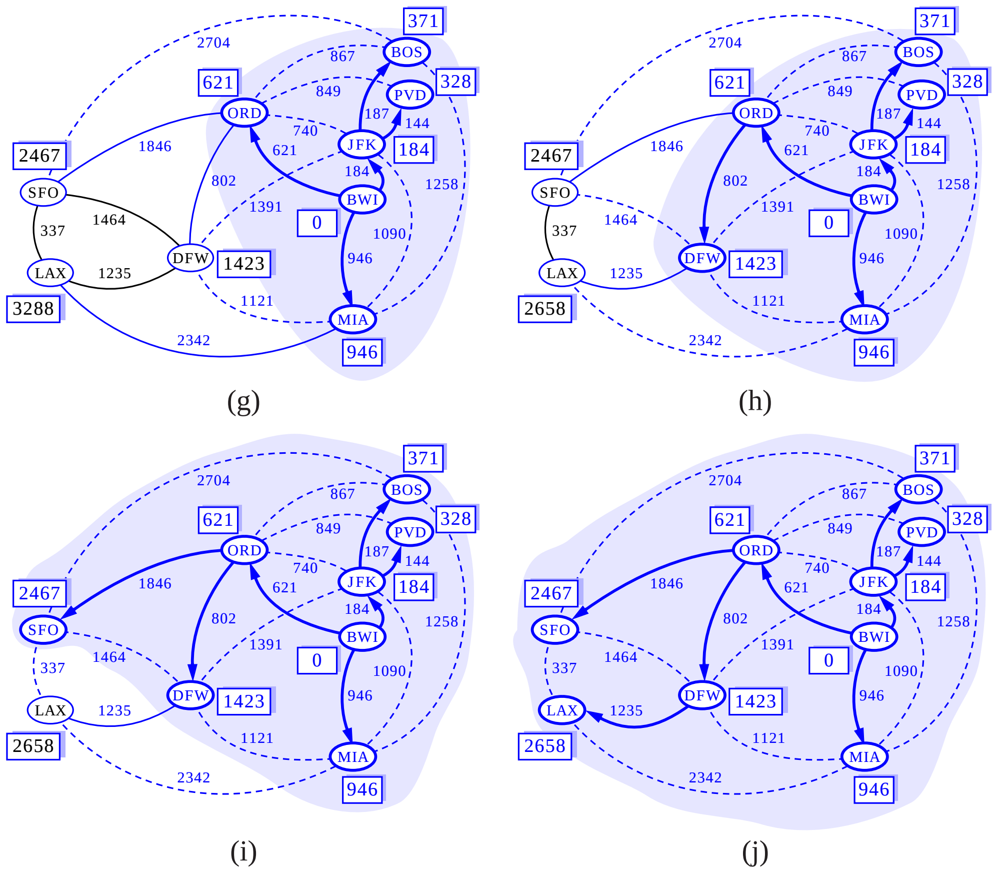


Figure 7.4: Visualization of Dijkstra's algorithm. (Continued from Figure 7.3.)

Why It Works

The interesting, and possibly even a little surprising, aspect of the Dijkstra algorithm is that, at the moment a vertex u is pulled into C, its label $D[u]$ stores the correct length of a shortest path from v to u. Thus, when the algorithm terminates, it will have computed the shortest-path distance from v to every vertex of G. That is, it will have solved the single-source shortest path problem.

It is probably not immediately clear why Dijkstra's algorithm correctly finds the shortest path from the start vertex v to each other vertex u in the graph. Why is it that the distance from v to u is equal to the value of the label $D[u]$ at the time vertex u is pulled into the cloud C (which is also the time u is removed from the priority queue Q)? The answer to this question depends on there being no negative-weight edges in the graph, for it allows the greedy method to work correctly, as we show in the lemma that follows.

Lemma 7.1: *In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v,u)$, the length of a shortest path from v to u.*

Proof: Suppose that $D[t] > d(v,t)$ for some vertex t in V, and let u be the ***first*** vertex the algorithm pulled into the cloud C (that is, removed from Q), such that $D[u] > d(v,u)$. There is a shortest path P from v to u (for otherwise $d(v,u) = +\infty = D[u]$). Therefore, let us consider the moment when u is pulled into C, and let z be the first vertex of P (when going from v to u) that is not in C at this moment. Let y be the predecessor of z in path P (note that we could have $y = v$). (See Figure 7.5.) We know, by our choice of z, that y is already in C at this point. Moreover, $D[y] = d(v,y)$, since u is the ***first*** incorrect vertex. When y was pulled into C, we tested (and possibly updated) $D[z]$ so that we had at that point

$$D[z] \le D[y] + w((y,z)) = d(v,y) + w((y,z)).$$

But since z is the next vertex on the shortest path from v to u, this implies that

$$D[z] = d(v,z).$$

But we are now at the moment when we are picking u, not z, to join C; hence,

$$D[u] \le D[z].$$

It should be clear that a subpath of a shortest path is itself a shortest path. Hence, since z is on the shortest path from v to u,

$$d(v,z) + d(z,u) = d(v,u).$$

Moreover, $d(z,u) \ge 0$ because there are no negative-weight edges. Therefore,

$$D[u] \le D[z] = d(v,z) \le d(v,z) + d(z,u) = d(v,u).$$

But this contradicts the definition of u; hence, there can be no such vertex u. ■

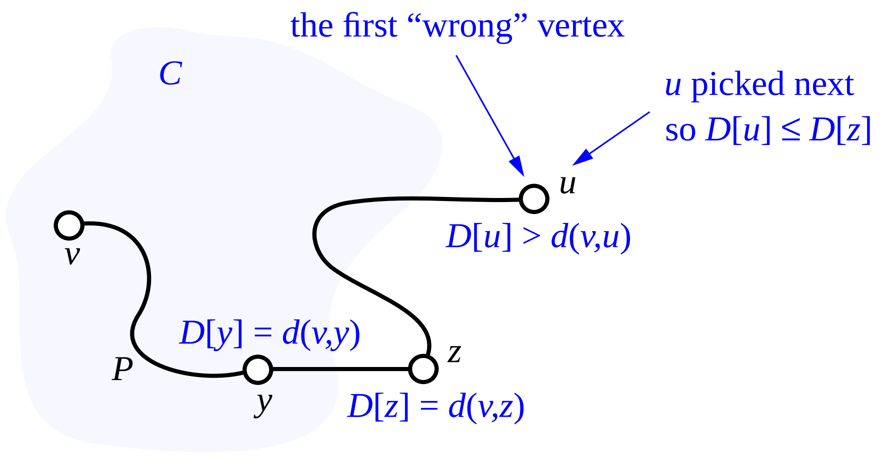


Figure 7.5: A schematic illustration for the justification of Theorem 7.1.

The Running Time of Dijkstra's Algorithm

In this section, we analyze the time complexity of Dijkstra's algorithm. We denote with n and m, the number of vertices and edges of the input graph G, respectively. We assume that the edge weights can be added and compared in constant time. Because of the high level of the description we gave for Dijkstra's algorithm in Algorithm 7.2, 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.

Let us first assume that we are representing the graph G using an adjacency list structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. It still does not settle all the details for the algorithm, however, for we must say more about how to implement the other main data structure in the algorithm—the priority queue Q.

An efficient implementation of the priority queue Q uses a heap (see Section 2.4.3). This allows us to extract the vertex u with smallest D label, by calling the removeMin method, in $O(\log n)$ time. As noted in the pseudo-code, each time we update a $D[z]$ label we need to update the key of z in the priority queue. If Q is implemented as a heap, then this key update can, for example, be done by first removing and then inserting z with its new key. If our priority queue Q supports the locator pattern (see Section 2.4.4), then we can easily implement such key updates in $O(\log n)$ time, since a locator for vertex z would allow Q to have immediate access to the item storing z in the heap (see Section 2.4.4). Assuming this implementation of Q, Dijkstra's algorithm runs in $O((n+m)\log n)$ time.

Referring back to Algorithm 7.2, the details of the running-time analysis are as follows:

- Inserting all the vertices in Q with their initial key value can be done in $O(n \log n)$ time by repeated insertions, or in $O(n)$ time using bottom-up heap construction (see Section 2.4.4).
- At each iteration of the **while** loop, we spend $O(\log n)$ time to remove vertex u from Q, and $O(\deg(v)\log n)$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the **while** loop is
$$\sum_{v \in G} (1+\deg(v))\log n,$$
which is $O((n+m)\log n)$ by Theorem 6.6.

Thus, we have the following.

Theorem 7.2: *Given a weighted n-vertex graph G with m edges, each with a nonnegative weight, Dijkstra's algorithm can be implemented to find all shortest paths from a vertex v in G in $O(m \log n)$ time.*

Note that if we wish to express the above running time as a function of n only, then it is $O(n^2 \log n)$ in the worst case, since we have assumed that G is simple.

An Alternative Implementation for Dijkstra's Algorithm

Let us now consider an alternative implementation for the priority queue Q using an unsorted sequence. This, of course, requires that we spend $\Omega(n)$ time to extract the minimum element, but it allows for very fast key updates, provided Q supports the locator pattern (Section 2.4.4). Specifically, we can implement each key update done in a relaxation step in $O(1)$ time—we simply change the key value once we locate the item in Q to update. Hence, this implementation results in a running time that is $O(n^2 + m)$, which can be simplified to $O(n^2)$ since G is simple.

Comparing the Two Implementations

We have two choices for implementing the priority queue in Dijkstra's algorithm: a locator-based heap implementation, which yields $O(m \log n)$ running time, and a locator-based unsorted sequence implementation, which yields an $O(n^2)$-time algorithm. Since both implementations would be fairly simple to code up, they are about equal in terms of the programming sophistication needed. These two implementations are also about equal in terms of the constant factors in their worst-case running times. Looking only at these worst-case times, we prefer the heap implementation when the number of edges in the graph is small (that is, when $m < n^2/\log n$), and we prefer the sequence implementation when the number of edges is large (that is, when $m > n^2/\log n$).

Theorem 7.3: *Given a simple weighted graph G with n vertices and m edges, such that the weight of each edge is nonnegative, and a vertex v of G, Dijkstra's algorithm computes the distance from v to all other vertices of G in $O(m \log n)$ time, or, alternatively, in $O(n^2)$ time.*

In Exercise R-7.3, we explore how to modify Dijkstra's algorithm to output a tree T rooted at v, such that the path in T from v to a vertex u is a shortest path in G from v to u. In addition, extending Dijkstra's algorithm for directed graphs is fairly straightforward. We cannot extend Dijkstra's algorithm to work on graphs with negative-weight edges, however, as Figure 7.6 illustrates.

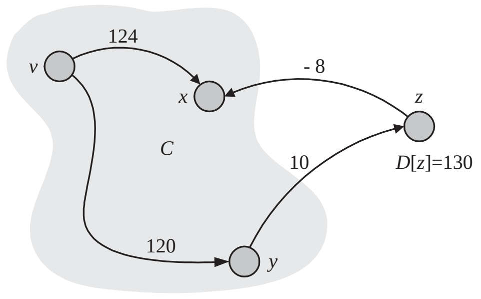


Figure 7.6: An illustration of why Dijkstra's algorithm fails for graphs with negative-weight edges. Bringing z into C and performing edge relaxations will invalidate the previously computed shortest path distance (124) to x.

7.1.2 The Bellman-Ford Shortest Paths Algorithm

There is another algorithm, which is due to Bellman and Ford, that can find shortest paths in graphs that have negative-weight edges. We must, in this case, insist that the graph be directed, for otherwise any negative-weight undirected edge would immediately imply a negative-weight cycle, where we traverse this edge back and forth in each direction. We cannot allow such edges, since a negative cycle invalidates the notion of distance based on edge weights.

Let $\vec{G}$ be a weighted directed graph, possibly with some negative-weight edges. The Bellman-Ford algorithm for computing the shortest-path distance from some vertex v in $\vec{G}$ to every other vertex in $\vec{G}$ is very simple. It shares the notion of edge relaxation from Dijkstra's algorithm, but does not use it in conjunction with the greedy method (which would not work in this context; see Exercise C-7.2). That is, as in Dijkstra's algorithm, the Bellman-Ford algorithm uses a label $D[u]$ that is always an upper bound on the distance $d(v,u)$ from v to u, and which is iteratively "relaxed" until it exactly equals this distance.

The Details of the Bellman-Ford Algorithm

The Bellman-Ford method is shown in Algorithm 7.7. It performs $n-1$ times a relaxation of every edge in the digraph. We illustrate an execution of the Bellman-Ford algorithm in Figure 7.8.

```
Algorithm BellmanFordShortestPaths($\vec{G}, v$):
   Input: A weighted directed graph $\vec{G}$ with $n$ vertices, and a vertex $v$ of $\vec{G}$
   Output: A label $D[u]$, for each vertex $u$ of $\vec{G}$, such that $D[u]$ is the distance from
      $v$ to $u$ in $\vec{G}$, or an indication that $\vec{G}$ has a negative-weight cycle
   $D[v] \leftarrow 0$
   for each vertex $u \neq v$ of $\vec{G}$ do
      $D[u] \leftarrow +\infty$
   for $i \leftarrow 1$ to $n-1$ do
      for each (directed) edge $(u,z)$ outgoing from $u$ do
         {Perform the relaxation operation on $(u,z)$}
         if $D[u] + w((u,z)) < D[z]$ then
            $D[z] \leftarrow D[u] + w((u,z))$
   if there are no edges left with potential relaxation operations then
      return the label $D[u]$ of each vertex $u$
   else
      return "$\vec{G}$ contains a negative-weight cycle"
```

Algorithm 7.7: The Bellman-Ford single-source shortest-path algorithm, which allows for negative-weight edges.

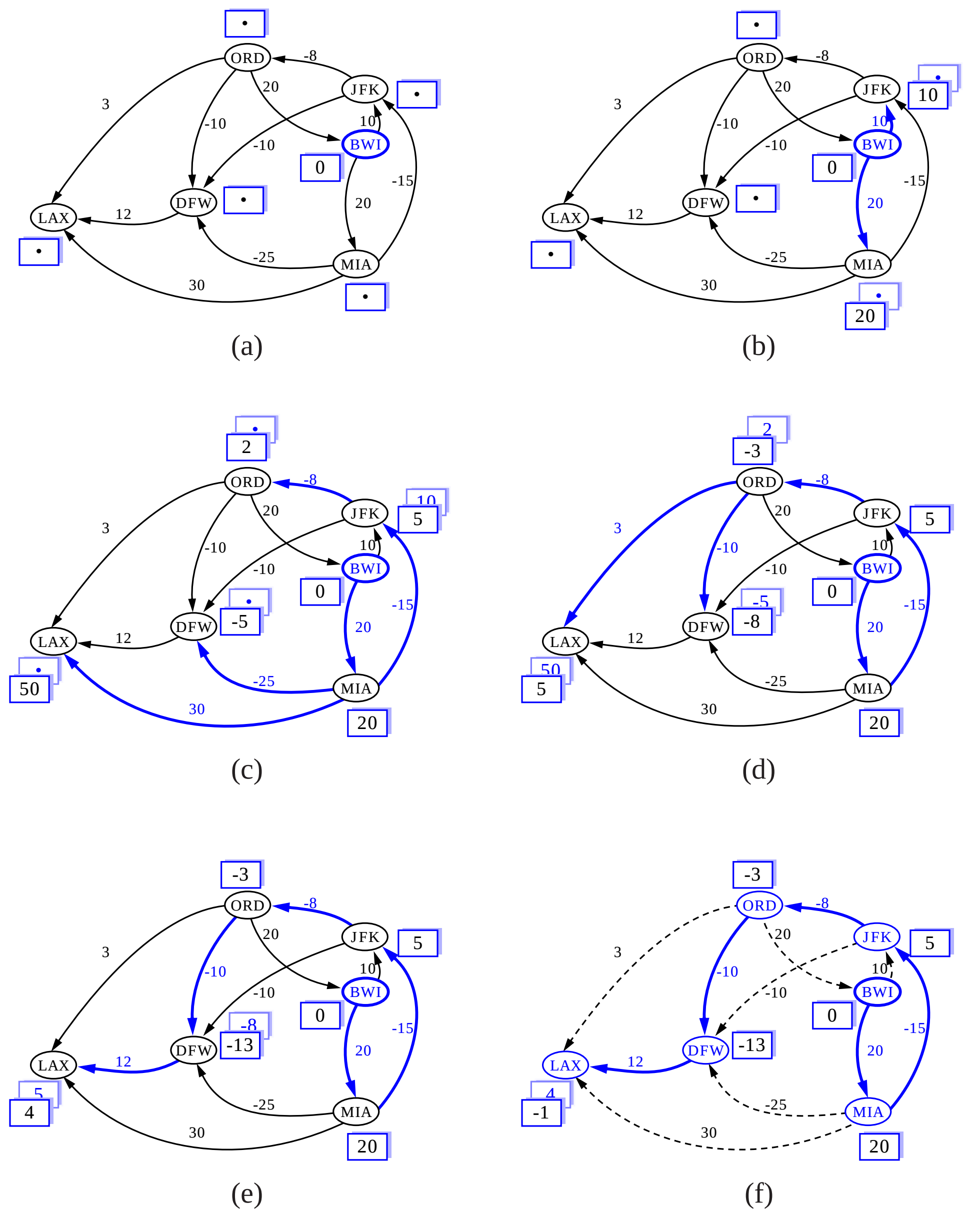


Figure 7.8: An illustration of an application of the Bellman-Ford algorithm. The start vertex is BWI. A box next to each vertex u stores the label $D[u]$, with “shadows” showing values revised during relaxations; the thick edges are causing such relaxations.

Lemma 7.4: *If at the end of the execution of Algorithm 7.7 there is an edge (u,z) that can be relaxed (that is, $D[u]+w((u,z)) < D[z]$), then the input digraph $\vec{G}$ contains a negative-weight cycle. Otherwise, $D[u] = d(v,u)$ for each vertex u in $\vec{G}$.*

Proof: For the sake of this proof, let us introduce a new notion of distance in a digraph. Specifically, let $d_i(v,u)$ denote the length of a path from v to u that is shortest among all paths from v to u that contain at most i edges. We call $d_i(v,u)$ the ***i*-edge distance** from v to u. We claim that after iteration i of the main for-loop in the Bellman-Ford algorithm $D[u] = d_i(v,u)$ for each vertex in $\vec{G}$. This is certainly true before we even begin the first iteration, for $D[v] = 0 = d_0(v,v)$ and, for $u \neq v$, $D[u] = +\infty = d_0(v,u)$. Suppose this claim is true before iteration i (we will now show that if this is the case, then this claim will be true after iteration i as well). In iteration i, we perform a relaxation step for every edge in the digraph. The i-edge distance $d_i(v,u)$, from v to a vertex u, is determined in one of two ways. Either $d_i(v,u) = d_{i-1}(v,u)$ or $d_i(v,u) = d_{i-1}(v,z) + w((z,u))$ for some vertex z in $\vec{G}$. Because we do a relaxation for ***every*** edge of $\vec{G}$ in iteration i, if it is the former case, then after iteration i we have $D[u] = d_{i-1}(v,u) = d_i(v,u)$, and if it is the latter case, then after iteration i we have $D[u] = D[z] + w((z,u)) = d_{i-1}(v,z) + w((z,u)) = d_i(v,u)$. Thus, if $D[u] = d_{i-1}(v,u)$ for each vertex u before iteration i, then $D[u] = d_i(v,u)$ for each vertex u after iteration i.

Therefore, after $n-1$ iterations, $D[u] = d_{n-1}(v,u)$ for each vertex u in $\vec{G}$. Now observe that if there is still an edge in $\vec{G}$ that can be relaxed, then there is some vertex u in $\vec{G}$, such that the n-edge distance from v to u is less than the $(n-1)$-edge distance from v to u, that is, $d_n(v,u) < d_{n-1}(v,u)$. But there are only n vertices in $\vec{G}$; hence, if there is a shortest n-edge path from v to u, it must repeat some vertex z in $\vec{G}$ twice. That is, it must contain a cycle. Moreover, since the distance from a vertex to itself using zero edges is 0 (that is, $d_0(z,z) = 0$), this cycle must be a negative-weight cycle. Thus, if there is an edge in $\vec{G}$ that can still be relaxed after running the Bellman-Ford algorithm, then $\vec{G}$ contains a negative-weight cycle. If, on the other hand, there is no edge in $\vec{G}$ that can still be relaxed after running the Bellman-Ford algorithm, then $\vec{G}$ does not contain a negative-weight cycle. Moreover, in this case, every shortest path between two vertices will have at most $n-1$ edges; hence, for each vertex u in $\vec{G}$, $D[u] = d_{n-1}(v,u) = d(v,u)$. ■

Thus, the Bellman-Ford algorithm is correct and even gives us a way of telling when a digraph contains a negative-weight cycle. The running time of the Bellman-Ford algorithm is easy to analyze. We perform the main for-loop $n-1$ times, and each such loop involves spending $O(1)$ time for each edge in $\vec{G}$. Therefore, the running time for this algorithm is $O(nm)$. We summarize as follows:

Theorem 7.5: *Given a weighted directed graph $\vec{G}$ with n vertices and m edges, and a vertex v of $\vec{G}$, the Bellman-Ford algorithm computes the distance from v to all other vertices of G or determines that $\vec{G}$ contains a negative-weight cycle in $O(nm)$ time.*

7.1.3 Shortest Paths in Directed Acyclic Graphs

As mentioned above, both Dijkstra's algorithm and the Bellman-Ford algorithm work for directed graphs. We can solve the single-source shortest paths problem faster than these algorithms can, however, if the digraph has no directed cycles, that is, it is a weighted directed acyclic graph (DAG).

Recall from Section 6.4.4 that a topological ordering of a DAG $\vec{G}$ is a listing of its vertices $(v_1, v_2, \ldots, v_n)$, such that if (v_i, v_j) is an edge in $\vec{G}$, then $i < j$. Also, recall that we can use the depth-first search algorithm to compute a topological ordering of the n vertices in an m-edge DAG $\vec{G}$ in $O(n+m)$ time. Interestingly, given a topological ordering of such a weighted DAG $\vec{G}$, we can compute all shortest paths from a given vertex v in $O(n+m)$ time.

The Details for Computing Shortest Paths in a DAG

The method, which is given in Algorithm 7.9, involves visiting the vertices of $\vec{G}$ according to the topological ordering, relaxing the outgoing edges with each visit.

```
Algorithm DAGShortestPaths($\vec{G}, s$):
  Input: A weighted directed acyclic graph (DAG) $\vec{G}$ with $n$ vertices and $m$ edges,
    and a distinguished vertex $s$ in $\vec{G}$
  Output: A label $D[u]$, for each vertex $u$ of $\vec{G}$, such that $D[u]$ is the distance from
    $v$ to $u$ in $\vec{G}$
  Compute a topological ordering $(v_1, v_2, \ldots, v_n)$ for $\vec{G}$
  $D[s] \leftarrow 0$
  for each vertex $u \neq s$ of $\vec{G}$ do
    $D[u] \leftarrow +\infty$
  for $i \leftarrow 1$ to $n-1$ do
    {Relax each outgoing edge from $v_i$}
    for each edge $(v_i, u)$ outgoing from $v_i$ do
      if $D[v_i] + w((v_i, u)) < D[u]$ then
        $D[u] \leftarrow D[v_i] + w((v_i, u))$
  Output the distance labels $D$ as the distances from $s$.
```

Algorithm 7.9: Shortest path algorithm for a directed acyclic graph.

The running time of the shortest path algorithm for a DAG is easy to analyze. Assuming the digraph is represented using an adjacency list, we can process each vertex in constant time plus an additional time proportional to the number of its outgoing edges. In addition, we have already observed that computing the topological ordering of the vertices in $\vec{G}$ can be done in $O(n+m)$ time. Thus, the entire algorithm runs in $O(n+m)$ time. We illustrate this algorithm in Figure 7.10.

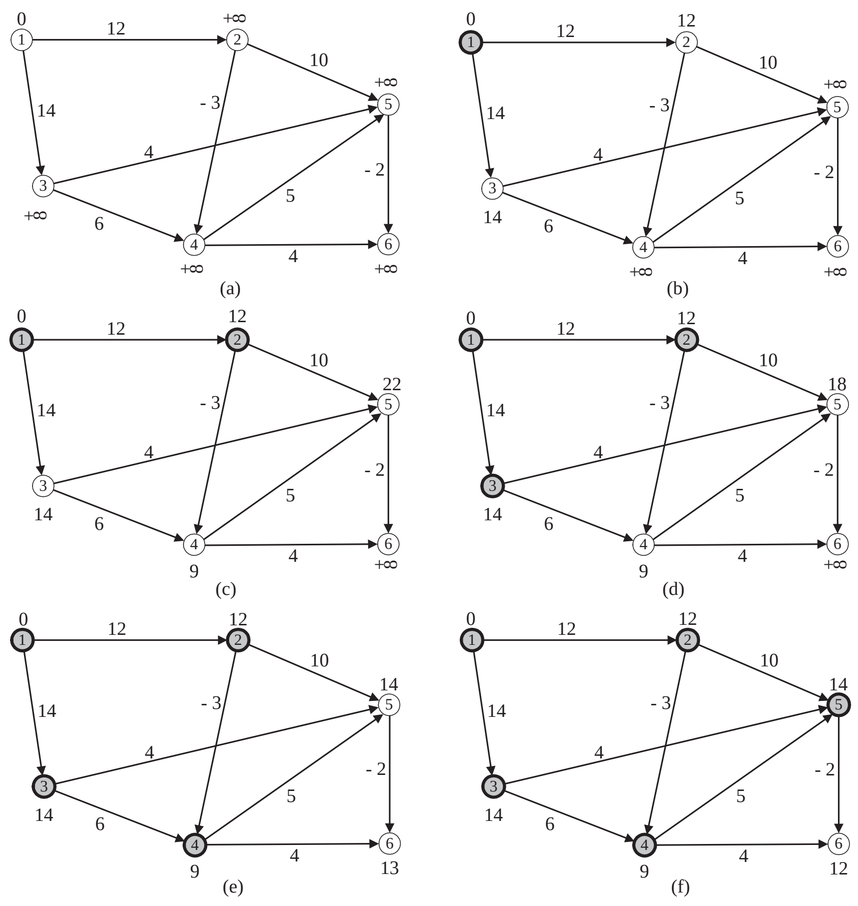


Figure 7.10: An illustration of the shortest-path algorithm for a DAG.

Theorem 7.6: DAGShortestPaths *computes the distance from a start vertex s to each other vertex in a directed n-vertex graph $\vec{G}$ with m edges in $O(n+m)$ time.*

Proof: Suppose, for the sake of a contradiction, that v_i is the first vertex in the topological ordering such that $D[v_i]$ is not the distance from s to v_i. First, note that $D[v_i] < +\infty$, for the initial D value for each vertex other than s is $+\infty$ and the value of a D label is only ever lowered if a path from s is discovered. Thus, if $D[v_j] = +\infty$, then v_j is unreachable from s. Therefore, v_i is reachable from s, so there is a shortest path from s to v_i. Let v_k be the penultimate vertex on a shortest path from s to v_i. Since the vertices are numbered according to a topological ordering, we have that $k < i$. Thus, $D[v_k]$ is correct (we may possibly have $v_k = s$). But when v_k is processed, we relax each outgoing edge from v_k, including the edge on the shortest path from v_k to v_i. Thus, $D[v_i]$ is assigned the distance from s to v_i. But this contradicts the definition of v_i; hence, no such vertex v_i can exist. ■

7.2 All-Pairs Shortest Paths

Suppose we wish to compute the shortest path distance between every pair of vertices in a directed graph $\vec{G}$ with n vertices and m edges. Of course, if $\vec{G}$ has no negative-weight edges, then we could run Dijkstra's algorithm from each vertex in $\vec{G}$ in turn. This approach would take $O(n(n+m)\log n)$ time, assuming $\vec{G}$ is represented using an adjacency list structure. In the worst case, this bound could be as large as $O(n^3 \log n)$. Likewise, if $\vec{G}$ contains no negative-weight cycles, then we could run the Bellman-Ford algorithm starting from each vertex in $\vec{G}$ in turn. This approach would run in $O(n^2 m)$ time, which, in the worst case, could be as large as $O(n^4)$. In this section, we consider algorithms for solving the all-pairs shortest path problem in $O(n^3)$ time, even if the digraph contains negative-weight edges (but not negative-weight cycles).

7.2.1 A Dynamic Programming Shortest Path Algorithm

The first all-pairs shortest path algorithm we discuss is a variation on an algorithm we have given earlier in this book, namely, the Floyd-Warshall algorithm for computing the transitive closure of a directed graph (Algorithm 6.16).

Let $\vec{G}$ be a given weighted directed graph. We number the vertices of $\vec{G}$ arbitrarily as $(v_1, v_2, \ldots, v_n)$. As in any dynamic programming algorithm (Section 5.3), the key construct in the algorithm is to define a parameterized cost function that is easy to compute and also allows us to ultimately compute a final solution. In this case, we use the cost function, $D^k_{i,j}$, which is defined as the distance from v_i to v_j using only intermediate vertices in the set $\{v_1, v_2, \ldots, v_k\}$. Initially,

$$D^0_{i,j} = \begin{cases} 0 & \text{if } i = j \\ w((v_i, v_j)) & \text{if } (v_i, v_j) \text{ is an edge in } \vec{G} \\ +\infty & \text{otherwise.} \end{cases}$$

Given this parameterized cost function $D^k_{i,j}$, and its initial value $D^0_{i,j}$, we can then easily define the value for an arbitrary $k > 0$ as

$$D^k_{i,j} = \min\{D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}\}.$$

In other words, the cost for going from v_i to v_j using vertices numbered 1 through k is equal to the shorter of two possible paths. The first path is simply the shortest path from v_i to v_j using vertices numbered 1 through $k-1$. The second path is the sum of the costs of the shortest path from v_i to v_k using vertices numbered 1 through $k-1$ and the shortest path from v_k to v_j using vertices numbered 1 through $k-1$. Moreover, there is no other shorter path from v_i to v_j using vertices of $\{v_1, v_2, \ldots, v_k\}$ than these two. If there was such a shorter path and it excluded v_k, then it would violate the definition of $D^{k-1}_{i,j}$, and if there was such a shorter path and it included v_k, then it would violate the definition of $D^{k-1}_{i,k}$ or $D^{k-1}_{k,j}$. In fact, note

```
Algorithm AllPairsShortestPaths($\vec{G}$):
    Input: A simple weighted directed graph $\vec{G}$ without negative-weight cycles
    Output: A numbering $v_1, v_2, \ldots, v_n$ of the vertices of $\vec{G}$ and a matrix $D$, such
      that $D[i,j]$ is the distance from $v_i$ to $v_j$ in $\vec{G}$

    let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            if $i = j$ then
                $D^0[i,i] \leftarrow 0$
            if $(v_i, v_j)$ is an edge in $\vec{G}$ then
                $D^0[i,j] \leftarrow w((v_i, v_j))$
            else
                $D^0[i,j] \leftarrow +\infty$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $D^k[i,j] \leftarrow \min\{D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j]\}$
    return matrix $D^n$
```

Algorithm 7.11: A dynamic programming algorithm to compute all-pairs shortest path distances in a digraph without negative cycles.

that this argument still holds even if there are negative cost edges in $\vec{G}$, just so long as there are no negative cost cycles. In Algorithm 7.11, we show how this cost-function definition allows us to build an efficient solution to the all-pairs shortest path problem.

The running time for this dynamic programming algorithm is clearly $O(n^3)$. Thus, we have the following theorem

Theorem 7.7: *Given a simple weighted directed graph $\vec{G}$ with n vertices and no negative-weight cycles, Algorithm 7.11 (*AllPairsShortestPaths*) computes the shortest-path distances between each pair of vertices of $\vec{G}$ in $O(n^3)$ time.*

7.2.2 Computing Shortest Paths via Matrix Multiplication

We can view the problem of computing the shortest-path distances for all pairs of vertices in a directed graph $\vec{G}$ as a matrix problem. In this subsection, we describe how to solve the all-pairs shortest-path problem in $O(n^3)$ time using this approach. We first describe how to use this approach to solve the all-pairs problem in $O(n^4)$ time, and then we show how this can be improved to $O(n^3)$ time by studying the problem in more depth. This matrix-multiplication approach to shortest paths is especially useful in contexts where we represent graphs using the adjacency matrix data structure.

The Weighted Adjacency Matrix Representation

Let us number the vertices of $\vec{G}$ as $(v_0, v_1, \ldots, v_{n-1})$, returning to the convention of numbering the vertices starting at index 0. Given this numbering of the vertices of $\vec{G}$, there is a natural weighted view of the adjacency matrix representation for a graph, where we define $A[i, j]$ as follows:

$$A[i,j] = \begin{cases} 0 & \text{if } i = j \\ w((v_i, v_j)) & \text{if } (v_i, v_j) \text{ is an edge in } \vec{G} \\ +\infty & \text{otherwise.} \end{cases}$$

(Note that this is the same definition used for the cost function $D^0_{i,j}$ from the previous subsection.)

Shortest Paths and Matrix Multiplication

In other words, $A[i, j]$ stores the shortest path distance from v_i to v_j using one or fewer edges in the path. Let us therefore use the matrix A to define another matrix A^2, such that $A^2[i, j]$ stores the shortest path distance from v_i to v_j using at most two edges. A path with at most two edges is either empty (a zero-edge path) or it adds an extra edge to a zero-edge or one-edge path. Therefore, we can define $A^2[i, j]$ as

$$A^2[i,j] = \min_{l=0,1,\ldots,n-1} \{A[i,l] + A[l,j]\}.$$

Thus, given A, we can compute the matrix A^2 in $O(n^3)$ time, by using an algorithm very similar to the standard matrix multiplication algorithm.

In fact, we can view this computation as a matrix multiplication in which we have simply redefined what the operators "plus" and "times" mean in the matrix multiplication algorithm (the programming language C++ specifically allows for such operator overloading). If we let "plus" be redefined to mean "min" and we let "times" be redefined to mean "$+$," then we can write $A^2[i, j]$ as a true matrix multiplication:

$$A^2[i,j] = \sum_{l=0,1,\ldots,n-1} A[i,l] \cdot A[l,j].$$

Indeed, this matrix-multiplication viewpoint is the reason why we have written this matrix as "A^2," for it is the square of the matrix A.

Let us continue this approach to define a matrix A^k, so that $A^k[i, j]$ is the shortest-path distance from v_i to v_j using at most k edges. Since a path with at most k edges is equivalent to a path with at most $k-1$ edges plus possibly one additional edge, we can define A^k so that

$$A^k[i,j] = \sum_{l=0,1,\ldots,n-1} A^{k-1}[i,l] \cdot A[l,j],$$

continuing the operator redefining so that "$+$" stands for "min" and "$\cdot$" stands for "$+$."

The crucial observation is that if $\vec{G}$ contains no negative-weight cycles, then A^{n-1} stores the shortest-path distance between each pair of vertices in $\vec{G}$. This observation follows from the fact that any well-defined shortest path contains at most $n-1$ edges. If a path has more than $n-1$ edges, it must repeat some vertex; hence, it must contain a cycle. But a shortest path will never contain a cycle (unless there is a negative-weight cycle in $\vec{G}$). Thus, to solve the all-pairs shortest path problem, all we need to do is to multiply A times itself $n-1$ times. Since each such multiplication can be done in $O(n^3)$ time, this approach immediately gives us the following.

Theorem 7.8: *Given a weighted directed n-vertex graph $\vec{G}$ containing no negative-weight cycles, and the weighted adjacency matrix A for $\vec{G}$, the all-pairs shortest path problem for $\vec{G}$ can be solved by computing A^{n-1}, which can be performed in $O(n^4)$ time.*

In Section 10.1.4, we discuss an exponentiation algorithm for numbers, which can be applied in the present context of matrix multiplication to compute A^{n-1} in $O(n^3 \log n)$ time. We can actually compute A^{n-1} in $O(n^3)$ time, however, by taking advantage of additional structure present in the all-pairs shortest-path problem.

Matrix Closure

As observed above, if $\vec{G}$ contains no negative-weight cycles, then A^{n-1} encodes all the shortest-path distances between pairs of vertices in $\vec{G}$. A well-defined shortest path can contain no cycles; hence, a shortest path restricted to contain at most $n-1$ edges must be a true shortest path. Likewise, a shortest path containing at most n edges is a true shortest path, as is a shortest path containing at most $n+1$ edges, $n+2$ edges, and so on. Therefore, if $\vec{G}$ contains no negative-weight cycles, then

$$A^{n-1} = A^n = A^{n+1} = A^{n+2} = \cdots.$$

The ***closure*** of a matrix A is defined as

$$A^* = \sum_{l=0}^{\infty} A^l,$$

if such a matrix exists. If A is a weighted adjacency matrix, then $A^*[i,j]$ is the sum of all possible paths from v_i to v_j. In our case, A is the weighted adjacency matrix for a directed graph $\vec{G}$ and we have redefined "+" as "min." Thus, we can write

$$A^* = \min_{i=0,\ldots,\infty} \{A^i\}.$$

Moreover, since we are computing shortest path distances, the entries in A^{i+1} are never larger than the entries in A^i. Therefore, for the weighted adjacency matrix of an n-vertex digraph $\vec{G}$ with no negative-weight cycles,

$$A^* = A^{n-1} = A^n = A^{n+1} = A^{n+2} = \cdots.$$

That is, $A^*[i,j]$ stores the length of the shortest path from v_i to v_j.

Computing the Closure of a Weighted Adjacency Matrix

We can compute the closure A^* by divide-and-conquer in $O(n^3)$ time. Without loss of generality, we may assume that n is a power of two (if not, then pad the digraph $\vec{G}$ with extra vertices that have no in-going or out-going edges). Let us divide the set V of vertices in $\vec{G}$ into two equal-sized sets $V_1 = \{v_0, \ldots, v_{n/2-1}\}$ and $V_2 = \{v_{n/2}, \ldots, v_{n-1}\}$. Given this division, we can likewise divide the adjacency matrix A into four blocks, B, C, D, and E, each with $n/2$ rows and columns, defined as follows:

- B: weights of edges from V_1 to V_1
- C: weights of edges from V_1 to V_2
- D: weights of edges from V_2 to V_1
- E: weights of edges from V_2 to V_2.

That is,

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}.$$

We illustrate these four sets of edges in Figure 7.12.

Likewise, we can partition A^* into four blocks W, X, Y, and Z, as well, which are similarly defined.

- W: weights of shortest paths from V_1 to V_1
- X: weights of shortest paths from V_1 to V_2
- Y: weights of shortest paths from V_2 to V_1
- Z: weights of shortest paths from V_2 to V_2,

That is,

$$A^* = \begin{pmatrix} W & X \\ Y & Z \end{pmatrix}.$$

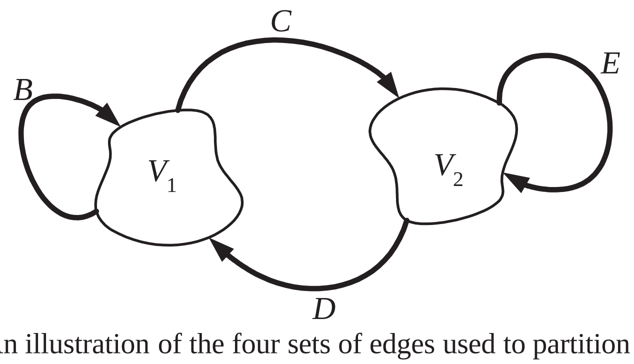


Figure 7.12: An illustration of the four sets of edges used to partition the adjacency matrix A in the divide-and-conquer algorithm for computing A^*.

Submatrix Equations

By these definitions and those above, we can derive simple equations to define W, X, Y, and Z directly from the submatrices B, C, D, and E.

- $W = (B + C \cdot E^* \cdot D)^*$, for paths in W consist of the closure of subpaths that either stay in V_1 or jump to V_2, travel in V_2 for a while, and then jump back to V_1.
- $X = W \cdot C \cdot E^*$, for paths in X consist of the closure of subpaths that start and end in V_1 (with possible jumps to V_2 and back), followed by a jump to V_2 and the closure of subpaths that stay in V_2.
- $Y = E^* \cdot D \cdot W$, for paths in Y consist of the closure of subpaths that stay in V_2, followed by a jump to V_1 and the closure of subpaths that start and end in V_1 (with possible jumps to V_2 and back).
- $Z = E^* + E^* \cdot D \cdot W \cdot C \cdot E^*$, for paths in Z consist of paths that stay in V_2 or paths that travel in V_2, jump to V_1, travel in V_1 for a while (with possible jumps to V_2 and back), jump back to V_2, and then stay in V_2.

Given these equations it is a simple matter to then construct a recursive algorithm to compute A^*. In this algorithm, we divide A into the blocks B, C, D, and E, as described above. We then recursively compute the closure E^*. Given E^*, we can then recursively compute the closure $(B + C \cdot E^* \cdot D)^*$, which is W.

Note that no other recursive closure computations are then needed to compute X, Y, and Z. Thus, after a constant number of matrix additions and multiplications, we can compute all the blocks in A^*. This gives us the following theorem.

Theorem 7.9: *Given a weighted directed n-vertex graph $\vec{G}$ containing no negative-weight cycles, and the weighted adjacency matrix A for $\vec{G}$, the all-pairs shortest path problem for $\vec{G}$ can be solved by computing A^*, which can be performed in $O(n^3)$ time.*

Proof: We have already argued why the computation of A^* solves the all-pairs shortest-path problem. Consider, then, the running time of the divide-and-conquer algorithm for computing A^*, the closure of the $n \times n$ adjacency matrix A. This algorithm consists of two recursive calls to compute the closure of $(n/2) \times (n/2)$ submatrices, plus a constant number of matrix additions and multiplications (using "min" for "+" and "+" for "·"). Thus, assuming we use the straightforward $O(n^3)$-time matrix multiplication algorithm, we can characterize the running time, $T(n)$, for computing A^* as

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + cn^3 & \text{if } n > 1, \end{cases}$$

where $b > 0$ and $c > 0$ are constants. Therefore, by the Master Theorem (5.6), we can compute A^* in $O(n^3)$ time. ■

7.3 Minimum Spanning Trees

Suppose we wish to connect all the computers in a new office building using the least amount of cable. Likewise, suppose we have an undirected computer network in which each connection between two routers has a cost for usage; we want to connect all our routers at the minimum cost possible. 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. Methods for finding such trees are the focus of this section.

Problem Definition

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,

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

We recall from Section 6.1 that a tree such as this, which contains every vertex of a connected graph G, is said to be a ***spanning tree***. Computing a spanning tree T with smallest total weight is the problem of constructing a ***minimum spanning tree*** (or ***MST***).

The development of efficient algorithms for the minimum-spanning-tree problem predates the modern notion of computer science itself. In this section, we discuss two algorithms for solving the MST problem. These algorithms are all classic applications of the ***greedy method***. As was discussed in Section 5.1, we apply the greedy method by iteratively choosing objects to join a growing collection, by incrementally picking an object that minimizes some cost function.

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 section by discussing a third algorithm, due to Barůvka, which applies the greedy approach in a parallel way.

As in Section 7.1.1, in order to simplify the description 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).

A Crucial Fact about Minimum Spanning Trees

Before we discuss the details of these algorithms, however, let us give a crucial fact about minimum spanning trees that forms the basis of the algorithms. In particular, all the MST algorithms we discuss are based on the greedy method, which in this case depends crucially on the following fact. (See Figure 7.13.)

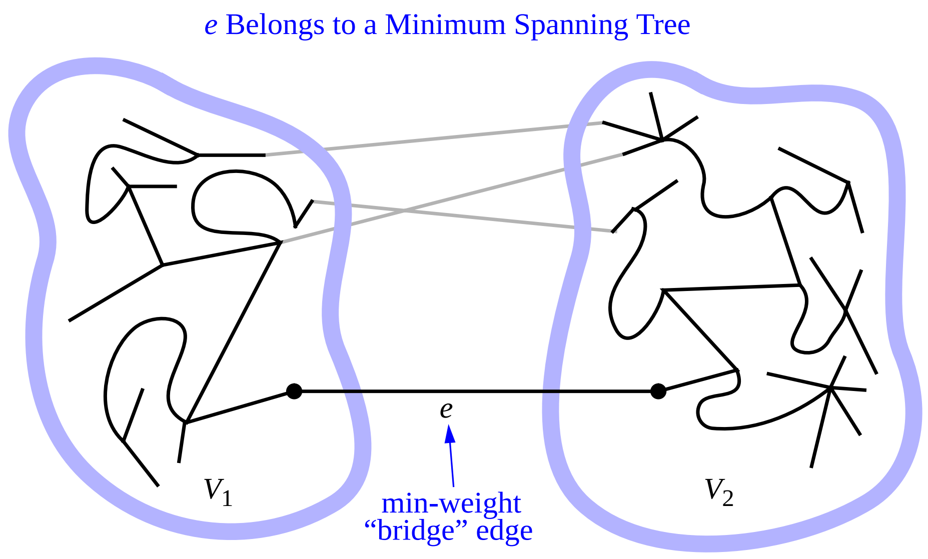


Figure 7.13: An illustration of the crucial fact about minimum spanning trees.

Theorem 7.10: *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: 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-7.5).

In addition, note that Theorem 7.10 remains valid even if the graph G contains negative-weight edges or negative-weight cycles, unlike the algorithms we presented for shortest paths.

7.3.1 Kruskal's Algorithm

The reason Theorem 7.10 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 pseudo-code for Kruskal's method for solving the MST problem in Algorithm 7.14, and we show the working of this algorithm in Figures 7.15, 7.16, and 7.17.

```
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 $C(v) \leftarrow \{v\}$.
   Initialize a priority queue $Q$ to contain all edges in $G$, using the 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 $v$, and 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 7.14: 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, Theorem 7.10. 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.

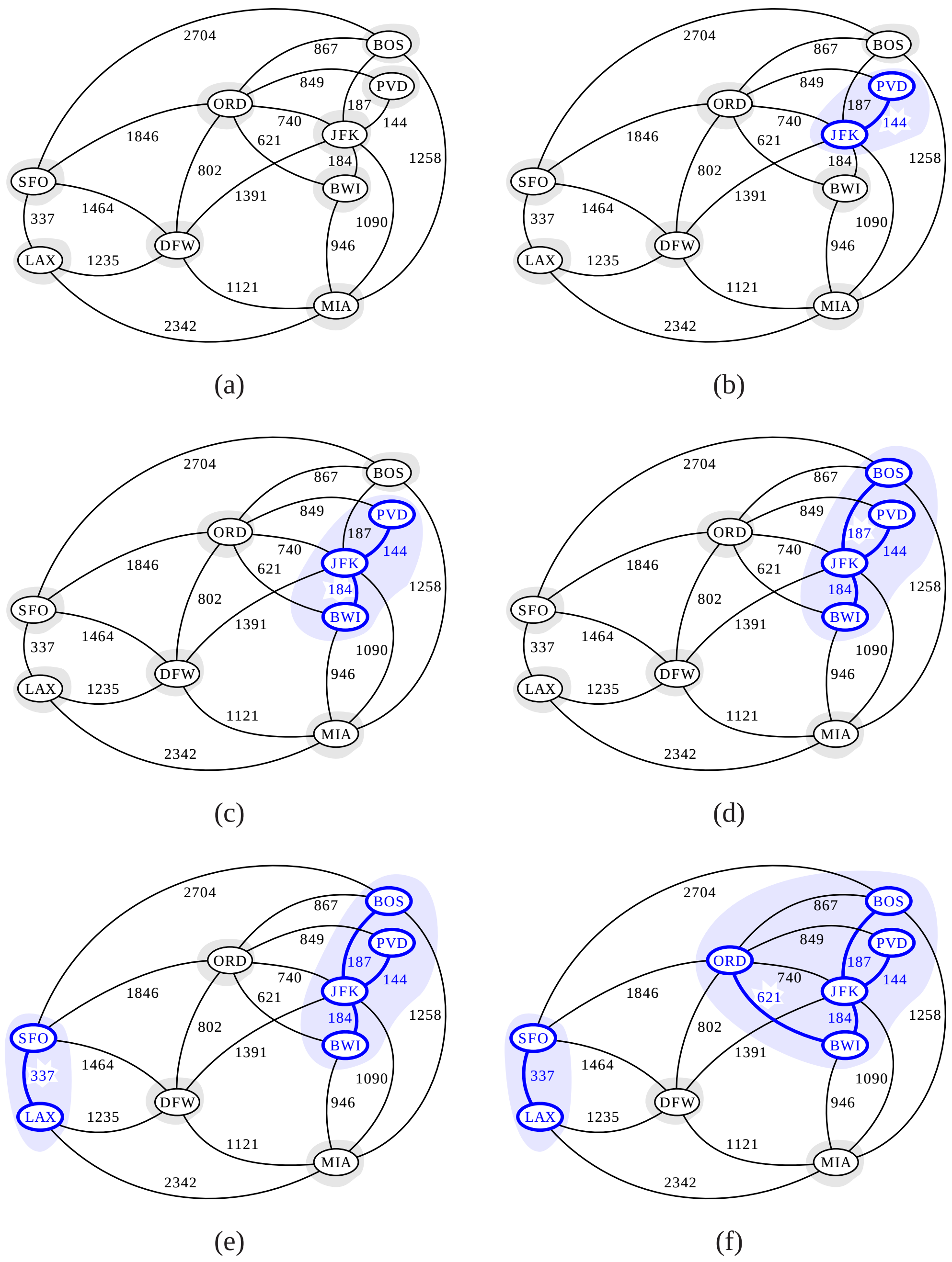


Figure 7.15: Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions and we highlight the edge being considered in each iteration (continued in Figure 7.16).

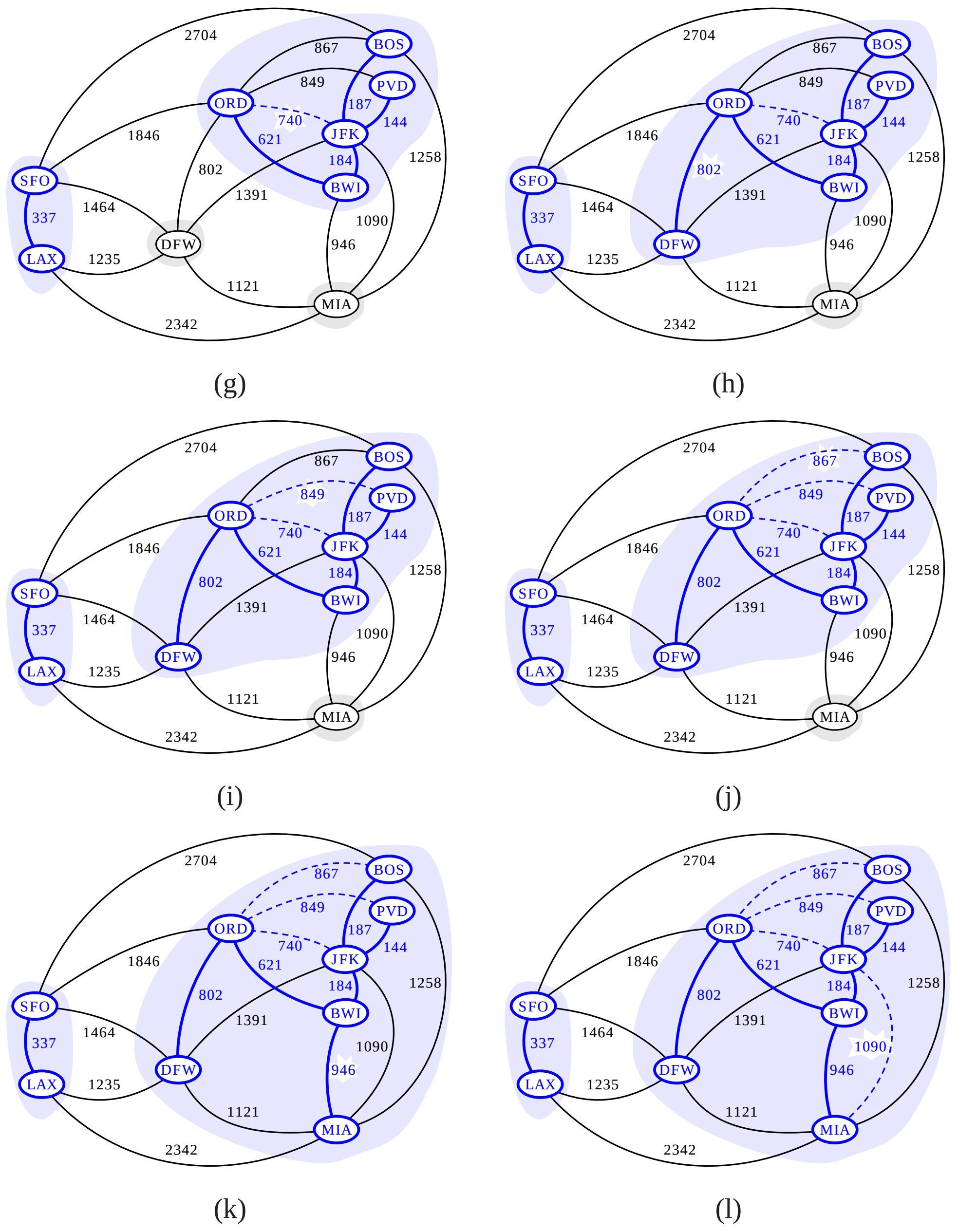


Figure 7.16: An example of an execution of Kruskal's MST algorithm (continued). Rejected edges are shown dashed. (continued in Figure 7.17).

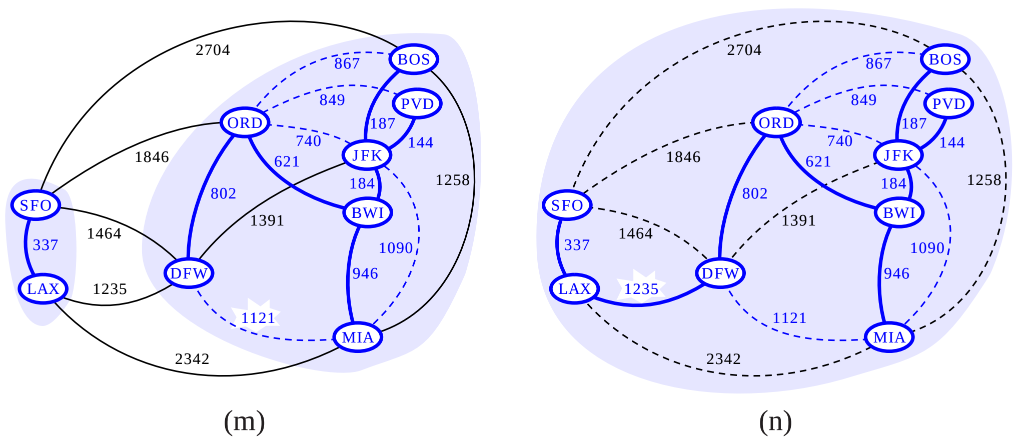


Figure 7.17: Example of an execution of Kruskal's MST algorithm (continued from Figures 7.15 and 7.16). The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm.

Implementing Kruskal's Algorithm

We denote the number of vertices and edges of the input graph G with n and m, respectively. 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 7.14, 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 2.4.4). 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.

A Simple Cluster Merging Strategy

We use a list-based implementation of a partition (Section 4.2.2) for the clusters. Namely, we 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 4.2.2), but this simple approach will be sufficient.

Lemma 7.11: *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 7.11 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 7.12: *Given a simple connected weighted graph G with n vertices and m edges, Kruskal's algorithm constructs a minimum spanning tree for G in time $O(m \log n)$.*

7.3.2 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 comes to play, 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.

Growing a Single 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 pseudo-code in Algorithm 7.18.

```
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 7.18: The Prim-Jarník algorithm for the MST problem.

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 2.4.4), 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 7.13: *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 7.19 and 7.20.

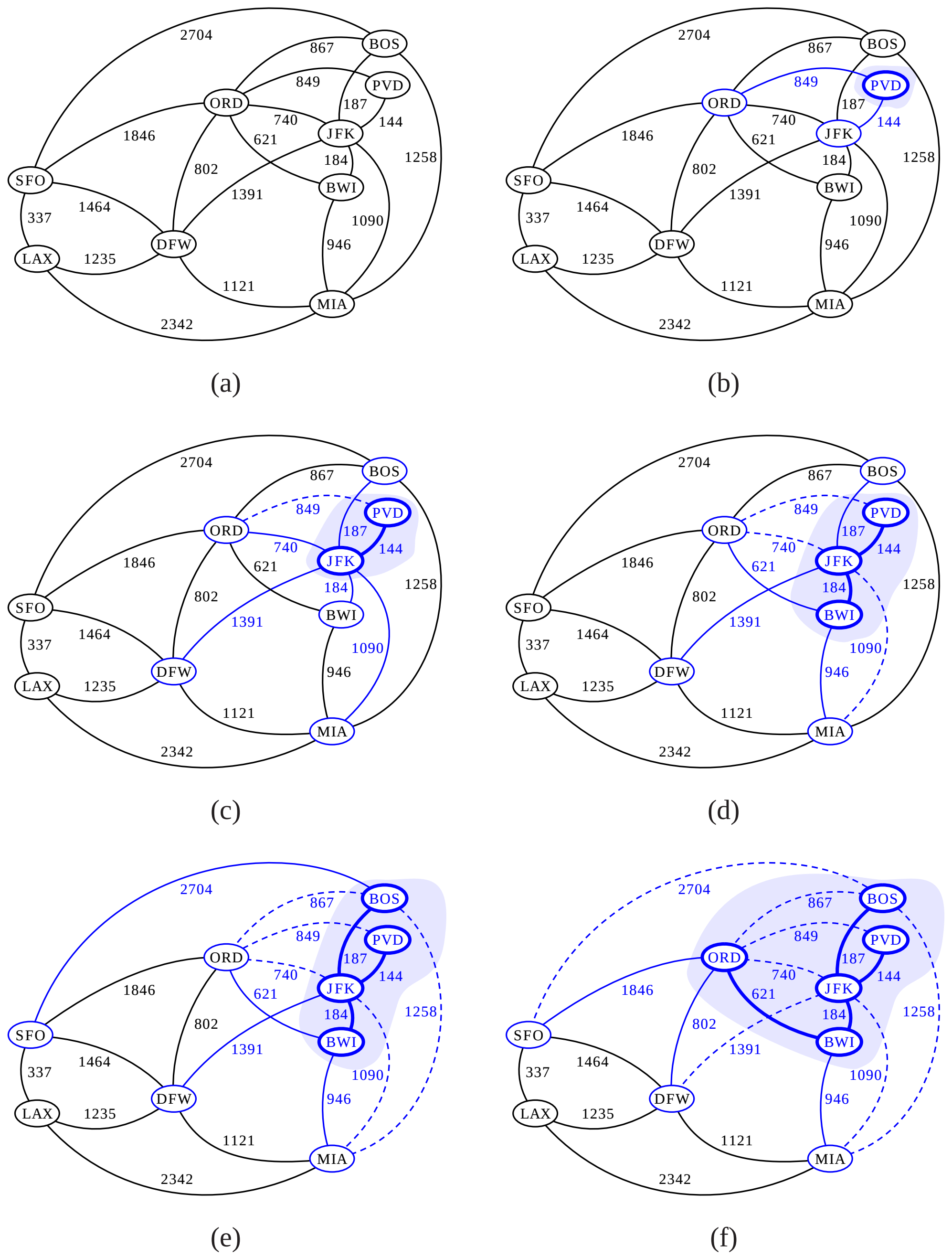


Figure 7.19: Visualizing the Prim-Jarník algorithm (continued in Figure 7.20).

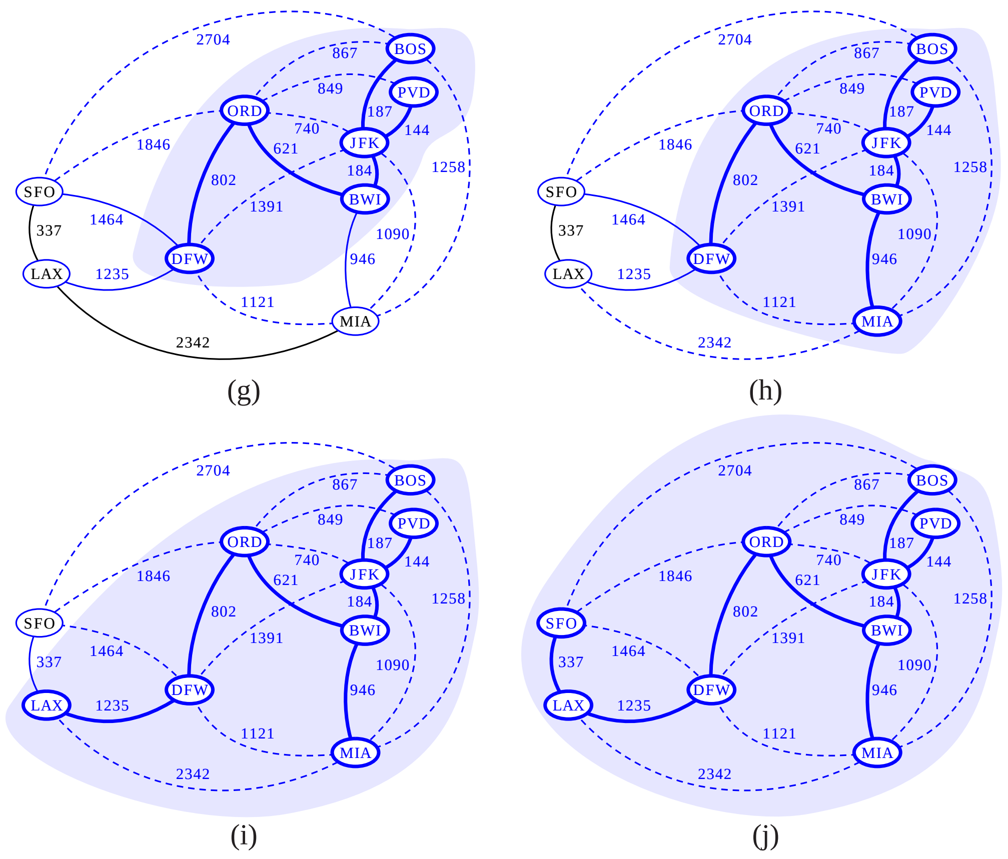


Figure 7.20: Visualizing the Prim-Jarník algorithm (continued from Figure 7.19).

7.3.3 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 pseudo-code description of Barůvka's minimum-spanning-tree algorithm in Algorithm 7.21, and we illustrate an execution of this algorithm in Figure 7.22.

```
Algorithm BarůvkaMST($G$):
    Input: A weighted connected graph $G = (V, E)$ with $n$ vertices and $m$ edges
    Output: A 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 $n-1$ edges {$T$ is not yet an MST} do
        for each connected component $C_i$ of $T$ do
            {Perform the MST edge addition procedure for cluster $C_i$}
            Find the smallest-weight edge $e = (v, u)$, 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 7.21: Pseudo-code for Barůvka's algorithm.

Implementing Barůvka's Algorithm

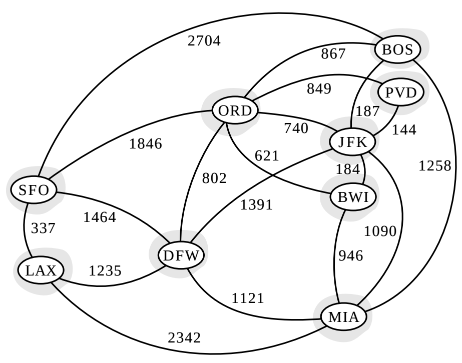
Implementing Barůvka's algorithm is quite simple, requiring only that we be able to do the following:

- 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.

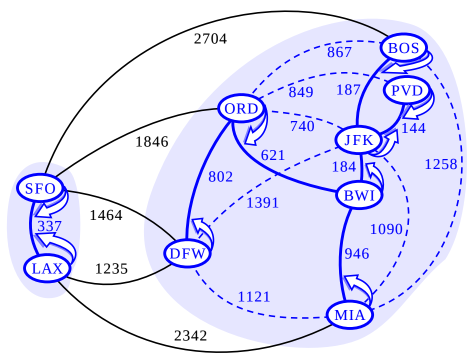
Like Kruskal's algorithm, Barůvka's algorithm builds the minimum spanning tree by growing a number of clusters of vertices in a series of rounds, not just one cluster, as was done by the Prim-Jarník algorithm. But in Barůvka's algorithm, the clusters are grown by applying the crucial fact about minimum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

Why Is This Algorithm Correct?

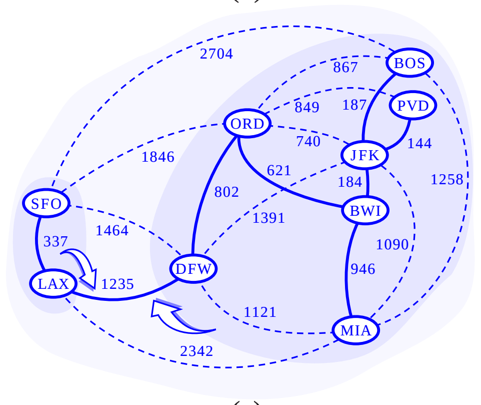
In each iteration of Barůvka's algorithm, we choose the smallest-weight edge coming out of each connected component C_i of the current set T of minimum-spanning-tree edges. In each case, this edge is a valid choice, for if we consider a partitioning of V into the vertices in C_i and those outside of C_i, then the chosen edge e for C_i satisfies the condition of the crucial fact about minimum spanning trees (Theorem 7.10) for guaranteeing that e belongs to a minimum spanning tree.



(a)



(b)



(c)

Figure 7.22: 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.

Analyzing Barůvka's Algorithm

Let us analyze the running time of Barůvka's algorithm (Algorithm 7.21). 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 $n \le m$). 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 7.14: *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.*

7.3.4 A Comparison of MST Algorithms

Although each of the above MST algorithms has the same worst-case running time, each one achieves this running time using different data structures and different approaches to building the minimum spanning tree.

Concerning auxiliary data structures, Kruskal's algorithm uses a priority queue, to store edges, and a collection of sets, implemented with lists, to store clusters. The Prim-Jarník algorithm uses only a priority queue, to store vertex-edge pairs. Thus, from an ease of programming viewpoint, the Prim-Jarník algorithm is preferable. Indeed, the Prim-Jarník algorithm is so similar to Dijkstra's algorithm that an implementation of Dijkstra's algorithm could be converted into an implementation for the Prim-Jarník algorithm without much effort. Barůvka's algorithm requires a way of representing connected components. Thus, from an ease of programming viewpoint, the Prim-Jarník and Barůvka algorithms seem to be the best.

In terms of the constant factors, the three algorithms are fairly similar in that they both have relatively small constant factors in their asymptotic running times. The asymptotic running time for Kruskal's algorithm can be improved if the edges are given in sorted order by their weights (using the partition data structure of Section 4.2.2). Also, the running time of Barůvka's algorithm can be changed to be $O(n^2)$ in the worst case with a slight modification to the algorithm (which we explore in Exercise C-7.12). Thus, there is no clear winner among these three algorithms, although Barůvka's algorithm is the easiest of the three to implement.

7.4 Java Example: Dijkstra's Algorithm

In this section, we present Java code for performing Dijkstra's algorithm (Algorithm 7.2), assuming we are given an undirected graph with positive integer weights.

We express our implementation of Dijkstra's algorithm by means of an abstract class Dijkstra (Code Fragments 7.23–7.25), which declares the abstract method weight(e) to extract the weight of edge e. Class Dijkstra is meant to be extended by subclasses that implement method weight(e). See, for example, class MyDijkstra shown in Code Fragment 7.26.

```
/** Dijkstra's algorithm for the single-source shortest path problem
  * in an undirected graph whose edges have integer weights. Classes
  * extending ths abstract class must define the weight(e) method,
  * which extracts the weight of an edge. */
public abstract class Dijkstra {
  /** Execute Dijkstra's algorithm. */
  public void execute(InspectableGraph g, Vertex source) {
    graph = g;
    dijkstraVisit(source);
  }
  /** Attribute for vertex distances. */
  protected Object DIST = new Object();
  /** Set the distance of a vertex. */
  protected void setDist(Vertex v, int d) {
    v.set(DIST, new Integer(d));
  }
  /** Get the distance of a vertex from the source vertex. This method
    * returns the length of a shortest path from the source to u after
    * method execute has been called. */
  public int getDist(Vertex u) {
    return ((Integer) u.get(DIST)).intValue();
  }
  /** This abstract method must be defined by subclasses.
    * @return weight of edge e. */
  protected abstract int weight(Edge e);
  /** Infinity value. */
  public static final int INFINITE = Integer.MAX_VALUE;
  /** Input graph. */
  protected InspectableGraph graph;
  /** Auxiliary priority queue. */
  protected PriorityQueue Q;
```

Code Fragment 7.23: Class Dijkstra implementing Dijkstra's algorithm (continued in Code Fragments 7.24 and 7.25).

The algorithm is executed by method dijkstraVisit. A priority queue Q supporting locator-based methods (Section 2.4.4) is used. We insert a vertex u into Q with method insert, which returns the locator of u in Q. Following the decorator pattern, we "attach" to u its locator by means of method setLoc, and we retrieve the locator of u with method getLoc. Changing the label of a vertex z to d in the relaxation procedure is done with method replaceKey(ℓ, d), where ℓ is the locator of z.

```
/** The actual execution of Dijkstra's algorithm.
  * @param v source vertex. */
protected void dijkstraVisit (Vertex v) {
  // initialize the priority queue Q and store all the vertices in it
  Q = new ArrayHeap(new IntegerComparator());
  for (VertexIterator vertices = graph.vertices(); vertices.hasNext();) {
    Vertex u = vertices.nextVertex();
    int u_dist;
    if (u==v)
      u_dist = 0;
    else
      u_dist = INFINITE;
    // setDist(u, u_dist);
    Locator u_loc = Q.insert(new Integer(u_dist), u);
    setLoc(u, u_loc);
  }
  // grow the cloud, one vertex at a time
  while (!Q.isEmpty()) {
    // remove from Q and insert into cloud a vertex with minimum distance
    Locator u_loc = Q.min();
    Vertex u = getVertex(u_loc);
    int u_dist = getDist(u_loc);
    Q.remove(u_loc); // remove u from the priority queue
    setDist(u, u_dist); // the distance of u is final
    destroyLoc(u); // remove the locator associated with u
    if (u_dist == INFINITE)
      continue; // unreachable vertices are not processed
    // examine all the neighbors of u and update their distances
    for (EdgeIterator edges = graph.incidentEdges(u); edges.hasNext();) {
      Edge e = edges.nextEdge();
      Vertex z = graph.opposite(u,e);
      if (hasLoc(z)) { // check that z is in Q, i.e., it is not in the cloud
        int e_weight = weight(e);
        Locator z_loc = getLoc(z);
        int z_dist = getDist(z_loc);
        if ( u_dist + e_weight < z_dist ) // relaxation of edge e = (u,z)
          Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
      }
    }
  }
}
```

Code Fragment 7.24: Method dijkstraVisit of class Dijkstra.

```
  /** Attribute for vertex locators in the priority queue Q. */
  protected Object LOC = new Object();
  /** Check if there is a locator associated with a vertex. */
  protected boolean hasLoc(Vertex v) {
    return v.has(LOC);
  }
  /** Get the locator in Q of a vertex. */
  protected Locator getLoc(Vertex v) {
    return (Locator) v.get(LOC);
  }
  /** Associate with a vertex its locator in Q. */
  protected void setLoc(Vertex v, Locator l) {
    v.set(LOC, l);
  }
  /** Remove the locator associated with a vertex. */
  protected void destroyLoc(Vertex v) {
    v.destroy(LOC);
  }
  /** Get the vertex associated with a locator. */
  protected Vertex getVertex(Locator l) {
    return (Vertex) l.element();
  }
  /** Get the distance of a vertex given its locator in Q. */
  protected int getDist(Locator l) {
    return ((Integer) l.key()).intValue();
  }
```

Code Fragment 7.25: Auxiliary methods of class Dijkstra. They assume that the vertices of the graph are decorable (continued from Algorithms 7.23 and 7.24).

```
/** A specialization of class Dijkstra that extracts edge weights from
  * decorations. */
public class MyDijkstra extends Dijkstra {
  /** Attribute for edge weights. */
  protected Object WEIGHT;
  /** Constructor that sets the weight attribute. */
  public MyDijkstra(Object weight_attribute) {
    WEIGHT = weight_attribute;
  }
  /** The edge weight is stored in attribute WEIGHT of the edge. */
  public int weight(Edge e) {
    return ((Integer) e.get(WEIGHT)).intValue();
  }
}
```

Code Fragment 7.26: Class MyDijkstra that extends Dijkstra and provides a concrete implementation of method weight(e).

7.5 Exercises

Reinforcement

R-7.1 Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.

R-7.2 Show how to modify Dijkstra's algorithm for the case when the graph is directed and we want to compute shortest ***directed paths*** from the source vertex to all the other vertices.

R-7.3 Show how to modify Dijkstra's algorithm to not only output the distance from v to each vertex in G, but also to output a tree T rooted at v, such that the path in T from v to a vertex u is actually a shortest path in G from v to u.

R-7.4 Draw a (simple) directed weighted graph G with 10 vertices and 18 edges, such that G contains a minimum-weight cycle with at least 4 edges. Show that the Bellman-Ford algorithm will find this cycle.

R-7.5 The dynamic programming algorithm of Algorithm 7.11 uses $O(n^3)$ space. Describe a version of this algorithm that uses $O(n^2)$ space.

R-7.6 The dynamic programming algorithm of Algorithm 7.11 computes only shortest-path distances, not actual paths. Describe a version of this algorithm that outputs the set of all shortest paths between each pair of vertices in a directed graph. Your algorithm should still run in $O(n^3)$ time.

R-7.7 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-7.8 Repeat the previous problem for the Prim-Jarník algorithm.

R-7.9 Repeat the previous problem for Barůvka's algorithm.

R-7.10 Consider the unsorted sequence implementation of the priority queue Q used in Dijkstra's algorithm. In this case, what is the best-case running time of Dijkstra's algorithm $\Omega(n^2)$ on an n-vertex graph?

Hint: Consider the size of Q each time the minimum element is extracted.

R-7.11 Describe the meaning of the graphical conventions used in Figures 7.3 and 7.4 illustrating Dijkstra's algorithm. What do the arrows signify? How about thick lines and dashed lines?

R-7.12 Repeat Exercise R-7.11 for Figures 7.15 and 7.17 illustrating Kruskal's algorithm.

R-7.13 Repeat Exercise R-7.11 for Figures 7.19 and 7.20 illustrating the Prim-Jarník algorithm.

R-7.14 Repeat Exercise R-7.11 for Figure 7.22 illustrating Barůvka's algorithm.

Creativity

C-7.1 Give an example of an n-vertex simple graph G that causes Dijkstra's algorithm to run in $\Omega(n^2 \log n)$ time when its implemented with a heap for the priority queue.

C-7.2 Give an example of a weighted directed graph $\vec{G}$ with negative-weight edges, but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex v.

C-7.3 Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

```
1: Initialize path to start.
2: Initialize VisitedVertices to {start}.
3: If start=goal, return path and exit. Otherwise, continue.
4: Find the edge (start,v) of minimum weight such that v is adjacent to start
   and v is not in VisitedVertices.
5: Add v to path.
6: Add v to VisitedVertices.
7: Set start equal to v and go to step 3.
```

Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counter example.

C-7.4★ Suppose we are given a weighted graph G with n vertices and m edges, such that the weight on each edge is an integer between 0 and n. Show that we can find a minimum spanning tree for G in $O(n \log^* n)$ time.

C-7.5 Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.

C-7.6 Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted digraph $\vec{G}$. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

C-7.7 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. The bandwidth of a path 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.

C-7.8 NASA wants to link n stations spread over the country using communication channels. Each pair of stations has a different bandwidth available, which is known a priori. NASA wants to select $n-1$ channels (the minimum possible) in such a way that all the stations are linked by the channels and the total bandwidth (defined as the sum of the individual bandwidths of the channels) is maximum. Give an efficient algorithm for this problem and determine its worst-case time complexity. Consider the weighted graph $G = (V, E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge $e \in E$ as the bandwidth of the corresponding channel.

C-7.9 Suppose you are given a *timetable*, which consists of:

- A set $\mathcal{A}$ of n airports, and for each airport $a \in \mathcal{A}$, a minimum connecting time $c(a)$
- A set $\mathcal{F}$ of m flights, and the following, for each flight $f \in \mathcal{A}$:
  - Origin airport $a_1(f) \in \mathcal{A}$
  - Destination airport $a_2(f) \in \mathcal{A}$
  - Departure time $t_1(f)$
  - Arrival time $t_2(f)$.

Describe an efficient algorithm for the flight scheduling problem. In this problem, we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed. What is the running time of your algorithm as a function of n and m?

C-7.10 As your reward for saving the Kingdom of Bigfunnia from the evil monster, "Exponential Asymptotic," the king has given you the opportunity to earn a big reward. Behind the castle there is a maze, and along each corridor of the maze there is a bag of gold coins. The amount of gold in each bag varies. You will be given the opportunity to walk through the maze, picking up bags of gold. You may enter only through the door marked "ENTER" and exit through the door marked "EXIT." (These are distinct doors.) While in the maze you may not retrace your steps. Each corridor of the maze has an arrow painted on the wall. You may only go down the corridor in the direction of the arrow. There is no way to traverse a "loop" in the maze. You will receive a map of the maze, including the amount of gold in and the direction of each corridor. Describe an algorithm to help you pick up the most gold.

C-7.11 Suppose we are given a directed graph $\vec{G}$ with n vertices, and let M be the $n \times n$ adjacency matrix corresponding to $\vec{G}$.

a. Let the product of M with itself (M^2) be defined, for $1 \le i, j \le n$, as follows:

$$M^2(i,j) = M(i,1) \odot M(1,j) \oplus \cdots \oplus M(i,n) \odot M(n,j),$$

where "$\oplus$" is the Boolean **or** operator and "$\odot$" is Boolean **and**. Given this definition, what does $M^2(i,j) = 1$ imply about the vertices i and j? What if $M^2(i,j) = 0$?

b. Suppose M^4 is the product of M^2 with itself. What do the entries of M^4 signify? How about the entries of $M^5 = (M^4)(M)$? In general, what information is contained in the matrix M^p?

c. Now suppose that $\vec{G}$ is weighted and assume the following:

1: for $1 \le i \le n$, $M(i,i) = 0$.
2: for $1 \le i, j \le n$, $M(i,j) = \mathit{weight}(i,j)$ if $(i,j) \in E$.
3: for $1 \le i, j \le n$, $M(i,j) = \infty$ if $(i,j) \notin E$.

Also, let M^2 be defined, for $1 \le i, j \le n$, as follows:

$$M^2(i,j) = \min\{M(i,1) + M(1,j), \ldots, M(i,n) + M(n,j)\}.$$

If $M^2(i,j) = k$, what may we conclude about the relationship between vertices i and j?

C-7.12 Show how to modify Barůvka's algorithm so that it runs in worst-case $O(n^2)$ time.

Projects

P-7.1 Implement Kruskal's algorithm assuming that the edge weights are integers.

P-7.2 Implement the Prim-Jarník algorithm assuming that the edge weights are integers.

P-7.3 Implement the Barůvka's algorithm assuming that the edge weights are integers.

P-7.4 Perform an experimental comparison of two of the minimum-spanning-tree algorithms discussed in this chapter (that is, two of Kruskal, Prim-Jarník, or Barůvka). Develop an extensive set of experiments to test the running times of these algorithms using randomly generated graphs.

Chapter Notes

The first known minimum-spanning-tree algorithm is due to Barůvka [22], and was published in 1926. The Prim-Jarník algorithm was first published in Czech by Jarník [108] in 1930 and in English in 1957 by Prim [169]. Kruskal published his minimum-spanning-tree algorithm in 1956 [127]. The reader interested in further study of the history of the minimum spanning tree problem is referred to the paper by Graham and Hell [89]. The current asymptotically fastest minimum-spanning-tree algorithm is a randomized method of Karger, Klein, and Tarjan [112] that runs in $O(m)$ expected time.

Dijkstra [60] published his single-source, shortest path algorithm in 1959. The Bellman-Ford algorithm is derived from separate publications of Bellman [25] and Ford [71].

The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [9], Cormen, Leiserson, and Rivest [55], Even [68], Gibbons [77], Mehlhorn [149], and Tarjan [200], and the book chapter by van Leeuwen [205].

Incidentally, the running time for the Prim-Jarník algorithm, and also that of Dijkstra's algorithm, can actually 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" [72] or the "Relaxed Heap" [61]. 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 the shortest-path and minimum-spanning-tree problems.