13.5.2 Dijkstra's Algorithm

The main idea behind 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 a minimum spanning tree.

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).

In Dijkstra's algorithm for finding shortest paths, 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 is equal to the true distance.

Edge Relaxation

Let us define a label $D[u]$ for each vertex u in V, which we use to approximate the distance in G from v to u. The meaning of these labels is that $D[u]$ always stores 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, because 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
    $D[z] \leftarrow D[u] + w((u,z))$
```

We give the pseudo-code for Dijkstra's algorithm in Code Fragment 13.24. Note that we use a priority queue Q to store the vertices outside of the cloud C.

```
Algorithm ShortestPath($G,v$):
  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 length of a
      shortest path from $v$ to $u$ in $G$
  Initialize $D[v] \leftarrow 0$ and $D[u] \leftarrow +\infty$ for each vertex $u \neq v$.
  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 $u$
```

Code Fragment 13.24: Dijkstra's algorithm for the single-source, shortest-path problem.

We illustrate several iterations of Dijkstra's algorithm in Figures 13.14 and 13.15.

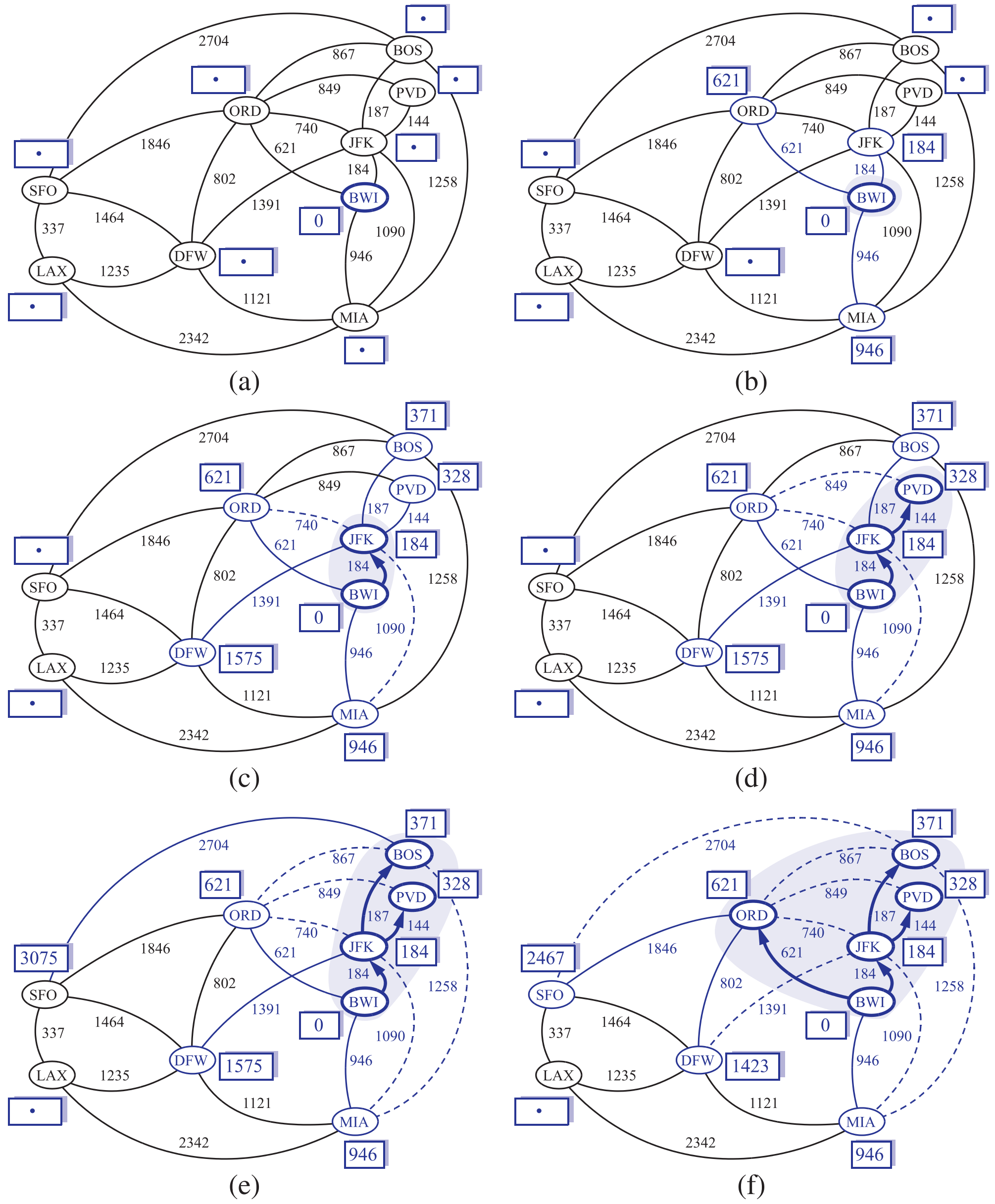

Figure 13.14: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex v stores the label $D[v]$. The symbol $\bullet$ is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick blue arrows and, for each vertex u outside the "cloud," we show the current best edge for pulling in u with a solid blue line. (Continues in Figure 13.15.)

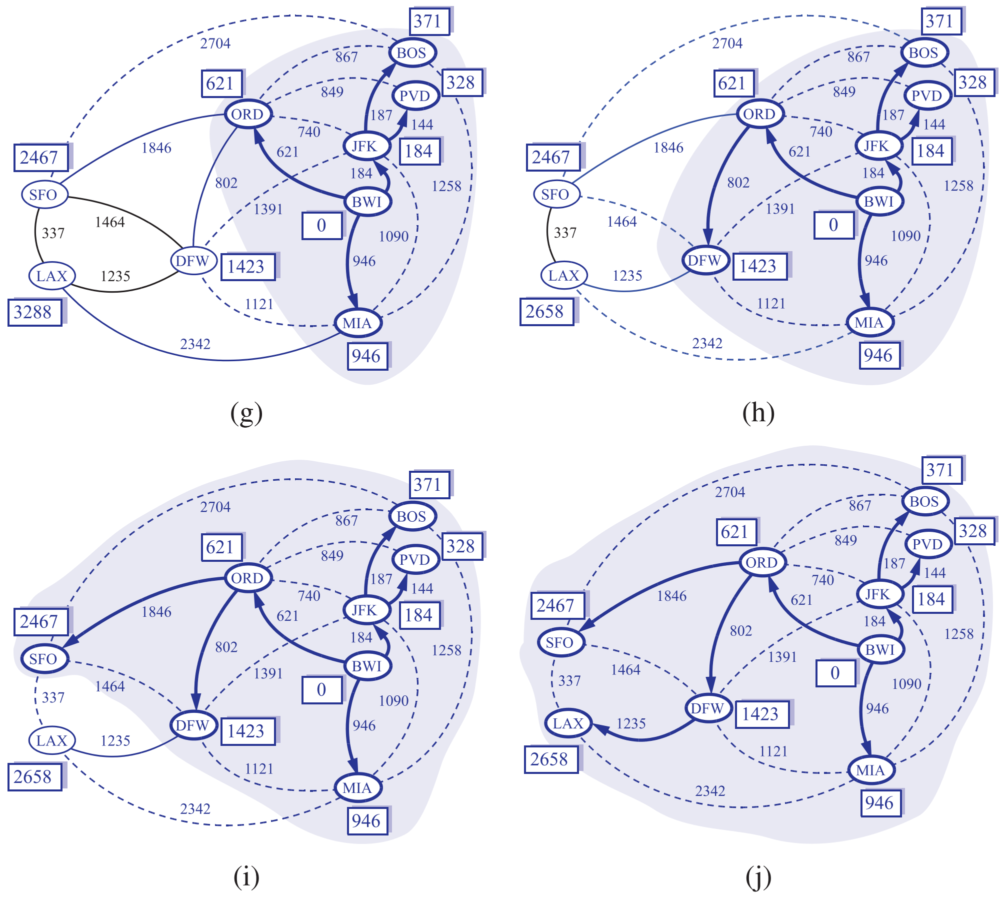

Figure 13.15: An example execution of Dijkstra's algorithm. (Continued from Figure 13.14.)

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, since that allows the greedy method to work correctly, as we show in the proposition that follows.

Proposition 13.23: *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.*

Justification: 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 $d(v,u) = +\infty = D[u]$). Let us therefore 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 13.16.) 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

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

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 pick u, not z, to join C; hence

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

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

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

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

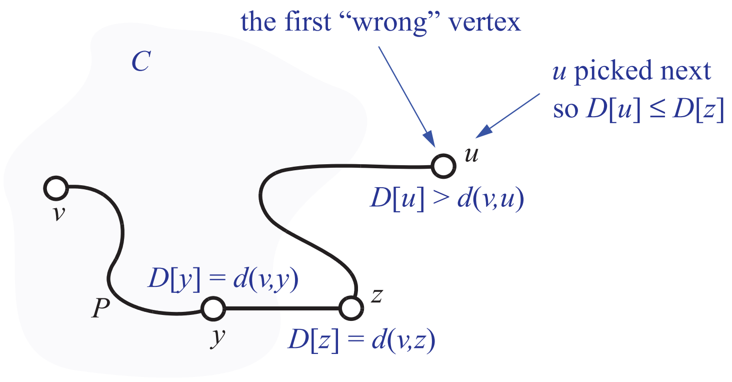

Figure 13.16: A schematic for the justification of Proposition 13.23.

The Running Time of Dijkstra's Algorithm

In this section, we analyze the time complexity of Dijkstra'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 added and compared in constant time. Because of the high level of the description we gave for Dijkstra's algorithm in Code Fragment 13.24, 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, as we must say more about how to implement the other principle data structure in the algorithm—the priority queue Q.

An efficient implementation of the priority queue Q uses a heap (Section 8.3). This allows us to extract the vertex u with smallest D label (call to the removeMin function) 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. Thus, we actually need a heap implementation of an adaptable priority queue (Section 8.4). If Q is an adaptable priority queue implemented as a heap, then this key update can, for example, be done using the $\mathsf{replace}(e,k)$, where e is the entry storing the key for the vertex z. If e is location aware, then we can easily implement such key updates in $O(\log n)$ time, since a location-aware entry for vertex z would allow Q to have immediate access to the entry e storing z in the heap (see Section 8.4.2). Assuming this implementation of Q, Dijkstra's algorithm runs in $O((n+m)\log n)$ time.

Referring back to Code Fragment 13.24, 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 8.3.6).
- At each iteration of the **while** loop, we spend $O(\log n)$ time to remove vertex u from Q, and $O(\mathsf{degree}(v)\log n)$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the **while** loop is

$$\sum_{v \text{ in } G} (1+\mathsf{degree}(v))\log n,$$

which is $O((n+m)\log n)$ by Proposition 13.6.

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