14.6.2 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 the source vertex s. In particular, we can use the greedy method to develop an algorithm that iteratively grows a "cloud" of vertices out of s, with the vertices entering the cloud in order of their distances from s. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to s. The algorithm terminates when no more vertices are outside the cloud (or when those outside the cloud are not connected to those within the cloud), at which point we have a shortest path from s to every vertex of G that is reachable from s. This approach is a simple, but nevertheless powerful, example of the greedy method design pattern. Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra's algorithm***.

Edge Relaxation

Let us define a label $D[v]$ for each vertex v in V, which we use to approximate the distance in G from s to v. The meaning of these labels is that $D[v]$ will always store the length of the best path we have found ***so far*** from s to v. Initially, $D[s] = 0$ and $D[v] = \infty$ for each $v \neq s$, and we define the set C, which is our "***cloud***" of vertices, to initially be the empty set. 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 general, we will use a priority queue to select among the vertices outside the cloud.) In the very first iteration we will, of course, pull s into C. Once a new vertex u is pulled into C, we then update the label $D[v]$ of each vertex v 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 v 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. The specific edge relaxation operation is as follows:

Edge Relaxation:

if $D[u] + w(u,v) < D[v]$ **then**
$D[v] = D[u] + w(u,v)$

Algorithm Description and Example

We give the pseudo-code for Dijkstra's algorithm in Code Fragment 14.12, and illustrate several iterations of Dijkstra's algorithm in Figures 14.15 through 14.17.

Algorithm ShortestPath(G,s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

```
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
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 = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q do
        {perform the relaxation procedure on edge (u,v)}
        if D[u] + w(u,v) < D[v] then
            D[v] = D[u] + w(u,v)
            Change to D[v] the key of vertex v in Q.
return the label D[v] of each vertex v
```

Code Fragment 14.12: Pseudo-code for Dijkstra's algorithm, solving the single-source shortest-path problem.

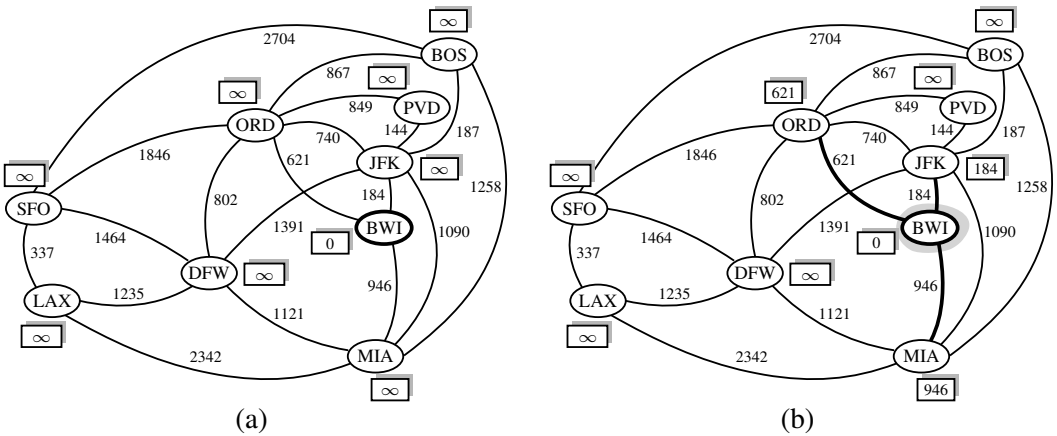

Figure 14.15: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex v stores the label $D[v]$. The edges of the shortest-path tree are drawn as thick arrows, and for each vertex u outside the "cloud" we show the current best edge for pulling in u with a thick line. (Continues in Figure 14.16.)

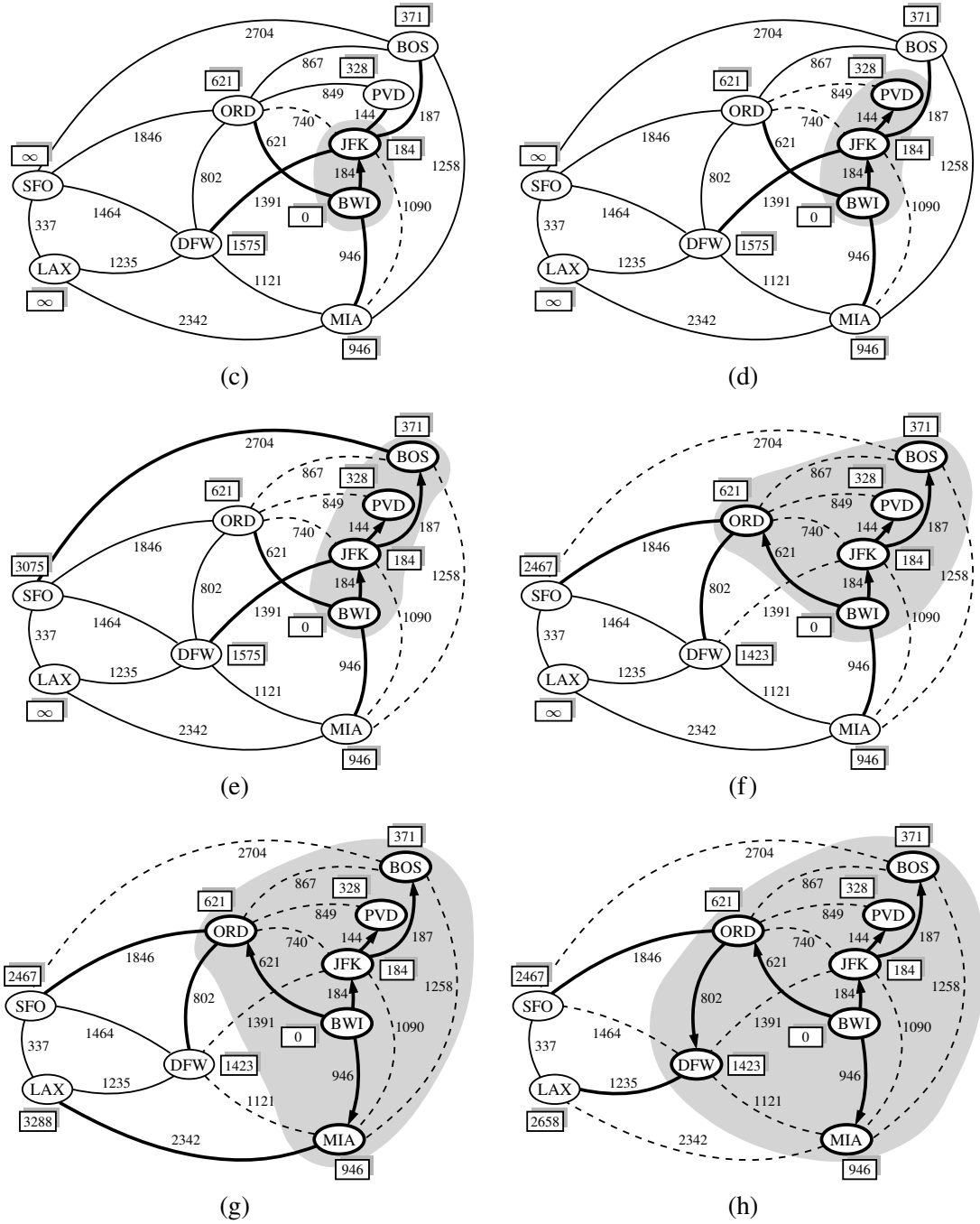

Figure 14.16: An example execution of Dijkstra's algorithm. (Continued from Figure 14.15; continued in Figure 14.17.)

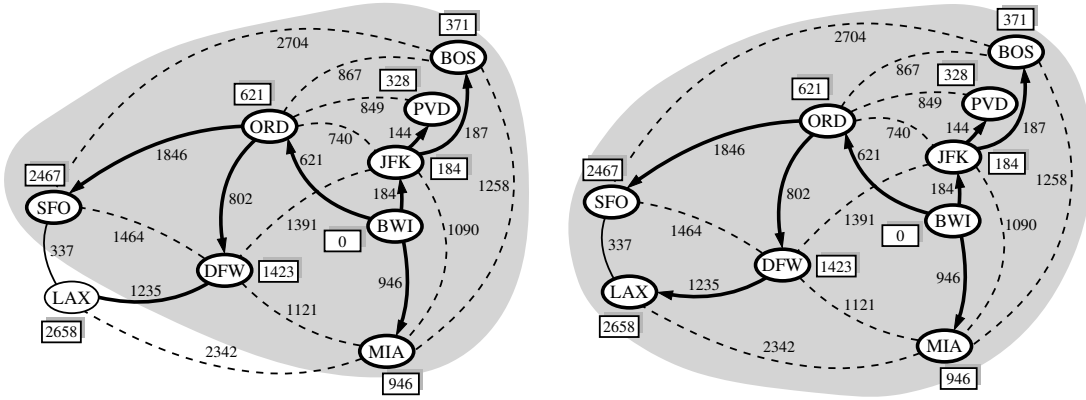

Figure 14.17: An example execution of Dijkstra's algorithm. (Continued from Figure 14.16.)

Why It Works

The interesting 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 s 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 s to each other vertex u in the graph. Why is it that the distance from s to u is equal to the value of the label $D[u]$ at the time vertex u is removed from the priority queue Q and added to the cloud C? 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 proposition that follows.

Proposition 14.23: *In Dijkstra's algorithm, whenever a vertex v is pulled into the cloud, the label $D[v]$ is equal to $d(s,v)$, the length of a shortest path from s to v.*

Justification: Suppose that $D[v] > d(s,v)$ for some vertex v in V, and let z be the ***first*** vertex the algorithm pulled into the cloud C (that is, removed from Q) such that $D[z] > d(s,z)$. There is a shortest path P from s to z (for otherwise $d(s,z) = \infty = D[z]$). Let us therefore consider the moment when z is pulled into C, and let y be the first vertex of P (when going from s to z) that is not in C at this moment. Let x be the predecessor of y in path P (note that we could have $x = s$). (See Figure 14.18.) We know, by our choice of y, that x is already in C at this point.

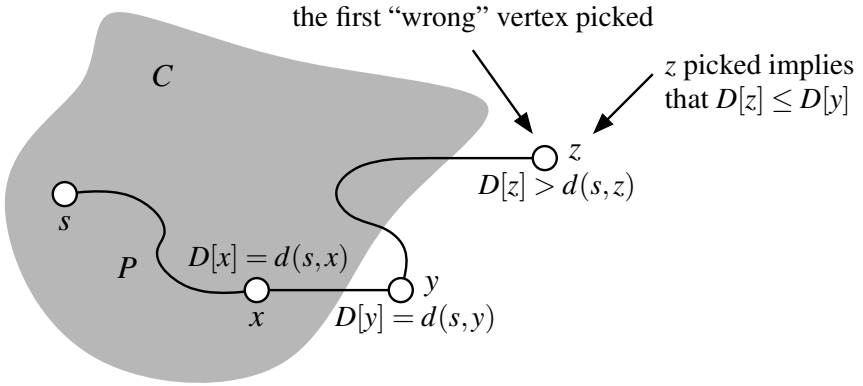

Figure 14.18: A schematic illustration for the justification of Proposition 14.23.

Moreover, $D[x] = d(s,x)$, since z is the ***first*** incorrect vertex. When x was pulled into C, we tested (and possibly updated) $D[y]$ so that we had at that point

$$D[y] \leq D[x] + w(x,y) = d(s,x) + w(x,y).$$

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

$$D[y] = d(s,y).$$

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

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

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

$$d(s,y) + d(y,z) = d(s,z).$$

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

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

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

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 Code Fragment 14.12, 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 or adjacency map structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. Therefore, the time spent in the management of the nested **for** loop, and the number of iterations of that loop, is

$$\sum_{u \text{ in } V_G} \text{outdeg}(u),$$

which is $O(m)$ by Proposition 14.9. The outer **while** loop executes $O(n)$ times, since a new vertex is added to the cloud during each iteration. This still does not settle all the details for the algorithm analysis, however, for we must say more about how to implement the other principal data structure in the algorithm—the priority queue Q.

Referring back to Code Fragment 14.12 in search of priority queue operations, we find that n vertices are originally inserted into the priority queue; since these are the only insertions, the maximum size of the queue is n. In each of n iterations of the **while** loop, a call to remove_min is made to extract the vertex u with smallest D label from Q. Then, for each neighbor v of u, we perform an edge relaxation, and may potentially update the key of v in the queue. Thus, we actually need an implementation of an ***adaptable priority queue*** (Section 9.5), in which case the key of a vertex v is changed using the method update(ℓ, k), where ℓ is the locator for the priority queue entry associated with vertex v. In the worst case, there could be one such update for each edge of the graph. Overall, the running time of Dijkstra's algorithm is bounded by the sum of the following:

- n insertions into Q.
- n calls to the remove_min method on Q.
- m calls to the update method on Q.

If Q is an adaptable priority queue implemented as a heap, then each of the above operations run in $O(\log n)$, and so the overall running time for Dijkstra's algorithm is $O((n+m)\log n)$. 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.

Let us now consider an alternative implementation for the adaptable priority queue Q using an unsorted sequence. (See Exercise P-9.58.) This, of course, requires that we spend $O(n)$ time to extract the minimum element, but it affords very fast key updates, provided Q supports location-aware entries (Section 9.5.1). 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 entry 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.