Algorithm Design:

Foundations, Analysis, and Internet Examples

Michael T. Goodrich
Department of Information and Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

John Wiley & Sons, Inc.

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.

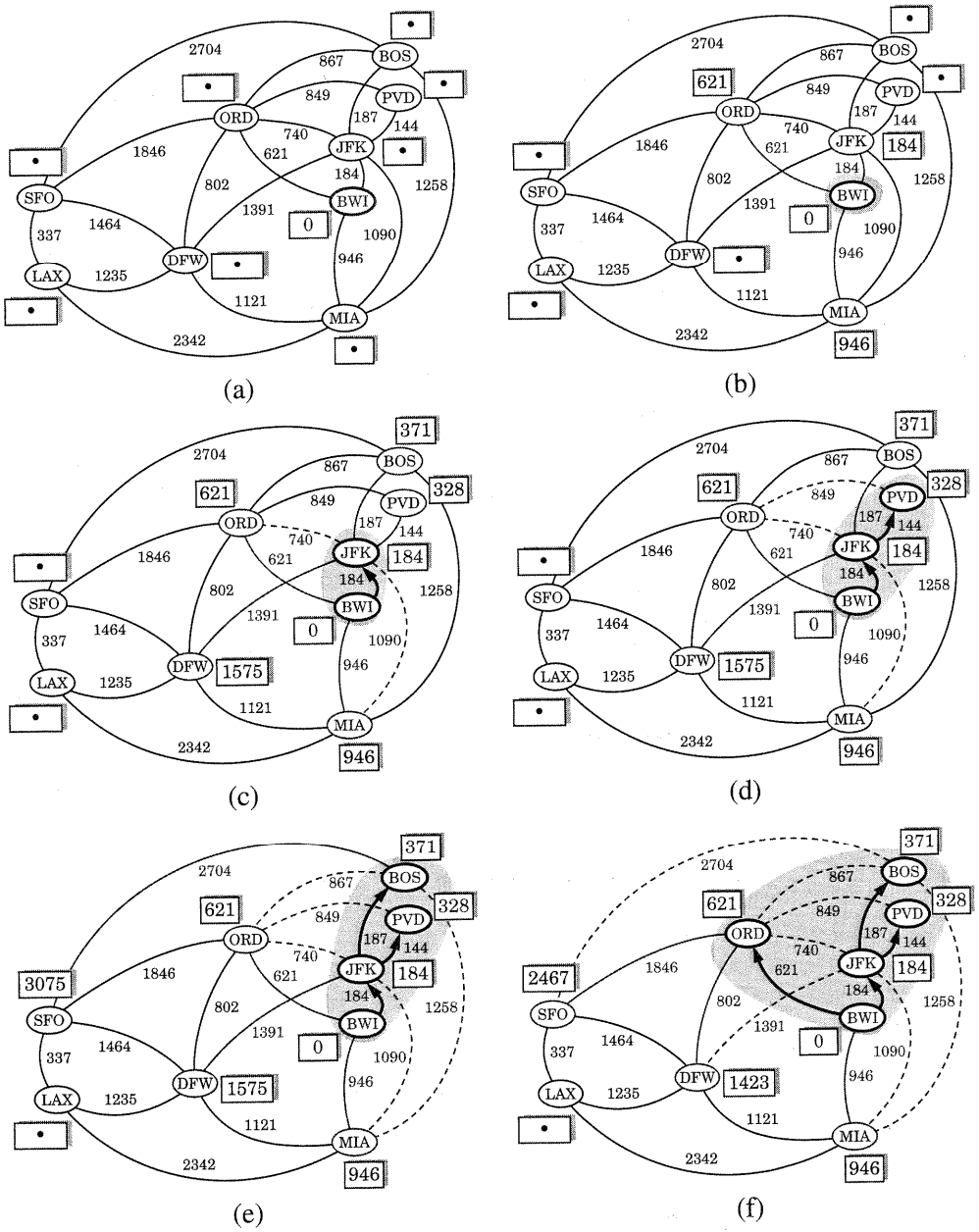

Figure 7.3: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex u stores the label $D[u]$. The symbol $\bullet$ is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick arrows, and for each vertex u outside the "cloud"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

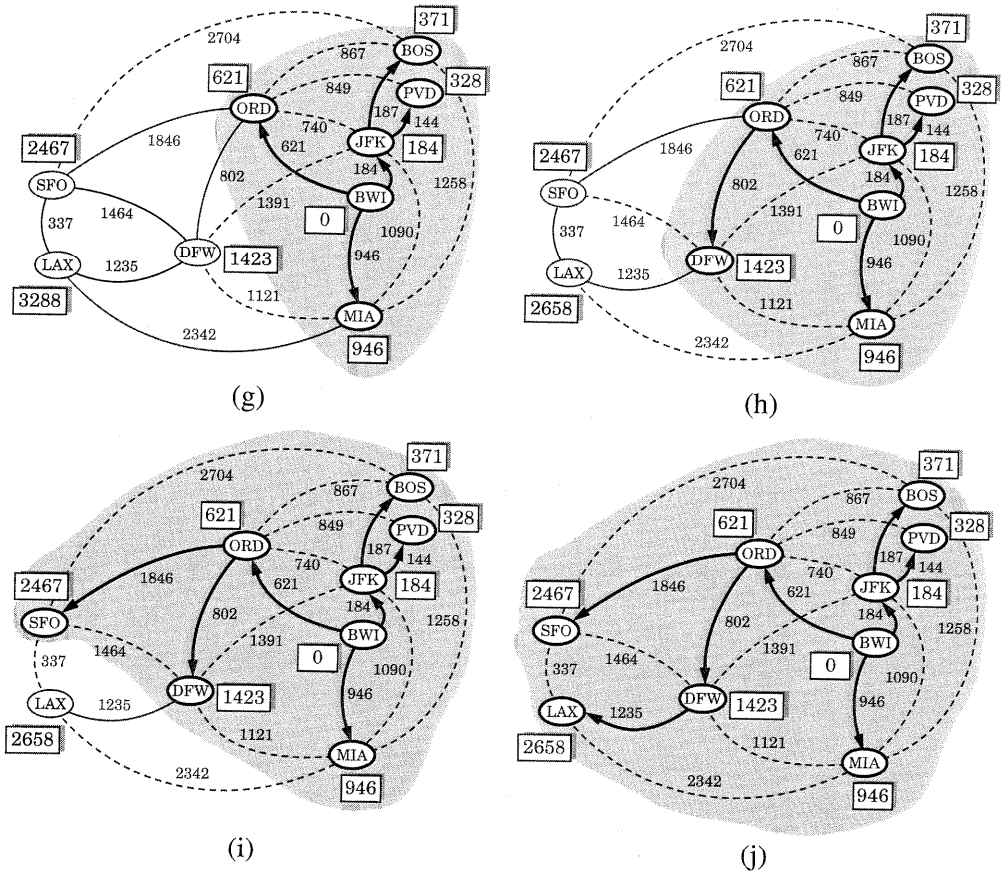

Figure 7.4: Visualization of Dijkstra's algorithm. (Continued from Figure 7.3.)

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.