Chapter 14 Shortest Paths

Lightning strike, 2009. U.S. government image. NOAA.

Contents

In a ***road network***, the interconnection structure of a set of roads is modeled as a graph whose vertices are intersections and dead ends in the set of roads, and edges are defined by segments of road that exists between pairs of such vertices. In such contexts, we often would like to find the shortest path that exists between two vertices in the road network. For example, such problems arise in GPS mapping contexts where we are interested in minimizing the driving distance between two points. Here, it typically would be inappropriate to consider one path shorter than another simply because it uses a fewer number of edges. Edges in a road network usually have varying lengths, and it can take longer to traverse some edges than it does others. Thus, edges in road networks have lengths, which represent some notion of distance for the edges, such as driving distance or driving time. Therefore, the length of a path in a road network is the sum of the lengths of the edges in that path, not the number of edges in the path.

In general, 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 explore in this chapter how to solve shortest path problems for weighted graphs, so as to solve such problems as finding optimal driving directions in road networks. We show an example of a weighted graph in Figure 14.1.

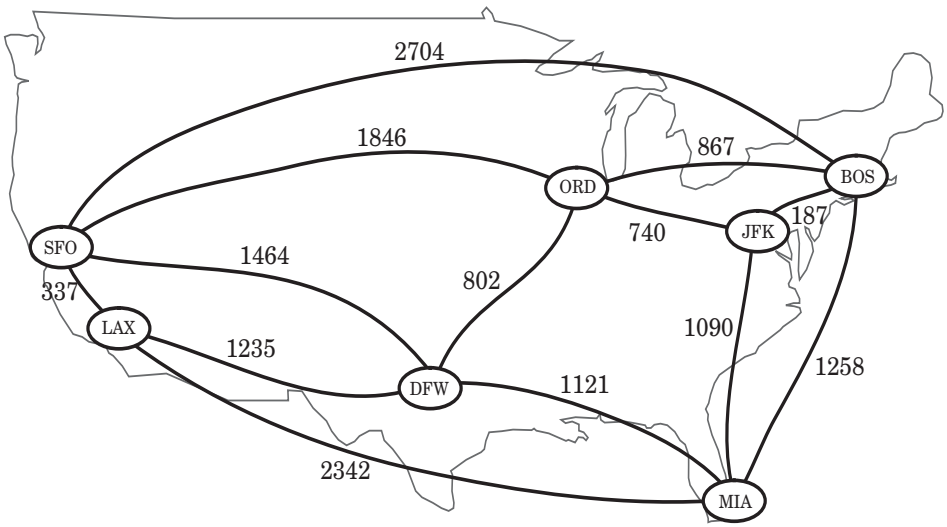

Figure 14.1: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the shortest path in the graph from JFK to LAX.

14.1 Single-Source Shortest Paths

Let G be a weighted graph. The ***length*** (or ***weight***) of a path, P, in G, 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 the next few sections, 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 13.3. There is an interesting approach for solving this single-source shortest problem based on the ***greedy method*** (Chapter 10), which is known as Dijkstra's algorithm.

The second single-source algorithm we discuss, the Bellman-Ford algorithm, is for the case where edges can have negative weights, and it does not use a greedy strategy. The next single-source algorithm we consider also allows for negative-weight edges, but it is specialized for directed acyclic graphs and it is instead based on a greedy strategy.

14.2 Dijkstra's Algorithm

A productive approach for 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.

A Greedy Method for Finding Shortest Paths

Applying the greedy method to the single-source shortest-path problem in this way results in an algorithm known as ***Dijkstra's algorithm***. In order to simplify our 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-14.2).

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 whether 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 pseudocode for Dijkstra's algorithm in Algorithm 14.2. 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 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 the key for vertex z in Q to $D[z]$
return the label $D[u]$ of each vertex u

Algorithm 14.2: 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 14.3 and 14.4.

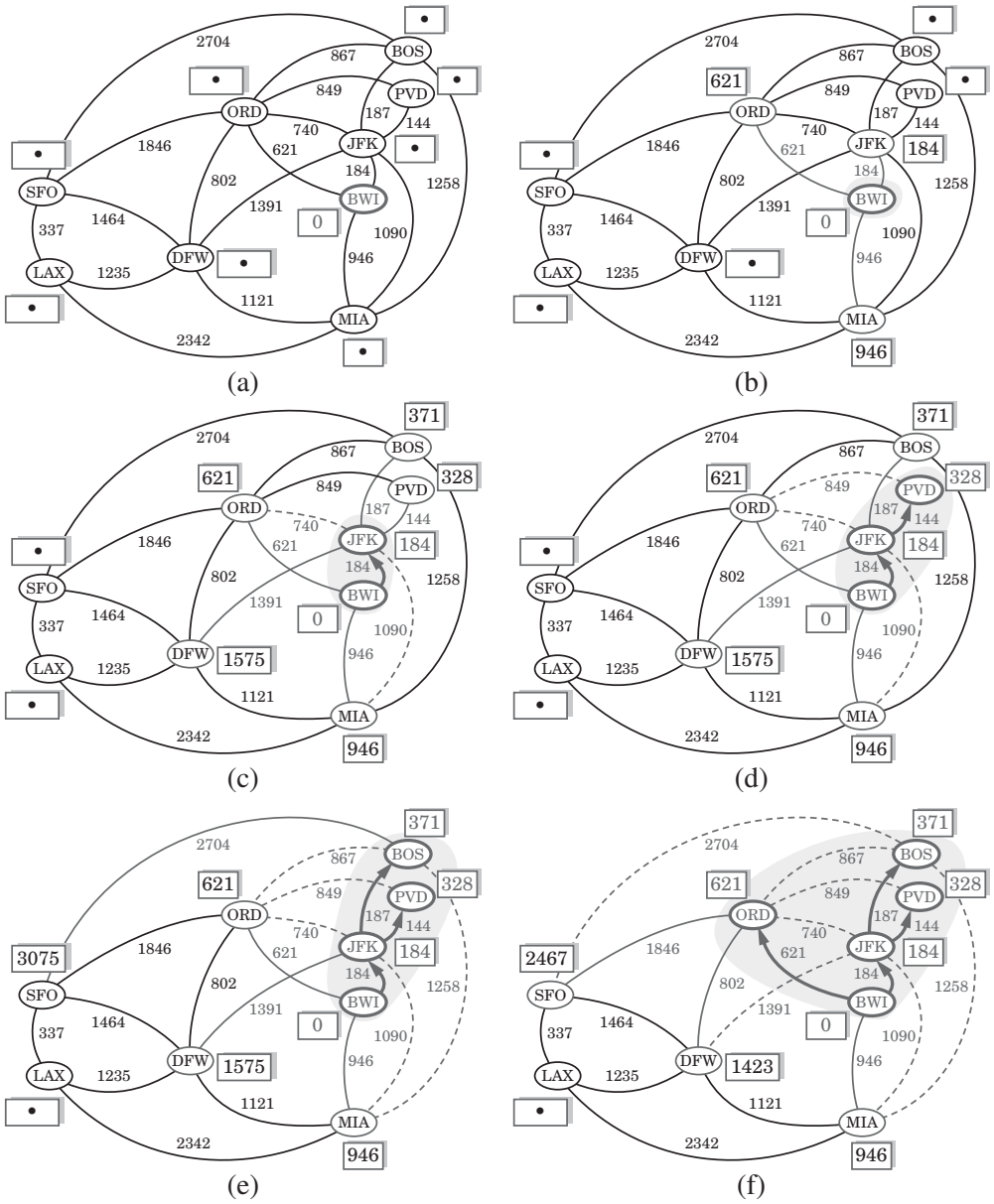

Figure 14.3: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex u stores the label $D[u]$. The symbol • is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick arrows, and for each vertex u outside the "cloud" we show the current best edge for pulling in u with a solid line. (Continued in Figure 14.4.)

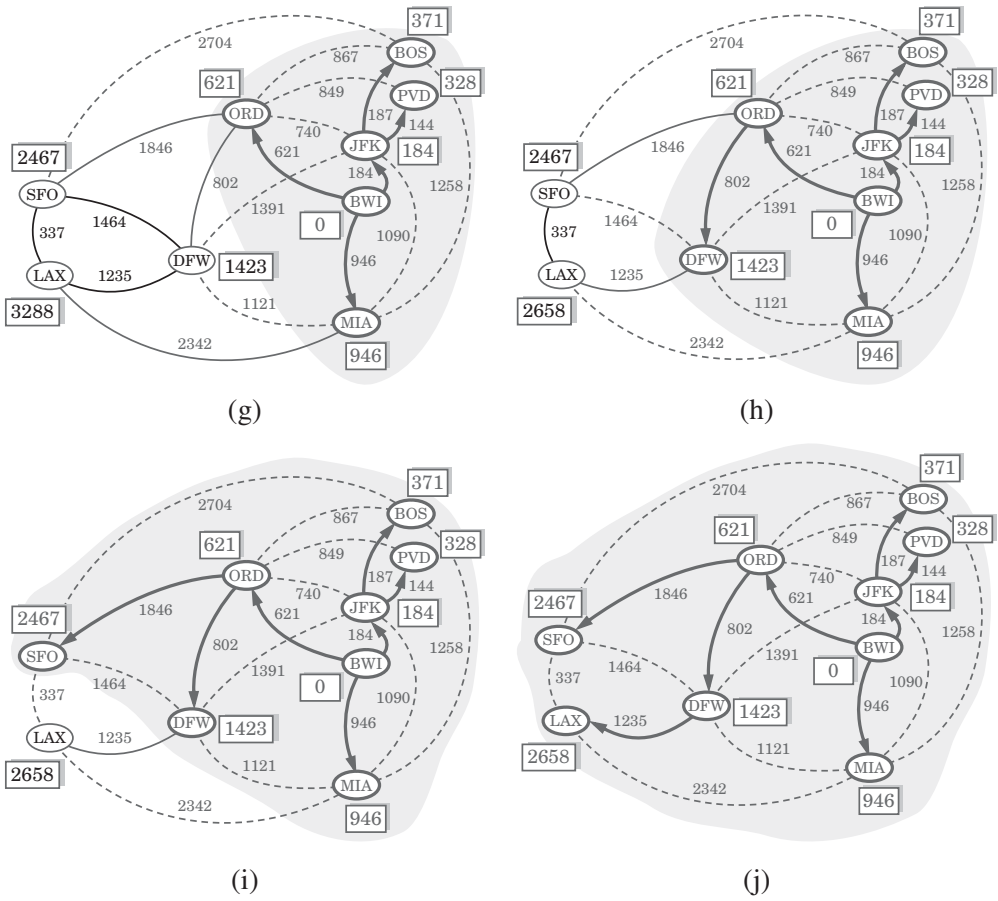

Figure 14.4: Visualization of Dijkstra's algorithm. (Continued from Figure 14.3.)

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 14.1: *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 14.5.) 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

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

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 are picking 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. ■

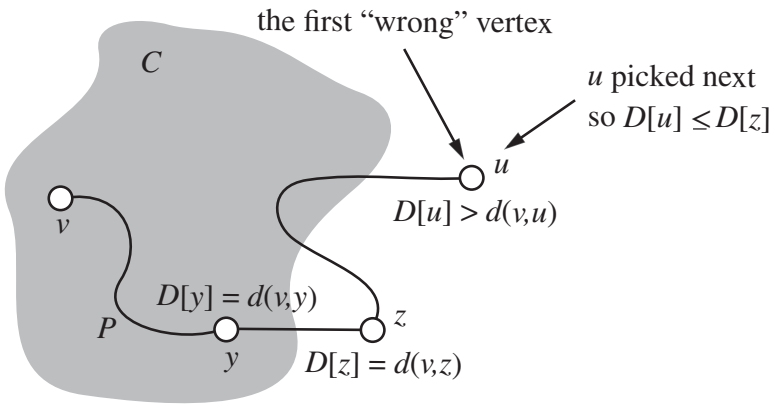

Figure 14.5: A schematic illustration for the justification of Lemma 14.1.

The Running Time of Dijkstra's Algorithm

Let us analyze the time complexity of Dijkstra's algorithm, where we use n and m to denote 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 14.2, 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 5.3). 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 pseudocode, 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. The standard heap data structure doesn't normally support a removal method for arbitrary elements, however. Instead, it supports insertion of items given as key-value pairs and the repeated removal of an item with smallest key. We can extend our priority queue implementation to support a removal operation, however, by using the locator concept described in Section 5.5. In Dijkstra's algorithm, this approach is roughly equivalent to our maintaining a pointer with each vertex, v, that supports constant-time access to the node in our heap that is holding v. Given a pointer to this node, we can remove v or update its key and perform the associated up-heap or down-heap bubbling as needed in $O(\log n)$ time.

Assuming this implementation of Q, implies that Dijkstra's algorithm runs in $O((n+m)\log n)$ time. Referring back to Algorithm 14.2, the details of this 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 5.4).
- 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 13.6.

Thus, we can implement Dijkstra's algorithm in $O(m \log n)$ time, but this is not the only way to implement this algorithm. There is an alternative implementation of Dijkstra's algorithm based on implementing the priority queue, Q, using an unsorted doubly linked list. This, of course, requires that we spend $O(n)$ time to remove the item with minimum key, but it allows for very fast key updates, provided Q supports use of the locator pattern or something like it. That is, we would need to support constant-time access from any vertex, v, to the node in the linked list for Q that is holding v. For example, maintaining a pointer for each vertex, v, to the node in Q that is holding v would suffice for this purpose.

This approach would allow us to implement each key update done in a relaxation step in $O(1)$ time, since we could 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.

Thus, we have at least two choices for implementing the priority queue in Dijkstra's algorithm. The two implementations we explored above are a locator-based heap implementation, which yields an algorithm with an $O(m \log n)$ running time, and a locator-based unsorted sequence implementation, which yields an $O(n^2)$-time algorithm. (In addition, we explore yet another way of implementing Dijkstra's algorithm in Exercise C-14.3, which avoids the use of locators.) Thus, we have the following.

Theorem 14.2: *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-14.3, 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 14.6 illustrates.

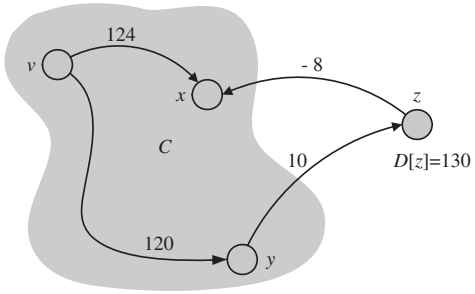

Figure 14.6: An illustration of why Dijkstra's algorithm fails for graphs with negative-weight edges. Bringing z into C and performing edge relaxations will invalidate the previously computed shortest-path distance (124) to x.

14.3 The Bellman-Ford 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-14.2). 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 is iteratively "relaxed" until it exactly equals this distance.

The Bellman-Ford method is shown in Algorithm 14.7. It performs $n-1$ times a relaxation of every edge in the digraph. We illustrate an execution of the Bellman-Ford algorithm in Figure 14.8.

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] ← 0
for each vertex u ≠ v of G⃗ do
    D[u] ← +∞
for i ← 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] ← 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 "G⃗ contains a negative-weight cycle"
```

Algorithm 14.7: The Bellman-Ford single-source shortest-path algorithm.

Lemma 14.3: *If, at the end of the execution of Algorithm 14.7, there is an edge (u,z) such that $D[u] + w((u,z)) < D[z]$, then the input digraph, $\vec{G}$, contains a negative-weight cycle. Otherwise, $D[u] = d(v,u)$ for each vertex u in $\vec{G}$.*

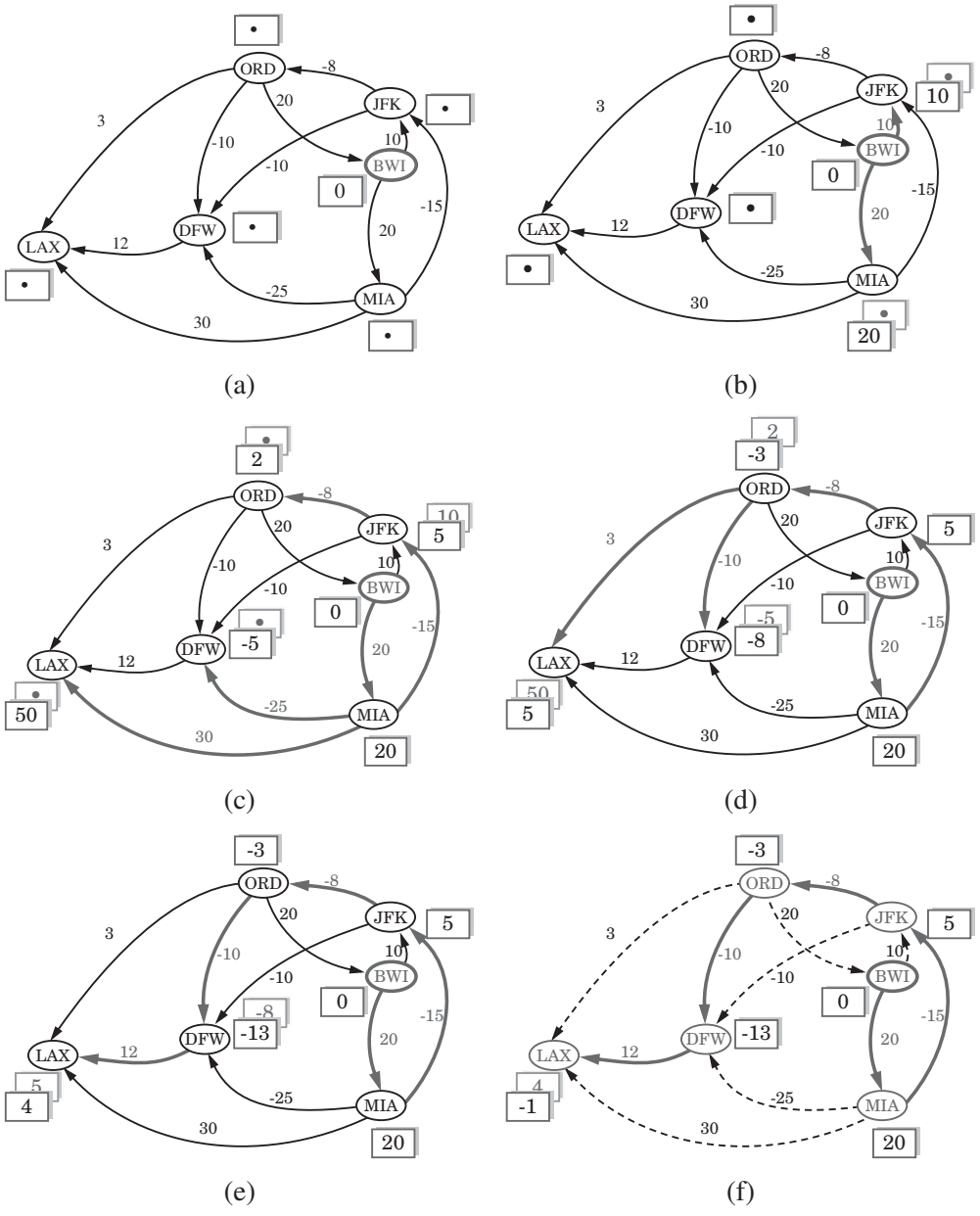

Figure 14.8: An illustration of an application of the Bellman-Ford algorithm. The start vertex is BWI. A box next to each vertex u stores the label $D[u]$, with "shadows" showing values revised during relaxations; the thick edges are causing such relaxations.

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 14.4: *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.*

14.4 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 13.4.4 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 method, which is given in Algorithm 14.9, 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, ..., v_n) for G
D[s] ← 0
for each vertex u ≠ s of G do
    D[u] ← +∞
for i ← 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] ← D[v_i] + w((v_i, u))
Output the distance labels D as the distances from s.
```

Algorithm 14.9: 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 14.10.

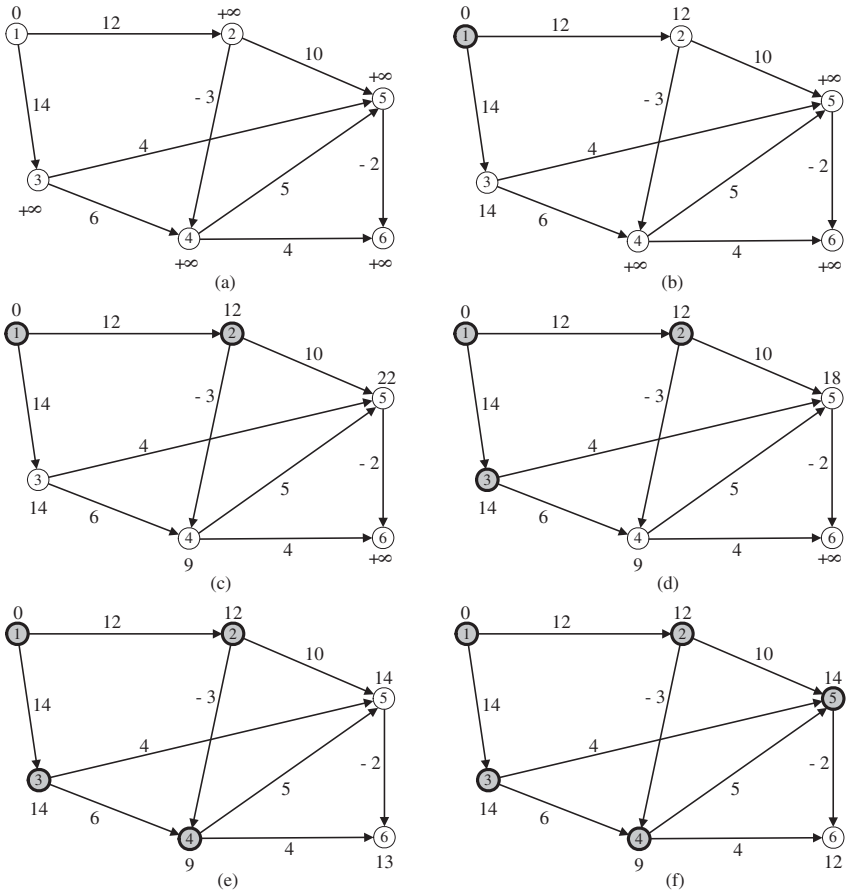

Figure 14.10: An illustration of the shortest-path algorithm for a DAG.

Theorem 14.5: 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. ■

14.5 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).

14.5.1 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 13.13).

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 (Chapter 12), the key construct in the algorithm is to define a parametrized 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 parametrized 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

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

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 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 14.11, 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 $O(n^3)$, which implies the following.

Theorem 14.6: *Given a simple weighted directed graph $\vec{G}$ with n vertices and no negative-weight cycles, Algorithm 14.11 (*AllPairsShortestPaths*) computes the shortest-path distances between each pair of vertices of $\vec{G}$ in $O(n^3)$ time.*

14.5.2 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

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

with the operators redefined 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 14.7: *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 24.2.1, 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 incoming or outgoing 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 14.12.

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,

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

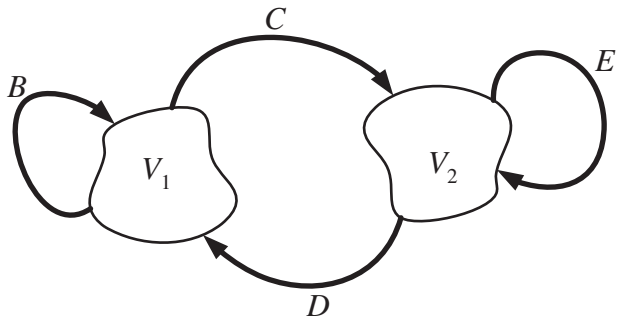

Figure 14.12: An illustration of the four sets of edges used to partition the adjacency matrix A in the divide-and-conquer algorithm for computing A^*.

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 14.8: *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^*, 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 (11.4), we can compute A^* in $O(n^3)$ time. ∎

14.6 Exercises

Reinforcement

R-14.1 Draw a simple, connected, weighted, undirected graph with 8 vertices and 16 edges, and with distinct edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.

R-14.2 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-14.3 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-14.4 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-14.5 The dynamic programming algorithm of Algorithm 14.11 uses $O(n^3)$ space. Describe a version of this algorithm that uses $O(n^2)$ space.

R-14.6 The dynamic programming algorithm of Algorithm 14.11 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-14.7 Consider the unsorted sequence implementation of the priority queue Q used in Dijkstra's algorithm. In this case, why 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-14.8 Describe the meaning of the graphical conventions used in Figures 14.3 and 14.4 illustrating Dijkstra's algorithm. What do the arrows signify? How about thick lines and dashed lines?

Creativity

C-14.1 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-14.2 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-14.3 There is an alternative way of implementing Dijkstra's algorithm that avoids use of the locator pattern but increases the space used for the priority queue, Q, from $O(n)$ to $O(m)$ for a weighted graph, G, with n vertices and m edges. The main

idea of this approach is simply to insert a new key-value pair, $(D[v], v)$, each time the $D[v]$ value for a vertex, v, changes, without ever removing the old key-value pair for v. This approach still works, even with multiple copies of each vertex being stored in Q, since the first copy of a vertex that is removed from Q is the copy with the smallest key. Describe the other changes that would be needed to the description of Dijsktra's algorithm for this approach to work. Also, what is the running time of Dijkstra's algorithm in this approach if we implement the priority queue, Q, with a heap?

C-14.4 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 counterexample.

C-14.5 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-14.6 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 $1 \leq i, j \leq n$, 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 $1 \leq i \leq n$, $M(i,i) = 0$.
2: for $1 \leq i, j \leq n$, $M(i,j) = weight(i,j)$ if $(i,j) \in E$.
3: for $1 \leq i, j \leq n$, $M(i,j) = \infty$ if $(i,j) \notin E$.
Also, let M^2 be defined, for $1 \leq i, j \leq n$, 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-14.7 Suppose you are given a connected weighted undirected graph, G, with n vertices and m edges, such that the weight of each edge in G is an integer in the interval $[1, c]$, for a fixed constant $c > 0$. Show how to solve the single-source shortest-paths problem, for any given vertex v, in G, in time $O(n + m)$.

Hint: Think about how to exploit the fact that the distance from v to any other vertex in G can be at most $O(cn) = O(n)$.

C-14.8 Suppose that every shortest path from some vertex, v, in an n-vertex weighted graph, G, to any other vertex in G has at most $k < n - 1$ edges. Show that it is sufficient to run the Bellman-Ford algorithm for only k iterations, instead of $n - 1$, to solve the single-source shortest-paths problem for v in G.

Applications

A-14.1 In a ***side-scrolling video game***, a character moves through an environment from, say, left-to-right, while encountering obstacles, attackers, and prizes. The goal is to avoid or destroy the obstacles, defeat or avoid the attackers, and collect as many prizes as possible while moving from a starting position to an ending position. We can model such a game with a graph, G, where each vertex is a game position, given as an (x, y) point in the plane, and two such vertices, v and w, are connected by an edge, given as a straight line segment, if there is a single movement that connects v and w. Furthermore, we can define the cost, $c(e)$, of an edge to be a combination of the time, health points, prizes, etc., that it costs our character to move along the edge e (where earning a prize on this edge would be modeled as a negative term in this cost). A path, P, in G is ***monotone*** if traversing P involves a continuous sequence of left-to-right movements, with no right-to-left moves. Thus, we can model an optimal solution to such a side-scrolling computer game in terms of finding a minimum-cost monotone path in the graph, G, that represents this game. Describe and analyze an efficient algorithm for finding a minimum-cost monotone path in such a graph, G.

A-14.2 Suppose that CONTROL, a secret U.S. government counterintelligence agency based in Washington, D.C., has build a communication network that links n stations spread across the world using m communication channels between pairs of stations. Suppose further that the evil spy agency, KAOS, is able to eavesdrop on some number, k, of these channels and that CONTROL knows the k channels that have been compromised. Now, CONTROL has a message, M, that it wants to send from its headquarters station, s, to one of its field stations, t. The problem is that the message is super secret and should traverse a path that minimizes the number of compromised edges that occur along this path. Explain how to model this problem as a shortest-path problem, and describe and analyze an efficient algorithm to solve it.

A-14.3 Suppose you live far from work and are trying to determine the best route to drive from your home to your workplace. In order to solve this problem, suppose further that you have downloaded, from a government website, a weighted graph, G, representing the entire road network for your state. Although the edges in G are labeled with their lengths, you are more interested in the amount of time that it takes to traverse each edge. So you have found another website that has a

function, $f_{i,j}$, defined for each edge, $e = (i, j)$, in G, such that each $f_{i,j}$ maps a time of day, t, to the amount of time it takes to go from i to j along the edge, $e = (i, j)$, if you enter that edge at time t. Here, time is measured in minutes and times of day are measured in terms of minutes since midnight. In addition, we assume that you will be leaving for work in the morning and you live close enough to your workplace so that you can get there before midnight. Moreover, the $f_{i,j}$ functions are defined to satisfy the normal rules of traffic flow, so that it is never possible to get to the end of an edge, (i, j), sooner than someone who entered that edge before you. That is, if $t_1 < t_2$, then

$$f_{i,j}(t_2) + t_2 - t_1 > f_{i,j}(t_1).$$

Describe an efficient algorithm that, given G and the $f_{i,j}$ functions for its edges, can determine, for any given time, t_0, that you might leave your home in the morning, the amount of time required for you to drive to work. What is the running time of your algorithm?

A-14.4 Suppose you are given a *timetable*, which consists of the following:

- 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 $f \in \mathcal{F}$:
  - 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?

A-14.5 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 and analyze an efficient algorithm to help you pick up the most gold in this maze while traversing a path from the start to the finish.

A-14.6 A part of doing business internationally involves the trading of different currencies, and the markets that facilitate such trades can fluctuate during a trading day in ways that create profit opportunities. For example, at a given moment during

a trading day, 1 U.S. dollar might be worth 0.98 Canadian dollar, 1 Canadian dollar might be worth 0.81 euros, and 1 euro might be worth 1.32 U.S. dollars. Sometimes, as in this example, it is possible for us to perform a cyclic sequence of currency exchanges, all at the same time, and end up with more money than we started with, which is an action known as ***currency arbitrage***. For instance, with the above exchange rates, we could perform a cyclic sequence of trades from U.S. dollars, to Canadian dollars, to euros, and back to U.S. dollars, which could turn $1,000,000 into $1,047,816, ignoring the commissions and other overhead costs for performing currency exchanges (which we will indeed be ignoring in this exercise). Suppose you are given a complete directed graph, $\vec{G}$, that represents the currency exchange rates that exist at a given moment in time on a trading day. Each vertex in $\vec{G}$ is a currency, and each directed edge, (v, w), in $\vec{G}$, is labeled with an exchange rate, $r(v, w)$, which is the amount of currency w that would be exchanged for 1 unit of currency v. In order to profit from this information, you need to find, as quickly as possible, a cycle, $(v_1, v_2, \ldots, v_k, v_1)$, that maximizes the product,

$$r(v_1, v_2) \cdot r(v_2, v_3) \cdot \cdots \cdot r(v_k, v_1),$$

such that this product is strictly greater than 1. Describe and analyze an efficient dynamic programming algorithm for finding such a cycle, if it exists.

Chapter Notes

Dijkstra [56] published his single-source shortest-path algorithm in 1959. The Bellman-Ford algorithm is derived from separate publications of Bellman [24] and Ford [74].

Incidentally, the running time for 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" [76] or the "Relaxed Heap" [58].

The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [10], Even [68], Gibbons [81], Mehlhorn [158], and Tarjan [207], and the book chapter by van Leeuwen [210]. For applications of shortest paths to social networks, see the book by Easley and Kleinberg [60].