25.1-10

Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

25.2 The Floyd-Warshall algorithm

In this section, we shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***, runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles. As in Section 25.1, we follow the dynamic-programming process to develop the algorithm. After studying the resulting algorithm, we present a similar method for finding the transitive closure of a directed graph.

The structure of a shortest path

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in Section 25.1. The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The Floyd-Warshall algorithm relies on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, as Figure 25.3 illustrates. By Lemma 24.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, \ldots, k-1\}$. There-

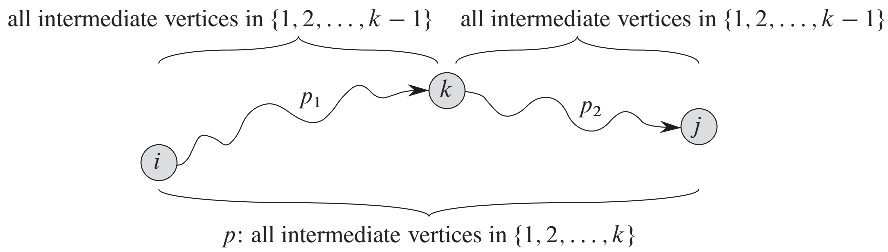


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

fore, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates that differs from the one in Section 25.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \, , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \, . \end{cases} \tag{25.5}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

Computing the shortest-path weights bottom up

Based on recurrence (25.5), we can use the following bottom-up procedure to compute the values $d_{ij}^{(k)}$ in order of increasing values of k. Its input is an $n \times n$ matrix W defined as in equation (25.1). The procedure returns the matrix $D^{(n)}$ of shortest-path weights.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Figure 25.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3–7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As in the final algorithm in Section 25.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. Exercise 25.1-6 asks you to implement this method so that it runs in $O(n^3)$ time. Given the predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure will print the vertices on a given shortest path.

Alternatively, we can compute the predecessor matrix Π while the algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

We can give a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases} \tag{25.6}$$

For $k \geq 1$, if we take the path $i \leadsto k \leadsto j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases} \tag{25.7}$$

We leave the incorporation of the $\Pi^{(k)}$ matrix computations into the FLOYD-WARSHALL procedure as Exercise 25.2-3. Figure 25.4 shows the sequence of $\Pi^{(k)}$ matrices that the resulting algorithm computes for the graph of Figure 25.1. The exercise also asks for the more difficult task of proving that the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. Exercise 25.2-7 asks for yet another way to reconstruct shortest paths.

Transitive closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, we might wish to determine whether G contains a path from i to j for all vertex pairs $i, j \in V$. We define the ***transitive closure*** of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\} .$$

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

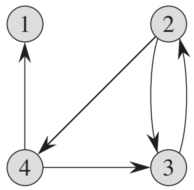
There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time that can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and $+$ in the Floyd-Warshall algorithm. For $i, j, k = 1, 2, \ldots, n$, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence (25.5), is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) . \tag{25.8}$$

As in the Floyd-Warshall algorithm, we compute the matrices $T^{(k)} = \left(t_{ij}^{(k)}\right)$ in order of increasing k.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 25.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

TRANSITIVE-CLOSURE(G)

$n = |G.V|$
let $T^{(0)} = \left(t_{ij}^{(0)}\right)$ be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        if $i == j$ or $(i, j) \in G.E$
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for $k = 1$ **to** n
    let $T^{(k)} = \left(t_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$

Figure 25.5 shows the matrices $T^{(k)}$ computed by the TRANSITIVE-CLOSURE procedure on a sample graph. The TRANSITIVE-CLOSURE procedure, like the Floyd-Warshall algorithm, runs in $\Theta(n^3)$ time. On some computers, though, logical operations on single-bit values execute faster than arithmetic operations on integer words of data. Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the Floyd-Warshall algorithm's by a factor corresponding to the size of a word of computer storage.

Exercises

25.2-1

Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

25.2-2

Show how to compute the transitive closure using the technique of Section 25.1.

25.2-3

Modify the FLOYD-WARSHALL procedure to compute the $\Pi^{(k)}$ matrices according to equations (25.6) and (25.7). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. (*Hint:* To show that $G_{\pi,i}$ is acyclic, first show that $\pi_{ij}^{(k)} = l$ implies $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$, according to the definition of $\pi_{ij}^{(k)}$. Then, adapt the proof of Lemma 24.16.)

25.2-4

As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W)
n = W.rows
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d_ij = min(d_ij, d_ik + d_kj)
return D
```

25.2-5

Suppose that we modify the way in which equation (25.7) handles equality:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

Is this alternative definition of the predecessor matrix Π correct?

25.2-6
How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

25.2-7
Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALL-PAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = \left(\phi_{ij}^{(n)}\right)$ as an input. How is the matrix Φ like the s table in the matrix-chain multiplication problem of Section 15.2?

25.2-8
Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

25.2-9
Suppose that we can compute the transitive closure of a directed acyclic graph in $f(|V|, |E|)$ time, where f is a monotonically increasing function of $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

25.3 Johnson's algorithm for sparse graphs

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. For sparse graphs, it is asymptotically faster than either repeated squaring of matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm, which Chapter 24 describes.

Johnson's algorithm uses the technique of ***reweighting***, which works as follows. If all edge weights w in a graph $G = (V, E)$ are nonnegative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex; with the Fibonacci-heap min-priority queue, the running time of this all-pairs algorithm is $O(V^2 \lg V + VE)$. If G has negative-weight edges but no negative-weight cycles, we simply compute a new set of nonnegative edge weights that allows us to use the same method. The new set of edge weights $\widehat{w}$ must satisfy two important properties:

1. For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\widehat{w}$.
2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is nonnegative.

As we shall see in a moment, we can preprocess G to determine the new weight function $\widehat{w}$ in $O(VE)$ time.

Preserving shortest paths by reweighting

The following lemma shows how easily we can reweight the edges to satisfy the first property above. We use δ to denote shortest-path weights derived from weight function w and $\widehat{\delta}$ to denote shortest-path weights derived from weight function $\widehat{w}$.

Lemma 25.1 (Reweighting does not change shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) . \tag{25.9}$$

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex v_0 to vertex v_k. Then p is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function $\widehat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\widehat{w}$.

Proof We start by showing that

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k) . \tag{25.10}$$

We have

$$\begin{aligned} \widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(because the sum telescopes)} \\ &= w(p) + h(v_0) - h(v_k) . \end{aligned}$$

Therefore, any path p from v_0 to v_k has $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Because $h(v_0)$ and $h(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using $\widehat{w}$. Thus, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\widehat{w}$. Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. By equation (25.10),

$$\begin{aligned} \widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) , \end{aligned}$$

and thus c has negative weight using w if and only if it has negative weight using $\widehat{w}$. ■

Producing nonnegative weights by reweighting

Our next goal is to ensure that the second property holds: we want $\widehat{w}(u, v)$ to be nonnegative for all edges $(u, v) \in E$. Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, we make a new graph $G' = (V', E')$, where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and $E' = E \cup \{(s, v) : v \in V\}$. We extend the weight function w so that $w(s, v) = 0$ for all $v \in V$. Note that because s has no edges that enter it, no shortest paths in G', other than those with source s, contain s. Moreover, G' has no negative-weight cycles if and only if G has no negative-weight cycles. Figure 25.6(a) shows the graph G' corresponding to the graph G of Figure 25.1.

Now suppose that G and G' have no negative-weight cycles. Let us define $h(v) = \delta(s, v)$ for all $v \in V'$. By the triangle inequality (Lemma 24.10), we have $h(v) \leq h(u) + w(u, v)$ for all edges $(u, v) \in E'$. Thus, if we define the new weights $\widehat{w}$ by reweighting according to equation (25.9), we have $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, and we have satisfied the second property. Figure 25.6(b) shows the graph G' from Figure 25.6(a) with reweighted edges.

Computing all-pairs shortest paths

Johnson's algorithm to compute all-pairs shortest paths uses the Bellman-Ford algorithm (Section 24.1) and Dijkstra's algorithm (Section 24.3) as subroutines. It assumes implicitly that the edges are stored in adjacency lists. The algorithm returns the usual $|V| \times |V|$ matrix $D = d_{ij}$, where $d_{ij} = \delta(i, j)$, or it reports that the input graph contains a negative-weight cycle. As is typical for an all-pairs shortest-paths algorithm, we assume that the vertices are numbered from 1 to $|V|$.

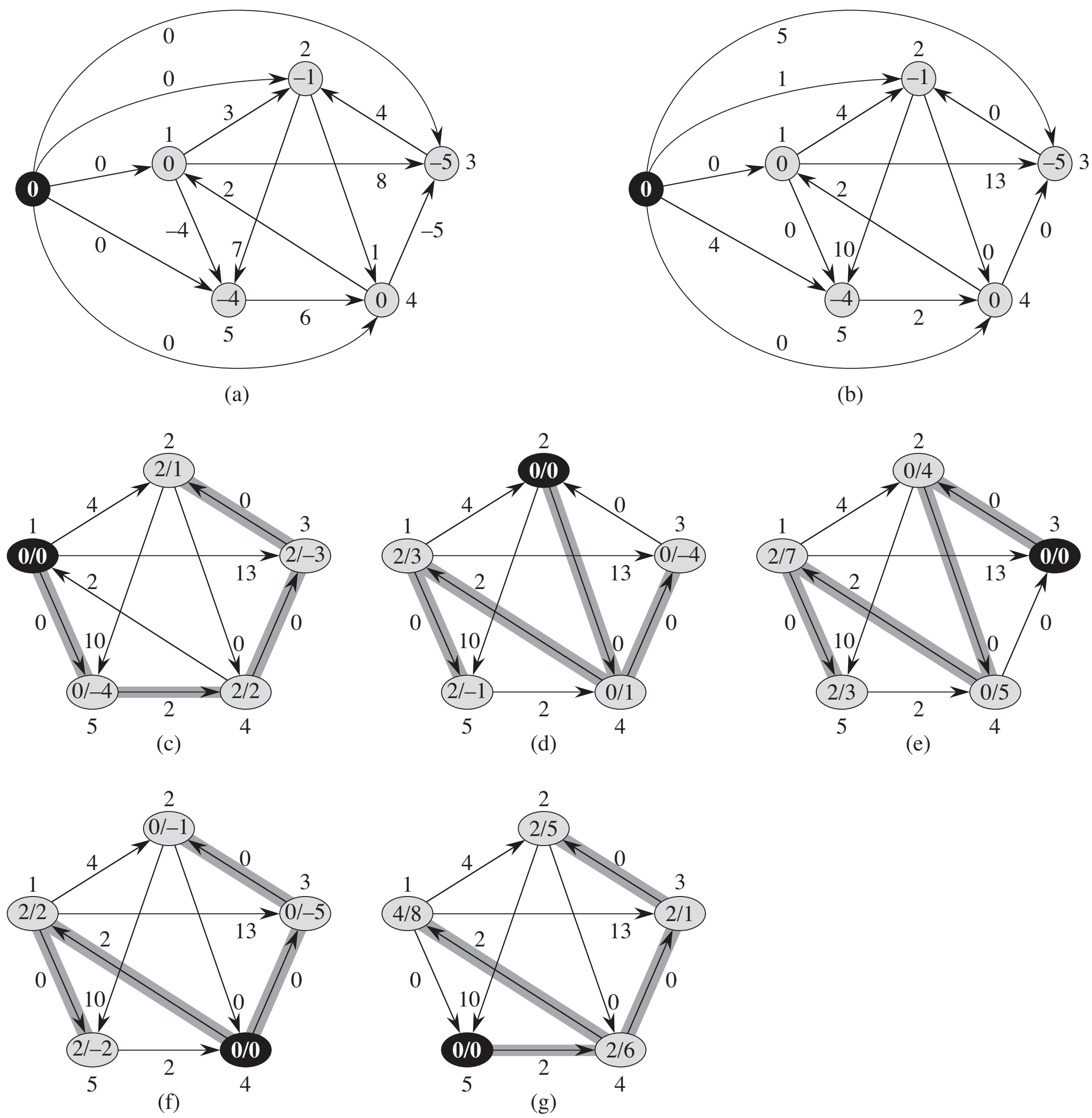


Figure 25.6 Johnson's all-pairs shortest-paths algorithm run on the graph of Figure 25.1. Vertex numbers appear outside the vertices. **(a)** The graph G' with the original weight function w. The new vertex s is black. Within each vertex v is $h(v) = \delta(s, v)$. **(b)** After reweighting each edge (u, v) with weight function $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)–(g)** The result of running Dijkstra's algorithm on each vertex of G using weight function $\hat{w}$. In each part, the source vertex u is black, and shaded edges are in the shortest-paths tree computed by the algorithm. Within each vertex v are the values $\hat{\delta}(u, v)$ and $\delta(u, v)$, separated by a slash. The value $d_{uv} = \delta(u, v)$ is equal to $\hat{\delta}(u, v) + h(v) - h(u)$.

```
JOHNSON(G, w)
1  compute G′, where G′.V = G.V ∪ {s},
         G′.E = G.E ∪ {(s, v) : v ∈ G.V}, and
         w(s, v) = 0 for all v ∈ G.V
2  if BELLMAN-FORD(G′, w, s) == FALSE
3        print "the input graph contains a negative-weight cycle"
4  else for each vertex v ∈ G′.V
5              set h(v) to the value of δ(s, v)
                    computed by the Bellman-Ford algorithm
6        for each edge (u, v) ∈ G′.E
7              ŵ(u, v) = w(u, v) + h(u) − h(v)
8        let D = (d_uv) be a new n × n matrix
9        for each vertex u ∈ G.V
10             run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11             for each vertex v ∈ G.V
12                   d_uv = δ̂(u, v) + h(v) − h(u)
13       return D
```

This code simply performs the actions we specified earlier. Line 1 produces G'. Line 2 runs the Bellman-Ford algorithm on G' with weight function w and source vertex s. If G', and hence G, contains a negative-weight cycle, line 3 reports the problem. Lines 4–12 assume that G' contains no negative-weight cycles. Lines 4–5 set $h(v)$ to the shortest-path weight $\delta(s, v)$ computed by the Bellman-Ford algorithm for all $v \in V'$. Lines 6–7 compute the new weights $\widehat{w}$. For each pair of vertices $u, v \in V$, the **for** loop of lines 9–12 computes the shortest-path weight $\widehat{\delta}(u, v)$ by calling Dijkstra's algorithm once from each vertex in V. Line 12 stores in matrix entry d_{uv} the correct shortest-path weight $\delta(u, v)$, calculated using equation (25.10). Finally, line 13 returns the completed D matrix. Figure 25.6 depicts the execution of Johnson's algorithm.

If we implement the min-priority queue in Dijkstra's algorithm by a Fibonacci heap, Johnson's algorithm runs in $O(V^2 \lg V + VE)$ time. The simpler binary min-heap implementation yields a running time of $O(VE \lg V)$, which is still asymptotically faster than the Floyd-Warshall algorithm if the graph is sparse.

Exercises

25.3-1

Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 25.2. Show the values of h and $\widehat{w}$ computed by the algorithm.

25.3-2
What is the purpose of adding the new vertex s to V, yielding V'?

25.3-3
Suppose that $w(u, v) \geq 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and $\widehat{w}$?

25.3-4
Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min_{(u,v)\in E} \{w(u, v)\}$, just define $\widehat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

25.3-5
Suppose that we run Johnson's algorithm on a directed graph G with weight function w. Show that if G contains a 0-weight cycle c, then $\widehat{w}(u, v) = 0$ for every edge (u, v) in c.

25.3-6
Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He claims that instead we can just use $G' = G$ and let s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

Problems

25-1 Transitive closure of a dynamic graph
Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.

a. Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.

b. Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.

c. Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of n insertions, your algorithm should run in total time $\sum_{i=1}^{n} t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

25-2 Shortest paths in ϵ-dense graphs

A graph $G = (V, E)$ is ***ϵ-dense*** if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \le 1$. By using d-ary min-heaps (see Problem 6-2) in shortest-paths algorithms on ϵ-dense graphs, we can match the running times of Fibonacci-heap-based algorithms without using as complicated a data structure.

a. What are the asymptotic running times for INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d-ary min-heap? What are these running times if we choose $d = \Theta(n^{\alpha})$ for some constant $0 < \alpha \le 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.

b. Show how to compute shortest paths from a single source on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (*Hint:* Pick d as a function of ϵ.)

c. Show how to solve the all-pairs shortest-paths problem on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

d. Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ-dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

Chapter notes

Lawler [224] has a good discussion of the all-pairs shortest-paths problem, although he does not analyze solutions for sparse graphs. He attributes the matrix-multiplication algorithm to the folklore. The Floyd-Warshall algorithm is due to Floyd [105], who based it on a theorem of Warshall [349] that describes how to compute the transitive closure of boolean matrices. Johnson's algorithm is taken from [192].

Several researchers have given improved algorithms for computing shortest paths via matrix multiplication. Fredman [111] shows how to solve the all-pairs shortest paths problem using $O(V^{5/2})$ comparisons between sums of edge weights and obtains an algorithm that runs in $O(V^3(\lg\lg V/\lg V)^{1/3})$ time, which is slightly better than the running time of the Floyd-Warshall algorithm. Han [159] reduced the running time to $O(V^3(\lg\lg V/\lg V)^{5/4})$. Another line of research demonstrates that we can apply algorithms for fast matrix multiplication (see the chapter notes for Chapter 4) to the all-pairs shortest paths problem. Let $O(n^\omega)$ be the running time of the fastest algorithm for multiplying $n \times n$ matrices; currently $\omega < 2.376$ [78]. Galil and Margalit [123, 124] and Seidel [308] designed algorithms that solve the all-pairs shortest paths problem in undirected, unweighted graphs in $(V^\omega p(V))$ time, where $p(n)$ denotes a particular function that is polylogarithmically bounded in n. In dense graphs, these algorithms are faster than the $O(VE)$ time needed to perform $|V|$ breadth-first searches. Several researchers have extended these results to give algorithms for solving the all-pairs shortest paths problem in undirected graphs in which the edge weights are integers in the range $\{1, 2, \ldots, W\}$. The asymptotically fastest such algorithm, by Shoshan and Zwick [316], runs in time $O(WV^\omega p(VW))$.

Karger, Koller, and Phillips [196] and independently McGeoch [247] have given a time bound that depends on E^*, the set of edges in E that participate in some shortest path. Given a graph with nonnegative edge weights, their algorithms run in $O(VE^* + V^2 \lg V)$ time and improve upon running Dijkstra's algorithm $|V|$ times when $|E^*| = o(E)$.

Baswana, Hariharan, and Sen [33] examined decremental algorithms for maintaining all-pairs shortest paths and transitive-closure information. Decremental algorithms allow a sequence of intermixed edge deletions and queries; by comparison, Problem 25-1, in which edges are inserted, asks for an incremental algorithm. The algorithms by Baswana, Hariharan, and Sen are randomized and, when a path exists, their transitive-closure algorithm can fail to report it with probability $1/n^c$ for an arbitrary $c > 0$. The query times are $O(1)$ with high probability. For transitive closure, the amortized time for each update is $O(V^{4/3} \lg^{1/3} V)$. For all-pairs shortest paths, the update times depend on the queries. For queries just giving the shortest-path weights, the amortized time per update is $O(V^3/E \lg^2 V)$. To report the actual shortest path, the amortized update time is $\min(O(V^{3/2}\sqrt{\lg V}), O(V^3/E \lg^2 V))$. Demetrescu and Italiano [84] showed how to handle update and query operations when edges are both inserted and deleted, as long as each given edge has a bounded range of possible values drawn from the real numbers.

Aho, Hopcroft, and Ullman [5] defined an algebraic structure known as a "closed semiring," which serves as a general framework for solving path problems in directed graphs. Both the Floyd-Warshall algorithm and the transitive-closure algorithm from Section 25.2 are instantiations of an all-pairs algorithm based on closed semirings. Maggs and Plotkin [240] showed how to find minimum spanning trees using a closed semiring.