(c) Obtain a recurrence between A^k and A^{k-1} similar to (5.8). Use the logical operators **or** and **and** rather than **min** and $+$.

(d) Write an algorithm, using the recurrence of part (c), to find A^*. Your algorithm can use only $O(n^2)$ space. What is its time complexity?

(e) Show that $A^+ = A \times A^*$, where matrix multiplication is defined as $A^+(i,j) = \vee_{k=1}^{n}(A(i,k) \wedge A^*(k,j))$. The operation $\vee$ is the logical **or** operation, and $\wedge$ the logical **and** operation. Hence A^+ may be computed from A^*.

5.4 SINGLE-SOURCE SHORTEST PATHS: GENERAL WEIGHTS

We now consider the single-source shortest path problem discussed in Section 4.8 when some or all of the edges of the directed graph G may have negative length. ShortestPaths (Algorithm 4.14) does not necessarily give the correct results on such graphs. To see this, consider the graph of Figure 5.9. Let $v = 1$ be the source vertex. Referring back to Algorithm 4.14, since $n = 3$, the loop of lines 12 to 22 is iterated just once. Also $u = 3$ in lines 15 and 16, and so no changes are made to $dist[\ ]$. The algorithm terminates with $dist[2] = 7$ and $dist[3] = 5$. The shortest path from 1 to 3 is $1, 2, 3$. This path has length 2, which is less than the computed value of $dist[3]$.

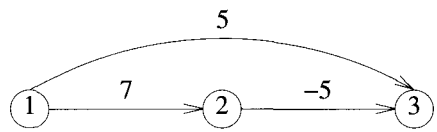


Figure 5.9 Directed graph with a negative-length edge

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges. For example, in the graph of Figure 5.5, the length of the shortest path from vertex 1 to vertex 3 is $-\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \cdots, 1, 2, 3$$

can be made arbitrarily small as was shown in Example 5.14.

When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n - 1$ edges

on it. To see this, note that a path that has more than $n-1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of ShortestPaths (Algorithm 4.14), we compute only the length, $dist[u]$, of the shortest path from the source vertex v to u. An exercise examines the extension needed to construct the shortest paths.

Let $dist^{\ell}[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most ℓ edges. Then, $dist^{1}[u] = cost[v,u]$, $1 \leq u \leq n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n-1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k-1$ edges, then $dist^{k}[u] = dist^{k-1}[u]$.

2. If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $\langle j,u \rangle$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i,u \rangle$ is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i,u]$ is the correct value for j.

These observations result in the following recurrence for $dist$:

$$dist^{k}[u] = \min \{dist^{k-1}[u], \min_{i} \{dist^{k-1}[i] + cost[i,u]\}\}$$

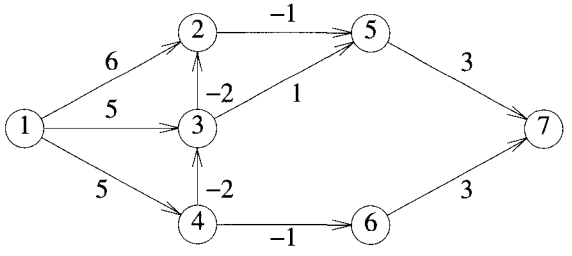
This recurrence can be used to compute $dist^{k}$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n-1$.

Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^{k}$, $k = 1, \ldots, 6$. These arrays were computed using the equation just given. For instance, $dist^{k}[1] = 0$ for all k since 1 is the source node. Also, $dist^{1}[2] = 6, dist^{1}[3] = 5$, and $dist^{1}[4] = 5$, since there are edges from

1 to these nodes. The distance $dist^1[]$ is ∞ for the nodes 5, 6, and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min\ \{dist^1[2], \min_i dist^1[i] + cost[i,2]\} \\ &= \min\ \{6, 0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\} = 3 \end{aligned}$$

Here the terms $0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty$, and $\infty+\infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7, respectively. The rest of the entries are computed in an analogous manner. □



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

An exercise shows that if we use the same memory location $dist[u]$ for $dist^k[u]$, $k = 1, \ldots, n-1$, then the final value of $dist[u]$ is still $dist^{n-1}[u]$. Using this fact and the recurrence for *dist* shown above, we arrive at the pseudocode of Algorithm 5.4 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

Each iteration of the **for** loop of lines 7 to 12 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. Here e is the number of edges in the graph. The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the *dist* values change on one iteration of the **for** loop of lines 7 to 12, then none will change on successive iterations. So, this loop can be rewritten to terminate either after $n-1$ iterations or after the

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Algorithm 5.4 Bellman and Ford algorithm to compute shortest paths

first iteration in which no *dist* values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose *dist* values changed on the previous iteration of the **for** loop. These are the only values for i that need to be considered in line 10 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 7 to 12 so that on each iteration, a vertex i is removed from the queue, and the *dist* values of all vertices adjacent from i are updated as in lines 11 and 12. Vertices whose *dist* values decrease as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. These two strategies to improve the performance of BellmanFord are considered in the exercises. Other strategies for improving performance are discussed in References and Readings. □

EXERCISES

1. Find the shortest paths from node 1 to every other node in the graph of Figure 5.11 using the Bellman and Ford algorithm.

2. Prove the correctness of BellmanFord (Algorithm 5.4). Note that this algorithm does not faithfully implement the computation of the recurrence for $dist^k$. In fact, for $k < n-1$, the *dist* values following iteration k of the **for** loop of lines 7 to 12 may not be $dist^k$.

3. Transform BellmanFord into a program. Assume that graphs are represented using adjacency lists in which each node has an additional field