Kruskal's Algorithm and Prim's

CHAPTER 21

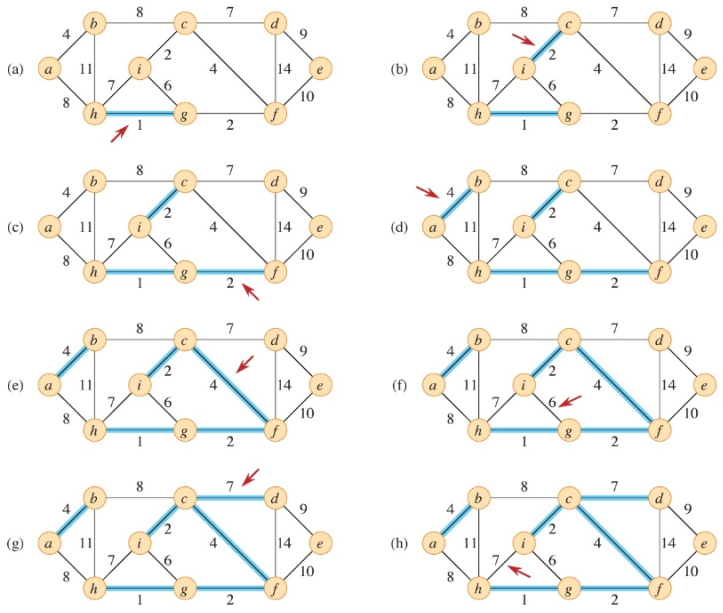


Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

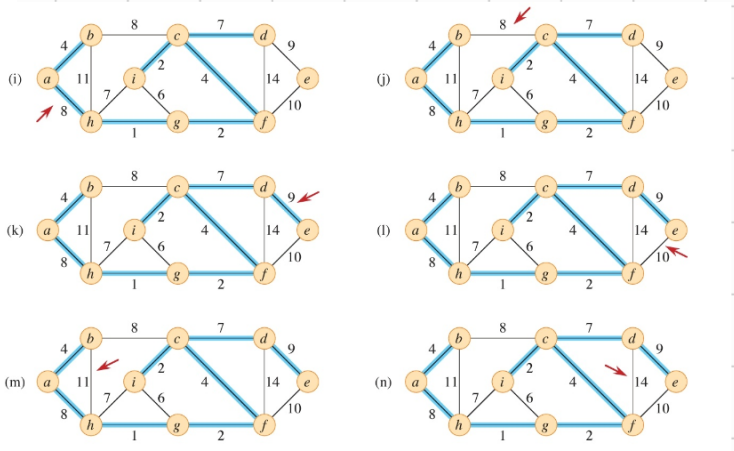


Figure 21.4, continued Further steps in the execution of Kruskal's algorithm.

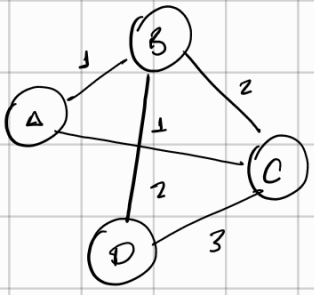
+ Use a disjoint-set data structure to process the edges.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7      if FIND-SET(u) ≠ FIND-SET(v)
8          A = A ∪ {(u, v)}
9          UNION(u, v)
10 return A
```

tion known. Initializing the set A in line 1 takes $O(1)$ time, creating a single list of edges in line 4 takes $O(V + E)$ time (which is $O(E)$ because G is connected), and the time to sort the edges in line 5 is $O(E \lg E)$. (We'll account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 6–9 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these disjoint-set operations take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 19.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\,\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

21.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.



Algorithm to sort the edges to return T:

→ For this we are going to make a slight modification on how KRUSKAL'S Algo. break ties.

1. Let T be the desired MST of G
2. Assign each edge in G a pair: $(w(e), p(e))$
   where w is the weight
   p is the priority of it $\begin{cases} 0 & \text{if } e \notin T \\ 1 & \text{if } e \in T. \end{cases}$
3. Sort the edges lexicographically by:

→ Primary key: w(e) (edge weight, ascending manner)
→ Second key: p(e) (priority, T-edges first.

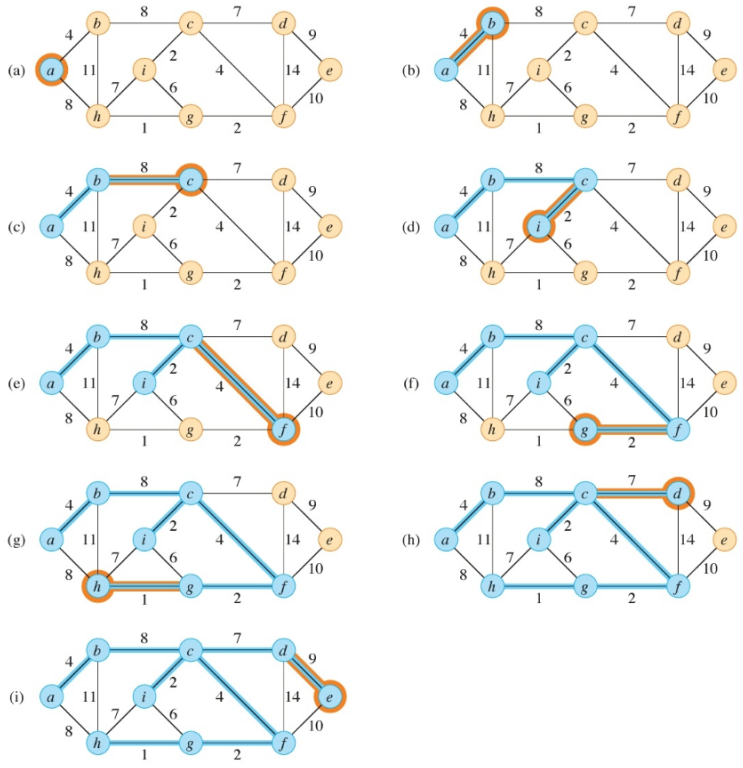


Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL          // root key
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap (see Chapter 6), including a way to map between vertices and their corresponding heap elements. The BUILD-MIN-HEAP procedure can perform lines 5–7 in $O(V)$ time. In fact, there is no need to call BUILD-MIN-HEAP. You can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 10–14 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, the test for membership in Q in line 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE-KEY in line 14 takes $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

You can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap (see page 478). If a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and each INSERT and DECREASE-KEY operation takes only $O(1)$ amortized time. Therefore, by using a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

21.2-2

Give a simple implementation of Prim's algorithm that runs in $O(V^2)$ time when the graph $G = (V, E)$ is represented as an adjacency matrix.

→ The graph is represented as a V x V matrix.
→ G[u][v] is the weight between vertices u and v.
→ Access of it takes O(1), but iteration through all the neighbors is O(V).

Pseudocode

```
PRIM-ADJLIST (G, V):
for each vertex u ∈ V:
    u.key = ∞; u.π = NIL;
    MST_set = {FALSE}
Key[0] = 0    // start from vertex 0 (for this implementation)
```

```
parent[0] = -1    // Root, doesn't have parent
for i = 0 to V-1:    O(V)
    u = EXTRACT-MIN(key, MST_set)  // vertex still not in the MST.
    MST_set[u] = TRUE
    // We need to update the values of adj vertecis of u.
    for v = 0 to V-1:    O(V)
        if G[u][v] ≠ 0 and G[u][v] < key[v] and MST_set[v] = FALSE:
            key[v] = G[u][v]
            parent[v] = u
```

21.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Answer:

We know that Kruskal's algorithm takes $O(V)$ time for initialization, $O(E \lg E)$ time to sort the edges, and $O(E\,\alpha(V))$ time for the disjoint-set operations, for a total running time of $O(V + E \lg E + E\,\alpha(V)) = O(E \lg E)$.

If we knew that all of the edge weights in the graph were integers in the range from 1 to $|V|$, then we could sort the edges in $O(V + E)$ time using counting sort. Since the graph is connected, $V = O(E)$, and so the sorting time is reduced to $O(E)$. This would yield a total running time of $O(V + E + E\,\alpha(V)) = O(E\,\alpha(V))$, again since $V = O(E)$, and since $E = O(E\,\alpha(V))$. The time to process the edges, not the time to sort them, is now the dominant term. Knowledge about the weights won't help speed up any other part of the algorithm, since nothing besides the sort uses the weight values.

If the edge weights were integers in the range from 1 to W for some constant W, then we could again use counting sort to sort the edges more quickly. This time, sorting would take $O(E + W) = O(E)$ time, since W is a constant. As in the first part, we get a total running time of $O(E\,\alpha(V))$.

★ ***21.2-7***

Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

Key Points about Kruskal algorithm:

- It sorts all the edges in $O(E \log E)$,
- With uniformly distributed weight, we can take advantage of this structure and use Bucket Sort or Radix sort to achieve $O(E)$
- Kruskal Algo. needs $O(V)$ for initialization, $O(E)$ (with the above discussed) for sorting and $O(E\alpha(V))$ for disjoint set operations.

And for definition $\alpha(n)$ we know is the inverse Ackerman function which grows extremely slowly, such as $\alpha(n) \leq 4$ for any reasonable n in real-world applications. An improve is observable!

Key Points of Prim's algorithm:

- Is based in two key operations EXTRACT-MIN, executed V times than $O(V \log V)$
- DECREASE-KEY, executed E times, this give us $O(E \log V)$

Thus operation doesn't get any benefit from the uniform distribution.

Thus, $O(V \log V + E \log V)$

Relations Between $|V|$ and $|E|$

For a connected graph:

- The minimum number of edges ($|E|$) is $|V| - 1$ (a spanning tree)
- In a dense graph and complete graph the maximum number of edges ($|E|$) is $|V|^2$.

Thus:

$$|V| - 1 \leq |E| \leq |V|^2$$

$$\log|V| - 1 \leq \log|E| \leq 2\log|V|$$

ER

CHAPT 22 Dijkstra's, Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

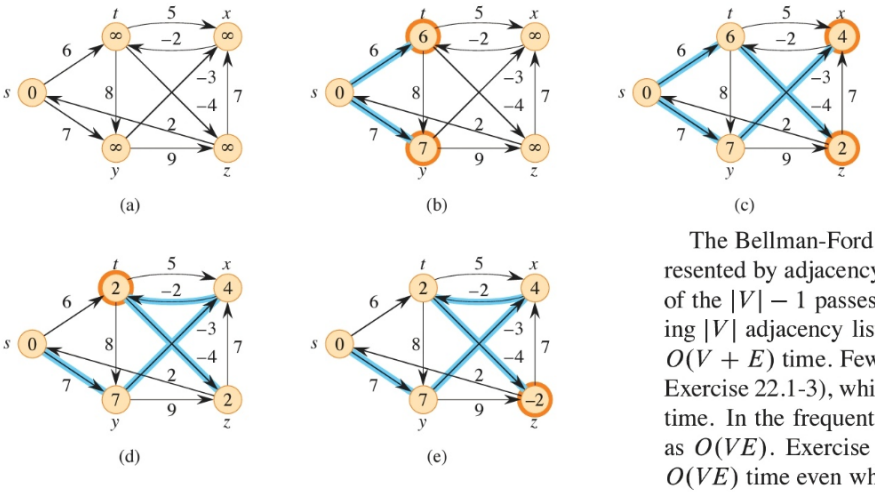


Figure 22.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and blue edges indicate predecessor values: if edge (u, v) is blue, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. Vertices whose shortest-path estimates and predecessors have changed due to a pass are highlighted in orange. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

The Bellman-Ford algorithm runs in $O(V^2 + VE)$ time when the graph is represented by adjacency lists, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(V + E)$ time (examining $|V|$ adjacency lists to find the $|E|$ edges), and the **for** loop of lines 5–7 takes $O(V + E)$ time. Fewer than $|V| - 1$ passes over the edges sometimes suffice (see Exercise 22.1-3), which is why we say $O(V^2 + VE)$ time, rather than $\Theta(V^2 + VE)$ time. In the frequent case where $|E| = \Omega(V)$, we can express this running time as $O(VE)$. Exercise 22.1-5 asks you to make the Bellman-Ford algorithm run in $O(VE)$ time even when $|E| = o(V)$.

22.1-3 ★

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Answer:

If the greatest number of edges on any shortest path from the source is m, then the path-relaxation property tells us that after m iterations of BELLMAN-FORD, every vertex v has achieved its shortest-path weight in $v.d$. By the upper-bound property, after m iterations, no d values will ever change. Therefore, no d values will change in the $(m + 1)$st iteration. Because we do not know m in advance, we cannot make the algorithm iterate exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after $m + 1$ iterations.

```
BELLMAN-FORD-(M+1)(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  changes = TRUE
  while changes == TRUE
      changes = FALSE
      for each edge (u, v) ∈ G.E
          RELAX-M(u, v, w)

RELAX-M(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
      changes = TRUE
```

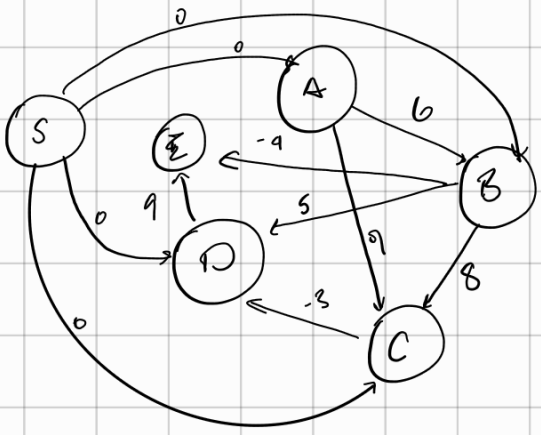
The test for a negative-weight cycle (based on there being a d value that would change if another relaxation step was done) has been removed above, because this version of the algorithm will never get out of the **while** loop unless all d values stop changing.

Pragmatism of Bellman-Ford Algorithm:

- For sparse graph (where $E \approx V$), the running time is closer to V^2.
- For dense graphs (where $E \approx V^2$), the running time approaches $O(V^3)$
- It is slower than Dijkstra Algo. on graphs with non-negative weights.

22.1-6

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for all vertices $v \in V$, the value $\delta^*(v) = \min\{\delta(u, v) : u \in V\}$.



Visualization

Algorithm

1. Create a super node S, for a $G = (V, E)$
Augmented graph $G' = (V \cup \{S\}, E')$
where $E' = E \cup \{(s, u) \mid u \in V\}$
2. Run Bellman-Ford from the source node S that we created.
   → For G' we have $V = |V| + 1 \approx |V|$
   $E' = |E| + |V| \approx |E|$
   → Thus the Algorithm still runs in $O(VE)$
3. The distance $dist[v]$ from S to v in G' will be $\min_{u \in V} \delta(u, v)$

22.1-7

Suppose that a weighted, directed graph $G = (V, E)$ contains a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Copy code

```
BellmanFordListNegativeCycle(G, s):
    // G = (V,E) with weight function w(u,v)
    // s is the source vertex
    // dist[] holds shortest distance estimates
    // π[] holds predecessor information

    for each vertex v in V:
        dist[v] = ∞
        π[v] = null
    dist[s] = 0

    // Relax edges |V|-1 times
    for i = 1 to |V|-1:
        for each edge (u,v) in E:
            if dist[u] + w(u,v) < dist[v]:
                dist[v] = dist[u] + w(u,v)
                π[v] = u

    // Check for a negative-weight cycle
    cycle_vertex = null
    for each edge (u,v) in E:
        if dist[u] + w(u,v) < dist[v]:
            dist[v] = dist[u] + w(u,v)
            π[v] = u
            cycle_vertex = v
```

```
    // If we reach here, cycle_vertex is part of or leads to a negative cycle.
    // Move cycle_vertex back |V| times along predecessors to ensure we're on the
cycle.
    y = cycle_vertex
    for i = 1 to |V|:
        y = π[y]

    // Now y is guaranteed to be on the cycle.
    // Follow predecessors from y until we get back to y to list the cycle.
    cycle = []
    z = y
    repeat
        cycle.append(z)
        z = π[z]
    until z == y

    // cycle now contains all vertices on one negative cycle
    print("Negative-weight cycle detected: ", cycle)
```

Shortest Paths in directed acyclic graphs (DAG)

DAG-SHORTEST-PATHS(G, w, s)
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** each vertex $u \in G.V$, taken in topologically sorted order
4 **for** each vertex v in $G.Adj[u]$
5 RELAX(u, v, w)

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

INITIALIZE-SINGLE-SOURCE(G, s)
1 **for** each vertex $v \in G.V$
2 $v.d = \infty$
3 $v.\pi$ = NIL
4 $s.d = 0$

RELAX(u, v, w)
1 **if** $v.d > u.d + w(u, v)$
2 $v.d = u.d + w(u, v)$
3 $v.\pi = u$

Program Evaluation and Review Technique chart.

22.2-3 ☆

An alternative way to represent a PERT chart looks more like the dag of Figure 20.7 on page 574. Vertices represent tasks and edges represent sequencing constraints, that is, edge (u, v) indicates that task u must be performed before task v. Vertices, not edges, have weights. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

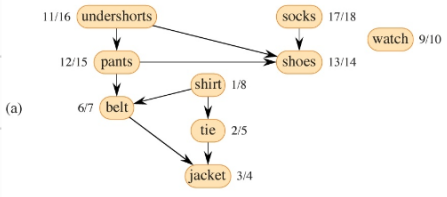


Figure 20.7

→ Instead of finding the shortest path we want to find the longest one.

INITIALIZE-SINGLE-SOURCE-VERTICES(G, s, dist):
for each vertex v in G:
w[v] = vertex's own weight
dist[v] = $-\infty$
π[v] = NIL
dist[s] = w[s] // The vertex own weight

DAG-LONGEST-PATH(G, w, s)
Topological sort vertices in G
INITIALIZE-SINGLE-SOURCE-VERTICES(G, s)
for each vertex u in G.V in topo order:
for each v in G.adj[u]
RELAX-LONGEST(u, v, w)
Find the vertex with the max distance
Backtrack the path using the π array.

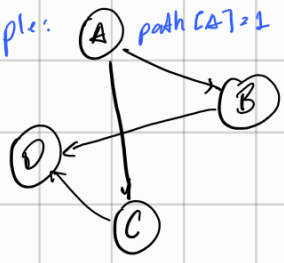
```
RELAX-LONGEST(u, v, w)
    if dist[u] + w[v] > dist[v]:
        dist[v] = dist[u] + w[v]
        π[v] = u
```

Procedure is similar to the DAG-SHORTEST-PATH, and we know that the algorithm has a $\Theta(V+E)$ complexity time. Therefore we can conclude our algorithm runs in linear time.

★ ***22.2-4***

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. The count should include all paths between all pairs of vertices and all paths with 0 edges. Analyze your algorithm.

Example:



path[B] = 2
{B, A→B}
path[C] = 2
{C, A→C}
path[D] = 5
{D, B→D, A→B→D, C→D, A→C→D}

Total = 10 paths

```
DAG-COUNTING-PATHS(G)
Topological sord the vertices in G:
Initialize a path[vertices] = {1}   // The trivial path
                                    // paths with 0 edges
for vertex u in G.V in topological order:
    for vertex v in G.adj[u]:
        path[v] += path[u]
for v in path array:
    paths_count += path[v]
return paths_count.
```

Complexity analysis:

- → Topological sort takes $O(V+E)$
- → Initialization, counting of paths takes $O(V)$
- → Iterate over the edges in topological sort and its neighbors to compute the paths $O(V+E)$

Dijkstra Algorithm

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

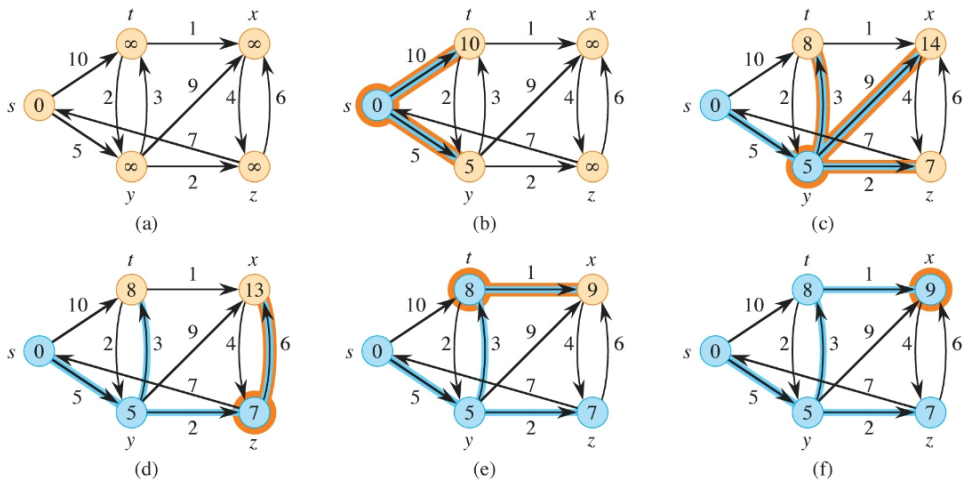


Figure 22.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set S, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex u in line 7, and each edge highlighted in orange caused a d value and a predecessor to change when the edge was relaxed. The d values and predecessors shown in part (f) are the final values.

22.3-2

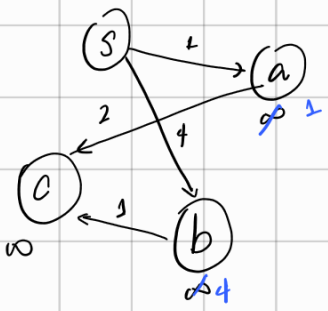
Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces an incorrect answer. Why doesn't the proof of Theorem 22.6 go through when negative-weight edges are allowed?

→ Any graph with a negative cycle. Relaxation of the edges is called a finite number of times. However, the distance to any vertex in the cycle is $-\infty$, Dijkstra algorithm failed to find the new shortest path.
From the Theorem about the correctness of Dijkstra algorithm, the proof doesn't go through negative edges, due there is no guarantee that the condition: $\delta(s,y) \leq \delta(s,u)$ will hold and from the proof we realize that if allow negative weights u.d may decrease further and u.d is not minimum anymore among V - S.

22.3-6

Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm relaxes the edges of a shortest path out of order.

→ The order is not important, Dijkstra uses a priority queue



to process the vertices.

Properties mentioned in the proof of Theorem 22.6 (Correctness of Dijkstra)

1. No path property: If there is no path from the source vertex s to a vertex u, then the shortest-path distance from s to u denoted as $\delta(s,u)$ is infinity (∞).

2. Path Relaxation Property:

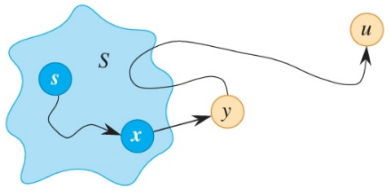


Figure 22.7 The proof of Theorem 22.6. Vertex u is selected to be added into set S in line 7 of DIJKSTRA. Vertex y is the first vertex on a shortest path from the source s to vertex u that is not in set S, and $x \in S$ is y's predecessor on that shortest path. The subpath from y to u may or may not re-enter set S.

Let y be the vertex directly before u on the shortest path from s to u, and let x be the predecesor of y in S.

By induction, $x.d = \delta(s,x)$ and $y.d = \delta(s,y)$ because y was relaxed during a prior iteration when x was added to S.

When x was added to S, the edge (x,y) was relaxed. By the converge property of Dijkstra algorithm, $y.d = \delta(s,y)$. Since y precedes u, the edge (y,u) is relaxed during the current iteration giving

$$u.d = \delta(s,u)$$

3. Upper-Bound property: At any point during the execution of the algorithm, $v.d \geq \delta(s,u)$ for all vertices u.

From the relaxation step, for an edge (u,v), the relaxation step updates: $v.d = \min(v.d, u.d + u \xrightarrow{\text{edge}} v)$.

Relaxation reduces $v.d$ only if $u.d + u \to v < v.d$, since $u.d \geq \delta(s,u)$ (inductive assumption): $v.d \geq \delta(s,v)$.

→ Following by induction at every step $u.d$ is reduced to $\delta(s,u)$ or remains $u.d > \delta(s,u)$. The distance between nodes never underestimate the shodest path $\delta(s,u)$. Relaxation ensures $v.d$ decrease, but $v.d \geq \delta(s,v)$ is true.

22.3-7

Consider a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \le r(u, v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v. Interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Answer:

To find the most reliable path between s and t, run Dijkstra's algorithm with edge weights $w(u, v) = -\lg r(u, v)$ to find shortest paths from s in $O(E+V \lg V)$ time. The most reliable path is the shortest path from s to t, and that path's reliability is the product of the reliabilities of its edges.

Here's why this method works. Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path $s \overset{p}{\rightsquigarrow} t$ such that $\prod_{(u,v)\in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg(\prod_{(u,v)\in p} r(u, v)) = \sum_{(u,v)\in p} \lg r(u, v)$, which is in turn equivalent to minimizing $\sum_{(u,v)\in p} -\lg r(u, v)$. (Note: $r(u, v)$ can be 0, and $\lg 0$ is undefined. So in this algorithm, define $\lg 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\lg r(u, v)$, we have a shortest-path problem.

Since $\lg 1 = 0$, $\lg x < 0$ for $0 < x < 1$, and we have defined $\lg 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \lg V)$ time.

Alternative solution

You can also work with the original probabilities by running a modified version of Dijkstra's algorithm that maximizes the product of reliabilities along a path instead of minimizing the sum of weights along a path.

In Dijkstra's algorithm, use the reliabilities as edge weights and substitute

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue,
- $\cdot$ for $+$ in relaxation,
- 1 (identity for $\cdot$) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max).

For example, we would use the following instead of the usual RELAX procedure:

```
RELAX-RELIABILITY(u, v, r)
  if v.d < u.d · r(u, v)
      v.d = u.d · r(u, v)
      v.π = u
```

22.3-11 ☆

Suppose that you are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

→ The negative edges are only affecting the source node, after processing this node all edges are non negative and the algorithm proceeds as usual.

→ To extend the explanation we know Dijkstra process the nodes using a priority queue where the vertex with minimum path is extracted. That is at the beginning the source vertex with s.d is extracted, then we relax its edges and update the values of the vertices adjacent to s. Add s to the set. Once is processed all remaining vertices only have non negative edges. Since the graph doesn't posses negative vertices, this guarantees that the graph does not cause infinite reductions in distances, and the shortest paths are well defined.

CHAPTER 23: All Pairs Shortest Paths
The Floyd - Warshall Algorithm

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

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

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

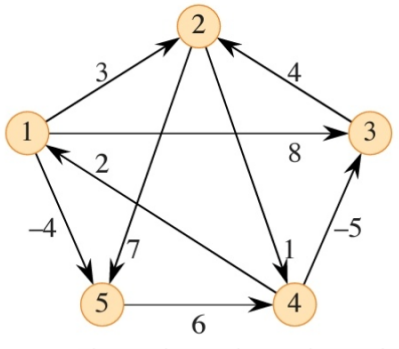
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL}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

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

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

Figure 23.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 23.1.

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



23.2-1

Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 23.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

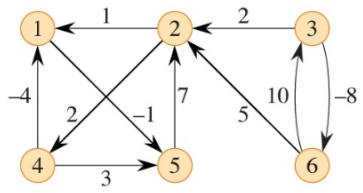


Figure 23.2 A weighted, directed graph for use in Exercises 23.1-1, 23.2-1, and 23.3-1.

$D^{(0)}$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Formula : $d[i,j]^{k} = min(d[:,j]^{k-1}, d[:,k]^{k-1}, d[k,j]^{k-1})$

$D^{(1)}$

	1	2	3	4	5	6	
1	0	∞	∞	∞	-1	∞	
2	1	0	∞	2	0	∞	
→ 3	∞	2	0	∞	∞	-8	4, 2
4	-4	∞	∞	0	-5	∞	∞
5	∞	7	∞	∞	0	∞	
6	∞	5	10	∞	∞	0	[4,1] + [1,2]
							-4 ∞

$D^{(2)}$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	∞
4		∞		0		
5		7			0	
6		5				0

It will continue ...

-4 [4,2] + [2,1]

23.2-4

As it appears on page 657, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since it creates $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the procedure FLOYD-WARSHALL′, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W, n)
1  D = W
2  for k = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              d_ij = min{d_ij, d_ik + d_kj}
6  return D
```

→ To start with the time complexity is $\Theta(n^3)$ since the algorithm creates $n \times n$ matrix $D^{(k)}$, since k can range from $k = 1$ to n, then it can create up to n matrices of size n, and thus: $O(n) \times O(n^2) = O(n^3)$

Proof:

With the superscripts, the computation is $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$. If, having dropped the superscripts, we were to compute and store d_{ik} or d_{kj} before using these values to compute d_{ij}, we might be computing one of the following:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right) ,$$
$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right) ,$$
$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right) .$$

In any of these scenarios, we're computing the weight of a shortest path from i to j with all intermediate vertices in $\{1, 2, \ldots, k\}$. If we use $d_{ik}^{(k)}$, rather than $d_{ik}^{(k-1)}$, in the computation, then we're using a subpath from i to k with all intermediate vertices in $\{1, 2, \ldots, k\}$. But k cannot be an *intermediate* vertex on a shortest path from i to k, since otherwise there would be a cycle on this shortest path. Thus, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. A similar argument applies to show that $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Hence, we can drop the superscripts in the computation.

23.2-6 ★

Show how to use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle.

A/

Here are two ways to detect negative-weight cycles:

1. Check the main-diagonal entries of the result matrix for a negative value. There is a negative weight cycle if and only if $d_{ii}^{(n)} < 0$ for some vertex i:
   - $d_{ii}^{(n)}$ is a path weight from i to itself; so if it is negative, there is a path from i to itself (i.e., a cycle), with negative weight.
   - If there is a negative-weight cycle, consider the one with the fewest vertices.
     - If it has just one vertex, then some $w_{ii} < 0$, so d_{ii} starts out negative, and since d values are never increased, it is also negative when the algorithm terminates.
     - If it has at least two vertices, let k be the highest-numbered vertex in the cycle, and let i be some other vertex in the cycle. $d_{ik}^{(k-1)}$ and $d_{ki}^{(k-1)}$ have correct shortest-path weights, because they are not based on negative-weight cycles. (Neither $d_{ik}^{(k-1)}$ nor $d_{ki}^{(k-1)}$ can include k as an intermediate vertex, and i and k are on the negative-weight cycle with the fewest vertices.) Since $i \rightsquigarrow k \rightsquigarrow i$ is a negative-weight cycle, the sum of those two weights is negative, so $d_{ii}^{(k)}$ will be set to a negative value. Since d values are never increased, it is also negative when the algorithm terminates.

   In fact, it suffices to check whether $d_{ii}^{(n-1)} < 0$ for some vertex i. Here's why. A negative-weight cycle containing vertex i either contains vertex n or it does not. If it does not, then clearly $d_{ii}^{(n-1)} < 0$. If the negative-weight cycle contains

vertex n, then consider $d_{nn}^{(n-1)}$. This value must be negative, since the cycle, starting and ending at vertex n, does not include vertex n as an intermediate vertex.

2. Alternatively, one could just run the normal FLOYD-WARSHALL algorithm one extra iteration to see if any of the d values change. If there are negative cycles, then some shortest-path cost will be cheaper. If there are no such cycles, then no d values will change because the algorithm gives the correct shortest paths.

Steps in Pseudocode

Text — Copy code

```
Floyd-Warshall-Negative-Cycle(W, n):
    D = Floyd-Warshall'(W, n)  // Run the optimized Floyd-Warshall algorithm

    for i = 1 to n:
        if D[i][i] < 0:
            print("Negative-weight cycle detected at vertex", i)
            return
    print("No negative-weight cycle found.")
```

Note:

Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

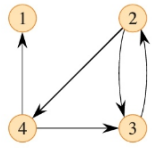
Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

23.2-8 ★

Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$. Assume that $|V| = O(E)$ and that the graph is represented with adjacency lists.

TRANSITIVE-CLOSURE(G, n)

```
1  let T^(0) = (t_ij^(0)) be a new n × n matrix
2  for i = 1 to n
3      for j = 1 to n
4          if i == j or (i, j) ∈ G.E
5              t_ij^(0) = 1
6          else t_ij^(0) = 0
7  for k = 1 to n
8      let T^(k) = (t_ij^(k)) be a new n × n matrix
9      for i = 1 to n
10         for j = 1 to n
11             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
12 return T^(n)
```



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

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

Figure 23.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

→ Solution using DFS

```
TRANSITIVE-CLOSURE(G):
    Initialize a V x V T array
    for each vertex u ∈ V:
        visited[1-|V|] = FALSE   // To track visited vertices
        DFS(u, v, T, visited)
```

```
DFS-TC(u, v, T, visited):
    T[u][v] = 1
    visited[v] = TRUE
    for each vertex w in G.Adj[v]:
        if not visited[w]
            DFS(u, w, T, visited)
```

Johnson's Algorithm:

```
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
```

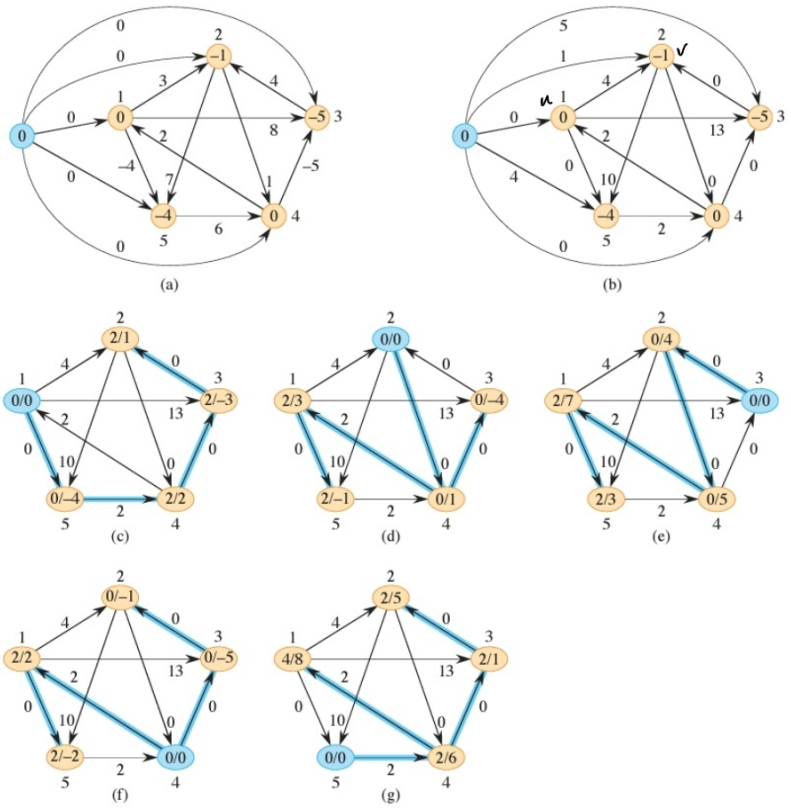


Figure 23.6 Johnson's all-pairs shortest-paths algorithm run on the graph of Figure 23.1. Vertex numbers appear outside the vertices. **(a)** The graph G' with the original weight function w. The new vertex s is blue. Within each vertex v is $h(v) = \delta(s, v)$. **(b)** After reweighting each edge (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 blue, and blue edges belong to 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 $\hat{\delta}(u, v) + h(v) - h(u)$.

23.3-1

Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 23.2. Show the values of h and $\hat{w}$ computed by the algorithm.

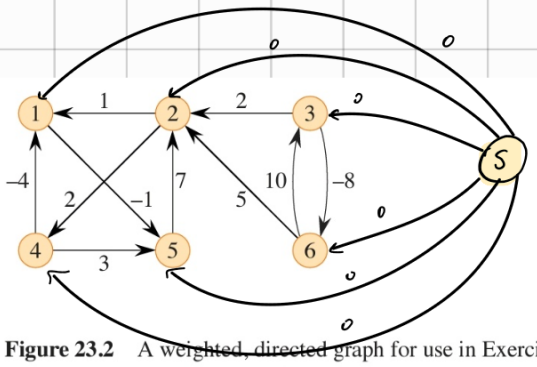
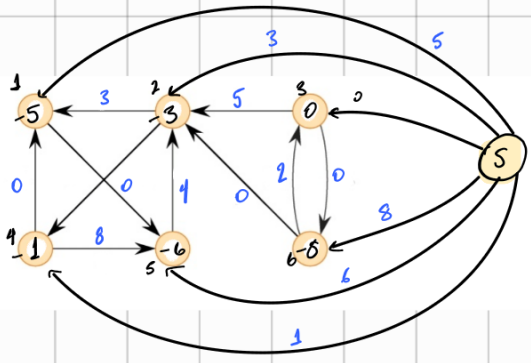


Figure 23.2 A weighted, directed graph for use in Exercises 23.1-1, 23.2-1, and 23.3-1.

Table:

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

→ After



Formula:

$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$

also $\hat{w}(u,v) = h(u) + w(u,v) - h(v)$

And finish writing a table for $\hat{w}(u,v)$

23.3-2 ★

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

→ Is to ensure that the Bellman-Ford algorithm can compute a valid shortest-path $h(v)$ for all vertices in the graph G, even if the original graph is not connected.

→ Also as for the algorithm we need to reweight the graph in order to eliminate negative edge weights and allow Dijkstra for a correct execution.

→ The estimation formula is as:

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

Let $h(v)$ be the shortest path distance from the source vertex s to vertex v using Bellman-Ford: $h(v) = \delta(s,v)$

Bellman-Ford Algo. also guarantees for any edge $(u,v) \in E$

$$h(v) \leq h(u) + w(u,v),$$

Rearranging these terms: $w(u,v) + h(u) - h(v) \geq 0$

$$\Rightarrow \hat{w}(u,v) \geq 0, \text{ and thus}$$

all reweighted edges, $\hat{w}(u,v)$, are non negative.

23.3-3

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

Answer

If all the edge weights are nonnegative, then the values computed as the shortest distances when running Bellman-Ford will be all zero. This is because when constructing G' on the first line of Johnson's algorithm, we place an edge of weight zero from s to every other vertex. Since any path within the graph has no negative edges, its cost cannot be negative, and so, cannot beat the trivial path that goes straight from s to any given vertex. Since we have that $h(u) = h(v)$ for every u and v, the reweighting that occurs only adds and subtracts 0, and so we have that $w(u, v) = \hat{w}(u, v)$

23.3-5

Show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

→ The sum over all the edges in the cycle c can be expressed as

$$\sum_{(u,v)\in c} \hat{w}(u,v) = \sum_{(u,v)\in c} [w(u,v) + h(u) - h(v)]$$

$$= \sum_{(u,v)\in c} w(u,v) + \sum_{(u,v)\in c} [h(u) - h(v)]$$

Since $\sum_{(u,v)\in c} w(u,v)$ is the weights of the edges in c that form the 0-weigh cycle, its value is 0, and we can simplify as:

$$\sum_{(u,v)\in c} \hat{w}(u,v) = 0 + \sum_{(u,v)\in c} [h(u) - h(v)]$$

$$= h(u_1) - h(u_2) + h(u_2) - h(u_3) + \dots + h(u_k) - h(u_1)$$

} all terms cancel out $= 0$

→ Thus, $\sum_{(u,v)\in c} \hat{w}(u,v) = 0$.

23.3-6

Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He suggests using $G' = G$ instead and letting 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. Assume that $\infty - \infty$ is undefined, and in particular, it is not 0. 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.

Answer

In this solution, we assume that $\infty - \infty$ is undefined; in particular, it's not 0.

Let $G = (V, E)$, where $V = \{s, u\}$, $E = \{(u, s)\}$, and $w(u, s) = 0$. There is only one edge, and it enters s. When we run Bellman-Ford from s, we get $h(s) = \delta(s, s) = 0$ and $h(u) = \delta(s, u) = \infty$. When we reweight, we get $\hat{w}(u, s) = 0 + \infty - 0 = \infty$. We compute $\hat{\delta}(u, s) = \infty$, and so we compute $d_{us} = \infty + 0 - \infty \neq 0$. Since $\delta(u, s) = 0$, we get an incorrect answer.

If the graph G is strongly connected, then we get $h(v) = \delta(s, v) < \infty$ for all vertices $v \in V$. Thus, the triangle inequality says that $h(v) \leq h(u) + w(u, v)$ for all edges $(u, v) \in E$, and so $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$. Moreover, all edge weights $\hat{w}(u, v)$ used in Lemma 25.1 are finite, and so the lemma holds. Therefore, the conditions we need in order to use Johnson's algorithm hold: that reweighting does not change shortest paths, and that all edge weights $\hat{w}(u, v)$ are nonnegative. Again relying on G being strongly connected, we get that $\hat{\delta}(u, v) < \infty$ for all edges $(u, v) \in E$, which means that $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$ is finite and correct.