CMPN302: Algorithms Design and Analysis

Lecture 09: Shortest Paths Algorithms

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2017

Shortest paths algorithms

Application:

- shortest-paths *routing*

Variants:

- Single-source shortest paths problem
  - Single-pair shortest path problem
  - Single-destination shortest paths problem
- All-source shortest paths problem
  - Can be faster than repeating the single-source one for every vertex

Main concept

- Optimal substructure:
  - Shortest-paths algorithms typically rely on the property that a shortest path between two vertices *contains other* shortest paths within it.

- Thus, dynamic programming and greedy approaches can be utilized:
  - Dijkstra’s algorithm (single-source) is a greedy one
  - Floyd-Warshall (all-source) algorithm is a dynamic programming one

Optimal substructure

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Single-source shortest paths

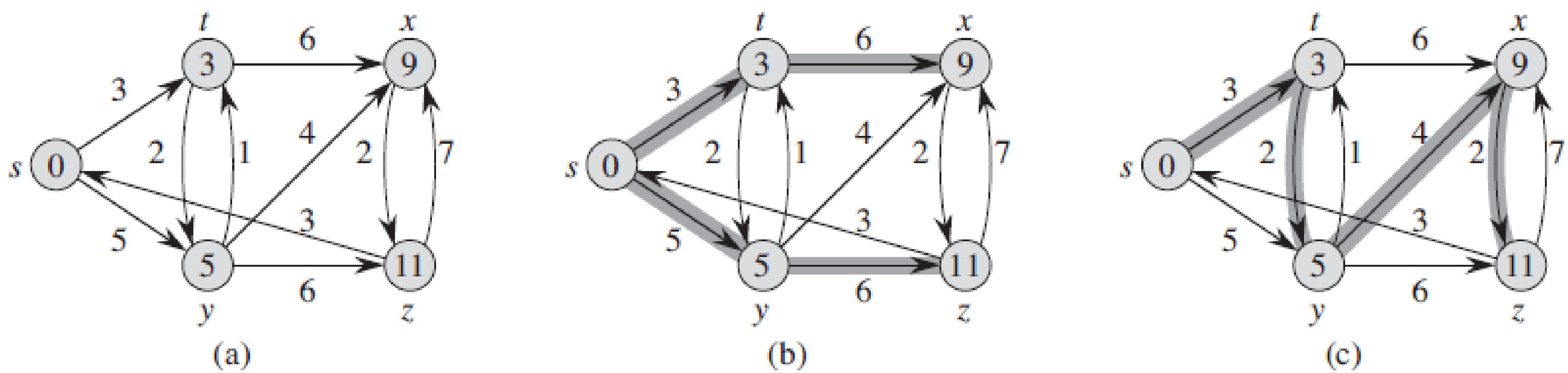


Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Negative edge weights

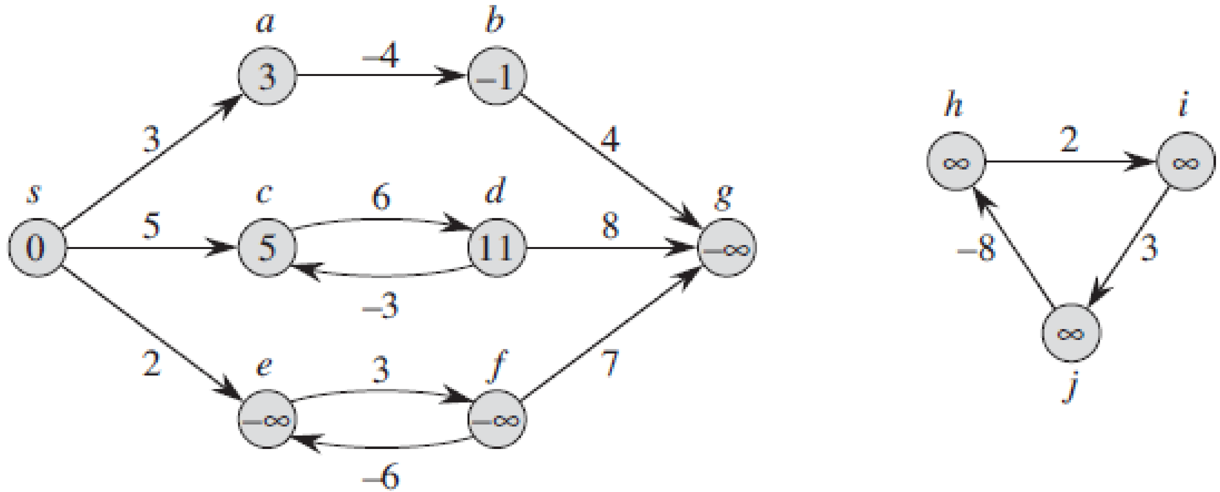


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

- Algorithms differ in how they allow/handle negative edges

Cycles

- Negative-weight cycles:
  - Negative edges prohibited
  - Such cycles are detected by algorithms
- Positive-weight cycles:
  - Theoretically not possible
- Zero-weight cycles:
  - Can be eliminated from the paths

Common subroutines

INITIALIZE-SINGLE-SOURCE(G, s)

```
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

RELAX(u, v, w)

```
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

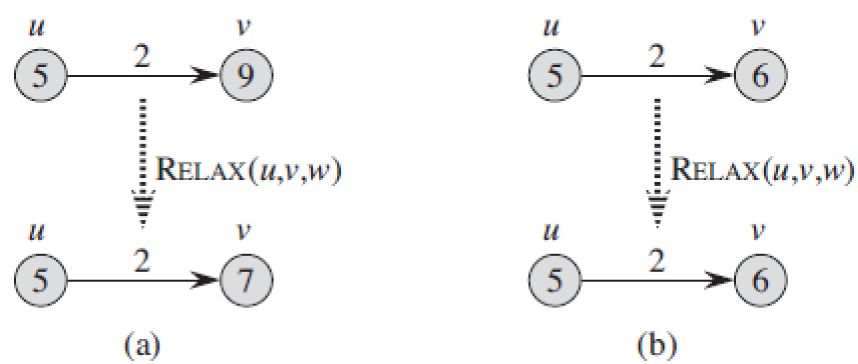


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

Bellman-Ford algorithm

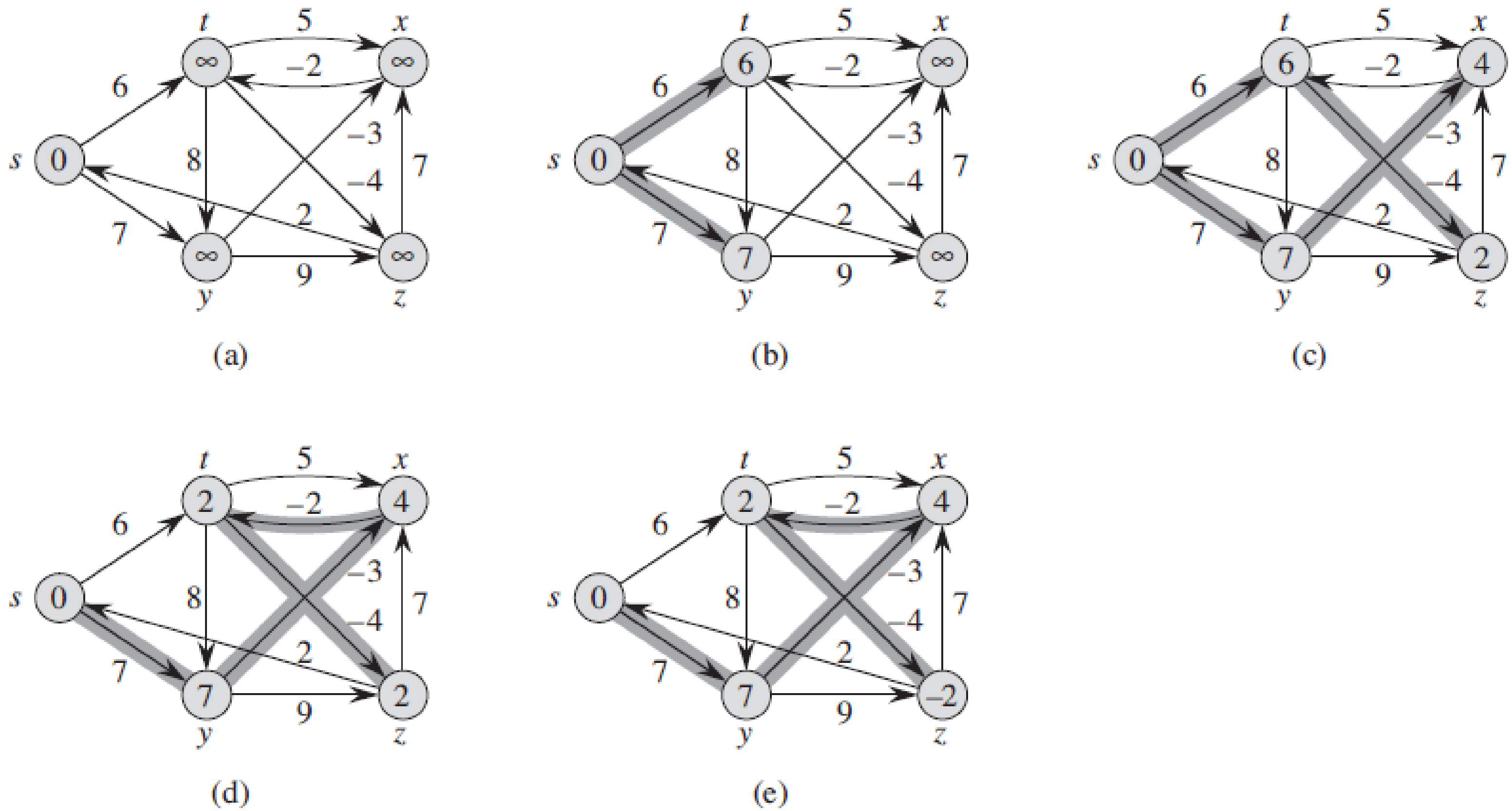


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Bellman-Ford algorithm

BELLMAN-FORD(G, w, s)

```
O(V)    INITIALIZE-SINGLE-SOURCE(G, s)
O(V)    for i = 1 to |G.V| - 1
O(E)        for each edge (u, v) ∈ G.E
O(1)            RELAX(u, v, w)
O(E)    for each edge (u, v) ∈ G.E
            if v.d > u.d + w(u, v)
                return FALSE
        return TRUE
```

- Complexity: $O(VE)$

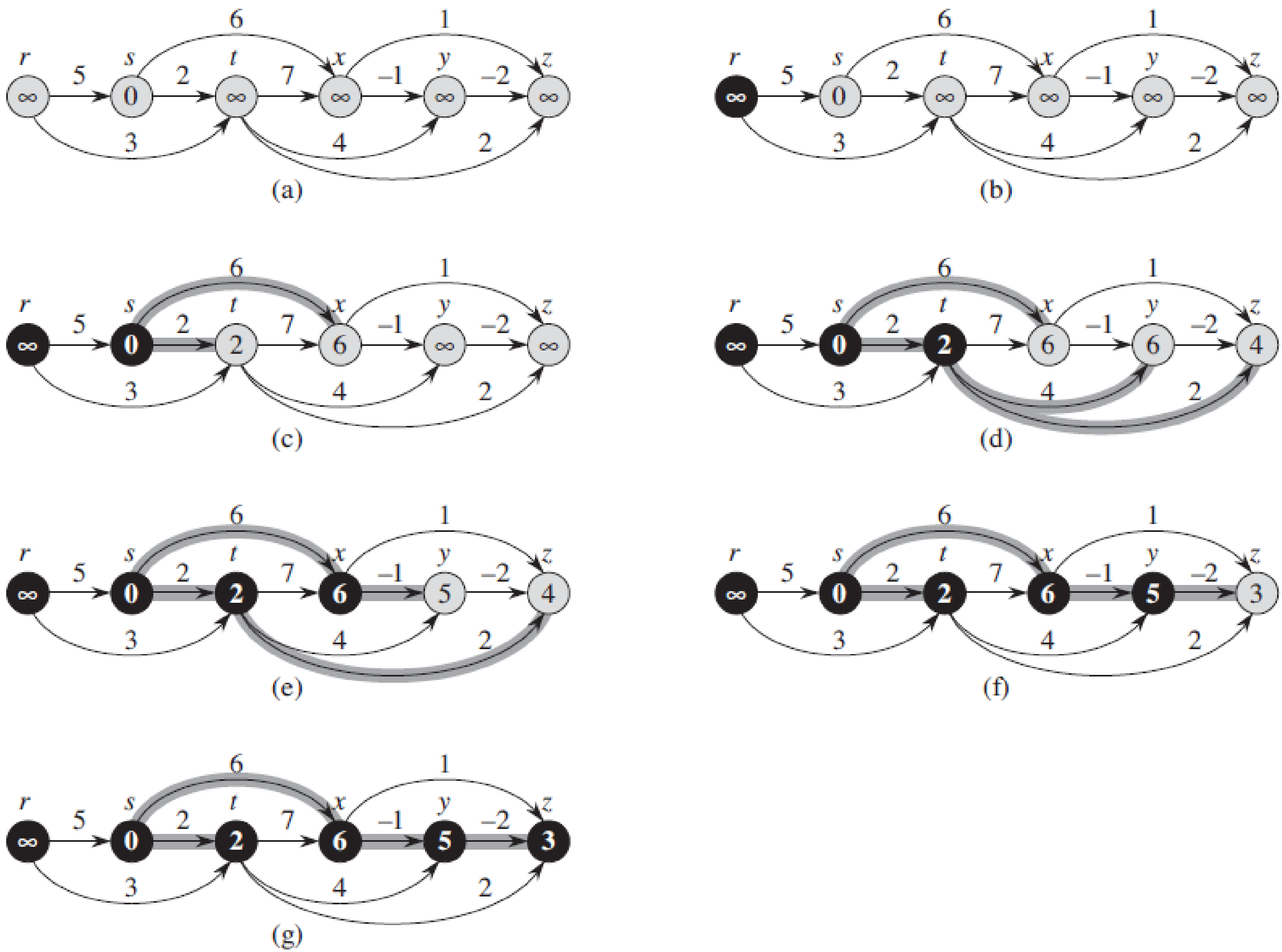
Bellman-Ford algorithm

- Correctness:

Lemma 24.2
Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ∎

DAG shortest-paths algorithm



DAG shortest-paths algorithm

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

- Complexity: $O(V+E)$

Dijkstra’s algorithm

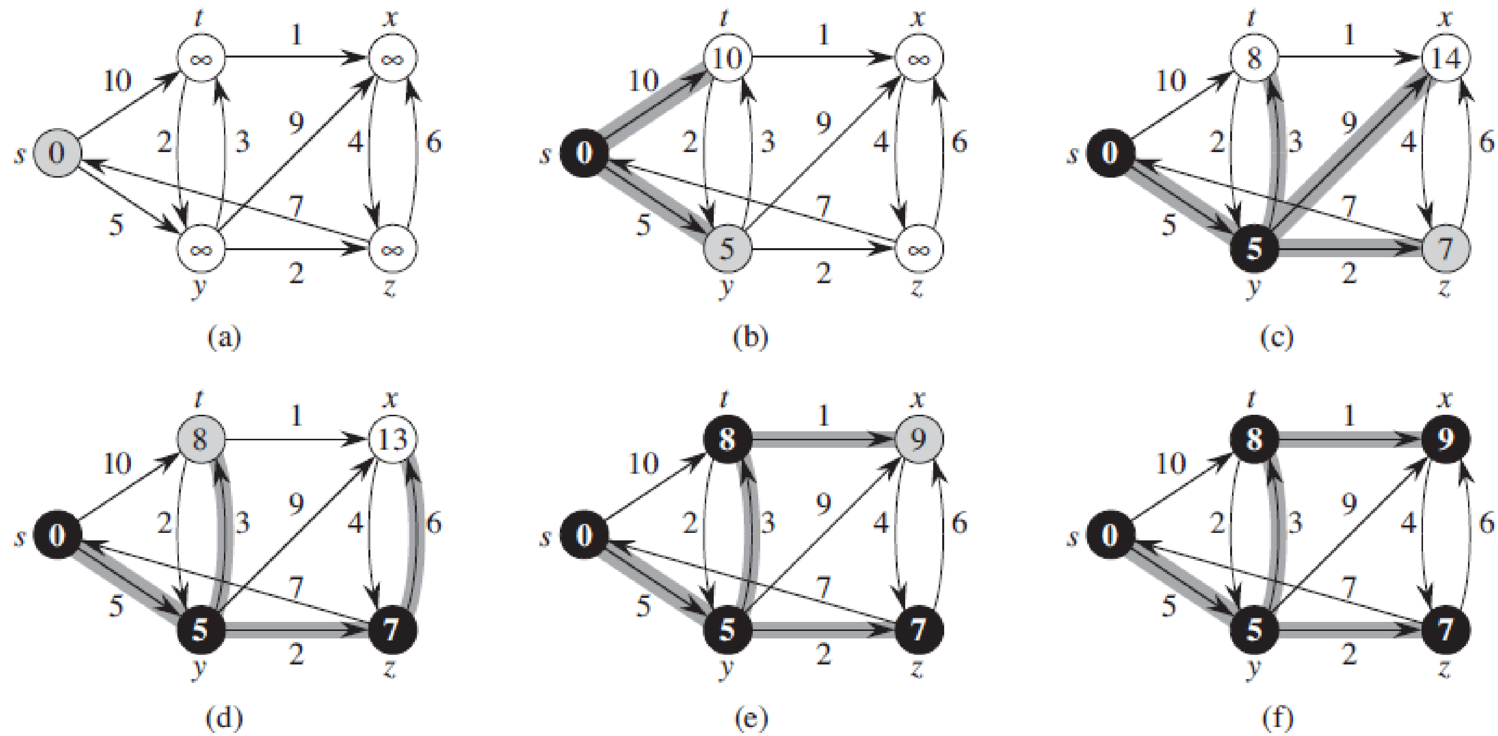


Figure 24.6 The execution of Dijkstra’s algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

Dijkstra's algorithm

```
                   DIJKSTRA(G, w, s)
                       INITIALIZE-SINGLE-SOURCE(G, s)
                       S = ∅
Insert             →   Q = G.V
                       while Q ≠ ∅
Extract – Min      →       u = EXTRACT-MIN(Q)
                           S = S ∪ {u}
                           for each vertex v ∈ G.Adj[u]
Decrease – Key     →           RELAX(u, v, w)
```

- Implement Q using:

$E = o(\frac{V^2}{\log V})$

	Array	Min-heap	Fibonacci-heap
Insert (once/total)	$O(1)/O(V)$	$-/O(V)$	$-/O(V)$
Extract-Min	$O(V)/O(V^2)$	$O(\log V)$ $/O(V\log V)$	$O(\log V)$ $/O(V\log V)$
Decrease-Key	$O(1)/O(E)$	$O(\log V)$ $/O(E\log V)$	$O(1)/O(E)$
Total	$O(V^2+E)$ $=O(V^2)$	$O((V+E)\log V)$ $=O(E\log V)$	$O(V\log V+E)$

All-pairs shortest paths

- Input:
  - Digraph $G(V,E)$, where $V=\{1,2,\ldots,n\}$, with edge-weight function $w: E \rightarrow R$.
- Output:
  - $n \times n$ matrix $W=(wij)$ of shortest-path lengths $\delta(i,j)$ for all $i,j \in V$

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \ . \end{cases}$$

All-pairs shortest paths

- Can simply run single-source method for each $v \in V$:
  - Dijkstra's (non-negative-weight edges):
    - Array: $O(V^3)$
    - Min-heap: $O(VE\log V)$ (is it better than array?)
    - Fibonacci heap: $O(V^2\log V + VE)$
  - Bellman-Ford:
    - $O(V^2E)$, if dense graph becomes $O(V^4)$
  - Can we do better?
  - Why?

Dynamic programming

City 1	City 2	Distance
Aswan	Cairo	800
Cairo	Alex	200
Cairo	Matrouh	600 (assume)
Alex	Matrouh	300

	Aswan	Cairo	Alex	Matrouh
Aswan	0	800	∞	∞
Cairo		0	200	600
Alex			0	300
Matrouh				0

At most 1 edge shortest paths
Weight matrix W

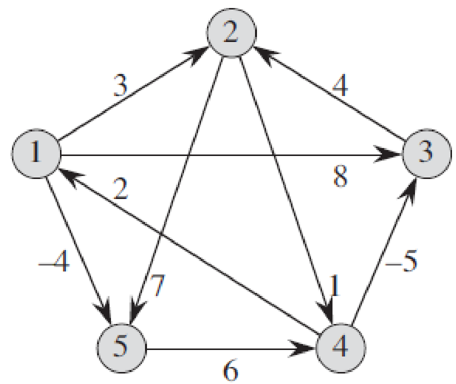
	Aswan	Cairo	Alex	Matrouh
Aswan	0	800	**1000**	1400
Cairo		0	200	**500**
Alex			0	300
Matrouh				0

At most 2 edges shortest paths

	Aswan	Cairo	Alex	Matrouh
Aswan	0	800	1000	**1300**
Cairo		0	200	500
Alex			0	300
Matrouh				0

At most 3 edges shortest paths

Dynamic programming



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

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

Figure 25.1 A directed graph and the sequence of matrices $L^{(m)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS. You might want to verify that $L^{(5)}$, defined as $L^{(4)} \cdot W$, equals $L^{(4)}$, and thus $L^{(m)} = L^{(4)}$ for all $m \geq 4$.

Dynamic programming

- **Structure of shortest path:**
  - Shortest path from i to j is composed of
    - shortest path $i \rightarrow k$
    - + shortest path from $k \rightarrow j$
- **Recursive solution to the problem:**
  - let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges

$$
\begin{aligned}
l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\
&= \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} .
\end{aligned}
$$

Dynamic programming

- **Computing the shortest-path weights bottom up**

EXTEND-SHORTEST-PATHS(L, W)

$n = L.rows$
let $L' = (l'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        $l'_{ij} = \infty$
        for $k = 1$ **to** n
            $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

- Complexity: $\Theta(n^3)$

Dynamic programming

- **Computing the shortest-path weights bottom up**

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
    let $L^{(m)}$ be a new $n \times n$ matrix
    $L^{(m)} =$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
return $L^{(n-1)}$

- Complexity: $\Theta(n^4) = \Theta(V^4)$

Dynamic programming

- Find analogy between:

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \{l_{ik}^{(m-1)} + w_{kj}\}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

$$\begin{aligned} l^{(m-1)} &\to a \,, \\ w &\to b \,, \\ l^{(m)} &\to c \,, \\ \min &\to + \,, \\ + &\to \cdot \end{aligned}$$

- We can rewrite the recursion as

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &= W \,, \\ L^{(2)} &= L^{(1)} \cdot W &= W^2 \,, \\ L^{(3)} &= L^{(2)} \cdot W &= W^3 \,, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1} \,. \end{aligned}$$

- Optimize??

Dynamic programming

- **Faster implementation:**

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
m = 1
while m < n − 1
    let L^(2m) be a new n × n matrix
    L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
    m = 2m
return L^(m)
```

- Complexity: $\Theta(n^3 \log n) = \Theta(V^3 \log V)$

Floyd-Warshall algorithm

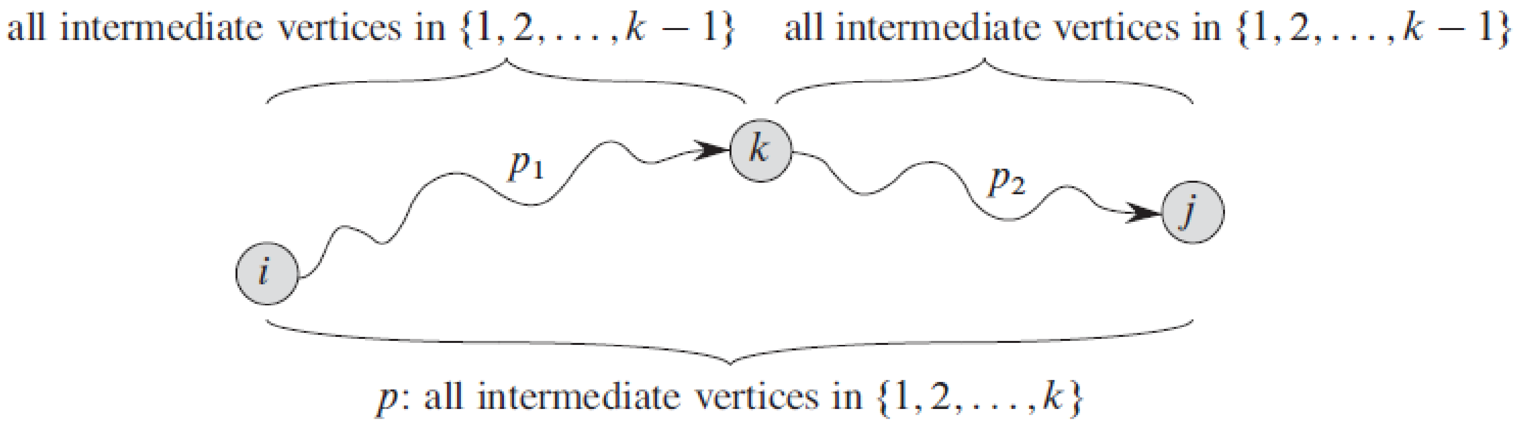


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

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

Floyd-Warshall algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

- Complexity: $\Theta(n^3) = \Theta(V^3)$

Floyd-Warshall algorithm

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

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

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

Floyd-Warshall algorithm

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

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

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

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.

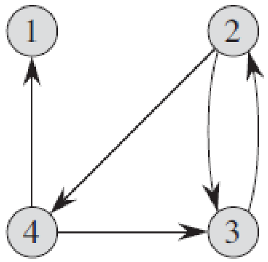
Transitive closure

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

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

- Solution:
  - Assign weight 1 to each edge of E
  - Run Floyd-Warshall algorithm
  - If $d_{ij} < n$, then i and j are connected, otherwise, $d_{ij} = \infty$ and hence are not connected
  - Simplify operations $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$

Transitive closure



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

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

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

Transitive closure

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 $T^{(n)}$

Johnson's algorithm

- For sparse graphs.
- If no negative-weight edges:
  - Apply Dijkstra's algorithm with Fibonacci heaps in $O(V^2 \log V + VE)$
- Else if no negative-weight cycles:
  - Compute a new set of nonnegative edge weights that allows us to use the same method
  - New edge weights $\hat{w}$ must satisfy
    - 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 $\hat{w}$
    - $\hat{w}(u, v)$ is nonnegative

Johnson's algorithm

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 $\widehat{w}$.

Johnson’s algorithm

Proof We start by showing that

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

We have

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

Johnson's algorithm

```
JOHNSON(G, w)
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
if BELLMAN-FORD(G′, w, s) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex v ∈ G′.V
        set h(v) to the value of δ(s, v)
            computed by the Bellman-Ford algorithm
    for each edge (u, v) ∈ G′.E
        ŵ(u, v) = w(u, v) + h(u) − h(v)
    let D = (d_uv) be a new n × n matrix
    for each vertex u ∈ G.V
        run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
        for each vertex v ∈ G.V
            d_uv = δ̂(u, v) + h(v) − h(u)
    return D
```

Johnson’s algorithm