Design and Analysis of Algorithms

Lecture 08: Shortest Paths Algorithms

Ahmed Hamdy

Agenda

- Single-source shortest paths
  - Bellman-Ford algorithm
  - DAG shortest path
  - Dijkstra's algorithm

- All-pair shortest paths
  - Dynamic programming algorithm
  - Floyd-Warshall algorithm
  - Transitive closure
  - Johnson's algorithm

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 (DP) 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

Single-source shortest paths

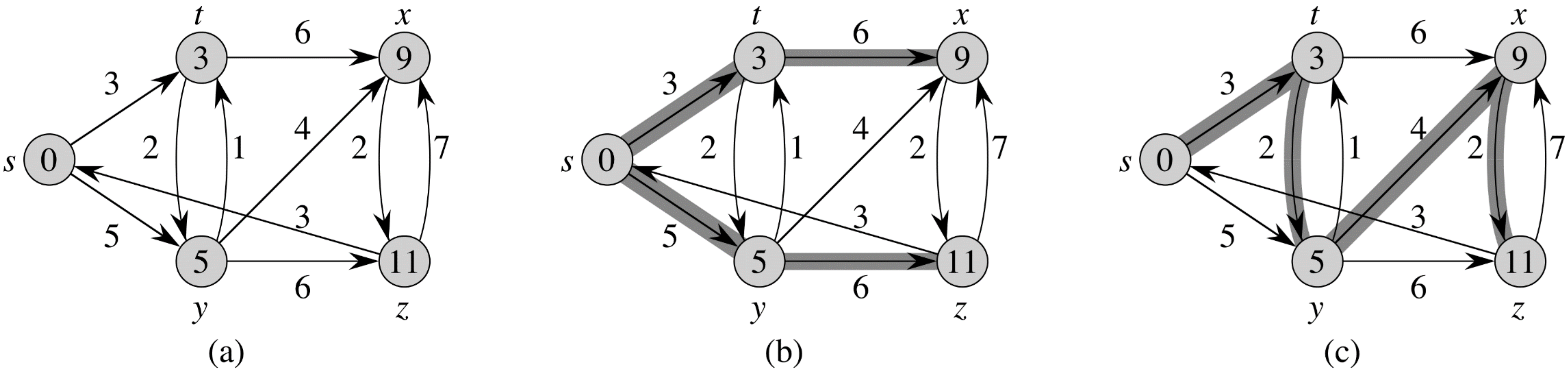


Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Negative edge weights

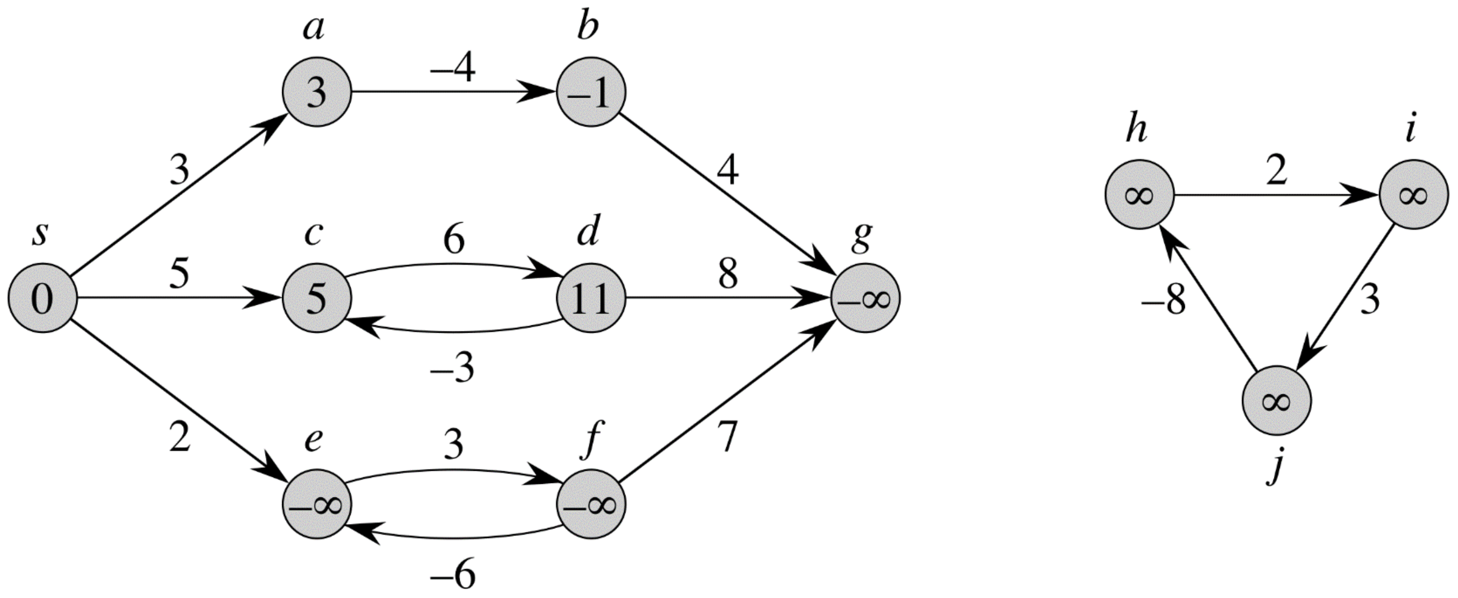


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path 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:
  - Either negative edges are prohibited in some algorithms
  - Or, such cycles are detected by other 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
```

Common subroutines

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

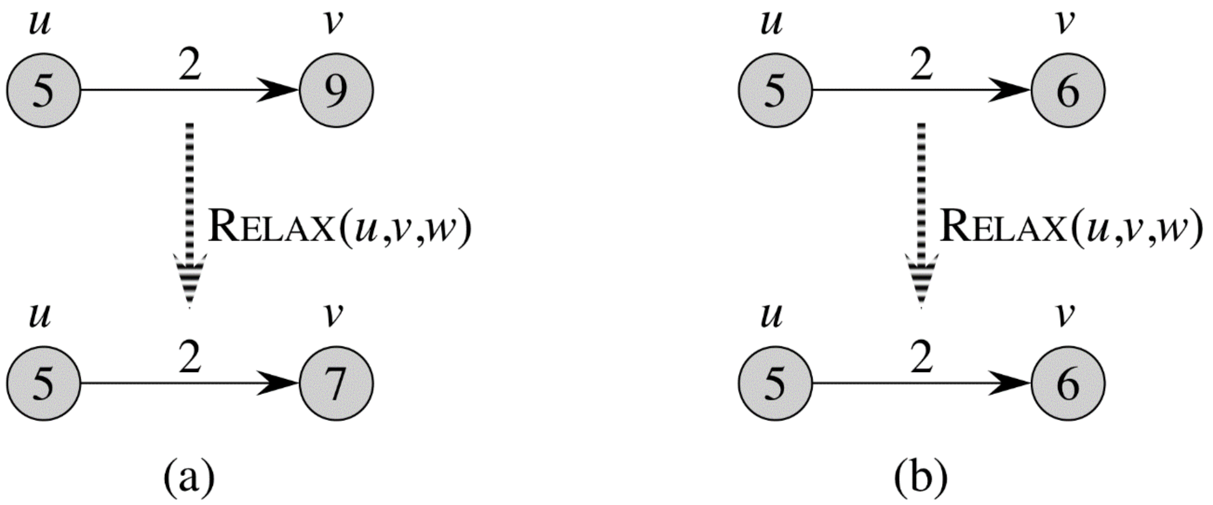


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation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

Bellman-Ford algorithm

- Any shortest path, at the end, is a sequence of edges between any pair (src, dest).
- To discover the shortest path, we need to start from the source and traverse edge by edge on the path to the destination and then relax the edges on this path. The distance to the destination should be the lowest compared to other paths.
- We can not guarantee to traverse the edges on the SP in the right order.
- Instead, traverse all edges for as many iterations as the longest possible path in the graph.

Bellman-Ford algorithm

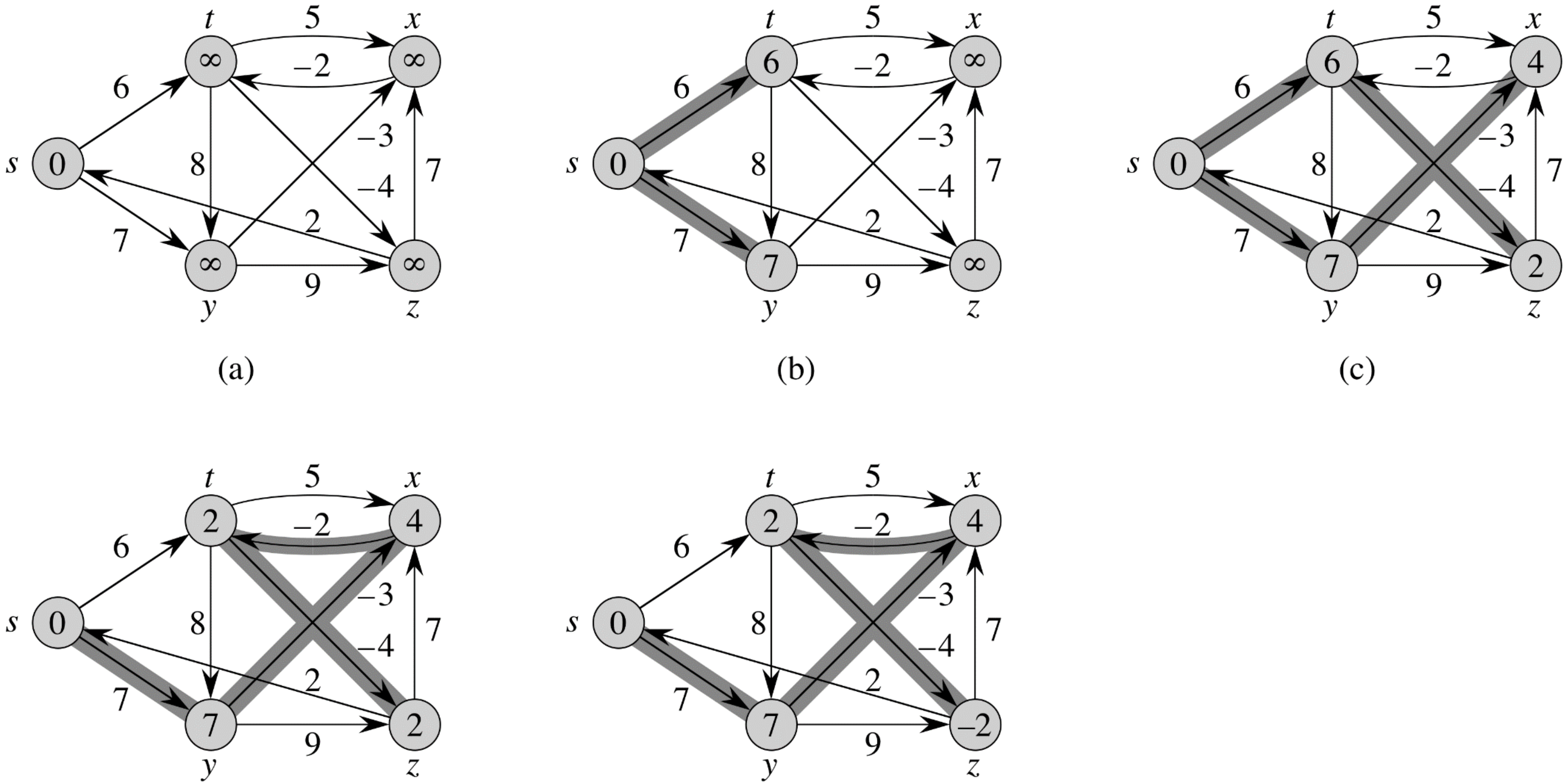


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(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
```

Relaxation

Check for negative cycles

- Complexity: $O(VE)$

Bellman-Ford algorithm

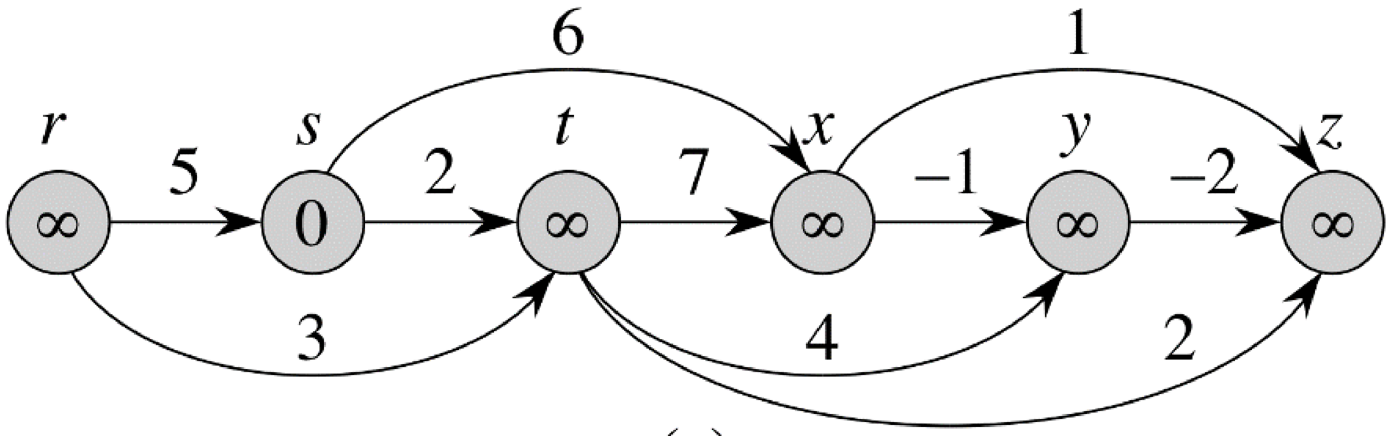
- Correctness:

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, 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 $k \le |V| - 1$. 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, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

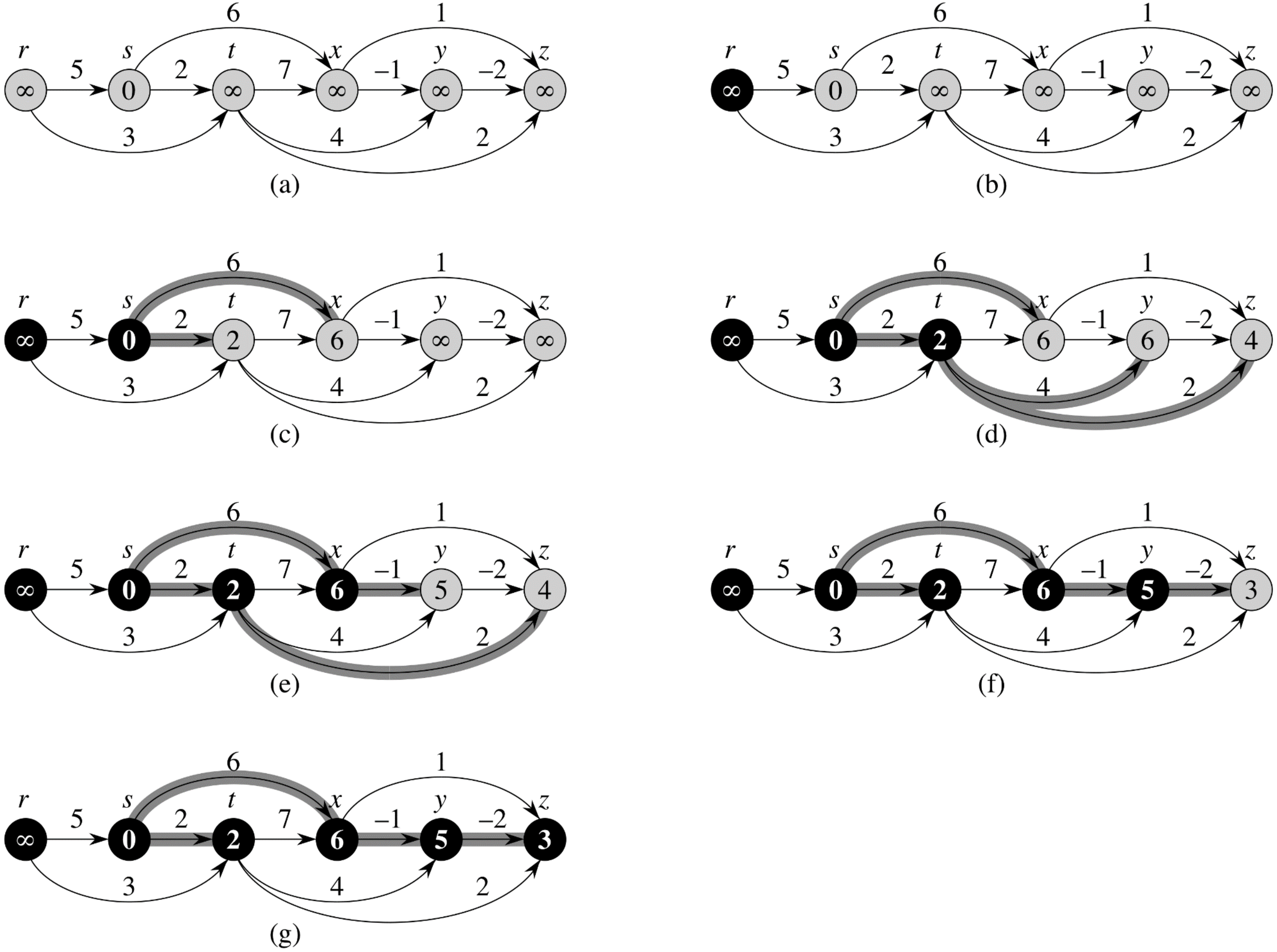
DAG shortest-paths algorithm



(a)

- DAG: Directed Acyclic Graph
- Being acyclic means we can topologically sort them
- Being acyclic means a node won’t be ***revisited*** by any back edge
- Being acyclic means even *negative edges* can’t form negative cycles

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

- Works only with positive edges.
- Greedily picks the node with smallest distance from source and relaxes its edges.
- Why this greedy choice is optimal? In other words, why doesn’t the node with smallest distance gets smaller, thus affecting relaxed values?
- It is kind of similar to BFS (finds shortest path too) but in the non-decreasing order of node distance from root.
- BFS can be seen as a special case when all edges have the same weight, then the order of nodes is the same as their order in the Q.

Dijkstra's algorithm

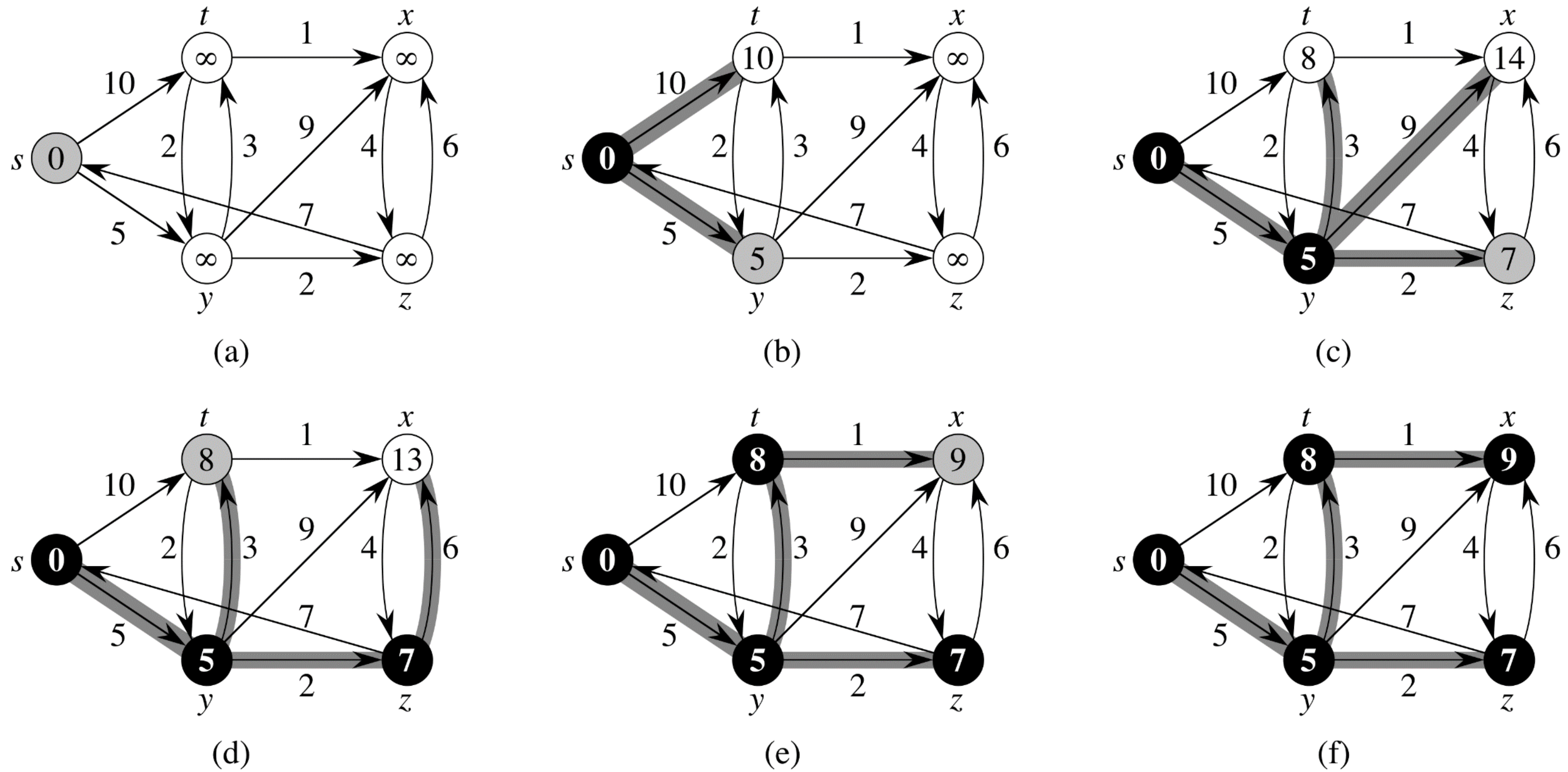


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation 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 = Ø
Insert           →  Q = G.V
                    while Q ≠ Ø
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: *Why unsorted??* $E = o\left(\frac{V^2}{\log V}\right)$

	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	$O(V^2 + E)$ $= O(V^2)$	$O\big((V + E)\log V\big)$ $= O(E \log V)$	$O(V \log V + E)$

Single-source shortest paths: Summary

	Supports Cycles	Negative Edges	Complexity
Bellman-Ford	Yes	Yes	$O(VE)$
DAG	No	Yes	$O(V+E)$
Dijkstra	Yes	No	$O(V \log V + E)$

All-Pair Shortest Paths

All-pairs shortest paths

- Input:
  - Digraph $G(V, E)$, where $V = \{1,2, \dots, n\}$, with edge-weight function $w: E \to R$.
- Output:
  - $n \times n$ matrix $W = (w_{ij})$ 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 $v \in V$, thus complexity gets multiplied by V:
  - Dijkstra's (non-negative-weight edges):
    - Array: $O(V^3)$
    - Min-heap: $O(VE \log V)$
    - 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

- Thinking in bottom-up approach, we would start by computing SP with length one between every pair.

- Then compute SP with length **at most** two between every pair based on previous SP with length one, and so on.

Dynamic programming

- **Structure of shortest path:**
  - Shortest path from i to j is composed of

    shortest path $i \to k$ **+** shortest path from $k \to j$

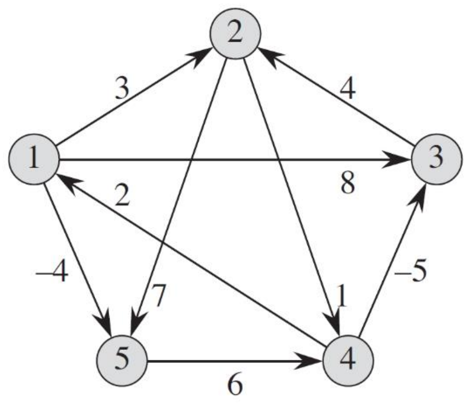
- **Recursive solution to the problem:**
  - Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges

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

Dynamic programming

SP between 3 and 5 is $3 \to 2 \to 4 \to 1 \to 5$:

- $L^{(2)}$ will connect one+one edges ($3 \to 2$ + $2 \to 4$) together to form SP between 3 and 4 as $3 \to 2 \to 4$
- $L^{(3)}$ will connect two+one edges ($3 \to 2 \to 4$ + $4 \to 1$) together to form SP between 3 and 4 as $3 \to 2 \to 4 \to 1$
- $L^{(4)}$ will connect two+one edges ($3 \to 2 \to 4 \to 1$ + $1 \to 5$) together to form SP between 3 and 4 as $3 \to 2 \to 4 \to 1 \to 5$



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

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

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

- **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
    $L^{(m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W)$
return $L^{(n-1)}$

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

Dynamic programming

- Analogy between:

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

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

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

- 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

- FW uses the same approach but on *vertices* rather than *edges*.

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

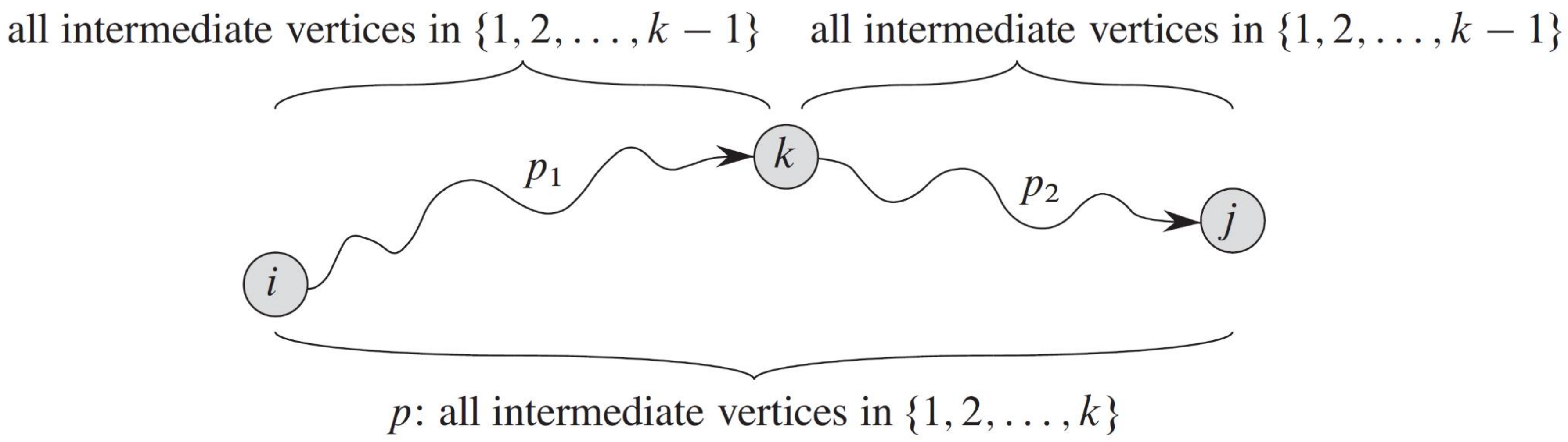
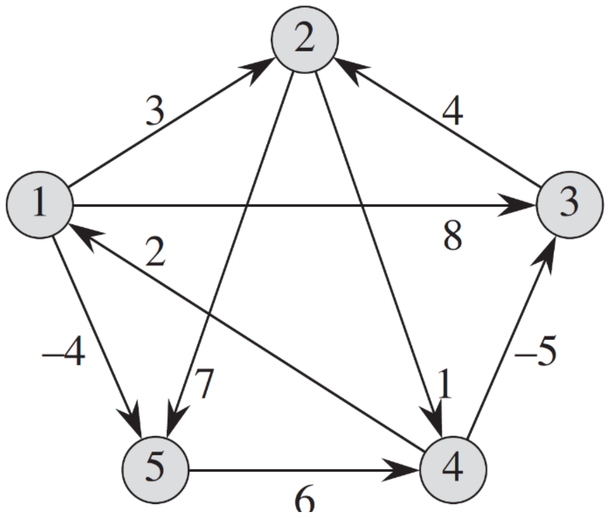


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

Floyd-Warshall algorithm



SP between 3 and 5 is $3 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 5$:

- During first iteration, node 1 will enhance 4 to 5 ($4 \rightarrow 1 \rightarrow 5$).
- During second iteration, node 2 will enhance 3 to 4 ($3 \rightarrow 2 \rightarrow 4$)
- During fourth iteration, node 4 will enhance 3 to 5 ($3 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 5$)

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

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

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

Floyd-Warshall algorithm

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

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

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

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.

To track the best route (3 → 2 → 4 → 1 → 5):

- Check (3 → 5) which indicates 1 is the parent of 5.
- Check (3 → 1) which indicates 4 is the parent of 1.
- Check (3 → 4) which indicates 2 is the parent of 4.
- Check (3 → 2) which indicates 3 is the parent of 2 which completes the path.

Floyd-Warshall algorithm

FLOYD-WARSHALL(W)

Try every node as intermediate node

Try node k as an intermediate node between every pair (i, j)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = (d_{ij}^{(k)})$ 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: $\Theta(n^3) = \Theta(V^3)$

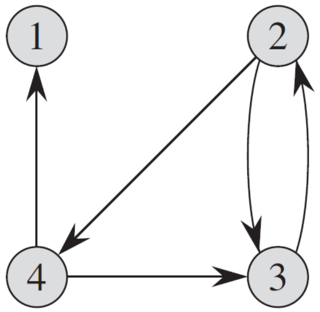
Transitive closure

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

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.

- Simplify operations $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$

Transitive closure

TRANSITIVE-CLOSURE(G)

$n = |G.V|$
let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix

Initialize weight matrix with ones in place of edges

for $i = 1$ **to** n
  for $j = 1$ **to** n
    if $i == j$ or $(i, j) \in G.E$
      $t_{ij}^{(0)} = 1$
    else $t_{ij}^{(0)} = 0$

Floyd-Warshall

for $k = 1$ **to** n
  let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

return $T^{(n)}$

Johnson's algorithm

For sparse graphs ($E \ll V^2$):

- If no negative-weight edges:
  - Apply Dijkstra's algorithm with Fibonacci heaps in $O(V^2 \log V + VE)$ (better than FW)
- 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 $\widehat{w}$ 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 $\widehat{w}$
    - $\widehat{w}(u, v)$ 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

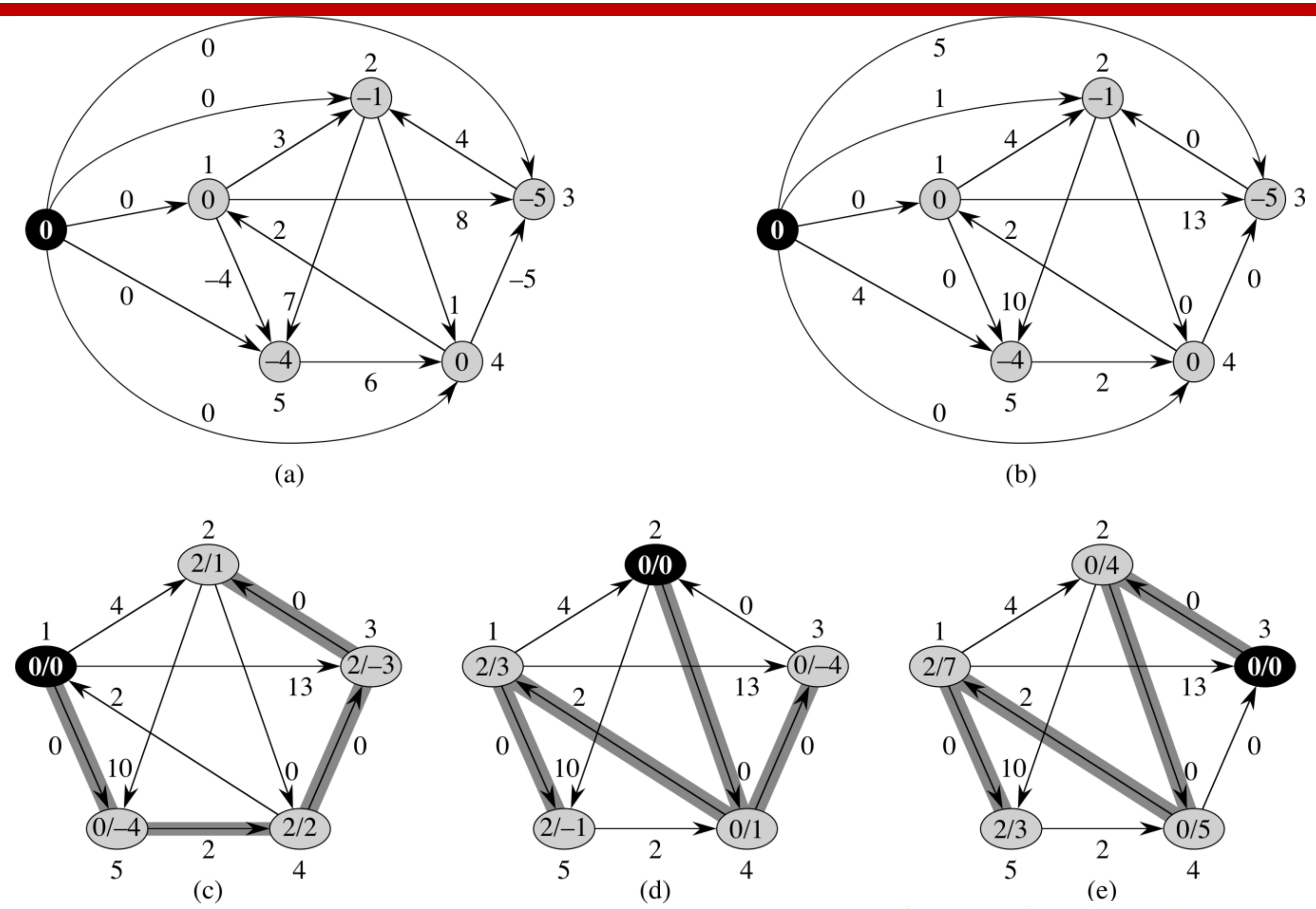
Proof We start by showing that

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

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



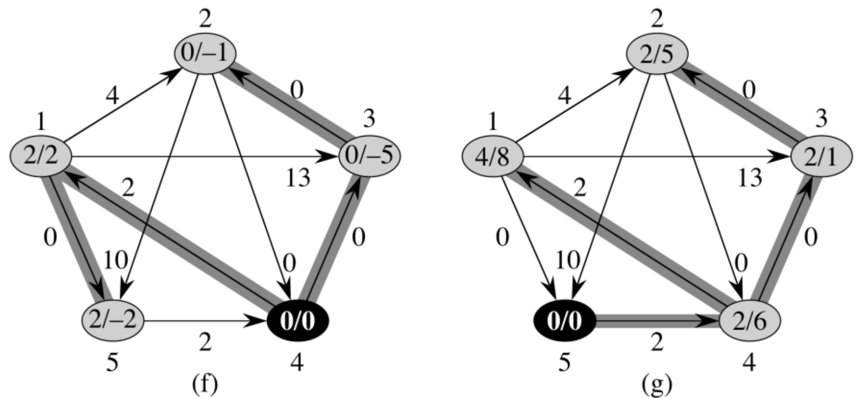


Figure 25.6 Johnson’s all-pairs shortest-paths algorithm run on the graph of Figure 25.1. Vertex numbers appear outside the vertices. **(a)** The graph G' with the original weight function w. The new vertex s is black. Within each vertex v is $h(v) = \delta(s, v)$. **(b)** After reweighting each edge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

Johnson's algorithm

JOHNSON(G, w)

compute G', where $G'.V = G.V \cup \{s\}$,
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
    $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD(G', w, s) == FALSE
    print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
        set $h(v)$ to the value of $\delta(s, v)$
            computed by the Bellman-Ford algorithm
    for each edge $(u, v) \in G'.E$
        $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
    let $D = (d_{uv})$ be a new $n \times n$ matrix
    for each vertex $u \in G.V$
        run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
        for each vertex $v \in G.V$
            $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
    return D