

CMP(N)302: Design and Analysis of Algorithms



Lecture 08: Shortest Paths Algorithms

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2019

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 $w : E \rightarrow \mathbb{R}$, let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ 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 $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, 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, $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ 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

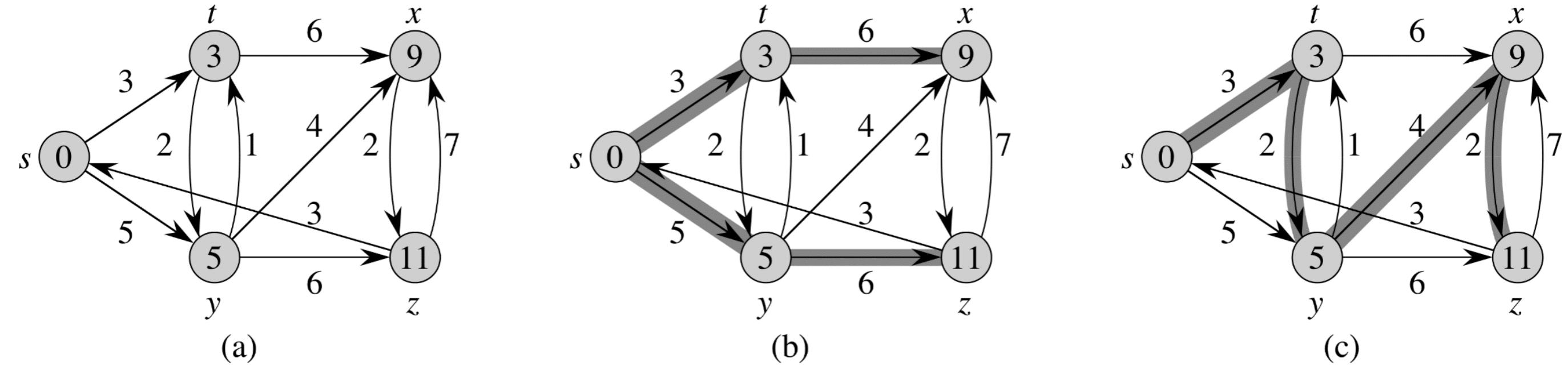


Figure 24.2 (a) A weighted, directed graph with shortest-path weights from source s . (b) The shaded edges form a shortest-paths tree rooted at the source s . (c) Another shortest-paths tree with the same root.

Negative edge weights

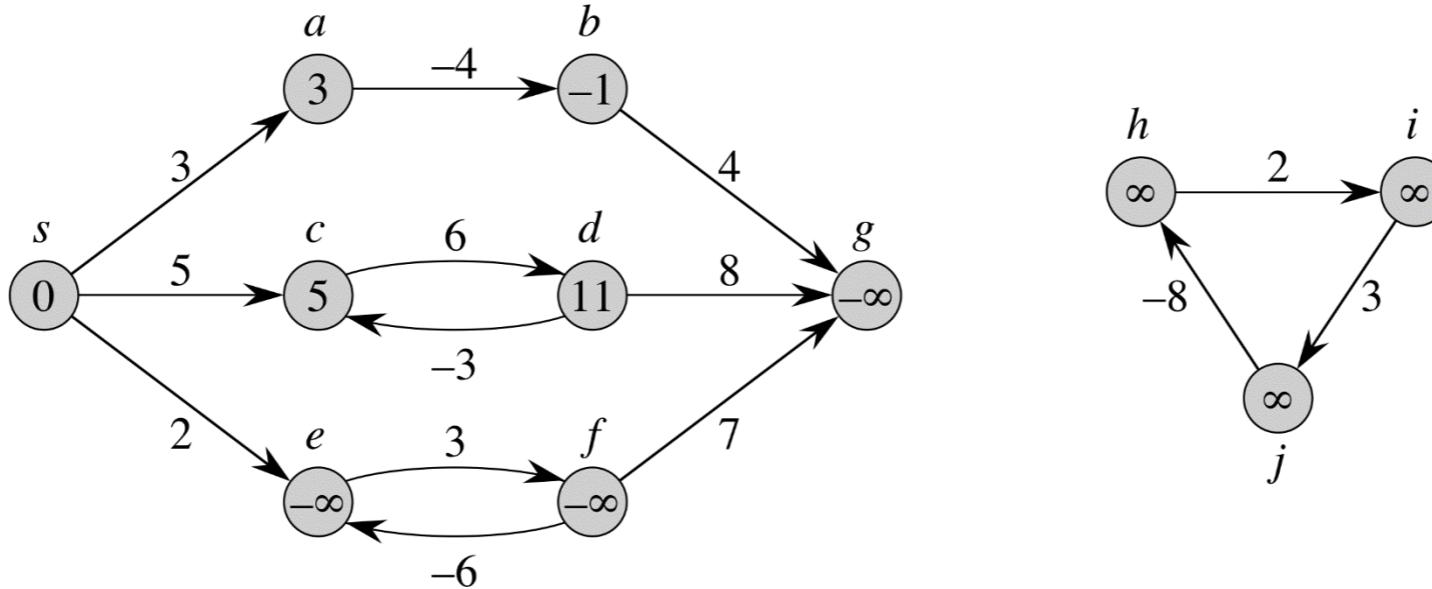


Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s , they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path 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

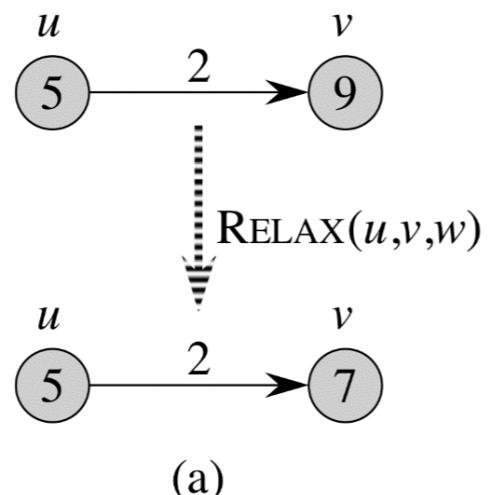
Cycles

- Negative-weight cycles:
 - 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)

```
1  for each vertex  $v \in G.V$ 
2       $v.d = \infty$ 
3       $v.\pi = \text{NIL}$ 
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       $v.\pi = u$ 
```

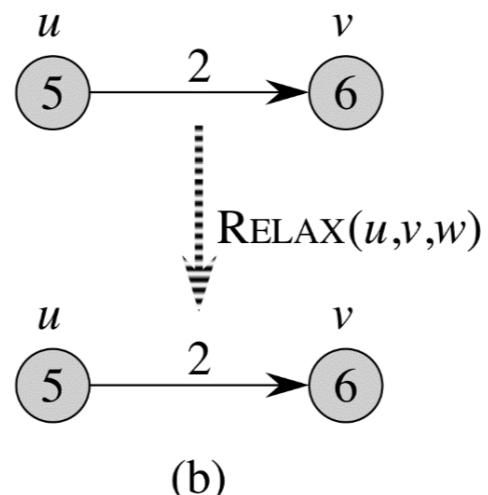


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. (a) Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. (b) Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step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

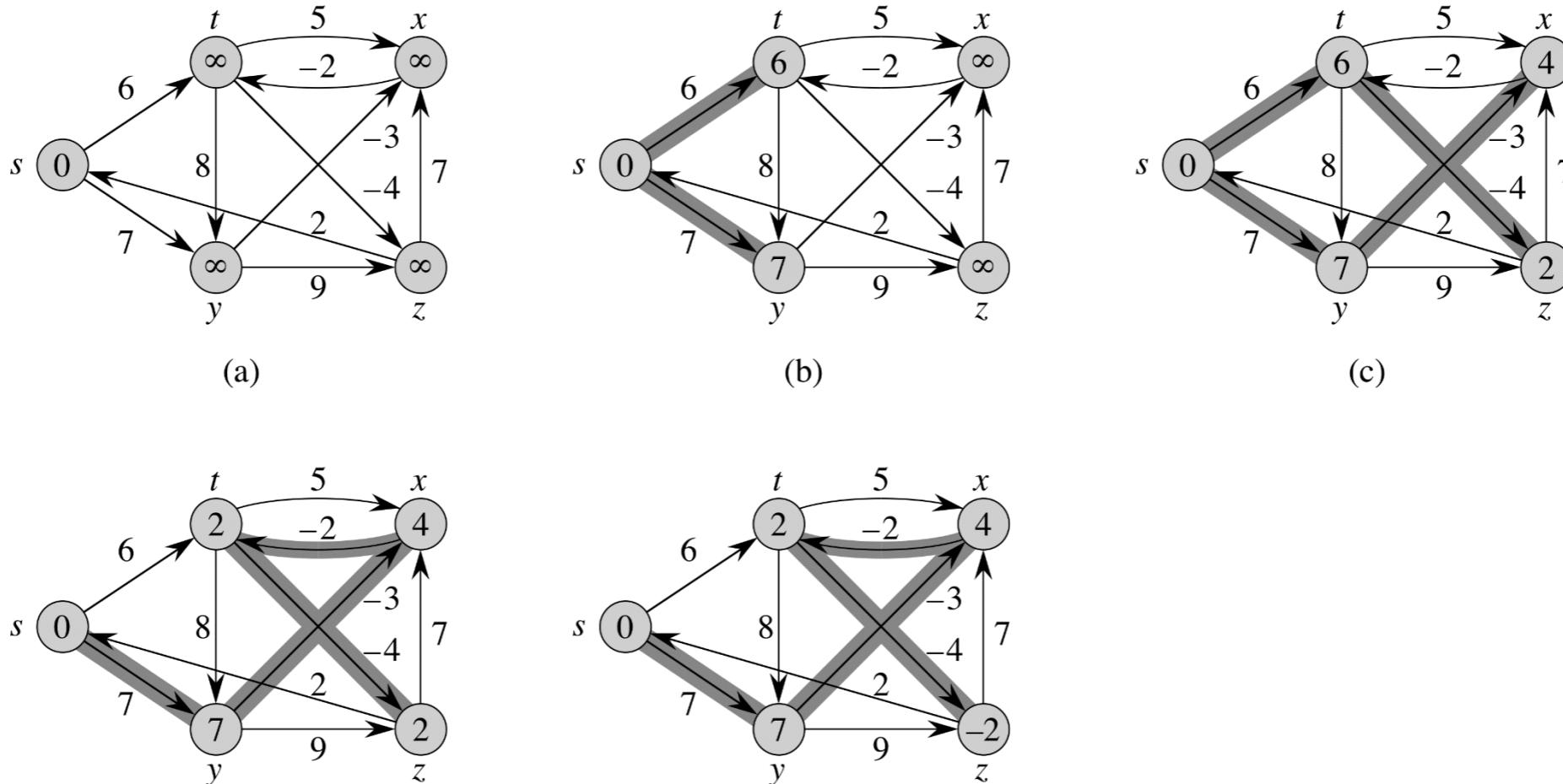


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s . The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t,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

$O(V) \rightarrow 1$ INITIALIZE-SINGLE-SOURCE(G, s)

$O(V) \rightarrow 2$ **for** $i = 1$ **to** $|G.V| - 1$

$O(E) \rightarrow 3$ **for** each edge $(u, v) \in G.E$

$O(1) \rightarrow 4$ RELAX(u, v, w)

$O(E) \rightarrow 5$ **for** each edge $(u, v) \in G.E$

6 **if** $v.d > u.d + w(u, v)$

7 **return** FALSE

8 **return** TRUE

Relaxation

Check for negative cycles

- Complexity: $O(VE)$

Bellman-Ford algorithm

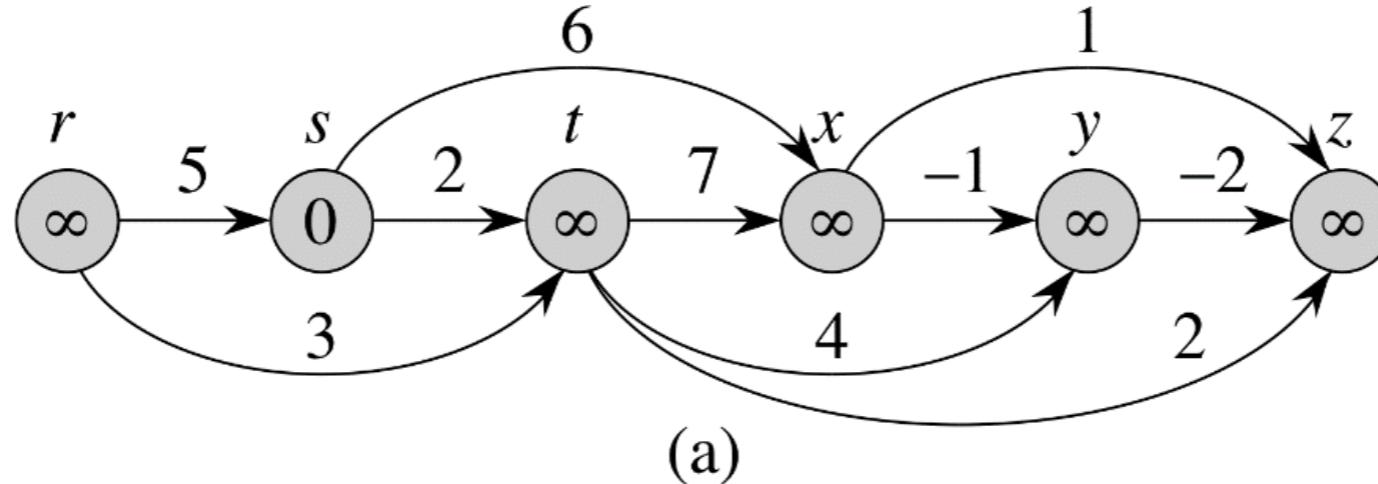
- Correctness:

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow \mathbb{R}$, 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 .

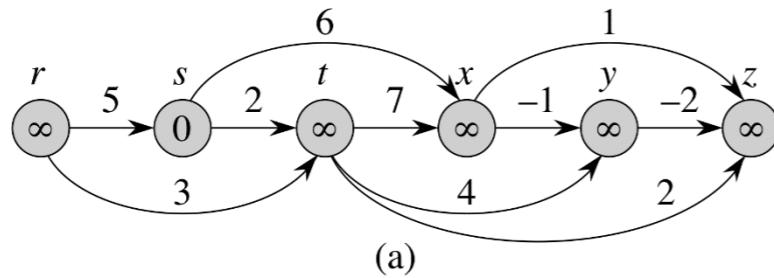
Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s , and let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v . Because shortest paths are simple, p has at most $|V| - 1$ edges, 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 i th iteration, for $i = 1, 2, \dots, k$, 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

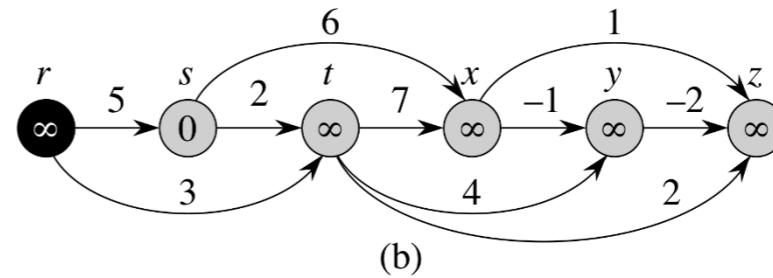


- DAG: **Directed Acyclic Graph**
- Being acyclic means we can topologically sort them
- Being acyclic means a node won't be **revisited** by any **back edge**

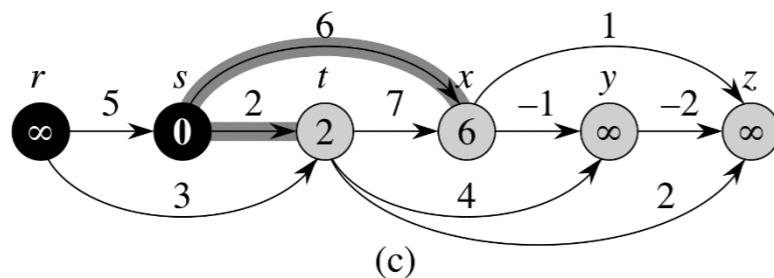
DAG shortest-paths algorithm



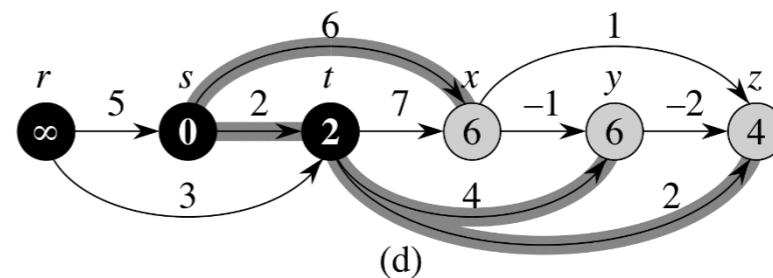
(a)



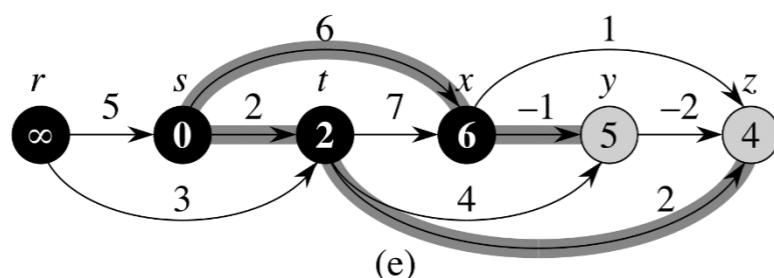
(b)



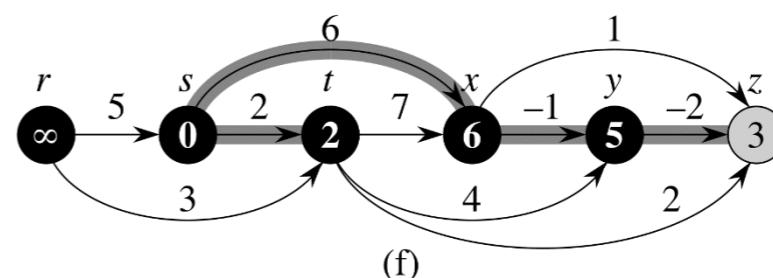
(c)



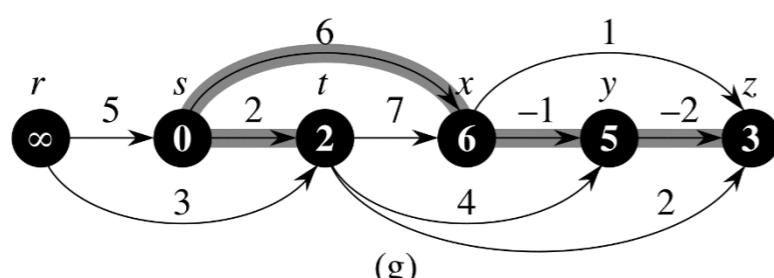
(d)



(e)



(f)



(g)

DAG shortest-paths algorithm

DAG-SHORTEST-PATHS(G, w, s)

- $O(V + E) \rightarrow 1$ topologically sort the vertices of G
- $O(V) \rightarrow 2$ INITIALIZE-SINGLE-SOURCE(G, s)
- $O(E) \rightarrow 3$ **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?

Dijkstra's algorithm

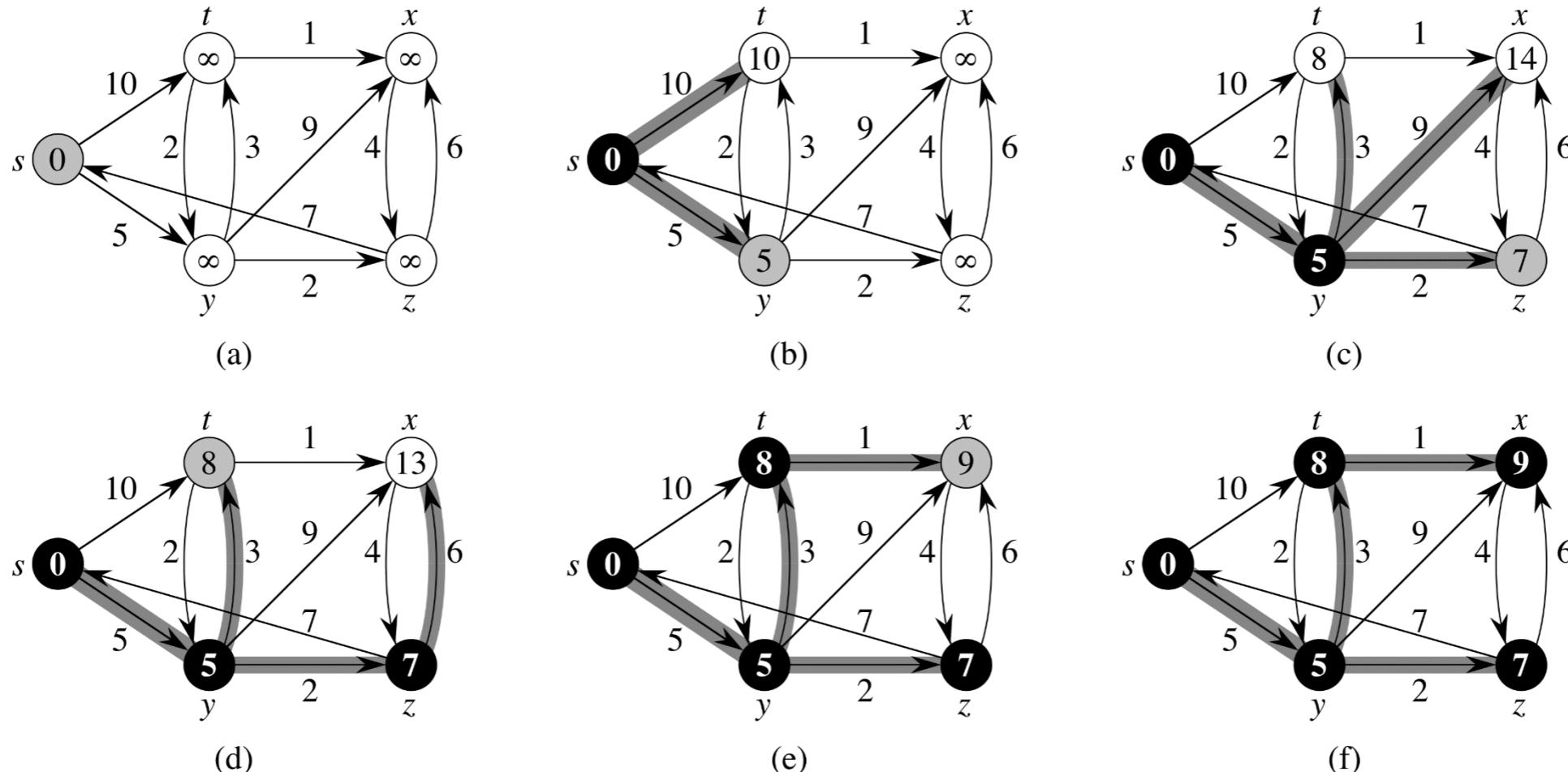


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S , and white vertices are in the min-priority queue $Q = V - S$. (a) The situation 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)

```

1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
Insert   → 3   $Q = G.V$ 
             4  while  $Q \neq \emptyset$ 
Extract – Min → 5     $u = \text{EXTRACT-MIN}(Q)$ 
                     6     $S = S \cup \{u\}$ 
                     7    for each vertex  $v \in G.\text{Adj}[u]$ 
Decrease – Key → 8    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	$O(V^2 + E)$ $= O(V^2)$	$O((V + E) \log V)$ $= O(E \log V)$	$O(V \log V + E)$

All-pairs shortest paths

- Input:
 - Digraph $G(V, E)$, where $V = \{1, 2, \dots, n\}$, with edge-weight function $w: E \rightarrow R$.
- 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 $v \in V$:
 - 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

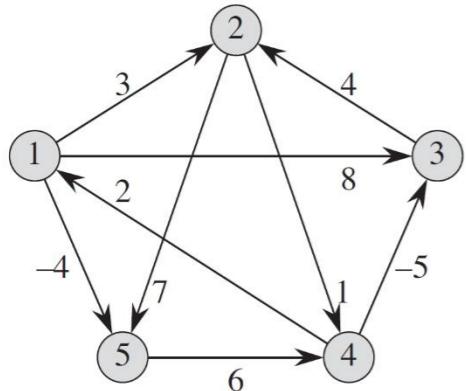
- Thinking in bottom-up approach, we would start by computing SP with length one between every pair.
- Then compute SP with length **at most** two between every pair based on previous SP with length one, and so on.

Dynamic programming

- **Structure of shortest path:**
 - Shortest path from i to j is composed of
shortest path $i \rightarrow k$ + 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

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

Dynamic programming



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

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

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

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

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

- Computing the shortest-path weights bottom up

EXTEND-SHORTEST-PATHS(L, W)

```
1   $n = L.\text{rows}$ 
2  let  $L' = (l'_{ij})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $l'_{ij} = \infty$ 
6          for  $k = 1$  to  $n$ 
7               $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$ 
8  return  $L'$ 
```

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

Dynamic programming

- Computing the shortest-path weights bottom up

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

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

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

Dynamic programming

- Find analogy between:

$$l_{ij}^{(m)} = \min \left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \right)$$
$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

$l^{(m-1)} \rightarrow a ,$
 $w \rightarrow b ,$
 $l^{(m)} \rightarrow c ,$
 $\min \rightarrow + ,$
 $+ \rightarrow \cdot$

- 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)

```
1   $n = W.\text{rows}$ 
2   $L^{(1)} = W$ 
3   $m = 1$ 
4  while  $m < n - 1$ 
5      let  $L^{(2m)}$  be a new  $n \times n$  matrix
6       $L^{(2m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m)}, L^{(m)})$ 
7       $m = 2m$ 
8  return  $L^{(m)}$ 
```

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

Floyd-Warshall algorithm

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

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

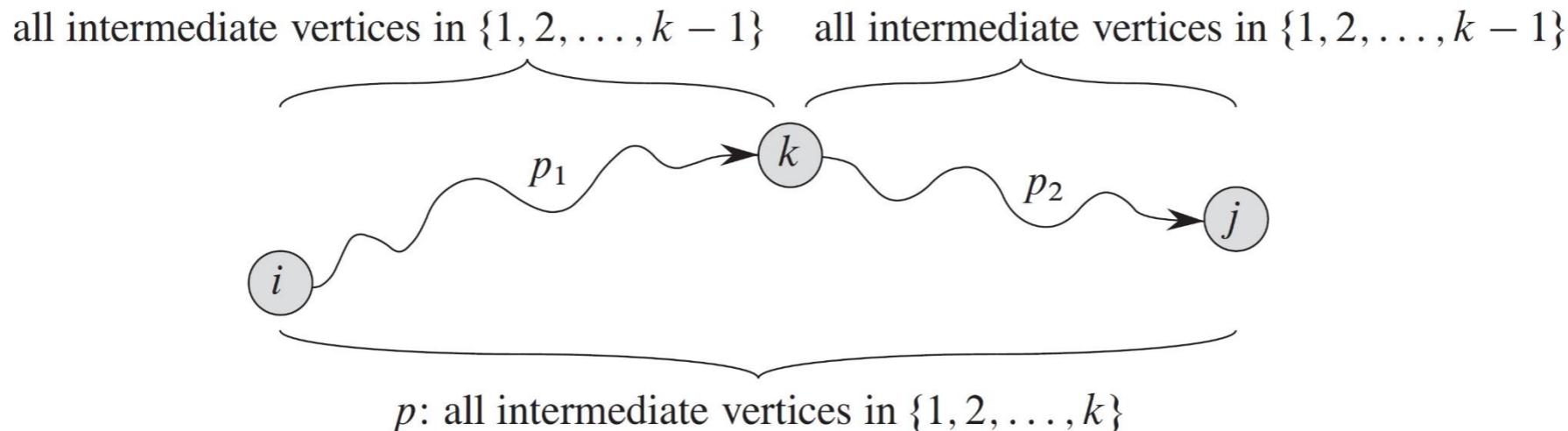
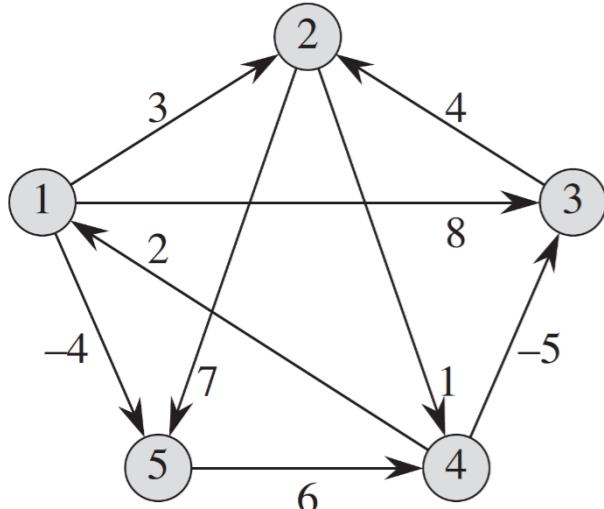


Figure 25.3 Path p is a shortest path from vertex i to vertex j , and k is the highest-numbered intermediate vertex of p . Path p_1 , the portion of path p from vertex i to vertex k , has all intermediate vertices in the set $\{1, 2, \dots, k-1\}$. The same holds for path p_2 from vertex k to vertex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

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

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

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

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.

Floyd-Warshall algorithm

FLOYD-WARSHALL(W)

- 1 $n = W.\text{rows}$
- 2 $D^{(0)} = W$
- 3 **for** $k = 1$ **to** n
 → 4 let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
 5 **for** $i = 1$ **to** n
 6 **for** $j = 1$ **to** n
 7 $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
8 **return** $D^{(n)}$

Try every node as intermediate node

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

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

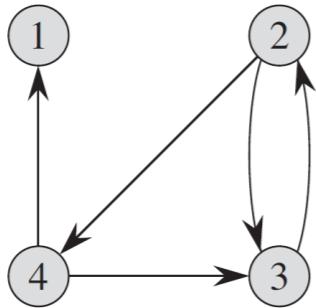
Transitive closure

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \dots, n\}$, 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 $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

Transitive closure

TRANSITIVE-CLOSURE(G)

Initialize weight
matrix with ones in
place of edges

Floyd-Warshall

```
1   $n = |G.V|$ 
2  let  $T^{(0)} = (t_{ij}^{(0)})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5          if  $i == j$  or  $(i, j) \in G.E$ 
6               $t_{ij}^{(0)} = 1$ 
7          else  $t_{ij}^{(0)} = 0$ 
8  for  $k = 1$  to  $n$ 
9      let  $T^{(k)} = (t_{ij}^{(k)})$  be a new  $n \times n$  matrix
10     for  $i = 1$  to  $n$ 
11         for  $j = 1$  to  $n$ 
12              $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
13 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 $O(V^2 \log V + VE)$
- 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 $w : E \rightarrow \mathbb{R}$, let $h : V \rightarrow \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

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

Let $p = \langle v_0, v_1, \dots, v_k \rangle$ 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 \hat{w} . That is, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function \hat{w} .

Johnson's algorithm

Proof We start by showing that

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

We have

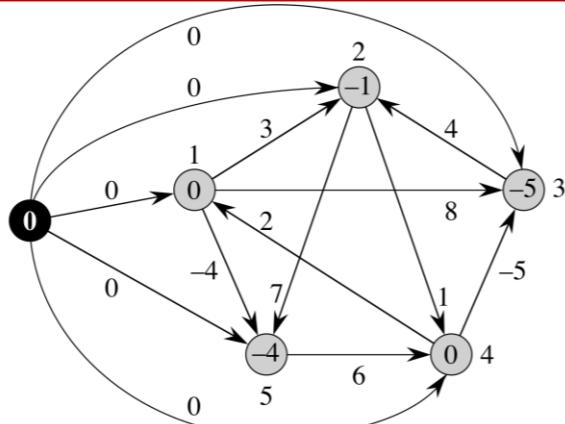
$$\begin{aligned}\hat{w}(p) &= \sum_{i=1}^k \hat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \quad (\text{because the sum telescopes}) \\ &= w(p) + h(v_0) - h(v_k).\end{aligned}$$

Johnson's algorithm

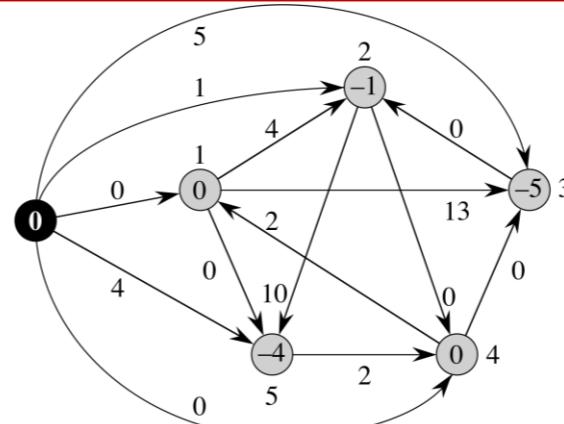
$\text{JOHNSON}(G, w)$

- 1 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 $v \in G.V$
- 2 **if** $\text{BELLMAN-FORD}(G', w, s) == \text{FALSE}$
3 print “the input graph contains a negative-weight cycle”
- 4 **else for** each vertex $v \in G'.V$
5 set $h(v)$ to the value of $\delta(s, v)$
 computed by the Bellman-Ford algorithm
- 6 **for** each edge $(u, v) \in G'.E$
7 $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
- 8 let $D = (d_{uv})$ be a new $n \times n$ matrix
- 9 **for** each vertex $u \in G.V$
10 run $\text{DIJKSTRA}(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
- 11 **for** each vertex $v \in G.V$
12 $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
- 13 **return** D

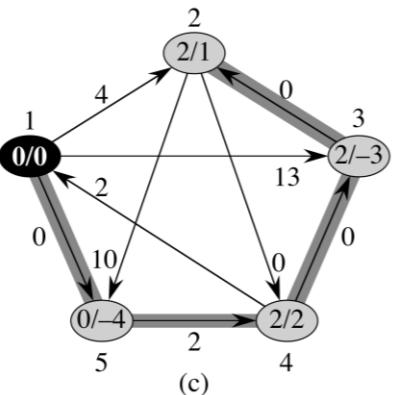
Johnson's algorithm



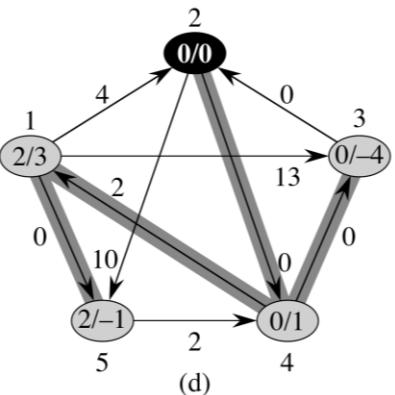
(a)



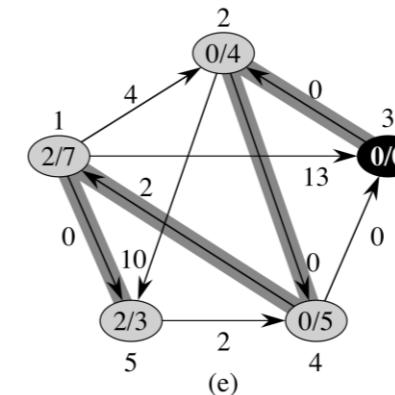
(b)



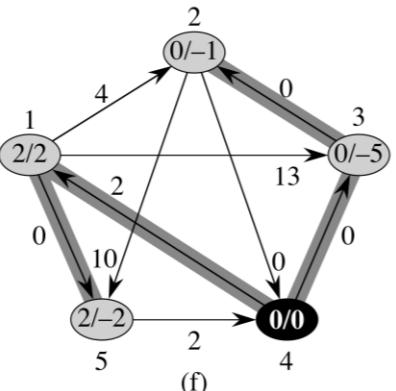
(c)



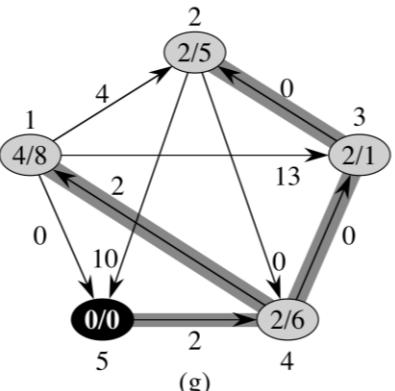
(d)



(e)



(f)



(g)