EL9343

Data Structure and Algorithm

Lecture 12: Shortest Paths

Instructor: Yong Liu

Last Lecture

- Greedy Algorithm (cont.)
  - Huffman codes
- Minimum Spanning Trees
  - Prim's algorithm
  - Kruskal's algorithm

Today

- Single-Source Shortest Paths
  - Nonnegative edge weights: Dijkstra's algorithm
  - Unweighted graphs: BFS
  - General case: Bellman-Ford algorithm
  - DAG: Topological sort + one pass Bellman-Ford

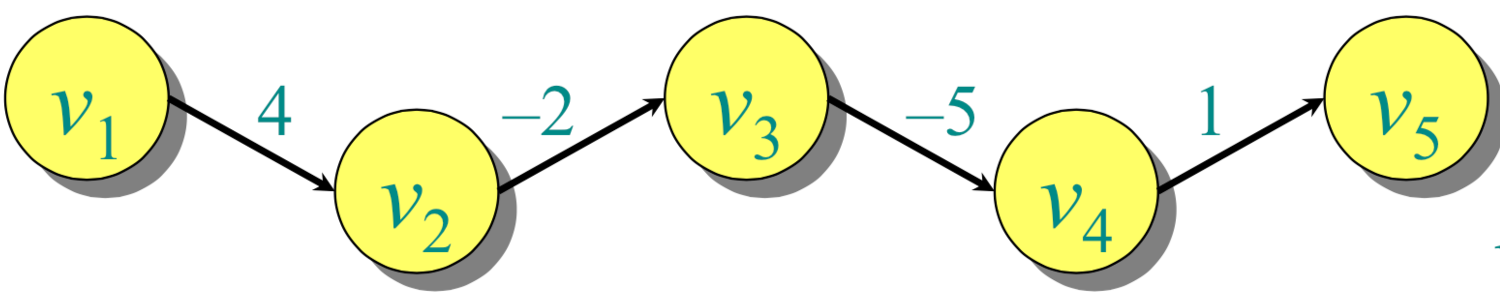
- All Pairs Shortest Paths
  - Nonnegative edge weights: |V| times of Dijkstra's algorithm
  - Unweighted graphs: |V| times of BFS
  - General case: Floyd-Warshall algorithm

Paths in graphs

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) .$$

Example:



$w(p) = -2$

Shortest paths

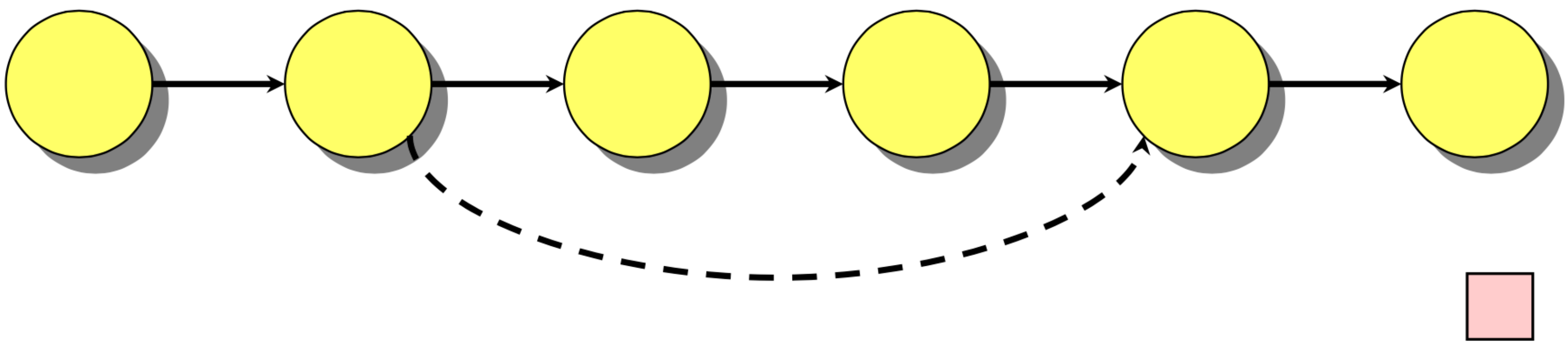
- A shortest path from **u** to **v** is a path of minimum weight from **u** to **v**.
- The shortest- path weight from **u** to **v** is defined as:

  $$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

- Note: $\delta(u, v) = \infty$ if no path from u to v exists.

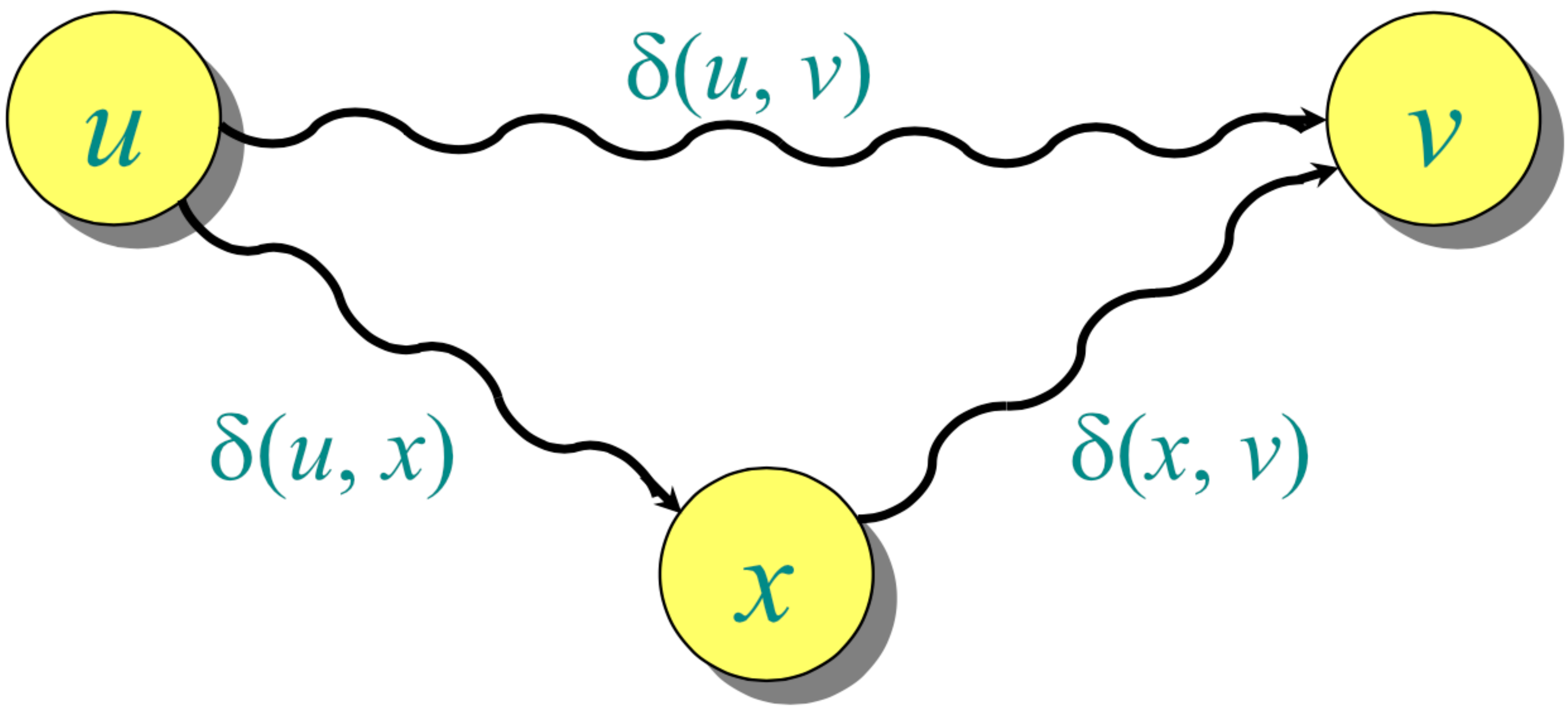
Optimal substructure

- Theorem: A subpath of a shortest path is a shortest path.
- Proof: Cut and paste



Triangle inequality

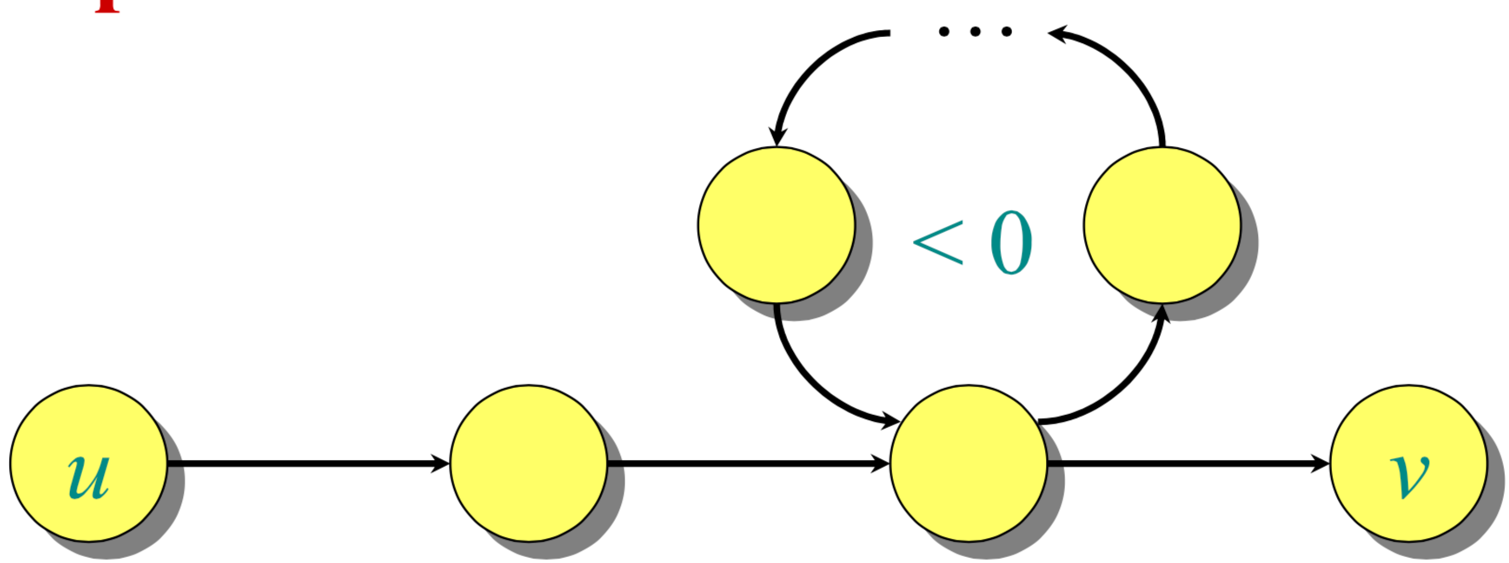
- Theorem: For all u, v, x ∈ V, we have $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$.



Negative-Weight Cycle

- If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Single-source shortest paths

- Problem. From a given source vertex $\mathbf{s} \in \mathbf{V}$, find the shortest-path weights $\boldsymbol{\delta}(\mathbf{s}, \mathbf{v})$ for all $\mathbf{v} \in \mathbf{V}$.
- If all edge weights $\mathbf{w}(\mathbf{u}, \mathbf{v})$ are nonnegative, all shortest-path weights must exist.

IDEA: Greedy

1. Maintain a set $\mathbf{S}$ of vertices whose shortest-path distances from $\mathbf{s}$ are known.
2. At each step add to $\mathbf{S}$ the vertex $\mathbf{v} \in \mathbf{V} - \mathbf{S}$ whose distance estimate from $\mathbf{s}$ is minimal.
3. Update the distance estimates of vertices adjacent to $\mathbf{v}$.

Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ $\triangleright$ Q is a priority queue maintaining $V - S$

Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ⊳ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$

relaxation step

Implicit DECREASE-KEY

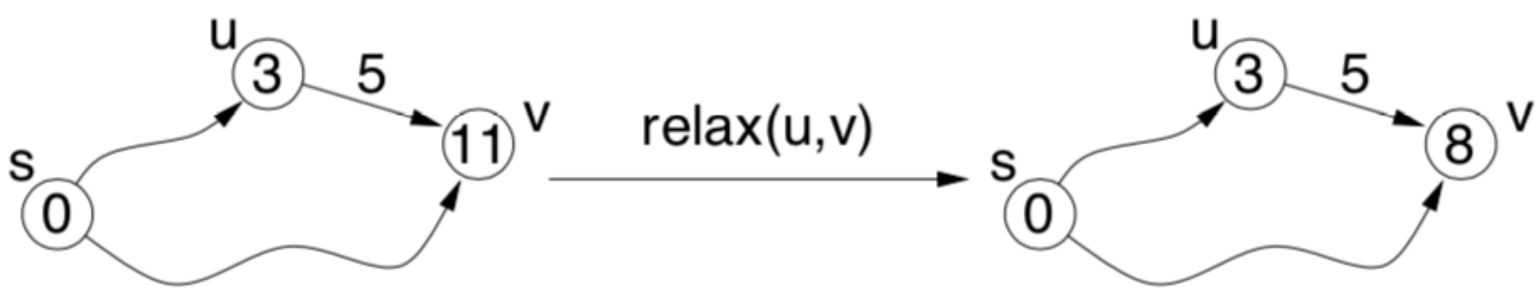
Shortest Paths and Relaxation

The basic structure of Dijkstra's algorithm is to maintain an *estimate* of the shortest path for each vertex, call this d[v].

- Algorithm works in iterations
- At each iteration, d[v] will be the length of the shortest path *that the algorithm knows of* from s to v. This value will always greater than or equal to the true shortest path distance from s to v.
- Initially d[s] = 0, no paths to all other vertices, so all other d[v] values are set to ∞.
- As the algorithm goes on, and sees more and more vertices, it attempts to update d[v] for each vertex in the graph, until all the d[v] values converge to the true shortest distances.

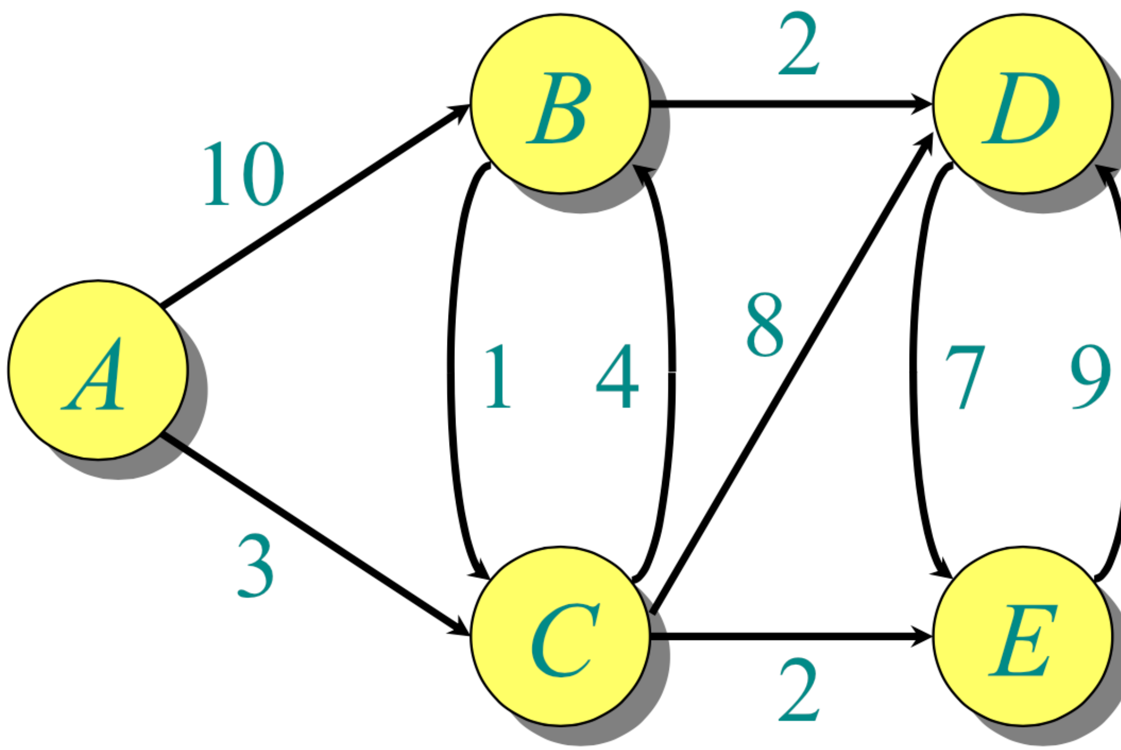
Shortest Paths and Relaxation

The process by which an estimate is updated is called *relaxation*. Here is how relaxation works. Intuitively, if you can see that your solution is not yet reached an optimum value, then push it a little closer to the optimum. In particular, if you discover a path from s to v shorter than d[v], then you need to update d[v].



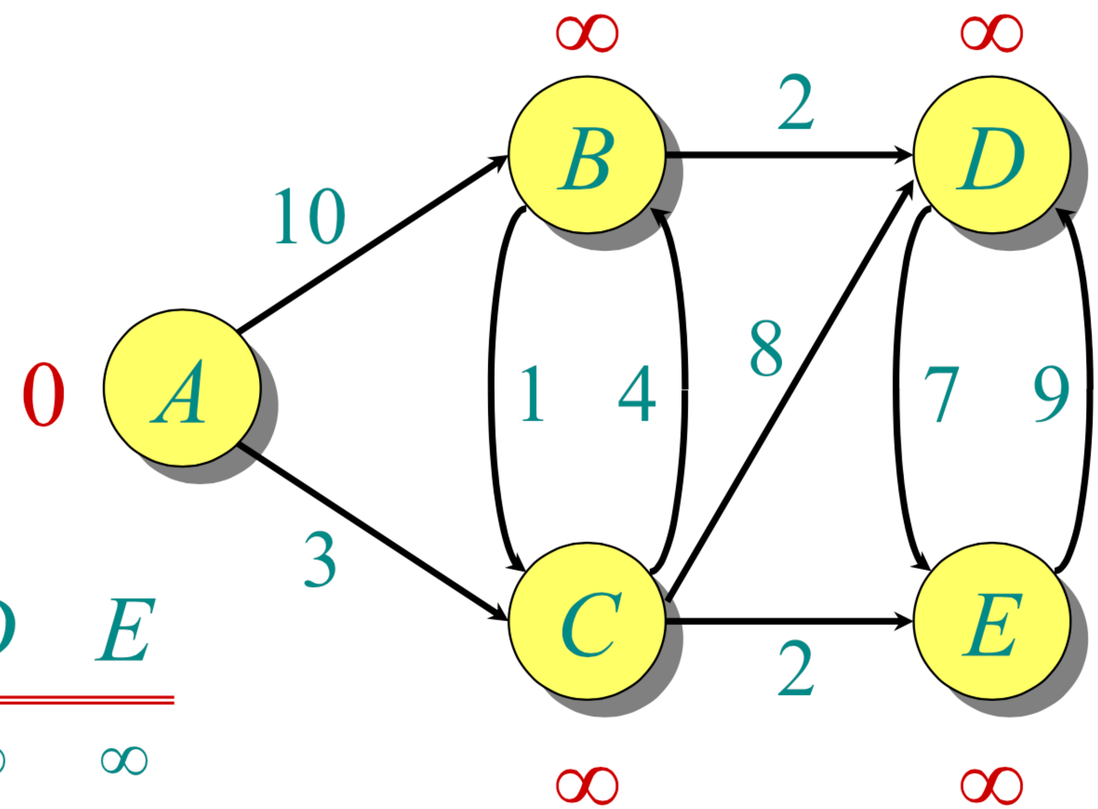
Example of Dijkstra’s algorithm

Graph with nonnegative edge weights



Example of Dijkstra's algorithm

Initialize:

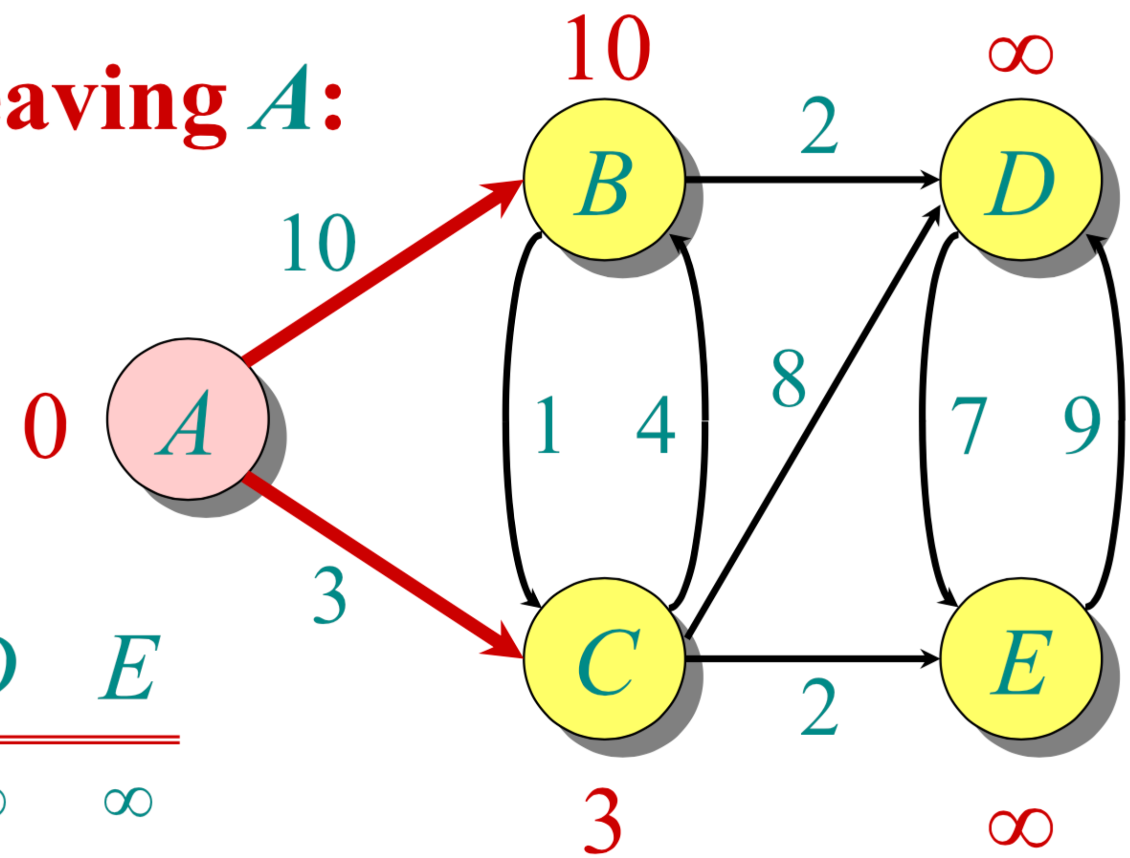


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

Relax all edges leaving *A*:

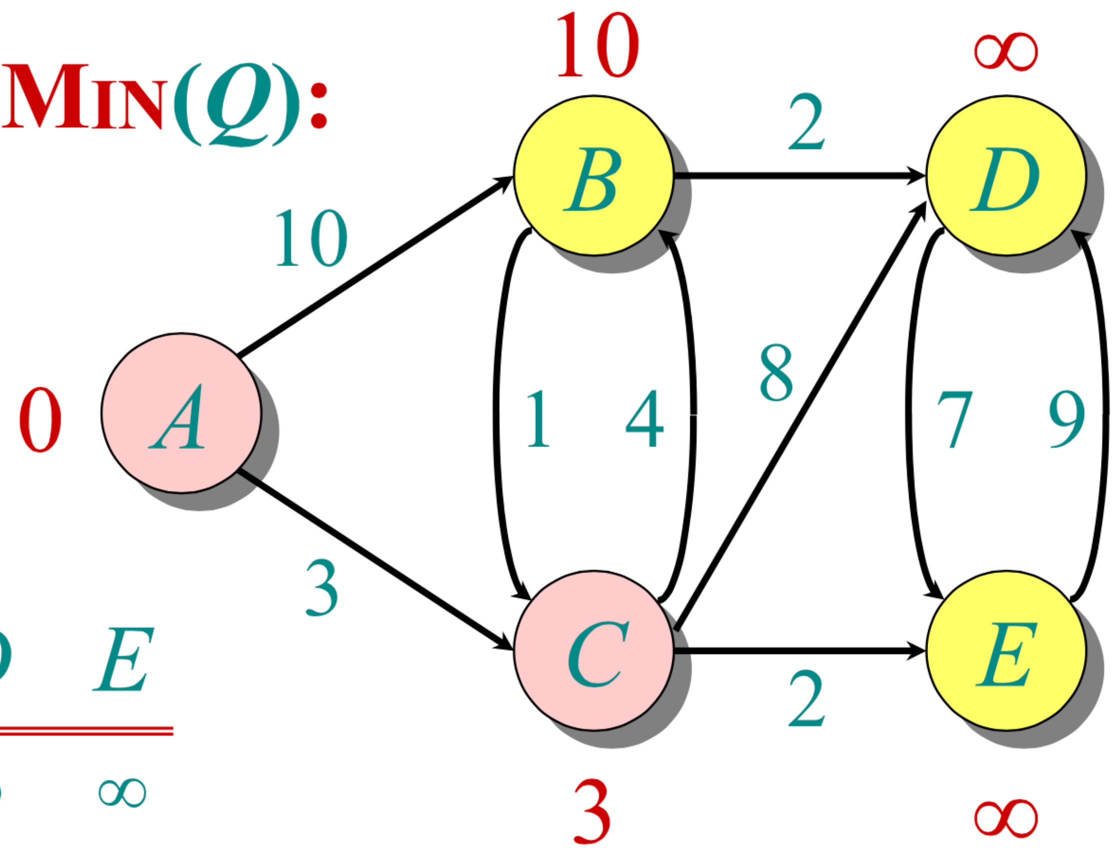


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Example of Dijkstra's algorithm

"C" $\leftarrow$ EXTRACT-MIN(Q):

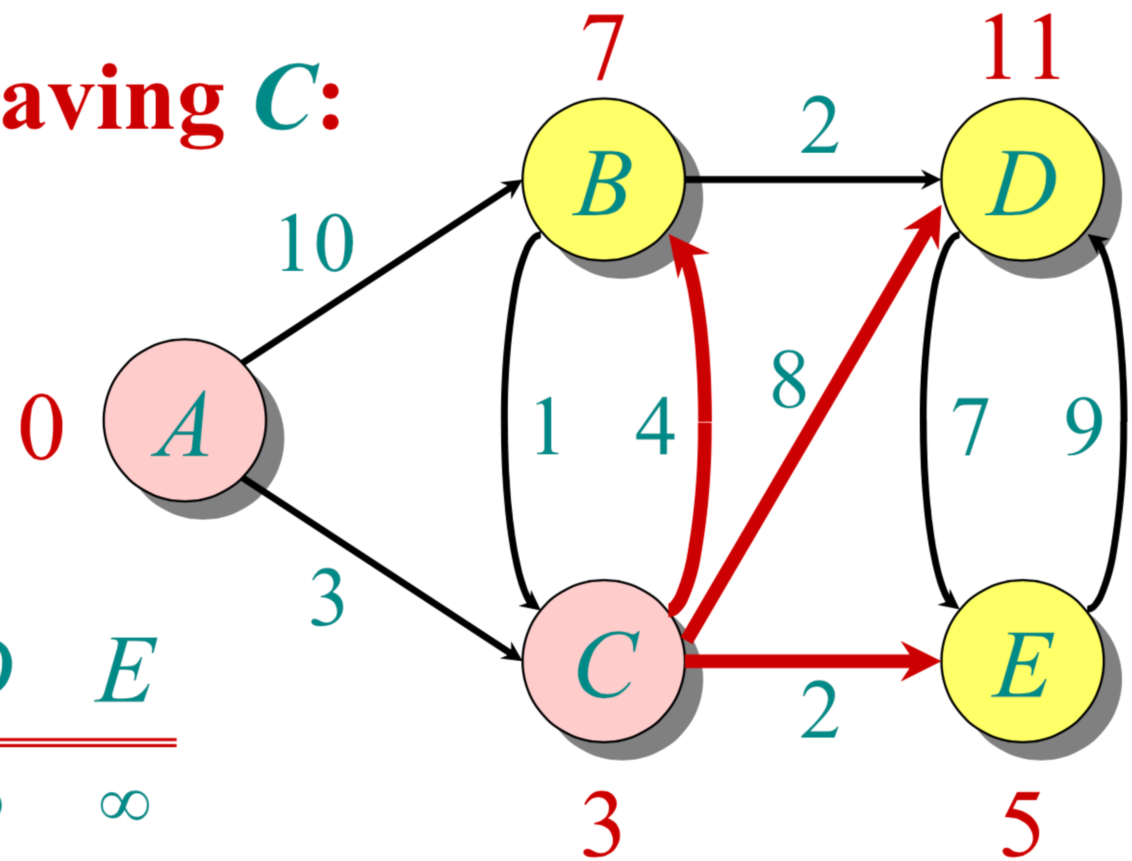


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: $\{ A, C \}$

Example of Dijkstra's algorithm

Relax all edges leaving C:

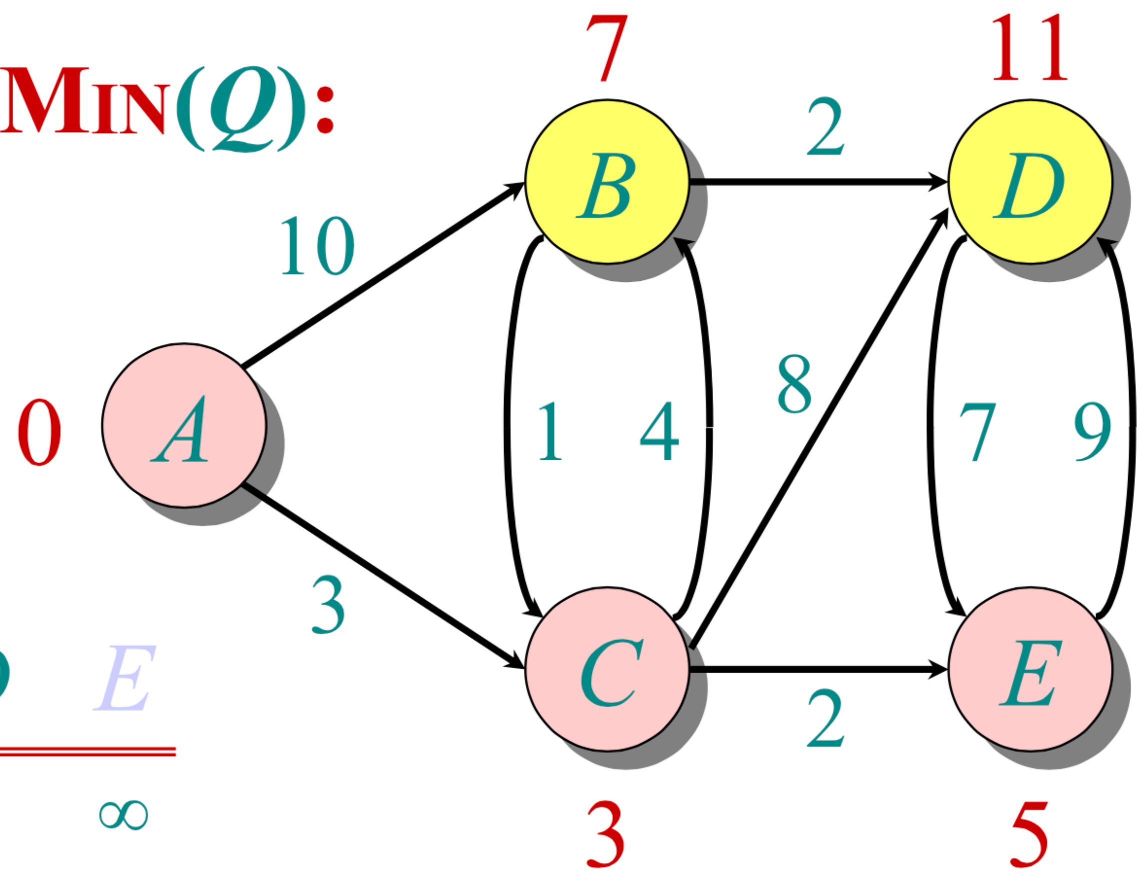


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: $\{ A, C \}$

Example of Dijkstra's algorithm

"E" ← EXTRACT-MIN(Q):



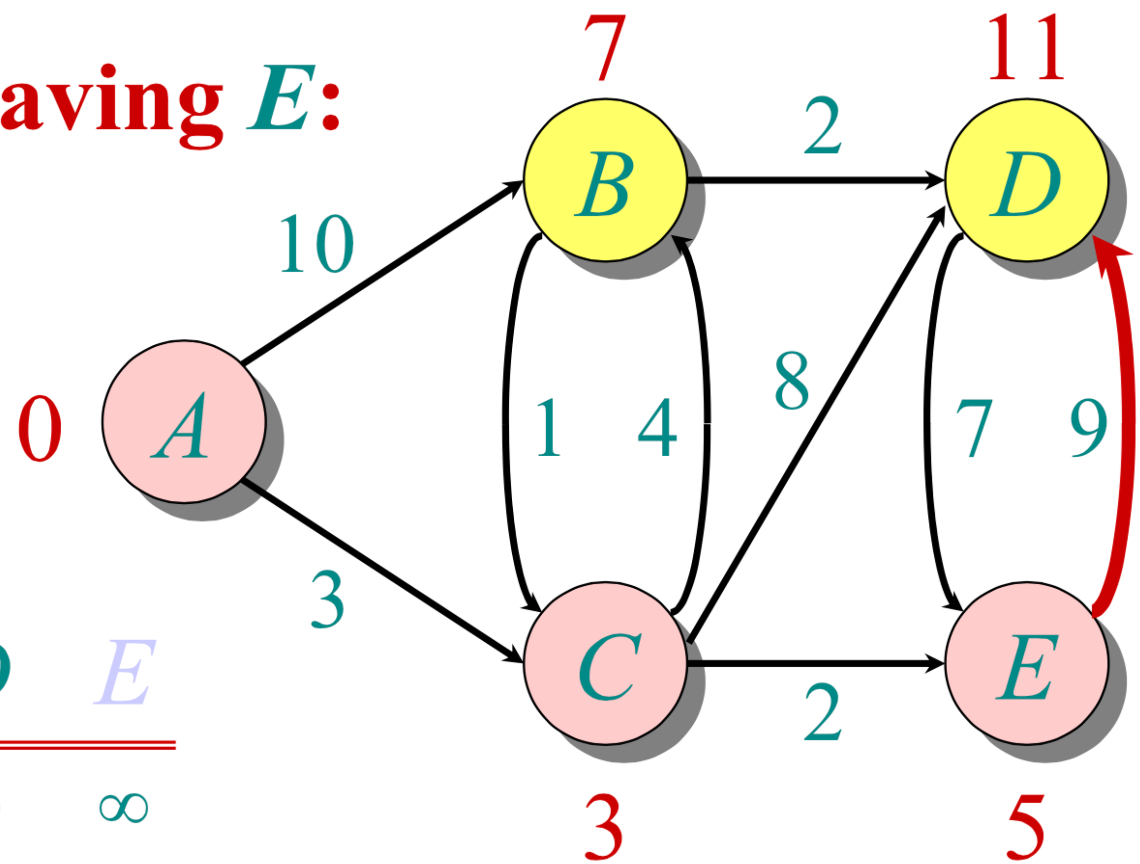
Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

S: { A, C, E }

Example of Dijkstra's algorithm

Relax all edges leaving E:

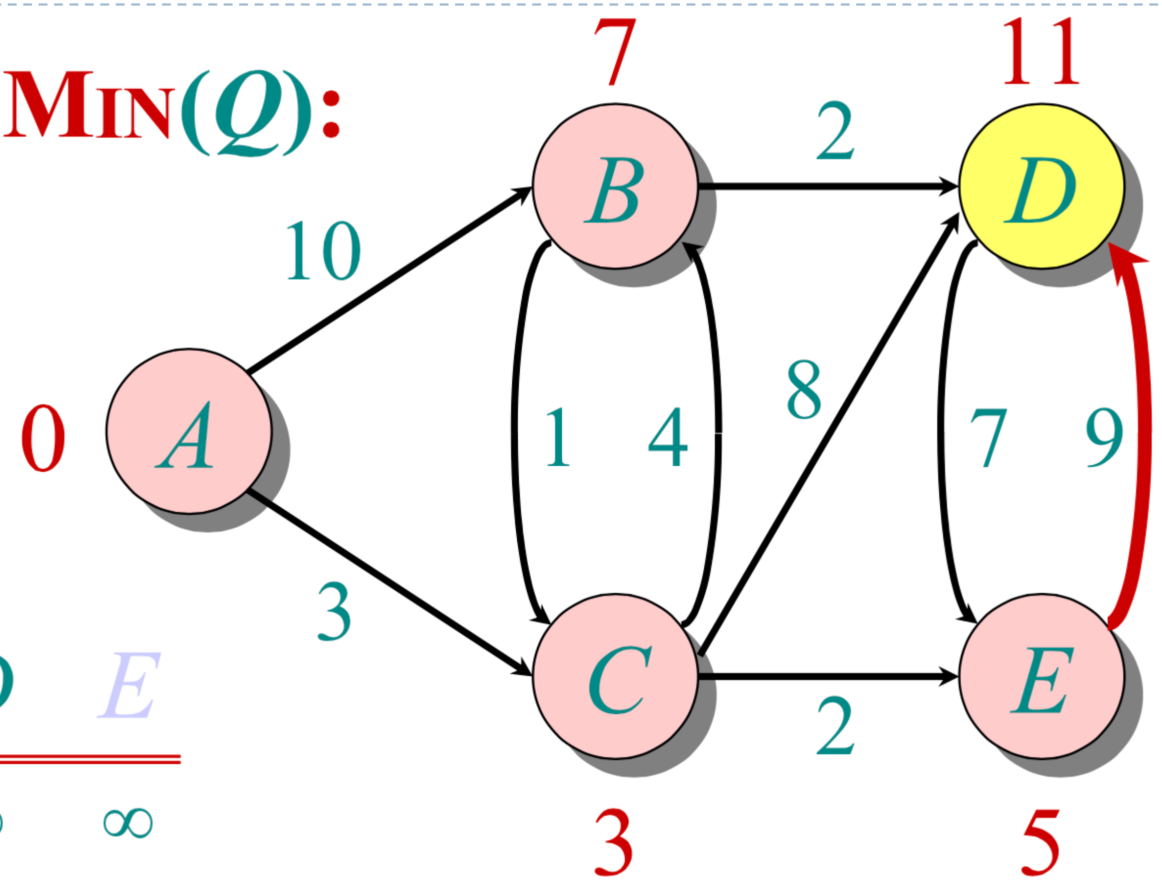


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Example of Dijkstra's algorithm

"B" $\leftarrow$ **EXTRACT-MIN(Q):**

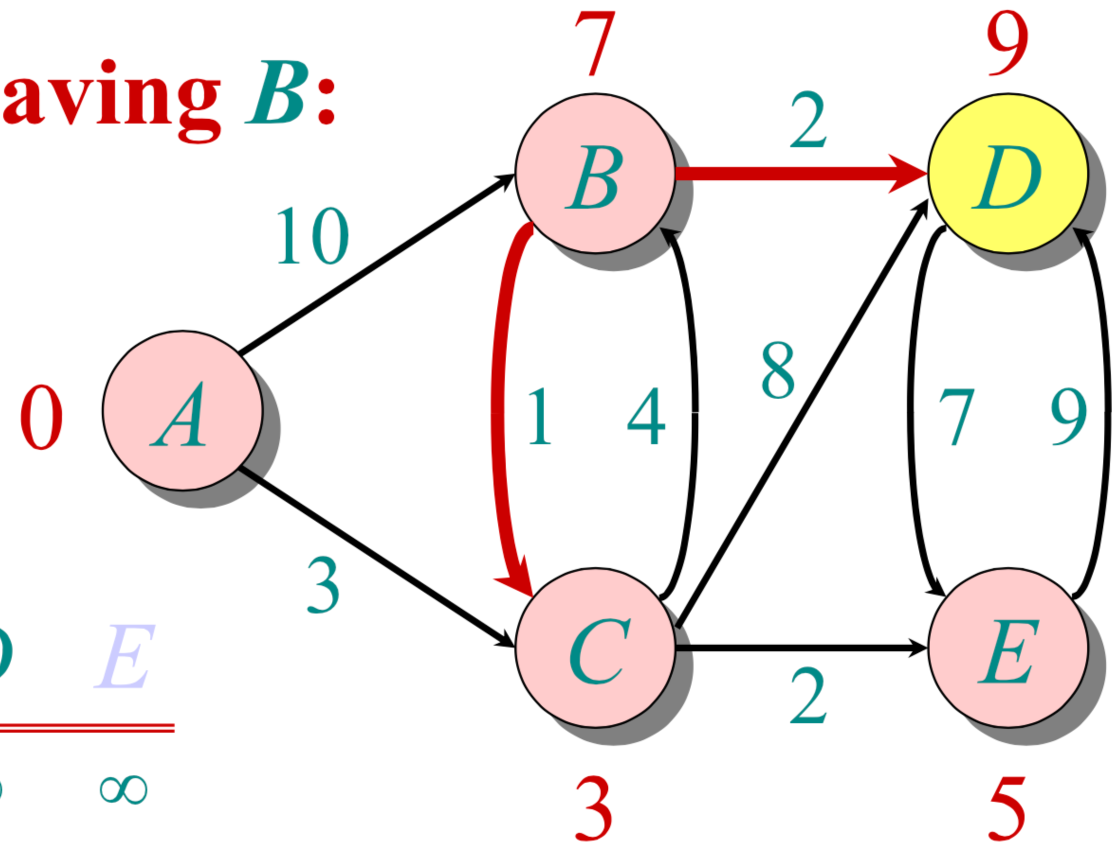


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example of Dijkstra’s algorithm

Relax all edges leaving *B*:

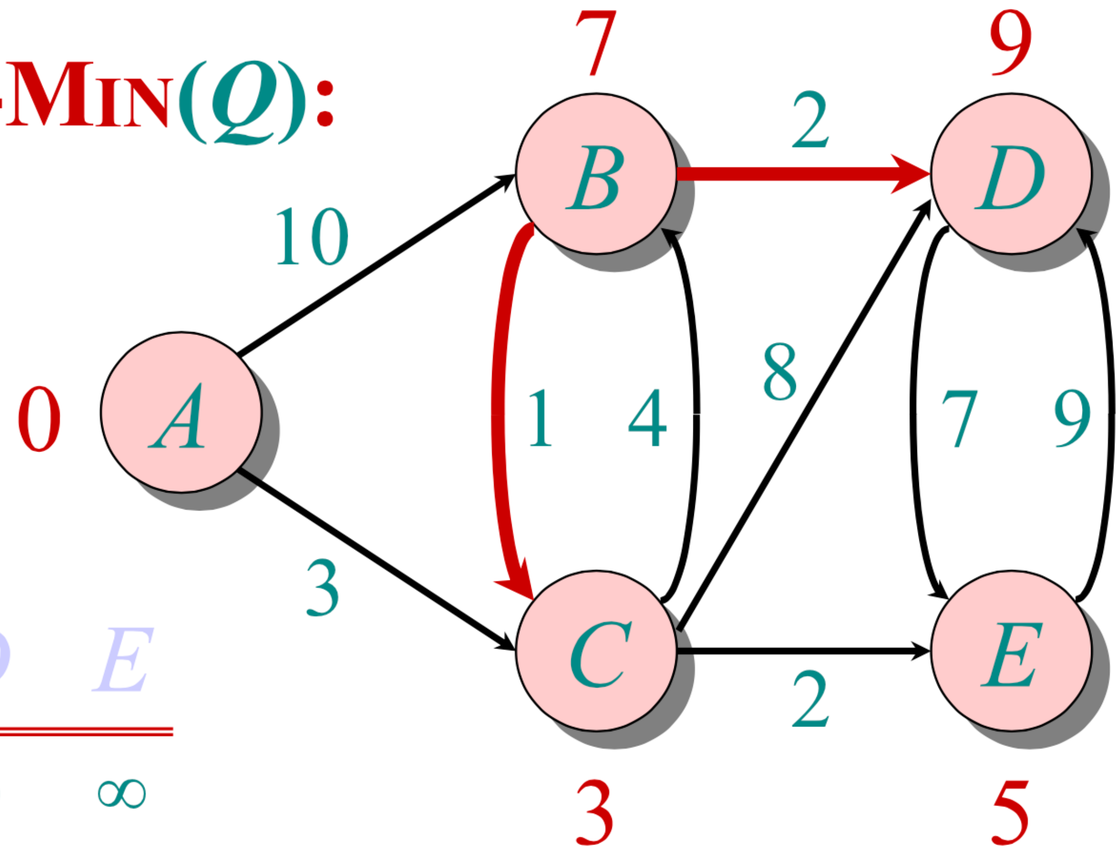


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Example of Dijkstra's algorithm

"D" $\leftarrow$ EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Analysis of Dijkstra's algorithm

$$
|V| \text{ times} \left\{
\begin{array}{l}
\textbf{while } Q \neq \emptyset \\
\quad \textbf{do } u \leftarrow \text{EXTRACT-MIN}(Q) \\
\quad \phantom{\textbf{do }} S \leftarrow S \cup \{u\} \\
\quad \phantom{\textbf{do }} degree(u) \text{ times} \left\{
\begin{array}{l}
\textbf{for} \text{ each } v \in Adj[u] \\
\quad \textbf{do if } d[v] > d[u] + w(u, v) \\
\quad\quad \textbf{then } d[v] \leftarrow d[u] + w(u, v)
\end{array}
\right.
\end{array}
\right.
$$

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

$$\text{Time} = \Theta(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}})$$

- Same formula as in the analysis of Prim's minimum spanning tree algorithm.

Analysis of Dijkstra's algorithm

$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

- Same formula as in the analysis of Prim's minimum spanning tree algorithm.

Unweighted graphs

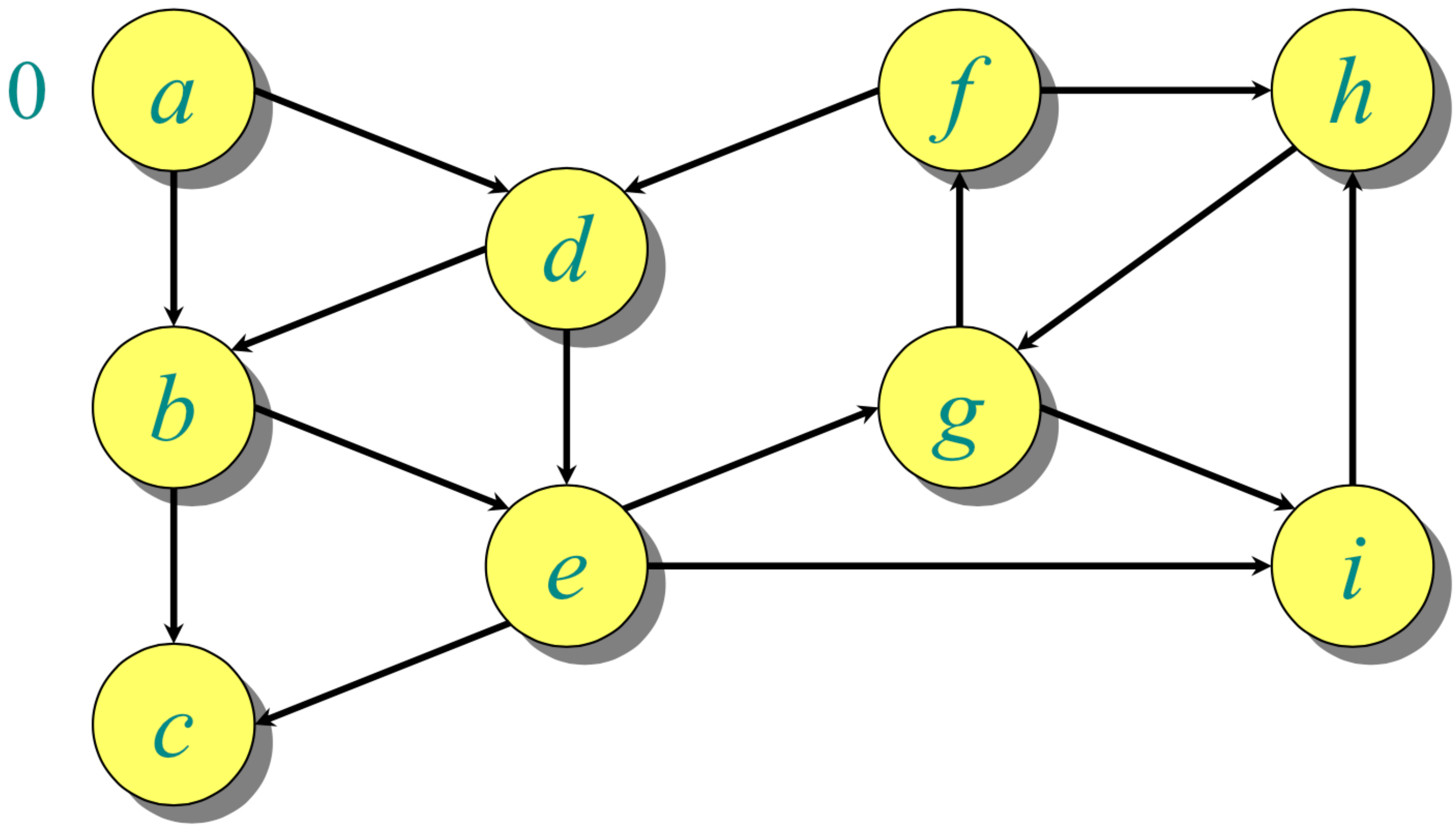
Suppose that **w(u, v) = 1** for all **(u, v)** $\in$ **E**. Can Dijkstra's algorithm be improved?

- **Breadth-first search!**

$$
\begin{array}{l}
\textbf{while } Q \neq \emptyset \\
\quad \textbf{do } u \leftarrow \text{DEQUEUE}(Q) \\
\quad\quad \textbf{for each } v \in Adj[u] \\
\quad\quad\quad \textbf{do if } d[v] = \infty \\
\quad\quad\quad\quad \textbf{then } d[v] \leftarrow d[u] + 1 \\
\quad\quad\quad\quad\quad\quad \text{ENQUEUE}(Q, v)
\end{array}
$$

Analysis: Time = O(V + E).

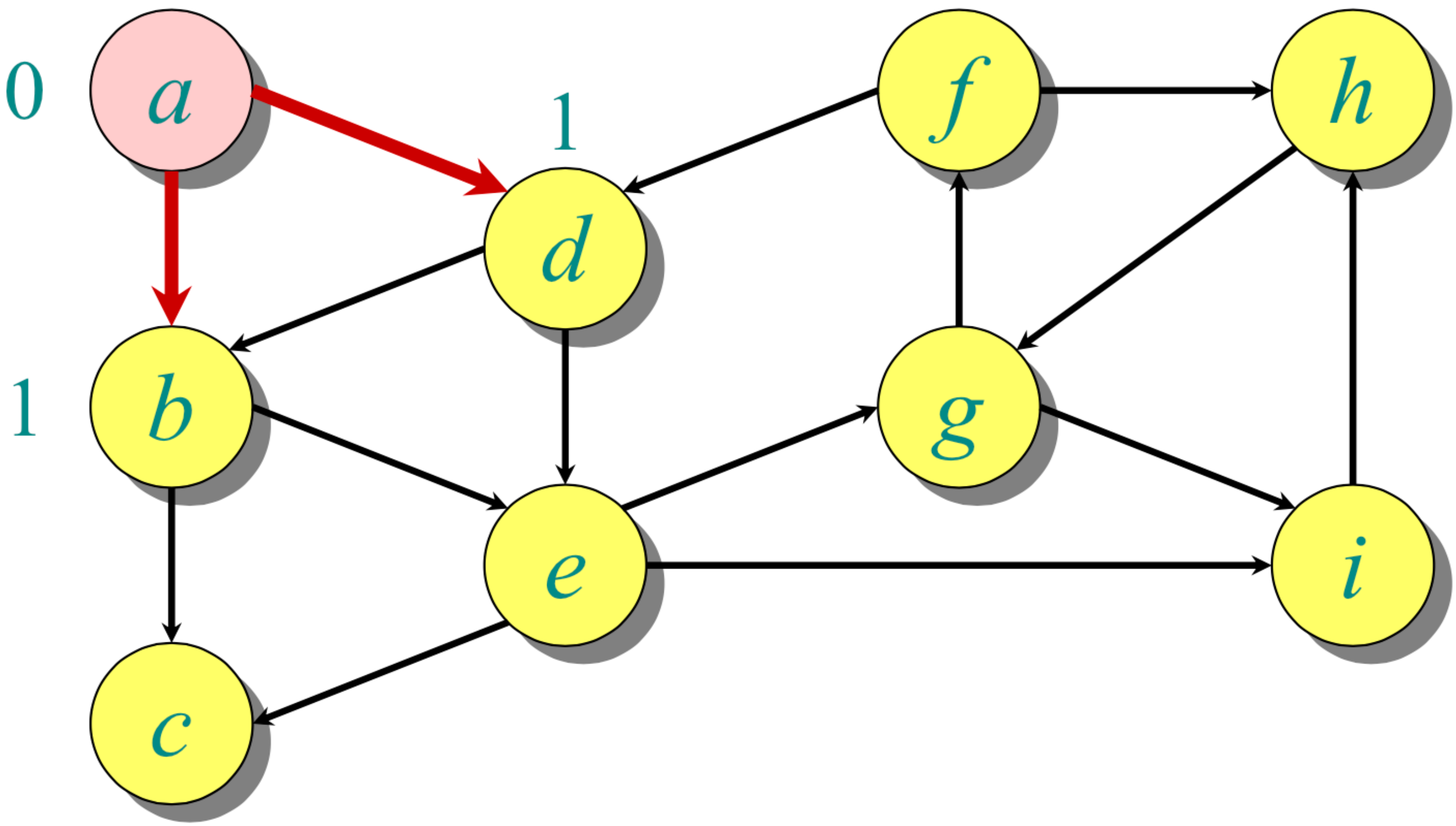
Example of breadth-first search



0

Q: a

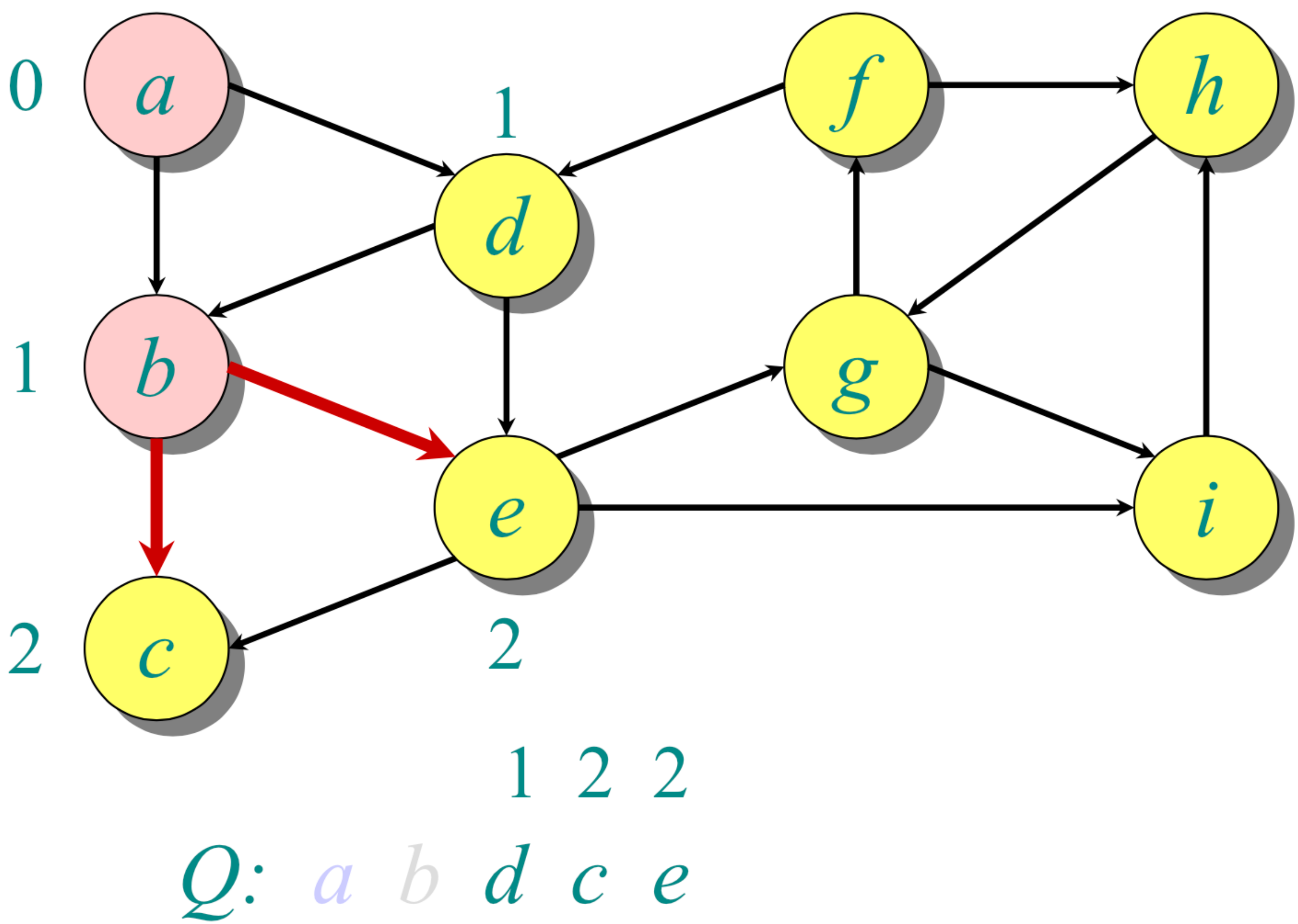
Example of breadth-first search



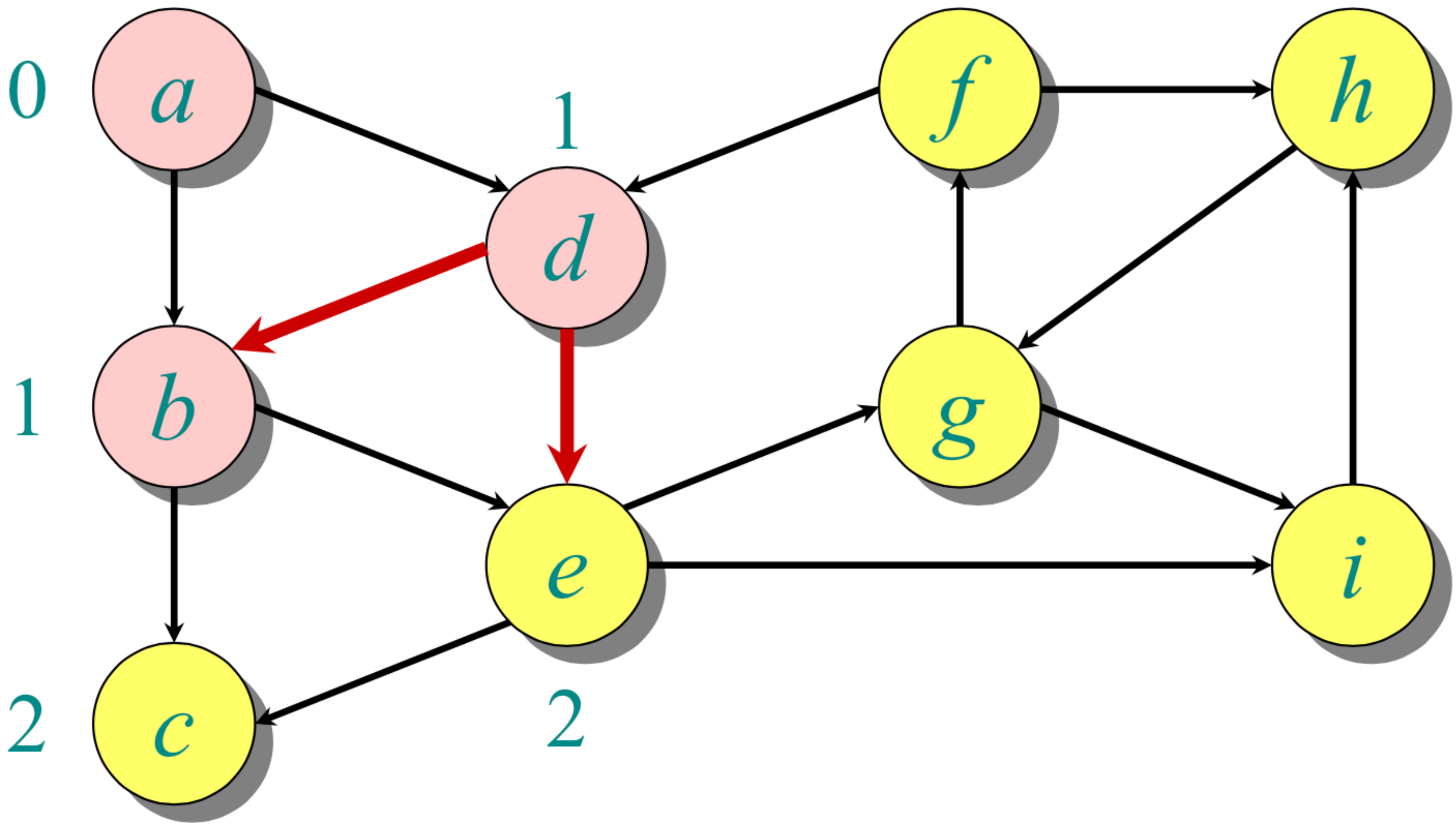
1 1

Q: *a* *b* *d*

Example of breadth-first search



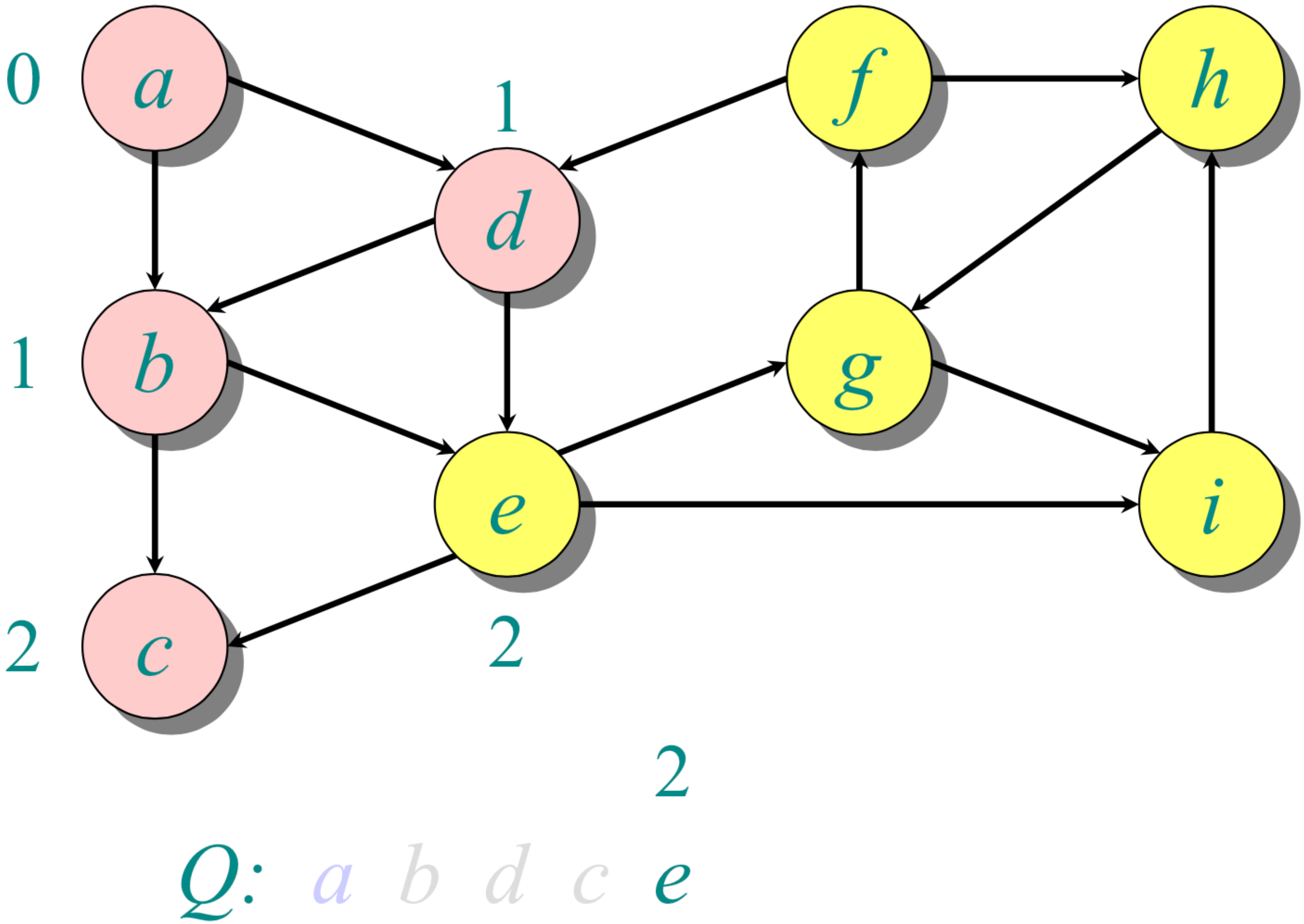
Example of breadth-first search



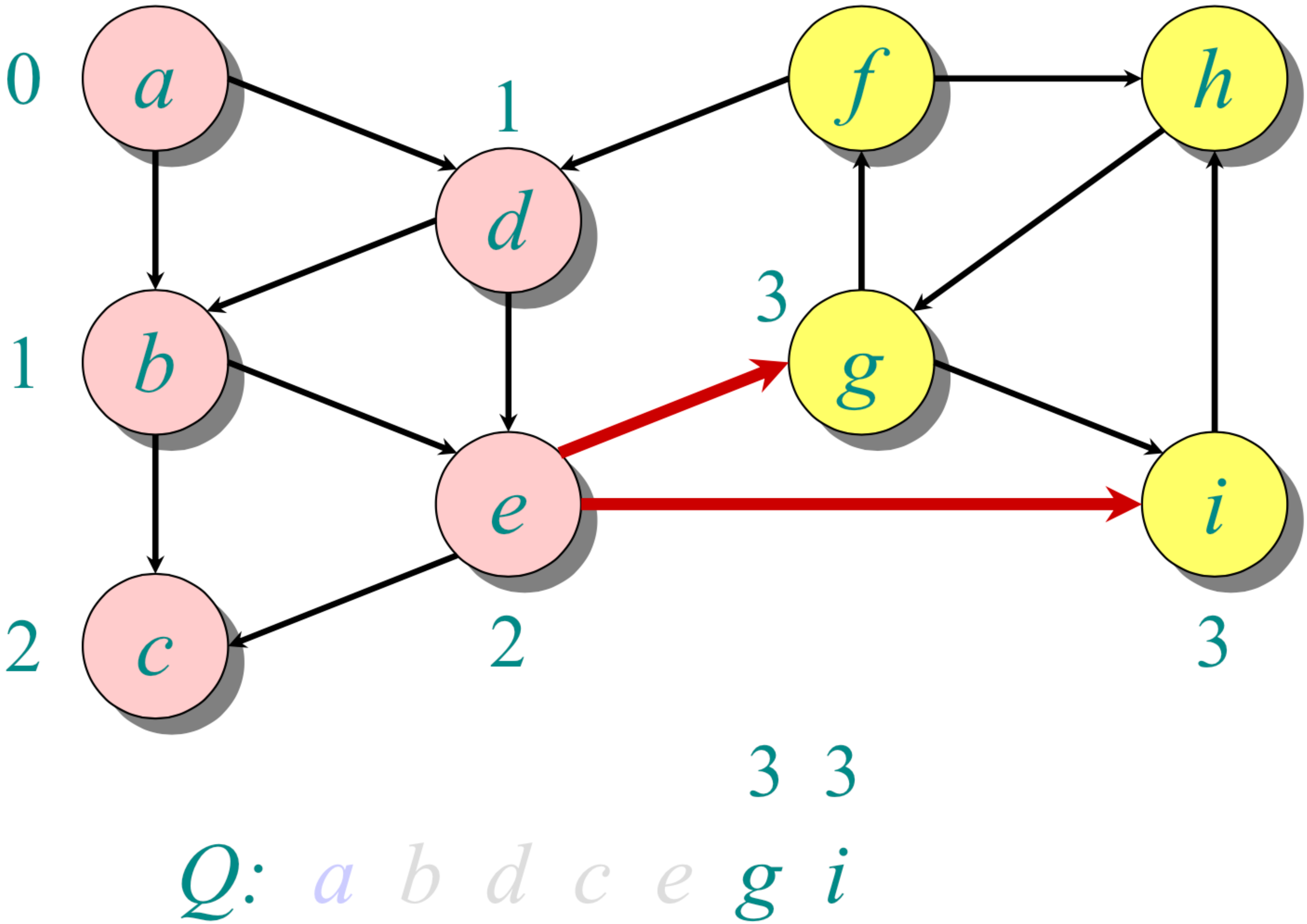
2 2

Q: a b d c e

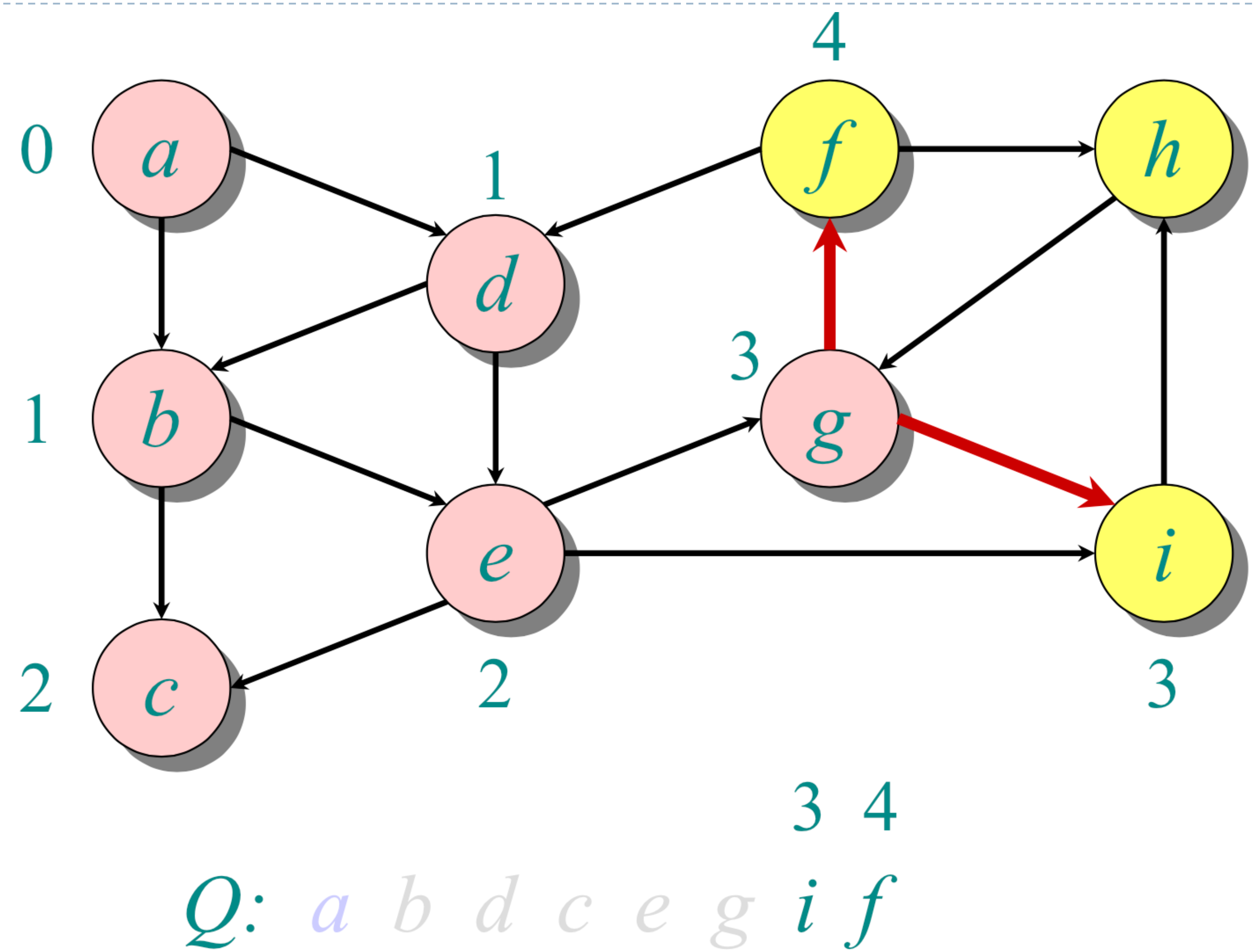
Example of breadth-first search



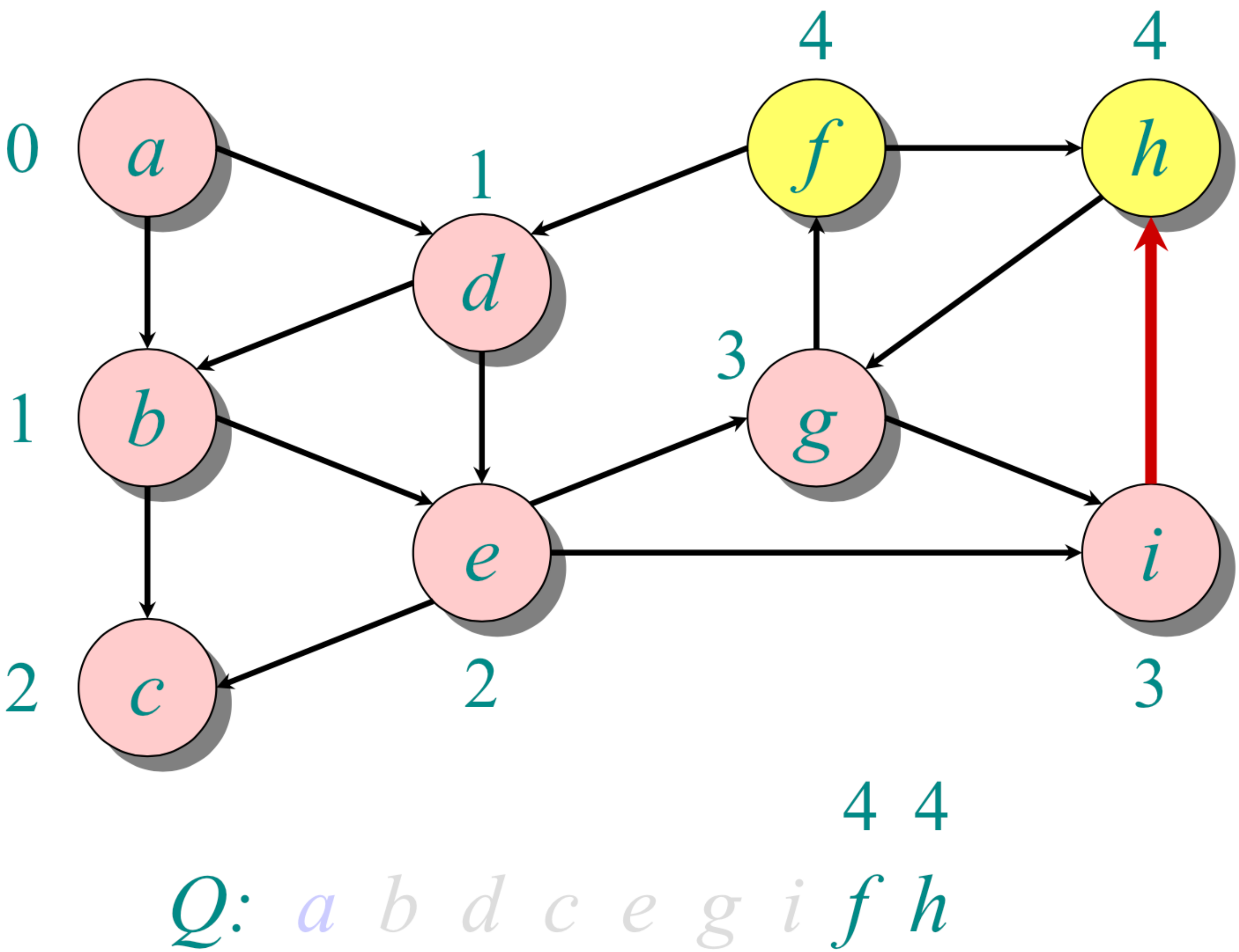
Example of breadth-first search



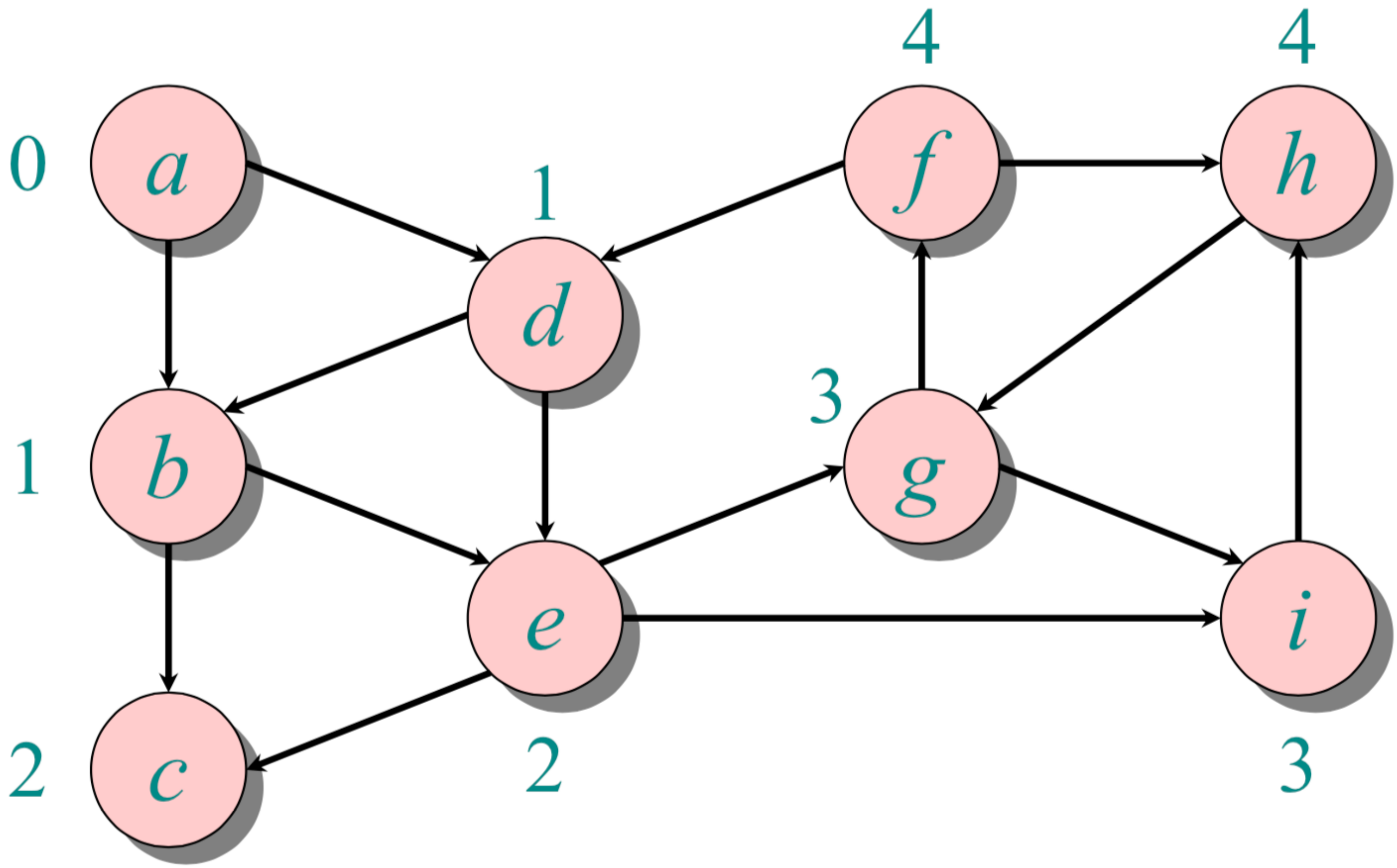
Example of breadth-first search



Example of breadth-first search



Example of breadth-first search

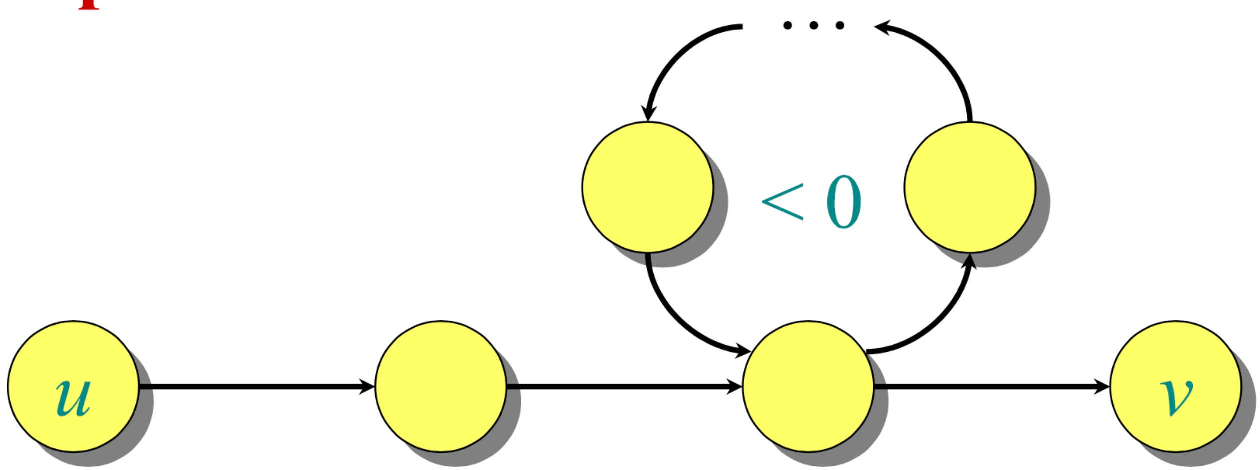


Q: a b d c e g i f h

Negative-Weight Cycle

- Recall: If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Bellman-Ford algorithm: Finds all shortest-path lengths from a source **s** $\in$ **V** to all **v** $\in$ **V** or determines that a negative-weight cycle exists.

Bellman-Ford algorithm

$$
\left.\begin{array}{l}
d[s] \leftarrow 0 \\
\textbf{for } \text{each } v \in V - \{s\} \\
\quad \textbf{do } d[v] \leftarrow \infty
\end{array}\right\} \text{ initialization}
$$

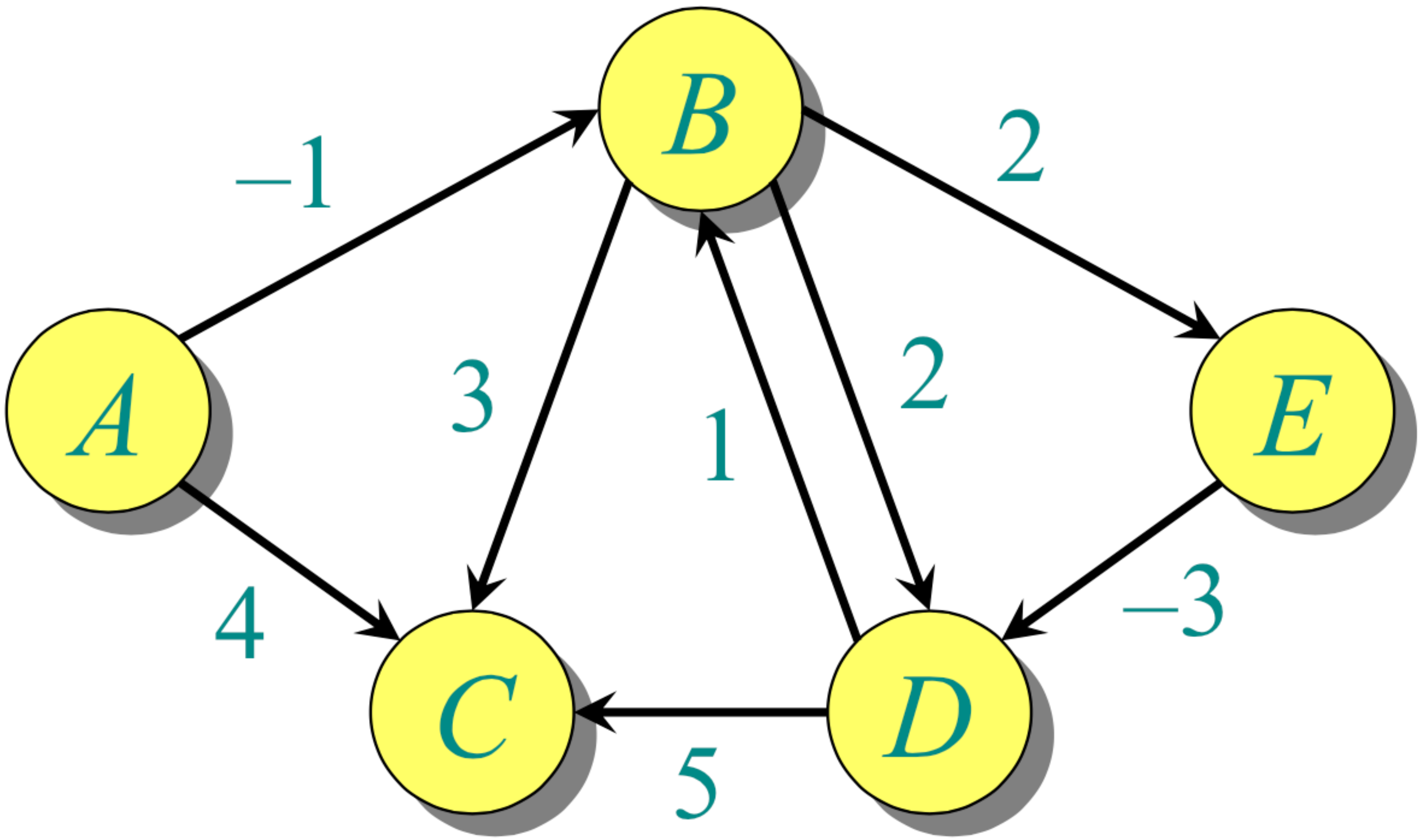
for $i \leftarrow 1$ **to** $|V| - 1$
    do for each edge $(u, v) \in E$
        do if $d[v] > d[u] + w(u, v)$
            then $d[v] \leftarrow d[u] + w(u, v)$ } ***relaxation step***

for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
        then report that a negative-weight cycle exists

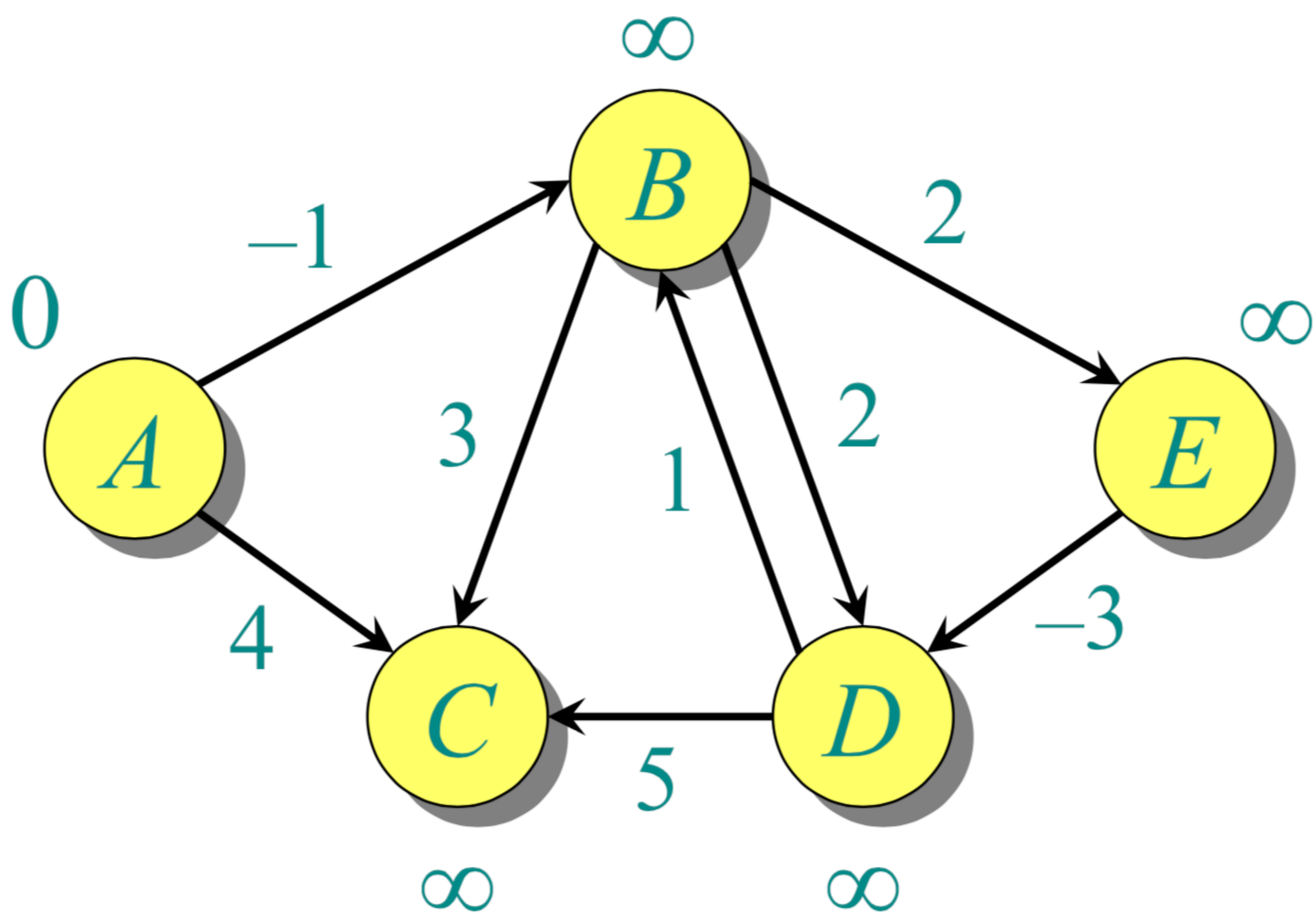
Time = O(VE).

At the end, **d[v] = δ(s, v)**, if no negative-weight cycles.

Example of Bellman-Ford

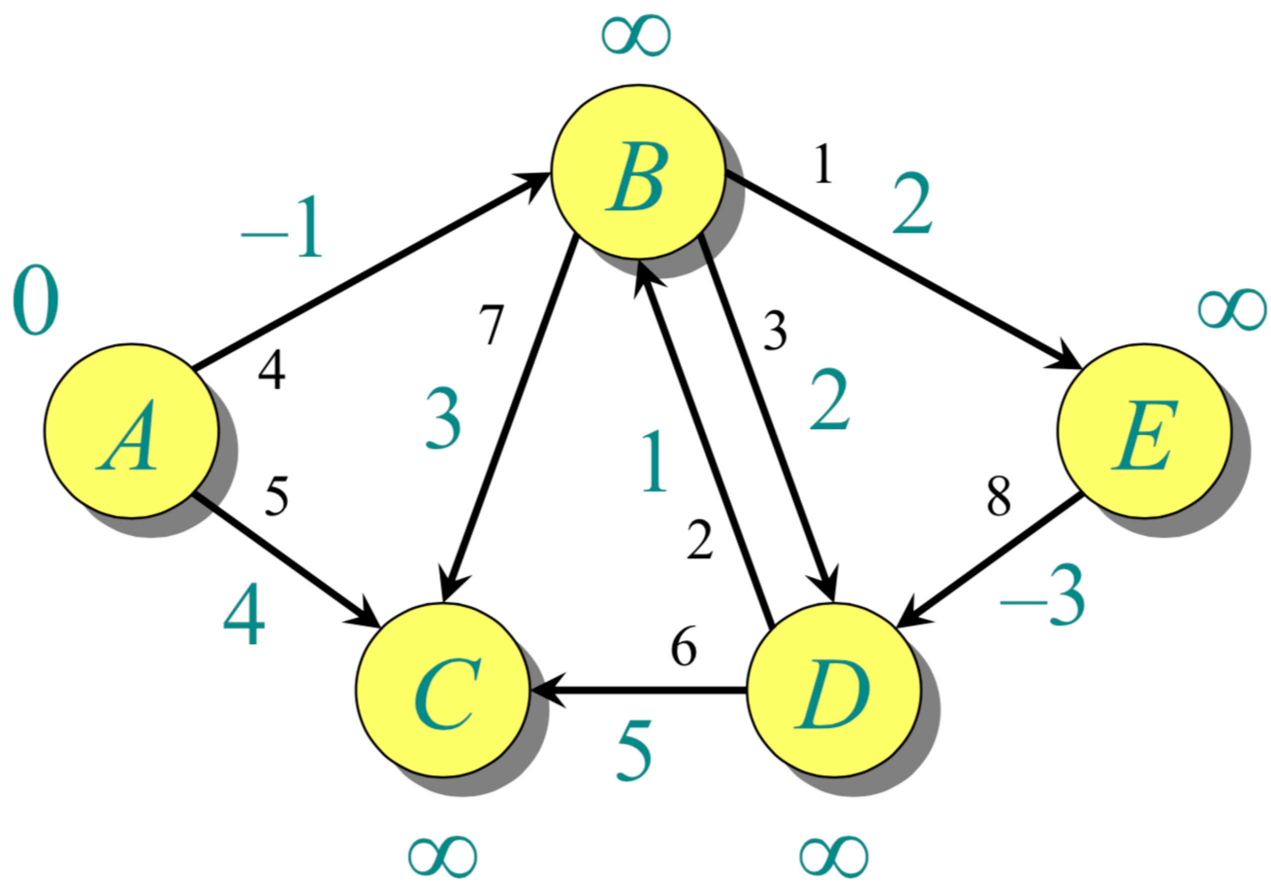


Example of Bellman-Ford



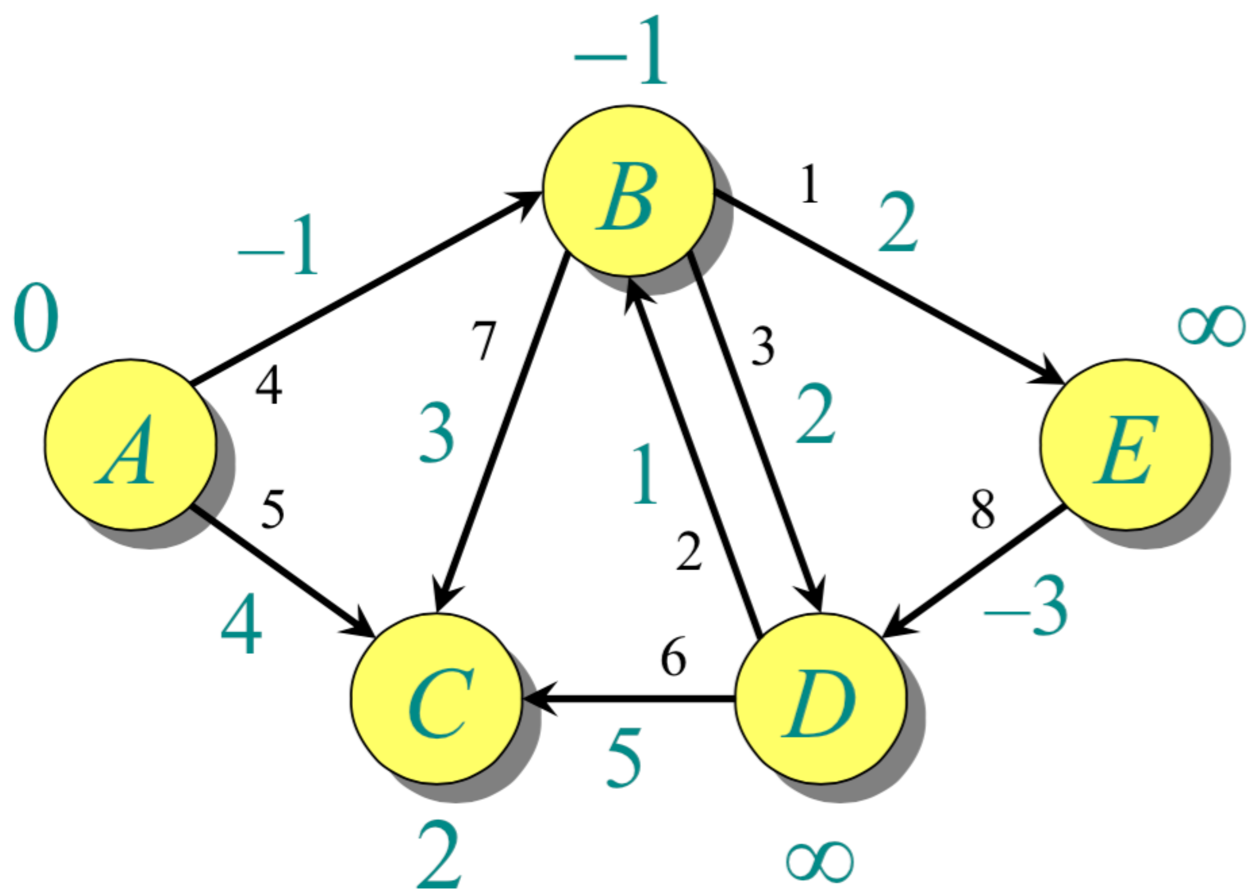
Initialization.

Example of Bellman-Ford



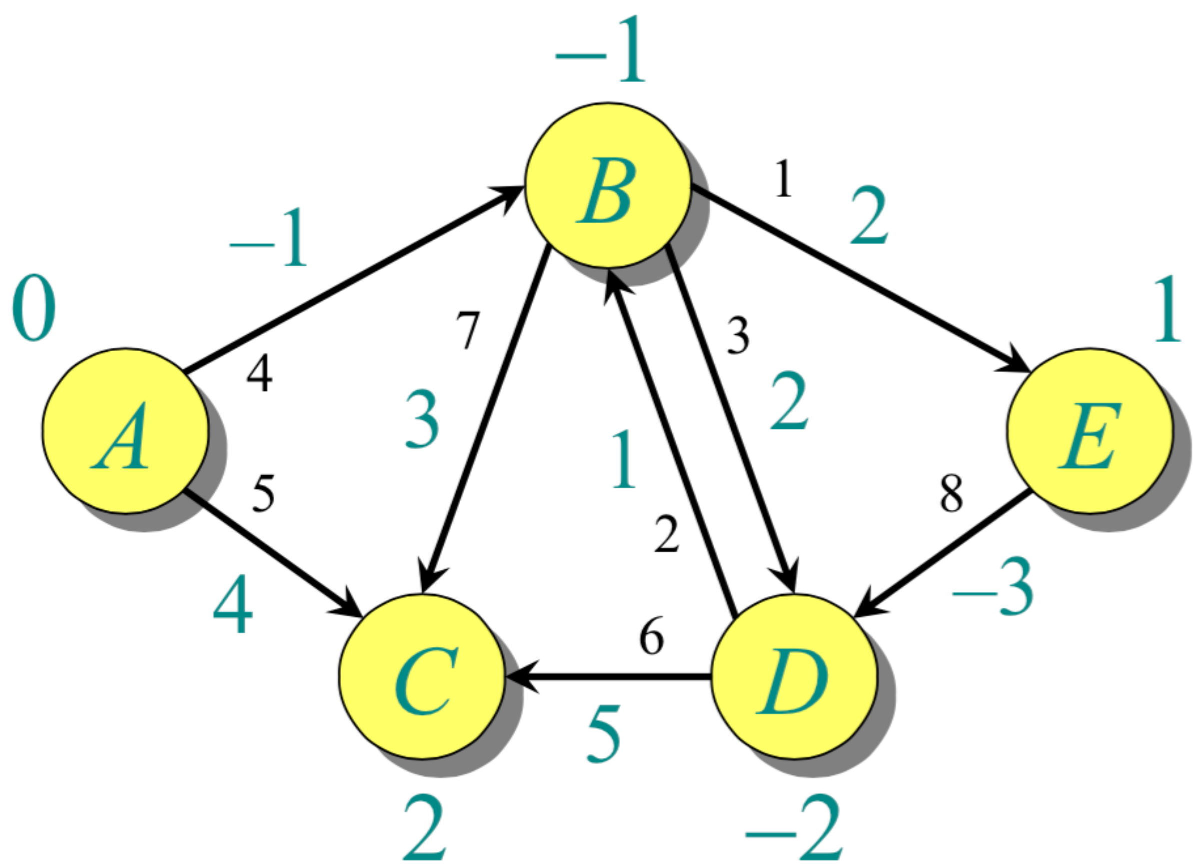
Order of edge relaxation.

Example of Bellman-Ford



End of pass 1.

Example of Bellman-Ford



End of pass 2 (and 3 and 4).

Detection of negative-weight cycles

- Corollary: If a value **d[v]** fails to converge after **|V| – 1** passes, there exists a negative-weight cycle in **G** reachable from **s**.

Shortest Paths in Directed Acyclic Graphs

- Shortest path are always well defined in a DAG: even if there are negative weight edges, no negative-weight cycle can exist.

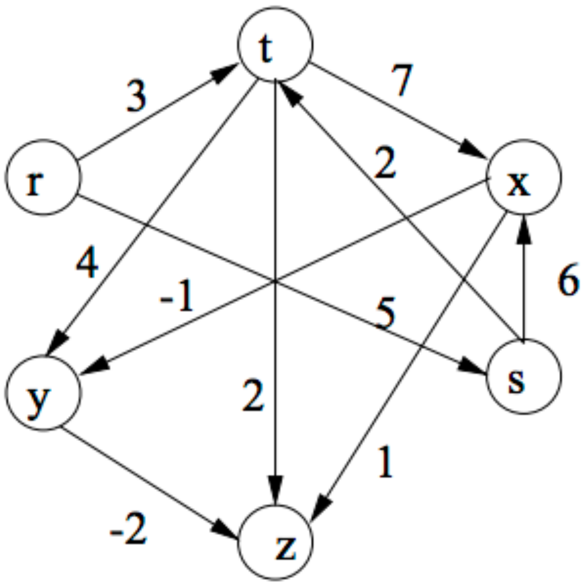
- By relaxing the edges of weighted DAG $G=(V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V+E)$ time
  - Topological sort on the DAG to impose linear ordering on vertices
  - Make one pass relaxation over the vertices in the topological sort order. As we process each vertex, relax each edge that leaves the vertex.

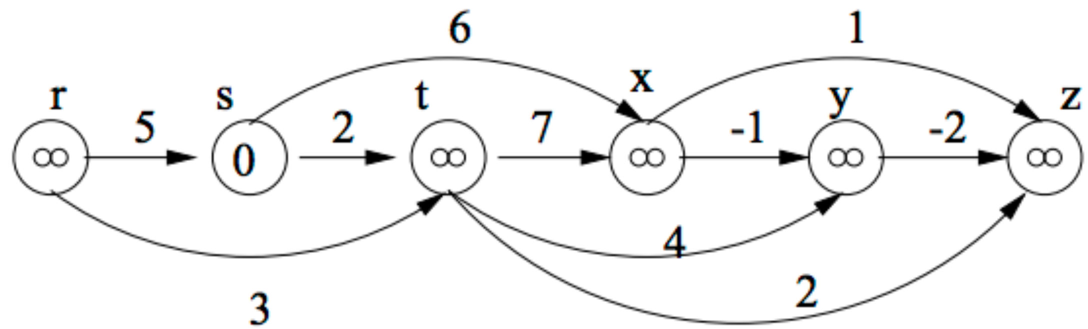
Shortest Paths in DAG

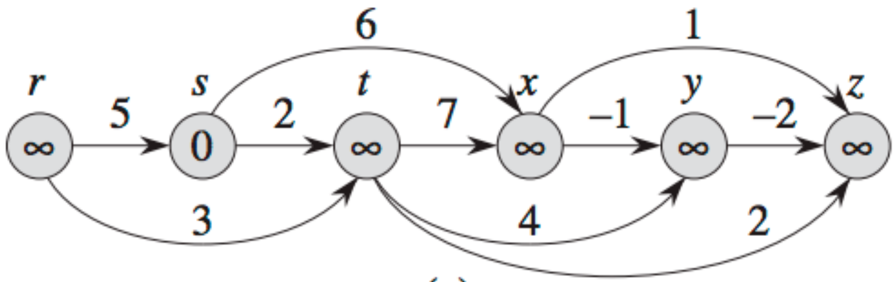
- Topological sort + one pass Bellman-Ford

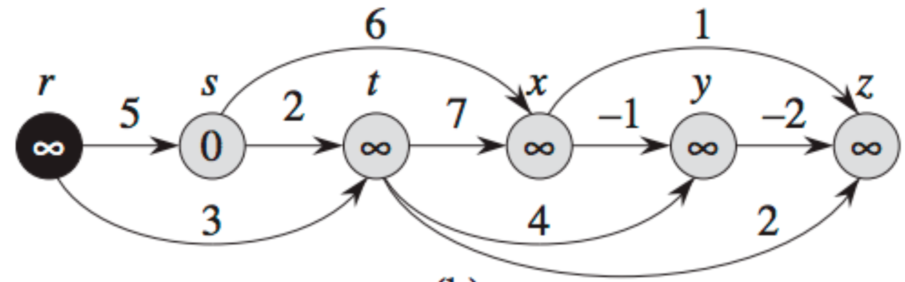
```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

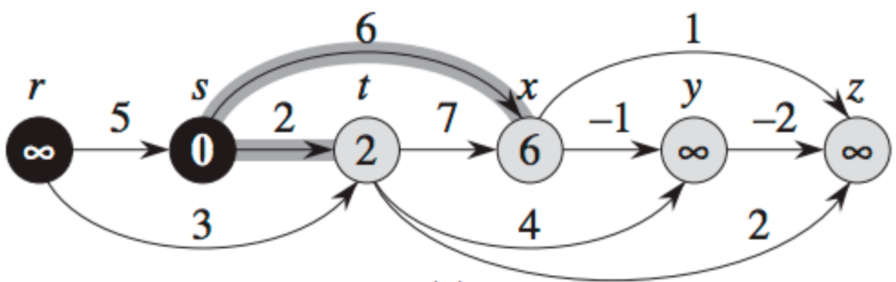
Example of Shortest Paths in DAG

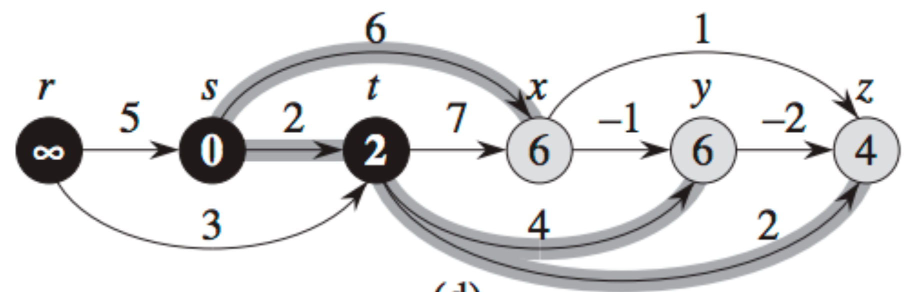


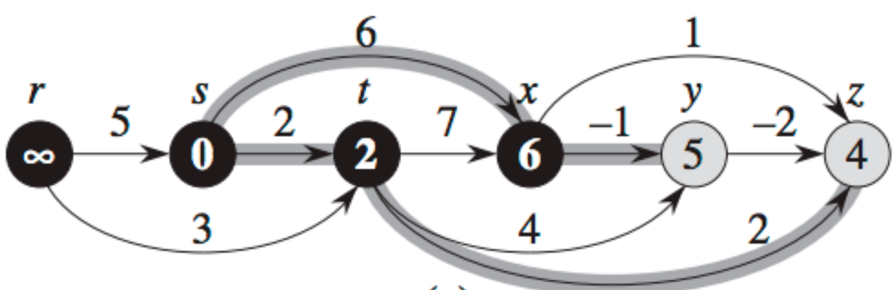


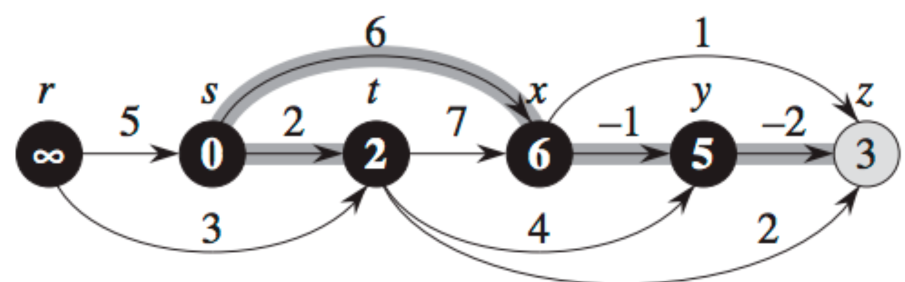

6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
∞
∞
∞
∞
3
4
2
(a)

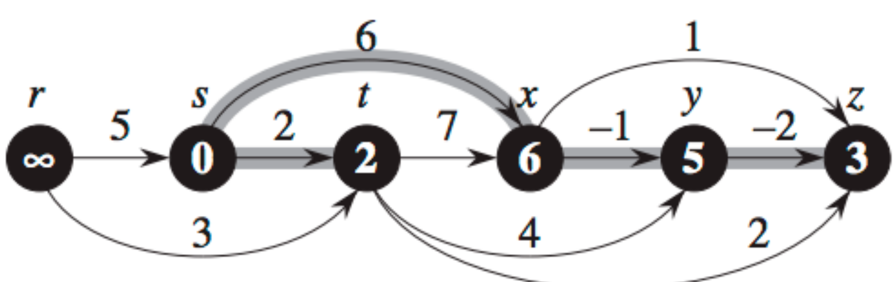

6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
∞
∞
∞
∞
3
4
2
(b)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
∞
∞
3
4
2
(c)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
6
4
3
4
2
(d)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
4
3
4
2
(e)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
3
3
4
2
(f)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
3
3
4
2
(g)

All-pairs shortest paths

Given a weighted digraph $G = (V, E)$ with weight function $w: E \rightarrow R$ (R is the set of real numbers),

determine the length of the shortest path (i.e., distance) between all pairs of vertices in G.

- Nonnegative edge weights: |V| times of Dijkstra's algorithm: $O(VE + V^2 \lg V)$
- Unweighted graphs: |V| times of BFS: $O(VE + V^2)$
- General case: Bellman-Ford algorithm: $O(V^2E)$.
  - Dense graph (V^2 edges) $\Rightarrow \Theta(V^4)$ time in the worst case.
- Better approach? Floyd-Warshall algorithm

Input and Output Formats

To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.

Assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

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

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ where d_{ij} is the length of the shortest path from vertex i to j.

The Floyd-Warshall Algorithm

Dynamic Programming Approach: The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path.

Definition: The vertices $v_2, v_3, ..., v_{l-1}$ are called the *intermediate vertices* of the path $p = \langle v_1, v_2, ..., v_{l-1}, v_l \rangle$.

Decomposition with Intermediate Vertices

Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.

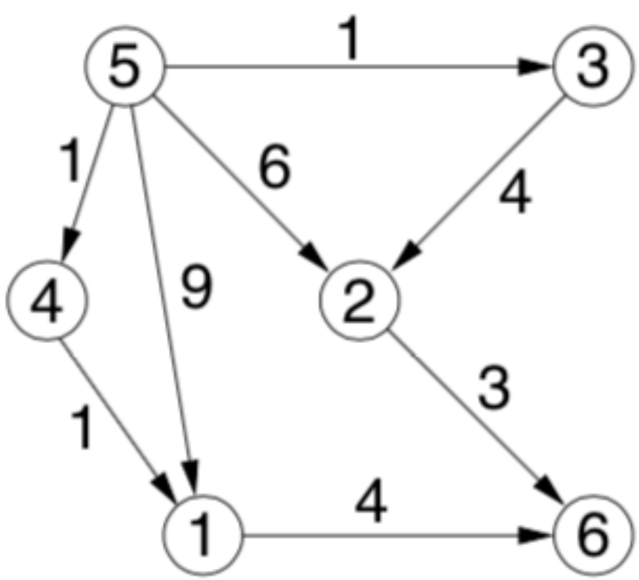
$d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.
Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

Claim: $d_{ij}^{(n)}$ is the distance from i to j. So our aim is to compute $D^{(n)}$.

Subproblems: compute $D^{(k)}$ for $k = 0, 1, \cdots, n$.

Decomposition with Intermediate Vertices

Example: how the value of $d_{5,6}^{(k)}$ changes as k varies



$d_{5,6}^{(0)}$ = INF (no path)

$d_{5,6}^{(1)} = 13$ (5,1,6)

$d_{5,6}^{(2)} = 9$ (5,2,6)

$d_{5,6}^{(3)} = 8$ (5,3,2,6)

$d_{5,6}^{(4)} = 6$ (5,4,1,6)

Structure of shortest paths

Observation 1: A shortest path does not contain the same vertex twice.

Proof: A path containing the same vertex twice contains a cycle. Removing cycle gives a shorter path.

Observation 2: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set {1,2,...,k}, there are two possibilities:

1. k is not a vertex on the path,
   The shortest such path has length $d_{ij}^{(k-1)}$.
2. k is a vertex on the path.
   The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Structure of shortest paths

Consider a shortest path from i to j containing the vertex k. It consists of a subpath from i to k and a subpath from k to j.
Each subpath can only contain intermediate vertices in $\{1, \ldots, k-1\}$, and must be as short as possible, namely they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

Hence the path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Combining the two cases we get

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

The Bottom-up Computation

A recursive solution to the all-pairs shortest-paths problem:

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

- Bottom: $D^{(0)} = [w_{ij}]$, the weight matrix.

- Compute $D^{(k)}$ from $D^{(k-1)}$ using

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

for $k = 1, \ldots, n$.

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$

$D^{(0)} = W$

for $k = 1$ **to** n dynamic programming

    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 $D^{(n)}$

The algorithm's running time is clearly $\Theta(n^3)$

Extracting Shortest Paths

To extract the final shortest paths, we compute the predecessor matrices: $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$ and $\Pi = \Pi^{(n)}$ where $\pi_{ij}^{(k)}$ is the predecessor of vertex j on the shortest path from vertex i with all intermediate vertices in the set of {1,2, …, k)

Compute Predecessor matrices

- Initialization:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases}$$

- Update:

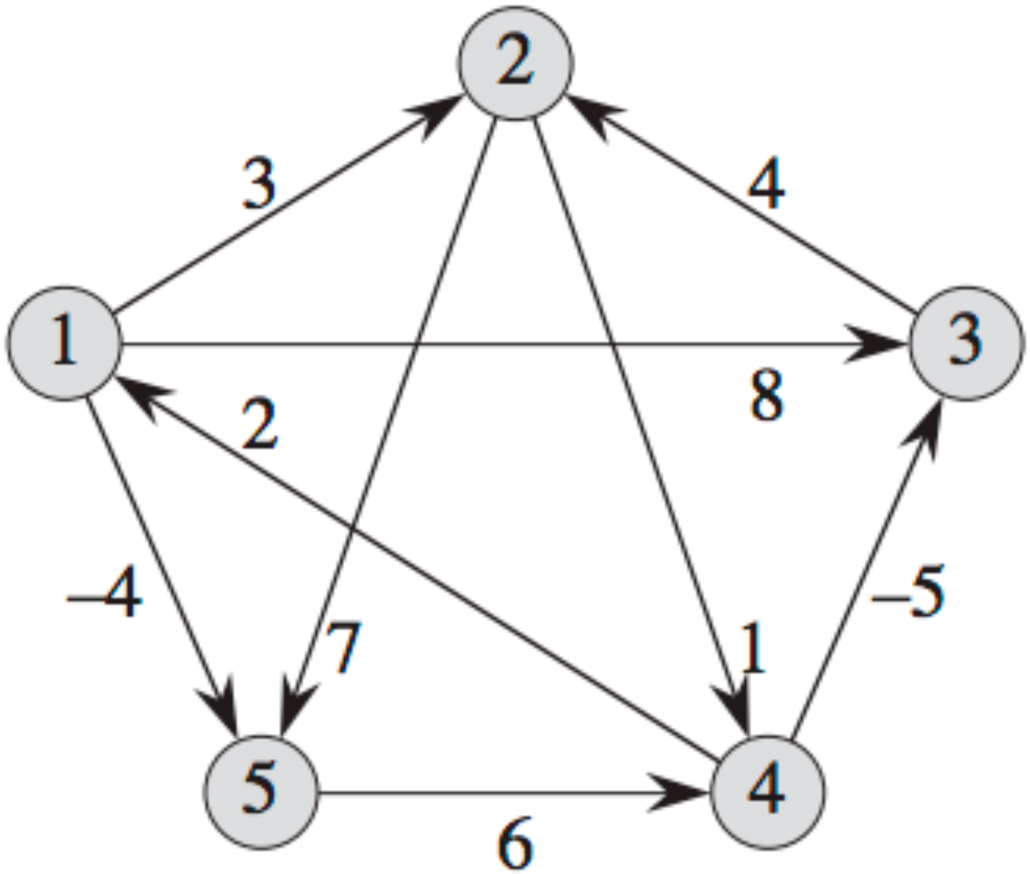
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

- Termination: $pred(i,j) \triangleq \pi_{ij} = \pi_{ij}^{(n)}$

Extracting the Shortest Paths

Path(i, j)
{
    if ($pred(i, j) = i$)   single edge
        output (i, j);
    else   compute the two parts of the path
    {
        Path($i, pred[i, j]$);
        Path($pred[i, j], j$);
    }
}

Example of Floyd-Warshall



Example of Floyd-Warshall

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

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

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

Example of Floyd-Warshall

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

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

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

Transitive Closure of a Directed Graph

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

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

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

Transitive Closure of a Directed Graph

Another Solution:

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

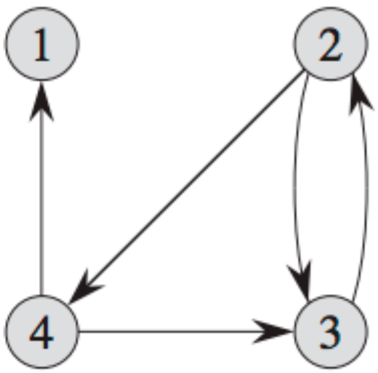
Recurrence:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

and for $k \geq 1$,

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

Example of Transitive Closure



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

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

Next Lecture

- Introduction to NP-Completeness
- Final Review