Design and Analysis of Algorithms
Part IV: Graph Algorithms
Lecture 12: Single-Source Shortest Paths Problem

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

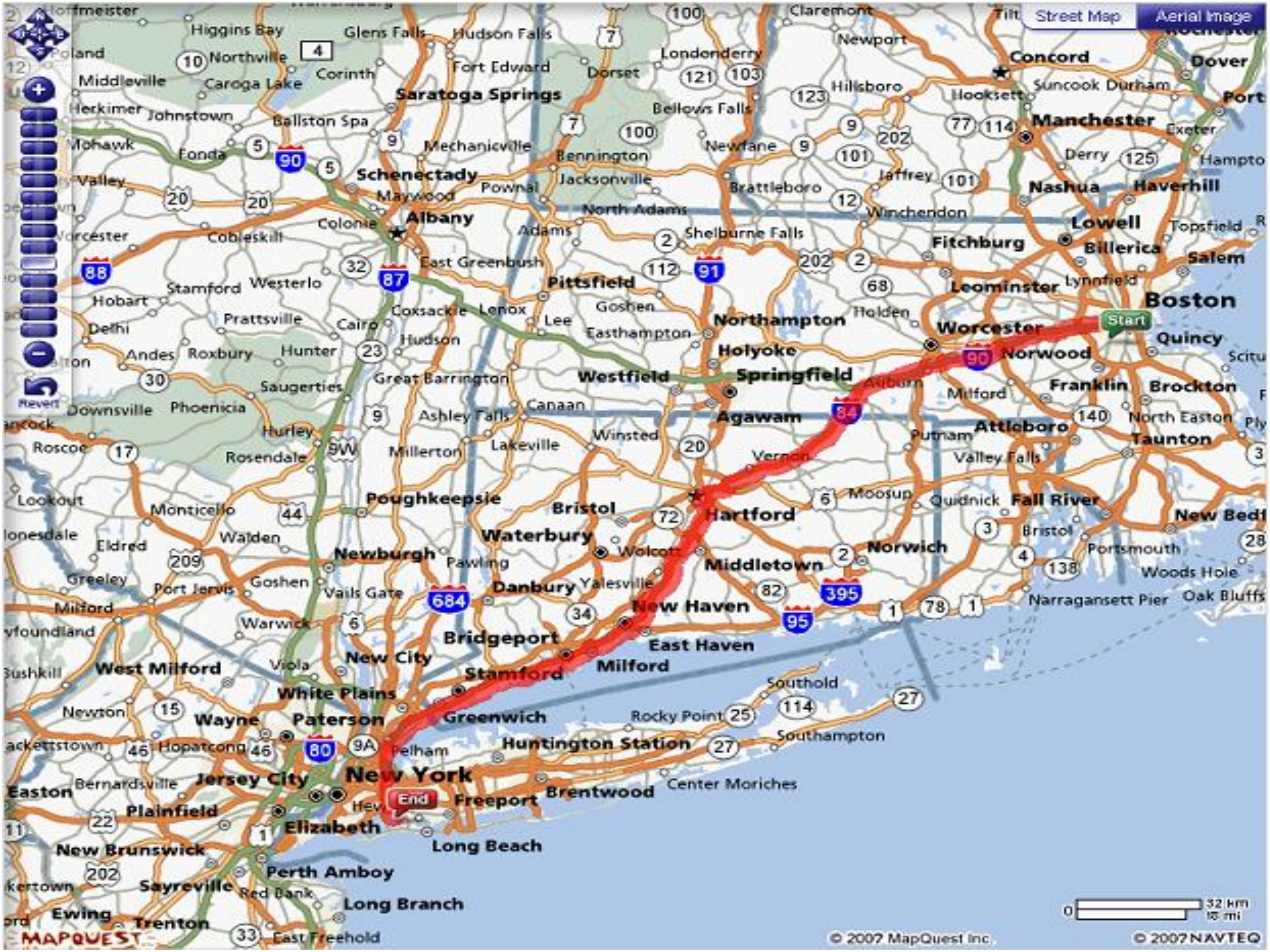
Introduction to Part IV

- In Part IV, we will illustrate several graph algorithm problems using several examples:
  - Basic Concepts of Graphs (图的基本概念)
  - Breadth-First Search [BFS] (广度优先搜索)
  - Depth-First Search [DFS] (深度优先搜索)
  - Topological Sort (拓扑排序)
  - Strongly Connected Components (强联通分量)
  - Minimum Spanning Trees (最小生成树)
  - Single-Source Shortest Paths (单源最短路径)
  - All-Pairs Shortest Paths (所有结点对的最短路径)
  - Maximum/Network Flows (最大流/网络流)

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

Single-Source Shortest Paths Problem



Shortest Path Problem for Weighted Graphs

- Let G = (V, E) be a weighted digraph (directed graph),

Shortest Path Problem for Weighted Graphs

- Let G = (V, E) be a weighted digraph (directed graph), with weight function $\mathrm{w}: \mathrm{E} \longmapsto \mathbb{R}$ mapping edges to real-valued weights

Shortest Path Problem for Weighted Graphs

- Let G = (V, E) be a weighted digraph (directed graph), with weight function $w: E \longmapsto \mathbb{R}$ mapping edges to real-valued weights
- If e = (u, v), we write w(u, v) for w(e).

Shortest Path Problem for Weighted Graphs

- Let G = (V, E) be a weighted digraph (directed graph), with weight function $w\colon E \longmapsto \mathbb{R}$ mapping edges to real-valued weights
- If e = (u, v), we write w(u, v) for w(e).

Definition

The length of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

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

Distance

Definition

The distance from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v;

Distance

Definition

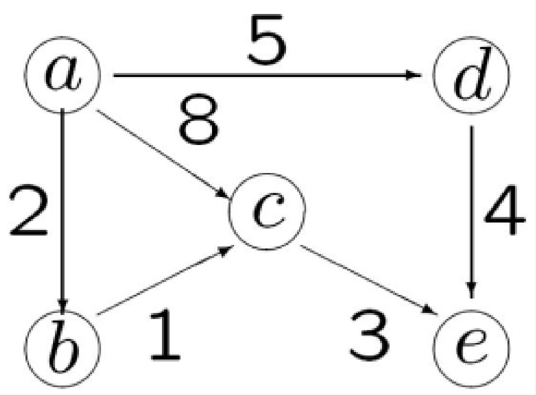
The distance from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v; and is ∞ otherwise.

Distance

Definition

The distance from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v; and is ∞ otherwise.

Example



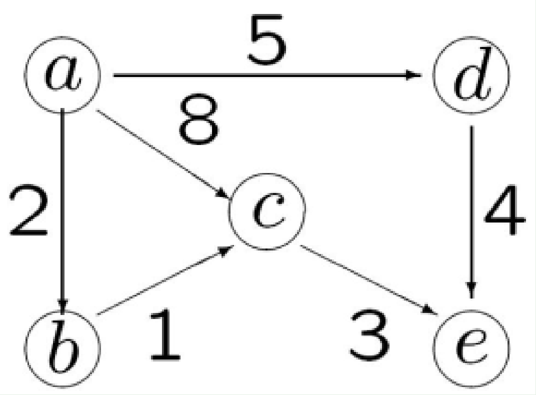
- $\text{length}(\langle a, b, c, e \rangle) =$

Distance

Definition

The distance from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v; and is ∞ otherwise.

Example



- length($\langle a, b, c, e\rangle$) $= 6$; distance from a to e is 6

Single-Source Shortest-Paths Problem

- Definition: Single-source shortest-paths problem
  - Given a digraph $G = (V, E)$ with no-negative edge weights w and a designated source vertex $s \in V$, determine the distance and a shortest path from the source vertex to every vertex in the digraph.

Single-Source Shortest-Paths Problem

- Definition: Single-source shortest-paths problem
  - Given a digraph $G = (V, E)$ with no-negative edge weights w and a designated source vertex $s \in V$, determine the distance and a shortest path from the source vertex to every vertex in the digraph.
  - Note: Weight between two vertices u, v can actually be **negative**.

Single-Source Shortest-Paths Problem

- Definition: Single-source shortest-paths problem
  - Given a digraph $G = (V, E)$ with no-negative edge weights w and a designated source vertex $s \in V$, determine the distance and a shortest path from the source vertex to every vertex in the digraph.
  - Note: Weight between two vertices u, v can actually be **negative**.

Question

How do you design an efficient algorithm for this problem?

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

The Rough Idea of Dijkstra’s Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.
  - S ⊆ V is a subset of vertices for which we know the true distance, that is d[v] = δ(s, v).

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.
  - $S \subseteq V$ is a subset of vertices for which we know the true distance, that is d[v] = δ(s, v).
- Initially
  - S =

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.
  - $S \subseteq V$ is a subset of vertices for which we know the true distance, that is d[v] = δ(s, v).
- Initially
  - $S = \emptyset$
  - d[s] =

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length $\delta(s, v)$ of the shortest path for each vertex v.
  - $S \subseteq V$ is a subset of vertices for which we know the true distance, that is $d[v] = \delta(s, v)$.
- Initially
  - $S = \emptyset$
  - $d[s] = 0$ and $d[v] = \infty$ for all others vertices v.

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.
  - S ⊆ V is a subset of vertices for which we know the true distance, that is d[v] = δ(s, v).
- Initially
  - S = ∅
  - d[s] = 0 and d[v] = ∞ for all others vertices v.
- One by one we select vertices from V \ S to add to S.

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length δ(s, v) of the shortest path for each vertex v.
  - S ⊆ V is a subset of vertices for which we know the true distance, that is d[v] = δ(s, v).
- Initially
  - S = ∅
  - d[s] = 0 and d[v] = ∞ for all others vertices v.
- One by one we select vertices from V \ S to add to S.
- Questions to answer at each step:

The Rough Idea of Dijkstra's Algorithm

- Maintain d[v] and S:
  - d[v] is an upper bound of the length $\delta(s, v)$ of the shortest path for each vertex v.
  - $S \subseteq V$ is a subset of vertices for which we know the true distance, that is $d[v] = \delta(s, v)$.
- Initially
  - $S = \emptyset$
  - $d[s] = 0$ and $d[v] = \infty$ for all others vertices v.
- One by one we select vertices from $V \setminus S$ to add to S.
- Questions to answer at each step:
  - Which vertex do we select?
  - How do we update the distance upper bounds after a vertex is added to S?

Selection of Vertex

Question

How does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Selection of Vertex

Question

How does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm.

Selection of Vertex

Question

How does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm.

- For each vertex in u ∈ V \ S, we have computed a distance upper bound d[u].

Selection of Vertex

Question

How does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm.

- For each vertex in $u \in V \setminus S$, we have computed a distance upper bound d[u].
- The next vertex processed is always a vertex $u \in V \setminus S$ for which d[u] is minimum

Selection of Vertex

Question

How does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm.

- For each vertex in u ∈ V \ S, we have computed a distance upper bound d[u].
- The next vertex processed is always a vertex u ∈ V \ S for which d[u] is minimum
  - that is, we take the unprocessed vertex that is closest (by our estimate) to s.

Updating Distance Estimates

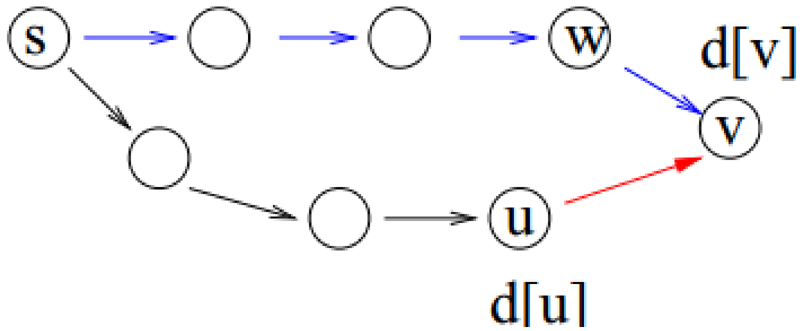
- Current distance upper bound for v: d[v].

Updating Distance Estimates

- Current distance upper bound for v: d[v].
- Vertex u just added to S. Edge (u, v) with weight w(u, v).

Updating Distance Estimates

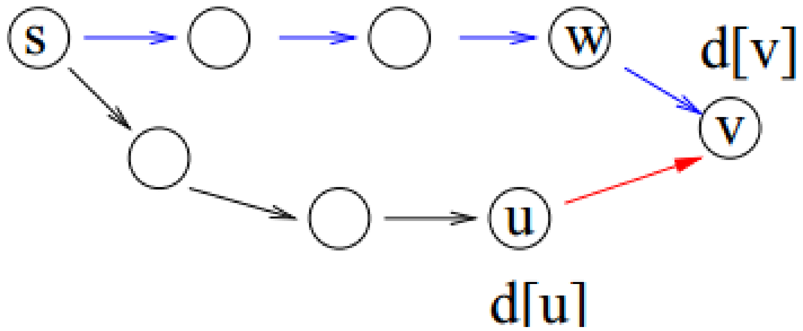
- Current distance upper bound for v: d[v].
- Vertex u just added to S. Edge (u, v) with weight w(u, v).
- How do we update d[v]?



- Current shortest path to v: <s, …, w, v>, length d[v].

Updating Distance Estimates

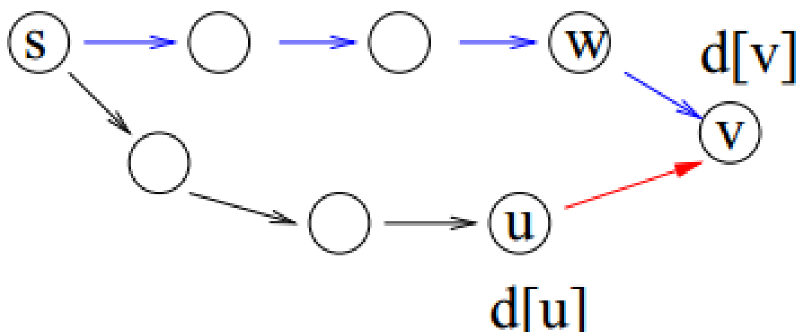
- Current distance upper bound for v: d[v].
- Vertex u just added to S. Edge (u, v) with weight w(u, v).
- How do we update d[v]?



- Current shortest path to v: <s, …, w, v>, length d[v].
- New path to v: <s, …, u, v>, length d[u] + w(u,v).

Updating Distance Estimates

- Current distance upper bound for v: d[v].
- Vertex u just added to S. Edge (u, v) with weight w(u, v).
- How do we update d[v]?

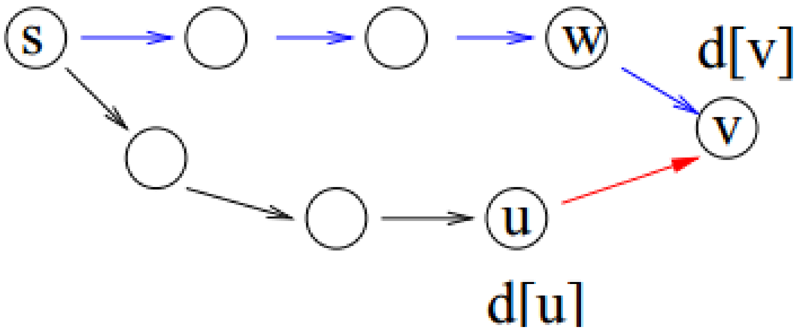


- Current shortest path to v: <s, …, w, v>, length d[v].
- New path to v: <s, …, u, v>, length d[u] + w(u,v).
- If new path is shorter(d[u] + w(u,v) < d[v]), update d[v]

$$d[v] = d[u] + w(u, v)$$

Updating Distance Estimates

- Current distance upper bound for v: d[v].
- Vertex u just added to S. Edge (u, v) with weight w(u, v).
- How do we update d[v]?



- Current shortest path to v: <s, …, w, v>, length d[v].
- New path to v: <s, …, u, v>, length d[u] + w(u,v).
- If new path is shorter(d[u] + w(u,v) < d[v]), update d[v]

$$d[v] = d[u] + w(u, v)$$

Now we have a better (tighter) upper bound for d[v]. This is called relaxing the edge (u, v).

The Algorithm for Relaxing an Edge

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u, v) < d[v]$ **then**

end

The Algorithm for Relaxing an Edge

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u, v) < d[v]$ **then**
  $d[v] \leftarrow$
end

The Algorithm for Relaxing an Edge

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u,v) < d[v]$ **then**
  $d[v] \leftarrow d[u] + w(u,v)$;
  $pred[v] \leftarrow$
end

The Algorithm for Relaxing an Edge

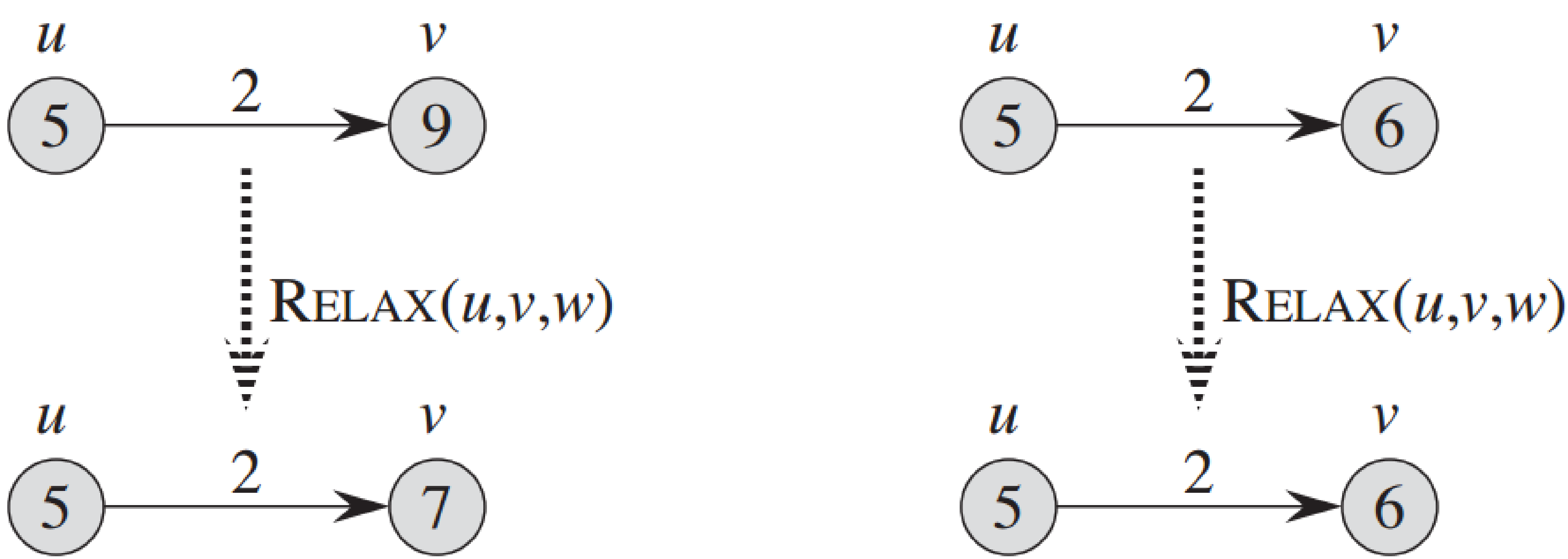
Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u,v) < d[v]$ **then**
  $d[v] \leftarrow d[u] + w(u,v)$;
  $pred[v] \leftarrow u$;
end

The Algorithm for Relaxing an Edge: An Example

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u, v) < d[v]$ **then**
  $d[v] \leftarrow d[u] + w(u, v)$;
  $pred[v] \leftarrow u$;
end



The Algorithm for Relaxing an Edge

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u, v) < d[v]$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$;
    $pred[v] \leftarrow u$;
end

- Remark 1: The predecessor pointer pred[ ] is for determining the shortest paths.
- Remark 2: After edge (u, v) is relaxed, we have

$$d[v] \leq d[u] + w(u, v)$$

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm(Optional)
  - The algorithm
  - Analysis of Bellman-Ford algorithm

The Selection in Dijkstra’s Algorithm

Question

How do we perform this selection efficiently?

The Selection in Dijkstra's Algorithm

Question

How do we perform this selection efficiently?

Answer: We store the vertices of $V \setminus S$ in a

The Selection in Dijkstra's Algorithm

Question

How do we perform this selection efficiently?

Answer: We store the vertices of $V \setminus S$ in a Priority queue, where the key value of each vertex v is

The Selection in Dijkstra's Algorithm

Question

How do we perform this selection efficiently?

Answer: We store the vertices of $V \setminus S$ in a Priority queue, where the key value of each vertex v is d[v].

The Selection in Dijkstra's Algorithm

Question

How do we perform this selection efficiently?

Answer: We store the vertices of V \ S in a Priority queue, where the key value of each vertex v is d[v].

- Note: if we implement the priority queue using a heap, we can perform the operations Insert(), Extract-Min(), and Decrease-Key(), each in O(   ) time.

The Selection in Dijkstra's Algorithm

Question

How do we perform this selection efficiently?

Answer: We store the vertices of V \ S in a Priority queue, where the key value of each vertex v is d[v].

- Note: if we implement the priority queue using a heap, we can perform the operations Insert(), Extract-Min(), and Decrease-Key(), each in O(log n) time.

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

| $d[u] \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

| $d[u] \leftarrow \infty, color[u] \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

| $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow 0;$

$pred[s] \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

| $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow 0;$

$pred[s] \leftarrow$ NULL;

$Q \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow 0$;

$pred[s] \leftarrow$ NULL;

$Q \leftarrow$queue with all vertices;

while *Non-Empty(Q)* **do**

  // Process all vertices

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow 0$;

$pred[s] \leftarrow$ NULL;

$Q \leftarrow$queue with all vertices;

while *Non-Empty(Q)* **do**

  // Process all vertices

  $u \leftarrow$;// Find new vertex

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**
  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize
end
$d[s] \leftarrow 0$;
$pred[s] \leftarrow$ NULL;
$Q \leftarrow$queue with all vertices;
while *Non-Empty*(Q) **do**
  // Process all vertices
  $u \leftarrow$Extract-Min(Q);// Find new vertex

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, source vertex s
Output: None
for u ∈ V do
  | d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
  | // Process all vertices
  | u ←Extract-Min(Q);// Find new vertex
  | for v ∈ Adj[u] do
  |   |
  |   |
  |
```

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**
  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize
end
$d[s] \leftarrow 0$;
$pred[s] \leftarrow$ NULL;
$Q \leftarrow$queue with all vertices;
while *Non-Empty(Q)* **do**
  // Process all vertices
  $u \leftarrow$Extract-Min(Q);// Find new vertex
  for $v \in Adj[u]$ **do**
    if $d[u] + w(u, v) < d[v]$ **then**
      // If estimate improves

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**

  $d[u] \leftarrow \infty, color[u] \leftarrow$ WHITE;// Initialize

end

$d[s] \leftarrow 0$;

$pred[s] \leftarrow$ NULL;

$Q \leftarrow$queue with all vertices;

while *Non-Empty(Q)* **do**

  // Process all vertices

  $u \leftarrow$Extract-Min(Q);// Find new vertex

  for $v \in Adj[u]$ **do**

    if $d[u] + w(u, v) < d[v]$ **then**

      // If estimate improves

      $d[v] \leftarrow$

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, source vertex s
Output: None
for u ∈ V do
  | d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
  | // Process all vertices
  | u ←Extract-Min(Q);// Find new vertex
  | for v ∈ Adj[u] do
  |   | if d[u] + w(u, v) < d[v] then
  |   |   | // If estimate improves
  |   |   | d[v] ← d[u] + w(u, v);// relax
```

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

```
for u ∈ V do
    d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
    // Process all vertices
    u ←Extract-Min(Q);// Find new vertex
    for v ∈ Adj[u] do
        if d[u] + w(u, v) < d[v] then
            // If estimate improves
            d[v] ← d[u] + w(u, v);// relax
            Decrease-Key(Q, v, d[v]);
```

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, source vertex s
Output: None
for u ∈ V do
  | d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
  | // Process all vertices
  | u ←Extract-Min(Q);// Find new vertex
  | for v ∈ Adj[u] do
  |   | if d[u] + w(u, v) < d[v] then
  |   |   | // If estimate improves
  |   |   | d[v] ← d[u] + w(u, v);// relax
  |   |   | Decrease-Key(Q, v, d[v]);
  |   |   | pred[v] ←
```

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

```
for u ∈ V do
    d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
    // Process all vertices
    u ←Extract-Min(Q);// Find new vertex
    for v ∈ Adj[u] do
        if d[u] + w(u, v) < d[v] then
            // If estimate improves
            d[v] ← d[u] + w(u, v);// relax
            Decrease-Key(Q, v, d[v]);
            pred[v] ← u;
        end
    end
    color[u] ←
```

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, source vertex s
Output: None
for u ∈ V do
 | d[u] ← ∞, color[u] ← WHITE;// Initialize
end
d[s] ← 0;
pred[s] ← NULL;
Q ←queue with all vertices;
while Non-Empty(Q) do
    // Process all vertices
    u ←Extract-Min(Q);// Find new vertex
    for v ∈ Adj[u] do
        if d[u] + w(u, v) < d[v] then
            // If estimate improves
            d[v] ← d[u] + w(u, v);// relax
            Decrease-Key(Q, v, d[v]);
            pred[v] ← u;
        end
    end
    color[u] ←BLACK;
end
```

An Example of Dijkstra's Algorithm

color

W	W	W	W	W

pred

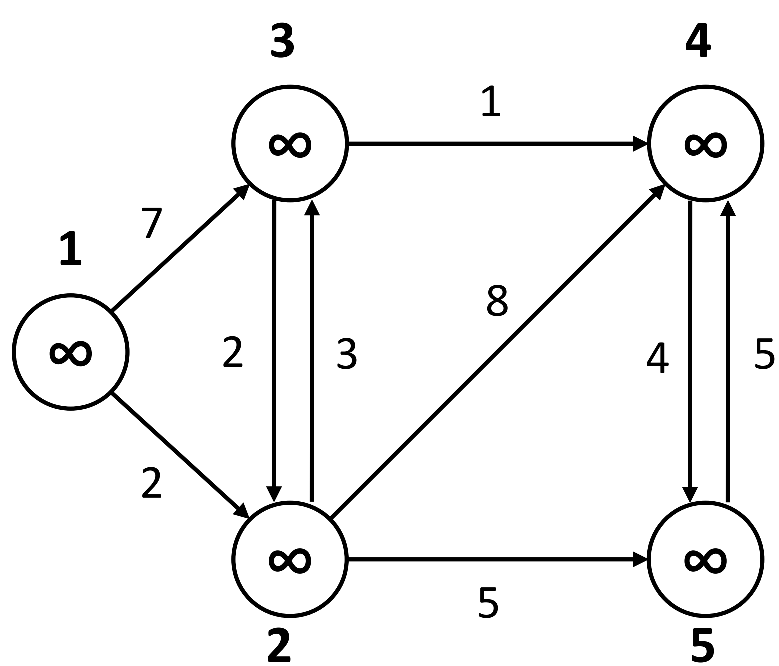
N	N	N	N	N

d

∞	∞	∞	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

W	**W**	**W**	**W**	**W**

pred

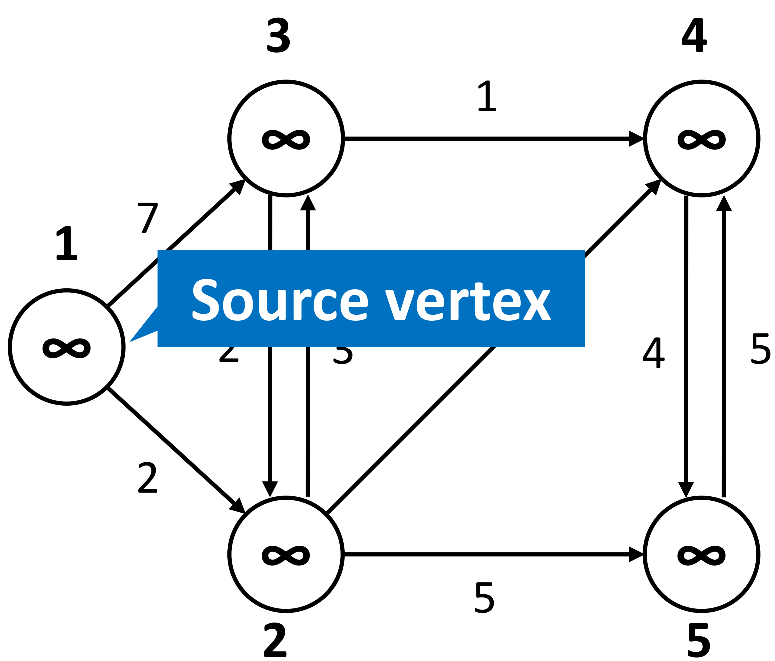
N	**N**	**N**	**N**	**N**

d

∞	**∞**	**∞**	**∞**	**∞**

Q(Priority Queue)

1	**2**	**3**	**4**	**5**



An Example of Dijkstra's Algorithm

color

W	W	W	W	W

d

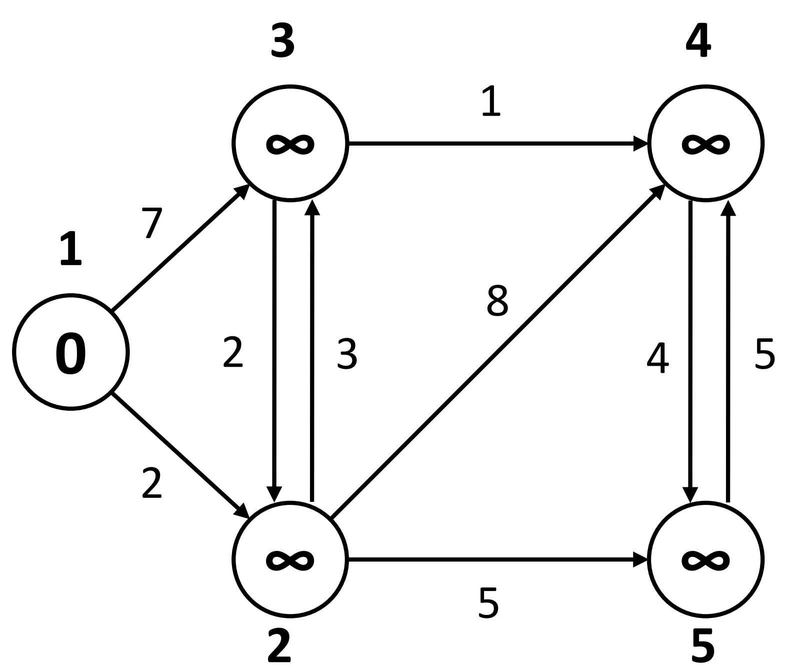
0	∞	∞	∞	∞

pred

N	N	N	N	N

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

W	W	W	W	W

pred

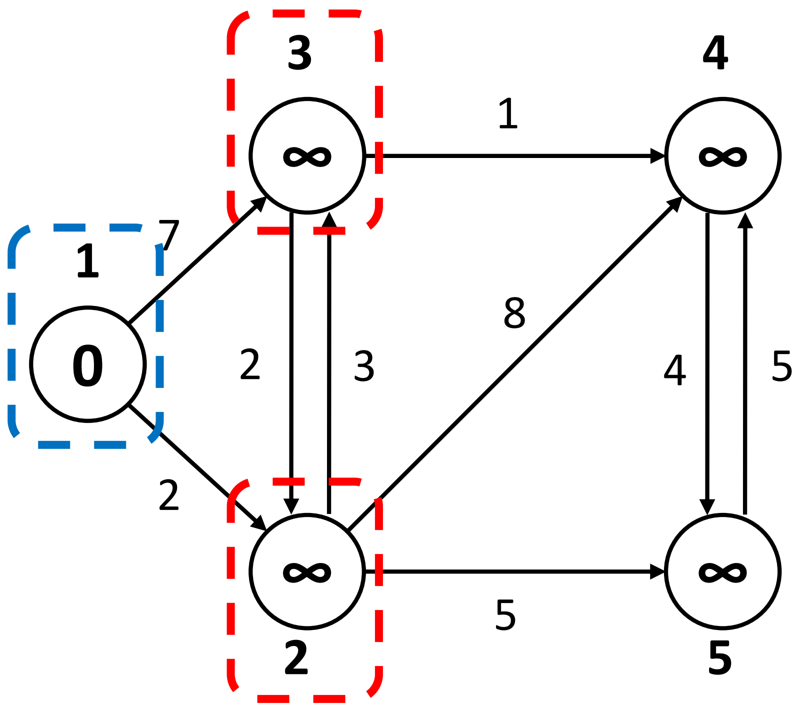
N	N	N	N	N

d

0	∞	∞	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

W	W	W	W	W

pred

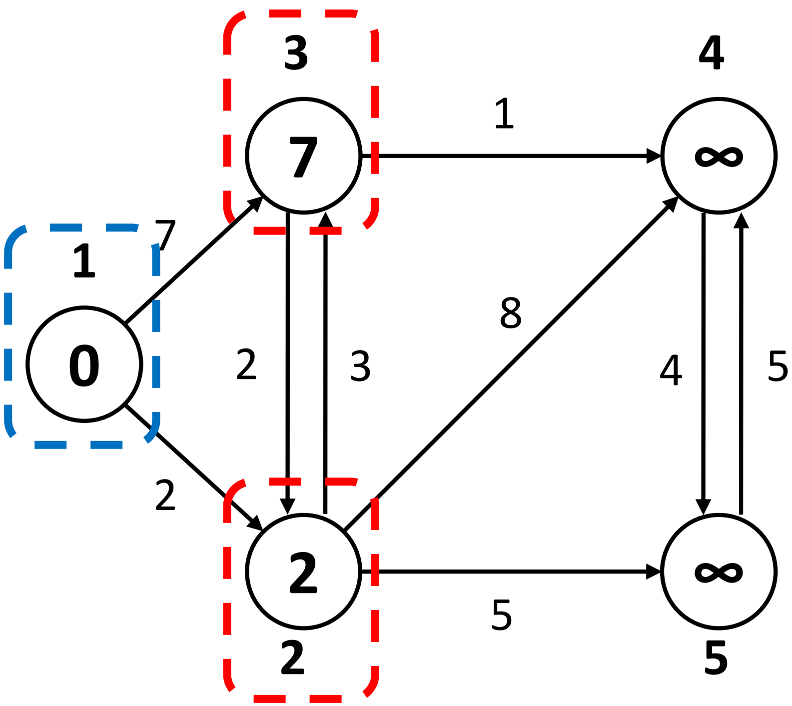
N	1	1	N	N

d

0	2	7	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

pred

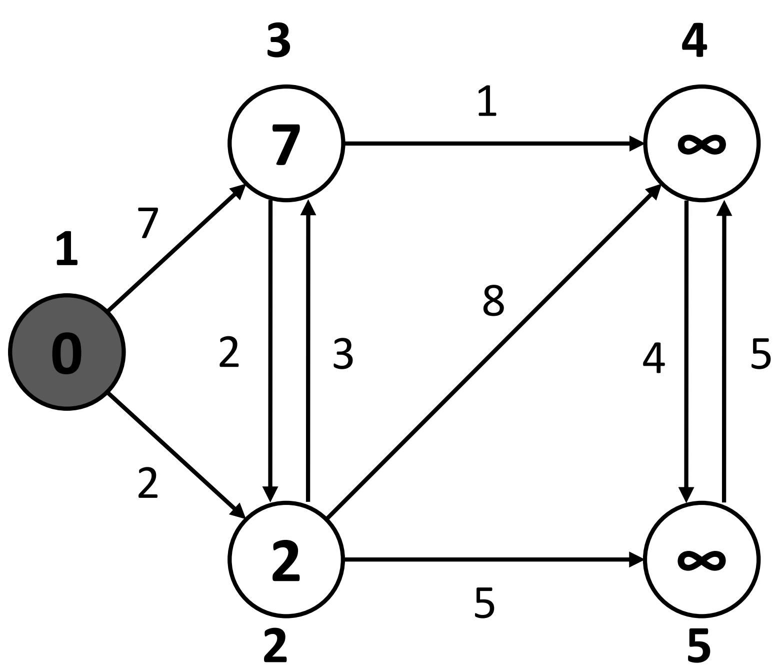
N	1	1	N	N

d

0	2	7	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

pred

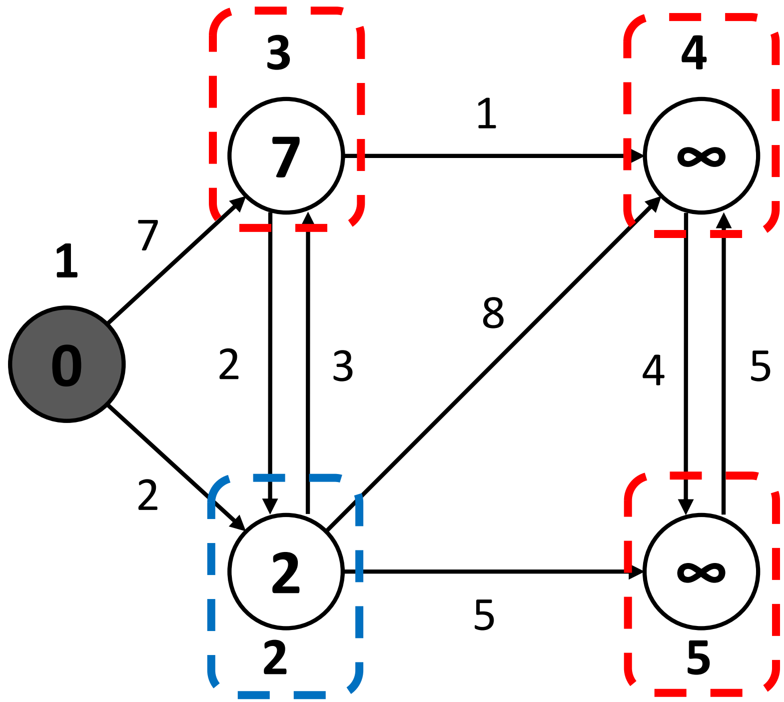
N	1	1	N	N

d

0	2	7	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

pred

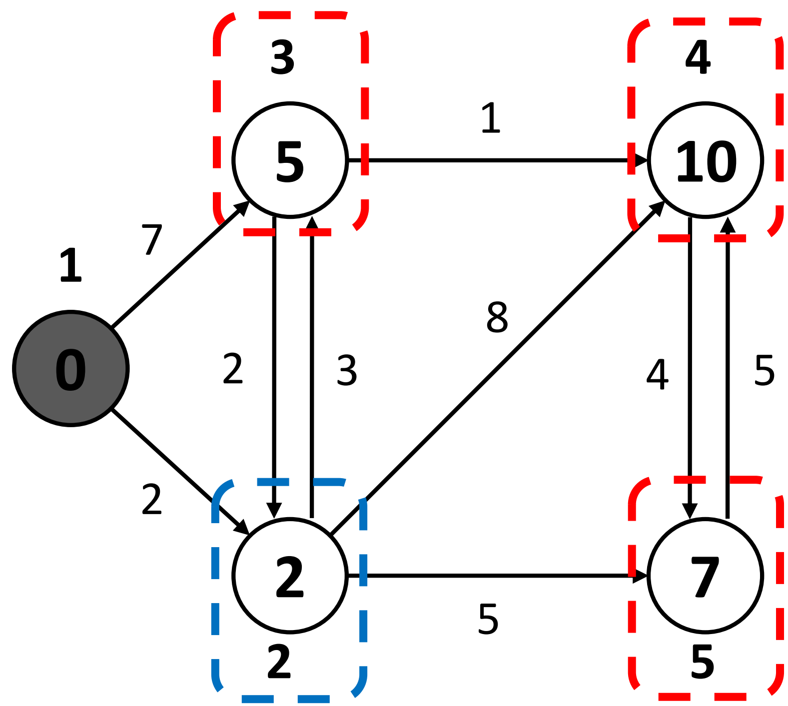
N	1	2	2	2

d

0	2	5	10	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

d

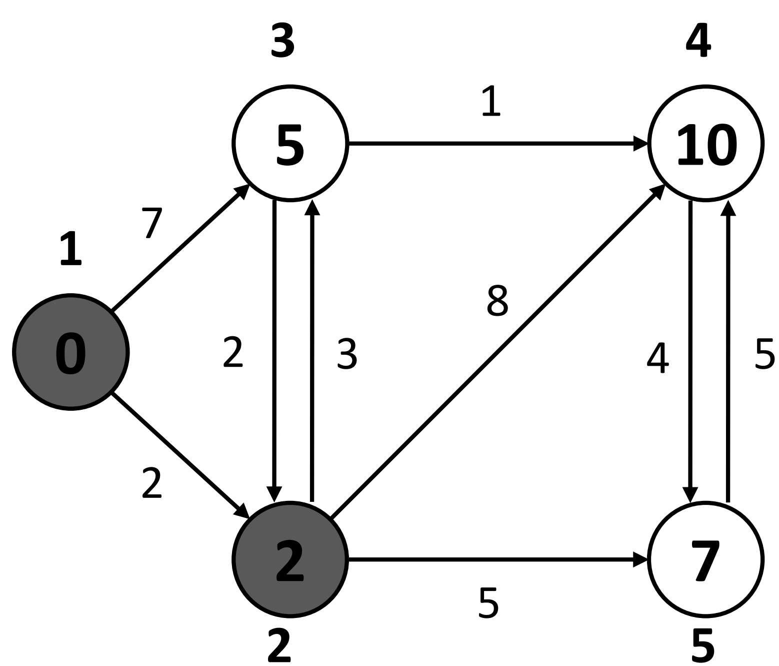
0	2	5	10	7

pred

N	1	2	2	2

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

pred

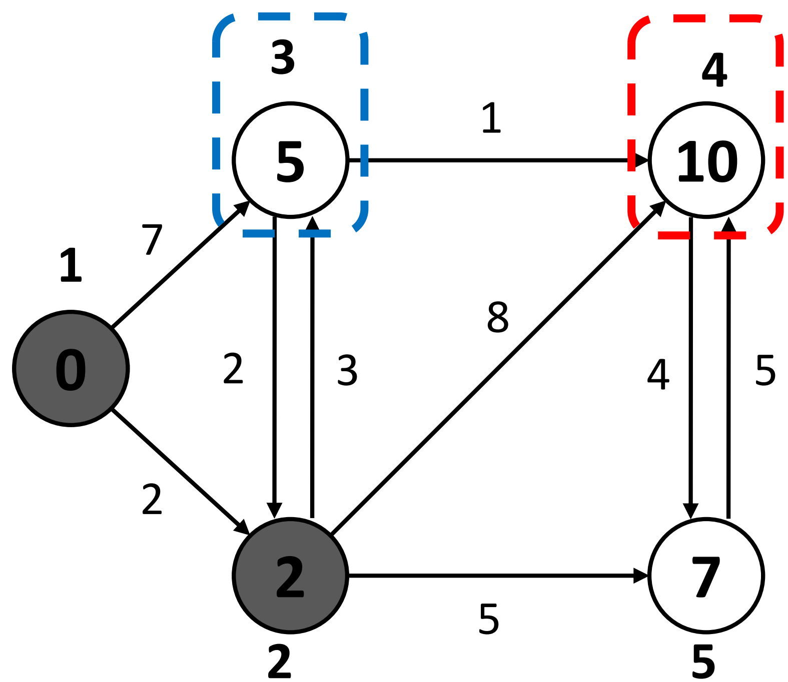
N	1	2	2	2

d

0	2	5	10	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

pred

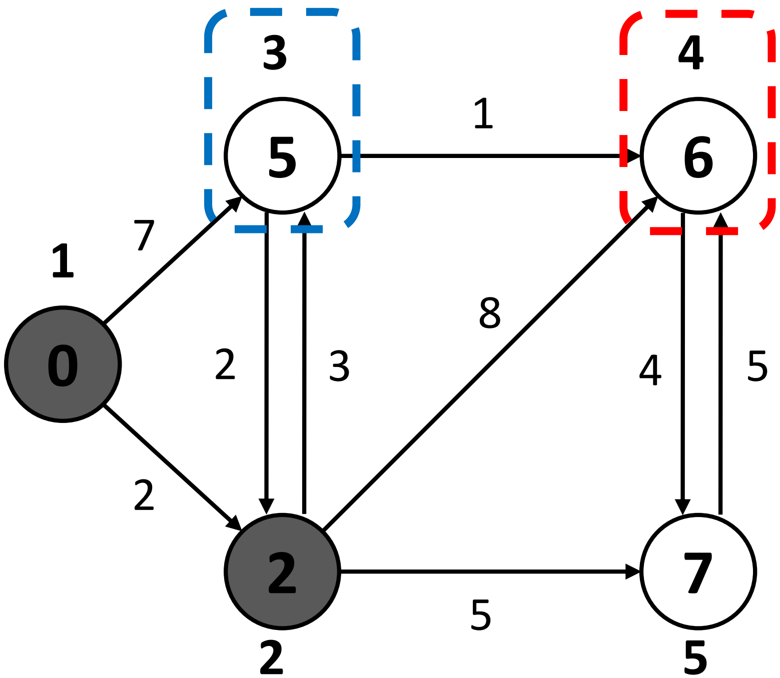
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	W	W

d

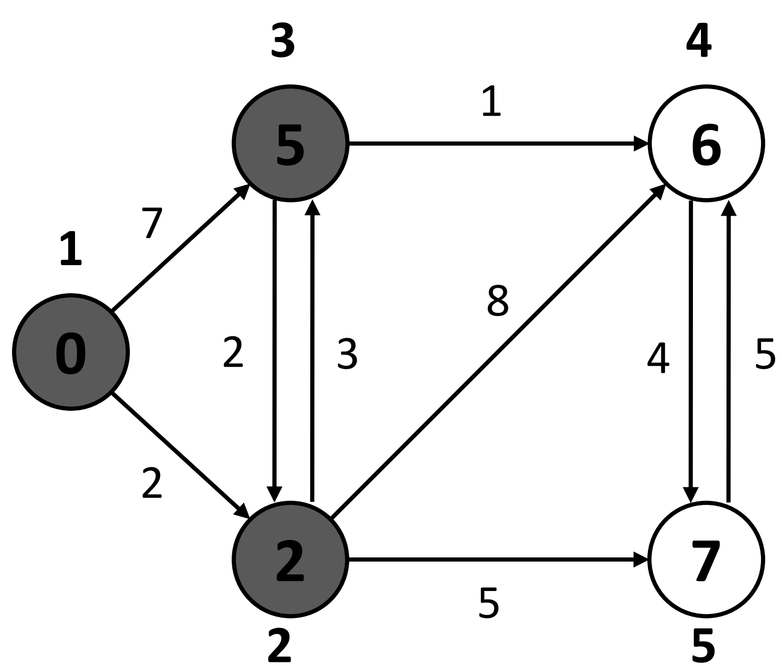
0	2	5	6	7

pred

N	1	2	3	2

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	W	W

d

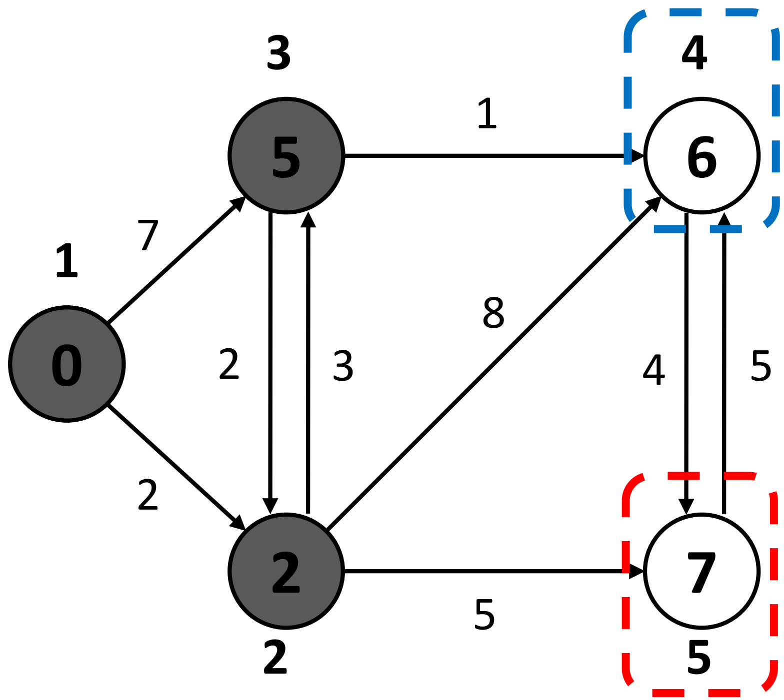
0	2	5	6	7

pred

N	1	2	3	2

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	**B**	**B**	**W**	**W**

pred

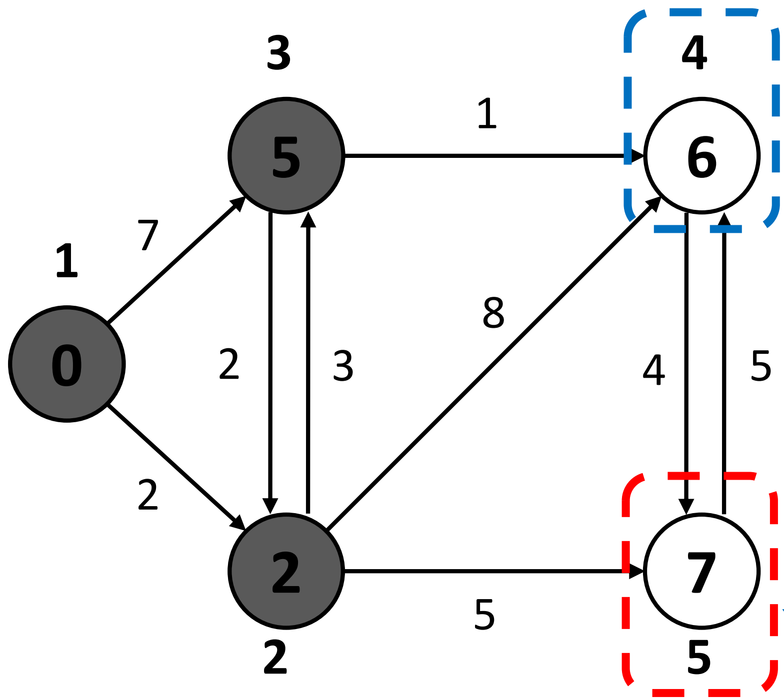
N	**1**	**2**	**3**	**2**

d

0	**2**	**5**	**6**	**7**

Q(Priority Queue)

1	**2**	**3**	**4**	**5**



d[5] is not needed to be updated because d[4]+4 > d[5]

An Example of Dijkstra's Algorithm

color

B	B	B	B	W

pred

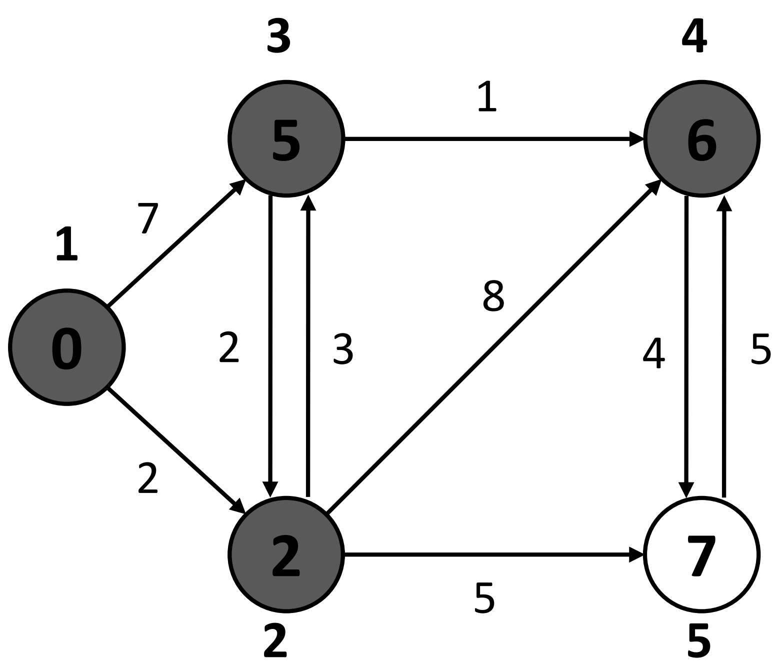
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	B	W

d

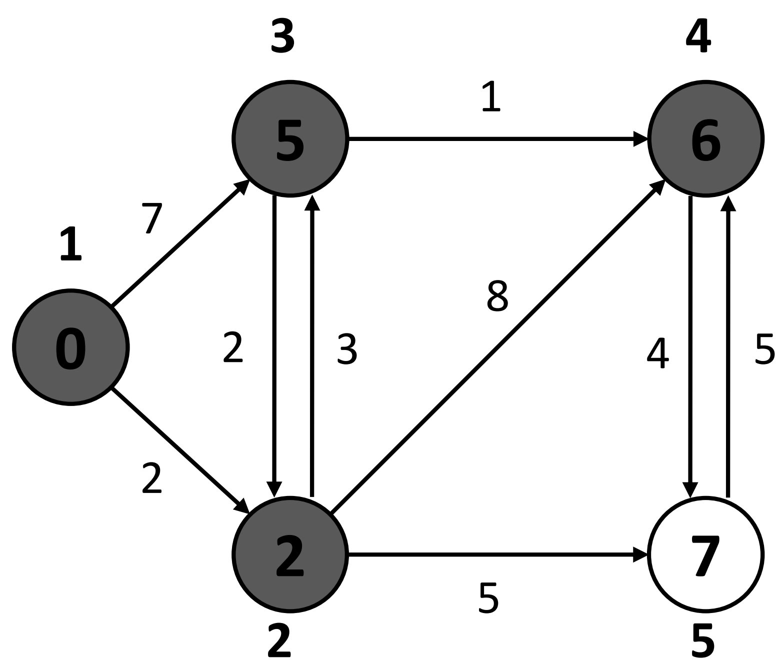
0	2	5	6	7

pred

N	1	2	3	2

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	B	B

pred

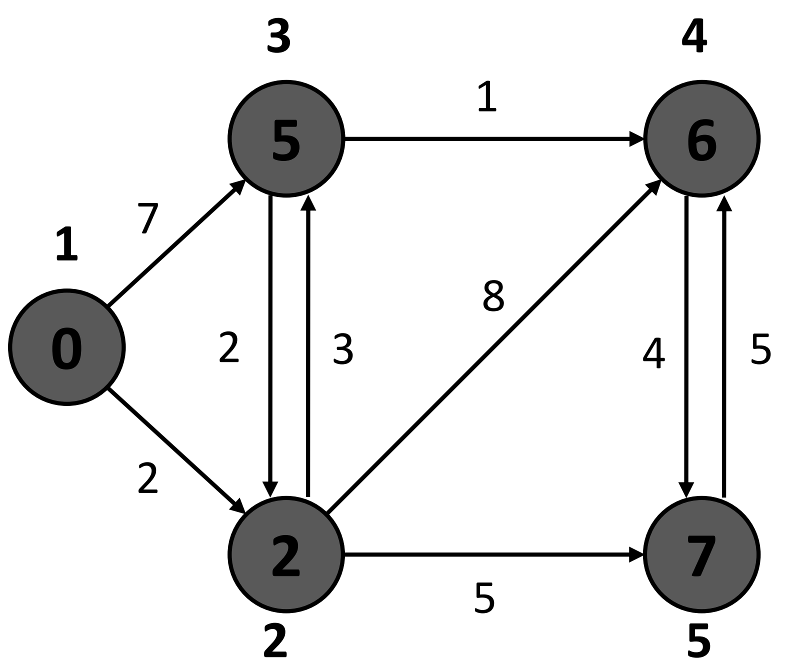
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	B	B

pred

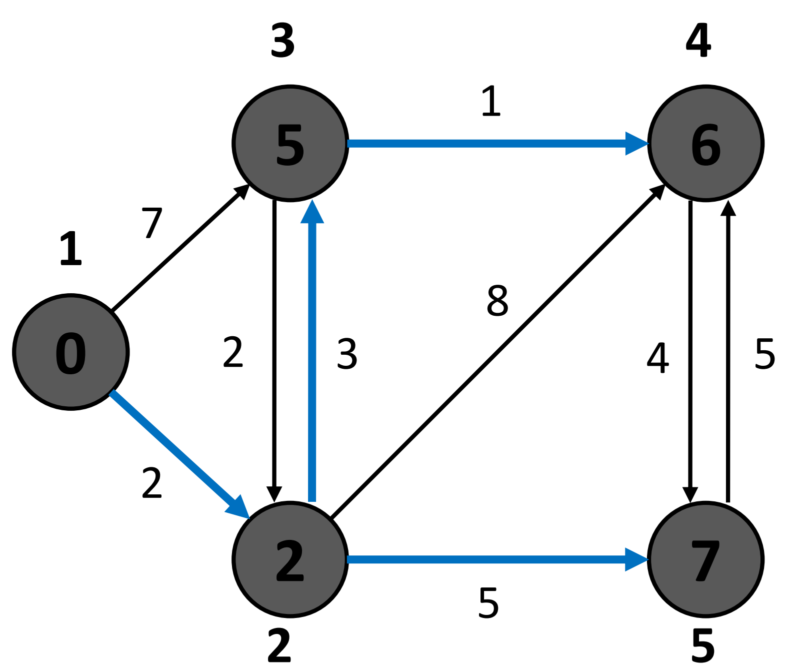
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



Shortest Path Tree for Dijkstra's Algorithm

Shortest Path Tree: T = (V ; A), where

$$A = \{(pred[v], v) | v \in V \backslash \{s\}\}$$

Shortest Path Tree for Dijkstra's Algorithm

Shortest Path Tree: T = (V ; A), where

$$A = \{(pred[v], v) | v \in V \backslash \{s\}\}$$

The array pred[v] is used to build the tree.

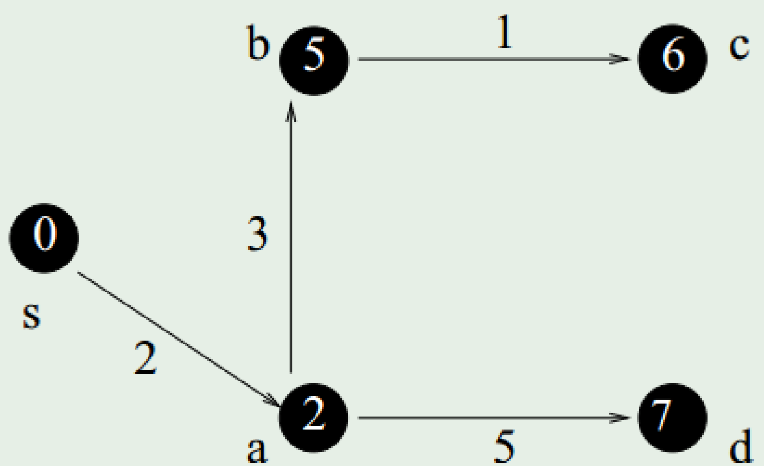
Shortest Path Tree for Dijkstra's Algorithm

Shortest Path Tree: T = (V ; A), where

$$A = \{(pred[v], v) | v \in V \backslash \{s\}\}$$

The array pred[v] is used to build the tree.

Example



v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	NIL	s	a	b	a

Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

Observation

Lemma

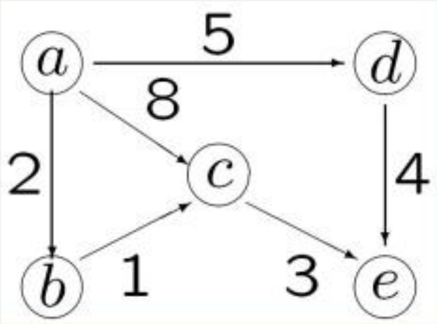
Any sub-path of a shortest path must also be a shortest path

Observation

Lemma

Any sub-path of a shortest path must also be a shortest path

Example



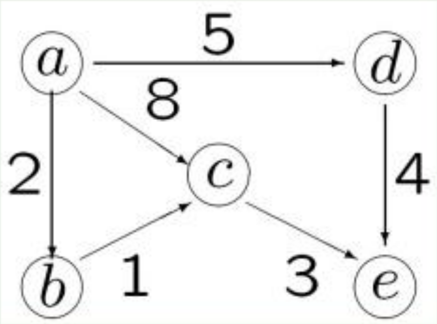
$\langle a, b, c, e \rangle$ is a shortest path;

Observation

Lemma

Any sub-path of a shortest path must also be a shortest path

Example



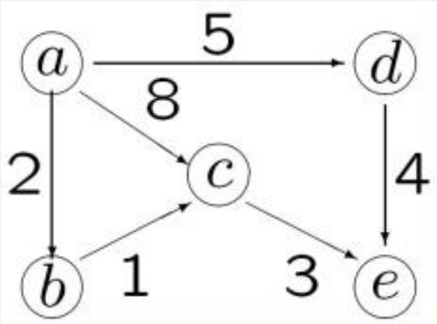
$\langle a, b, c, e \rangle$ is a shortest path; sub-path $\langle a, b, c \rangle$ is also a shortest path.

Observation

Lemma

Any sub-path of a shortest path must also be a shortest path

Example



$\langle a, b, c, e \rangle$ is a shortest path; sub-path $\langle a, b, c \rangle$ is also a shortest path.

Question

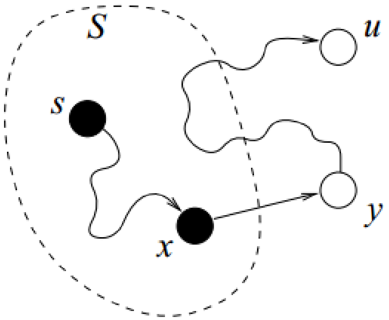
Why?

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

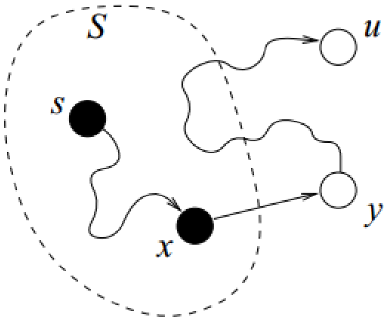


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

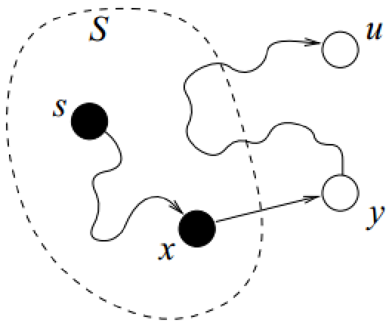


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s, u)$.

Correctness of Dijkstra’s Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

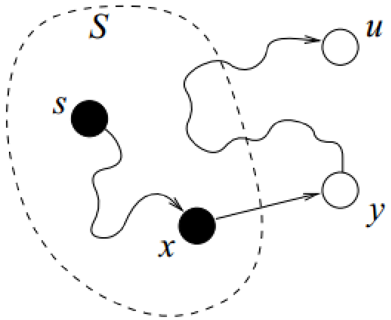


Proof:

- Suppose to the contrary that at some point Dijkstra’s algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s, u)$.
- Consider the situation just prior to the insertion of u.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

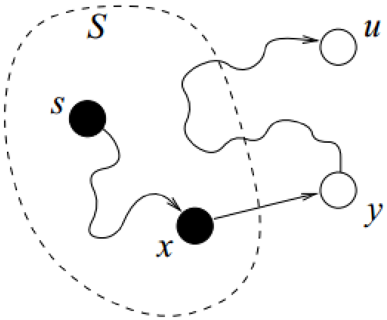


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s, u)$.
- Consider the situation just prior to the insertion of u.
- Consider the true shortest path from s to u.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

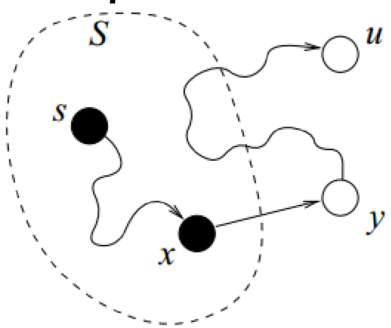


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s, u)$.
- Consider the situation just prior to the insertion of u.
- Consider the true shortest path from s to u.
- Because $s \in S$ and u $\in V \backslash S$, at some point this path must

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s,u)$.

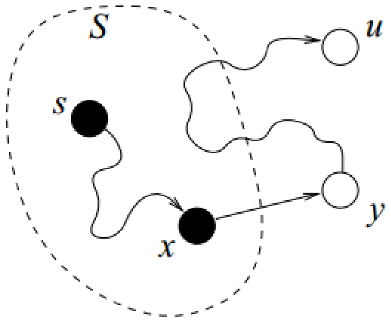


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s,u)$.
- Consider the situation just prior to the insertion of u.
- Consider the true shortest path from s to u.
- Because $s \in S$ and u $\in V \backslash S$, at some point this path must first take a jump out of S.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

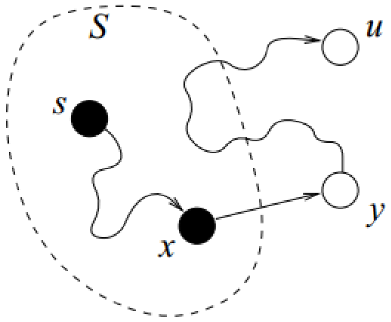


Proof:

- Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s, u)$.
- Since $d[u] = \infty$ initially, and remains an upper bound of the true shortest distance, we have $d[u] > \delta(s, u)$.
- Consider the situation just prior to the insertion of u.
- Consider the true shortest path from s to u: <s,..., x, y,..., u>.
- Because $s \in S$ and u ∈ $V \backslash S$, at some point this path must first take a jump out of S. Let (x, y) be the edge taken by the path, where $x \in S$ and y ∈ $V \backslash S$ (it may be that x=s or/and y=u).

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.

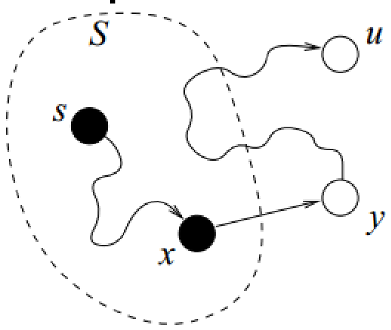


Proof:

We now prove that $y \neq u$.

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



Proof:

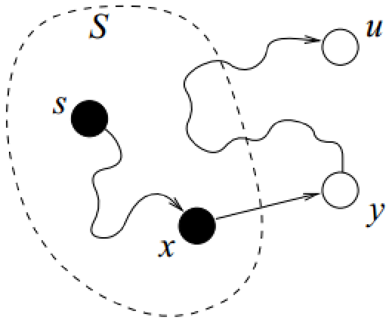
We now prove that $y \neq u$.

- We have done relaxation when processing x, so

$$d[y] \leq d[x] + w(x, y) \quad (1)$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



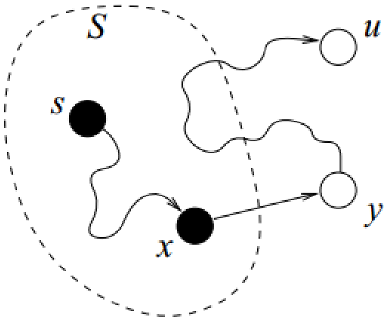
Proof:

We now prove that $y \neq u$.

- We have done relaxation when processing x, so
$$d[y] \leq d[x] + w(x, y) \tag{1}$$
- Since x is added to S earlier, by hypothesis,
$$d[x] = \delta(s, x) \tag{2}$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



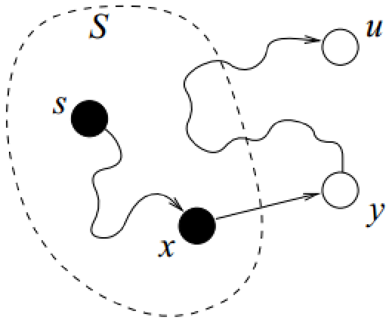
Proof:

We now prove that $y \neq u$.

- We have done relaxation when processing x, so
$$d[y] \leq d[x] + w(x, y) \quad (1)$$
- Since x is added to S earlier, by hypothesis,
$$d[x] = \delta(s, x) \quad (2)$$
- Since <s, …, x, y> is sub-path of a shortest path,
$$\delta(s, y) = \delta(s, x) + w(x, y) = d[x] + w(x, y) \quad (3)$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



Proof:

We now prove that $y \neq u$.

- We have done relaxation when processing x, so

$$d[y] \leq d[x] + w(x, y) \qquad (1)$$

- Since x is added to S earlier, by hypothesis,

$$d[x] = \delta(s, x) \qquad (2)$$

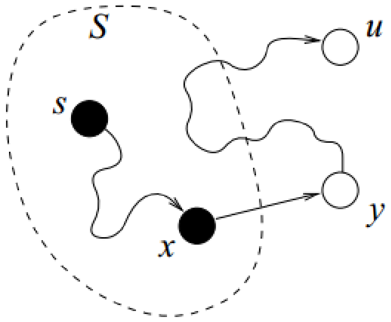
- Since <s, ..., x, y> is sub-path of a shortest path,

$$\delta(s, y) = \delta(s, x) + w(x, y) = d[x] + w(x, y) \qquad (3)$$

- By (1) and (3), $d[y] \leq \delta(s, y)$. Hence $d[y] = \delta(s, y)$.

Correctness of Dijkstra’s Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s,u)$.



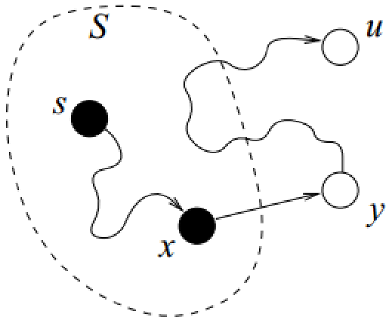
Proof:

We now prove that $y \neq u$.

- We have done relaxation when processing x, so
$$d[y] \leq d[x] + w(x,y) \quad (1)$$
- Since x is added to S earlier, by hypothesis,
$$d[x] = \delta(s,x) \quad (2)$$
- Since <s, …, x, y> is sub-path of a shortest path,
$$\delta(s,y) = \delta(s,x) + w(x,y) = d[x] + w(x,y) \quad (3)$$
- By (1) and (3), $d[y] \leq \delta(s,y)$. Hence $d[y] = \delta(s,y)$. So $y \neq u$ (because we suppose $d[u] > \delta(s,u)$).

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



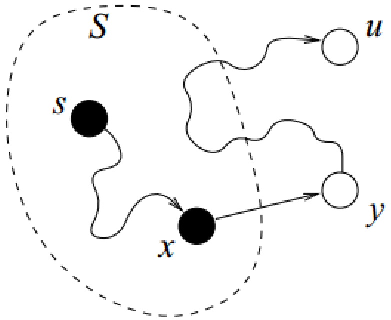
Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] =$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



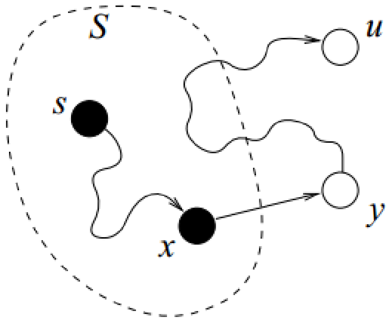
Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] = \delta(s, y)$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



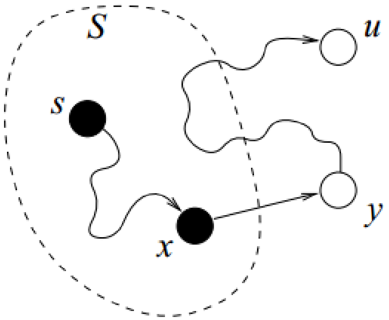
Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] = \delta(s, y) \leq \delta(s, u)$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s,u)$.



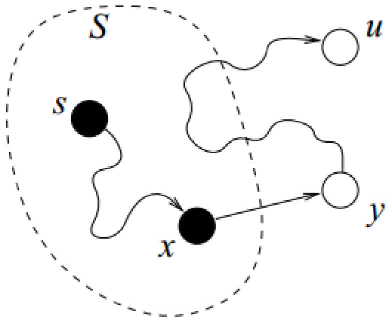
Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s,y) \le \delta(s,u)$, and thus

$$d[y] = \delta(s,y) \le \delta(s,u) \quad d[u]$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



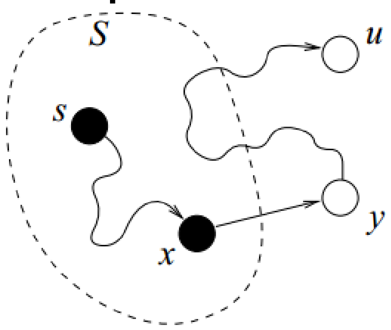
Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$$

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s,u)$.



Proof:

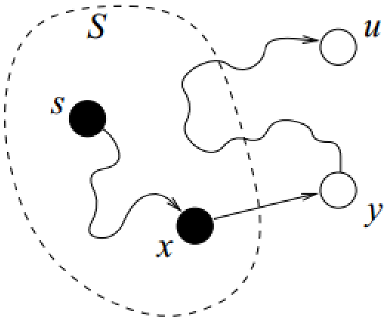
Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s,y) \le \delta(s,u)$, and thus

$$d[y] = \delta(s,y) \le \delta(s,u) < d[u]$$

Thus y should have been added to S u,

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



Proof:

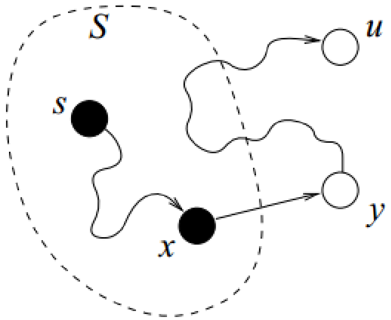
Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$$

Thus y should have been added to S before u,

Correctness of Dijkstra's Algorithm

Theorem: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s, u)$.



Proof:

Since y appears midway on the path from s to u, and all subsequent edges have non-negative weights, we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$$

Thus y should have been added to S before u, in contradiction to our assumption that u is the next vertex to be added to S.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(   ) time.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once,

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., times in total.

Analysis of Dijkstra’s Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph. Each call of the inner loop does O(1) work

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph. Each call of the inner loop does O(1) work plus, possibly, one Decrease-Key operation.

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph. Each call of the inner loop does O(1) work plus, possibly, one Decrease-Key operation.
- Recalling that all of the priority queue operations require O(log |Q|) = O(log|V|) time, we have that the algorithm uses

Analysis of Dijkstra's Algorithm

- The initialization uses only O(|V|) time.
- Each vertex is processed exactly once, so Non-Empty() and Extract-Min() are called exactly once, i.e., O(|V|) times in total.
- The inner loop for (each v ∈ Adj[u]) is called once for each edge in the graph. Each call of the inner loop does O(1) work plus, possibly, one Decrease-Key operation.
- Recalling that all of the priority queue operations require O(log |Q|) = O(log|V|) time, we have that the algorithm uses

$$O(|V|) + |E| \cdot O(1 + \log|V|) = O(|E| \log |V|)$$

time.

Description of Dijkstra's Algorithm

Dijkstra(G,w,s)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, source vertex $\boldsymbol{s}$

Output: None

for $u \in V$ **do**
    $d[u] \leftarrow \infty$, $color[u] \leftarrow$ WHITE;// Initialize
end
$d[s] \leftarrow 0$;
$pred[s] \leftarrow$ NULL;
$Q \leftarrow$queue with all vertices;
while *Non-Empty(Q)* **do**
    // Process all vertices
    $u \leftarrow$Extract-Min(Q);// Find new vertex
    for $v \in Adj[u]$ **do**
        if $d[u] + w(u, v) < d[v]$ **then**
            // If estimate improves
            $d[v] \leftarrow d[u] + w(u, v)$;// relax
            Decrease-Key($Q, v, d[v]$);
            $pred[v] \leftarrow u$;
        end
    end
    $color[u] \leftarrow$BLACK;
end

$\boldsymbol{O(|V|)}$

$\boldsymbol{O(\log |V|) \cdot |E|}$

Prim’s Algorithm vs. Dijkstra’s Algorithm

- Dijkstra’s algorithm looks similar to Prim’s algorithm.

- To understand the differences clearly, try them both on some example.

Outline

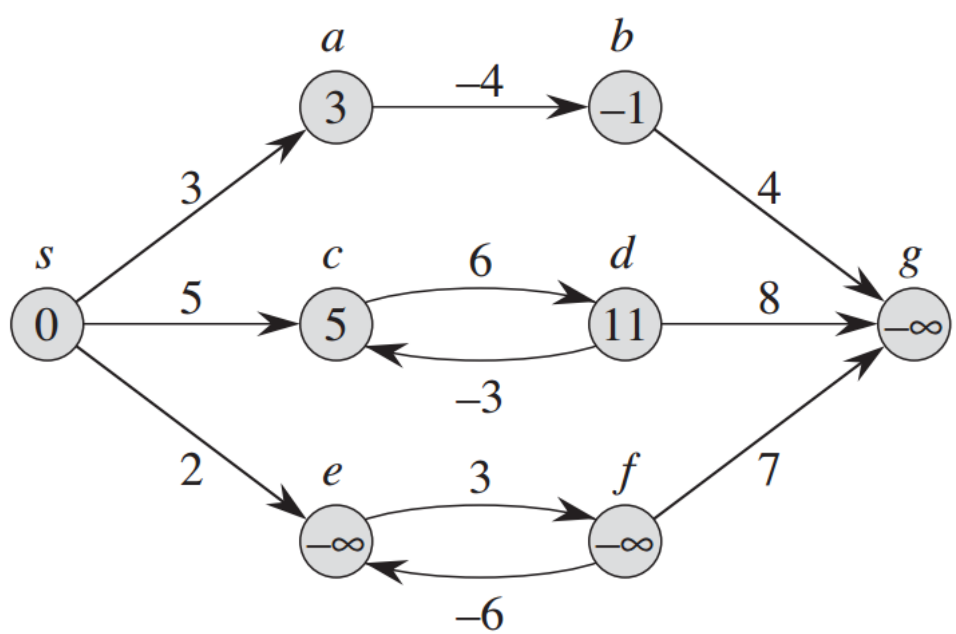
- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra’s Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra’s algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

Negative-weight edges

- If the graph contains a negative-weight cycle reachable from the source vertex s, shortest-path weights are not well defined.

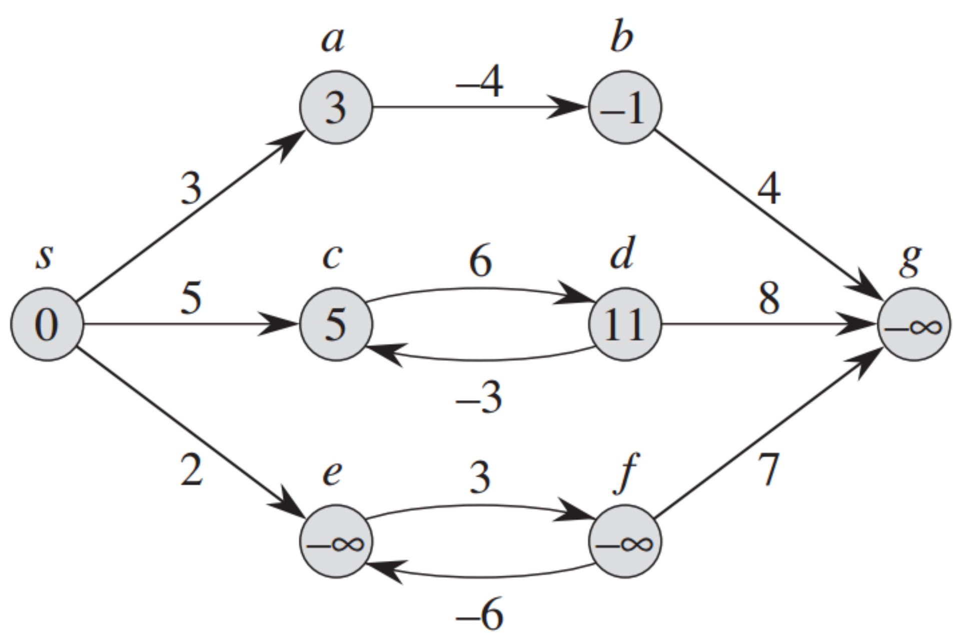
Negative-weight edges

- If the graph contains a negative-weight cycle reachable from the source vertex s, shortest-path weights are not well defined.



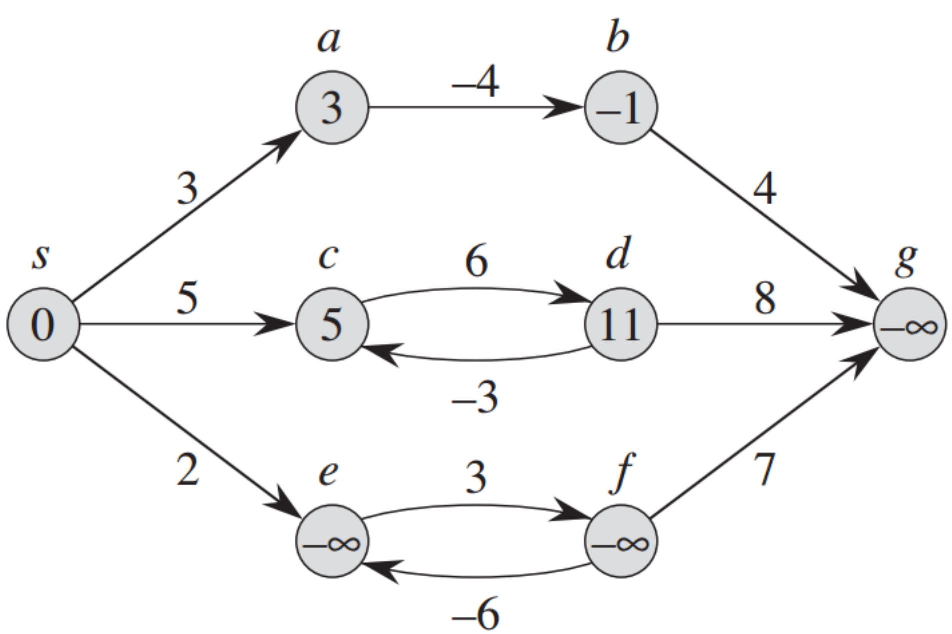
Negative-weight edges

- If the graph contains a negative-weight cycle reachable from the source vertex s, shortest-path weights are not well defined.
- By following the proposed "shortest" path and then traversing the negative-weight cycle, we can always find a path with weight.



Negative-weight edges

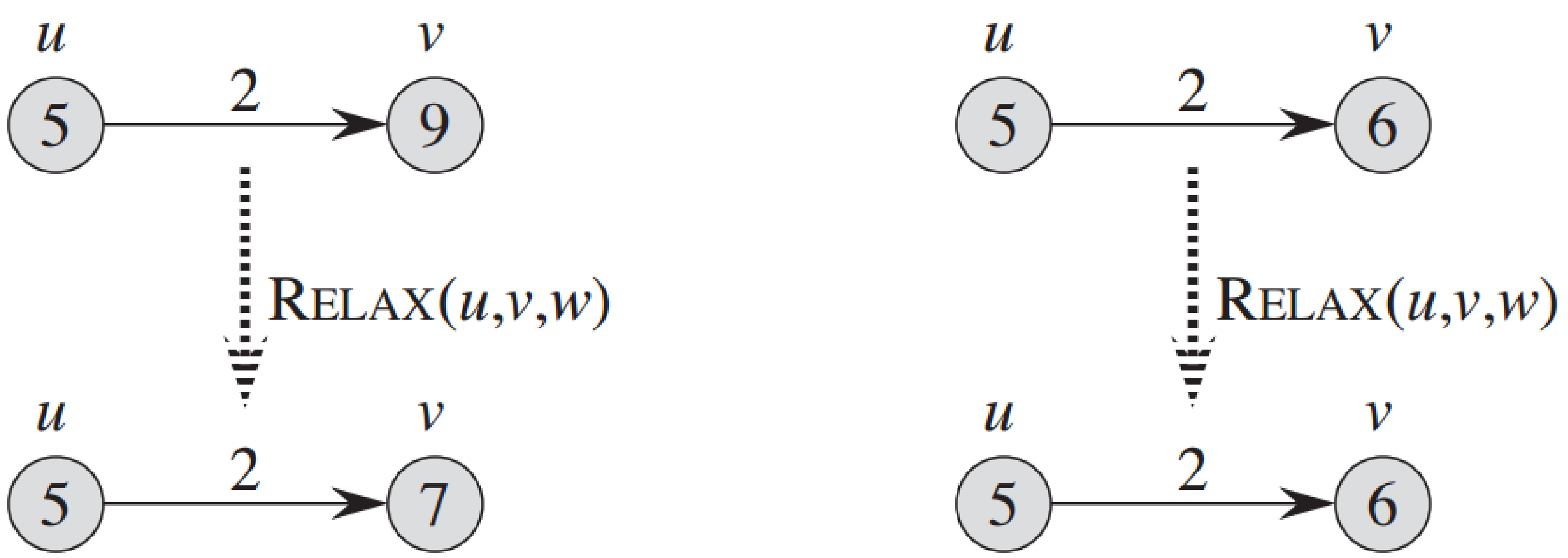
- If the graph contains a negative-weight cycle reachable from the source vertex s, shortest-path weights are not well defined.
- By following the proposed "shortest" path and then traversing the negative-weight cycle, we can always find a path with lower weight.



Review of The Algorithm for Relaxing an Edge

Relax(u,v)

Input: Update estimation of $\boldsymbol{u}$ according to distance of $\boldsymbol{v}$
Output: None
if $d[u] + w(u, v) < d[v]$ **then**
  $d[v] \leftarrow d[u] + w(u, v)$;
  $pred[v] \leftarrow u$;
end



Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
|  d[u] ← ∞, pred[u] ←NIL;// Initialize
end
```

Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
|   d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
|   for e ∈ E do
|   |
|   end
end
```

Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
  | d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
  | for e ∈ E do
  |   | RELAX(u, v, w);
  | end
end
```

The Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

```
Input: A directed graph G,weights w, and the source vertex s
Output: Return FALSE if G contains negative cycle, return TRUE if
        shortest paths from s to any other vertices obtained.
for u ∈ V do
 | d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
 | for e ∈ E do
 |  | RELAX(u, v, w);
 | end
end
for e ∈ E do
 | if d[v] > d[u] + w(u, v) then
 |  | return FALSE;
 | end
end
return
```

Description of Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
  | d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
  | for e ∈ E do
  |   | RELAX(u, v, w);
  | end
end
for e ∈ E do
  | if d[v] > d[u] + w(u, v) then
  |   | return FALSE;
  | end
end
return TRUE;
```

An Example of Bellman-Ford Algorithm

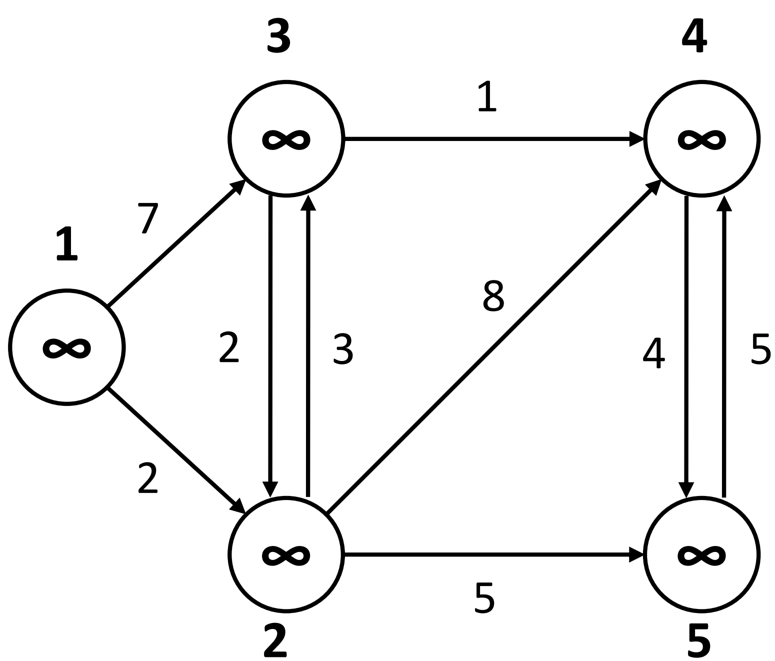
Initialization

d

∞	∞	∞	∞	∞

pred

N	N	N	N	N



An Example of Bellman-Ford Algorithm

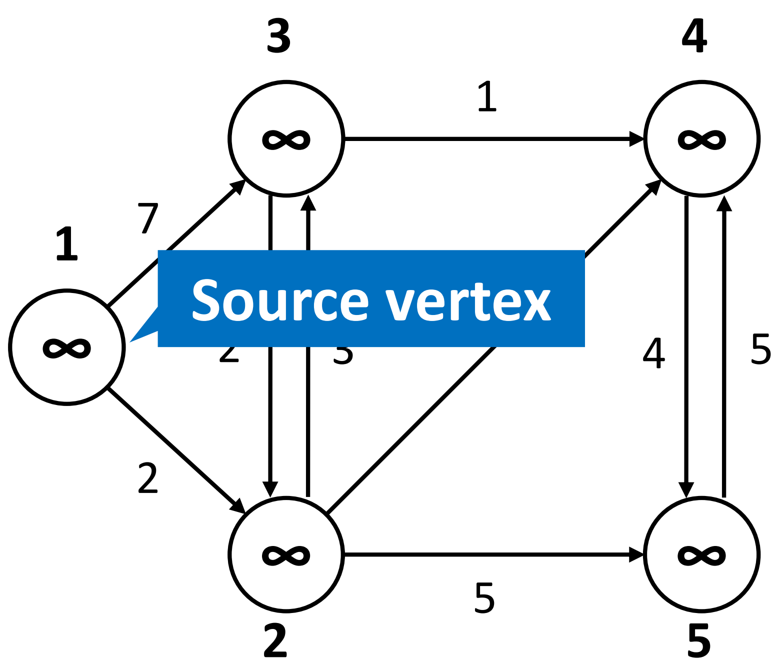
Initialization

d

∞	∞	∞	∞	∞

pred

N	N	N	N	N



An Example of Bellman-Ford Algorithm

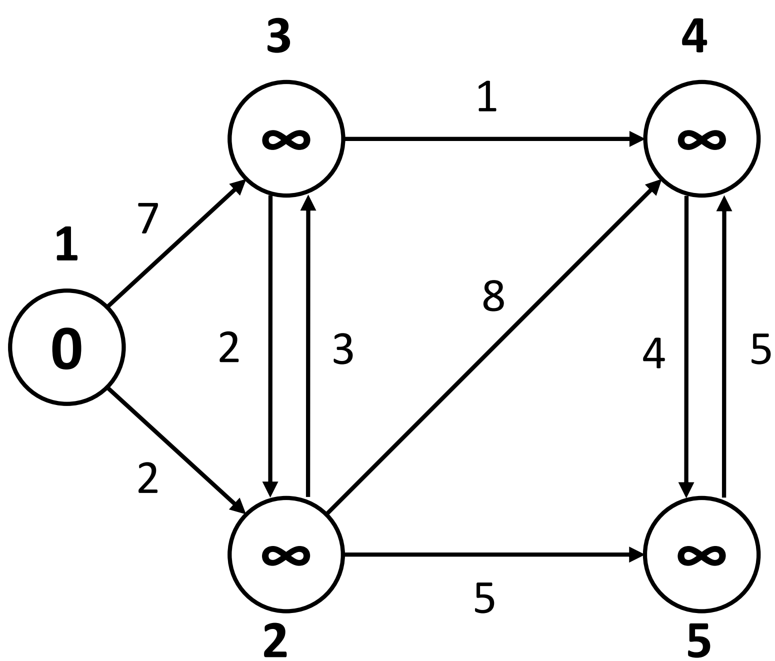
Initialization

d

0	∞	∞	∞	∞

pred

N	**N**	**N**	**N**	**N**



An Example of Bellman-Ford Algorithm

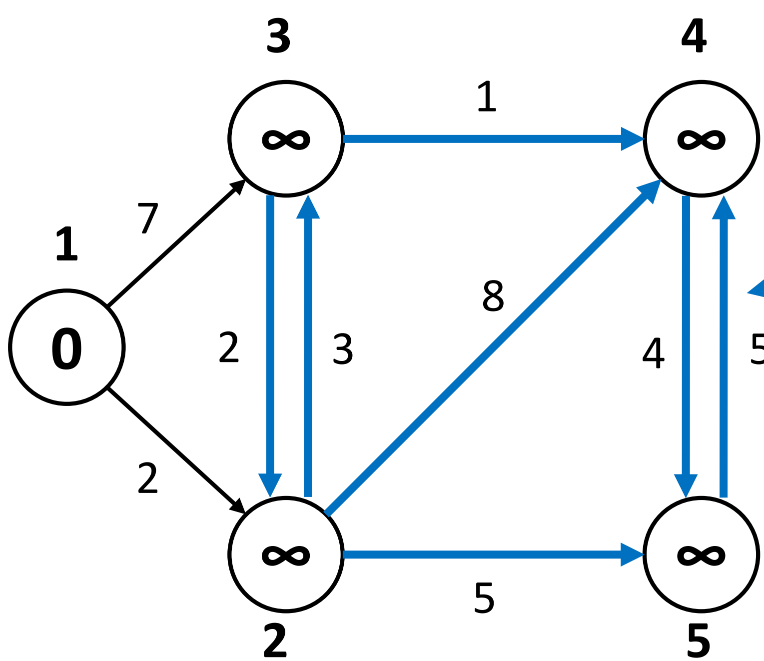
1st round

d

0	**∞**	**∞**	**∞**	**∞**

pred

N	**N**	**N**	**N**	**N**



These edges do not cause Relaxation.

An Example of Bellman-Ford Algorithm

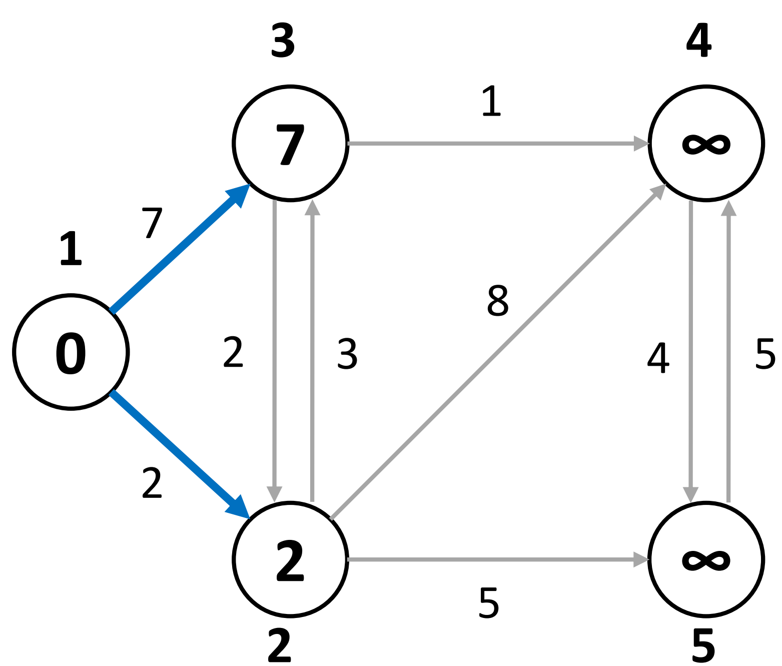
1st round

d

0	**2**	**7**	**∞**	**∞**

pred

N	**1**	**1**	**N**	**N**



An Example of Bellman-Ford Algorithm

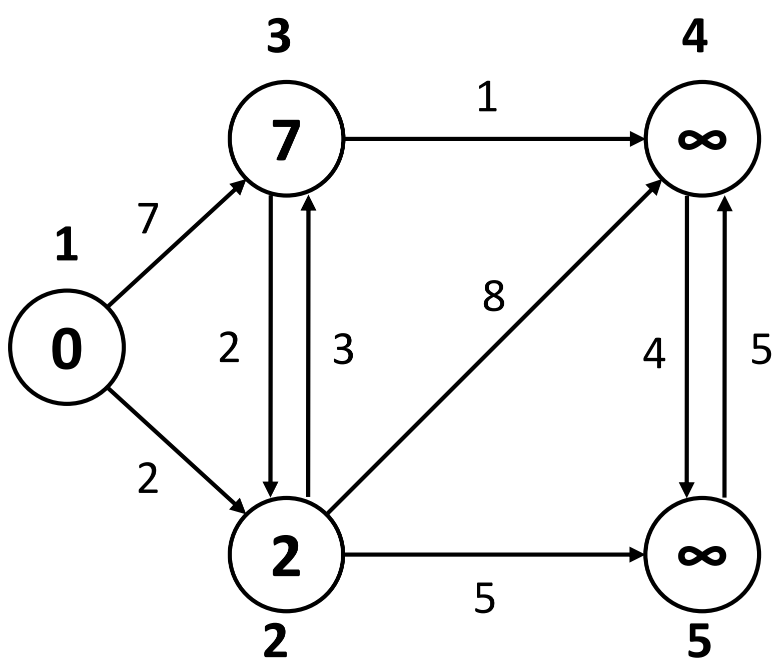
2nd round

d

0	2	7	∞	∞

pred

N	1	1	N	N



An Example of Bellman-Ford Algorithm

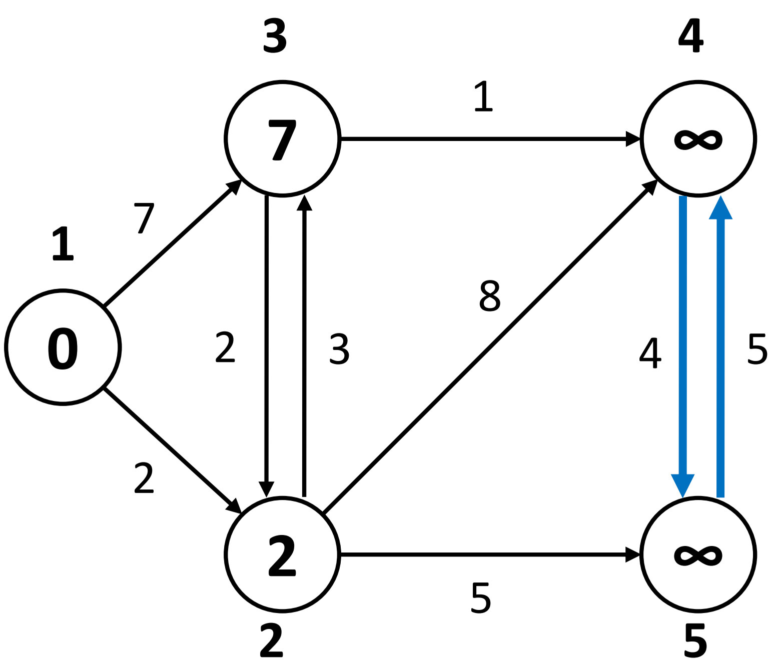
2nd round

d

0	**2**	**7**	**∞**	**∞**

pred

N	**1**	**1**	**N**	**N**



An Example of Bellman-Ford Algorithm

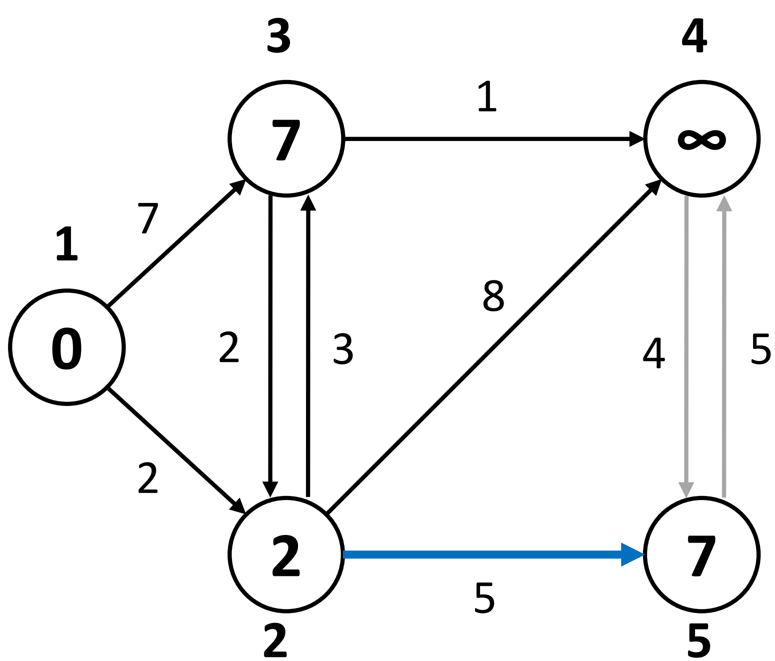
2nd round

d

0	**2**	**7**	**∞**	**7**

pred

N	**1**	**1**	**N**	**2**



An Example of Bellman-Ford Algorithm

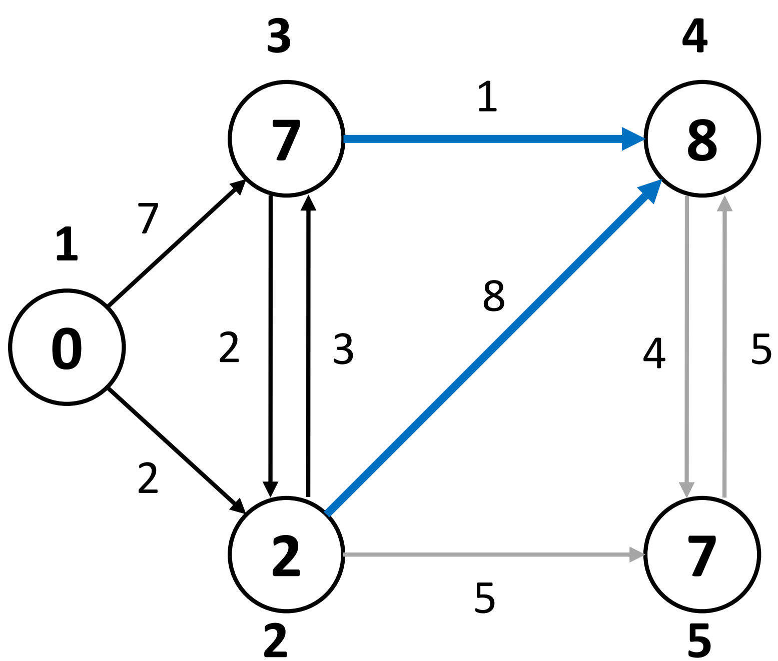
2nd round

d

0	**2**	**7**	**8**	**7**

pred

N	**1**	**1**	**3**	**2**



An Example of Bellman-Ford Algorithm

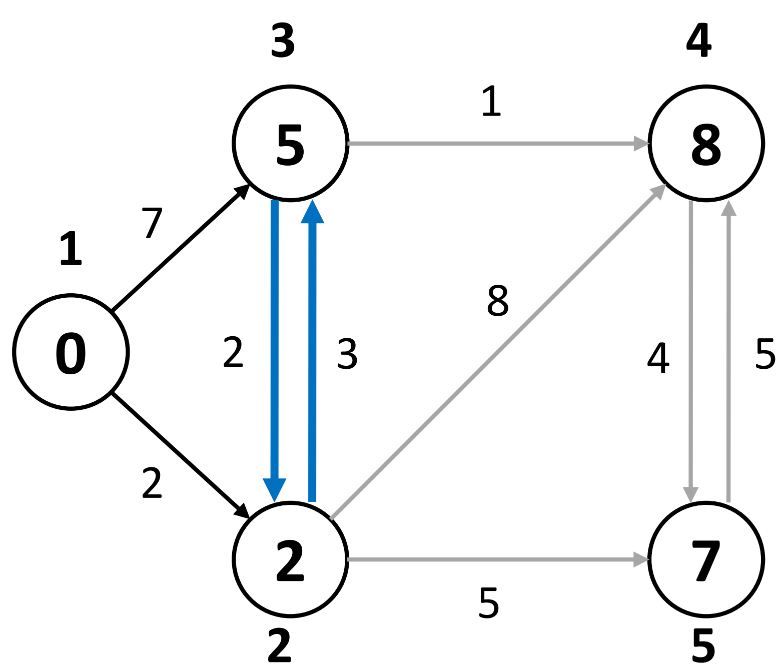
2nd round

d

0	**2**	**5**	**8**	**7**

pred

N	**1**	**2**	**3**	**2**



An Example of Bellman-Ford Algorithm

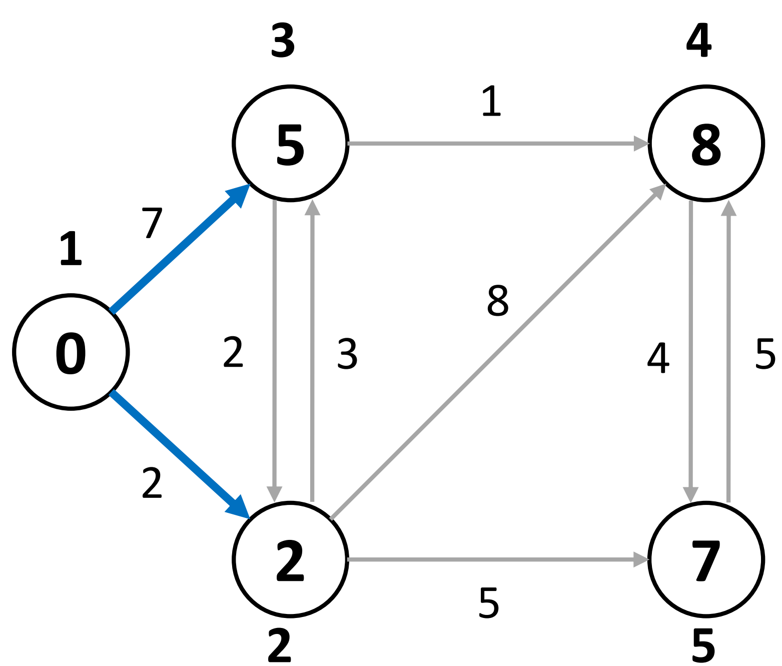
2nd round

d

0	**2**	**5**	**8**	**7**

pred

N	**1**	**2**	**3**	**2**



An Example of Bellman-Ford Algorithm

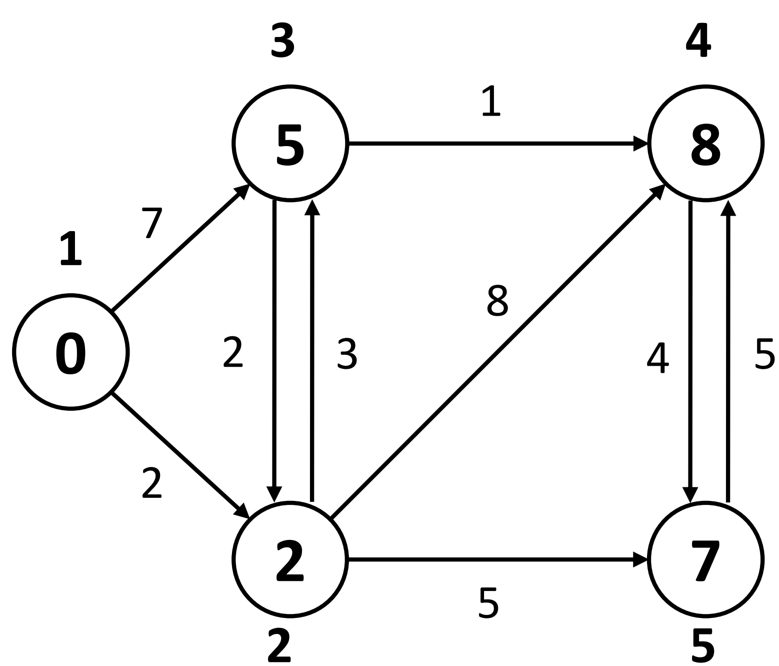
3rd round

d

0	2	5	8	7

pred

N	1	2	3	2



An Example of Bellman-Ford Algorithm

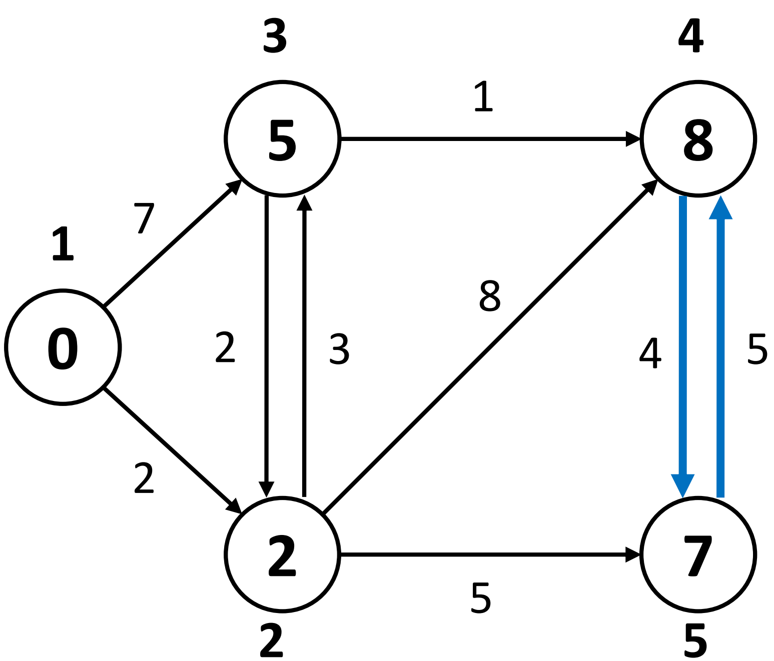
3rd round

d

0	2	5	8	7

pred

N	1	2	3	2



An Example of Bellman-Ford Algorithm

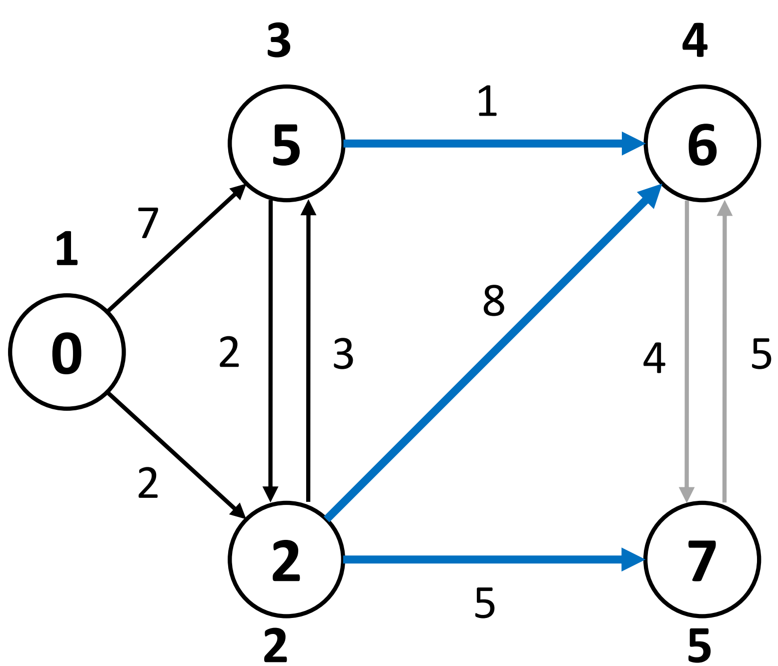
3rd round

d

0	**2**	**5**	**6**	**7**

pred

N	**1**	**2**	**3**	**2**



An Example of Bellman-Ford Algorithm

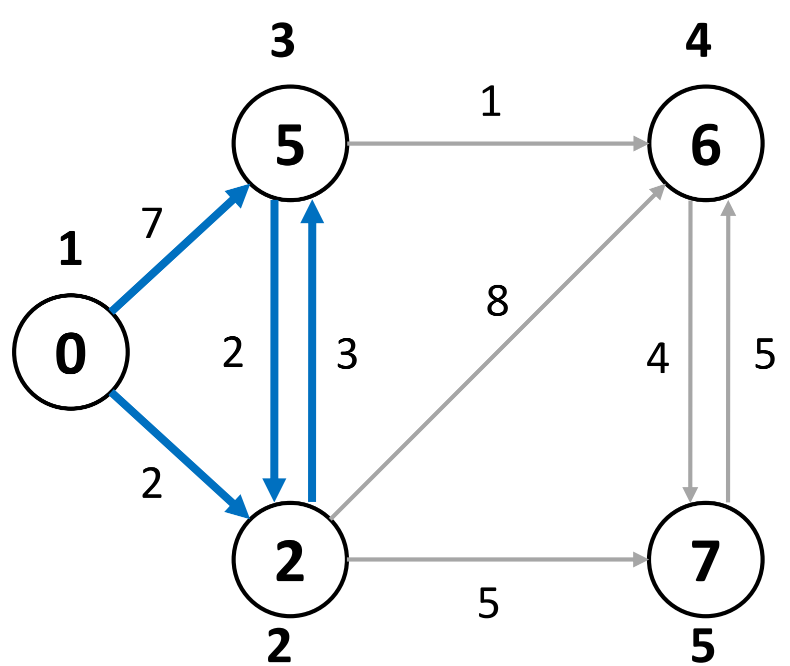
3rd round

d

0	2	5	6	7

pred

N	1	2	3	2



An Example of Bellman-Ford Algorithm

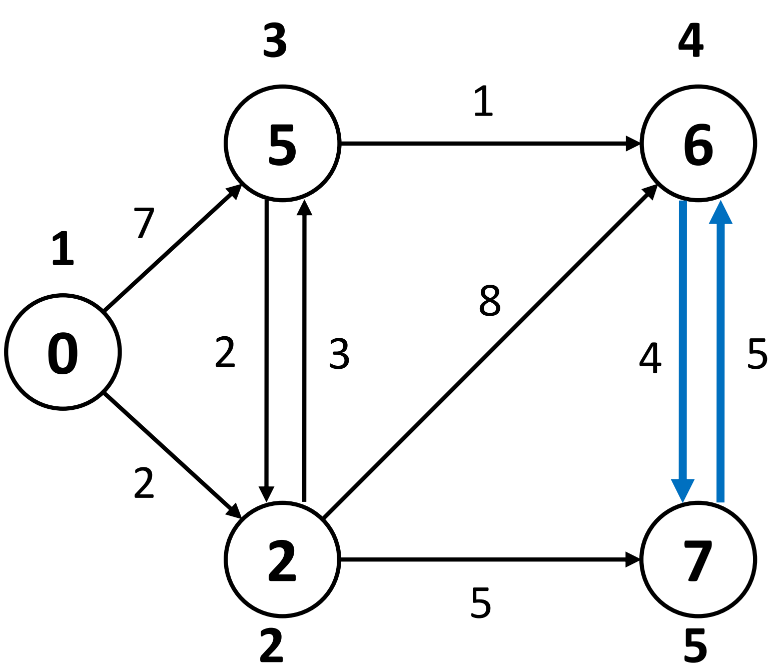
4th round

d

0	2	5	6	7

pred

N	1	2	3	2



An Example of Bellman-Ford Algorithm

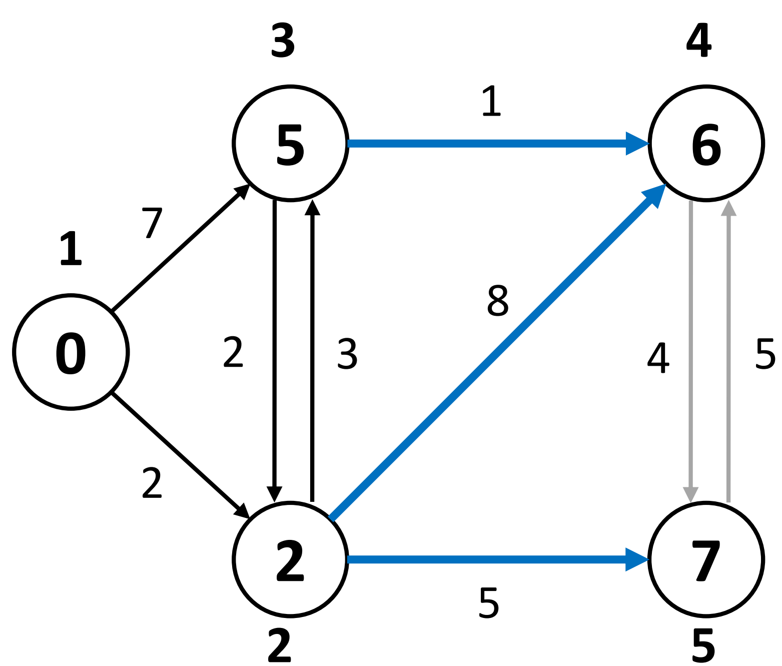
4th round

d

0	**2**	**5**	**6**	**7**

pred

N	**1**	**2**	**3**	**2**



An Example of Bellman-Ford Algorithm

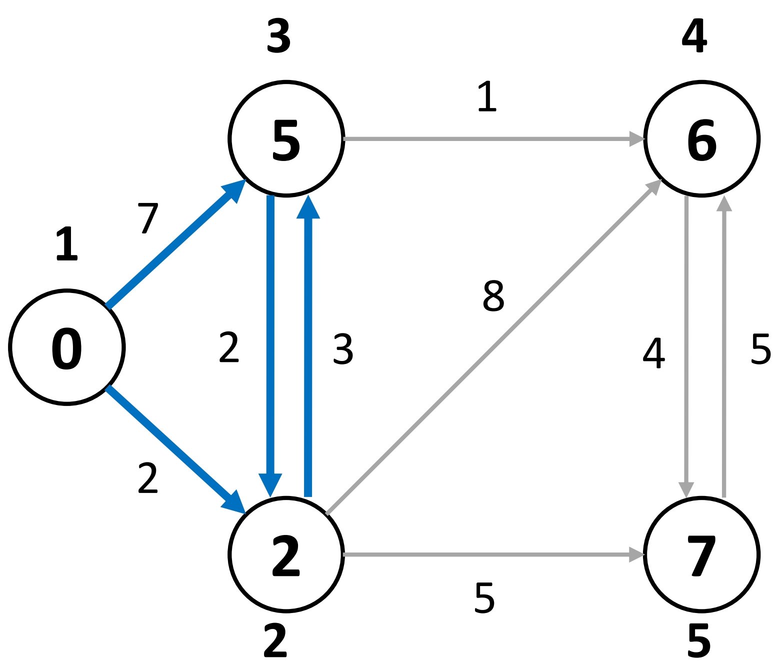
4th round

d

0	**2**	**5**	**6**	**7**

pred

N	**1**	**2**	**3**	**2**



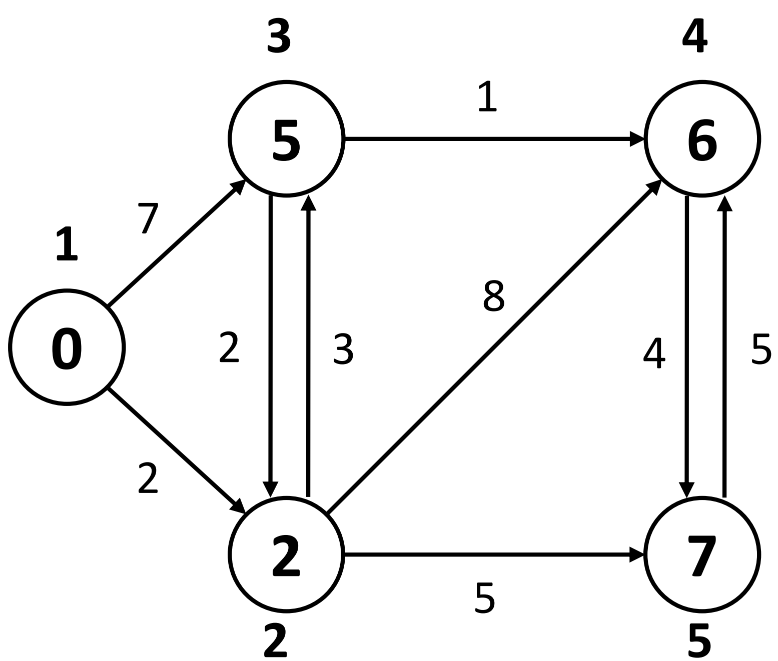
An Example of Bellman-Ford Algorithm

d

0	**2**	**5**	**6**	**7**

pred

N	**1**	**2**	**3**	**2**



Outline

- Review to Part IV
- Single-Source Shortest Paths Problem
- Dijkstra's Algorithm
  - The idea
  - The algorithm
  - Analysis of Dijkstra's algorithm
- The Bellman-Ford Algorithm
  - The algorithm
  - Analysis of Bellman-Ford algorithm

Analysis of Bellman-Ford Algorithm

- The Bellman-Ford algorithm runs in time $O(|V| \cdot |E|)$ since the initialization takes $O(|V|)$ time, each of the $|V| - 1$ passes over the edges takes $O(|E|)$ time, and the **for** loop takes $O(|E|)$ time.

Bellman-Ford(G,w,s)

```
Input: A directed graph G,weights w, and the source vertex s
Output: Return FALSE if G contains negative cycle, return TRUE if
        shortest paths from s to any other vertices obtained.
for u ∈ V do
  | d[u] ← ∞, pred[u] ←NIL;// Initialize
end
for i ← 1 to |V| − 1 do
  | for e ∈ E do
  |   | RELAX(u, v, w);
  | end
end
for e ∈ E do
  | if d[v] > d[u] + w(u, v) then
  |   | return FALSE;
  | end
end
return TRUE;
```

Analysis of Bellman-Ford Algorithm

- The Bellman-Ford algorithm runs in time $O(|V| \cdot |E|)$ since the initialization takes $O(|V|)$ time, each of the $|V| - 1$ passes over the edges takes $O(|E|)$ time, and the **for** loop takes $O(|E|)$ time.

Bellman-Ford(G,w,s)

Input: A directed graph $\boldsymbol{G}$,weights $\boldsymbol{w}$, and the source vertex $\boldsymbol{s}$
Output: Return FALSE if G contains negative cycle, return TRUE if shortest paths from s to any other vertices obtained.

```
for u ∈ V do
 | d[u] ← ∞, pred[u] ←NIL;// Initialize        O(|V|)
end
for i ← 1 to |V| − 1 do                        O(|V|·|E|)
 | for e ∈ E do
 |  | RELAX(u, v, w);
 | end
end
for e ∈ E do                                   O(|E|)
 | if d[v] > d[u] + w(u, v) then
 |  | return FALSE;
 | end
end
return TRUE;
```


dank u
ju faleminderit
Tack
谢谢
Tak
mulţumesc
Asante
kiitos
Gracias
Salamat!
Terima kasih
Aliquam
Merci
Dankie
Obrigado
ありがとう
köszönöm
grazie
Aliquam
Go raibh maith agat
děkuji
Thank you
gam