COMP 550
Algorithm and Analysis

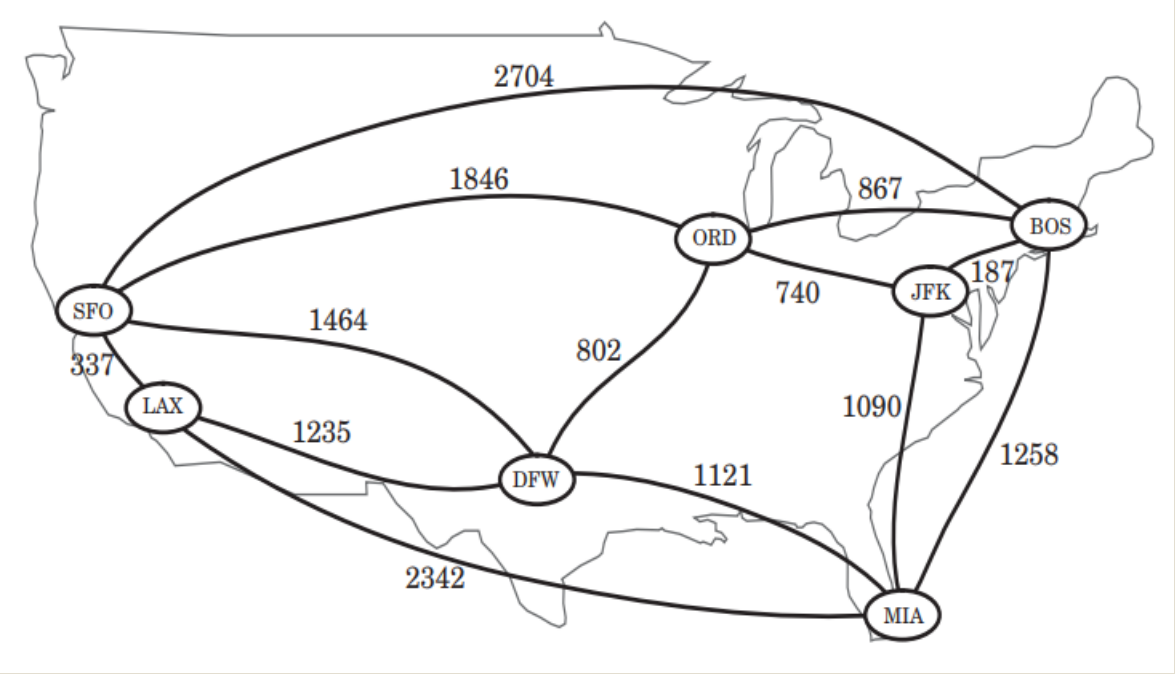
Shortest Path Algorithms

Based on CLRS Secs. 22 and 23

Some slides are adapted from ones by Prof. Jim Anderson

Single-Source Shortest Path (SSSP)

- Input: A graph $G = (V, E)$ and a source vertex $v \in V$
- Output: Shortest path from v to all other vertices



Source: Algorithm Design and Application by Goodrich and Tamassia

Single-Source Shortest Path (SSSP)

- Input: A graph $G = (V, E)$ and a source vertex $v \in V$
- Output: Shortest path from v to all other vertices

- If G is an unweighted graph, then BFS solves the single-source shortest path problem.
- What about weighted graph? (this chapter)

Single-Source Shortest Path (SSSP)

- There can by different variations:
  - Single destination shortest path problem: Shortest path to v from all other vertices
  - Single-pair shortest path problem: Shortest path between a single pair of vertices
  - All pair shortest path problem: Shortest path between all pair of vertices
- All of the above can be solved if we can solve Single-Source version.

Optimal Substructure

- Does shortest-path problem have optimal substructure property?
  - Does shortest path from u to v contain shortest path between some other vertices?
- Yes!

Lemma 22.1. (**Subpaths of shortest paths are shortest paths**)

If $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a shortest path from v_0 to v_k, then $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ with $0 \leq i \leq j \leq k$ is a shortest path from v_i to v_j.

Exercise: Does longest path problem have optimal substructure property? (CLRS Sec 14.3)

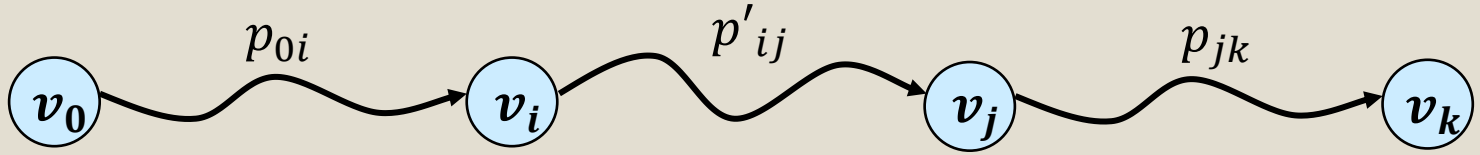
Optimal Substructure

Lemma 22.1. (**Subpaths of shortest paths are shortest paths**)

If $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a shortest path from v_0 to v_k, then $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ with $0 \le i \le j \le k$ is a shortest path from v_i to v_j.

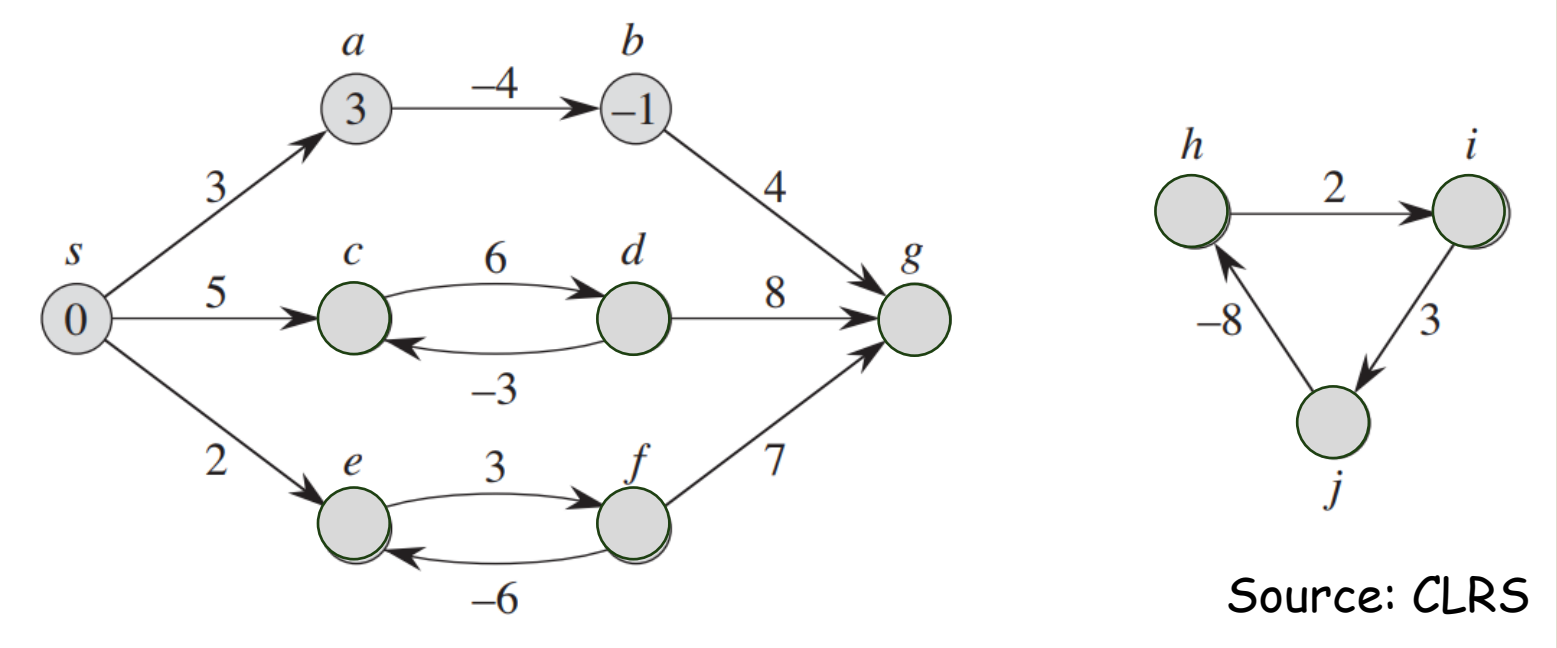
- Proof: "cut and paste".
- Shortest path from v_0 to v_k looks like this

- If p_{ij} is not the shortest path between v_i and v_j, then assume p'_{ij} is the shortest path between them.
- Using p'_{ij} creates shorter path from v_0 to v_k, contradiction. (p is not shortest)



Negative-Weight Edges

- Notation: $\delta(s, u) =$ Shortest distance from s to v
- $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$

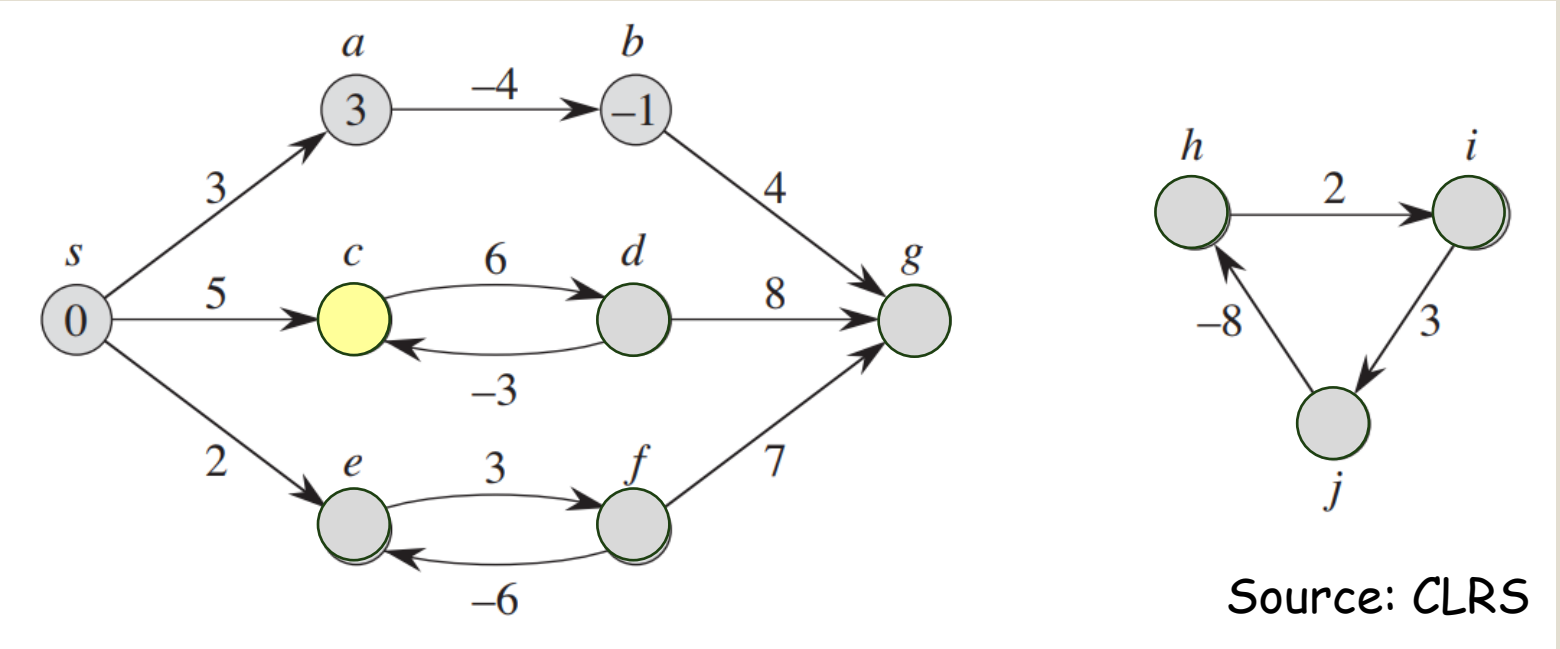


Source: CLRS

A graph with 11 vertices

Negative-Weight Edges

- What is $\delta(s, c)$?
  - Many paths: $\langle s, c\rangle, \langle s, c, d, c\rangle, \langle s, c, d, c, d, c\rangle, \ldots$
  - Weight of the loop $\langle c, d, c\rangle$ is positive.
  - $\delta(s, c) =$ Weight/cost of path $\langle s, c\rangle = 5$

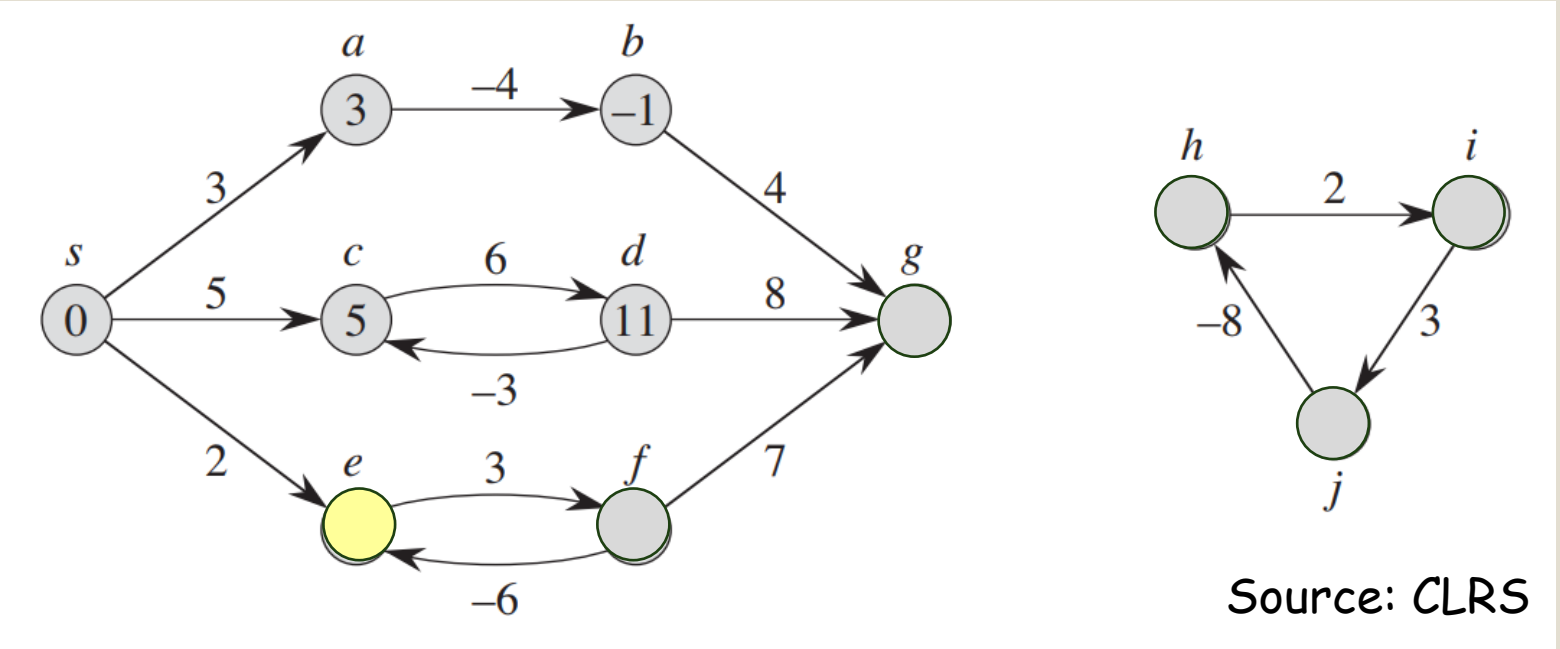


Source: CLRS

A graph with 11 vertices

Negative-Weight Edges

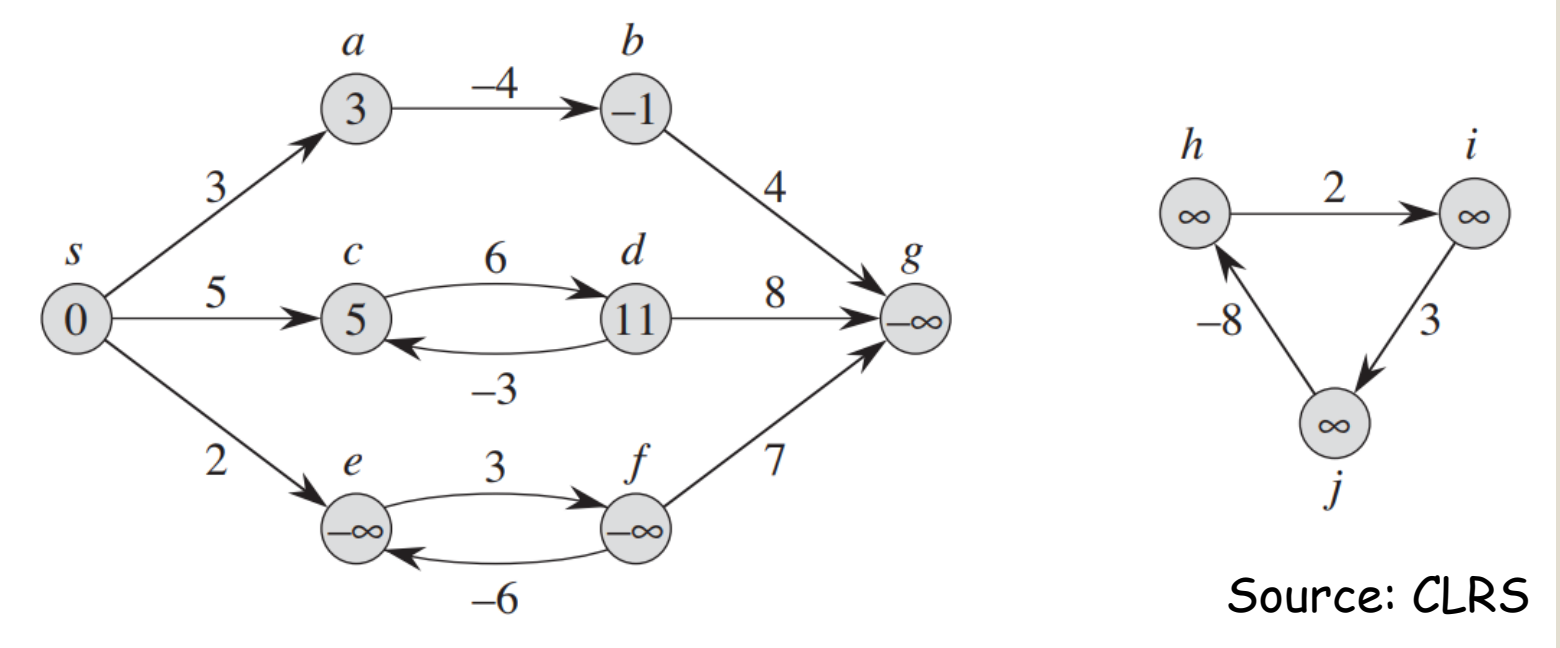
- What is $\delta(s, e)$?
  - Many paths: $\langle s, e\rangle, \langle s, e, d, e\rangle, \langle s, e, d, e, d, e\rangle, \ldots$
  - Weight of the loop $\langle c, d, e\rangle$ is negative.
  - $\delta(s, e) = -\infty$ (No shortest path from s to e)



A graph with 11 vertices

Negative-Weight Edges

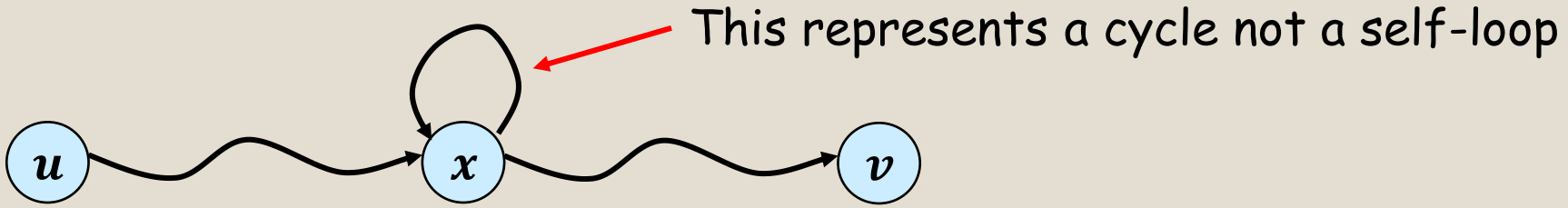
- $\delta(s, f) = \delta(s, g) = -\infty$ (Reachable via a negative-weight cycle)
- $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$ (Not reachable from s)



A graph with 11 vertices

Cycles

- Shortest paths do not contain cycles
  - Negative–weight cycle ⇒ No shortest path
  - Positive-weight cycle ⇒ Removing cycle decreases path weight
  - 0-weight cycle ⇒ Removing cycle maintains the path weight



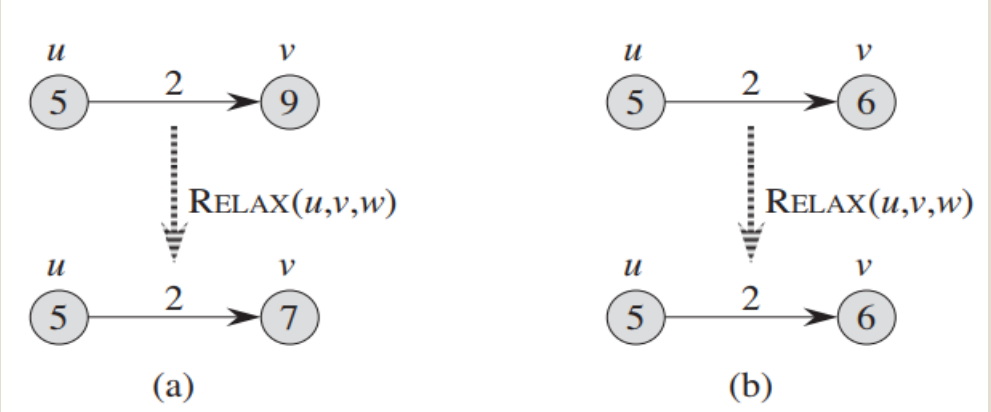
Relaxation

- Shortest-path algorithms keep track of $v.d$ and $v.\pi$
  - $v.d$ = shortest-path estimates, $v.\pi$ = predecessor vertex in shortest path
  - Will call the following two procedures in different algorithms

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = 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.π = u
```

Relax edge (u, v): two cases



Source: CLRS

(u, v) = An edge, w = weight function

Properties of Relaxation

- Consider any algorithm in which $v.d$, and $v.\pi$ are first initialized by calling $\text{Initialize}(G, s)$ [s is the source], and are only changed by calling **Relax**. We have:

Lemma 22.11 (Upper bound property):
$(\forall\ v :: v.d \geq \delta(s, v))$ is an invariant.

Corollary 22.12 (No-path property):
If there is no path from s to v, then $v.d = \infty$ is an invariant.

Lemma 22.14 (Convergence property):
If $s \rightsquigarrow u \longrightarrow v$ is a SP and $u.d = \delta(s, u)$ prior to calling `Relax(u,v,w)`, then $v.d = \delta(s, v)$ all time after `Relax(u,v,w)` is returned

More on CLRS

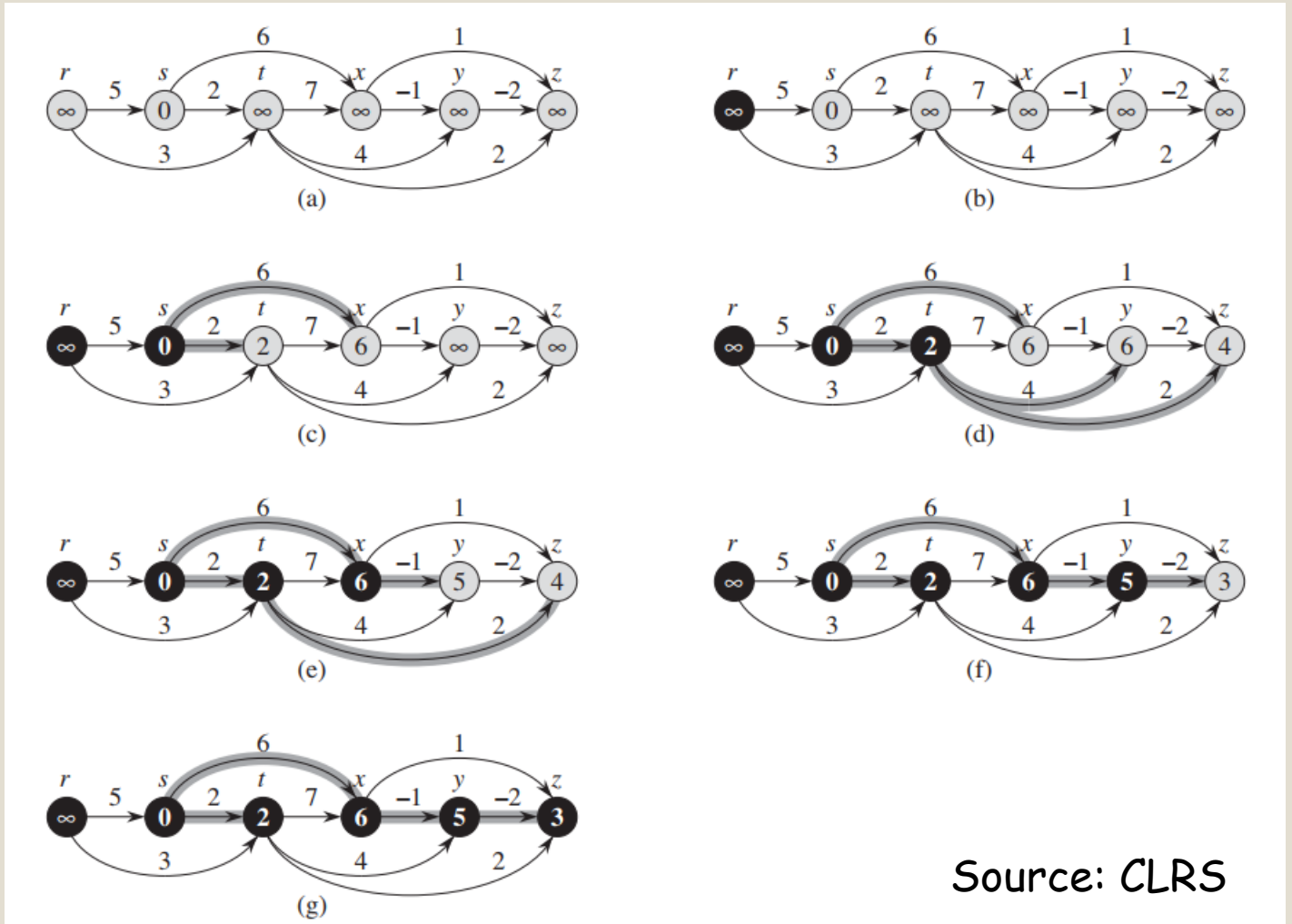
Exercise: Prove them (Proofs in Sec. 22.5)

SSSP in DAG

- Input graph is a directed acyclic graph

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

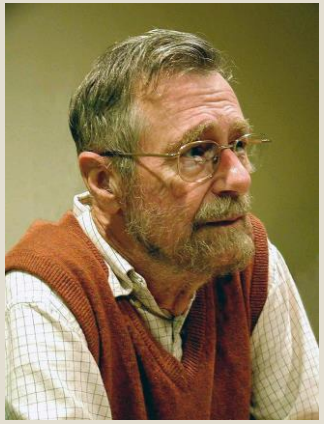
Running time: $\Theta(V + E)$



Source: CLRS

SSSP When Nonnegative-Weight Edge

- Input graph is a weighted graph with no negative-weight edge
- **Dijkstra's Algorithm** finds SSSPs for such graphs
  - Basically, generalized BFS
- Idea: When a node is the closest undiscovered node to the source vertex, we have found its shortest path



"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.…One of the reasons that it is so nice was that I designed it without pencil and paper. ….Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame."
-Edsger Dijkstra (Comm. Of ACM 2001)

Dijkstra's Algorithm

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = 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.π = u
```

All vertices are inserted in Q

S is the head of Q now

Greedy strategy: Extract the head of Q (its shortest distance has been found)

Update any shorter distance via node u

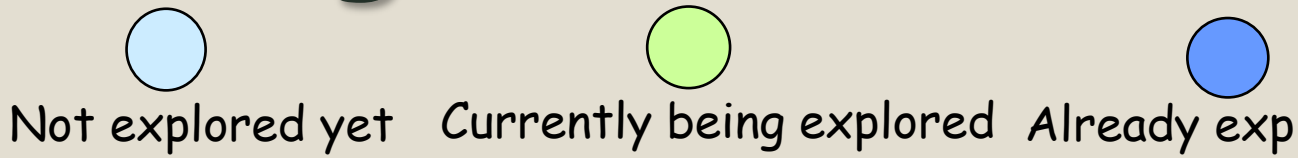
```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S = ∅
3   Q = ∅
4   for each vertex u ∈ G.V
5       INSERT(Q, u)
6   while Q ≠ ∅
7       u = EXTRACT-MIN(Q)
8       S = S ∪ {u}
9       for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

Q: Min Priority Queue (order by $v.d$), contains unexplored nodes

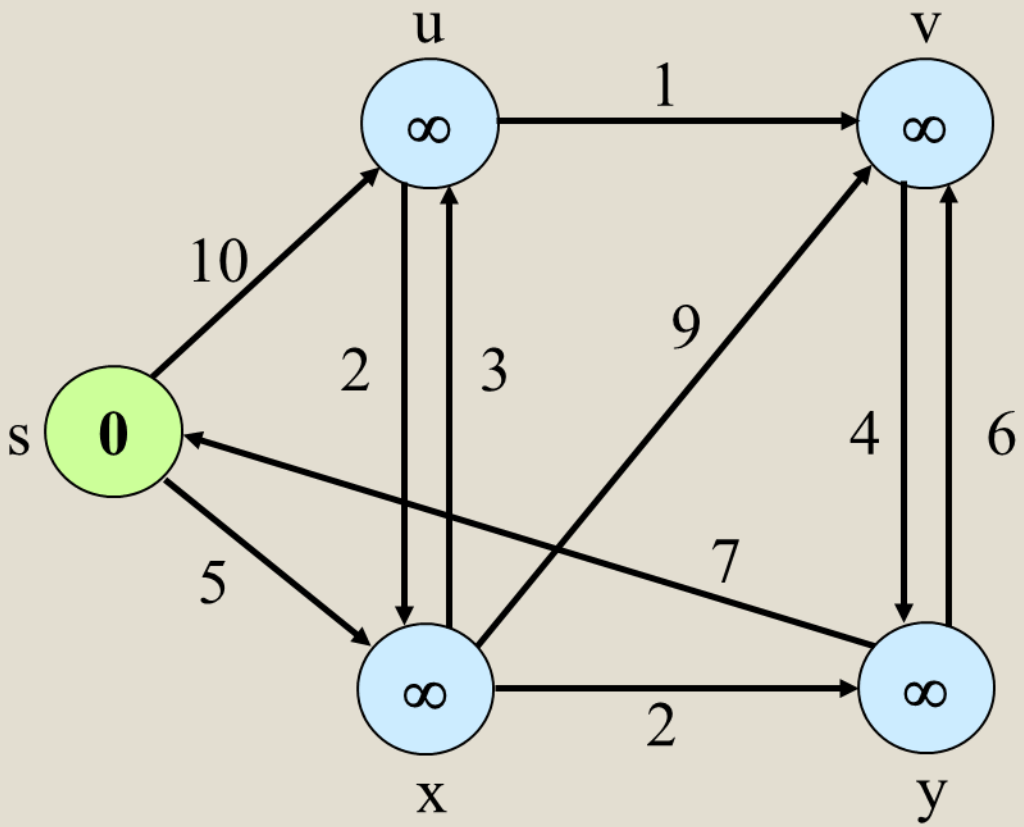
S: vertices where we know we've found the shortest path. (Like gray nodes in BFS)

Source: CLRS

Dijkstra's Algorithm




```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5       INSERT(Q, u)
 6  while Q ≠ ∅
 7       u = EXTRACT-MIN(Q)
 8       S = S ∪ {u}
 9       for each vertex v in G.Adj[u]
10            RELAX(u, v, w)
11            if the call of RELAX decreased v.d
12                 DECREASE-KEY(Q, v, v.d)
```

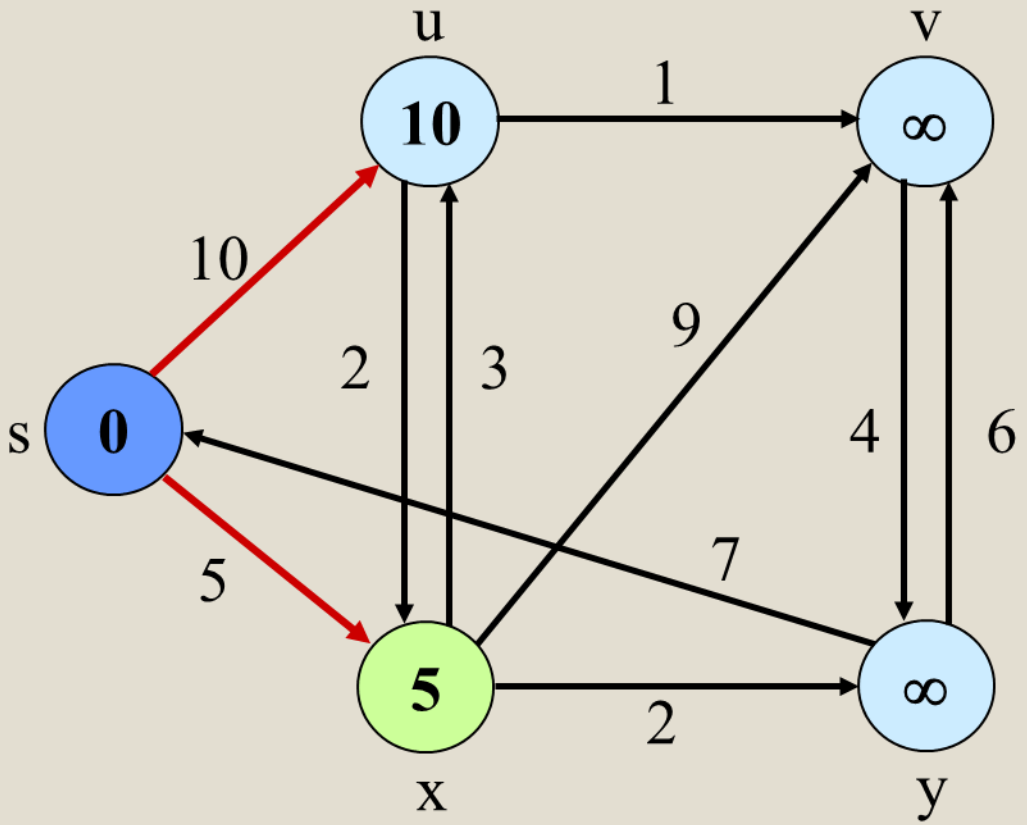


Example Courtesy: Prof. Jim Anderson

Dijkstra's Algorithm

Not explored yet Currently being explored Already explored

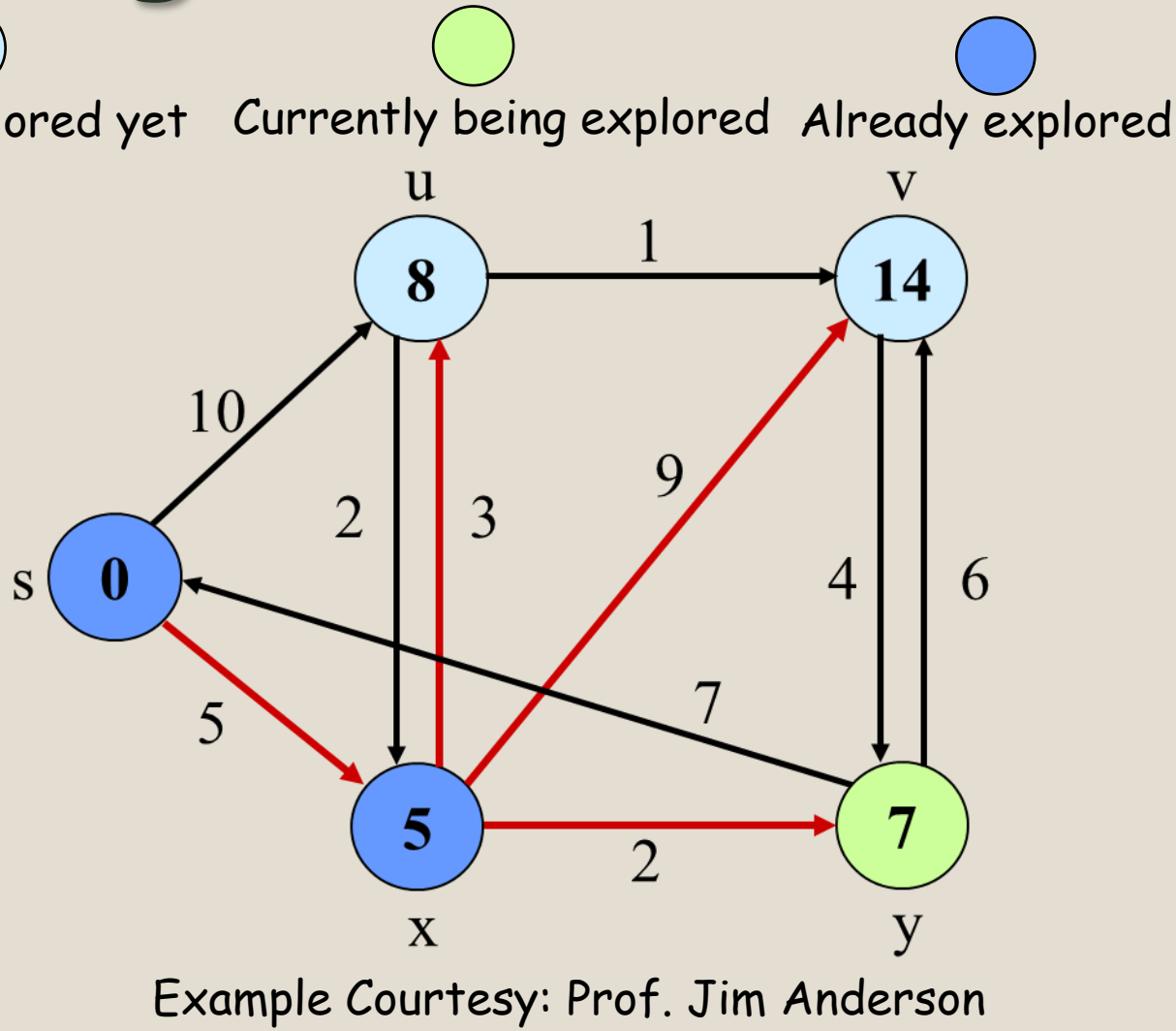
```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5      INSERT(Q, u)
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      S = S ∪ {u}
 9      for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```



Example Courtesy: Prof. Jim Anderson

Dijkstra's Algorithm

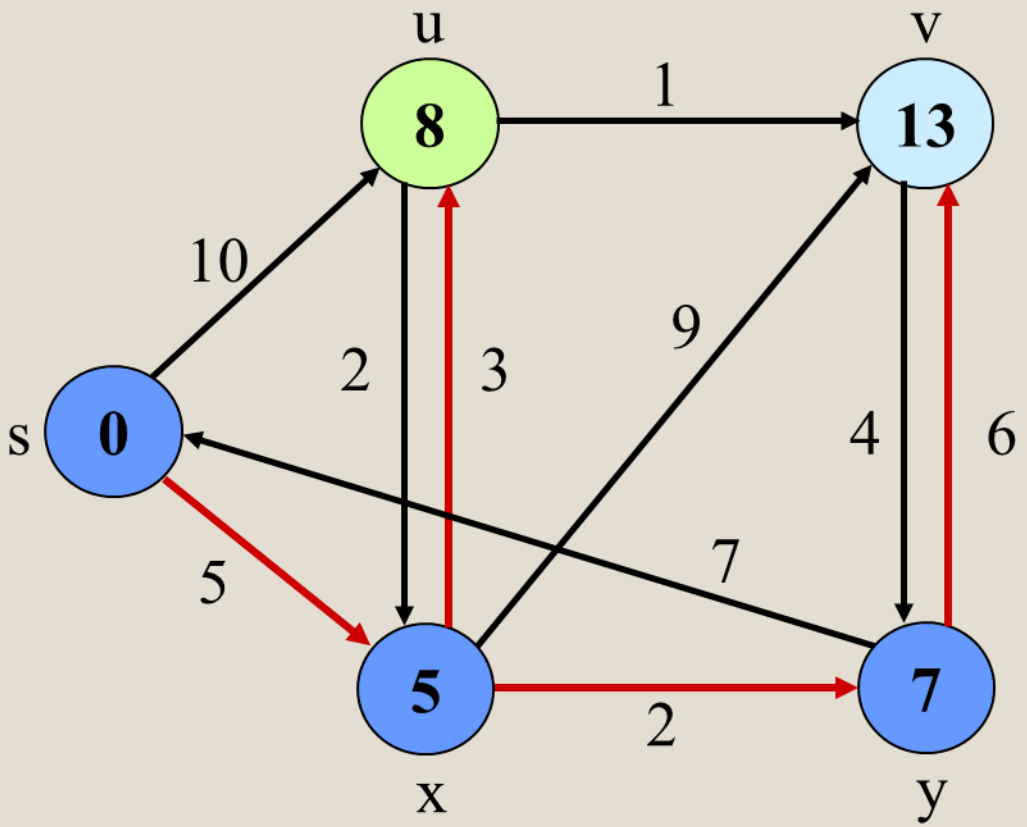
```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5       INSERT(Q, u)
 6  while Q ≠ ∅
 7       u = EXTRACT-MIN(Q)
 8       S = S ∪ {u}
 9       for each vertex v in G.Adj[u]
10            RELAX(u, v, w)
11            if the call of RELAX decreased v.d
12                 DECREASE-KEY(Q, v, v.d)
```



Example Courtesy: Prof. Jim Anderson

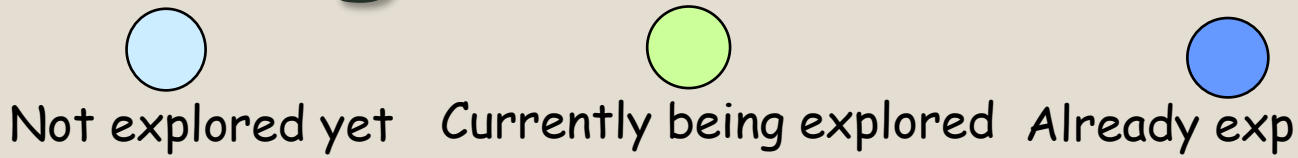
Dijkstra's Algorithm


```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5       INSERT(Q, u)
 6  while Q ≠ ∅
 7       u = EXTRACT-MIN(Q)
 8       S = S ∪ {u}
 9       for each vertex v in G.Adj[u]
10            RELAX(u, v, w)
11            if the call of RELAX decreased v.d
12                 DECREASE-KEY(Q, v, v.d)
```

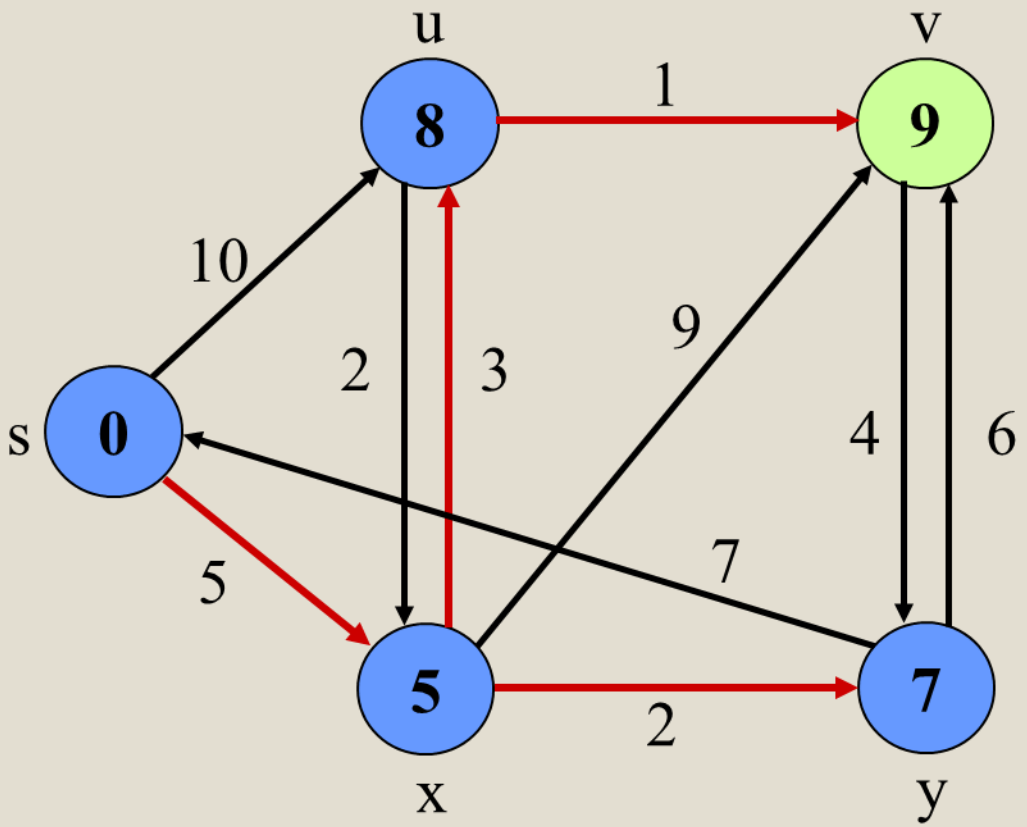


Example Courtesy: Prof. Jim Anderson

Dijkstra's Algorithm




```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5       INSERT(Q, u)
 6  while Q ≠ ∅
 7       u = EXTRACT-MIN(Q)
 8       S = S ∪ {u}
 9       for each vertex v in G.Adj[u]
10            RELAX(u, v, w)
11            if the call of RELAX decreased v.d
12                 DECREASE-KEY(Q, v, v.d)
```



Example Courtesy: Prof. Jim Anderson

Correctness

Theorem 24.6: Upon termination, $u.d = \delta(s,u)$ for all $u \in V$. (assuming non-negative weights).

Loop invariant: At the start of each iteration of the while loop, $v.d = \delta(s,v)$ for all $v \in S$. ($S = V$ when terminate)

- Initialization: $S = \emptyset$ (Trivial)
- IH: At the start of k-th iteration, $u.d = \delta(s,u)$ for all $u \in V$.
- Consider the k-th iteration and show that the LI holds at the start of (k+1)-st iteration
  - The k-th iteration inserts a new vertex to S
  - Need to show that the LI holds for the new vertex

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S = ∅
3   Q = ∅
4   for each vertex u ∈ G.V
5       INSERT(Q, u)
6   while Q ≠ ∅
7       u = EXTRACT-MIN(Q)
8       S = S ∪ {u}
9       for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

Correctness

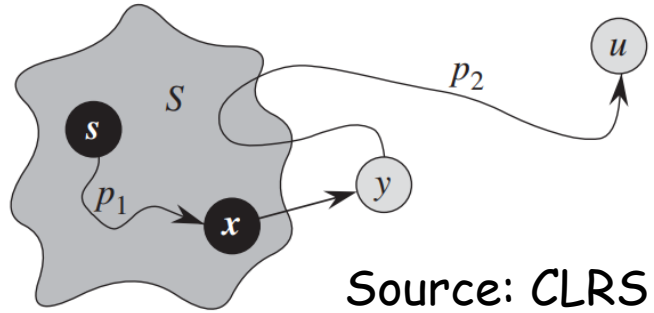
Theorem 24.6: Upon termination, $u.d = \delta(s,u)$ for all $u \in V$. (assuming non-negative weights).

Maintenance (Inductive Step):

- u is extracted from $Q = V - S$ (our goal is to show $u.d = \delta(s,u)$)
- If no s to u path exists, then $u.d = \delta(s,u) = \infty$
- Otherwise, let $\langle s, \dots, x, y, \dots u \rangle$ be a shortest path where
  - y is the first vertex in the path that is NOT is S
- $\delta(s,y) \le \delta(s,u)$, because **no negative edge**
- $u.d \le y.d$, because u is just extracted from Q, but y is not.
- Using the upper-bound property,

$$\delta(s,y) \le \delta(s,u) \le u.d \le y.d \qquad (1)$$

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```



Source: CLRS

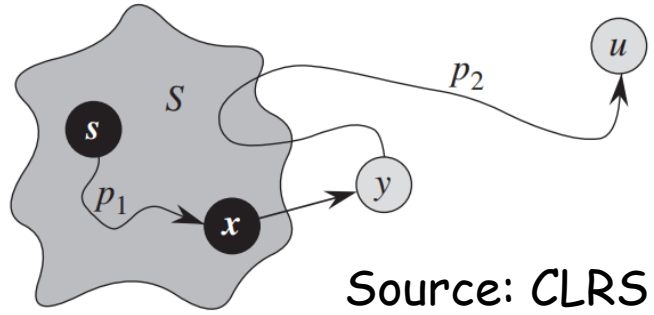
Correctness

Theorem 24.6: Upon termination, $u.d = \delta(s,u)$ for all $u \in V$. (assuming non-negative weights).

Maintenance (Inductive Step):

- $x.d = \delta(s,x)$, because $x \in S$ (by Inductive Hypothesis)
- edge (x,y) was relaxed during the loop that added x to S
- Since (x,y) is in a SP from s to y and (x,y) was relaxed when $x.d = \delta(s,x)$, $y.d$ gets the value of $\delta(s,y)$ at that time (convergence property: Lemma 22.14).
- So, when u is extracted from Q, $y.d = \delta(s,y)$ already holds.
- From (1), we get $u.d = \delta(s,u)$

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```



Source: CLRS

Running Time

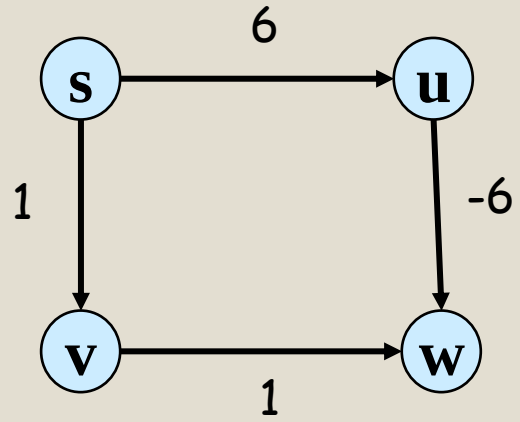
	Insert			Extract-Max			Decrease-Key			Total Running Time
	# calls	Running time per call	Total running time	# calls	Running time per call	Total running time	# calls	Running time per call	Total running time	
Node-indexed array	$O(V)$	$O(1)$	$O(V)$	$O(V)$	$O(V)$	$O(V^2)$	$O(E)$	$O(1)$	$O(E)$	$O(V^2)$
Binary Heap	Use Build-Heap instead of n Inserts		$O(V)$	$O(V)$	$O(\lg V)$	$O(V \lg V)$	$O(E)$	$O(\lg V)$	$O(E \lg V)$	$O((V + E) \lg V)$

Running time with binary heap is $O((V + E) \lg V)$, which is $O(E \lg V)$ if $E = \Omega(V)$

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

SSSP with Negative-Weight Edge

- Dijkstra's algorithm fails when there are negative edges



- Dijkstra's algorithm discovers w via v. ($w.d$ is set to 2).

SSSP with Negative-Weight Edge

- Bellman-Ford Algorithm
  - Can handle negative-weight edges and "detect" reachable negative-weight cycles.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Running time: $O(V \cdot E)$

Negative cycle exists

No negative cycle

Negative cycle detection

Source: CLRS

Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer for loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Another Loop invariant for the outer for loop:

For every vertex v, if there is a shortest path from s to v with at most $i-1$ edges, then $v.d = \delta(s, v)$.

Why the outer loop runs $|V|-1$ times?

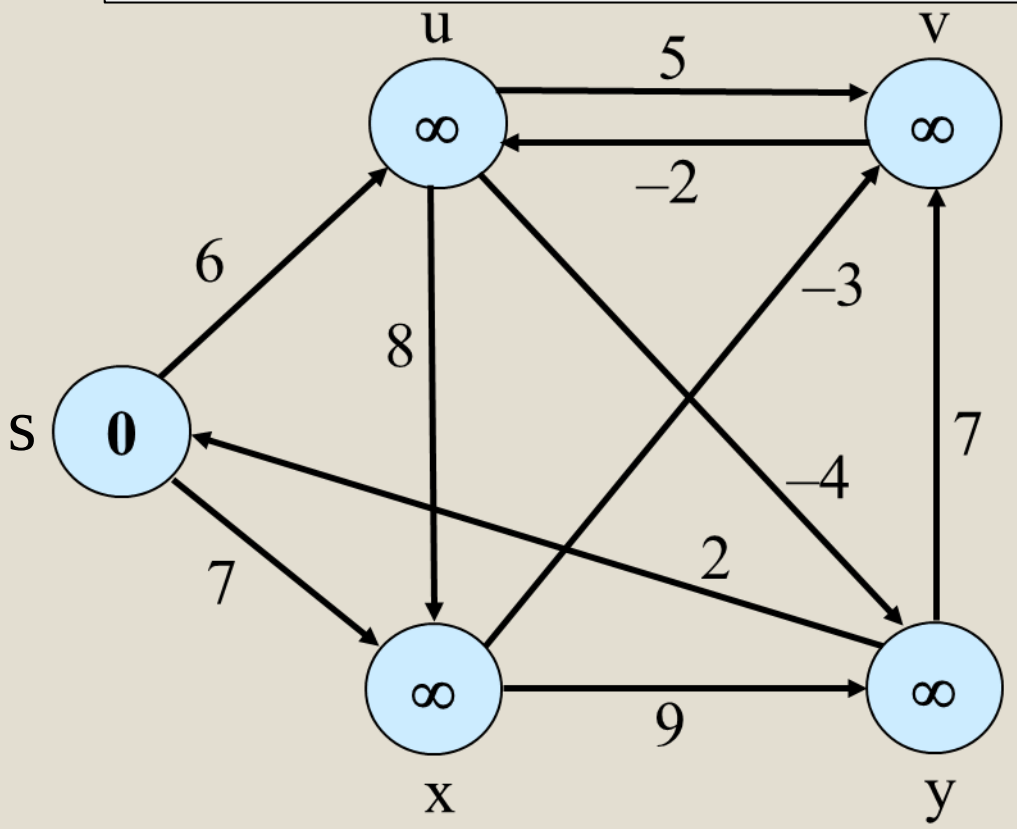
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

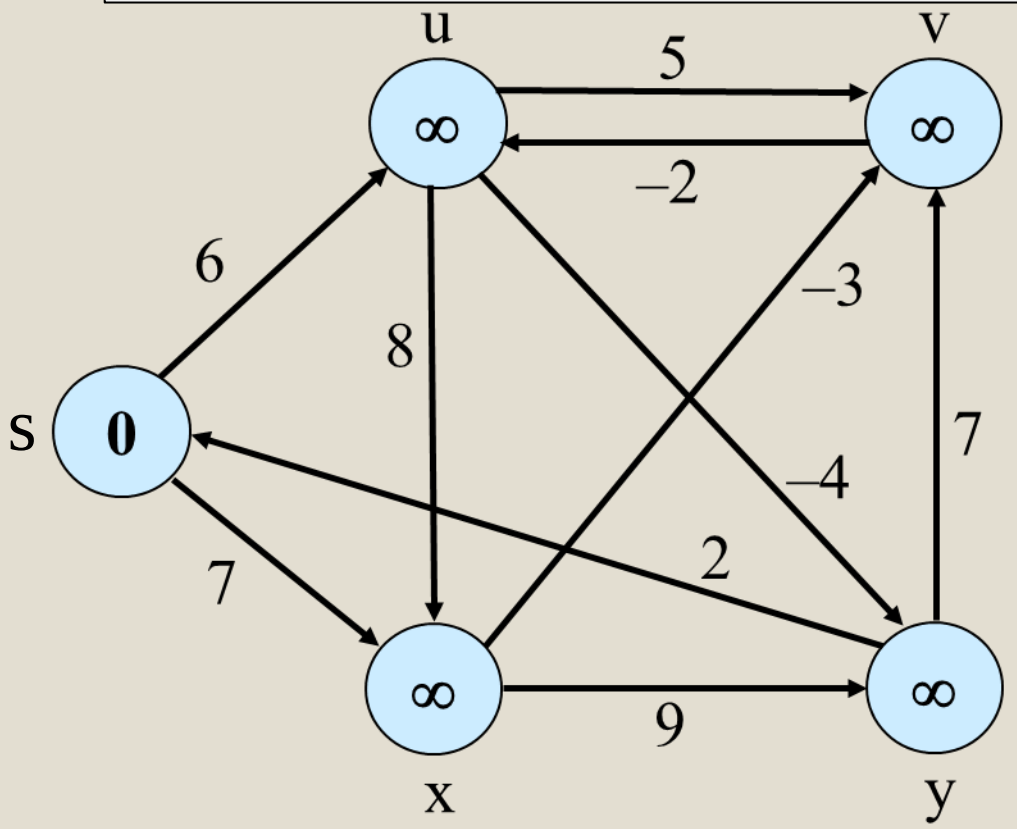
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

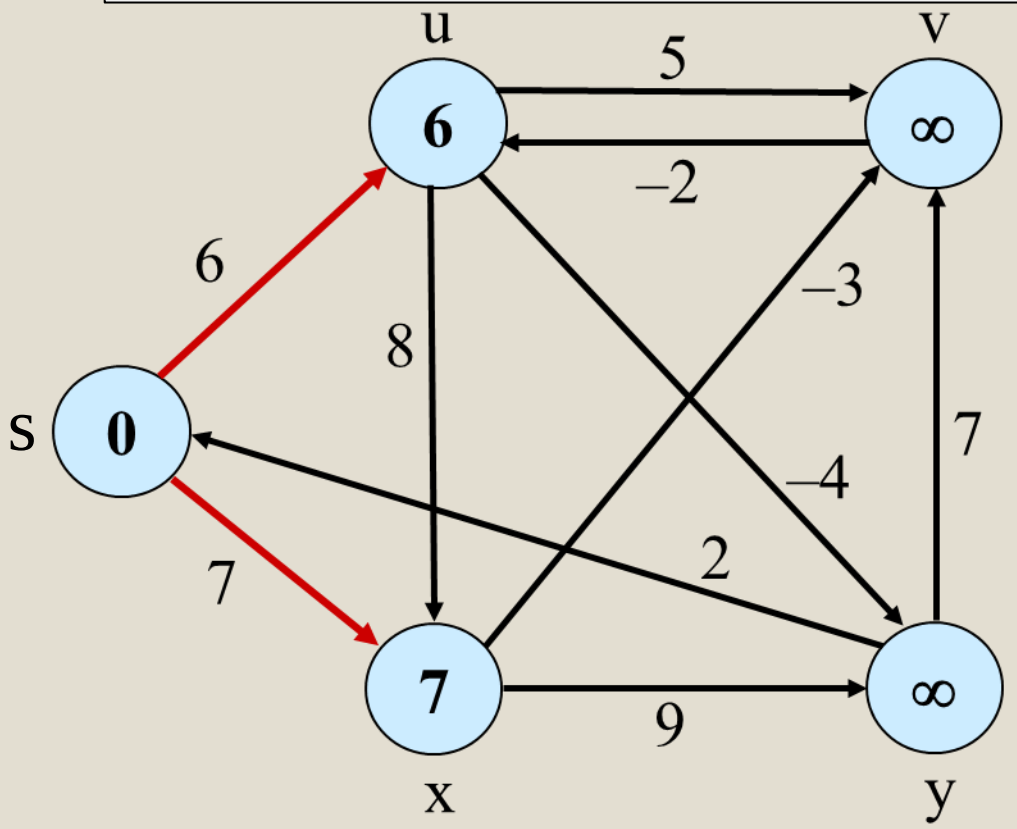
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

Red edges denote predecessor relations ($v.\pi$)

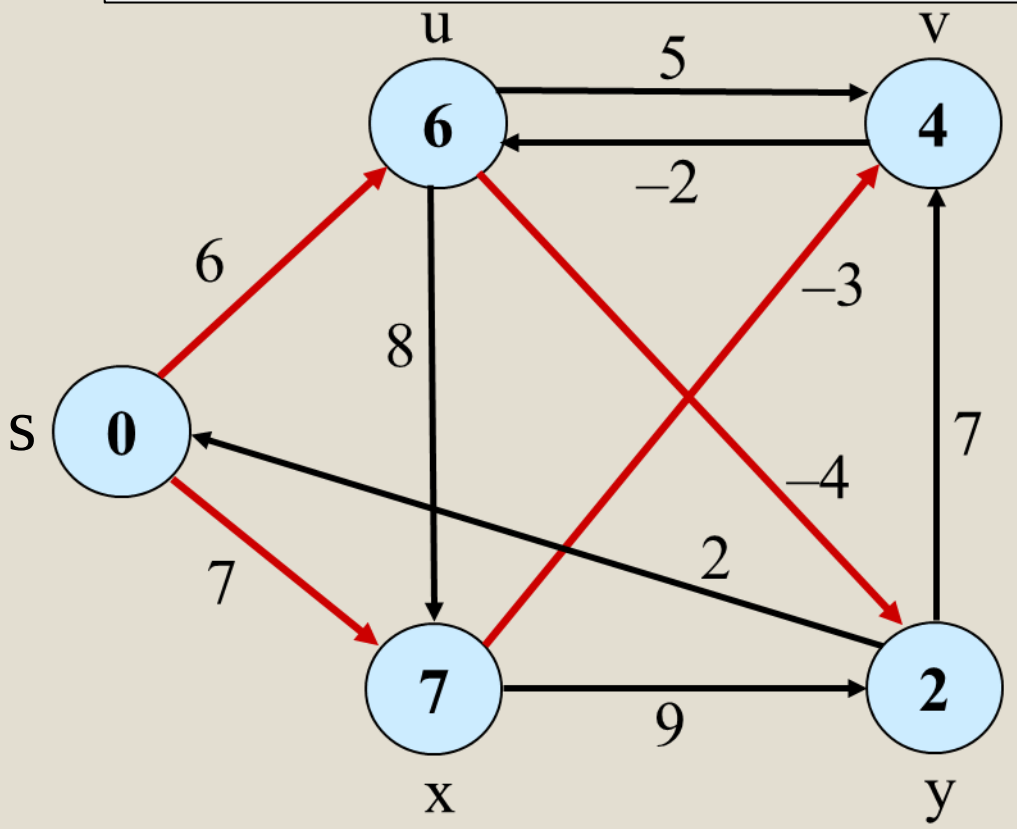
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

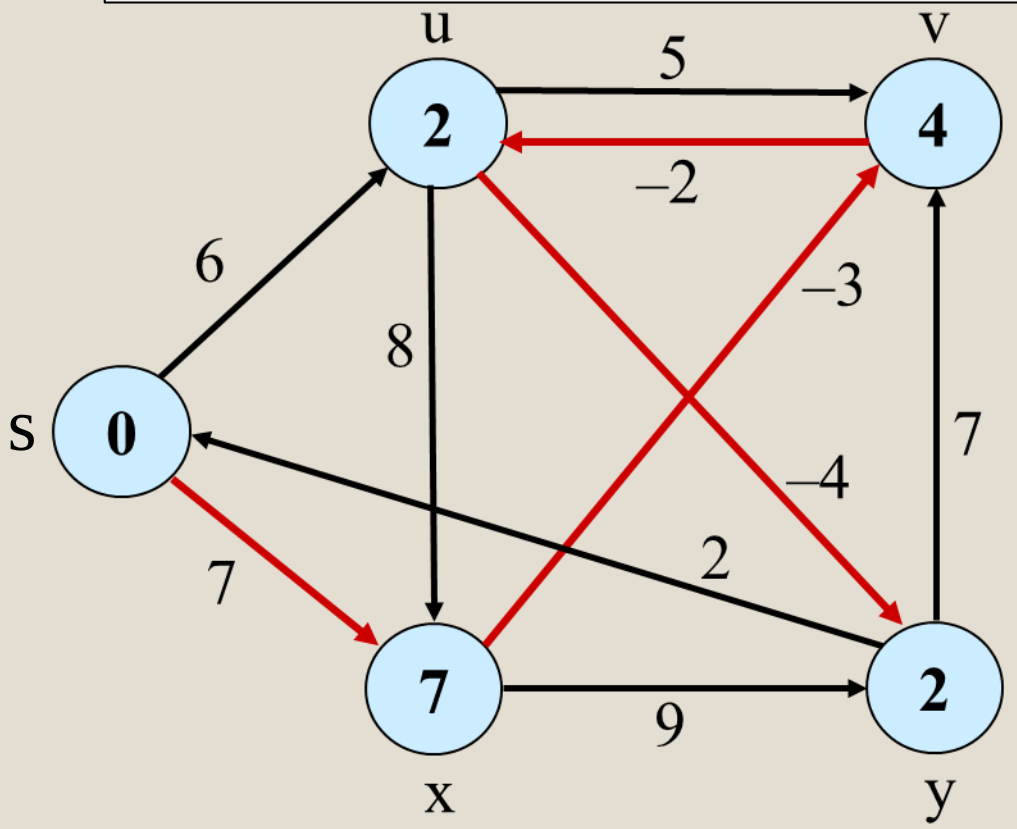
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

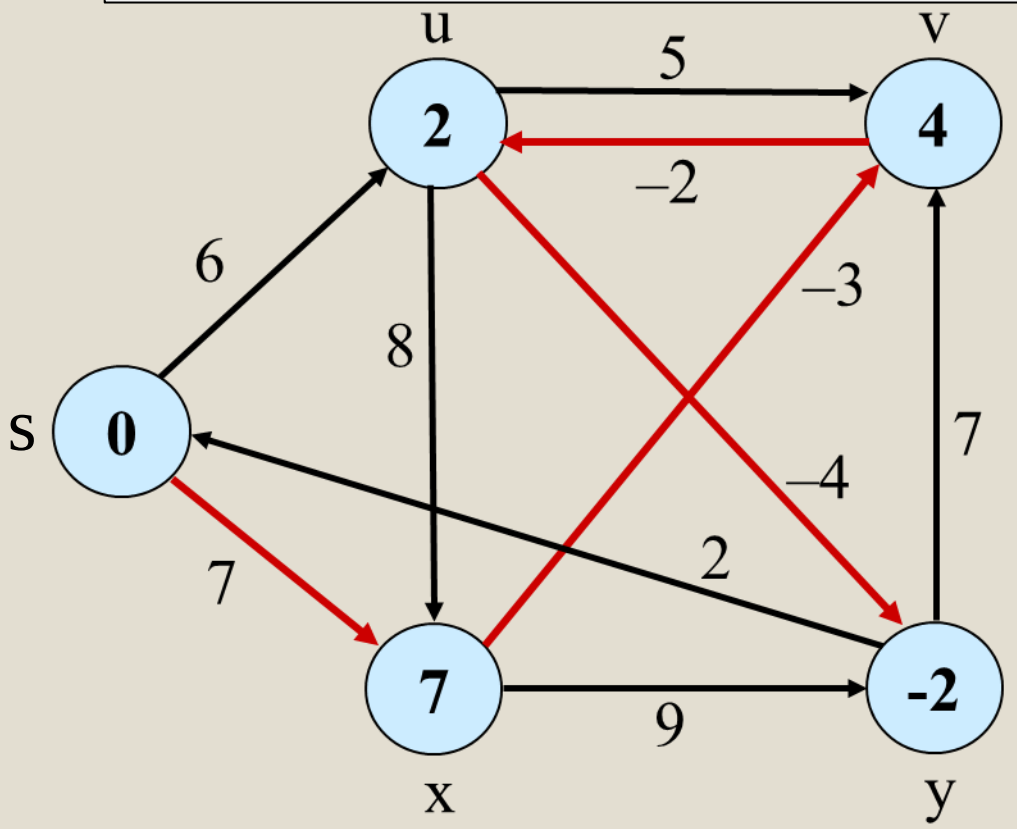
Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Loop invariant for the outer **for** loop:

For every vertex v, $v.d \leq$ shortest-path length from s to v involving at most $i-1$ edges.

Edge ordering: $(u,\cdot), (v,\cdot), (x,\cdot), (y,\cdot), (s,\cdot)$



Example Courtesy: Prof. Jim Anderson

Bellman-Ford Algorithm

The algorithm is based on **dynamic programming** ideas

$d(v, i)$ = weight of the shortest path from s to v that has at most i hops.

$$d(v,i) = \begin{cases} 0 & \text{, if } v = s \text{ and } j = 0 \\ \infty & \text{, if } v \neq s \text{ and } j = 0 \\ \min(\{d(u, i-1) + w(u,v) : v \in Adj[u]\} \cup d(v, i-1)) & \text{, otherwise} \end{cases}$$

Table (that is not explicitly stored) for the example graph:

	s	u	v	x	y
	1	2	3	4	5
0	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	–2

Note: i-th row is the computed distance after the i-th iteration (depends on edge ordering in the CLRS pseudo-code, a better distance may be computed early)

Correctness

A Loop invariant for the outer for loop:

For every vertex v, if there is a shortest path from s to v with at most $i-1$ edges, then $v.d = \delta(s,v)$.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Initialization: $i = 1$, only $\delta(s,s) = 0$ involves at most 0 edges.

IH: At the start of k-th iteration, $v.d = \delta(s,v)$ for all $u \in V$ that has a SP from s involving at most $k-1$ edges.

Maintenance: Assume that a SP from s to v has k edges.

Let u be the predecessor of v in this path. (So, a SP from s to u has $k-1$ edges)

By IH, $u.d = \delta(s,u)$ holds at the start of k-th iteration.

Relaxing edge (u,v) produces $v.d = \delta(s,v)$ (Convergence property Lem. 22.14 in CLRS)

Correctness

Lemma 22.2: Assuming no negative-weight cycles reachable from s, $v.d = \delta(s, v)$ holds upon termination for all vertices v reachable from s.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

- Since there is no negative-weight reachable cycle, any shortest path from s to v has at most $|V| - 1$ edges.
- By the loop invariant, upon termination, if there is a shortest path from s to v with at most $|V| - 1$ edges, then $v.d = \delta(s, v)$.
- So, for all reachable $v, v.d = \delta(s, v)$.

Correctness

Theorem 22.4: If there is no negative-weight cycles reachable from s, then return **true**, otherwise return **false**.

- See from book

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

All-Pair Shortest Paths

Application: Computing distance table for a road atlas.

	Atlanta	Chicago	Detroit	…
Atlanta	-	650	520	
Chicago	650	-	210	
Detroit	520	210	-	
⋮				

One Approach: Run single-source SP algorithm |V| times.

Nonnegative Edges: Use Dijkstra.

Time complexity:

$O(V^3)$ with linear array

$O(VE \lg V)$ with binary heap

Negative Edges: Use Bellman-Ford.

Time Complexity:

$O(V^2E) = O(V^4)$ for dense graphs

Better approach:

Floyd-Warshall: $O(V^3)$, allows negative edges.

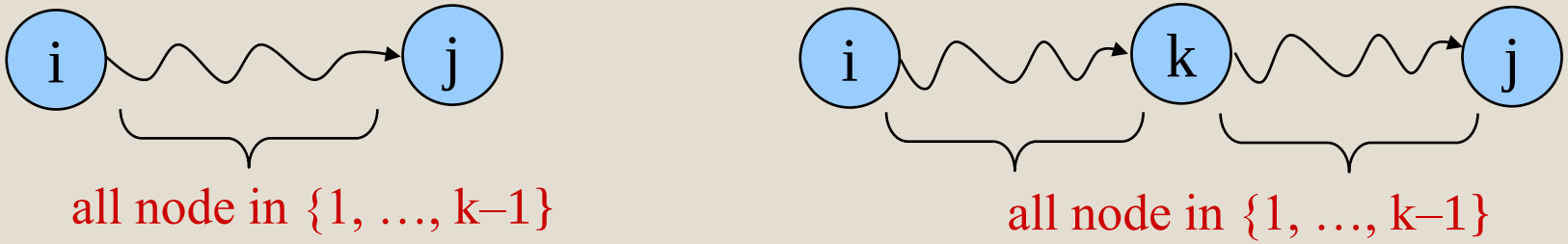
Floyd-Warshall Algorithm

Dynamic programming algorithm

$d_{ij}^{(k)}$ = weight of SP from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$

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

two possibilities



Floyd-Warshall Algorithm

$$\delta(i,j) = d_{ij}^{(n)}$$

So, want to compute $D^{(n)} = (d_{ij}^{(n)})$.

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

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × 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)
```

Exercise:
Run on example graph.

Source: CLRS

Can reduce space from $O(V^3)$ to $O(V^2)$ — see Exercise 25.2-4.
Can also modify to compute predecessor matrix.

Thank You!