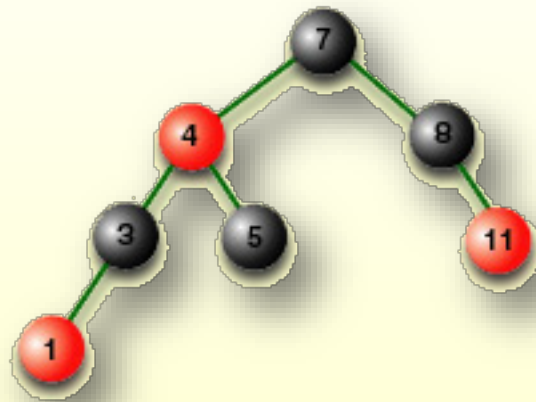


CS161: Design and Analysis of Algorithms



Lecture 15

Leonidas Guibas

Outline

- ◆ Last lecture: Minimum spanning tree algorithms
- ◆ Today: Single source shortest path algorithms
 - ◆ shortest path properties; edge relaxation
 - ◆ Shortest paths on DAGs
 - ◆ Dijkstra's algorithm
 - ◆ Bellman-Ford algorithm

Slides modified from

- <http://www.cs.bilkent.edu.tr/~atat/502/SingleSourceSP.ppt>
- <http://www.cs.unc.edu/.../comp122/>

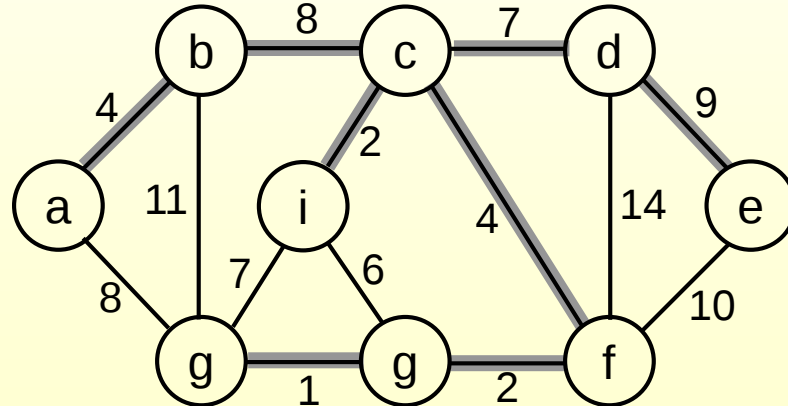
Minimum Spanning Trees

- ◆ Spanning Tree

- ◆ A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph

- ◆ Minimum Spanning Tree

- ◆ Spanning tree with the **minimum sum of weights**



- ◆ Spanning forest

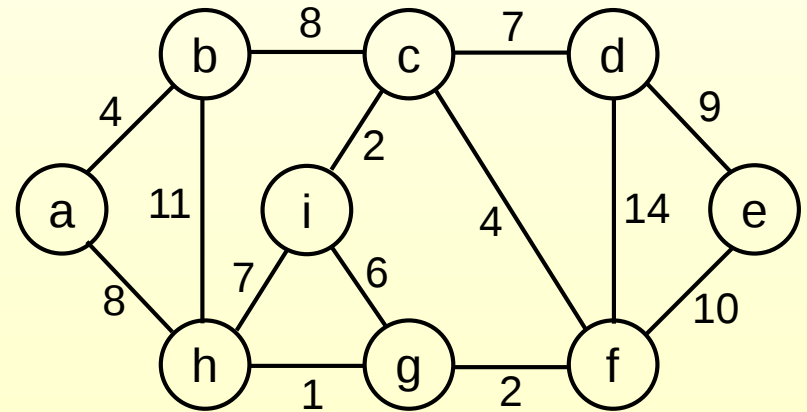
- ◆ If a graph is not connected, then there is a spanning tree for each connected component of the graph

Greedy MST Algorithms

- ◆ Greedy algorithms
 - ◆ iteratively make “myopic” decisions – aimed at locally optimal choice
 - ◆ but somehow everything works out to yield the global optimum at the end
- ◆ While growing a partial MST, an edge not currently in the tree is **safe**, if it can be added while still being part of some MST

Generic MST algorithm

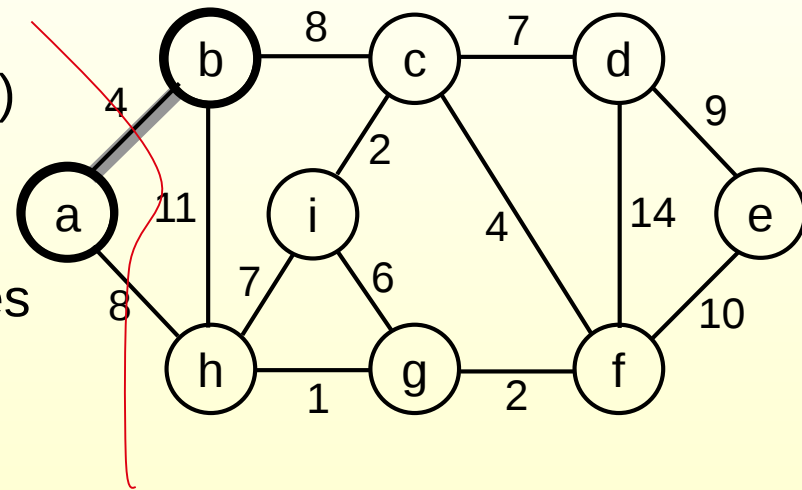
1. $A \leftarrow \emptyset$
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is **safe** for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A



◆ Key: how do we find safe edges?

Prim's Algorithm

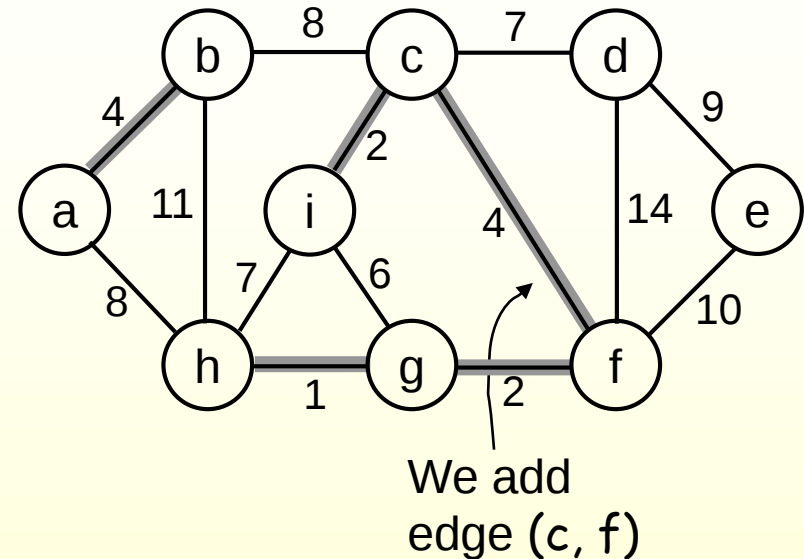
- ◆ The edges in set A always form a single tree
- ◆ Start from an arbitrary “root”: $V_A = \{a\}$
- ◆ At each step:
 - ◆ Find a light edge crossing $(V_A, V - V_A)$
 - ◆ Add this edge to A
 - ◆ Repeat until the tree spans all vertices



Greedy approach

Kruskal's Algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them
- Which components to consider at each iteration?
 - Scan the set of edges in monotonically increasing order by weight (guarantees lightness)



Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many “clouds” at once, but is more “parallel”.

Algorithm *BaruvkaMST*(G)

```
 $T \leftarrow V$  {just the vertices of  $G$ }  
while  $T$  has fewer than  $|V|-1$  edges do  
  for each connected component  $C$  in  $T$  do  
    Let edge  $e$  be the smallest-weight edge from  $C$  to another component in  $T$ .  
    if  $e$  is not already in  $T$  then  
      Add edge  $e$  to  $T$   
return  $T$ 
```

- Each iteration of the while-loop halves the number of connected components in T .
- The running time of all three algorithms is basically $O(E \log V)$.

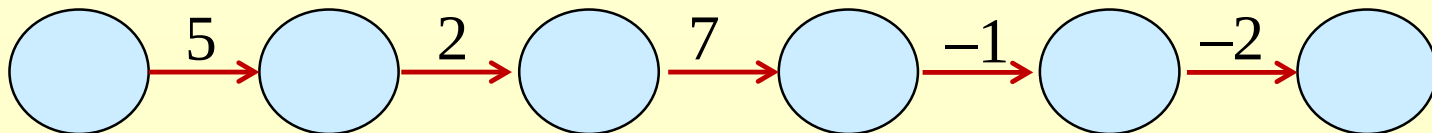
Introduction: Shortest Paths

Generalization of simple BFS to handle weighted graphs

- ◆ Direct Graph $G = (V, E)$, edge weight *function* $w : E \rightarrow \mathbb{R}$
- ◆ In simple BFS, we have $w(e)=1$ for all $e \in E$

Weight of path $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ is

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



Shortest Path

Shortest Path = Path of minimum weight
between two vertices u and v

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\}; & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

Distance from u to v = length of shortest
path from u to v

Shortest-Path Variants

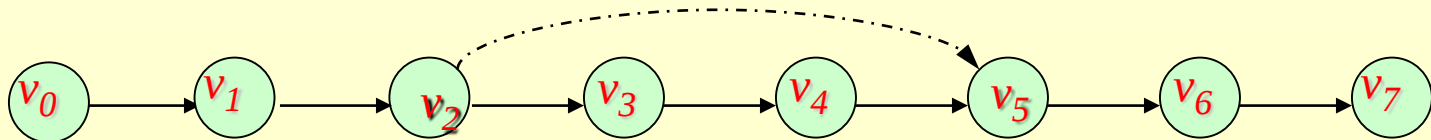
◆ Shortest-Path problems

- ◆ **Single-source shortest-paths problem:** Find the shortest path from s to each vertex v . (e.g. BFS)
- ◆ **Single-destination shortest-paths problem:** Find a shortest path to a given *destination* vertex t from each vertex v .
- ◆ **Single-pair shortest-path problem:** Find a shortest path from u to v for given vertices u and v .
- ◆ **All-pairs shortest-paths problem:** Find a shortest path from u to v for every pair of vertices u and v .

Optimal Substructure Property

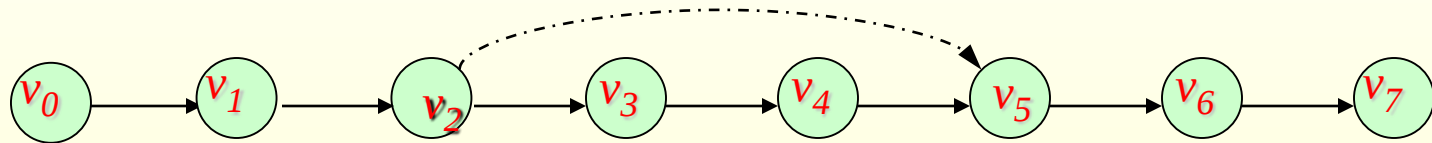
Theorem: Subpaths of shortest paths are also shortest paths

- ◆ Let $P_{1k} = \langle v_1, \dots, v_k \rangle$ be a shortest path from v_1 to v_k
- ◆ Let $P_{ij} = \langle v_i, \dots, v_j \rangle$ be subpath of P_{1k} from v_i to v_j for any $1 \leq i \leq j \leq k$
- ◆ Then P_{ij} is itself a shortest path from v_i to v_j



Optimal Substructure Property

Proof: By cut and paste



- ◆ If some subpath *were not* a shortest path
- ◆ We could substitute a shorter subpath in the original path to create a *shorter total path*
- ◆ Hence, the original path would not be shortest path

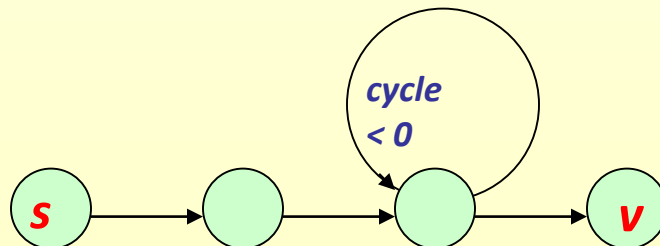
Negative Weight Cycles

Definition:

- ◆ $\delta(u, v)$ = weight of a shortest path(s) from u to v

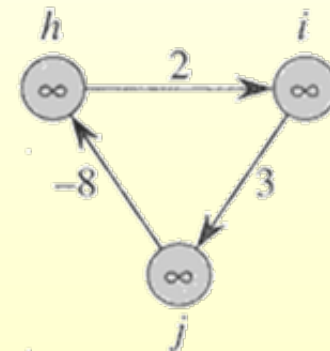
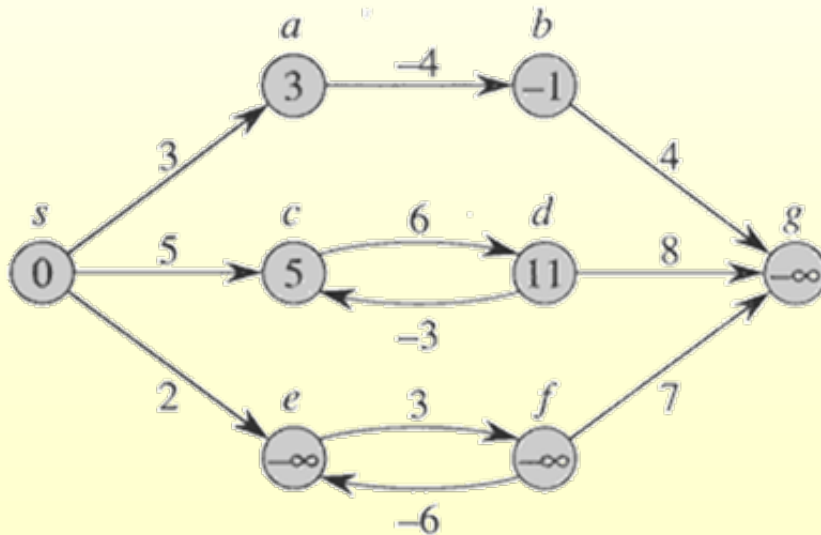
Not always well defined:

- ◆ *negative-weight cycle in graph*: Some shortest paths may not be defined
- ◆ *argument*: can always get a shorter path by going around the negative cycle again



Negative-Weight Edges

- ◆ No problem, as long as no negative-weight cycles are reachable from the source
- ◆ Otherwise, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.



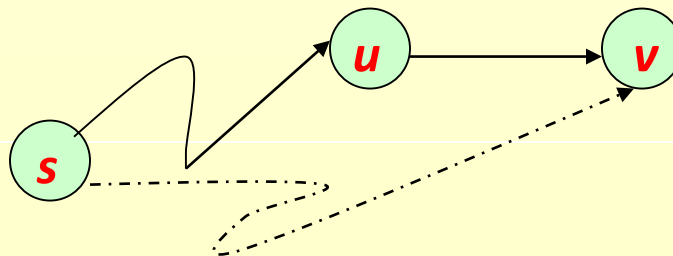
Triangle Inequality

Lemma 1: for a given vertex $s \rightsquigarrow v$ and for every edge $(u,v) \in E$,

◆ $\delta(s,v) \leq \delta(s,u) + w(u,v)$

Proof: shortest path $s \rightsquigarrow v$ is not longer than any other path.

◆ in particular the path that takes the shortest path $s \rightsquigarrow u$ and then takes edge (u,v)



Shortest Path Upper Bounds

- ◆ Maintain $d[v]$ for each $v \rightsquigarrow V$
- ◆ $d[v]$ is called a *shortest-path weight estimate* and is an *upper bound* on $\delta(s, v)$

```
INIT( $G, s$ )
```

```
  for each  $v \in V$  do
```

```
     $d[v] \leftarrow \infty$ 
```

```
     $\pi[v] \leftarrow \text{NIL}$ 
```

```
   $d[s] \leftarrow 0$ 
```

as before, predecessor on
shortest path from s to v

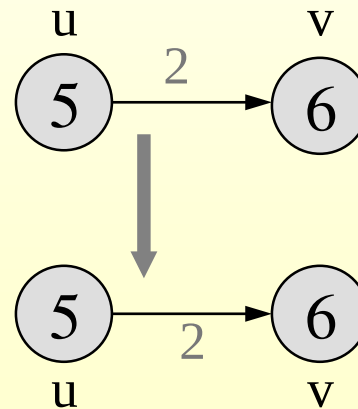
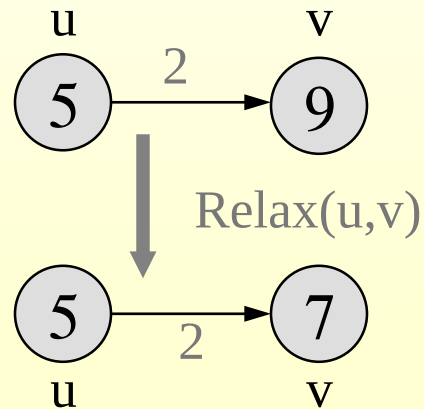
Edge Relaxation

RELAX(u, v)

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

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

$\pi[v] \leftarrow u$



Properties of Relaxation

Shortest path algorithms work by relaxing edges. They differ in

- *how many times* they relax each edge, and
- *the order* in which they relax edges

Question: How many times each edge is relaxed in BFS?

Answer: Only once!

Properties of Relaxation

Given:

- ♦ An edge weighted directed graph $G = (V, E)$ with *edge weight function* ($w: E \rightarrow R$) and a source vertex $s \in V$
- ♦ G is initialized by *INIT*(G, s)

Lemma 2: Immediately after relaxing edge (u,v) ,
 $d[v] \leq d[u] + w(u,v)$

Lemma 3: For any sequence of relaxation steps over E ,
(a) the invariant $d[v] \geq \delta(s,v)$ is maintained
(b) once $d[v]$ achieves its lower bound, it never changes.

Properties of Relaxation

Proof of (a): certainly true after

INIT(G,s) : $d[s] = 0 = \delta(s,s)$; $d[v] = \infty \geq \delta(s,v) \forall v \in V - \{s\}$

- ◆ *Proof by contradiction:* Let v be the **first vertex** for which

RELAX(u, v) causes $d[v] < \delta(s, v)$

- ◆ After **RELAX(u, v)** we have

- $d[u] + w(u,v) = d[v] < \delta(s, v)$
 $\leq \delta(s, u) + w(u,v)$ by **L2**
- $d[u] + w(u,v) < \delta(s, u) + w(u, v) \Rightarrow d[u] < \delta(s, u)$
contradicting the assumption

Properties of Relaxation

Proof of (b):

- ◆ $d[v]$ cannot decrease after achieving its lower bound; because $d[v] \geq \delta(s, v)$
- ◆ $d[v]$ cannot increase since relaxations don't increase d values.

Properties of Relaxation

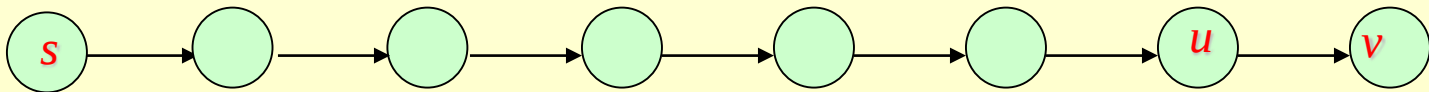
C1 : For any vertex v which is not reachable from s , we have the invariant $d[v] = \delta(s, v)$ that is maintained over any sequence of relaxations

Proof: By *L3(b)*, we always have $\infty = \delta(s, v) \leq d[v]$
 $\Rightarrow d[v] = \infty = \delta(s, v)$

Properties of Relaxation

Lemma 4: Let $s \rightsquigarrow u \rightarrow v$ be a *shortest path* from s to v for some $u, v \rightsquigarrow V$

- Suppose that a sequence of relaxations including **RELAX(u, v)** were performed on E
- If $d[u] = \delta(s, u)$ at any time prior to **RELAX(u, v)**
- then $d[v] = \delta(s, v)$ at all times after **RELAX(u, v)**

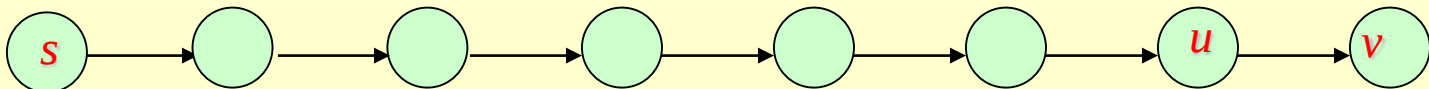


Properties of Relaxation

Proof: If $d[v] = \delta(s, v)$ prior to **RELAX**(u, v)
 $d[v] = \delta(s, v)$ hold thereafter by **L3(b)**

- After **RELAX**(u, v), we have $d[v] \leq d[u] + w(u, v)$ by **L2**
 - $= \delta(s, u) + w(u, v)$ hypothesis
 - $= \delta(s, v)$ by **optimal subst. property**
- Thus $d[v] \leq \delta(s, v)$
- But $d[v] \geq \delta(s, v)$ by **L3(a)** $\Rightarrow d[v] = \delta(s, v)$

Q.E.D.



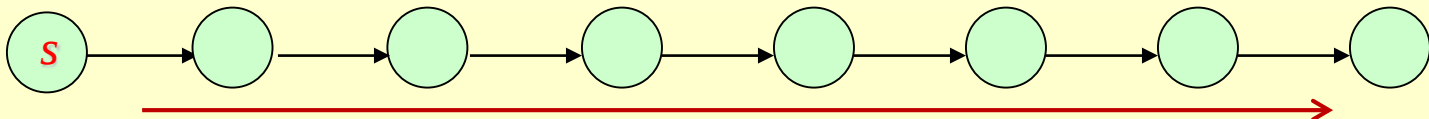
Single-Source Shortest Paths in DAGs

- ◆ Shortest paths are always *well-defined* in *dags*
 - no cycles $\Rightarrow$ no negative-weight cycles even if there are negative-weight edges
- ◆ **Idea:** If we were lucky
 - To process vertices on each shortest path from left to right, we would be done in 1 pass due to *L4*

Single-Source Shortest Paths in DAGs

In a DAG:

- ◆ Every path is a subsequence of the topologically sorted vertex order
- ◆ If we do topological sort and process edges in the order of their origins
- ◆ We will process each path in forward order
 - Never relax edges out of a vertex until have processed all edges into the vertex
- ◆ Thus, just one pass is sufficient



Single-Source Shortest Paths in DAGs

DAG-SHORTEST PATHS(G, s)

TOPOLOGICALLY-SORT the vertices of G

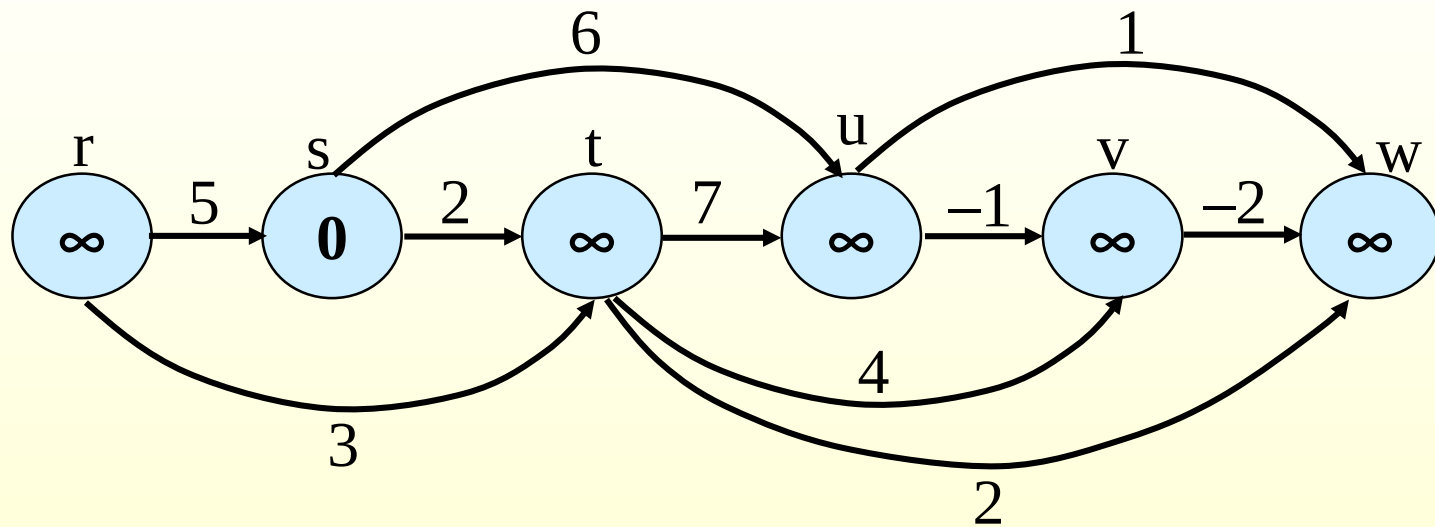
INIT(G, s)

for each vertex u taken in topologically sorted order **do**

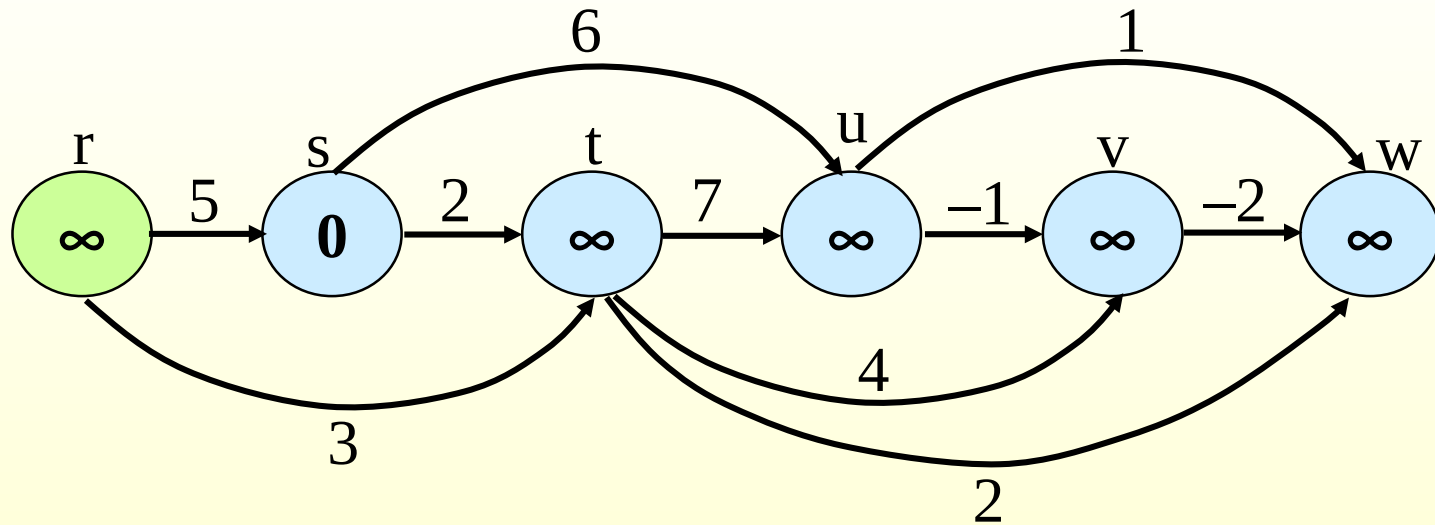
for each $v \rightsquigarrow \text{Adj}[u]$ **do**

RELAX(u, v)

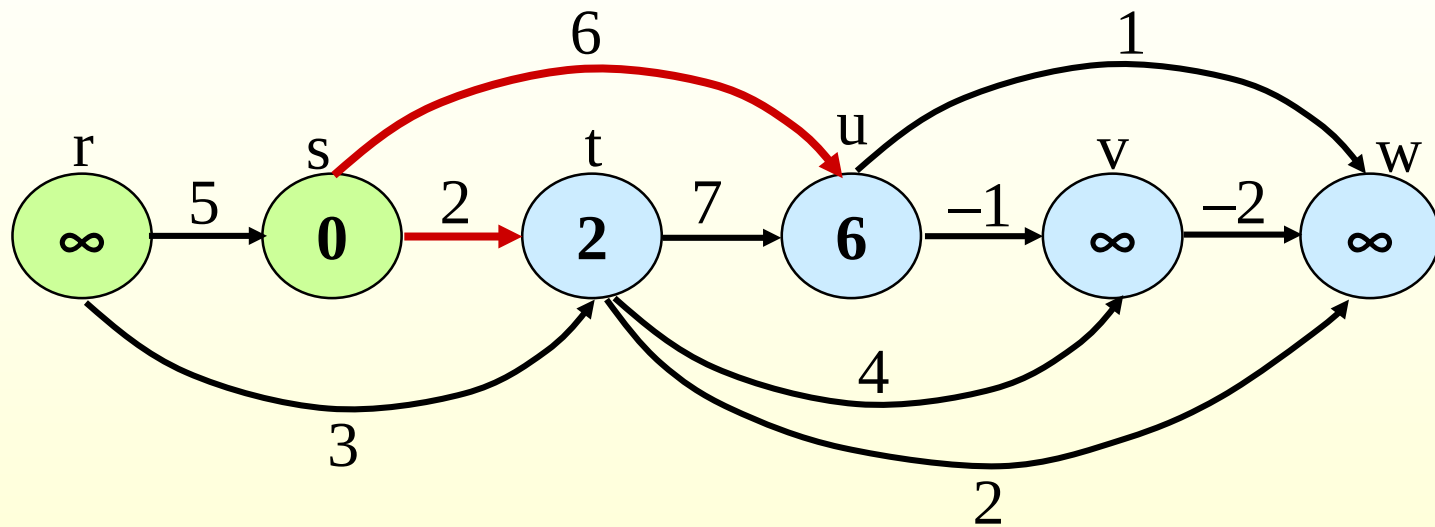
Example



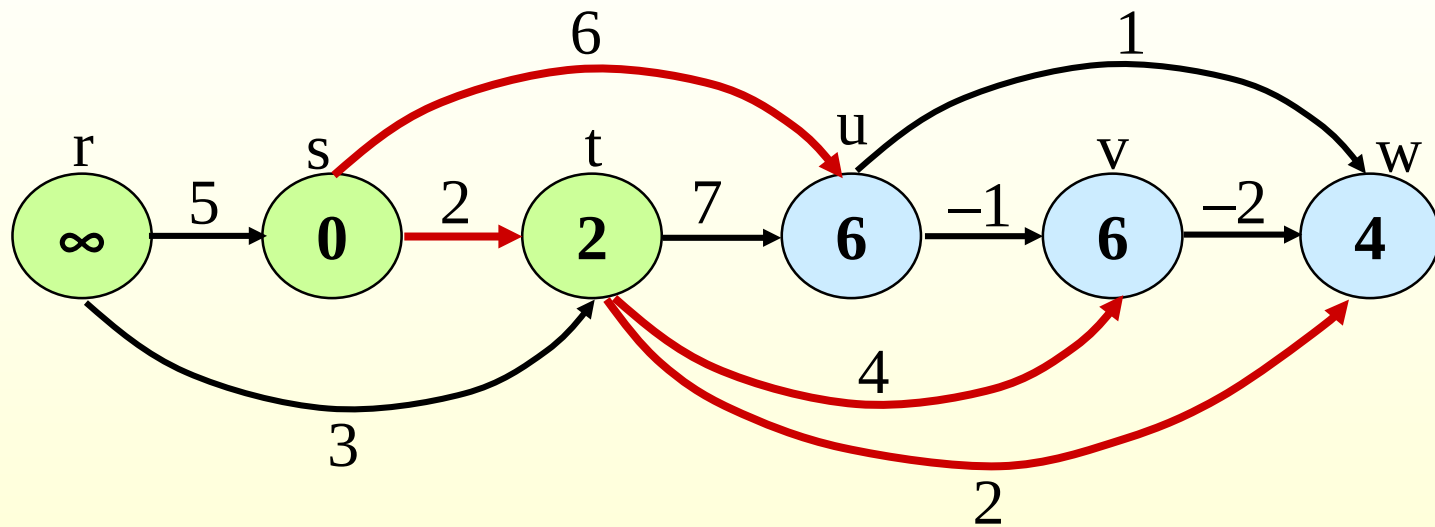
Example



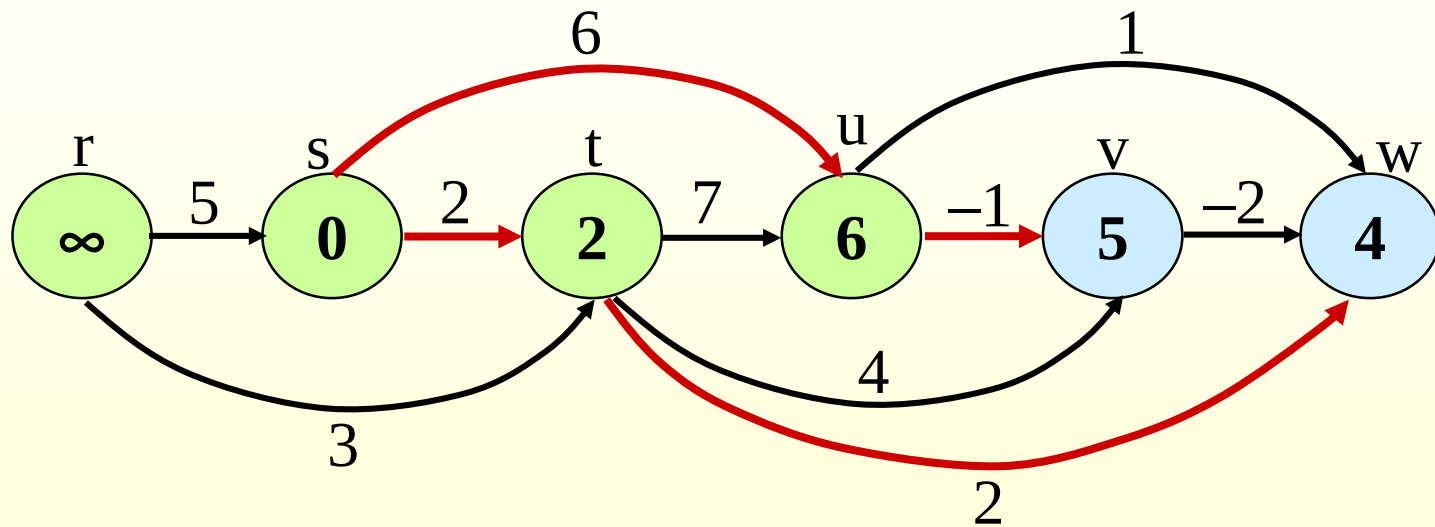
Example



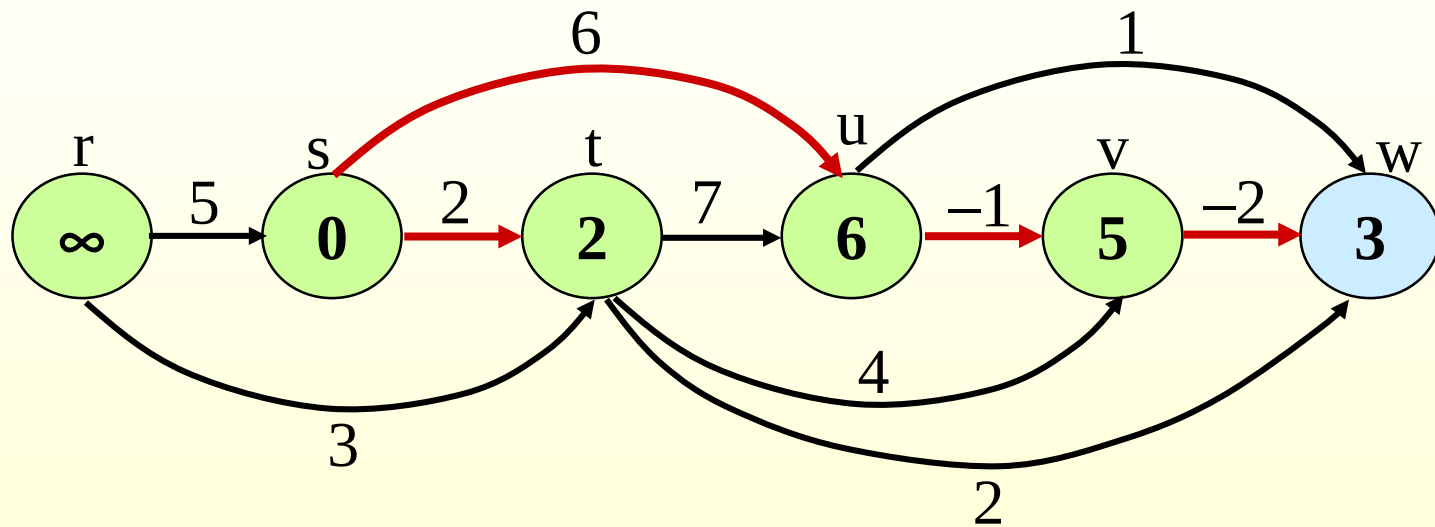
Example



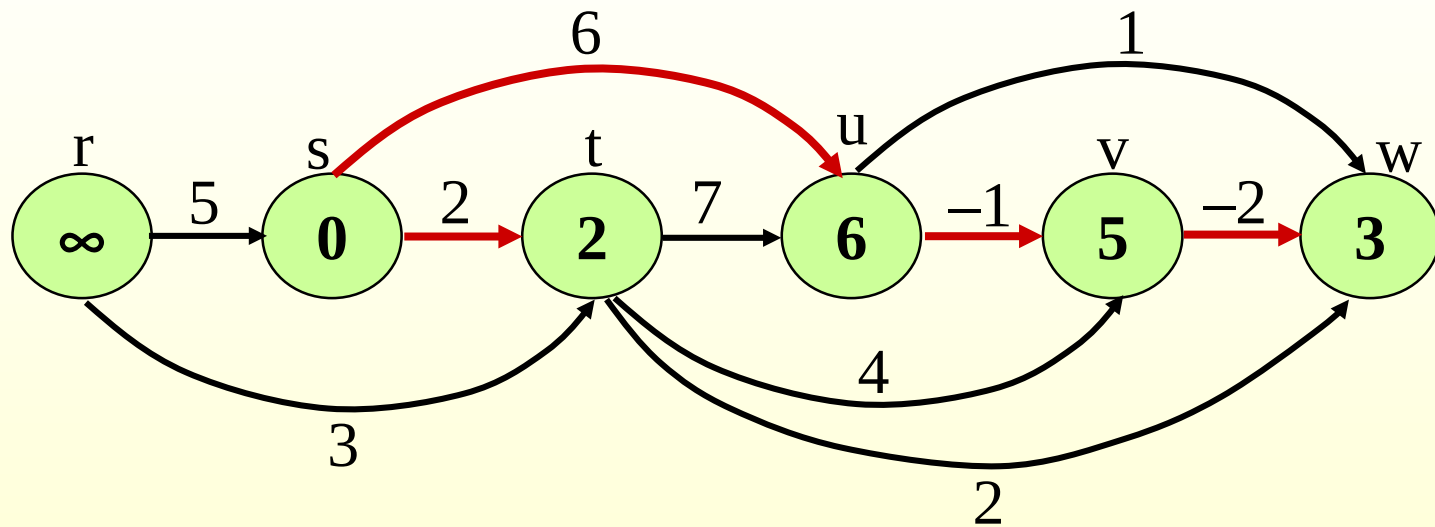
Example



Example



Example



Single-Source Shortest Paths in DAGs

Runs in linear time: $\Theta(V+E)$

- topological sort: $\Theta(V+E)$
- initialization: $\Theta(V+E)$
- *for-loop:* $\Theta(V+E)$
 - each vertex processed exactly once
 - => each edge processed exactly once: $\Theta(V+E)$

Single-Source Shortest Paths in DAGs

Thm: (Correctness of *DAG-SHORTEST-PATHS*):

At termination of *DAG-SHORTEST-PATHS*
procedure

$$d[v] = \delta(s, v) \text{ for all } v \rightsquigarrow V$$

Single-Source Shortest Paths in DAGs

Proof: If $v \in R_s$, *then* $d[v] = \delta(s, v) \quad \forall v \rightsquigarrow V$

- ◆ If $v \in R_s$, so $\exists$ a shortest path
 $p = \langle v_0=s, v_1, v_2, \dots, v_k=v \rangle$
- ◆ Because we process vertices in topologically sorted order
 - Edges on p are relaxed in the order
 $(u_0, u_1), (u_1, u_2), \dots, (u_{k-1}, u_k)$
- ◆ A simple induction on k using *L4* shows that
 - $d[v_i] = \delta(s, v)$ at termination for $i = 0, 1, 2, \dots, k$

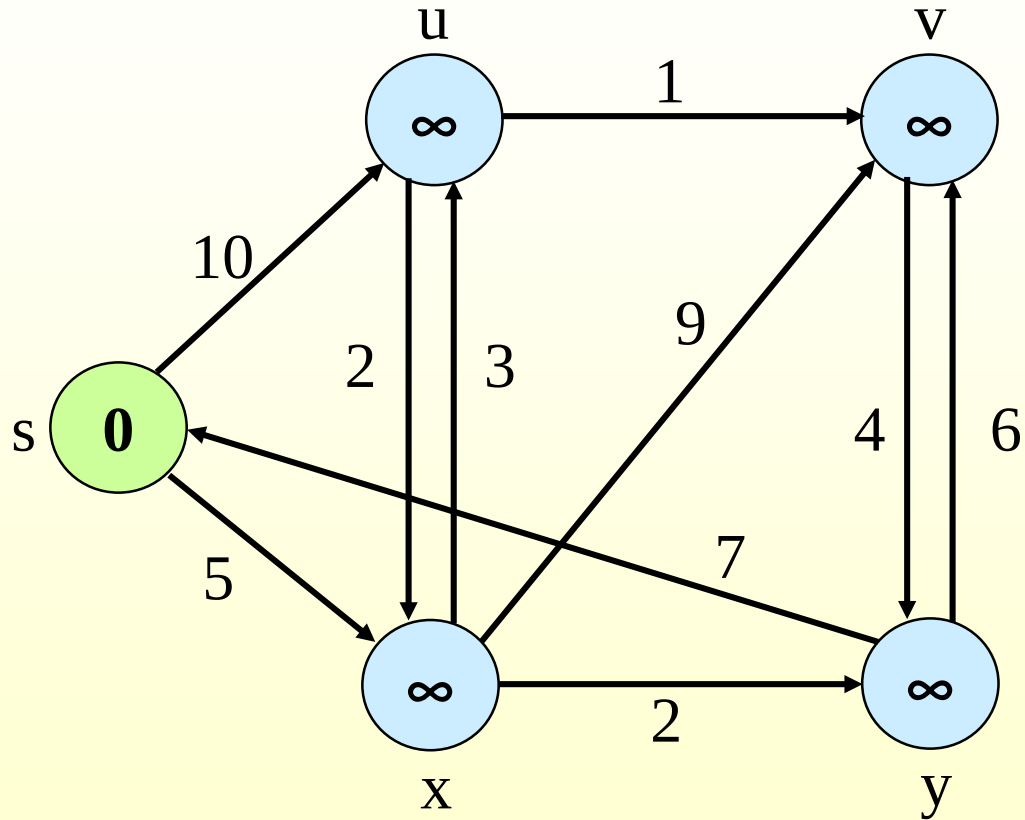
Dijkstra's Algorithm For Shortest Paths

- ◆ Non-negative edge weights
- ◆ Like BFS: If all edge weights are equal, then use BFS, otherwise use this algorithm
- ◆ Use Q = priority queue keyed on $d[v]$ values (note: BFS uses FIFO)

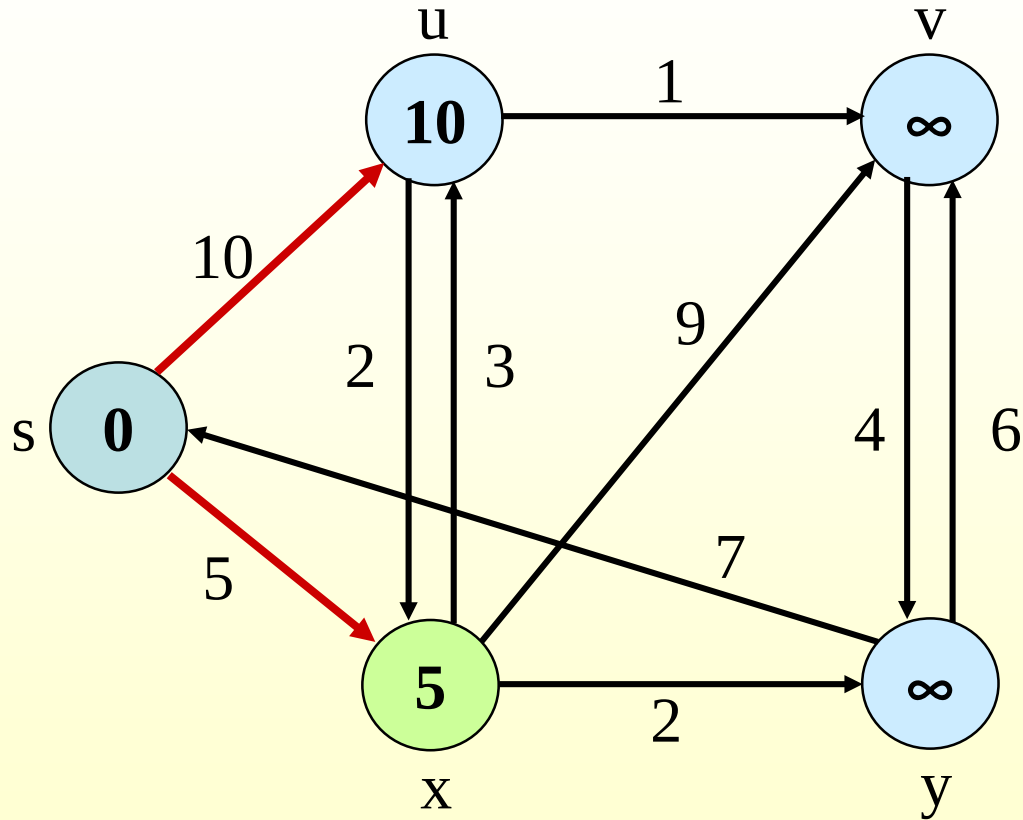
Dijkstra's Algorithm For Shortest Paths

```
DIJKSTRA(G, s)
  INIT(G, s)
  S ← ∅           > set of discovered nodes
  Q ← V[G]
  while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)
    S ← S ∪ {u}
    for each v ~ Adj[u] do
      RELAX(u, v) > may cause
                   > DECREASE-KEY(Q, v, d[v])
```

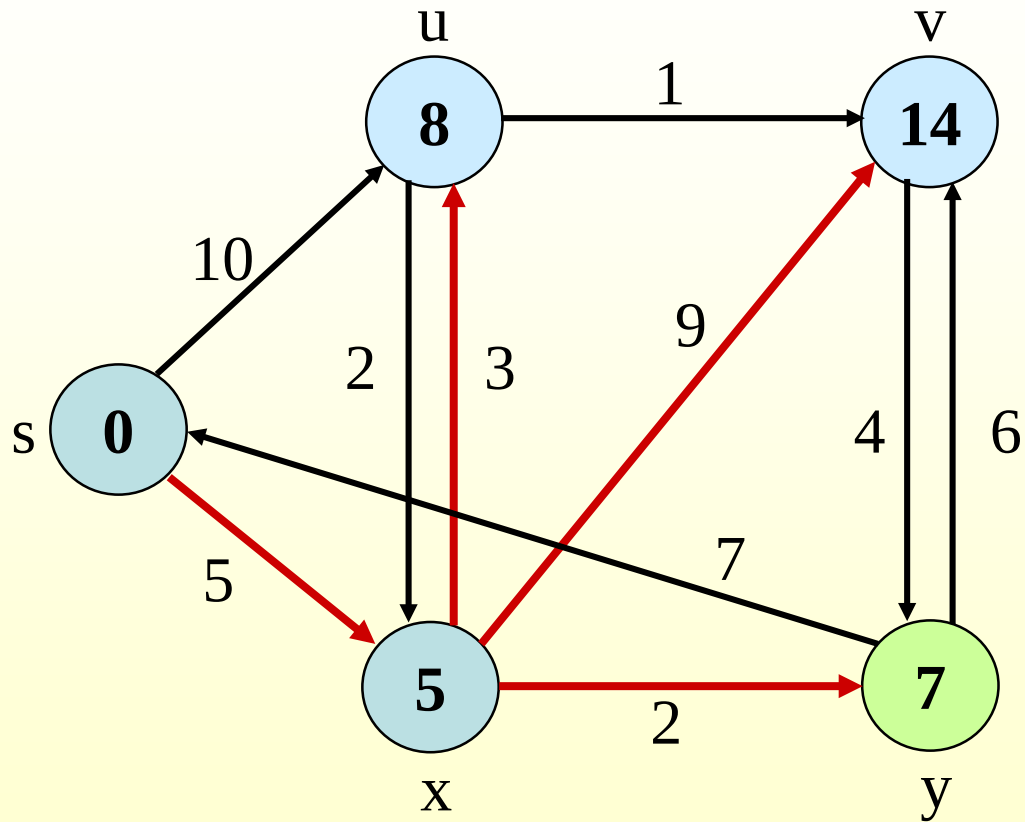
Example



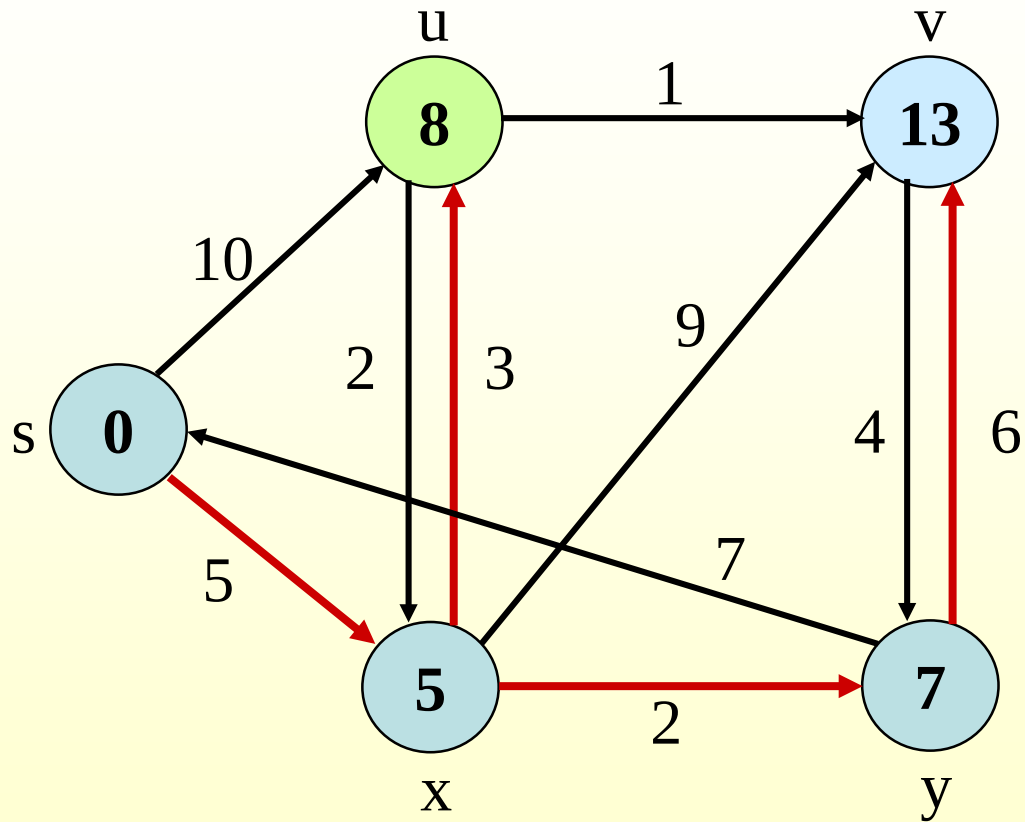
Example



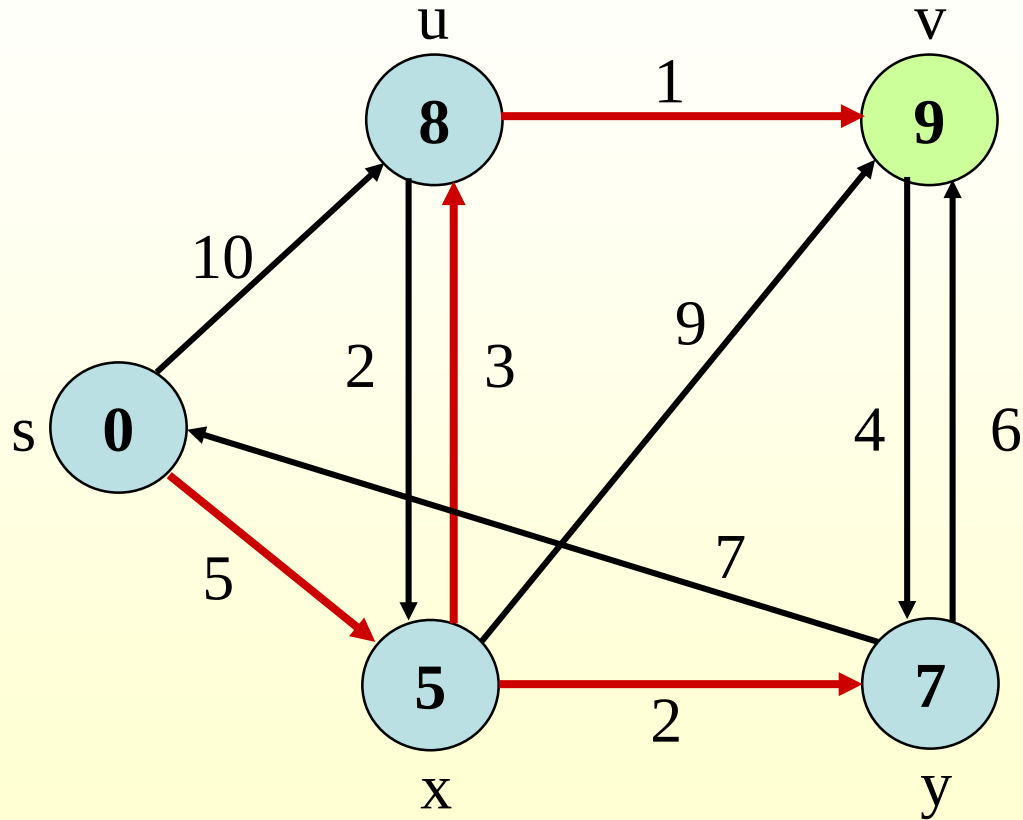
Example



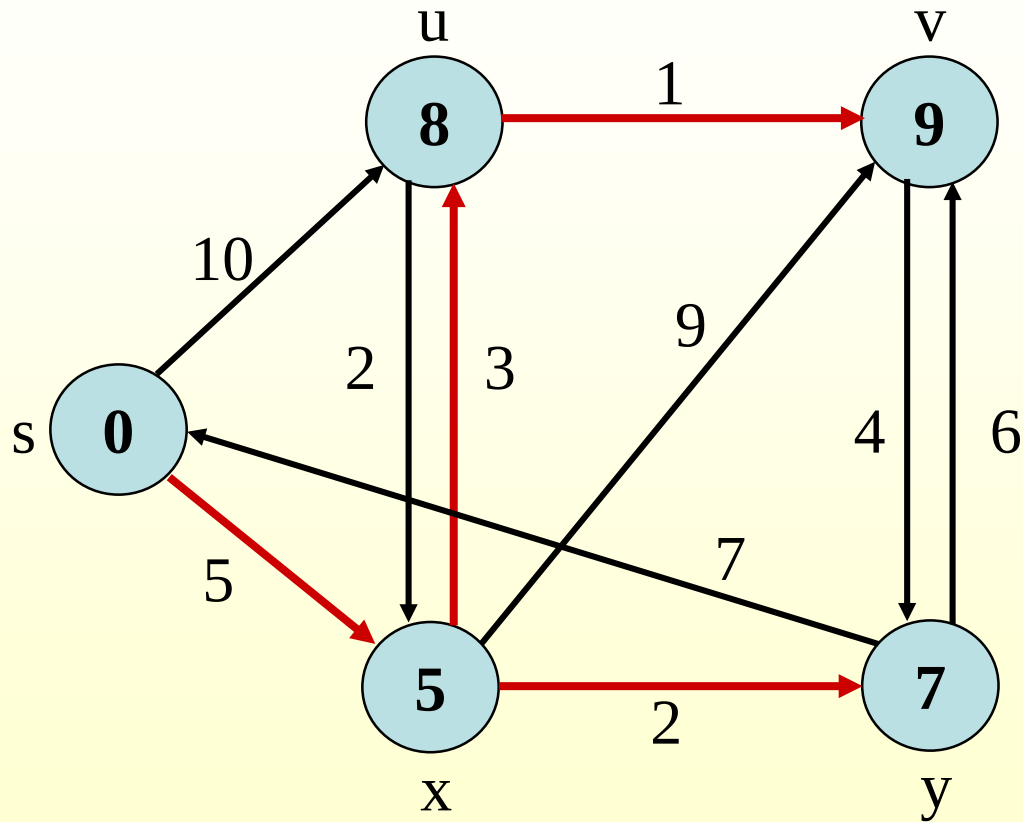
Example



Example



Example



Red edges indicate
shortest path tree from s.

Dijkstra's Algorithm For Shortest Paths

Observe :

- ◆ Each vertex is extracted from Q and inserted into S exactly once
- ◆ Each edge is relaxed exactly once
- ◆ S = set of vertices whose final shortest paths have already been determined
 - i.e. , $S = \{v \rightsquigarrow V: d[v] = \delta(s, v) \neq \infty\}$

Dijkstra's Algorithm For Shortest Paths

- ◆ *Similar to BFS algorithm:* S corresponds to the set of black vertices in **BFS** which have their correct breadth-first distances already computed
- ◆ *Greedy strategy:* Always chooses the **closest (lightest)** vertex in $Q = V - S$ to insert into S
- ◆ **Relaxation** may reset $d[v]$ values thus updating $Q = \text{DECREASE-KEY}$ operation.

Dijkstra's Algorithm For Shortest Paths

- ◆ Similar to Prim's MST algorithm: Both algorithms use a priority queue to find the lightest vertex outside a given set S
- ◆ Insert this vertex into the set
- ◆ Adjust weights of remaining adjacent vertices outside the set accordingly

Correctness

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

Proof:

By Lemma 3(b), once $d[u] = \delta(s, u)$ holds, it continues to hold.

We prove: For each u in V , $d[u] = \delta(s, u)$ when u is inserted in S .

Suppose not. **Let u be the first vertex such that $d[u] \neq \delta(s, u)$ when inserted in S .**

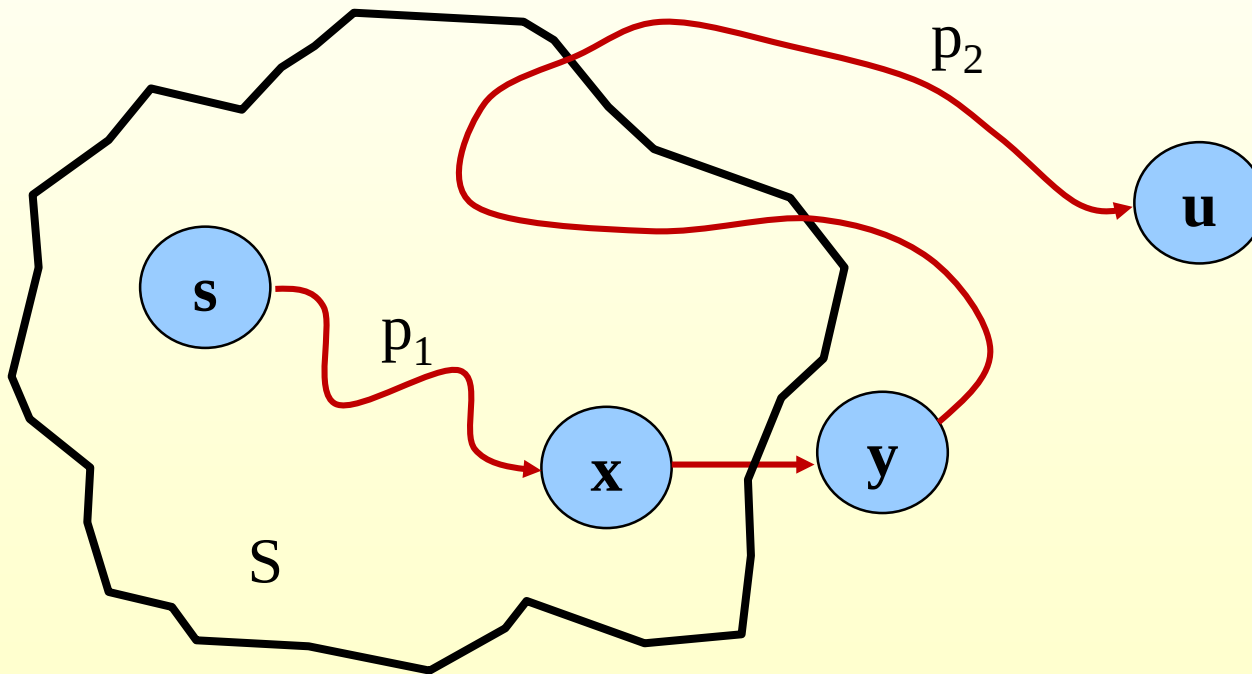
Note that $d[s] = \delta(s, s) = 0$ when s is inserted, so $u \neq s$.

$\Rightarrow S \neq \emptyset$ just before u is inserted (in fact, $s \in S$).

Proof (Continued)

Note that there exists a path from s to u , for otherwise $d[u] = \delta(s, u) = \infty$ by Corollary 24.12.

$\Rightarrow$ there exists a SP from s to u . Say SP looks like this:



Proof (Continued)

Claim: $d[y] = \delta(s, y)$ when u is inserted into S .

We had $d[x] = \delta(s, x)$ when x was inserted into S .

Edge (x, y) was relaxed at that time.

By Lemma 3(b), this implies the claim.

Now, we have: $d[y] = \delta(s, y)$, by Claim.
 $\leq \delta(s, u)$, nonnegative edge weights.
 $\leq d[u]$, by Lemma 3(a).

Because u was added to S before y , $d[u] \leq d[y]$.

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

Contradiction.

Computing Paths (not just Distances)

- ◆ Maintain for each node v a predecessor node $\pi(v)$
- ◆ $\pi(v)$ is initialized to be null
- ◆ Whenever an edge (u,v) is relaxed such that $d(v)$ improves, then $\pi(v)$ can be set to be u
- ◆ Paths can be generated from this data structure

Running Time Analysis of Dijkstra's Algorithm

- ◆ Look at different Q implementation, as we did for Prim's algorithm
- ◆ Initialization (INIT) : $\Theta(V)$ time
- ◆ While-loop:
 - **EXTRACT-MIN** executed $|V|$ times
 - **DECREASE-KEY** executed $|E|$ times
- ◆ Time $T = |V| \times T_{E-MIN} + |E| \times T_{D-KEY}$

Running Time Analysis of Dijkstra's Algorithm

- Look at different Q implementation, as did for Prim's algorithm

Q	T_{E-MIN}	T_{D-KEY}	TOTAL
Linear Unsorted Array:	$O(V)$	$O(1)$	$O(V^2+E)$
Binary Heap:	$O(\lg V)$	$O(\log V)$	$O(V \lg V + E \lg V) = O(E \lg V)$
Fibonacci heap:	$O(\lg V)$ (Amortized)	$O(1)$ (Amortized)	$O(V \lg V + E)$ (Worst Case)

Running Time Analysis of Dijkstra's Algorithm

Q = unsorted-linear array:

- ✦ Scan the whole array for EXTRACT-MIN
- ✦ Joint index for DECREASE-KEY

Q = Fibonacci heap: note advantage of amortized analysis

- ✦ Can use amortized Fibonacci heap bounds per operation in the analysis as if they were worst-case bound
- ✦ Still get (real) worst-case bounds on aggregate running time

Bellman-Ford Algorithm for Single Source Shortest Paths

- More general than Dijkstra's algorithm:
 - Allows edge-weights can be negative
- As a by-product, it detects the existence of negative-weight cycle(s) reachable from s .

Bellman-Ford Algorithm for Single Source Shortest Paths

BELMAN-FORD(G, s)

INIT(G, s)

 for $i \leftarrow 1$ to $|V|-1$ do

 for each edge $(u, v) \in E$ do

RELAX(u, v)

 for each edge $(u, v) \in E$ do

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

 return **FALSE** $>$ neg-weight cycle

 return **TRUE**

Bellman-Ford Algorithm for Single Source Shortest Paths

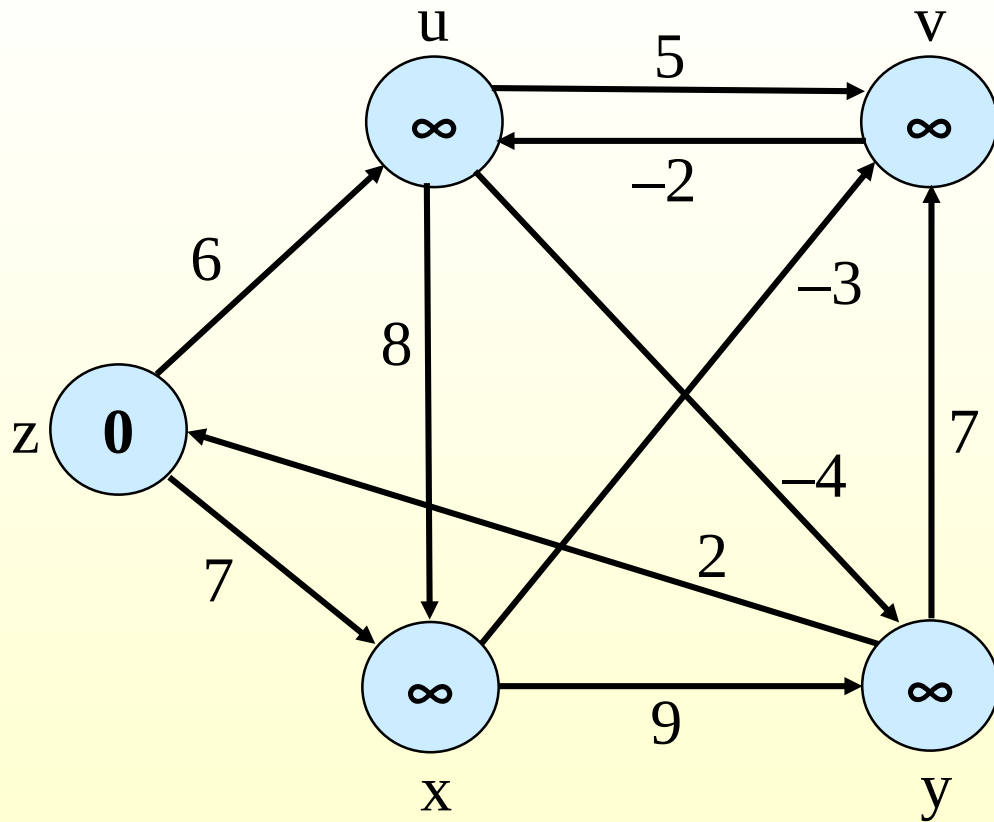
Observe:

- First nested for-loop performs $|V|-1$ relaxation passes; relax every edge at each pass
- Last for-loop checks the existence of a negative-weight cycle reachable from s

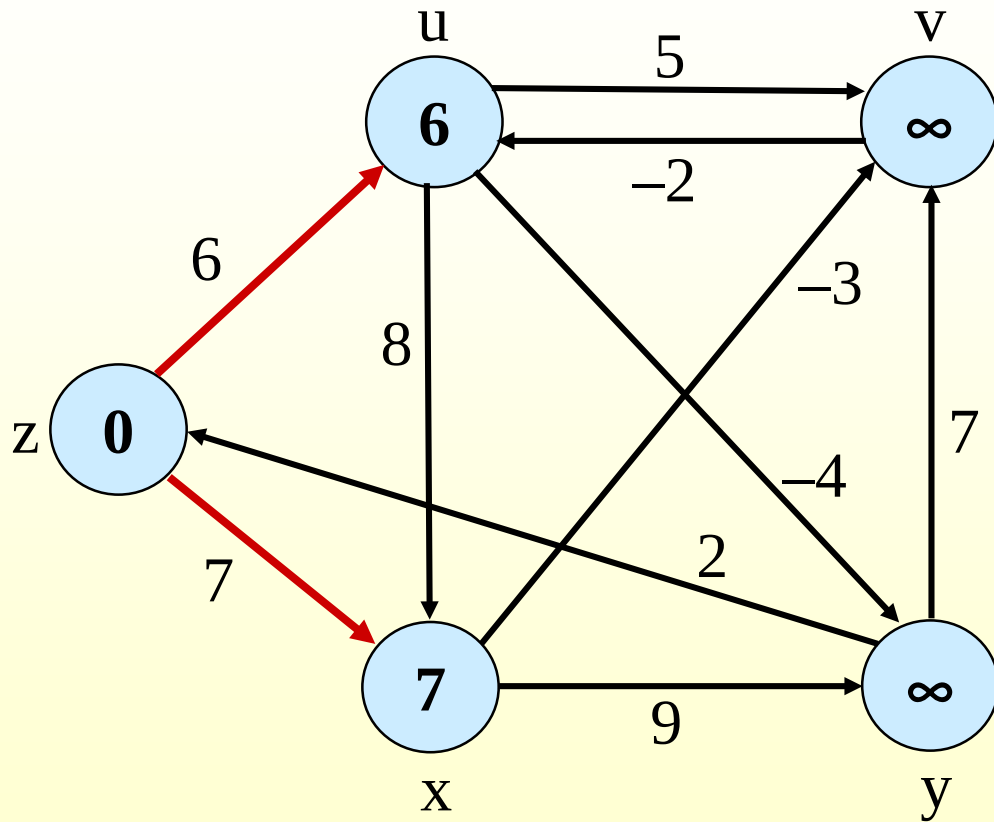
Bellman-Ford Algorithm for Single Source Shortest Paths

- *Running time* = $O(V E)$
Constants are good; it's simple, short code(very practical)
- *Example:* Run algorithm on a sample graph with no negative weight cycles.

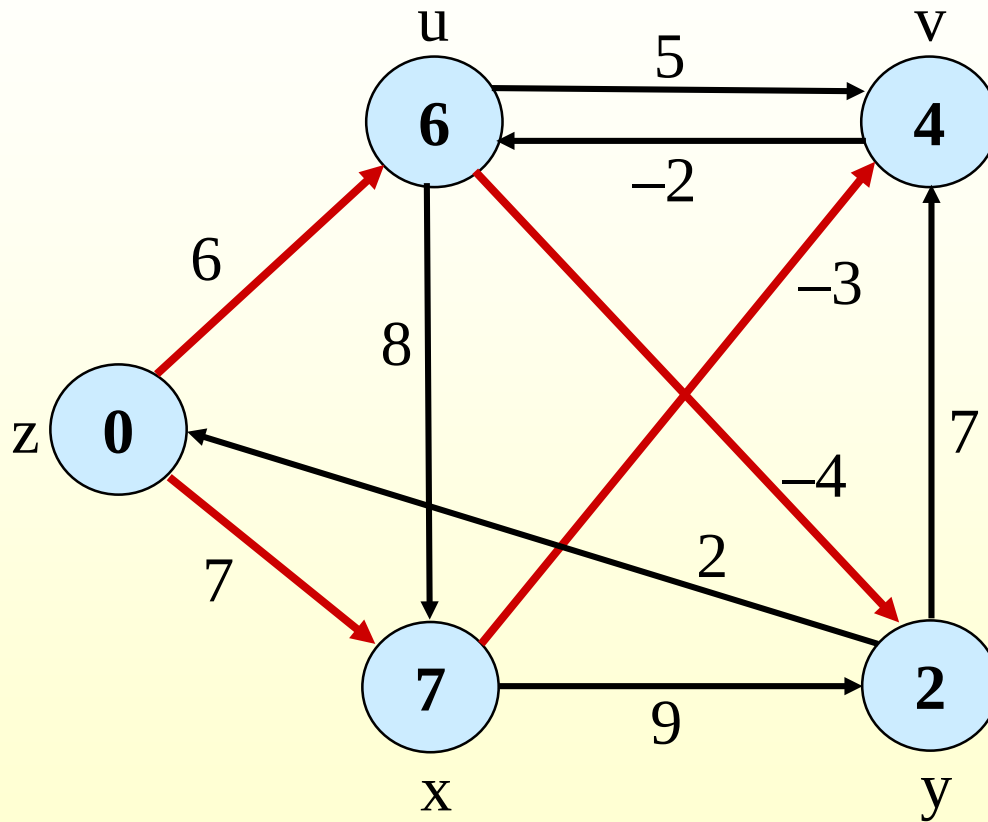
Example



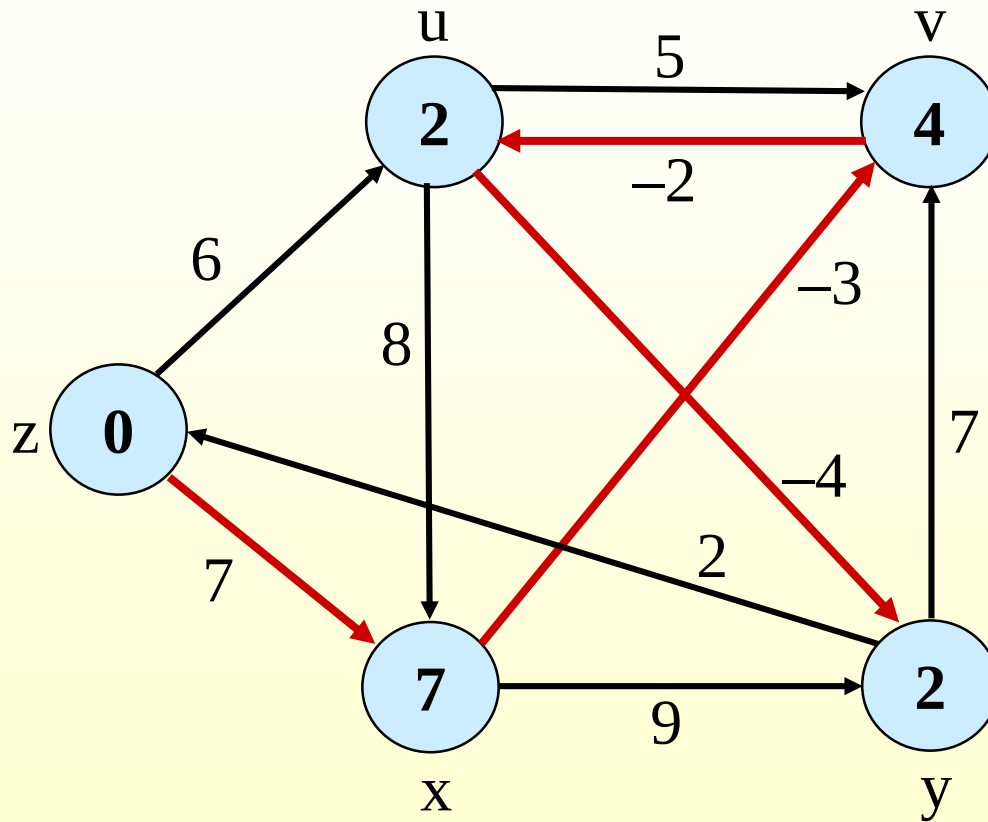
Example



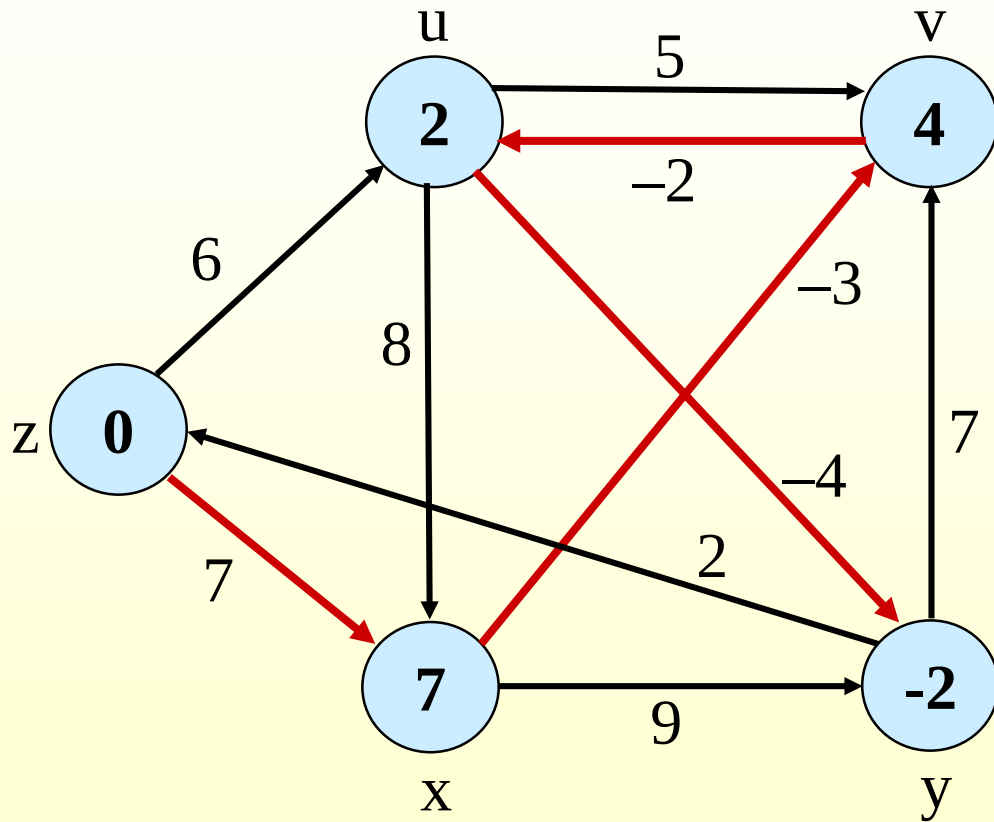
Example



Example



Example



Bellman-Ford Algorithm for Single Source Shortest Paths

- Converges in just 2 relaxation passes
- Values you get on each pass & how early converges depend on edge process order
- d value of a vertex may be updated more than once in a pass

Bellman-Ford Correctness

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

Proof:

Consider a SP p , where $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = s$ and $v_k = v$.

Assume $k \leq |V| - 1$, otherwise p has a cycle.

Claim: $d[v_i] = \delta(s, v_i)$ holds after the i^{th} pass over edges.

Proof follows by induction on i .

By Lemma 3(b), once $d[v_i] = \delta(s, v_i)$ holds, it continues to hold.

Correctness

Claim: Algorithm returns the correct value.

(Part of Theorem 24.4. Other parts of the theorem follow easily from earlier results.)

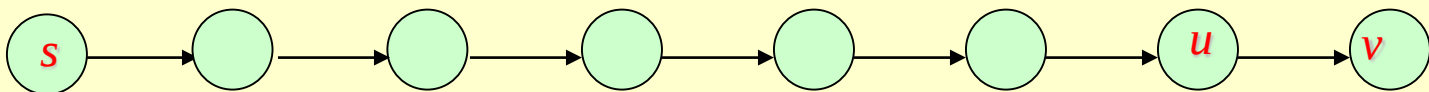
Case 1: There is no reachable negative-weight cycle.

Upon termination, we have for all (u, v) :

$$\begin{aligned} d[v] &= \delta(s, v) && , \text{ if } v \text{ is reachable;} \\ &= \delta(s, v) = \infty && \text{ otherwise.} \end{aligned}$$

$$\begin{aligned} &\leq \delta(s, u) + w(u, v) \\ &= d[u] + w(u, v) \end{aligned}$$

So, algorithm returns **true**.



Case 2

Case 2: There exists a reachable negative-weight cycle $C = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = v_k$.

We have $\sum_{i=1, \dots, k} w(v_{i-1}, v_i) < 0$. (*)

Suppose algorithm returns true. Then, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, \dots, k$. (because Relax didn't change any $d[v_i]$). Thus,

$$\sum_{i=1, \dots, k} d[v_i] \leq \sum_{i=1, \dots, k} d[v_{i-1}] + \sum_{i=1, \dots, k} w(v_{i-1}, v_i)$$

But, $\sum_{i=1, \dots, k} d[v_i] = \sum_{i=1, \dots, k} d[v_{i-1}]$.

Can show no $d[v_i]$ is infinite. Hence, $0 \leq \sum_{i=1, \dots, k} w(v_{i-1}, v_i)$.

Contradicts (*). Thus, algorithm returns **false**.