COT 6405 Introduction to Theory of Algorithms

Topic 16. Single source shortest path

Problem definition

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" -> Weight of the path is minimum
  - Weight of a path is the sum of the weight of edges
  - E.g., a road map: what is the shortest path from USF ENB to USF water tower?

Formal definition

- ***W(p),*** Weight of path $p = (v_0, v_1, \ldots, v_k)$
- ***W(p)*** $= \sum_{i=1}^{k} w(v_{i-1}, v_i)$

  = sum of edge weights on path *p*

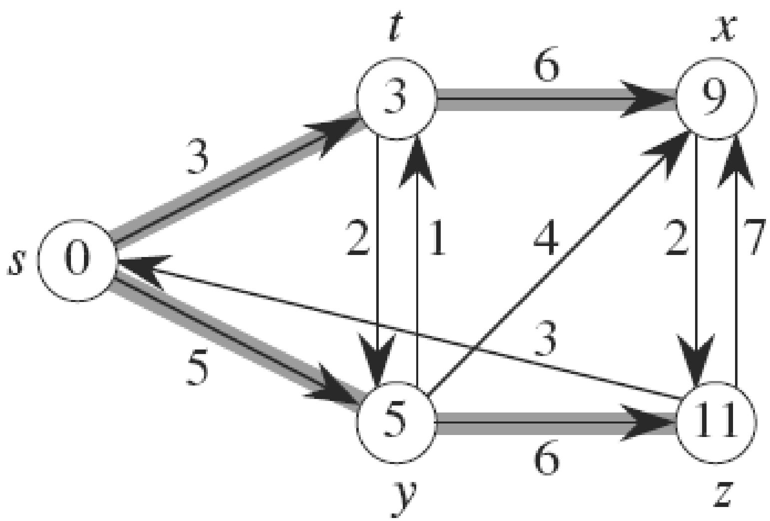
- Shortest-path weight, $\delta(u, v)$, *from* u to v

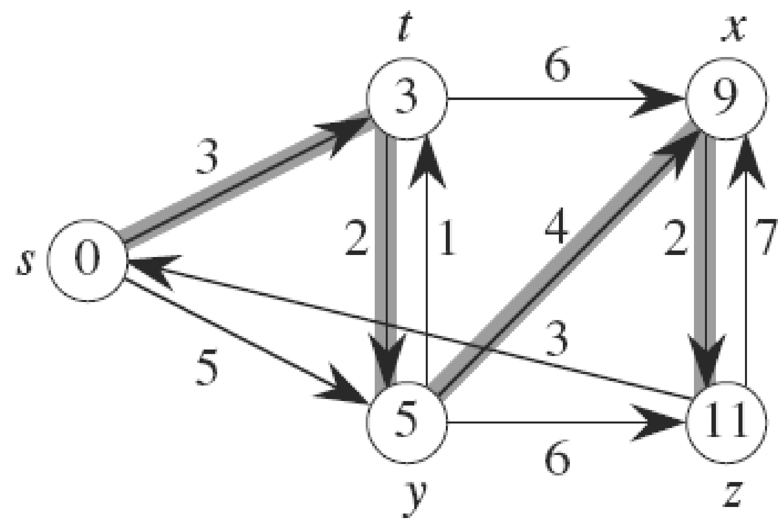
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path } u \rightsquigarrow v\,, \\ \infty & \text{otherwise}\,. \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example: shortest paths from *s*

[d values appear inside vertices. Shaded edges show shortest paths.]





- This example shows that the shortest path might not be unique
- It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Single source shortest path

- We can think of weights as representing any measure that
  - accumulates linearly along a path
  - we want to minimize
- Examples: time, cost, penalties, loss.
- We can use the breadth-first search to find shortest paths for un-weighted graphs

Variants can be solved by SSSP

- ***Single-source**:* Find shortest paths from a given *source* vertex $s \in V$ to every vertex $v \in V$.
- ***Single-destination**:* Find shortest paths to a given destination vertex.
- ***Single-pair**:* Find shortest path from u to v.
- ***All-pairs**:* Find shortest path from u to v for all $u, v \in V$.

Shortest path properties: optimal

Lemma

Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.

Suppose this path p is a shortest path from u to v.

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\rightsquigarrow} y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p':

Cont'd

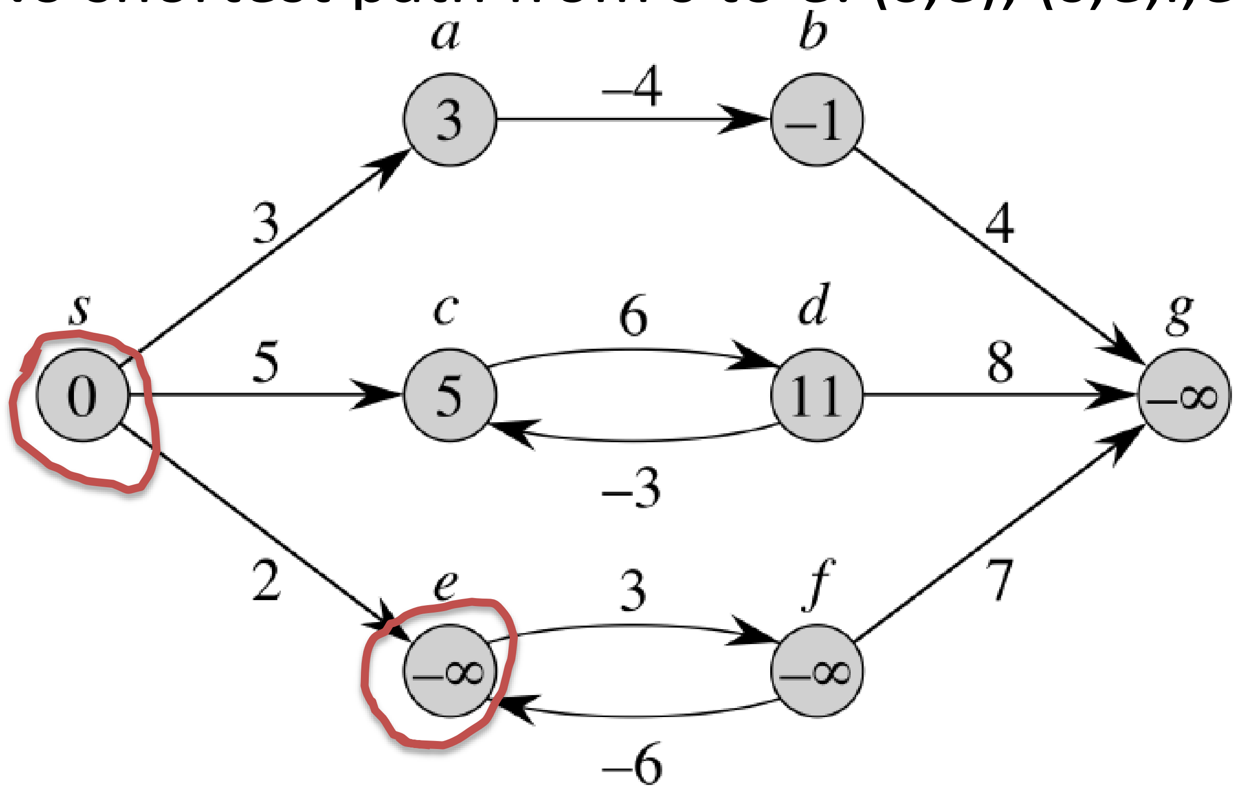
Then

$$
\begin{aligned}
w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\
&< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\
&= w(p) .
\end{aligned}
$$

Contradicts the assumption that p is a shortest path.

Shortest path properties

- In graphs with negative weight cycles, some shortest paths will not exist:
  - No shortest path from s to e: (s,e), (s,e,f,e), …



Negative-weight edges

- Negative weight edges are ok for some cases
  - as long as no negative-weight cycles are reachable from the source
  - If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- Some algorithms work only if there are no negative-weight edges in the graph.
  - Dijkstra algorithm works on nonnegative weights
  - We'll be clear when they're allowed and not allowed
- Normally, we assume nonnegative weights

Cycles

- Shortest paths cannot contain cycles:
  - We already ruled out negative-weight cycles
  - Positive-weight cycle ⇒ Removing the cycle will give us a path with less weight
  - Zero-weight cycle: no reason to use them ⇒ assume that our solutions won’t use them.

Output of SSSP algorithm

- For each vertex $v \in V$, $v.d = \delta(s, v)$
  - Initially, v.$d = \infty$
  - Reduces as algorithms progress. But always maintain v.$d \geq \delta(s, v)$
  - Call v.d a *shortest-path estimate*
- $\pi[v]$ = predecessor of v on a shortest path from s
  - If no predecessor, $\pi[v]$ = NIL.
  - π induces a tree: *shortest-path tree*.

Initialization

- All the shortest-paths algorithms start with INIT-SINGLE-SOURCE

INIT-SINGLE-SOURCE*(G, s)*

for each vertex $v \in G.V$

v.d = ∞

$v.\pi =$ NIL

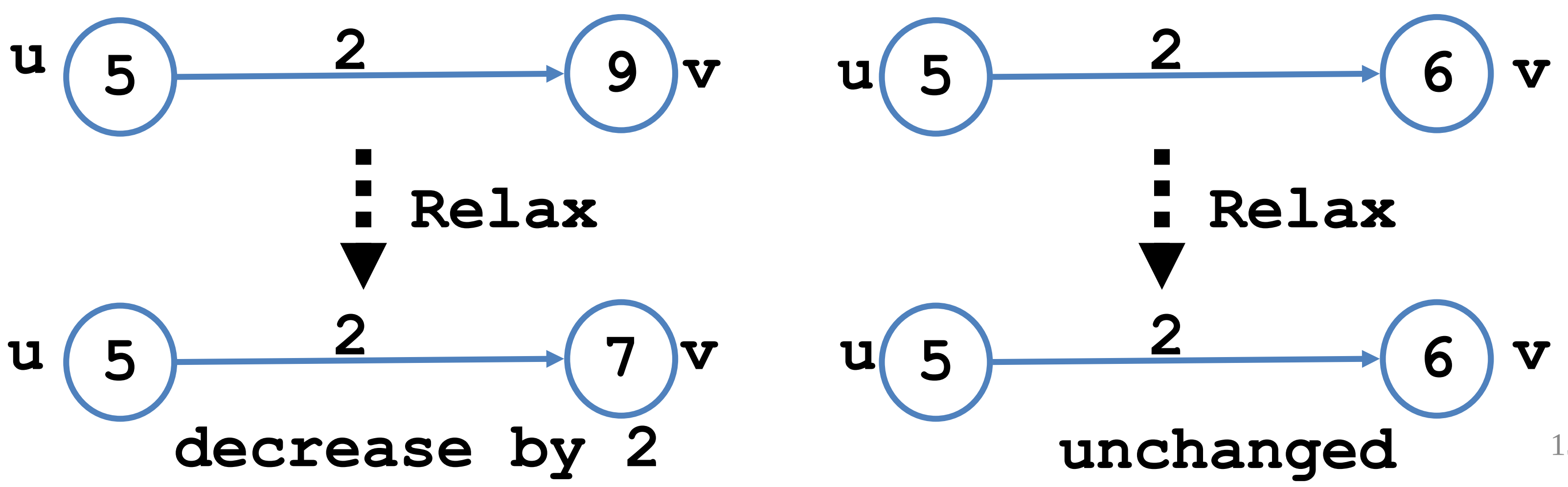
$s.d =$ 0

Initialization

- For all the single-source shortest-paths algorithms we’ll look at,
  – start by calling INIT-SINGLE-SOURCE,
  – then relax edges by decreasing the path weight if possible
- The algorithms differ in the order and how many times they relax each edge.

Relaxation: reach v by u

```
Relax(u, v, w) {
    if (v.d > u.d + w(u,v))
        v.d = u.d + w(u,v)
        v.π = u
}
```

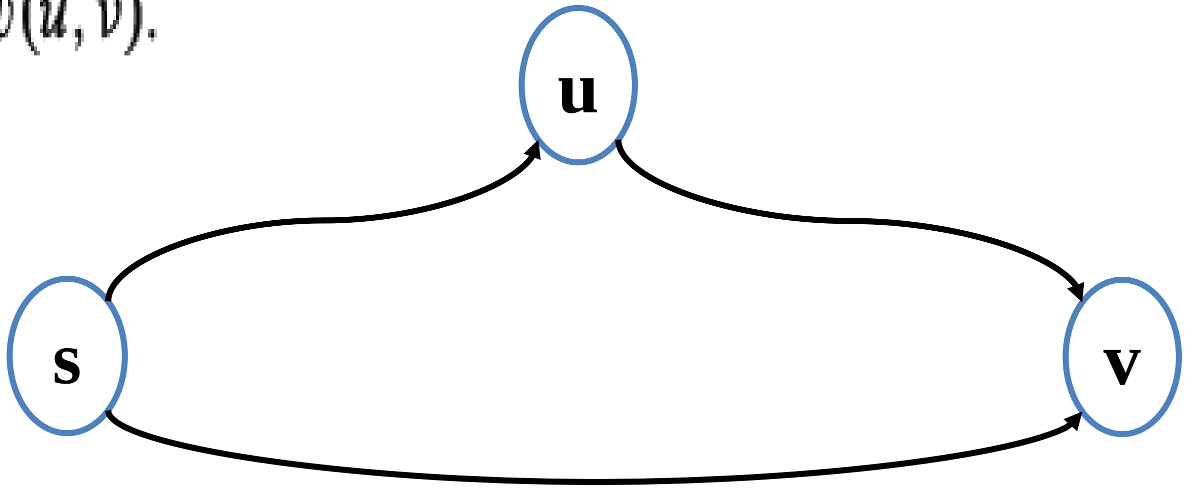


Properties of shortest paths

- Triangle inequality

For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof Weight of shortest path $s \leadsto v$ is $\leq$ weight of any path $s \leadsto v$. Path $s \leadsto u \to v$ is a path $s \leadsto v$, and if we use a shortest path $s \leadsto u$, its weight is $\delta(s, u) + w(u, v)$. ■



Upper-bound property

- Always have v.d $\geq \delta(s,v)$
  - Once v.d = $\delta(s,v)$, it never changes
- Proof: Initially, it is true: v.d = ∞
- Supposed v.d < $\delta(s,v)$
- Without loss of generality, v is the first vertex for this happens
- Let u be the vertex that causes v.d to change
- Then v.d = u.d + w(u,v)
- So, v.d < $\delta(s,v) \leq \delta(s,u)$ + w (u,v) < u.d + w(u,v)
- Then v.d < u.d + w(u,v)
- Contradict to v.d = u.d + w(u,v)

No-path property

- If $\delta(s,v) = \infty$, then v.d = ∞ always
- Proof: v.d $\geq \delta(s,v) = \infty \rightarrow$ v.d = ∞

Convergence property

If $s \leadsto u \to v$ is a shortest path, $u.\mathbf{d} = \delta(s, u)$, and we call RELAX$(u, v, w)$, then $v.\mathbf{d} = \delta(s, v)$ afterward.

Proof After relaxation:

$$
\begin{aligned}
v.\mathbf{d} &\le u.\mathbf{d} + w(u, v) && \text{(RELAX code)} \\
&= \delta(s, u) + w(u, v) \\
&= \delta(s, v) && \text{(lemma—optimal substructure)}
\end{aligned}
$$

Since $v.\mathbf{d} \ge \delta(s, v)$, must have $v.\mathbf{d} = \delta(s, v)$. ■

Path relaxation property

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be **a shortest path** from $s = v_0$ **to** v_k. If we **relax, in order**, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even **intermixed** with **other relaxations**, then $v_k.\mathrm{d} = \delta(s, v_k)$.

Proof **Induction to show that** $v_i.\mathrm{d} = \delta(s, v_i)$ **after** (v_{i-1}, v_i) **is relaxed.**

Basis: $i = 0$. **Initially**, $v_0.\mathrm{d} = 0 = \delta(s, v_0) = \delta(s, s)$.

Inductive step: Assume $v_{i-1}.\mathrm{d} = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i). **By convergence property**, $v_i.\mathrm{d} = \delta(s, v_i)$ **afterward and** $v_i.\mathrm{d}$ **never changes.** ■

Bellman-Ford Algorithm

- Allows negative-weight edges.
- Computes *v.d* and v.π for all $v \in V$.
- Returns
  - TRUE, if no negative-weight cycles reachable from *s* ;
  - FALSE, otherwise.

Bellman-Ford algorithm

```
BellmanFord(G, w, s)
   INIT-SINGLE-SOURCE(G, s)
    for i=1 to |G.V|-1
       for each edge (u,v) ∈ G.E
          Relax(u, v, w);
    for each edge (u,v) ∈ G.E
       if (v.d > u.d + w(u,v))
            return "no solution";
```

Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution Under what condition do we get a solution?

```
Relax(u,v,w): if (v.d > u.d + w(u,v))
                             v.d = u.d + w(u,v)
```

Bellman-Ford Algorithm

```
BellmanFord(G, w, s)
   INIT-SINGLE-SOURCE(G, s)
    for i=1 to |G.V|-1
       for each edge (u,v) ∈ G.E
          Relax(u, v, w);
    for each edge (u,v) ∈ G.E
       if (v.d > u.d + w(u,v))
            return "no solution";
```

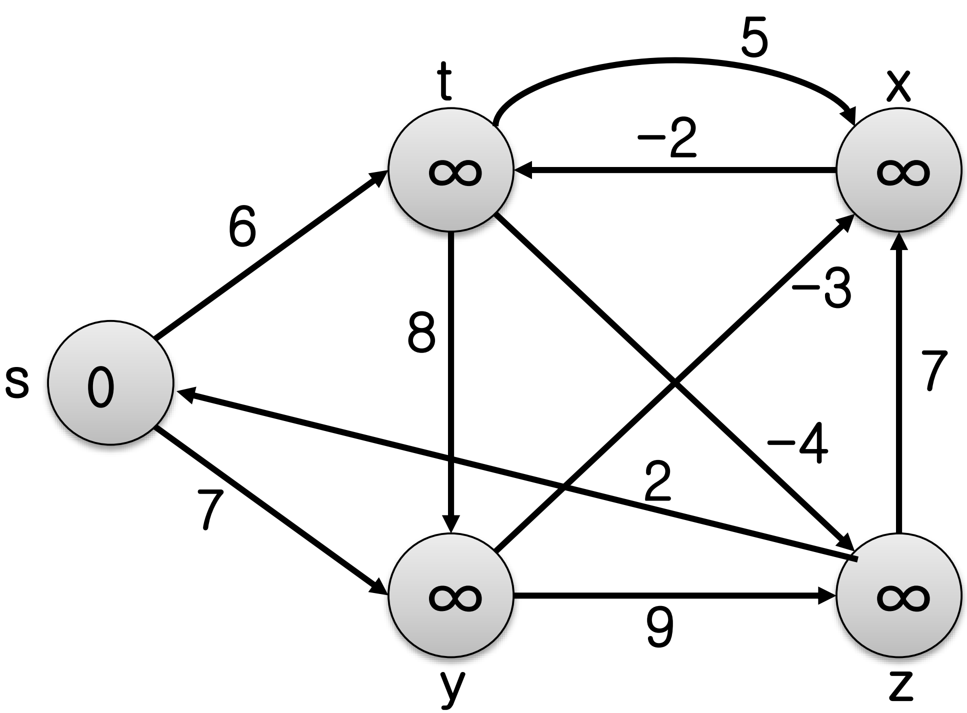
What will be the running time?

A: O(VE)

```
Relax(u,v,w): if (v.d > u.d + w(u,v))
                   v.d = u.d + w(u,v)
```

Example

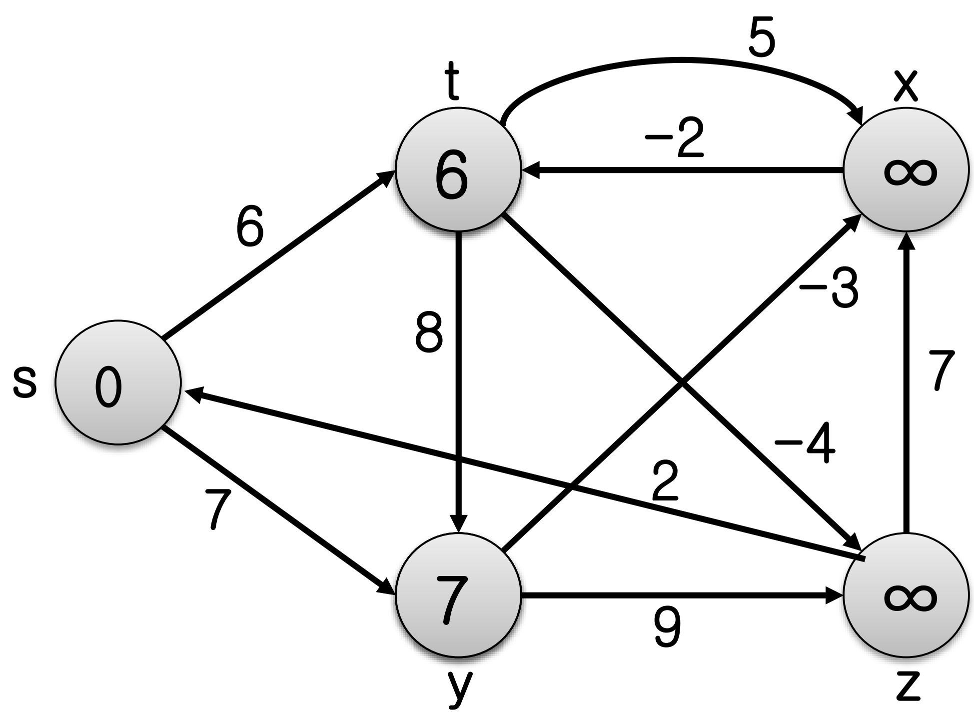
- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



	d_s	d_t	d_x	d_y	d_z
inital	0	∞	∞	∞	∞
After Pass 1					
After Pass 2					
After Pass 3					
After Pass 4					

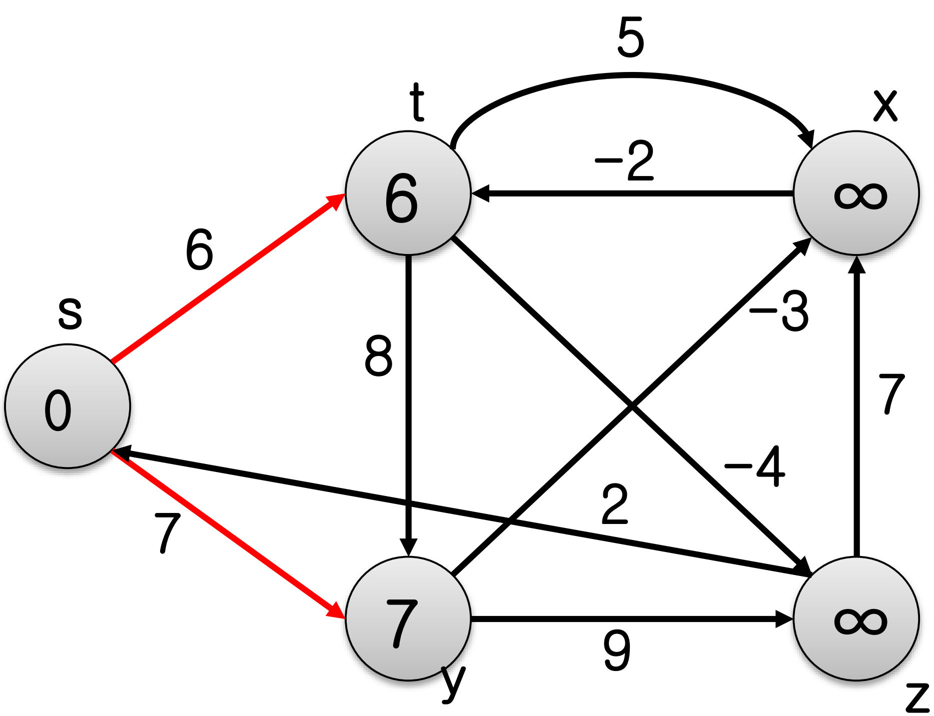
Pass 1

- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



Example

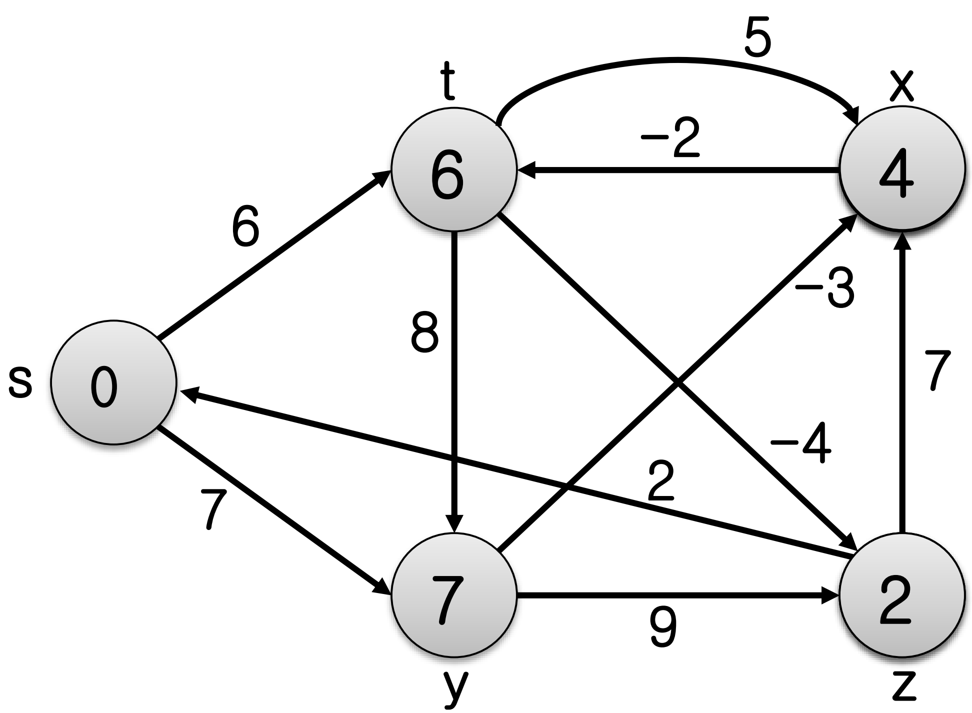
- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



	d_s	d_t	d_x	d_y	d_z
inital	0	∞	∞	∞	∞
After Pass 1	0	6,s	∞	7,s	∞
After Pass 2	0				
After Pass 3	0				
After Pass 4	0				

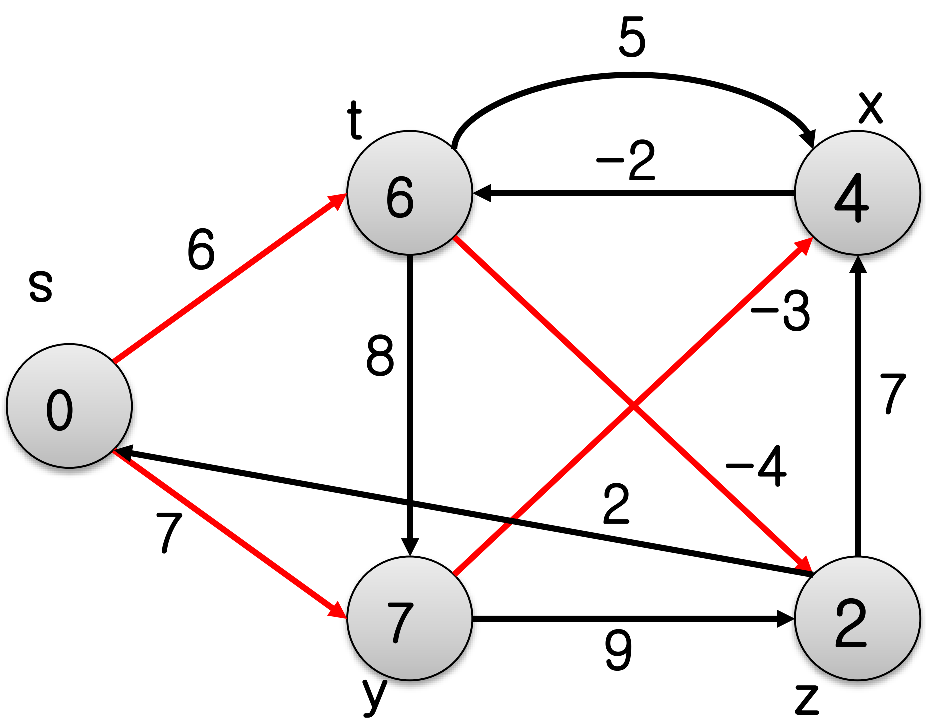
Pass 2

- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



Example

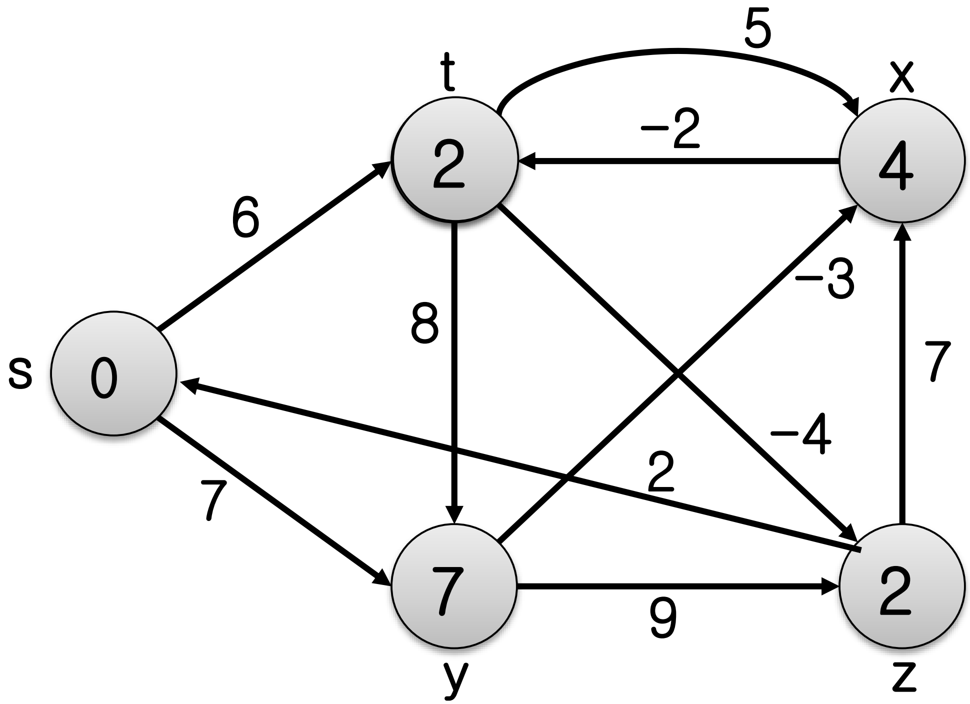
(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



	d_s	d_t	d_x	d_y	d_z
inital	0	∞	∞	∞	∞
After Pass 1	0	6,s	∞	7,s	∞
After Pass 2	0	6,s	4,y	7,s	2,t
After Pass 3					
After Pass 4					

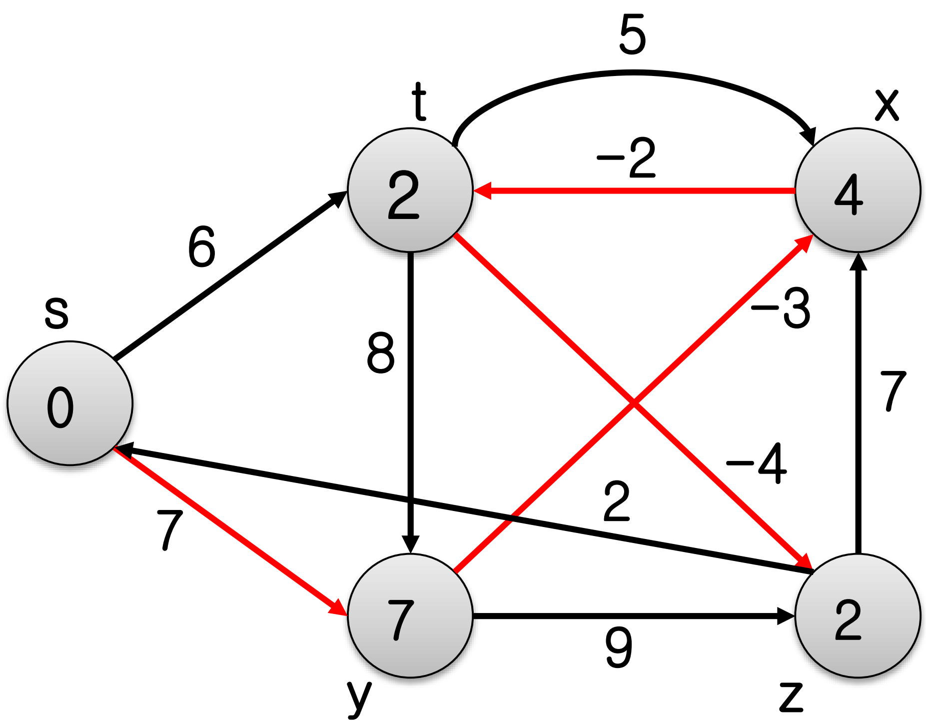
Pass 3

- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



Example

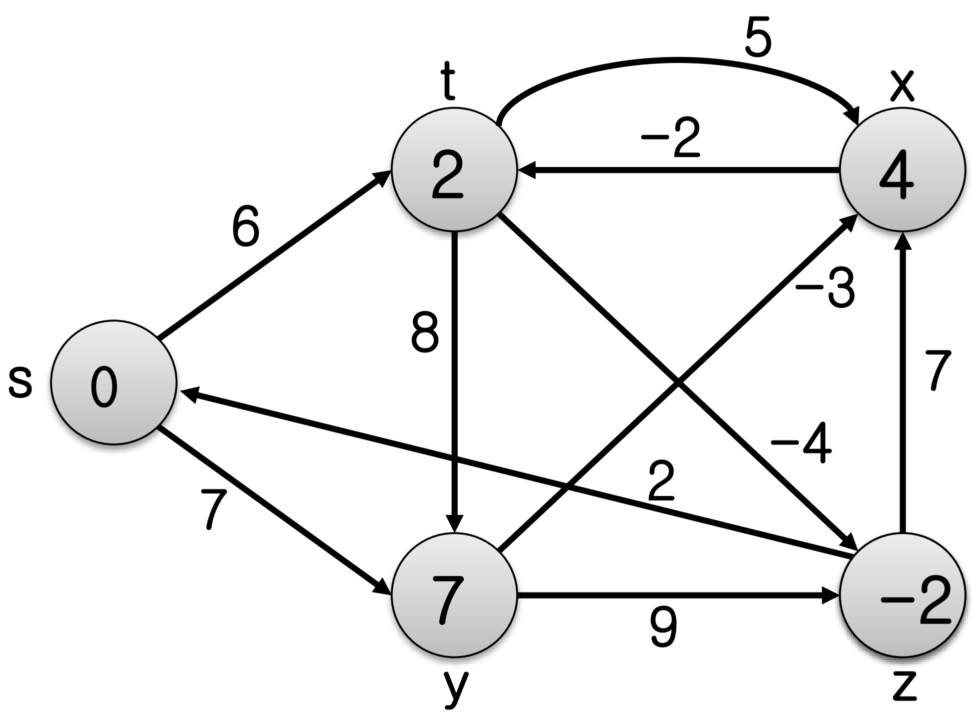
(t,x), (t,y), (t,z), (x,t), (y,x), (y,z),
(z,x), (z,s), (s,t), (s,y)



	d_s	d_t	d_x	d_y	d_z
inital	0	∞	∞	∞	∞
After Pass 1	0	6,s	∞	7,s	∞
After Pass 2	0	6,s	4,y	7,s	2,t
After Pass 3	0	2,x	4,y	7,s	2,t
After Pass 4	0				

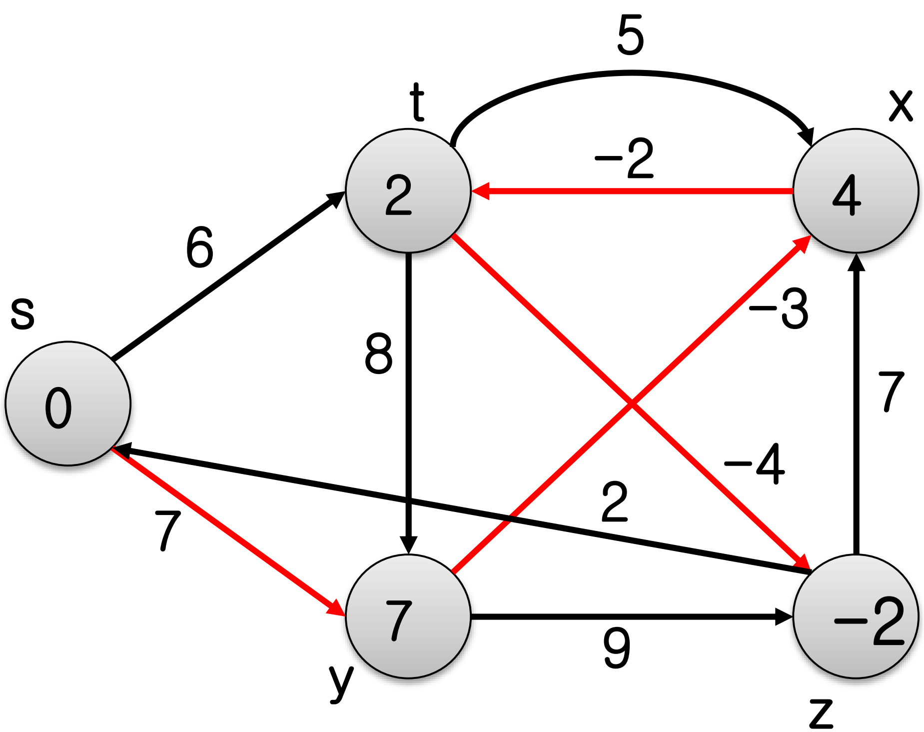
Pass 4

- (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)



Example

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z),
(z,x), (z,s), (s,t), (s,y)



	d_s	d_t	d_x	d_y	d_z
inital	0	∞	∞	∞	∞
After Pass 1	0	6,s	∞	7,s	∞
After Pass 2	0	6,s	4,y	7,s	2,t
After Pass 3	0	2,x	4,y	7,s	2,t
After Pass 4	0	2,x	4,y	7,s	-2,t

Running time

- Initialization: $\Theta(V)$
- Line 2-4 : $\Theta(E)$ * $|V|$-1 passes
- Line 5-7 : $O(E)$
- $O(VE)$

Correctness

Proof Use path-relaxation property.

Let v be reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$. Since p is acyclic, it has $\leq |V| - 1$ edges, so $k \leq |V| - 1$.

Each iteration of the for loop relaxes all edges:

- First iteration relaxes (v_0, v_1).
- Second iteration relaxes (v_1, v_2).
- kth iteration relaxes (v_{k-1}, v_k).

By the path-relaxation property, $v.\mathrm{d} = v_k.\mathrm{d} = \delta(s, v_k) = \delta(s, v)$. ■

Correctness

How about the TRUE/FALSE return value?

- Suppose there is no negative-weight cycle reachable from s.

  At termination, for all $(u, v) \in E$,

  $$\begin{aligned} v.\mathbf{d} &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &= u.\mathbf{d} + w(u, v) . \end{aligned}$$

  So BELLMAN-FORD returns TRUE.

Now suppose there exists negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, reachable from s.

Then $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

Then $v_i.\mathrm{d} \le v_{i-1}.\mathrm{d} + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

Sum around c:

$$\begin{aligned}
\sum_{i=1}^{k} v_i.\mathrm{d} &\le \sum_{i=1}^{k} (v_{i-1}.\mathrm{d} + w(v_{i-1}, v_i)) \\
&= \sum_{i=1}^{k} v_{i-1}.\mathrm{d} + \sum_{i=1}^{k} w(v_{i-1}, v_i)
\end{aligned}$$

The contradiction

- $\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- =>
  $\sum_{i=1}^{k} v_i.d - \sum_{i=1}^{k} v_{i-1}.d \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- => $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq \ v_k.d$ - $v_0.d$
- Since $v_0 = v_k$ (c is a cycle),
- $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq$0
- This contradicts c being a negative-weight cycle

Dijkstra's Algorithm

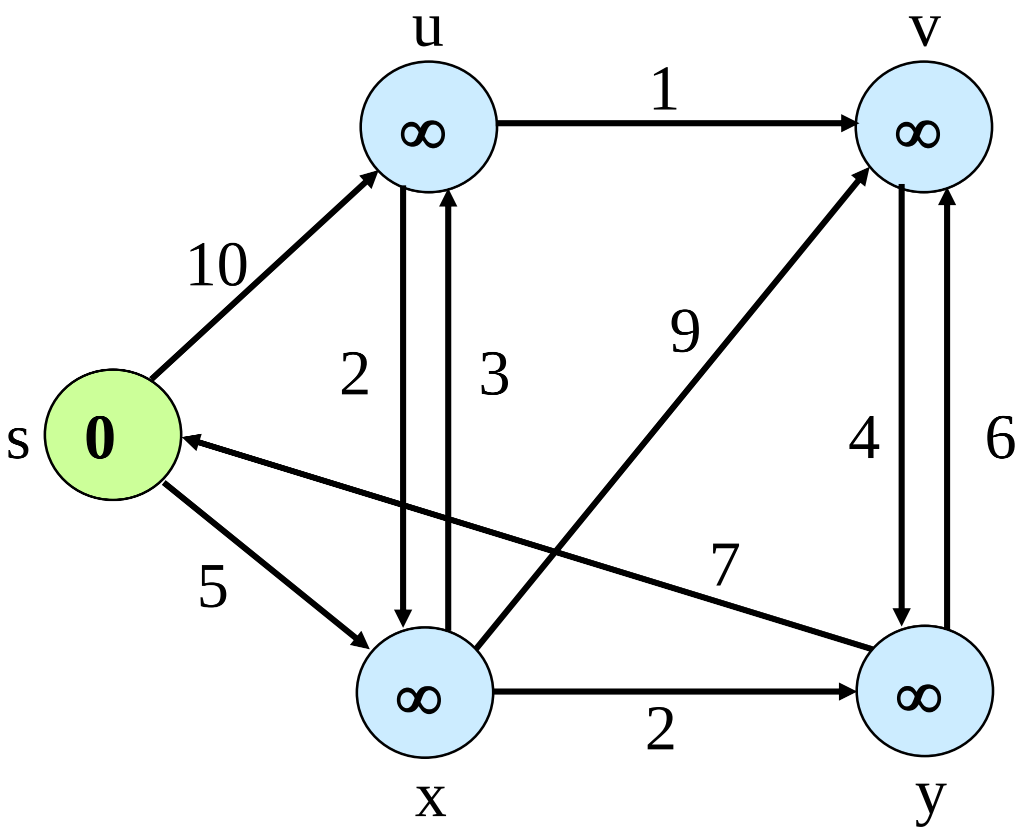
- If no negative edge weights, we can beat Bellman Ford
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - Use a priority queue keyed on v.d

Dijkstra's Algorithm

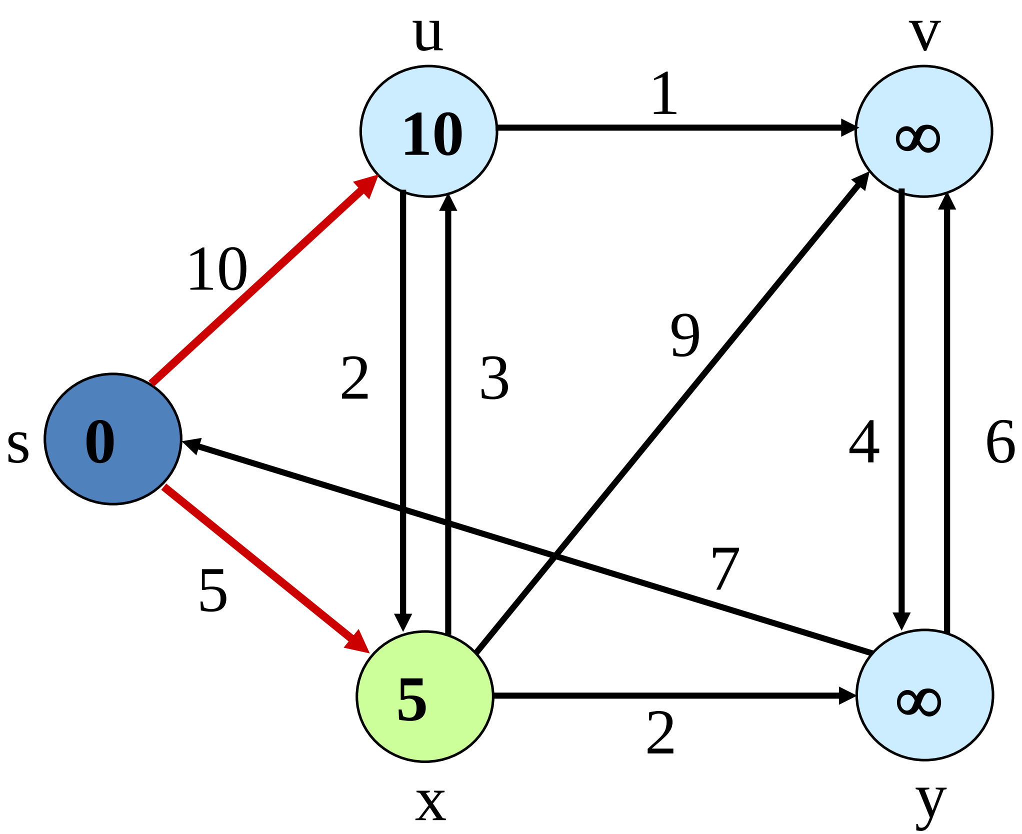
- Assumes **no negative-weight edges**.
- Maintains a vertex set S whose shortest path from s has been determined.
- Repeatedly selects u in V–S with minimum Shortest Path estimate (greedy choice).
- Store V–S in priority queue Q.

```
DIJKSTRA(G, w, s)
Initialize-SINGLE-SOURCE(G, s);
S = ∅;
Q = G.V;
while Q ≠ ∅
    u = Extract-Min(Q);
    S = S ∪ {u};
    for each v ∈ G.Adj[u]
        Relax(u, v, w)
```

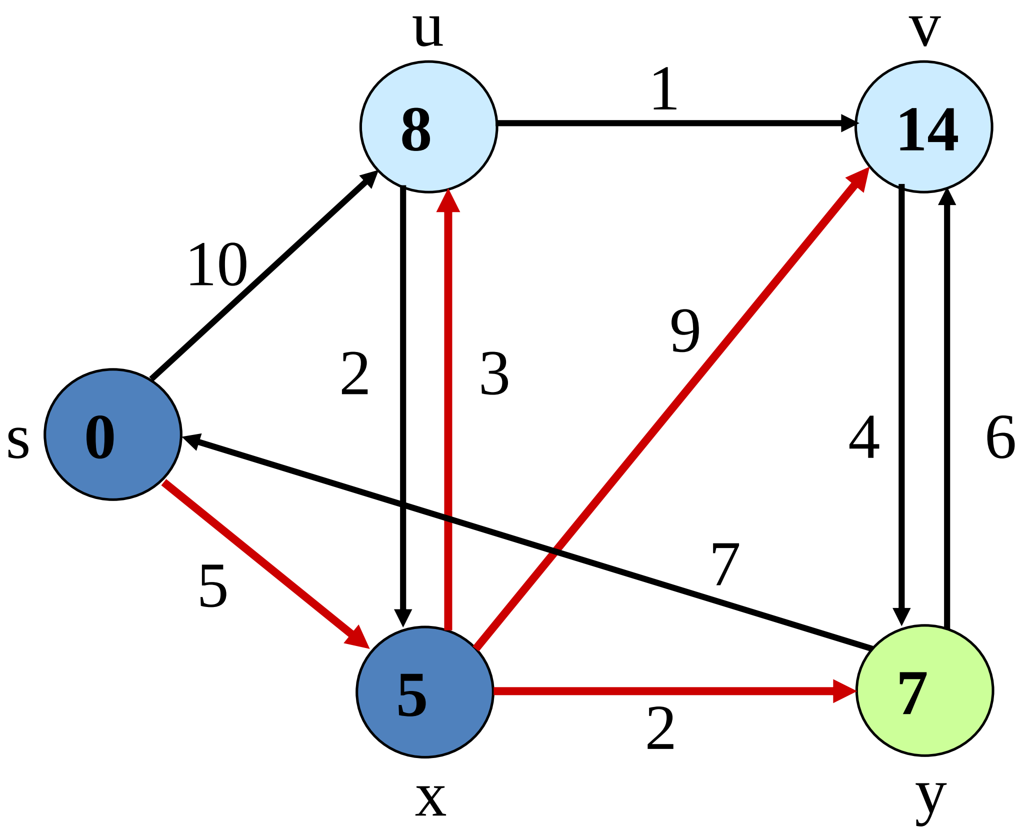
Example



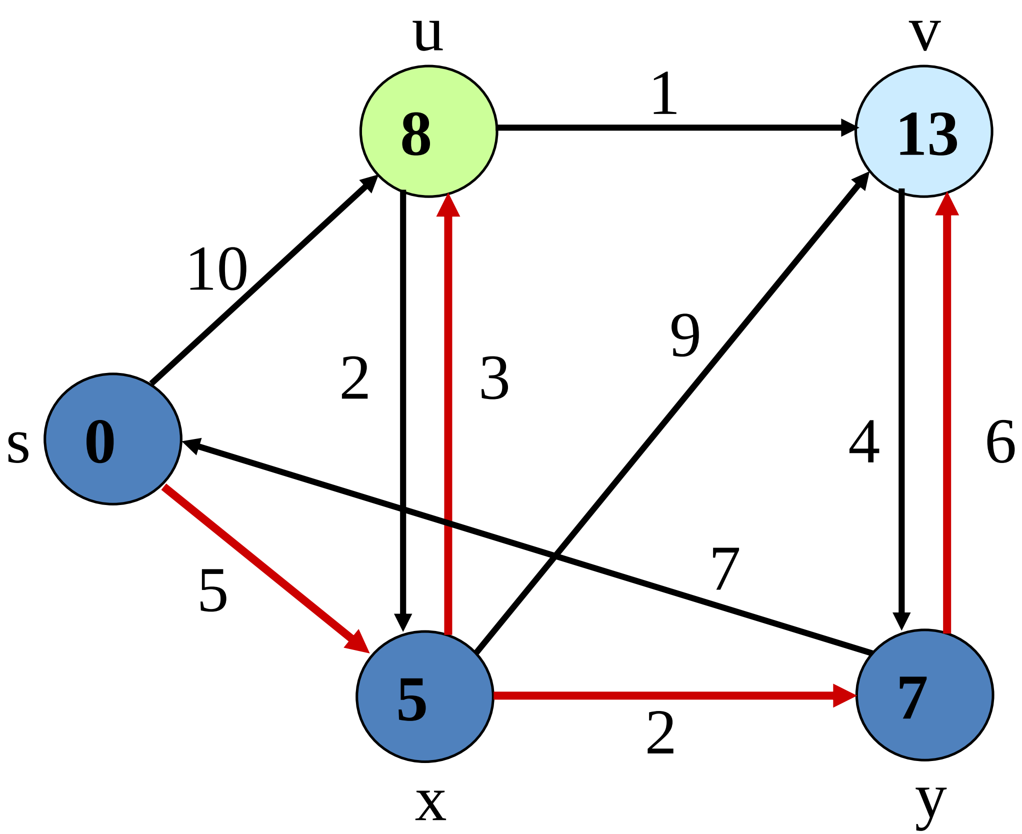
Example



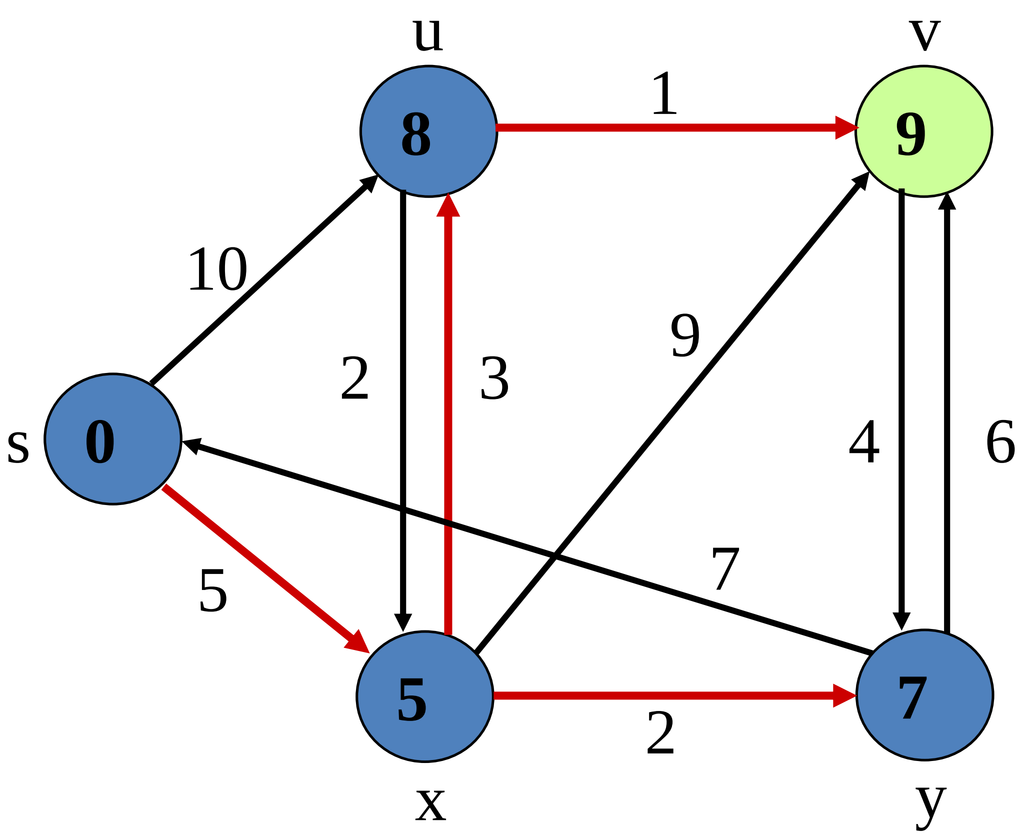
Example



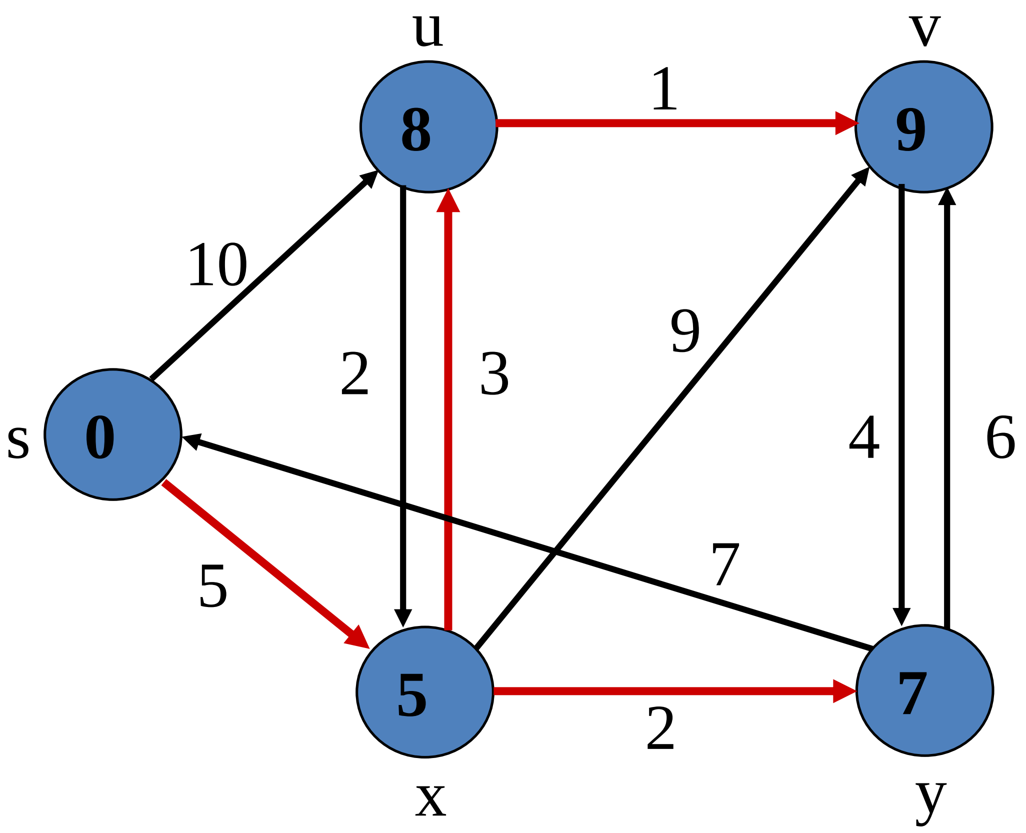
Example



Example



Example



Dijkstra's Algorithm

```
Dijkstra(G)
    for each v ∈ V
        v.d = ∞;
    s.d = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ u->G.Adj[]
            if (v.d > u.d+w(u,v))
                v.d = u.d + w(u,v);
```

Relaxation Step

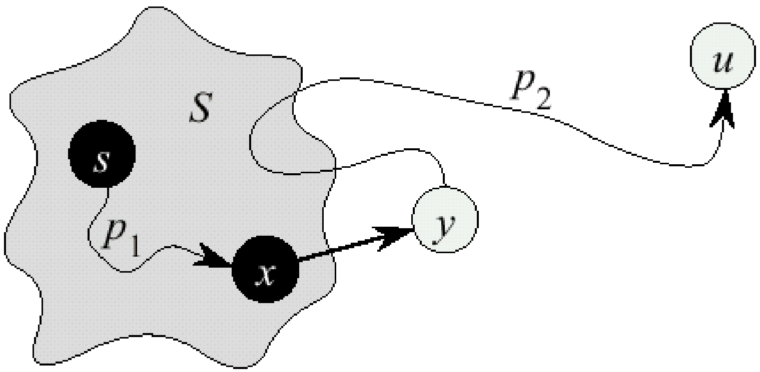
Note: this is really a call to Q->DecreaseKey()

Dijkstra's correctness

- We will prove that whenever u is added to S, $u.d = \delta(s,u)$, i.e., that d is minimum, and that equality is maintained thereafter
- Proof
  - Note that $\forall v$, $v.d \geq \delta(s,v)$
  - let u be the first vertex for which $u.d \neq \delta(s, u)$ (i.e., u.d $> \delta(s,u)$) when it is added to set S.
  - We will show that the assumption of such a vertex leads to a contradiction

Correctness (Cont'd)

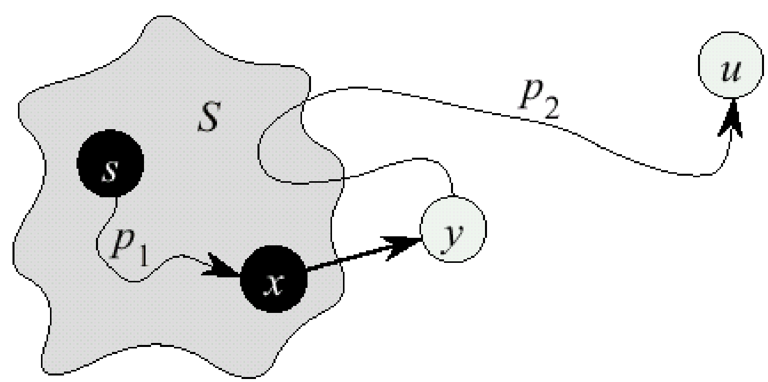
- A shortest path p from source s to vertex u can be decomposed into :
  - $p_1\ s \rightarrow x$,
  - $x \rightarrow y$
  - $p_2 :\ y \rightarrow u$



- where *y* is the first vertex on the path that is not in *S* and $x \in S$ immediately precedes *y*

Correctness (Cont'd)

- Then, it must be that $y.d = \delta(s,y)$ because
  - $X.d$ is set correctly for y's predecessor $x \in S$ on the shortest path (by choice of u as the first vertex for which d is set incorrectly)
  - when the algorithm inserted x into S, it relaxed the edge (x,y), assigning $y.d$ the correct value



Correctness (Cont'd)

- Thus, $y.d = \delta(s,y)$

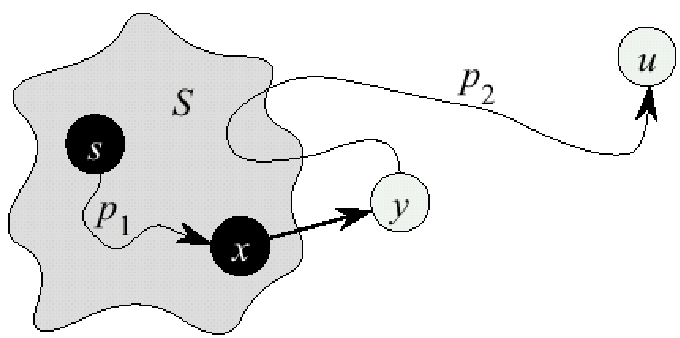
  $\leq \delta(s,u)$ (y appears before u on the shortestpath)

  $\leq$ u.d (upper-bound property)

But because both u and y are in V-S when u was chosen, we have u.d $\leq$ y.d, and therefore the two inequalities are in fact equalities,

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

Consequently, u.d = $\delta(s,u)$, which contradicts our choice of u



Dijkstra's running time

```
Dijkstra(G)
    for each v ∈ V
        v.d= ∞;
    s.d = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ u->Adj[]
            if (v.d > u.d+w(u,v))
                DecreaseKey(v.d,u.d+w(u,v));
```

How many times is ExtractMin() called?

A: |V|

How many times is DecreaseKey() called?

A: |E|

What will be the total running time?

Dijkstra's Running Time

- Extract-Min executed |V| time
- Decrease-Key executed |E| time
- Time = $|V|\ T_{\text{Extract-Min}} + |E|\ T_{\text{Decrease-Key}}$
- Time = O(VlgV) + O (ElgV) = O(ElgV)

Summary

- **We learned**
  - Shortest-Path Problems
  - Properties of Shortest Paths, Relaxation
  - Bellman-Ford Algorithm
  - Dijkstra’s Algorithm
- Common mistakes: Do not forget to relax all edges in all algorithms.