CS146: Data Structures and Algorithms
Lecture 18

SINGLE-SOURCE SHORTEST PATH (CH 24)

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Single-Source Shortest Path (Ch 24)

- *Input*: given a weighted directed graph G=(V,E), n=|V| and m=|E|.
- *Output*: find the path from a given source vertex s to another vertex v
- *Goal*: minimum-weight path
  - "Shortest-path" = minimum weight
  - Weight of path is sum of weighted edges
  - E.g., a road map: what is the shortest path from San Jose to Palo Alto?

Formal Definition shortest path

- *Input*: G=(V,E), w: E-> R
- *Output*: weight w(p) of path p=<v_0, v_1,…,v_k>

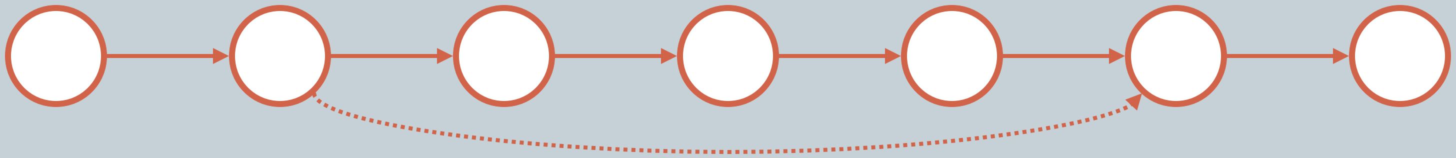
  i.e. , w(p)= $\sum_{i=1}^{k} w(v_{i-1}, v_i)$
- *Goal*: minimum w(p)

- Shortest-path weight δ(u,v) from u to v

- $\delta(u,v) = \begin{cases} \min\{w(p)\} & there\ is\ a\ path\ from\ u\ to\ v \\ \infty, & otherwise \end{cases}$

Shortest Path Properties

- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



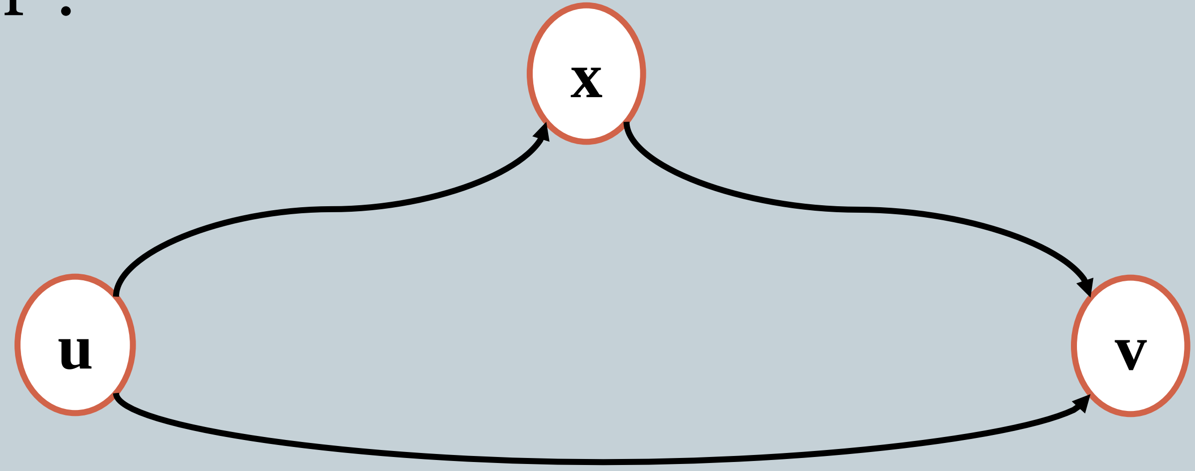
  - Proof: suppose some subpath is not a shortest path
    - There must then exist a shorter subpath
    - Could substitute the shorter subpath for a shorter path
    - But then overall path is not shortest path. Contradiction

Shortest Path Properties

- Define δ(u,v) to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*:

$$\delta(u,v) \le \delta(u,x) + \delta(x,v)$$

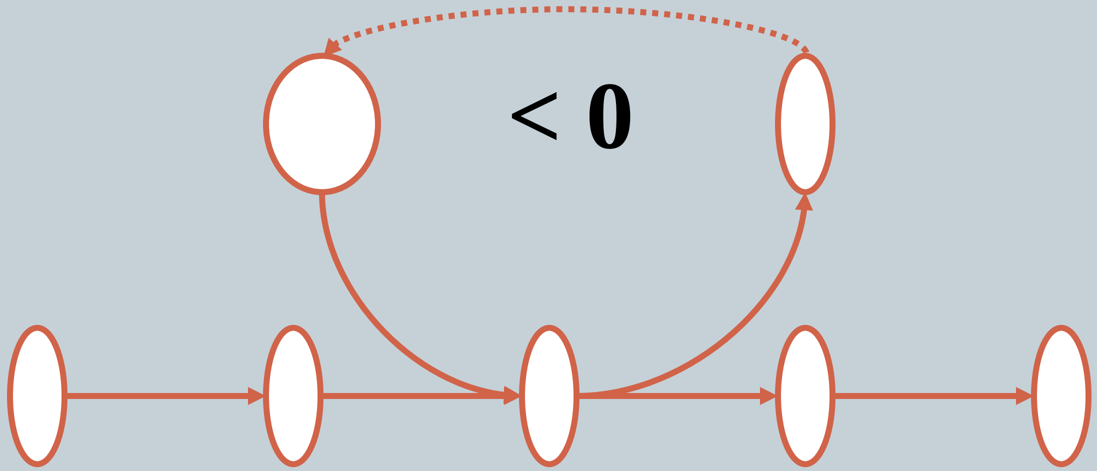
- "Proof":



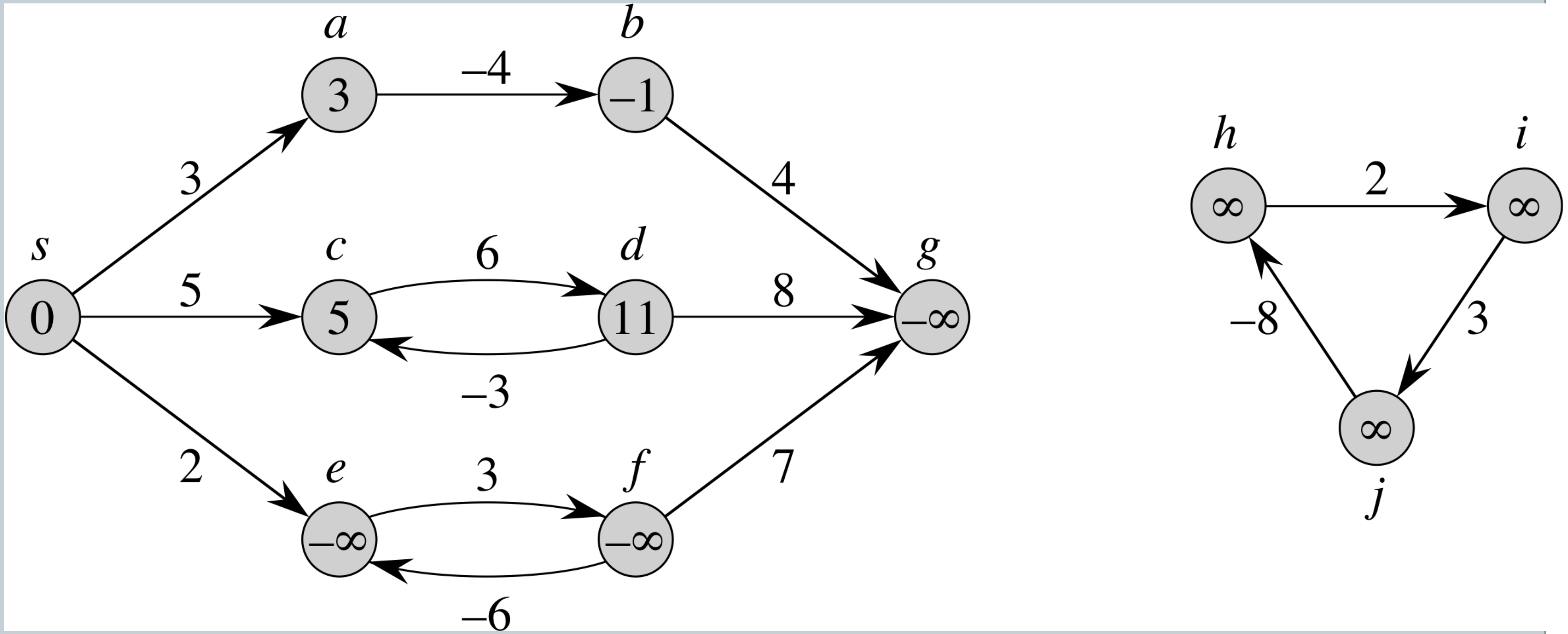
This path is no longer than any other path

Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist (*Why*?):



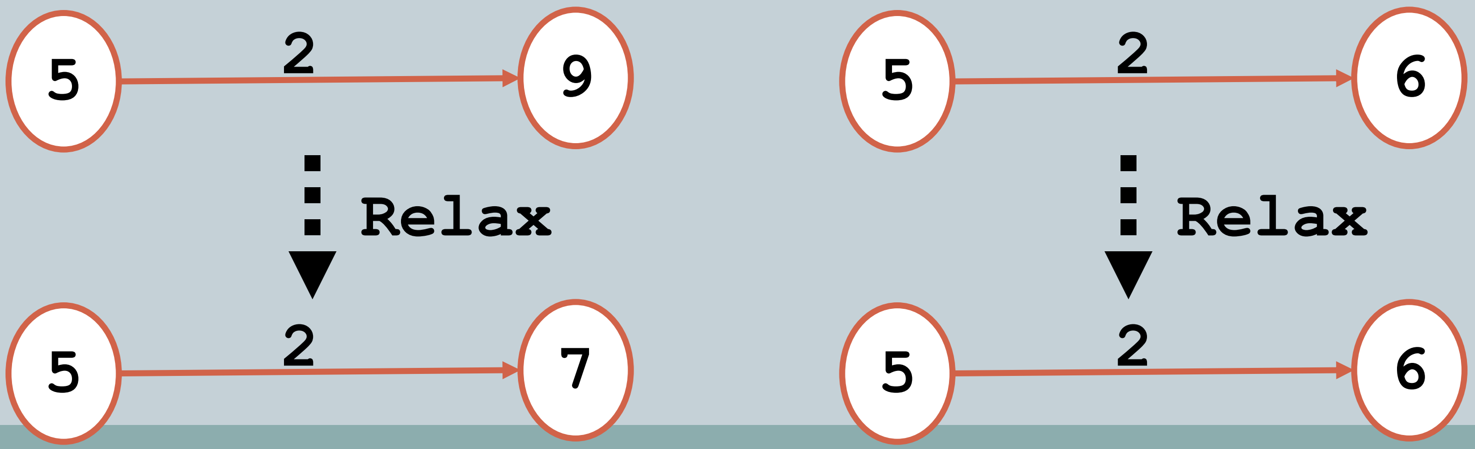
Example Negative cycles



Relaxation

- A key technique in shortest path algorithms is *relaxation*
  - Idea: for all *v*, maintain upper bound v.d on δ(s,v)

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



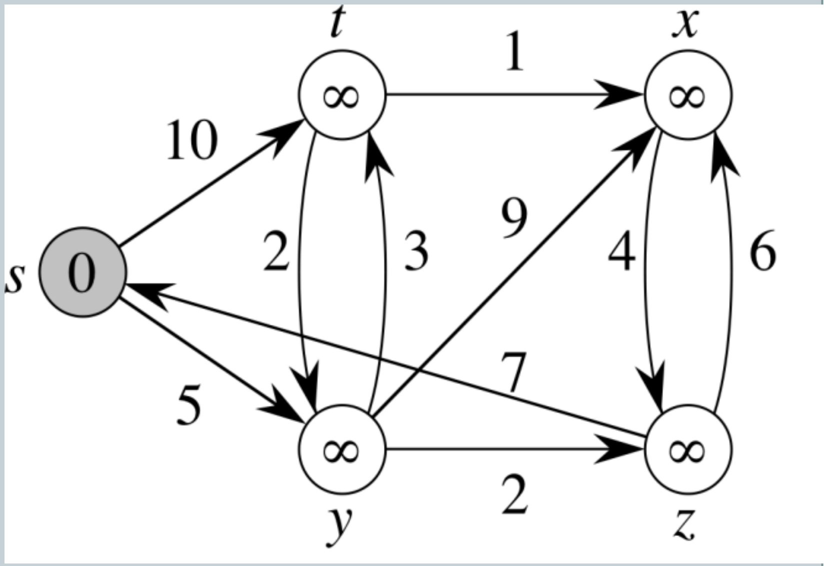
Dijkstra's Algorithm

- If no negative edge weights (special case)
- 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

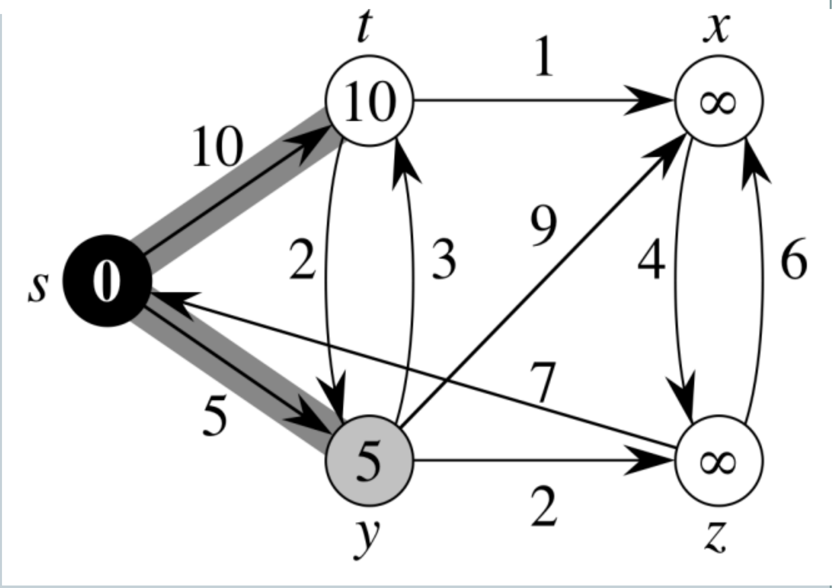
```
Dijkstra(G)
    for each v ∈ V
        v.d = ∞;
    s.d = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

$\}$ Relaxation Step



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 ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

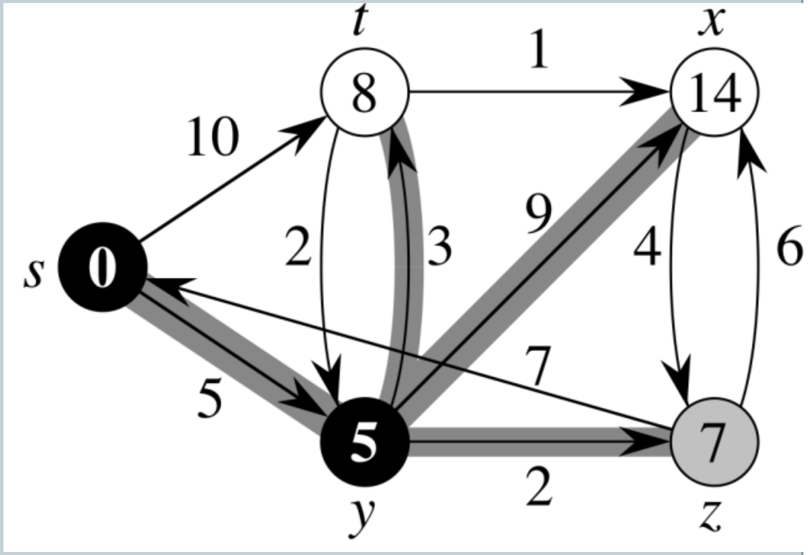


} Relaxation Step

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 ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

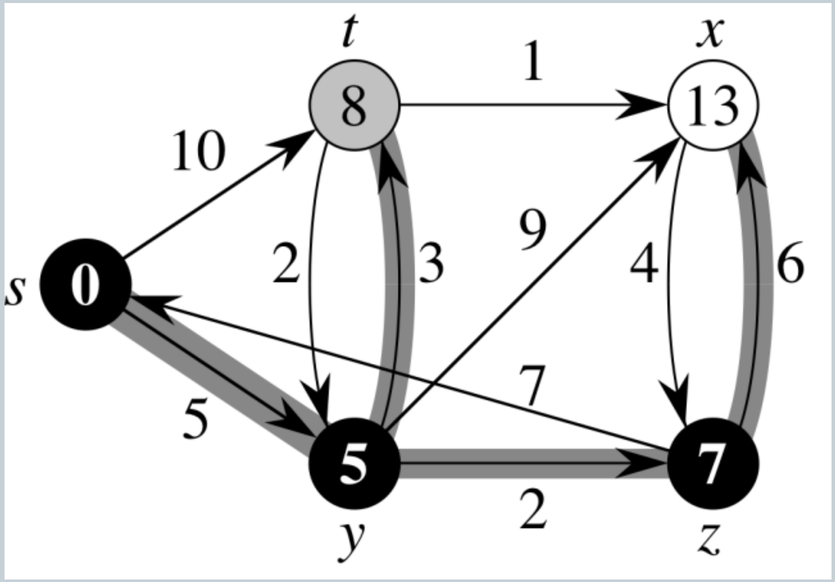
} Relaxation Step



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};
        for each v ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

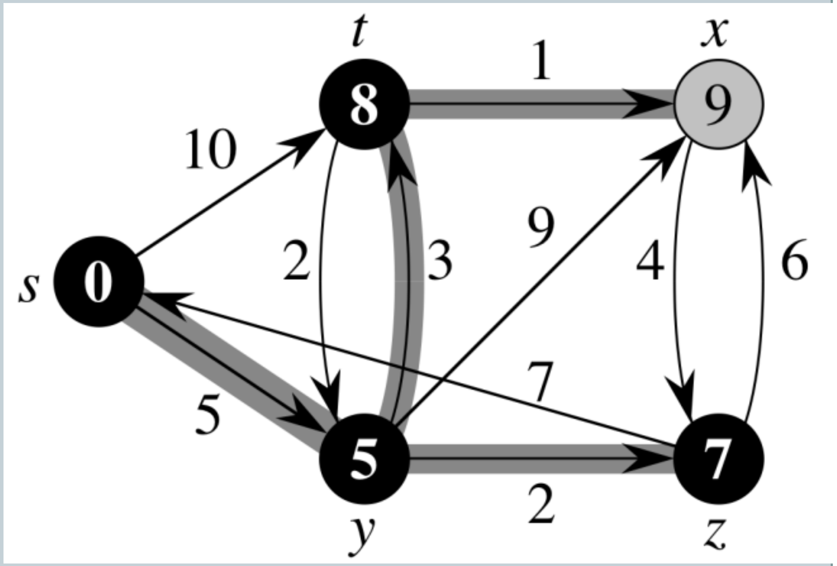
} Relaxation Step



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 ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

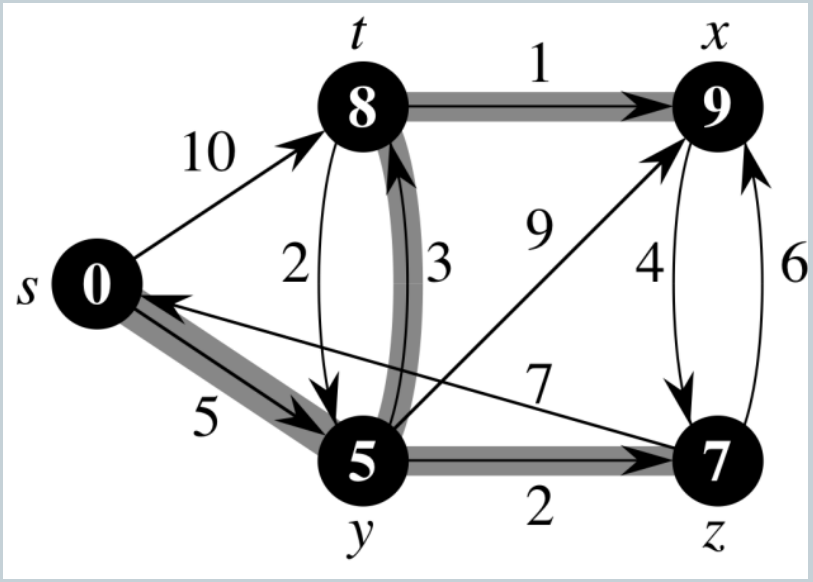
} Relaxation Step



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 ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

$\}$ Relaxation Step



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 ∈ Adj[u]
            if (v.d > u.d+w(u,v))
                v.d = u.d+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecreaseKey() called?

total running time?

A: O(m lg n) using binary heap for Q

Can achive O(n lg n + m) with Fibonacci heaps

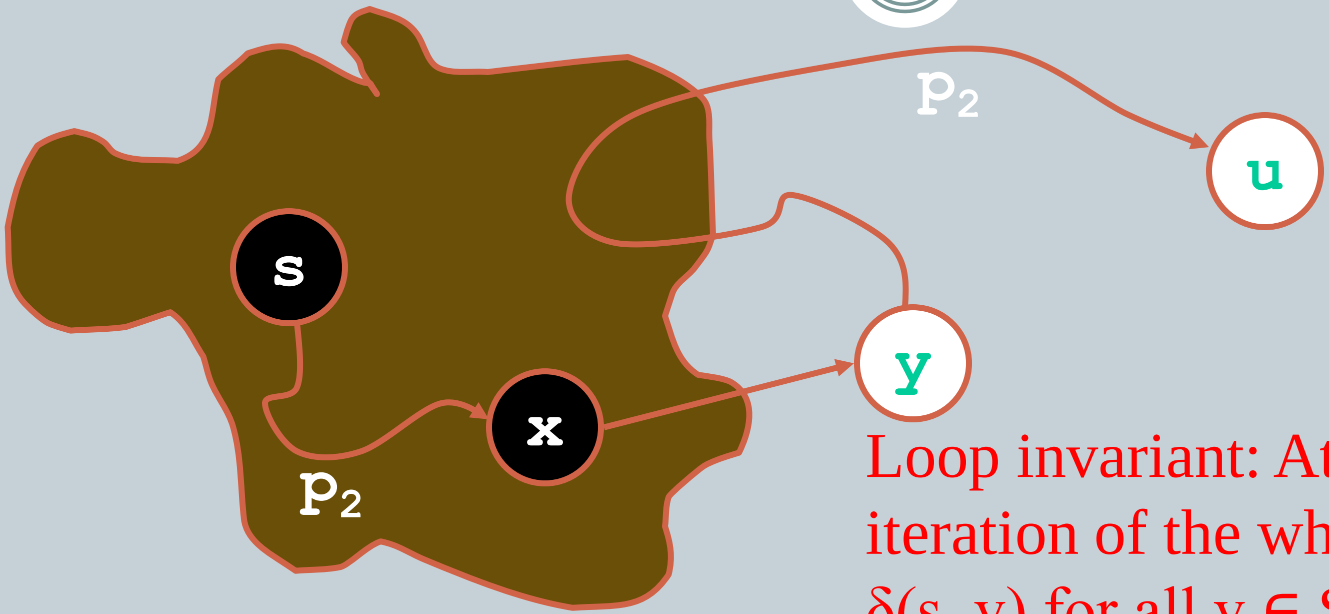
Dijkstra’s Algorithm

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

Loop invariant: At the start of each iteration of the while loop, v.d = δ(s, v) for all v ∈ S.

Correctness: we must show that when u is removed from Q, it has already converged

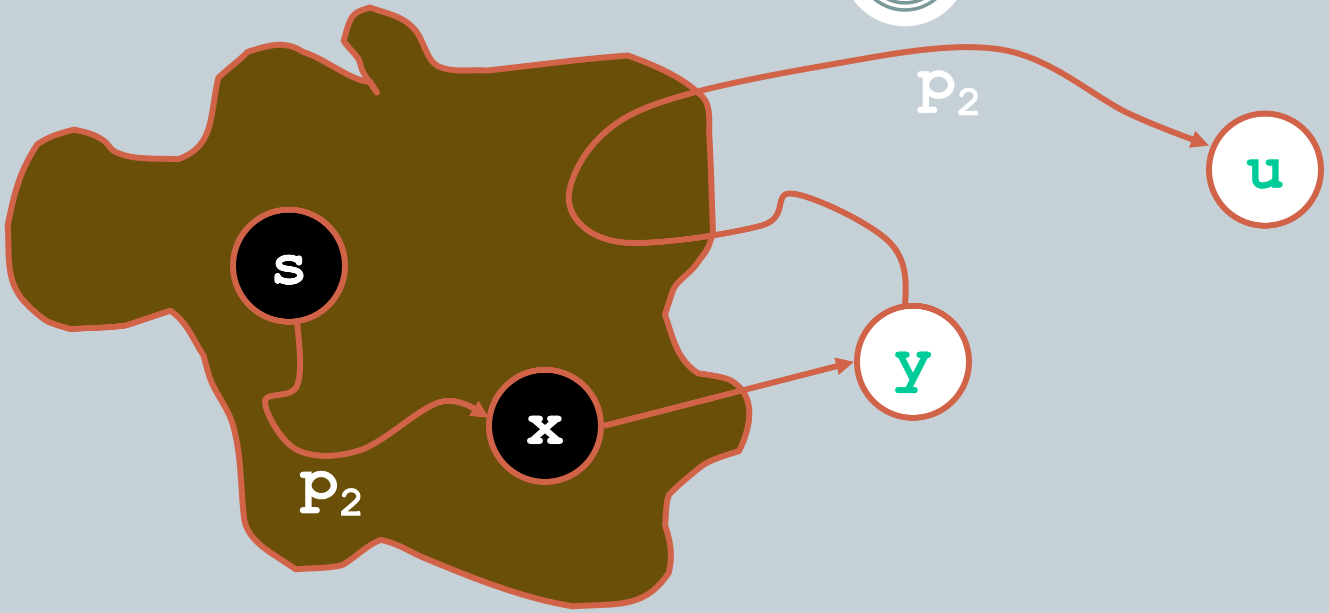
Correctness Of Dijkstra's Algorithm



Loop invariant: At the start of each iteration of the while loop, v.d = δ(s, v) for all v ∈ S.

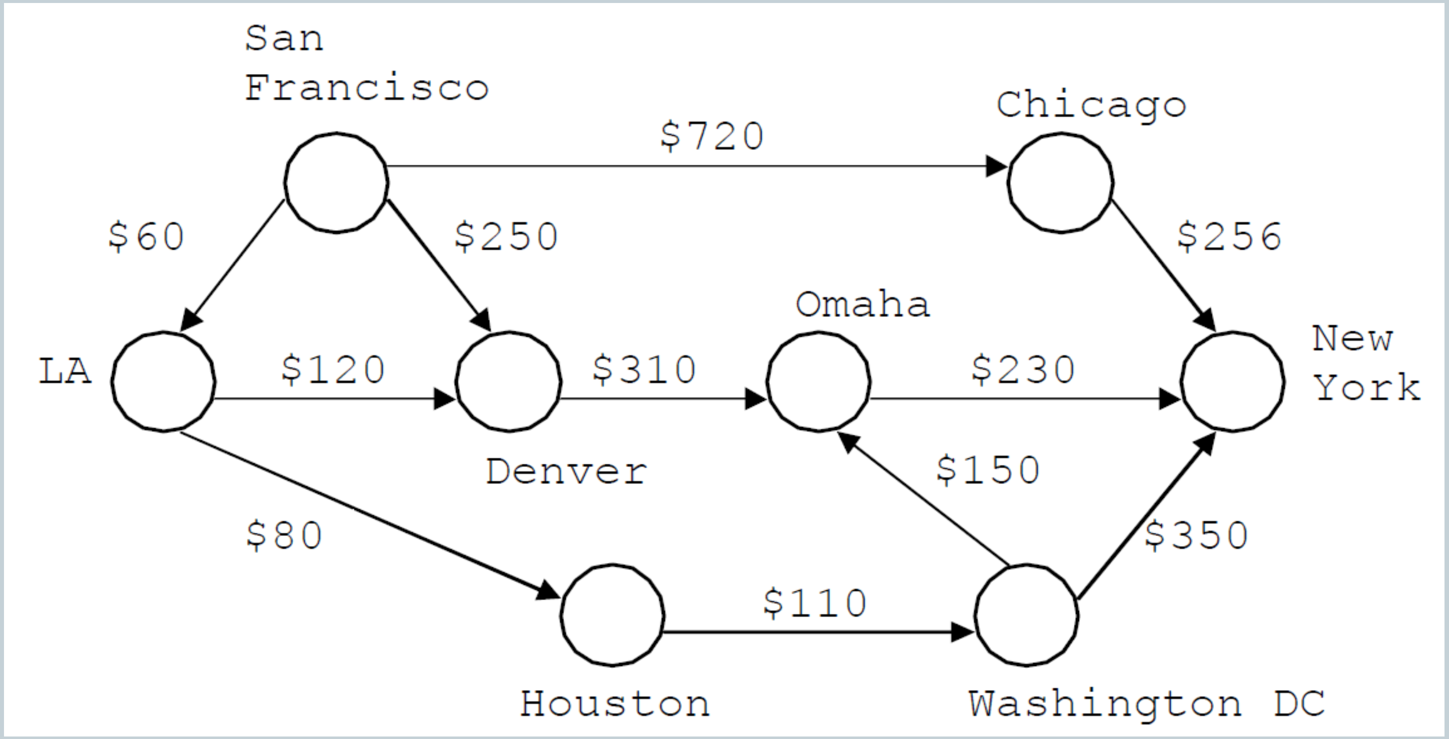
- Note that $v.d \geq \delta(s,v)\ \forall v$
- Let u be first vertex picked s.t. ∃ shorter path than u.d $\Rightarrow u.d > \delta(s,u)$
- Let y be first vertex $\in V-S$ on actual shortest path from $s \rightarrow u$ $\Rightarrow y.d = \delta(s,y)$
  - Because x.d is set correctly for y's predecessor $x \in S$ on the shortest path, and
  - When we put x into S, we relaxed (x,y), giving y.d the correct value

Correctness Of Dijkstra's Algorithm



- Note that $v.d \geq \delta(s,v)\ \forall v$
- Let u be first vertex picked s.t. $\exists$ shorter path than u.d $\Rightarrow u.d > \delta(s,u)$
- Let y be first vertex $\in V-S$ on actual shortest path from $s \rightarrow u$ $\Rightarrow y.d = \delta(s,y)$
- u.d $> \delta(s,u)$
  $= \delta(s,y) + \delta(y,u)$ (*Why?*)
  $= y.d + \delta(y,u)$
  $\geq y.d$ But if u.d > y.d, wouldn't have chosen u. Contradiction.

Find the cheapest route from San Francisco to New York



Single Source Shortest Paths

- Dijkstra is fast but all weights must be positive
- The general case (positive and negative weights) can be solved by Bellman-Ford algorithm.
  - Finds single source shortest paths if no negative cycle
  - In the presence of a negative cycle reports the negative cycle and no shortest paths are computed

Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        v.d = ∞;
    s.d = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (v.d > u.d + w(u,v))
            return "no solution";
```

} Initialize d, which will converge to shortest-path value δ

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

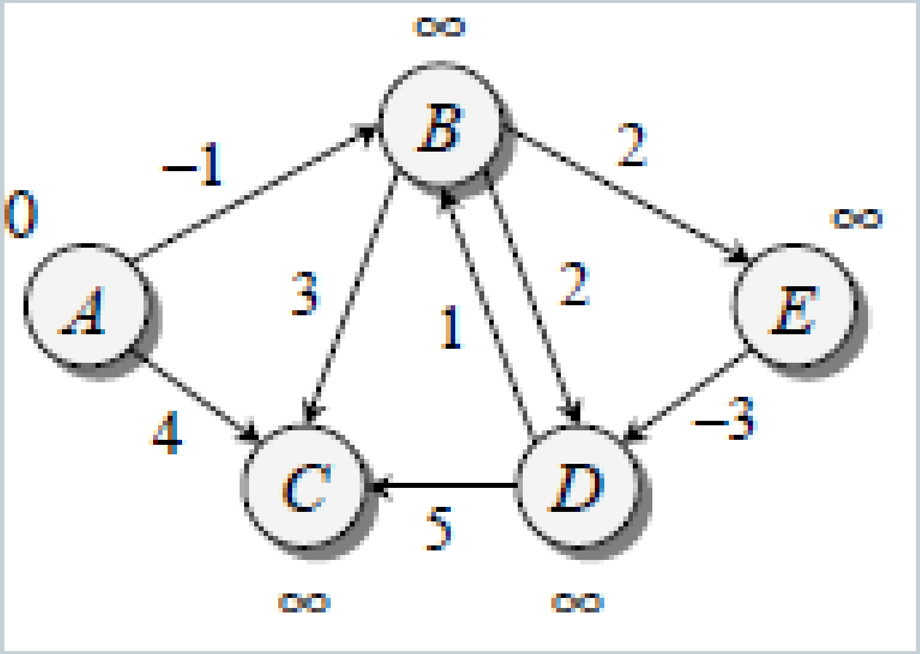
} Test for solution
Under what condition do we get a solution?

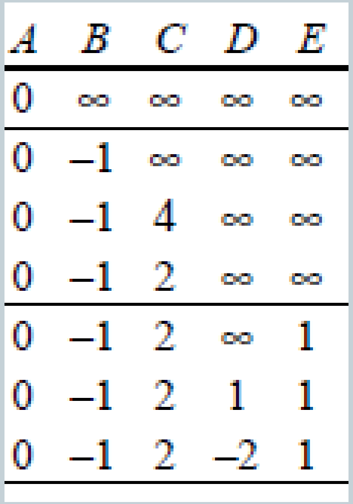
```
Relax(u,v,w): if (v.d > u.d+w) then v.d=u.d+w
```

Time Complexity: O(nm)

example

- Order of edges: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C) (B,C), (E,D)





A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Bellman-Ford

Note that order in which edges are processed affects how quickly it converges

Correctness: show v.d = δ(s,v) after |V|-1 passes

Lemma: v.d ≥ δ(s,v) always

Initially true

Let v be first vertex for which v.d < δ(s,v)

Let u be the vertex that caused v.d to change: *v.d = u.d + w(u,v)*

Then v.d < δ(s,v)

δ(s,v) ≤ δ(s,u) + w(u,v)(*triangle inequality*)

δ(s,u) + w(u,v) ≤ u.d + w(u,v) (*Why?*)

So v.d < u.d + w(u,v). Contradiction.

DAG Shortest Paths

- Problem: finding shortest paths in DAG
  - Bellman-Ford takes O(VE) time.
  - *How can we do better?*
  - Idea: use topological sort
    - If were lucky and processes vertices on each shortest path from left to right, would be done in one pass
    - Every path in a dag is subsequence of topologically sorted vertex order, so processing vertices in that order, we will do each path in forward order (will never relax edges out of vertices before doing all edges into vertices).
    - Thus: just one pass. *What will be the running time?*

Example source is s