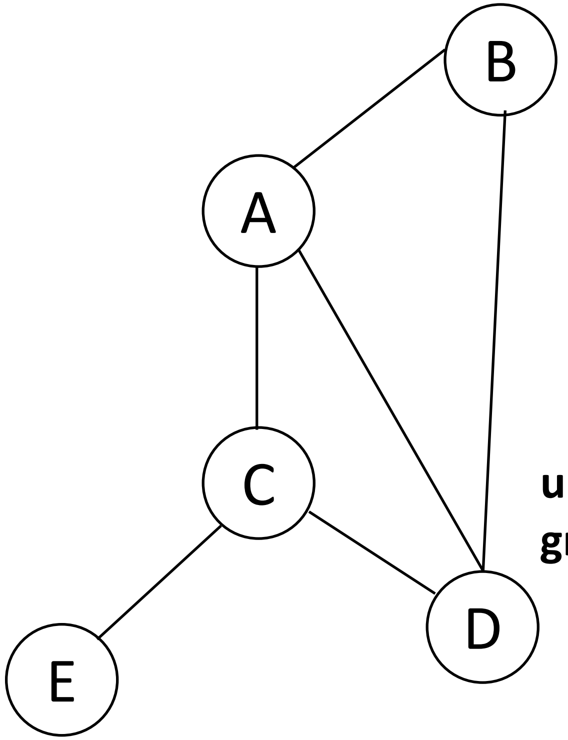
Shortest Path Problem

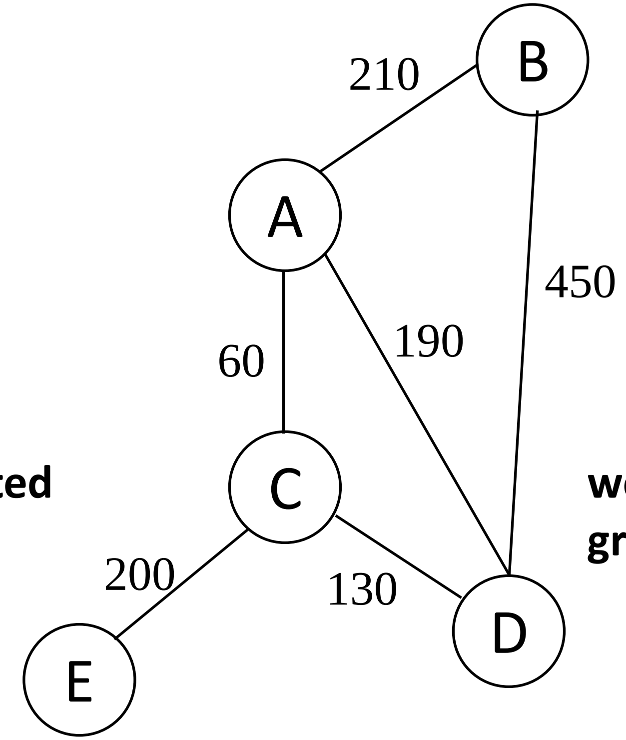
Dikjstra’s Algorithm

Shortest Path Problems

- **What is shortest path ?**
  - shortest length between two vertices for an unweighted graph:
  - smallest cost between two vertices *for* a weighted graph:



unweighted graph



weighted graph

Shortest Path Problems

- How can we find the shortest route between two points on a map?
- Model the problem as a graph problem:
  - Road map is a weighted graph:

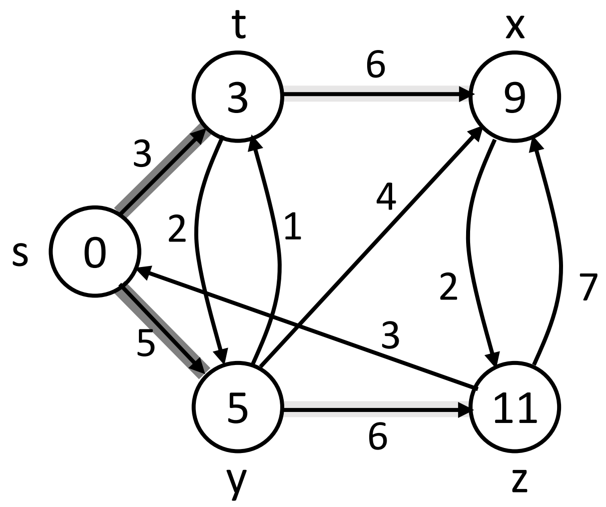
    vertices = cities

    edges = road segments between cities

    edge weights = road distances
  - Goal: find a shortest path between two vertices (cities)

Shortest Path Problems

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**



- **Weight of path** p = $\langle v_0, v_1, \ldots, v_k \rangle$

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

- **Shortest-path weight** from u to v:

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

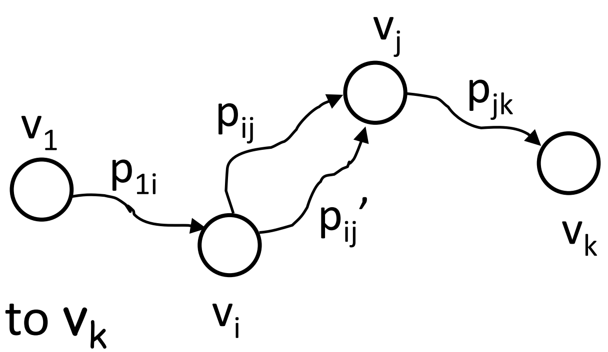
- Shortest path u to v is any path p such that w(p) = δ(u, v)

Variants of Shortest Paths

- **Single-source shortest path**
  - G = (V, E) $\Rightarrow$ find a shortest path from a given source vertex **s** to each vertex v $\in$ V
- **Single-destination shortest path**
  - Find a shortest path to a given destination vertex **t** from each vertex v
  - Reverse the direction of each edge $\Rightarrow$ single-source
- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
  - Solve the single-source problem
- **All-pairs shortest-paths**
  - Find a shortest path from u to v for every pair of vertices u and v

Optimal Substructure of Shortest Paths

Given:



- A weighted, directed graph $G = (V, E)$
- A weight function $w: E \to \mathbf{R}$,
- A shortest path $p_{1k} = \langle v_1, v_2, \ldots, v_k \rangle$ from v_1 to v_k
- A subpath of p: $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, with $1 \le i \le j \le k$

Then: p_{ij} is a shortest path from v_i to v_j

Proof: $p = v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$

Assume $\exists\, p_{ij}'$ from v_i to v_j with $w(p_{ij}') < w(p_{ij})$

$\Rightarrow w(p') = w(p_{1i}) + w(p_{ij}') + w(p_{jk}) < w(p)$ contradiction!

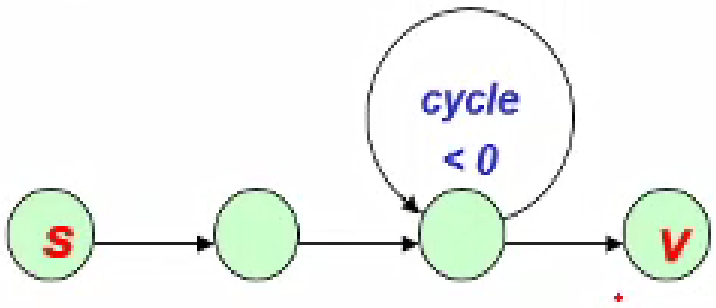
Representation

Definition:

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

Well Definedness:

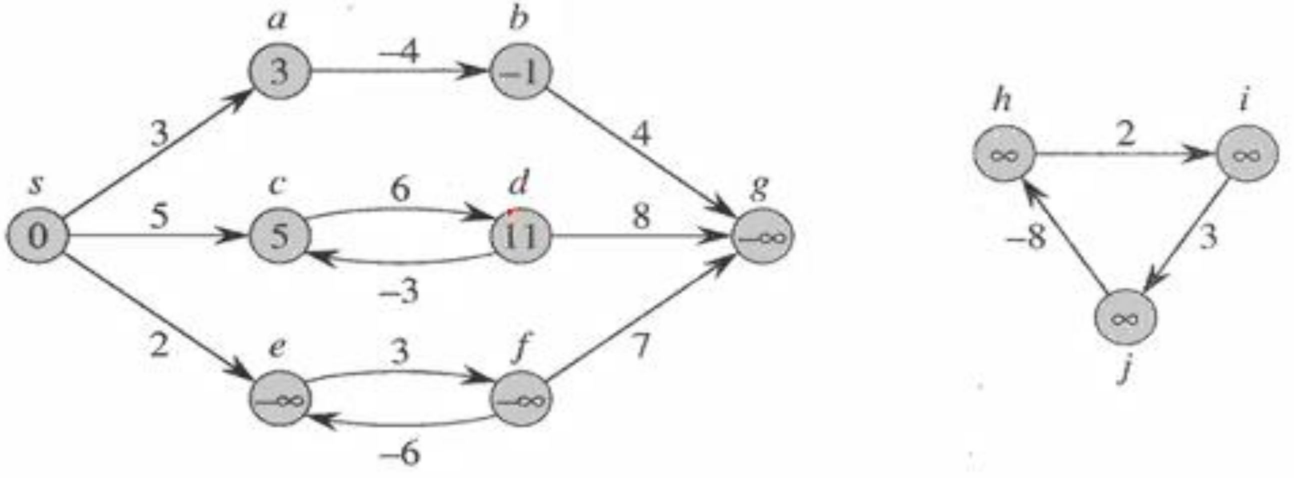
- *negative-weight cycle in graph*: Some shortest paths may not be defined
- *argument:* can always get a shorter path by going around the cycle again



Negative-edge weights

- 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.



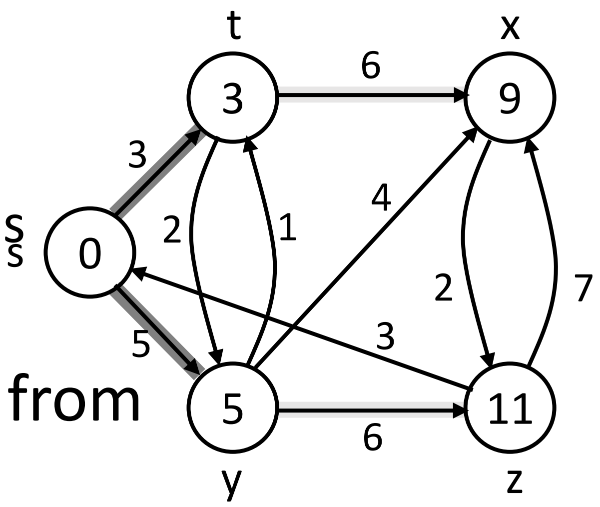
Cycles

- Can shortest paths contain cycles?
- Negative-weight cycles
  - Shortest path is not well defined
- Positive-weight cycles:
  - By removing the cycle, we can get a shorter path
- Zero-weight cycles
  - No reason to use them
  - Can remove them to obtain a path with same weigh

Shortest-Path Representation

For each vertex $v \in V$:

- $d[v] = \delta(s, v)$: a **shortest-path estimate**
  - Initially, $d[v]=\infty$
  - Reduces (reaches close to $\delta(s, v)$) as algorithms progress
- $\pi[v]$ = **predecessor** of v on a shortest path from ***s***
  - If no predecessor, $\pi[v]$ = NIL
  - π induces a tree—**shortest-path tree**
- Shortest paths & shortest path trees are not unique



Initialization

Alg.: INIT (G, s)

1. **for** each v $\in$ V

2. **do** d[v] ← ∞

3. π[v] ← NIL

4. d[s] ← 0

- All the shortest-paths algorithms start with INIT
- Maintain d[v] for each v $\in$ V
- d[v] is called the shortest-path weight estimate and it is the upper bound on δ(s, v)

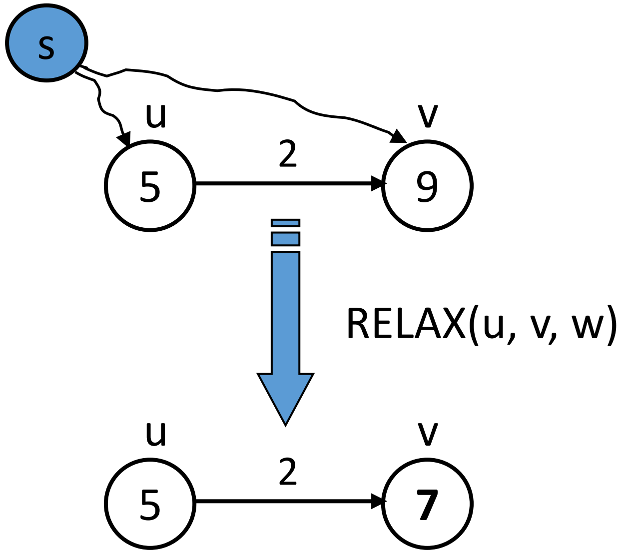
Relaxation

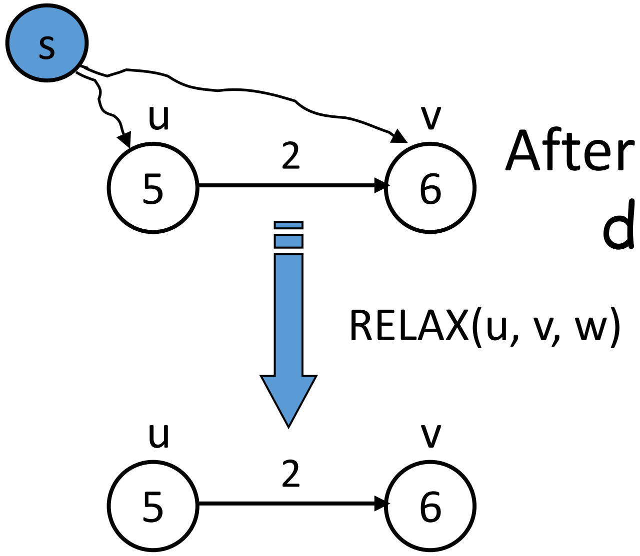
- **Relaxing** an edge (u, v) = testing whether we can improve the shortest path to v found so far by going through u

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

  we can improve the shortest path to v

  $\Rightarrow$ update $d[v]$ and $\pi[v]$





After relaxation:

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

RELAX(u, v)

1. **if** $d[v] > d[u] + w(u, v)$

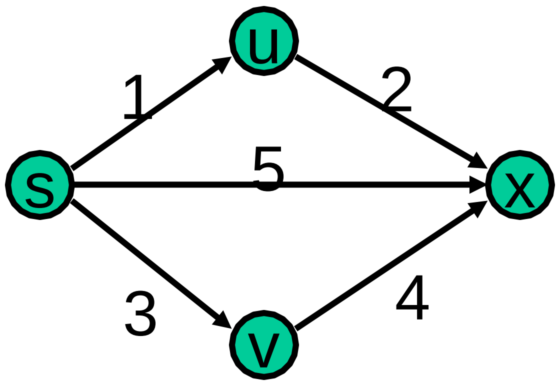
2. **then** $d[v] \leftarrow d[u] + w(u, v)$

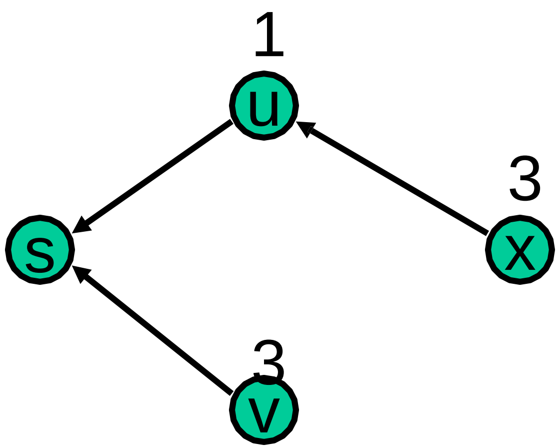
3. $\pi[v] \leftarrow u$

- All the single-source shortest-paths algorithms
  - start by calling INIT
  - then relax edges
- The algorithms differ in the order and how many times they relax each edge (it affects the correctness and time complexity of the algorithm)

Single Source Shortest Path Problem

- Given a graph and a start vertex s
  - Determine distance of every vertex from s
  - Identify shortest paths to each vertex
    - Express concisely as a “shortest paths tree”
    - Each vertex has a pointer to a predecessor on shortest path





Using BFS

Generalization of BFS to handle weighted graphs

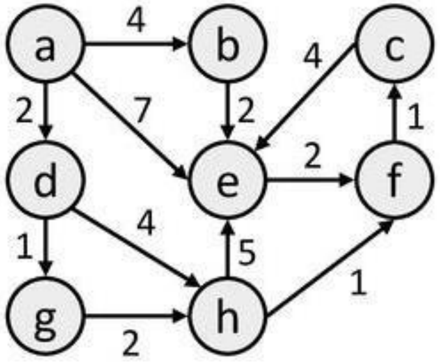
- Direct Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, edge weight *f*n ; w : $\boldsymbol{E} \longrightarrow \boldsymbol{R}$
- In BFS w(e)=1 for all e $\in$ E

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

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

Why BFS is not Enough

- Breadth-first visit order is “cautious” in the sense that it examines every path of length i before going on to paths of length i+1
- Breadth-first search does not work!
  - Minimum number of hops does not mean minimum distance.
  - BFS will yield a, e, f as SP between a and f
  - a, d, g, h, f has lower distance



A Greedy Algorithm

- Assume that every node is infinitely far away from the source.
  - i.e., every node is ∞ miles away from s (except s, which is 0 miles away).
  - Now perform something similar to breadth-first search, and *optimistically guess that we have found the best path to each node as we encounter it*.
  - If we later discover we are wrong and find a better path to a particular node, then update the distance to that node (edge Relaxation).

Intuition Behind Dijkstra's Algorithm

- For our SP problem, we can start by guessing that every node is ∞ miles away.
  - Mark each node with this guess.
- Find all verices/cities one hop away from s, and check whether the distance is less than what is currently marked for that node.
  - If so, then revise the guess.
- Continue for 2 hops, 3 hops, etc.

Dijkstra's Algorithm For Shortest Paths

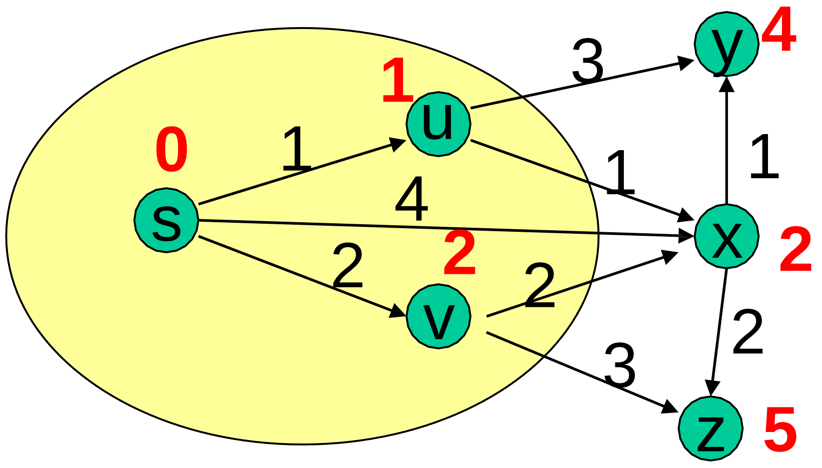
Assumes **no negative-weight edges**.

Maintains a set S of vertices whose SP from s has been determined.

Repeatedly selects u in V–S with minimum SP estimate (greedy choice).

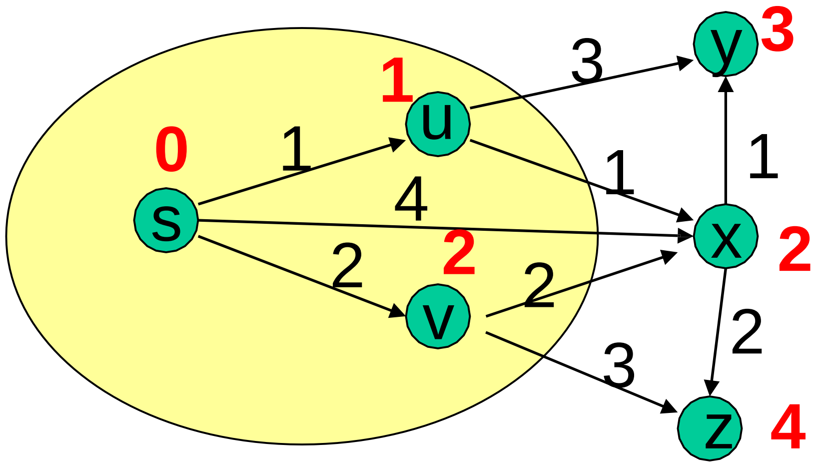
Store V–S in priority queue Q.

Like BFS: If all edge weights are equal, then use BFS, otherwise use this algorithm (note: BFS uses FIFO)



Dijkstra’s Algorithm For Shortest Paths

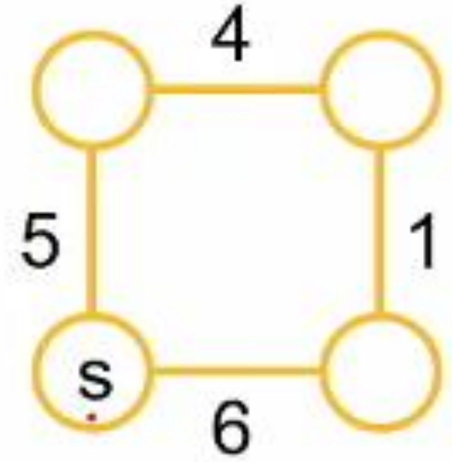
x is selected, then its edges are relaxed.



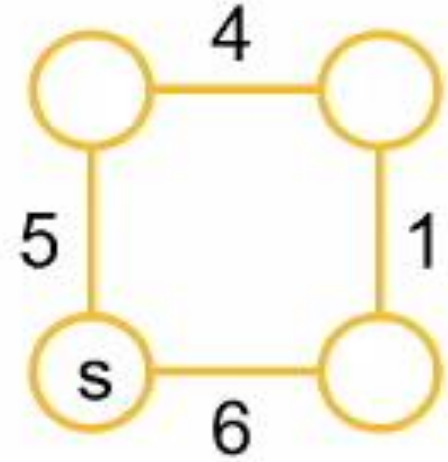
Dijkstra's Algorithm

- Similar to Prim's MST algorithm
- Start with source node s and iteratively construct a tree rooted at s
- Each node keeps track of tree node that provides cheapest path from s (not just cheapest path from any tree node)
- At each iteration, include the node whose cheapest path from s is the overall cheapest

Prim's vs. Dijkstra's

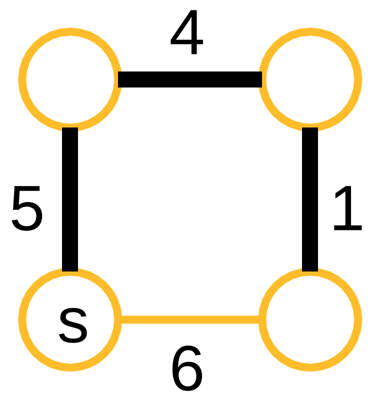


Prim's MST

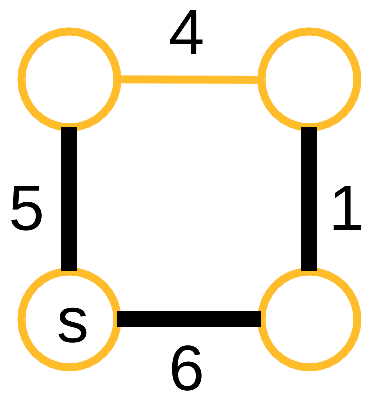


Dijkstra's SSSP

Prim's vs. Dijkstra's



Prim's MST



Dijkstra's SSSP

Dijkstra's Algorithm For Shortest Paths

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

Running Time Analysis

- Look at different Q implementation, as did for Prim's algorithm
- Initialization (INIT) : Θ(V) time
- While-loop:
  - ***EXTRACT-MIN*** executed |V| times
  - ***DECREASE-KEY*** executed |E| times

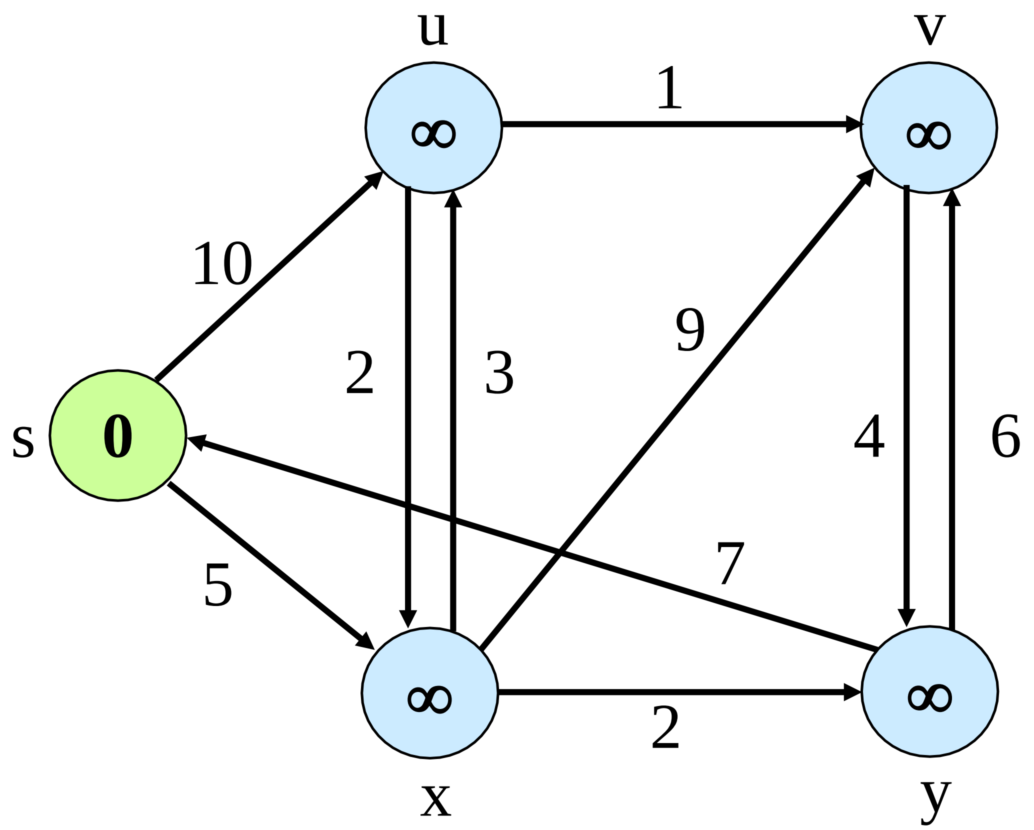
- Time $T = |V| \times T_{E\text{-}MIN} + |E| \times T_{D\text{-}KEY}$

Running Time Analysis

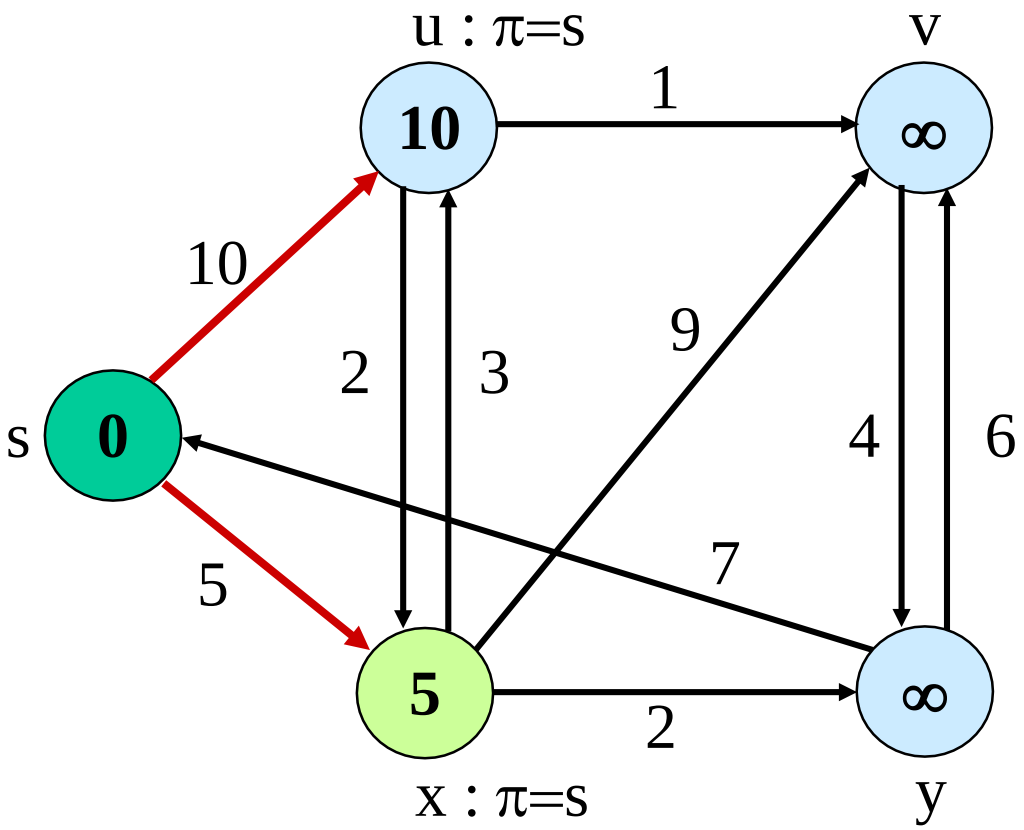
Look at different Q implementation

Q	$T_{E\text{-}MIN}$	$T_{D\text{-}KEY}$	TOTAL
• Linear Unsorted Array:	O(V)	O(1)	$O(V^2+E)$
• Binary Heap:	O(lgV)	O(logV)	O(VlgV+ElgV) = O(ElgV)

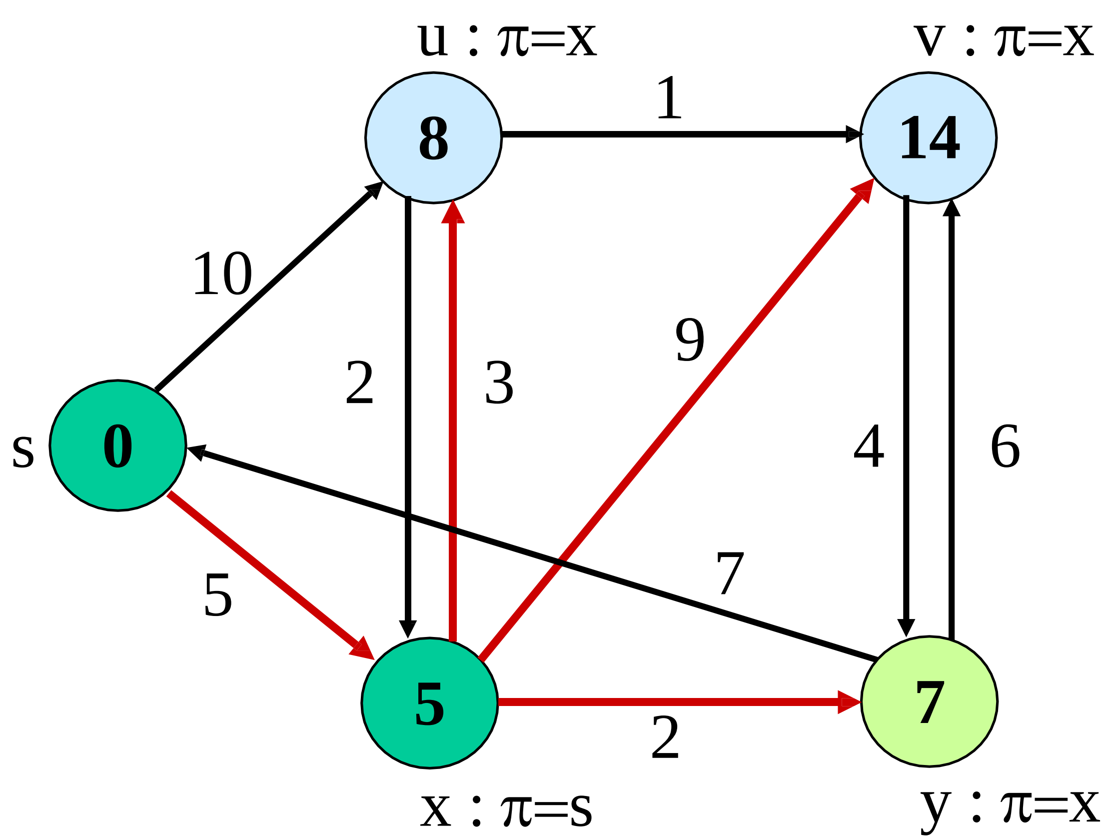
Example



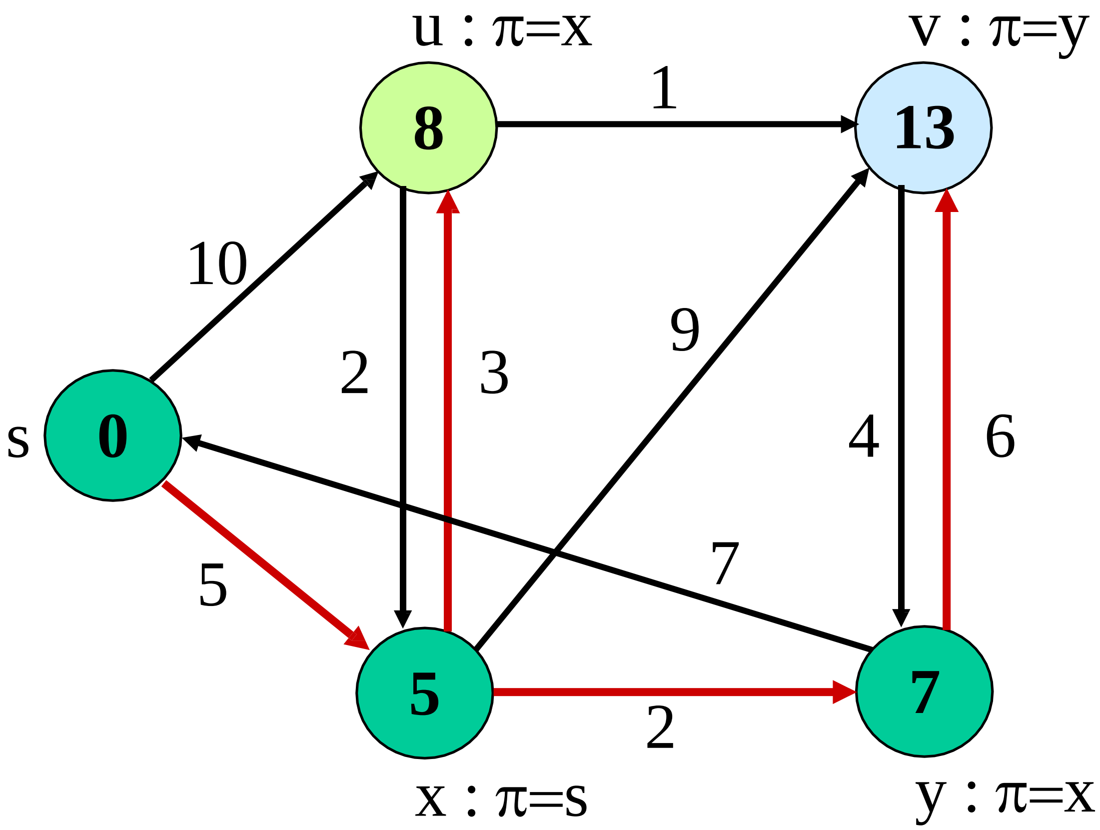
Example



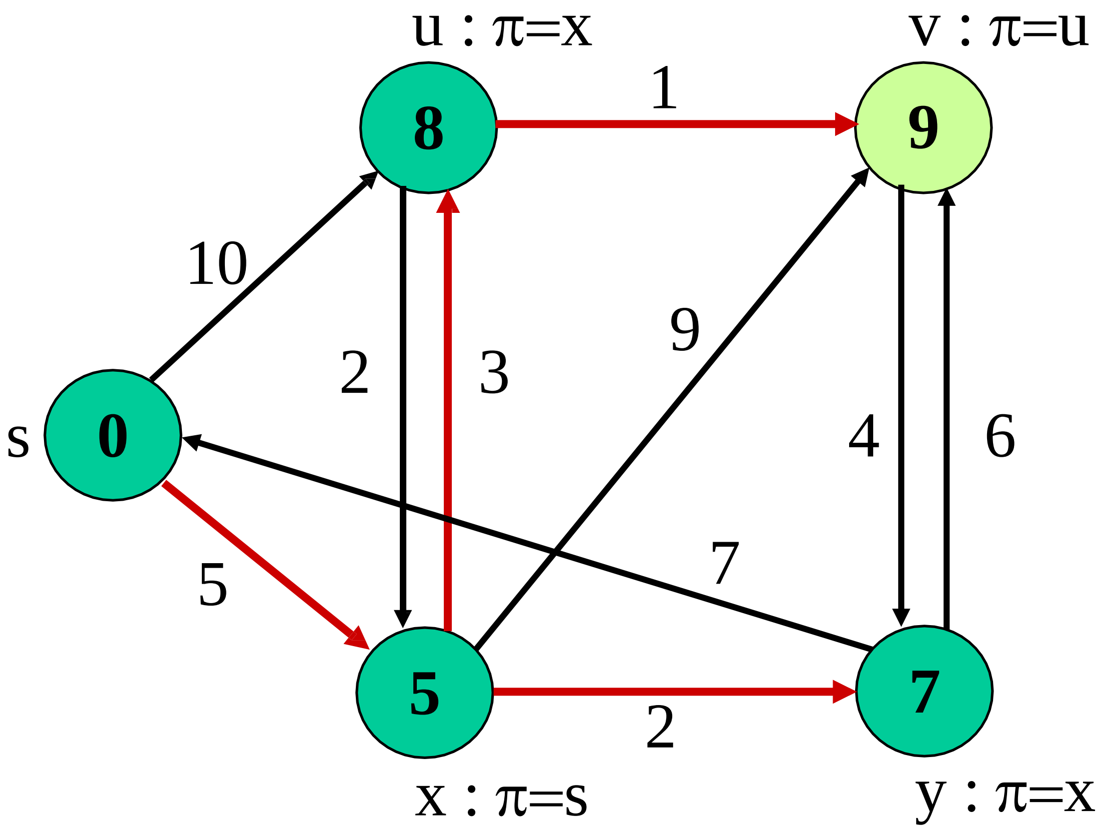
Example



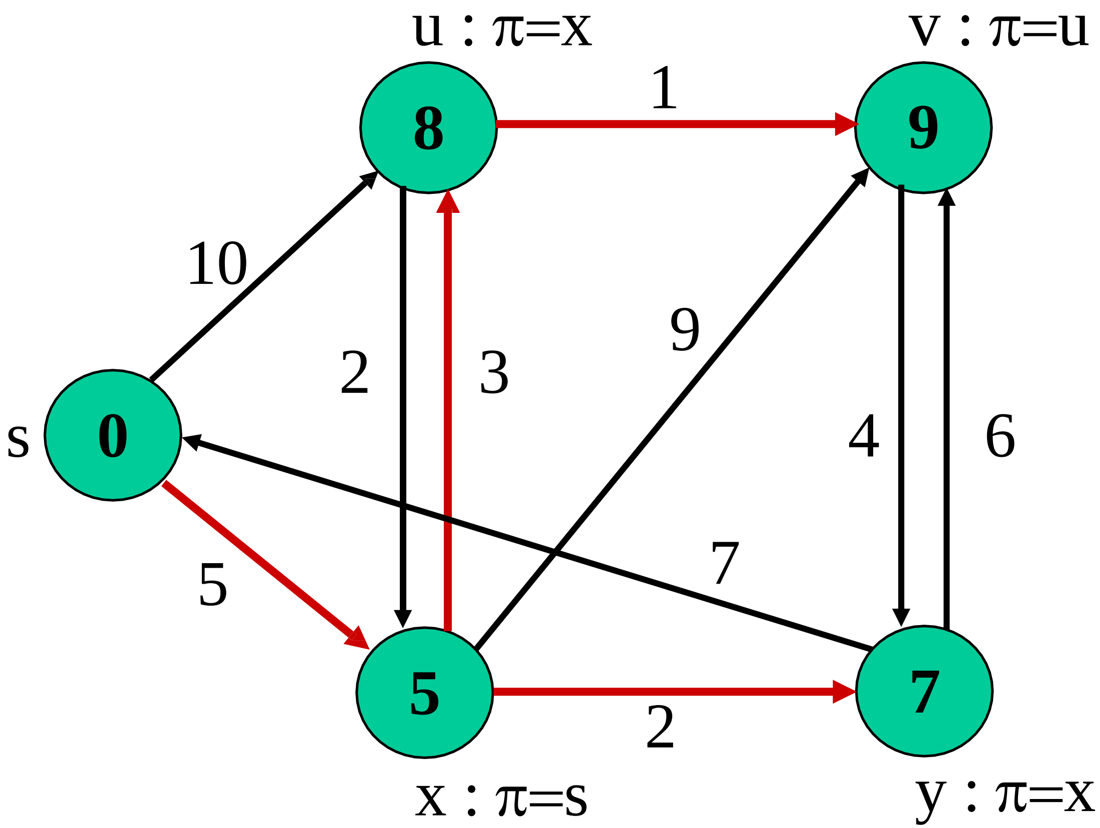
Example



Example



Example



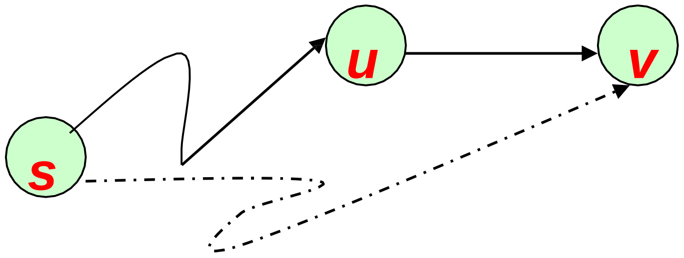
Triangle Inequality

Lemma 1: for a given vertex $s \in \boldsymbol{V}$ *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 no longer than any other path.*

- in particular the path that takes the shortest path $s \rightsquigarrow v$ *and* then takes cycle (u,v)



Properties of Relaxation

Algorithms 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 *L1*
  - $d[u]+w(u,v) < \delta(s, u) + w(u, v) \Rightarrow d[u] < \delta(s, u)$

    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 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* $u,v \in V$

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

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

Correctness

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

Proof:

By Lemma3(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.

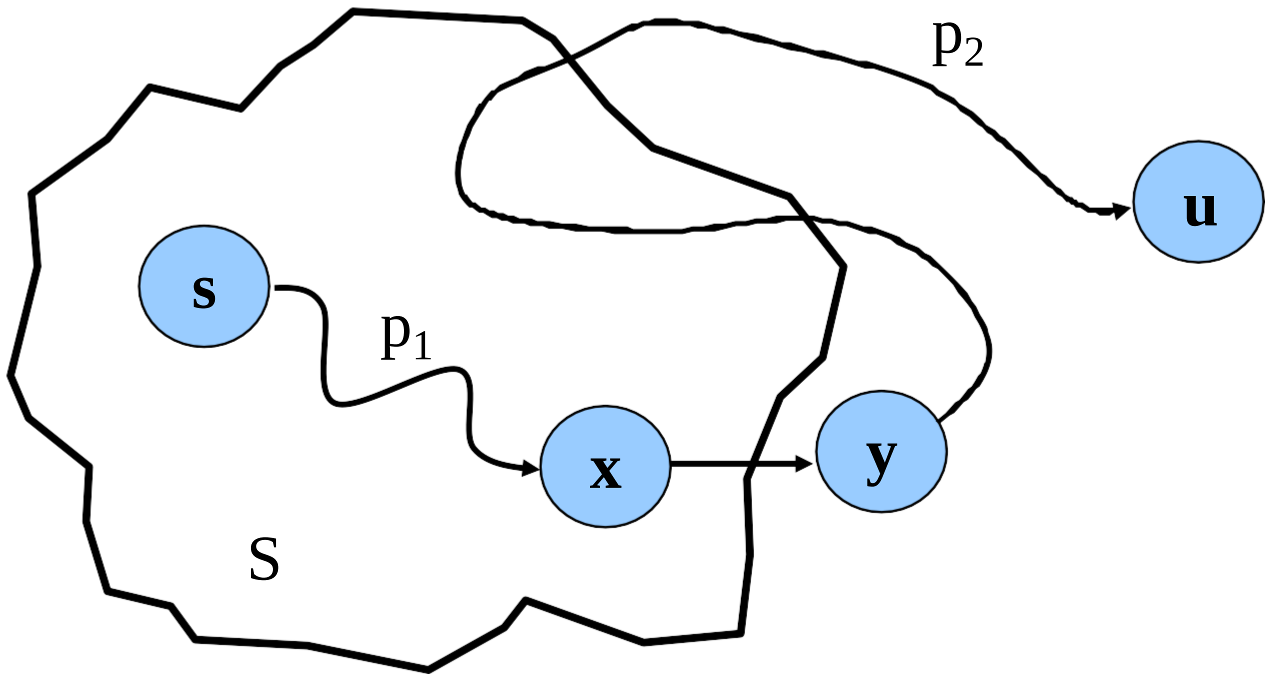
Note that $d[s] = \delta(s, s) = 0$ when s is inserted, so $u \neq s$.

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

Proof (Continued)

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

$\Rightarrow$ there exists a SP from s to u. 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 Lemma4, 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 Lemma3(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.