CSC3100: Graph

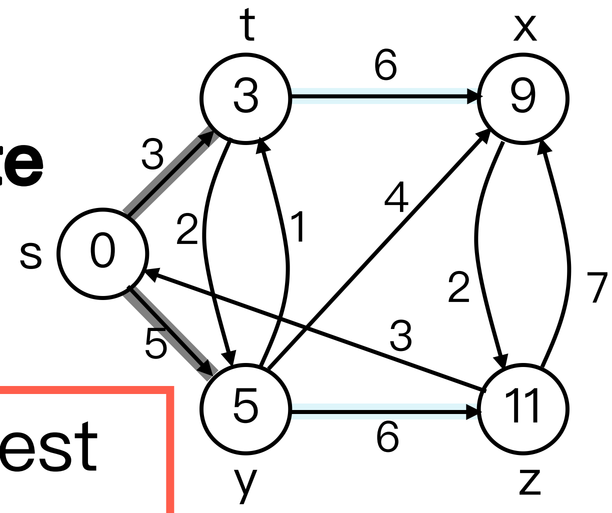
Tianshu Yu

2021–2022

Shortest–Paths Notation

For each vertex $v \in V$:

- $\delta(s, v)$: **shortest–path weight**
- $d[v]$: shortest–path weight **estimate**
  - Initially, $d[v]=\infty$
  - $d[v] \rightarrow \delta(s,v)$ as algorithm progresses
- $\pi[v]$ = **predecessor** of v on a shortest path from ***s***
  - If no predecessor, $\pi[v]$ = NIL
  - π induces a tree—**shortest–path tree**



Relaxation Step

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

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

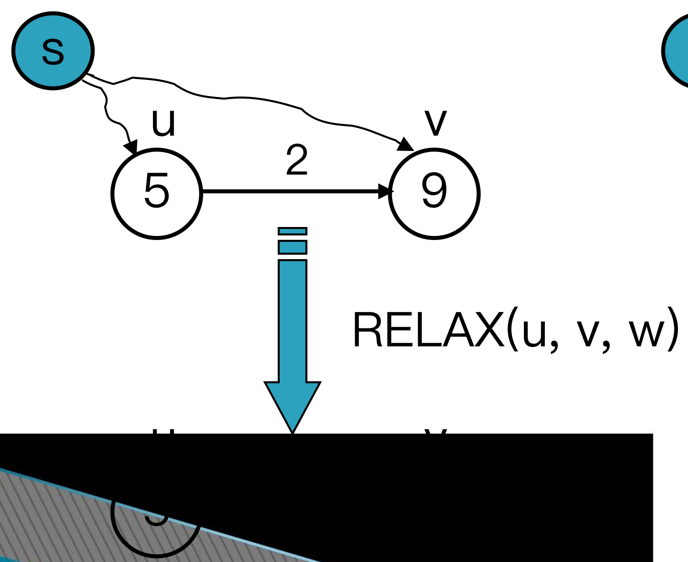
  we can improve the shortest path to v

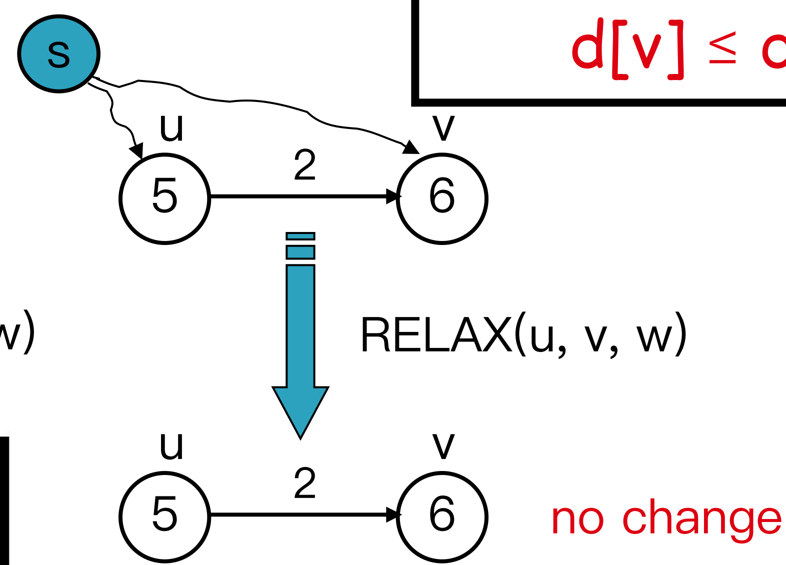
  $\Rightarrow d[v]=d[u]+w(u,v)$

  $\Rightarrow \pi[v] \leftarrow u$

After relaxation:

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





Bellman–Ford Algorithm

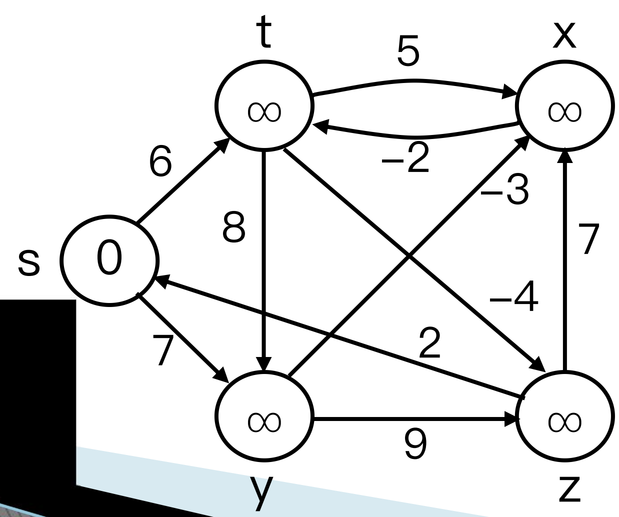
- Single-source shortest path problem
  - Computes $\delta(s, v)$ and $\pi[v]$ for all $v \in V$

- Allows negative edge weights – can detect negative cycles.
  - Returns TRUE if no negative-weight cycles are reachable from the source s
  - Returns FALSE otherwise $\Rightarrow$ no solution exists

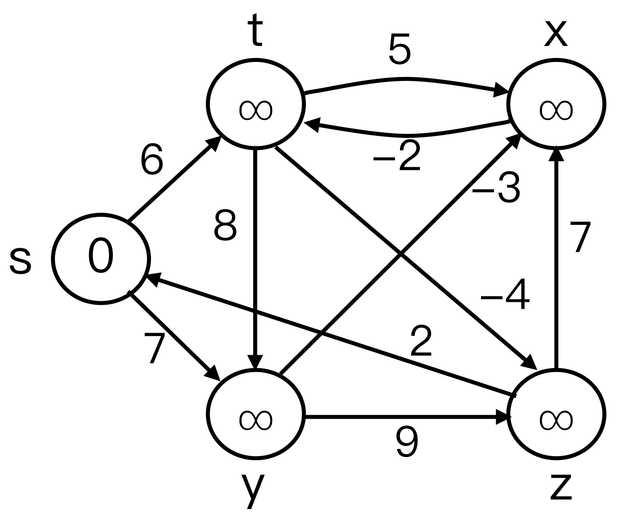
Bellman–Ford Algorithm (cont'd)

- Idea:
  - Each edge is relaxed |V–1| times by making |V–1| passes over the whole edge set.
  - Any path will contain at most |V–1| edges

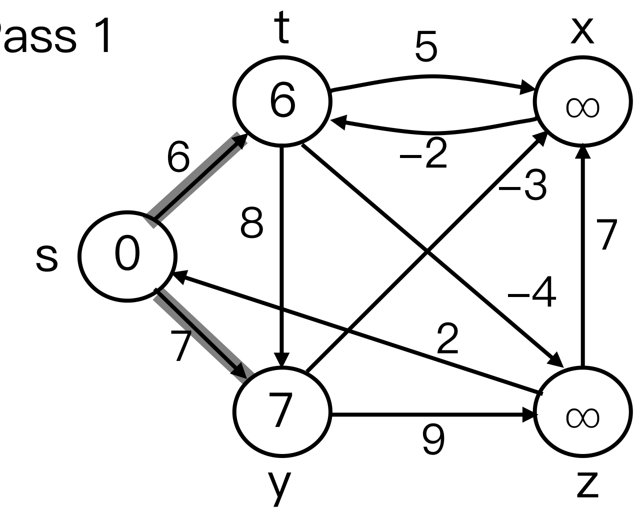
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



BELLMAN-FORD(V, E, w, s)



Pass 1

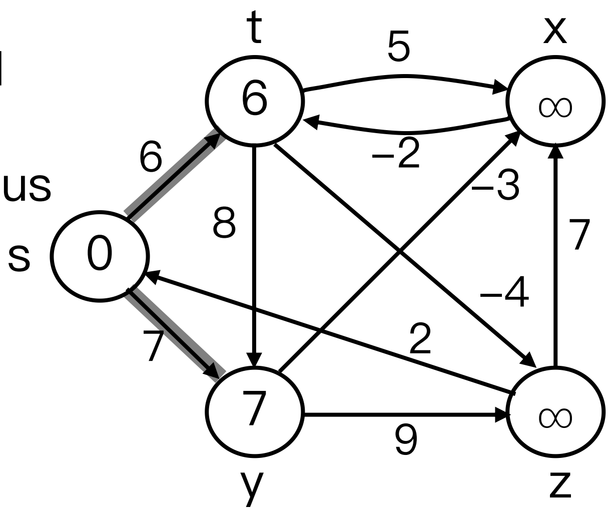


E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

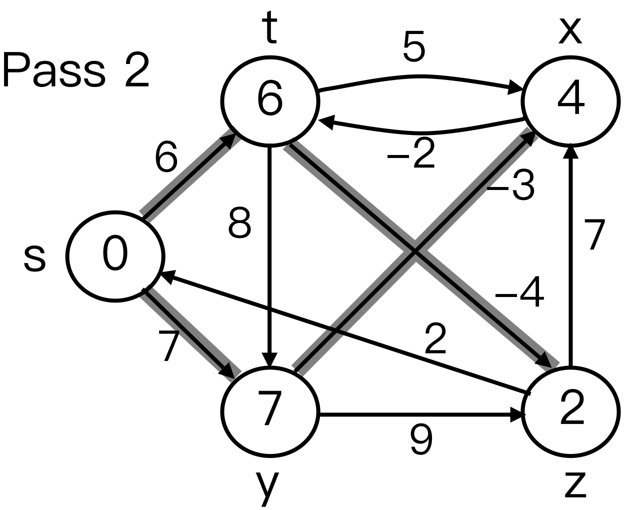
Example

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

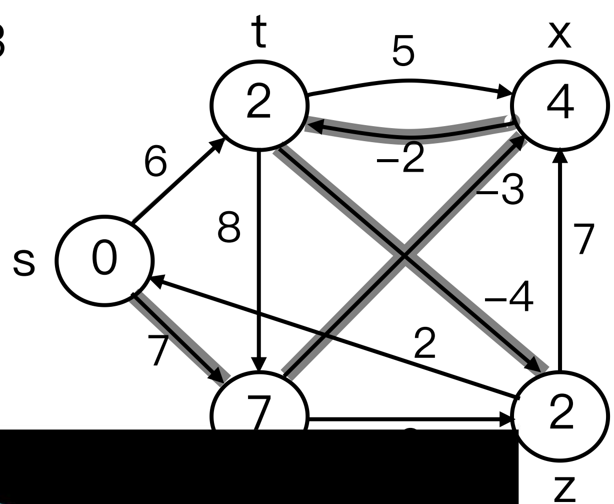
Pass 1 (from previous slide)



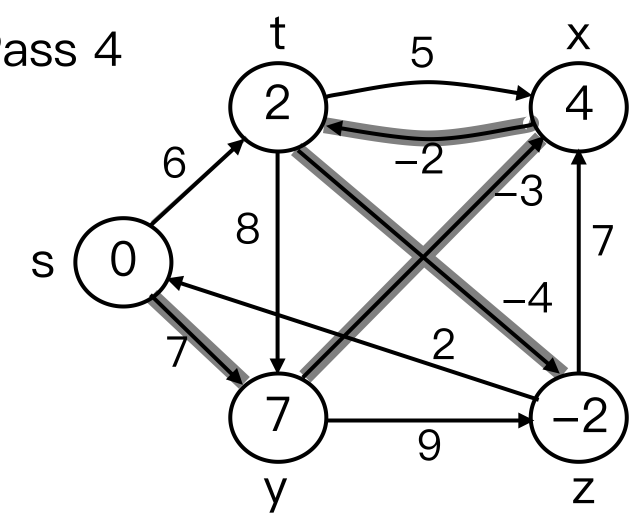
Pass 2



Pass 3

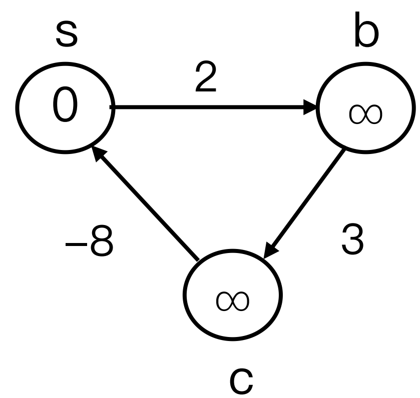


Pass 4

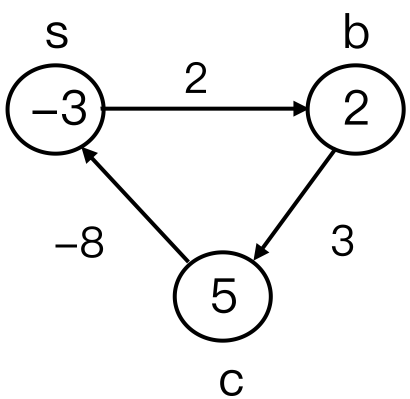


Detecting Negative Cycles
(perform extra test after V–1 iterations)

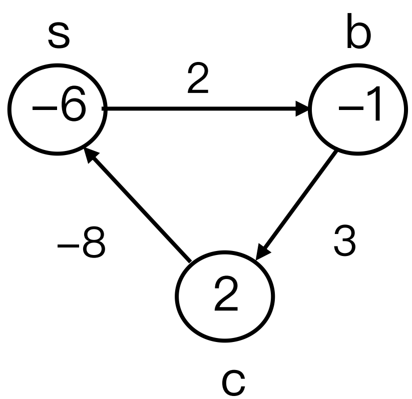
- **for** each edge $(u, v) \in E$
- **do if** $d[v] > d[u] + w(u, v)$
- **then return** FALSE
- **return** TRUE



1st pass



2nd pass



[redacted]

Look at edge (s, b):

$d[b] = -1$
$d[s] + w(s, b) = -4$

$\Rightarrow d[b] > d[s] + w(s, b)$

BELLMAN-FORD(V, E, w, s)

```
INITIALIZE-SINGLE-SOURCE(V, s)              ← Θ(V)
for i ← 1 to |V| - 1                        ← O(V)
    do for each edge (u, v) ∈ E             ← O(E)
            do RELAX(u, v, w)
for each edge (u, v) ∈ E                    ← O(E)
    do if d[v] > d[u] + w(u, v)
            then return FALSE
return TRUE
```

$\left.\begin{array}{l} O(V) \\ O(E) \end{array}\right\}$ **O(VE)**

Running time: O(V+VE+E)=O(VE)

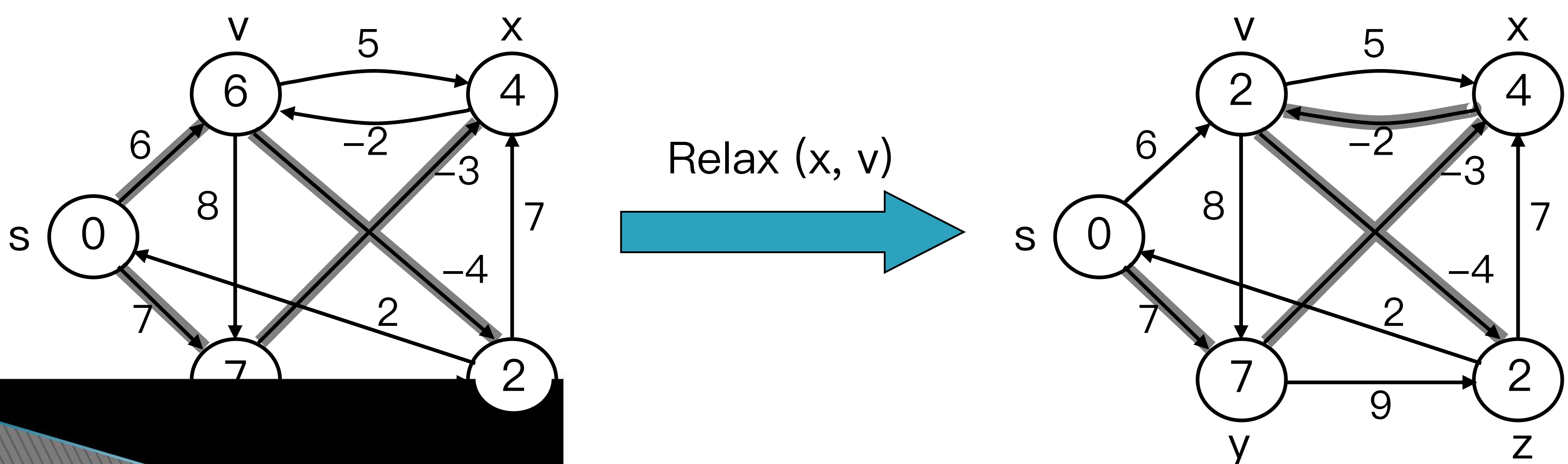
Key points of BELLMAN–FORD

- After ||V||–1 iterations, d values will not be updated or can't be lower any more. Why?

- After ||V||–1 iterations, d values store the measure of the shortest path. Why?

Shortest Path Properties

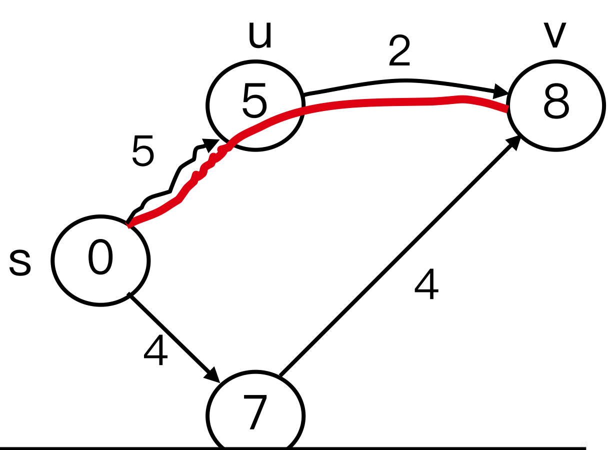
- **Upper-bound property**
  - We always have d[v] ⩾ δ (s, v) for all v.
  - The estimate never goes up — relaxation only lowers the estimate



Shortest Path Properties

- **Convergence property**

  If $s \rightsquigarrow u \rightarrow v$ is a shortest path, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times after relaxing (u, v).



- If $d[v] > \delta(s, v) \Rightarrow$ after relaxation:

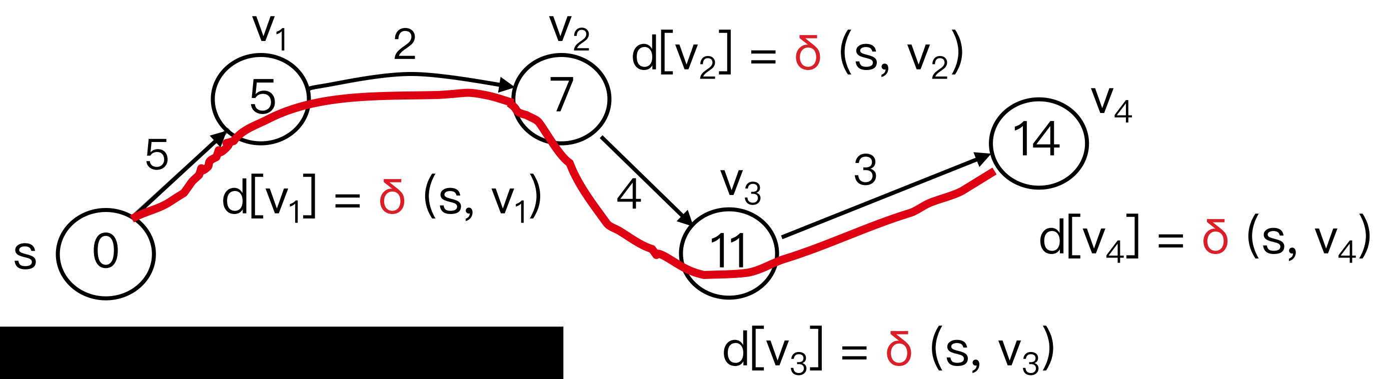
  $d[v] = d[u] + w(u, v)$

  $d[v] = 5 + 2 = 7$
- Otherwise, the value remains unchanged, because it must have been the shortest path value

Shortest Path Properties

- **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 $d[v_k] = \delta(s, v_k)$.



Correctness of Belman-Ford Algorithm

- **Theorem:** $d[v] = \delta(s, v)$, for every v, after $|V-1|$ passes.

  Case 1: G does not contain negative cycles which are reachable from s

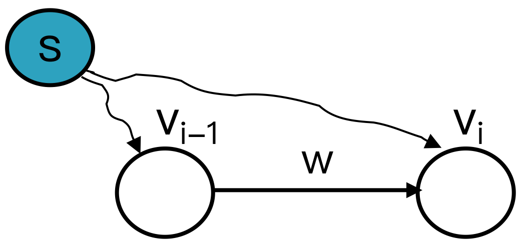
  - Assume that the shortest path from s to v is
    $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s=v_0$ and $v=v_k$, $k \leqslant |V-1|$
  - Use mathematical induction on the number of passes i to show that:
    $d[v_i] = \delta(s, v_i)$, $i=0,1,\ldots,k$

Correctness of Belman–Ford Algorithm (cont.)

Base Case: i=0 $d[v_0]= \delta\ (s, v_0)= \delta\ (s, s)= 0$

Inductive Hypothesis: $d[v_{i-1}]= \delta\ (s, v_{i-1})$

Inductive Step: $d[v_i]= \delta\ (s, v_i)$



$d[v_{i-1}]= \delta\ (s, v_{i-1})$

After relaxing (v_{i-1}, v_i) (convergence property) :

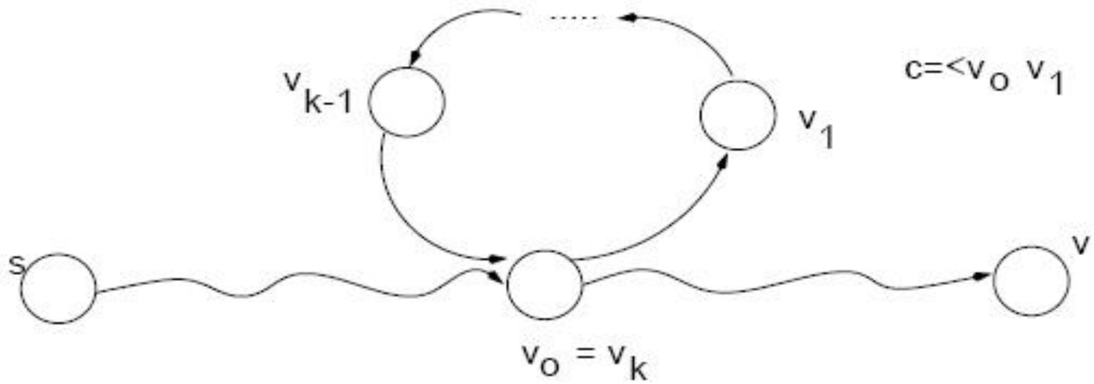
$d[v_i] \leqslant d[v_{i-1}]+w= \delta\ (s, v_{i-1})+w= \delta\ (s, v_i)$

From the upper bound property: $d[v_i] \geqslant \delta\ (s, v_i)$

[redacted] v_i)

Correctness of Belman–Ford Algorithm (cont.)

- Case 2: G contains a negative cycle which is reachable from s



c=<v o v 1 ... v k> is a negative cycle

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

Proof by Contradiction: suspose the algorithm returns a solution

After relaxing (v_{i-1}, v_i): $dist[v_i] \leq dist[v_{i-1}] + w(v_{i-1}, v_i)$

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

$$\Rightarrow \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0 \ \left(\sum_{i=1}^{k} dist[v_i] = \sum_{i=1}^{k} dist[v_{i-1}]\right)$$

Contradiction!

Spanning Trees

- Given (connected) graph $G(V,E)$, a spanning tree $T(V',E')$:
  - Is a subgraph of G; that is, $V' \subseteq V$, $E' \subseteq E$.
  - Spans the graph $(V' = V)$
  - Forms a tree (no cycle);
  - So, E' has $|V| - 1$ edges

Minimum Spanning Trees

- Edges are weighted: find minimum cost spanning tree
- Applications
  - Find cheapest way to wire your house
  - Find minimum cost to send a message on the Internet

Strategy for Minimum Spanning Tree

- For any spanning tree T, inserting an edge e_{new} not in T creates a cycle
- But
  - Removing any edge e_{old} from the cycle gives back a spanning tree
  - If e_{new} has a lower cost than e_{old} we have progressed!

Strategy

- Strategy for construction:
  - Add an edge of minimum cost that does not create a cycle (greedy algorithm)
  - Repeat |V| –1 times
  - Correct since if we could replace an edge with one of lower cost, the algorithm would have picked it up

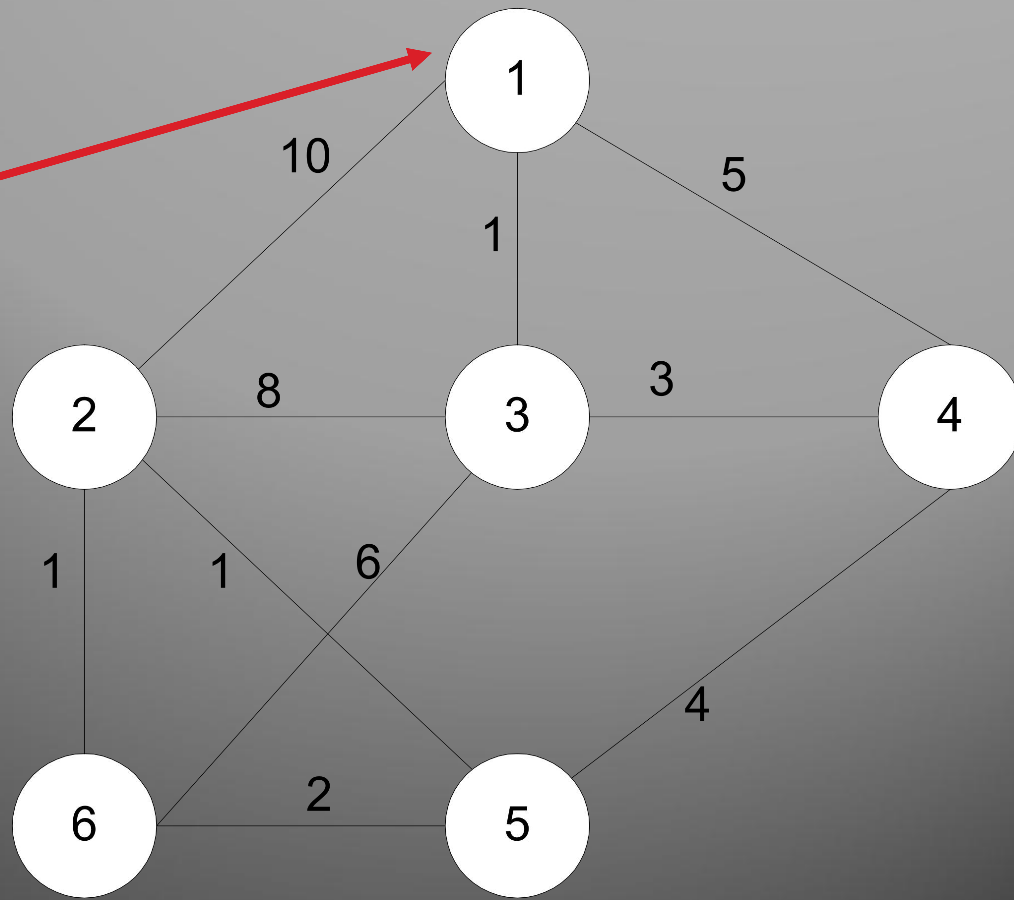
Two Algorithms

- Prim: (build tree incrementally)
  - Pick lower cost edge connected to known (incomplete) spanning tree that does not create a cycle and expand to include it in the tree
- Kruskal: (build forest that will finish as a tree)
  - Pick lowest cost edge not yet in a tree that does not create a cycle. Then expand the set of included edges to include it. (It will be somewhere in the forest.)

Prim's algorithm

Starting from empty T, choose a vertex at random and initialize

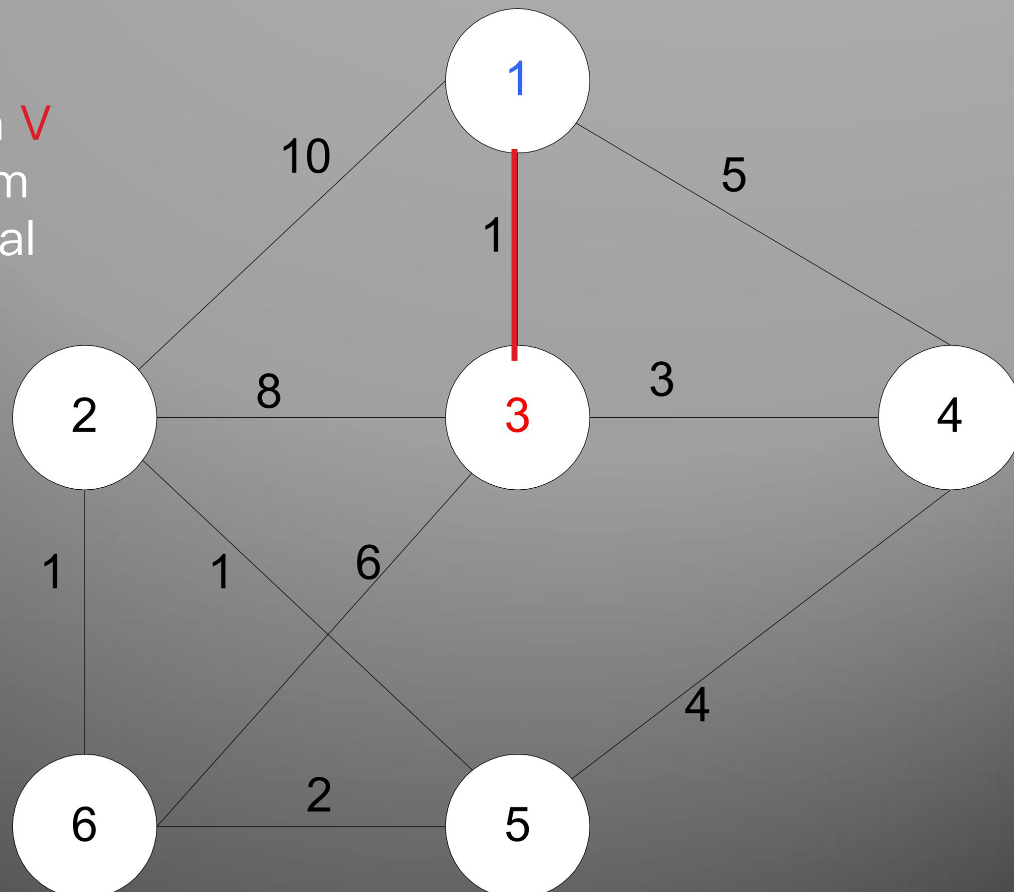
V = {1), E' ={}



Prim's algorithm

Choose the vertex u not in V such that edge weight from u to a vertex in V is minimal (greedy!)

V={1,3} E'= {(1,3) }



Prim's algorithm

Repeat until all vertices have been chosen

Choose the vertex u not in V such that edge weight from v to a vertex in V is minimal (greedy!)

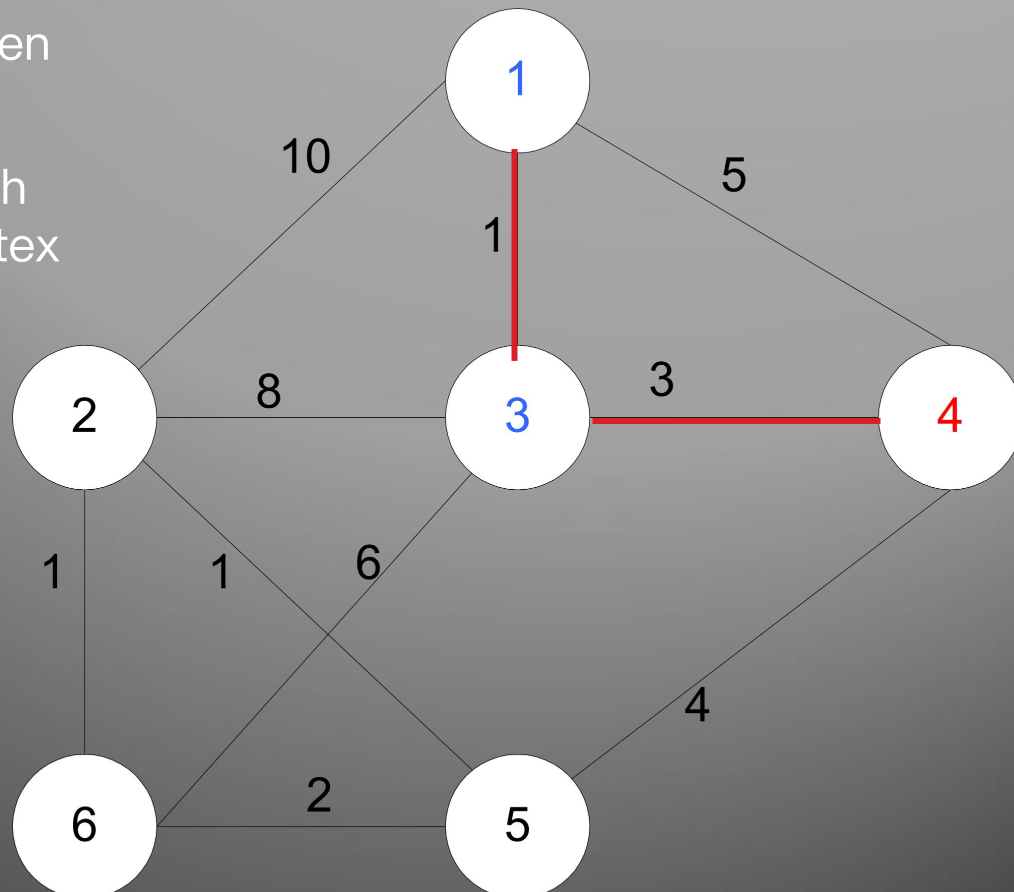
V= {1,3,4} E'= {(1,3),(3,4)}

V={1,3,4,5} E'={(1,3),(3,4),(4,5)}

….

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}



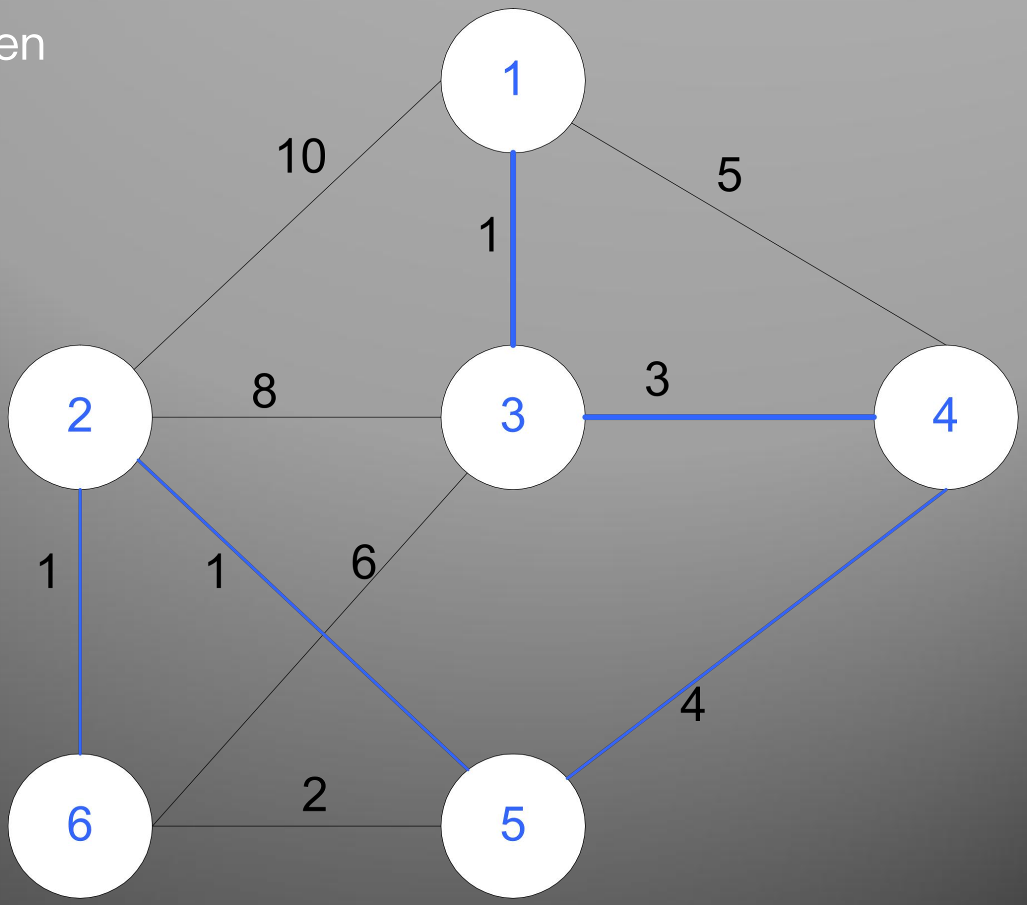
Prim's algorithm

Repeat until all vertices have been chosen

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}

Final Cost: 1 + 3 + 4 + 1 + 1 = 10



Prim's Algorithm Implementation

- Assume adjacency list representation

Initialize connection cost of each node to "inf" and "unmark" them

Choose one node, say v and set cost[v] = 0 and prev[v] =0

While they are unmarked nodes

    Select the unmarked node **u** with minimum cost; mark it

    For each unmarked node **w** adjacent to **u**

        if cost(u,w) < cost(w) then cost(w) := cost (u,w)

            prev[w] = u

Kruskal's Algorithm

- Select edges in order of increasing cost
- Accept an edge to expand tree or forest only if it does not cause a cycle

Kruskal's Algorithm

Initialize a forest of trees, each tree being a single node

Build a priority queue of edges with priority being lowest cost

Repeat until |V| –1 edges have been accepted {

  Deletemine edge from priority queue

  If it forms a cycle then discard it

    else accept the edge – It will join 2 existing trees yielding a larger tree and reducing the forest by one tree

}

The accepted edges form the minimum spanning tree

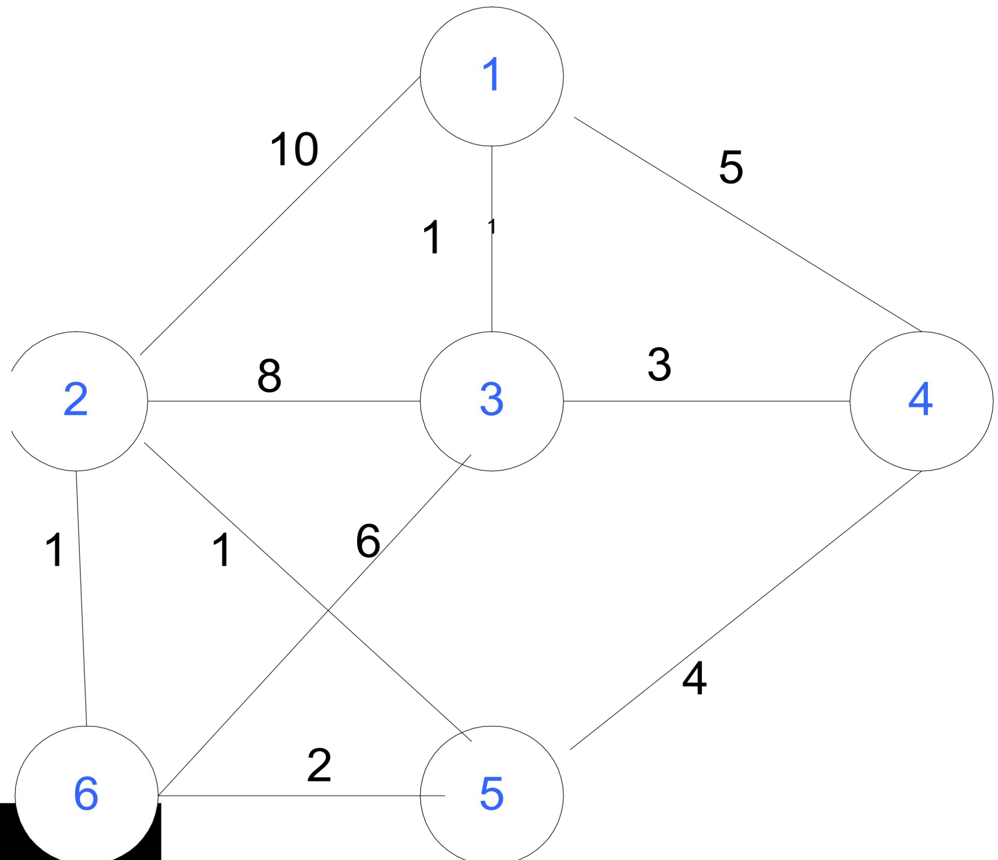
Detecting Cycles

- If the edge to be added (u,v) is such that vertices u and v belong to the same tree, then by adding (u,v) you would form a cycle
  - Therefore to check, Find(u) and Find(v). If they are the same discard (u,v)
  - If they are different Union(Find(u),Find(v))

Properties of trees in K's algorithm

- Vertices in different trees are disjoint
  - True at initialization and Union won't modify the fact for remaining trees
- Trees form equivalent classes under the relation "is connected to"
  - u connected to u (reflexivity)
  - u connected to v implies v connected to u (symmetry)
  - u connected to v and v connected to w implies a path from u to w so u connected to w (transitivity)

Example

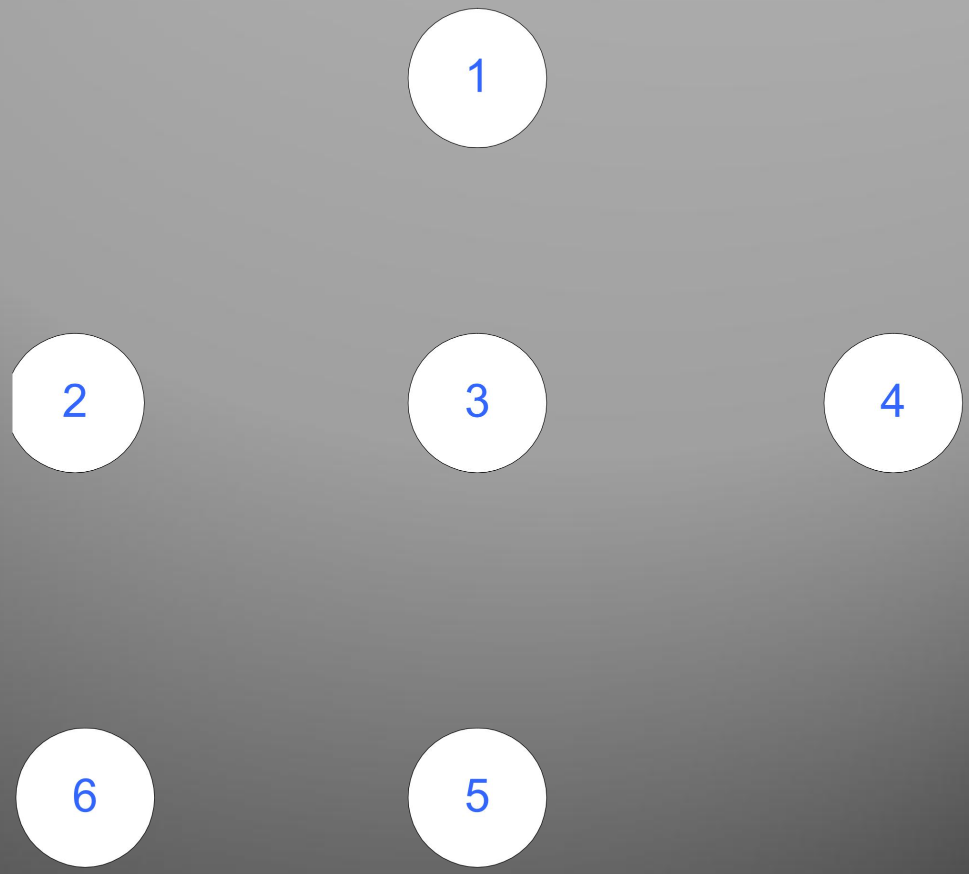


Initialization

Initially, Forest of 6 trees

F= {{1},{2},{3},{4},{5},{6}}

Edges in a heap (not shown)



Step 1

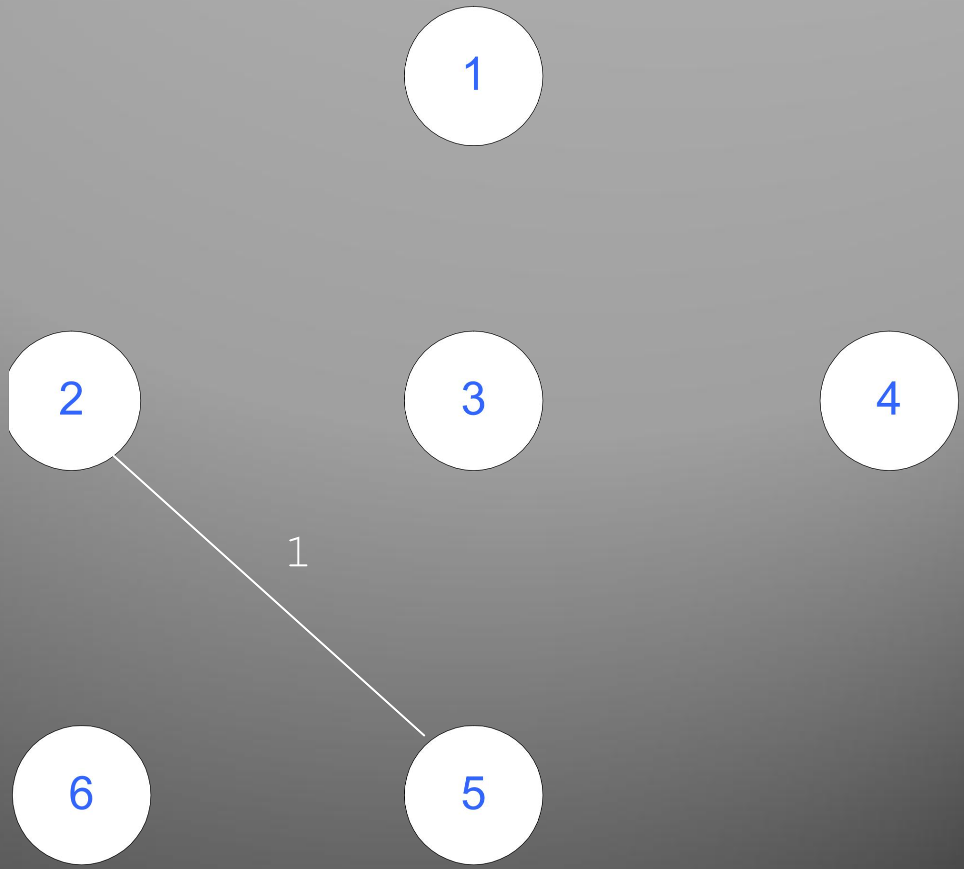
Select edge with lowest cost (2,5)

Find(2) = 2, Find (5) = 5

Union(2,5)

F= {{1},{2,5},{3},{4},{6}}

1 edge accepted



Step 2

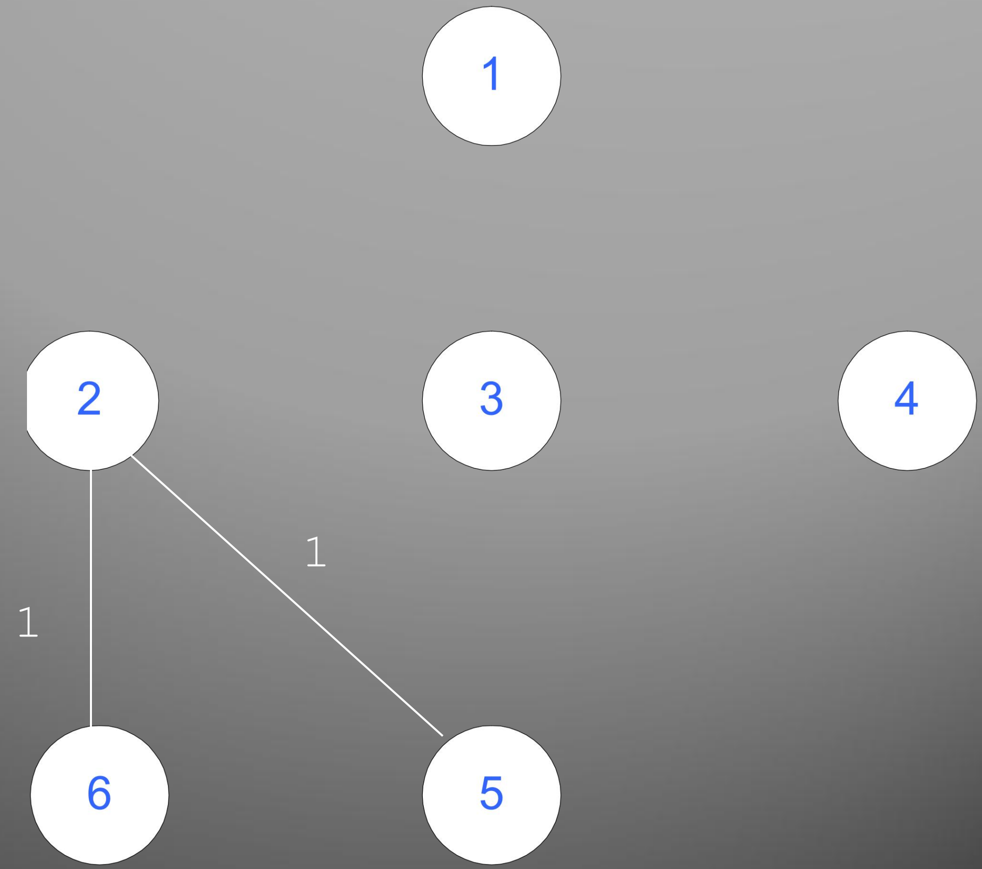
Select edge with lowest cost (2,6)

Find(2) = 2, Find (6) = 6

Union(2,6)

F= {{1},{2,5,6},{3},{4}}

2 edges accepted



Step 3

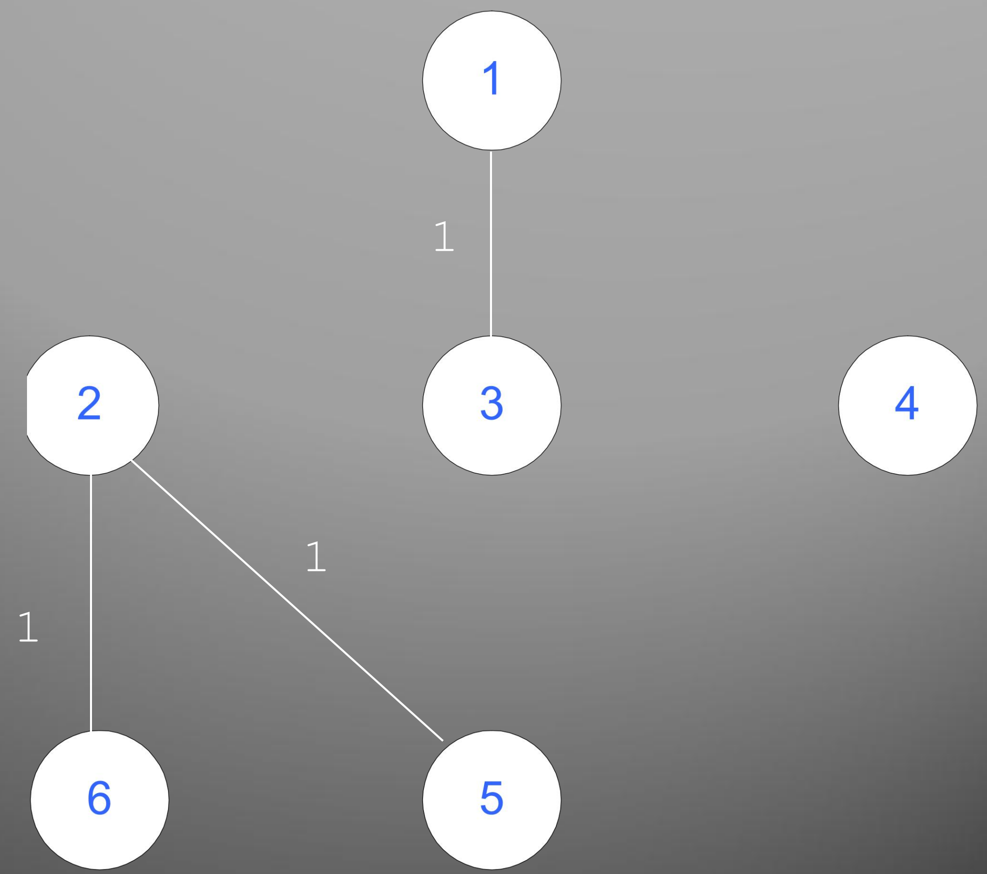
Select edge with lowest cost (1,3)

Find(1) = 1, Find (3) = 3

Union(1,3)

F= {{1,3},{2,5,6},{4}}

3 edges accepted



Step 4

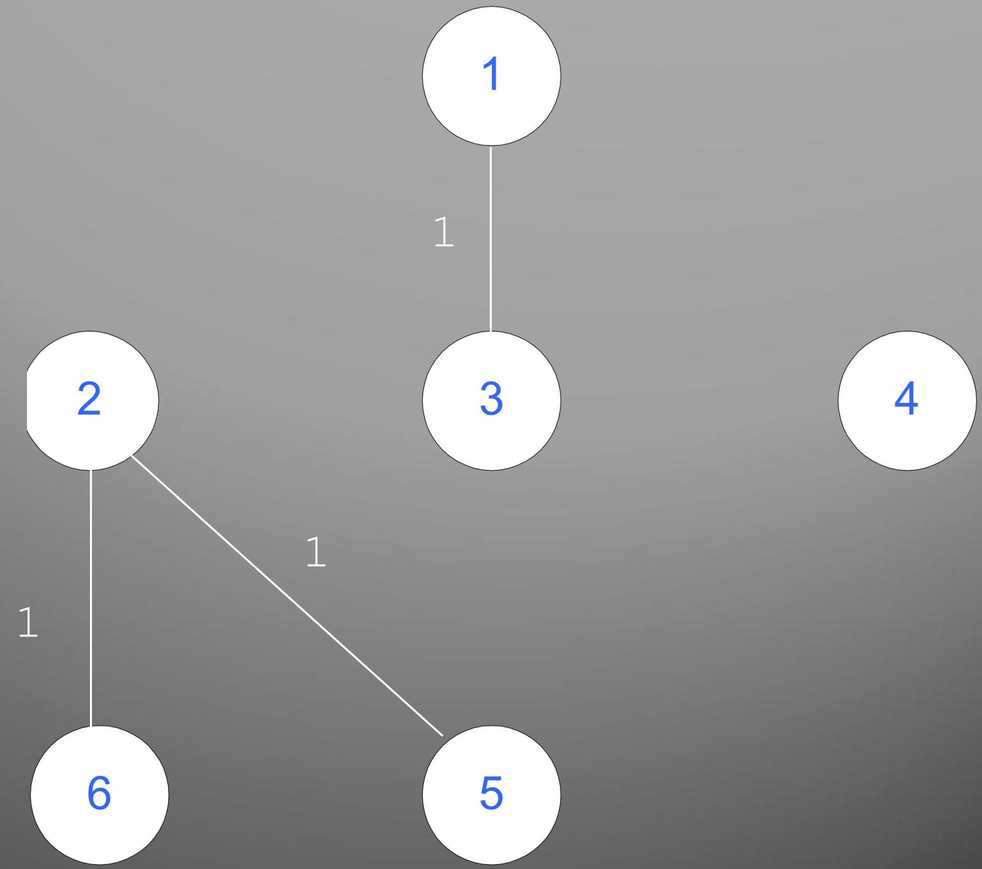
Select edge with lowest cost (5,6)

Find(5) = 2, Find (6) = 2

Do nothing

F= {{1,3},{2,5,6},{4}}

3 edges accepted



Step 5

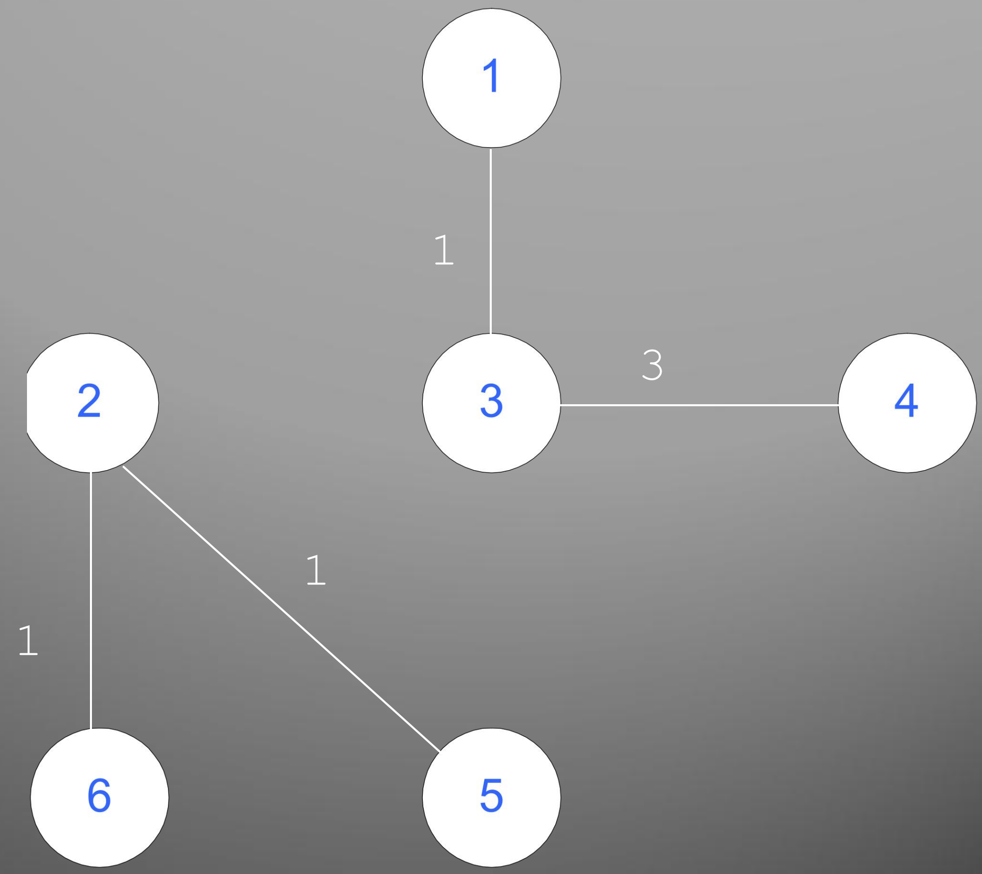
Select edge with lowest cost (3,4)

Find(3) = 1, Find (4) = 4

Union(1,4)

F= {{1,3,4},{2,5,6}}

4 edges accepted



Step 6

Select edge with lowest cost (4,5)

Find(4) = 1, Find (5) = 2

Union(1,2)

F= {{1,3,4,2,5,6}}

5 edges accepted : end

Total cost = 10

Although there is a unique spanning tree in this example, this is not generally the case