Graph Algorithms

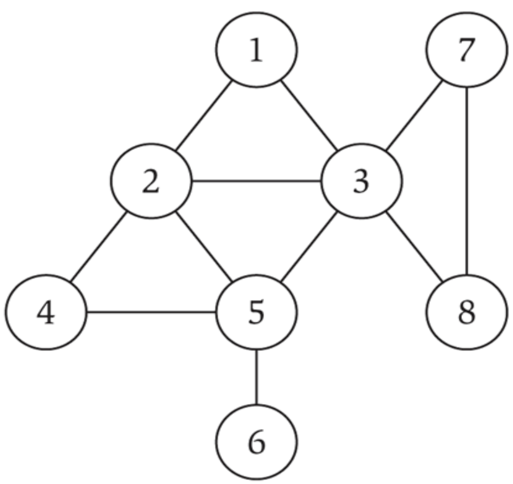
Part 1 BFS & DFS

CS 325

Introduction to graph theory

Graph – mathematical object consisting of a set of:

- Denoted by $G = (V, E)$.
- V = **nodes** (vertices, points). $V(G)$ and V_G
- E = **edges** (links, arcs) between pairs of nodes. Also denoted by $E(G)$ and E_G ; $E \subseteq V \times V$
- **Graph size** parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { (1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6) }

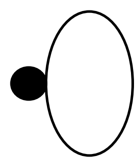
n = 8

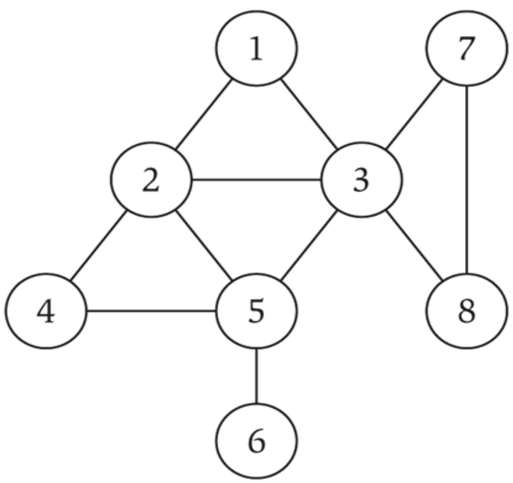
m = 11

Introduction to graph theory

For graph *G(V,E)*:

- If edge e=*(u,v)* ∈ *E(G)*, we say that *u* and *v* are **adjacent** or **neighbors**
- *u* and *v* are **incident** with *e*
- *u* and *v* are **end-vertices** of *e*
- An edge where the two end vertices are the same is called a **loop**, or a **self-loop**





V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { (1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6) }

n = 8

m = 11

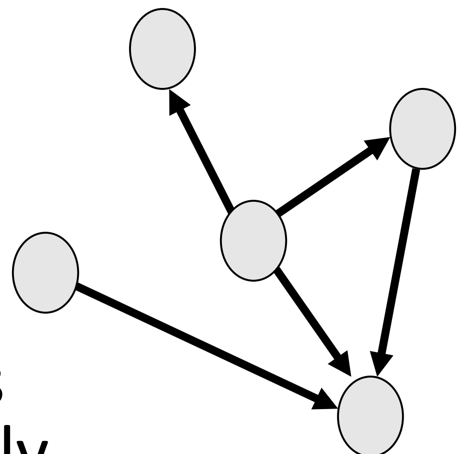
3

Directed graph (digraph)

- Directed edge
  - ordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$

- Undirected edge
  - unordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$

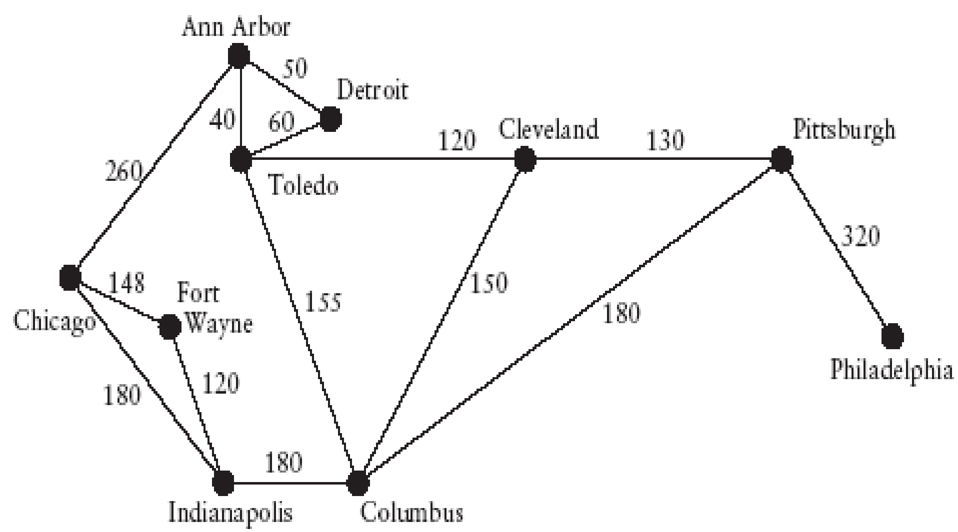
- A graph with directed edges is called a directed graph or digraph



- A graph with undirected edges is an undirected graph or simply a graph

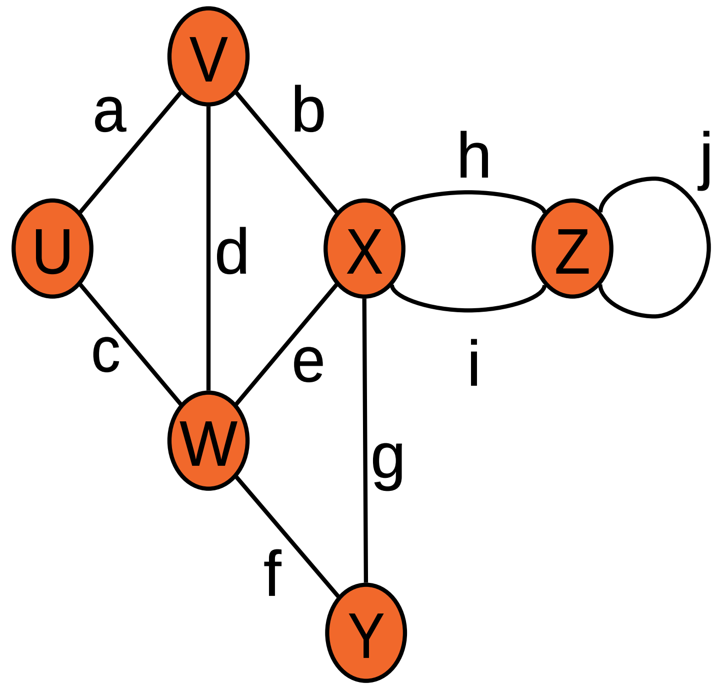
Weighted Graphs

- The edges in a graph may have values associated with them known as their weights
- A graph with weighted edges is known as a weighted graph



Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Self-loop
  - j is a self-loop

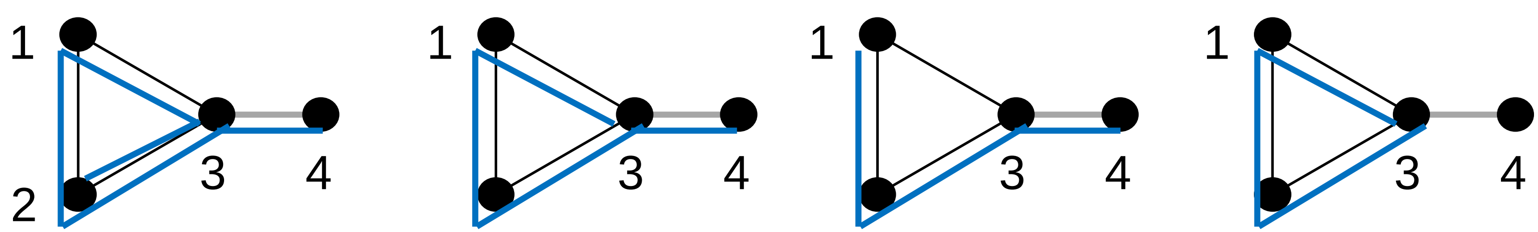


Terminology

- A **path** in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E. So, nodes can repeat, but edges do not.

- A **walk:** a path in which edges/nodes can be repeated.

- A path is **simple** if all nodes are distinct.
- A **cycle** is a path in which the first and final vertices are the same



Walk:
(2,3), (3,1), (1,2), (2,3), (3,4)

Path:
(3,1), (1,2), (2,3), (3,4)

Simple Path:
(1,2), (2,3), (3,4)

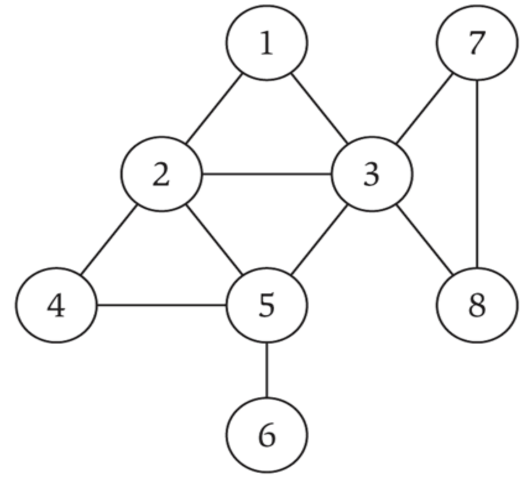
Cycle

Terminology

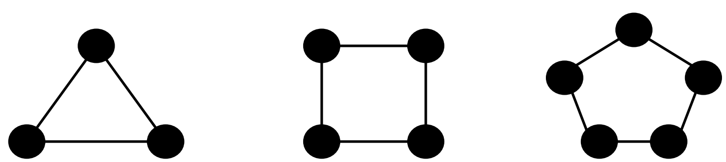
Simple cycle: a cycle in which all vertices and edges are distinct

simple cycle C = 1-2-4-5-3-1

1-2-3-7-8-3-1 is **not** a simple cycle,



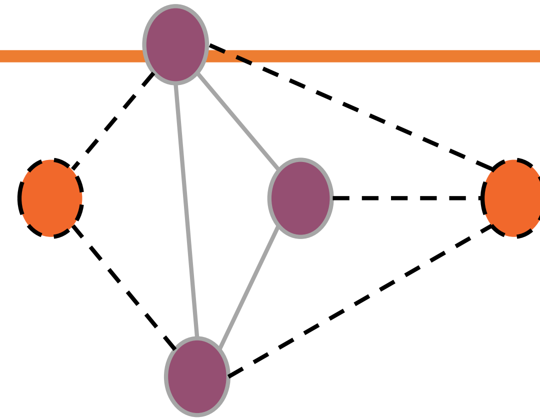
Cycles are denoted by C_k, where k is the number of nodes in the cycle



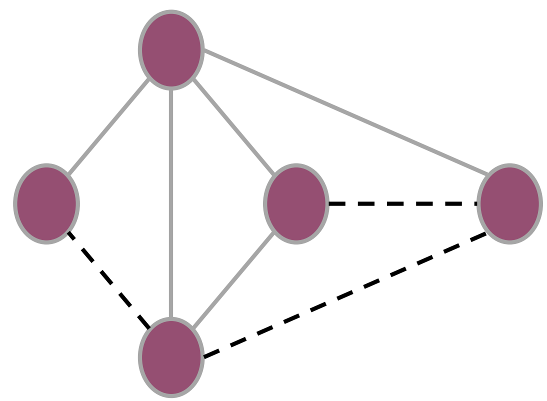
C_3 C_4 C_5

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



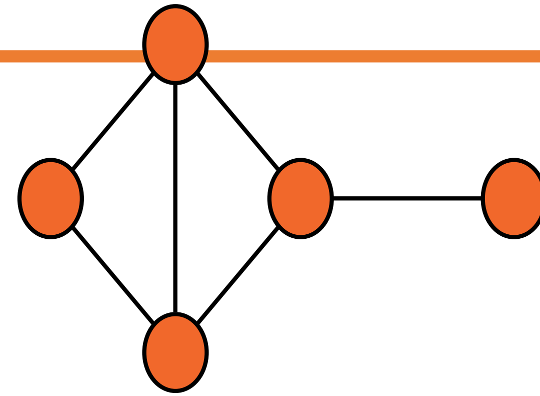
Subgraph



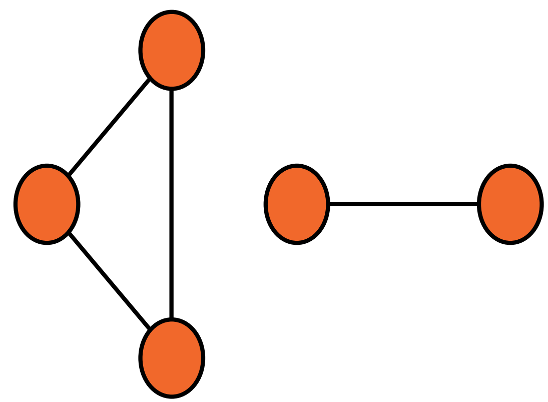
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



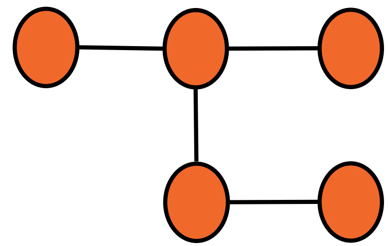
Connected graph



Non connected graph with two connected components

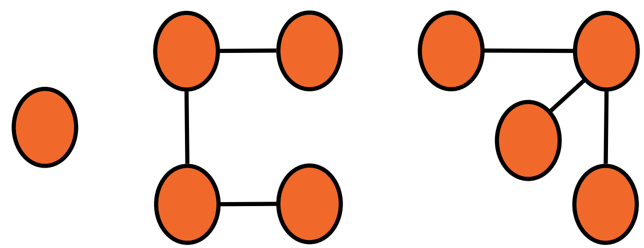
Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles



Tree

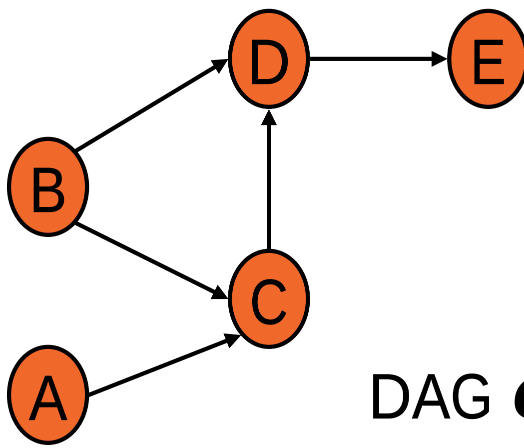
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



Forest

DAG

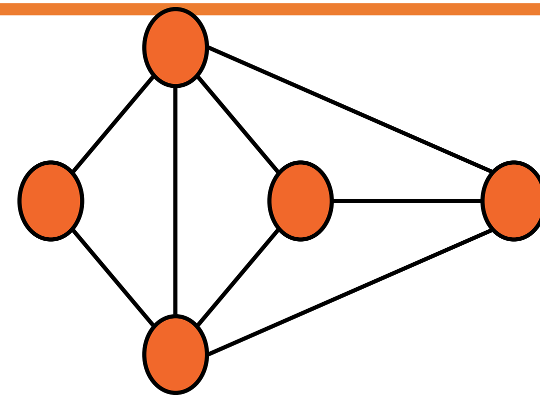
- A directed acyclic graph (DAG) is a digraph that has no directed cycles



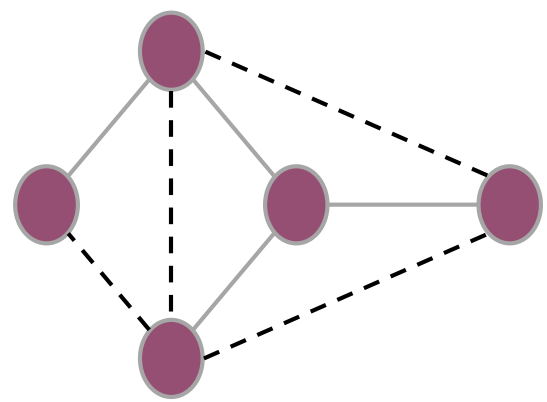
DAG G

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree



Graph



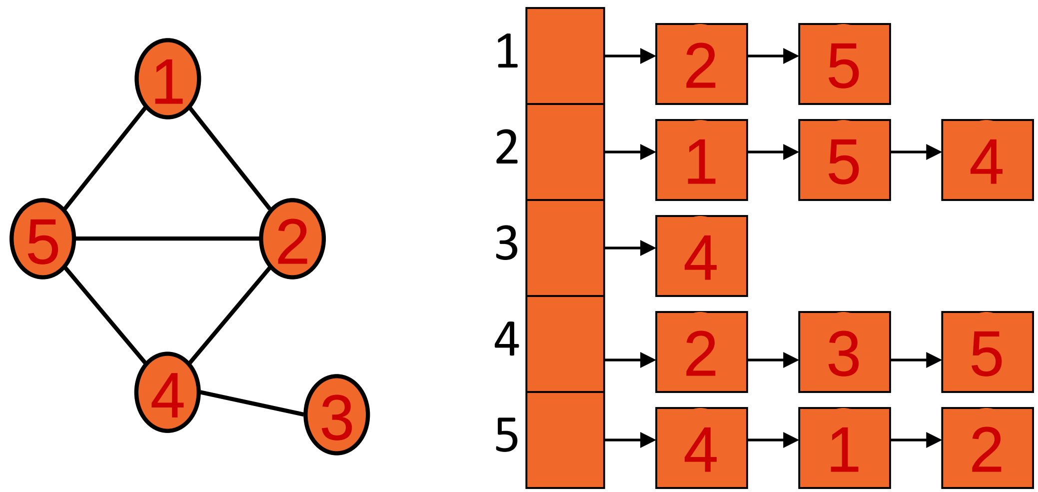
Spanning tree

Representation of Graphs

Two standard ways:

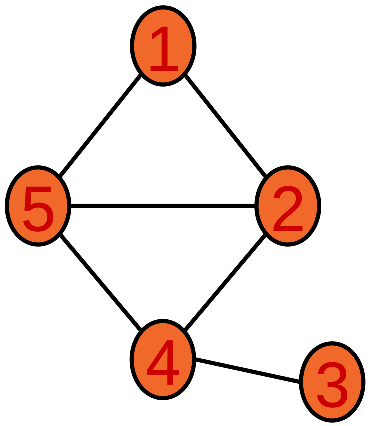
- Adjacency List
  - preferred for sparse graphs (|E| is much less than |V|^2)
  - Unless otherwise specified we will assume this representation
- Adjacency Matrix
  - Preferred for dense graphs

Adjacency List



- An array Adj of |V| lists, one per vertex
- For each vertex u in V,
  - Adj[u] contains all vertices v such that there is an edge (u,v) in E (i.e. all the vertices adjacent to u)
- Space required $\Theta(|V|+|E|)$ (Following CLRS, we will use V for |V| and E for |E|) thus $\Theta(V+E)$

Adjacency Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	0	1	1
3	0	0	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Graph Traversals

- For solving most problems on graphs
  - Need to systematically visit all the vertices and edges of a graph

- Two major traversals
  - Breadth-First Search (BFS)
  - Depth-First Search(DFS)

BFS

- Starts at some source vertex s
- Discover every vertex that is reachable from s
- Also produces a BFS tree with root s and including all reachable vertices

- Discovers vertices in increasing order of distance from s
    - Distance between v and s is the minimum number of edges on a path from s to v

- i.e. discovers vertices in a series of layers

BFS : vertex colors stored in color[]

- Initially all undiscovered: white
- When first discovered: gray
  - They represent the frontier of vertices between discovered and undiscovered
  - Frontier vertices stored in a queue
  - Visits vertices across the entire breadth of this frontier
- When processed: black

Review: Breadth-First Search

- “Explore” a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find (“discover”) its children, then their children, etc.

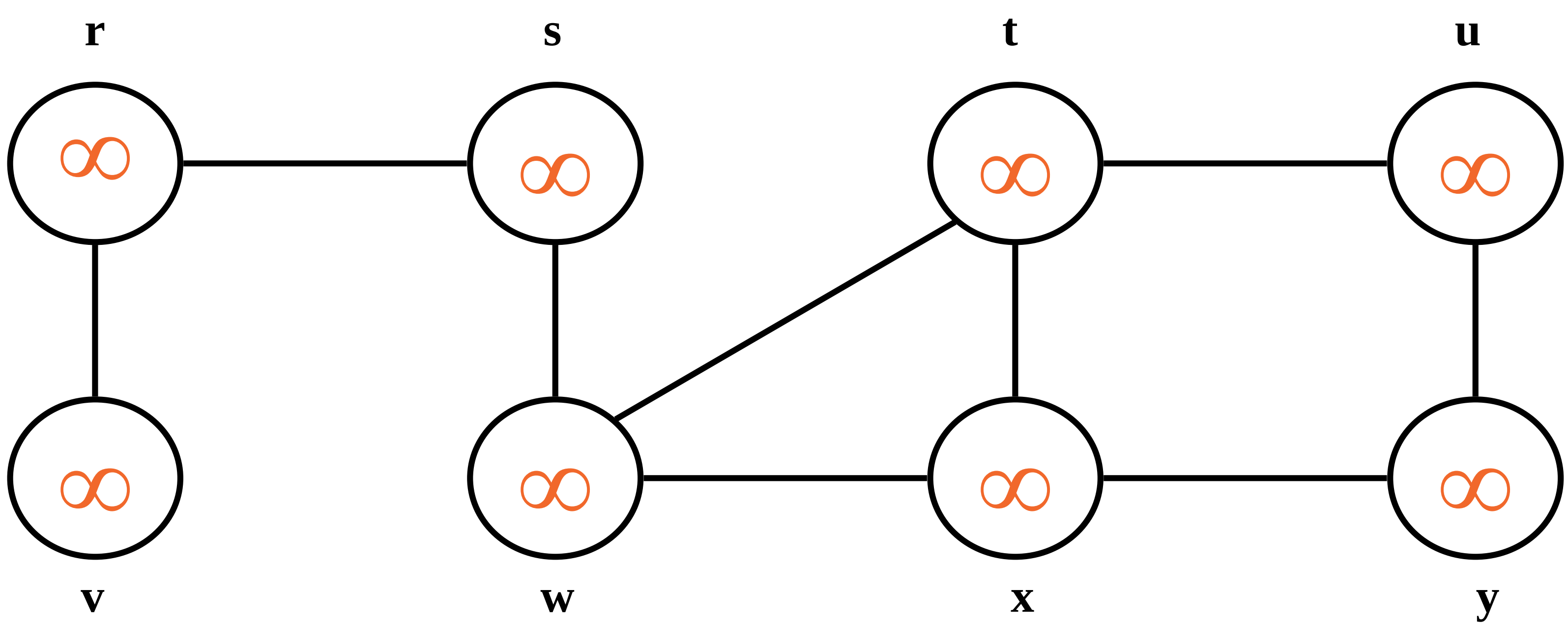
Breadth-First Search

- Again will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

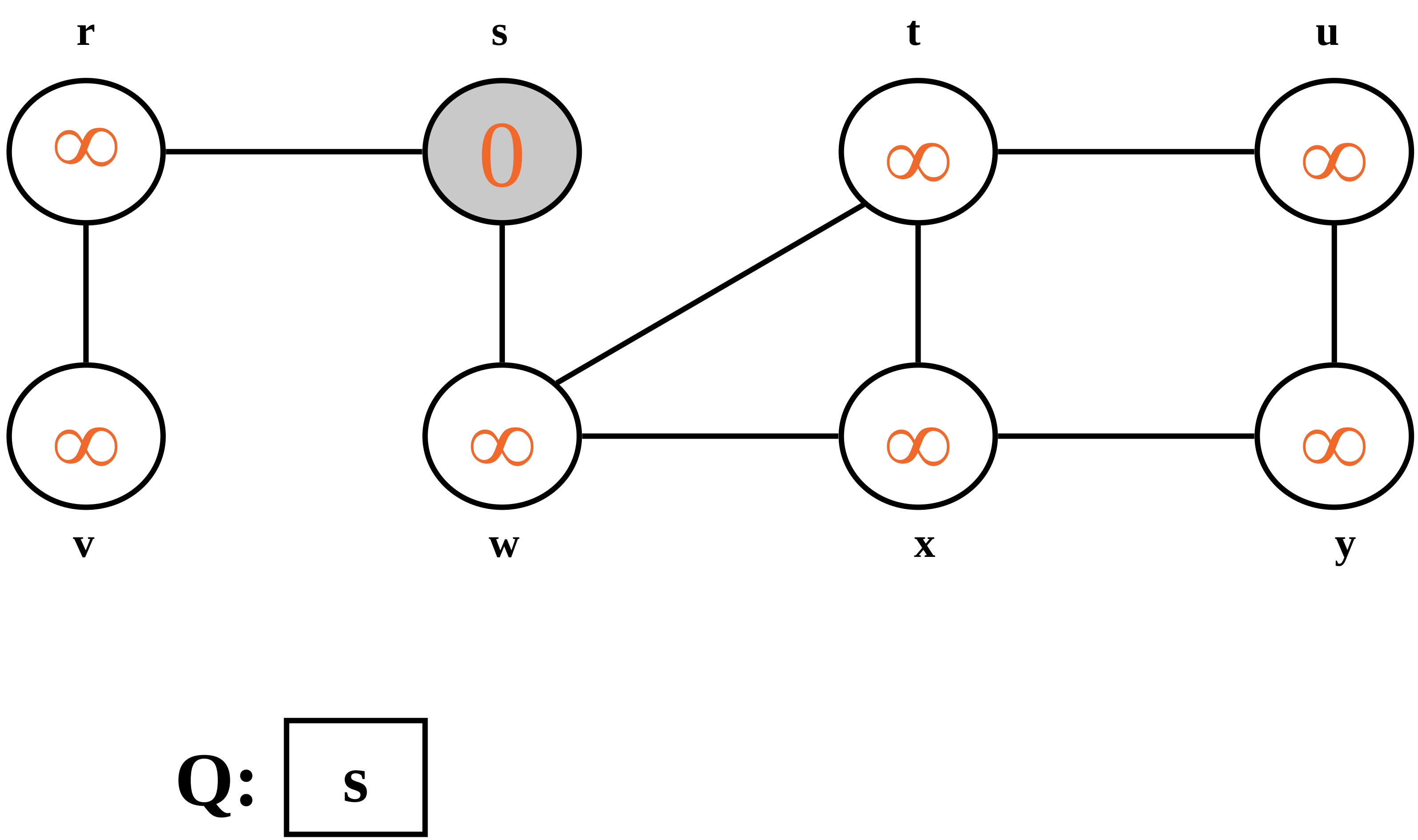
Review: Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};            // Q is a queue initialize to s
    while (Q not empty) {
        u = DEQUEUE(Q);
        for each v ∈ G.Adj[u] {
            if (v.color == WHITE)
                v.color = GREY;
                v.d = u.d + 1;
                v.p = u;
                ENQUEUE(Q, v);
        }
        u.color = BLACK;
    }
}
```

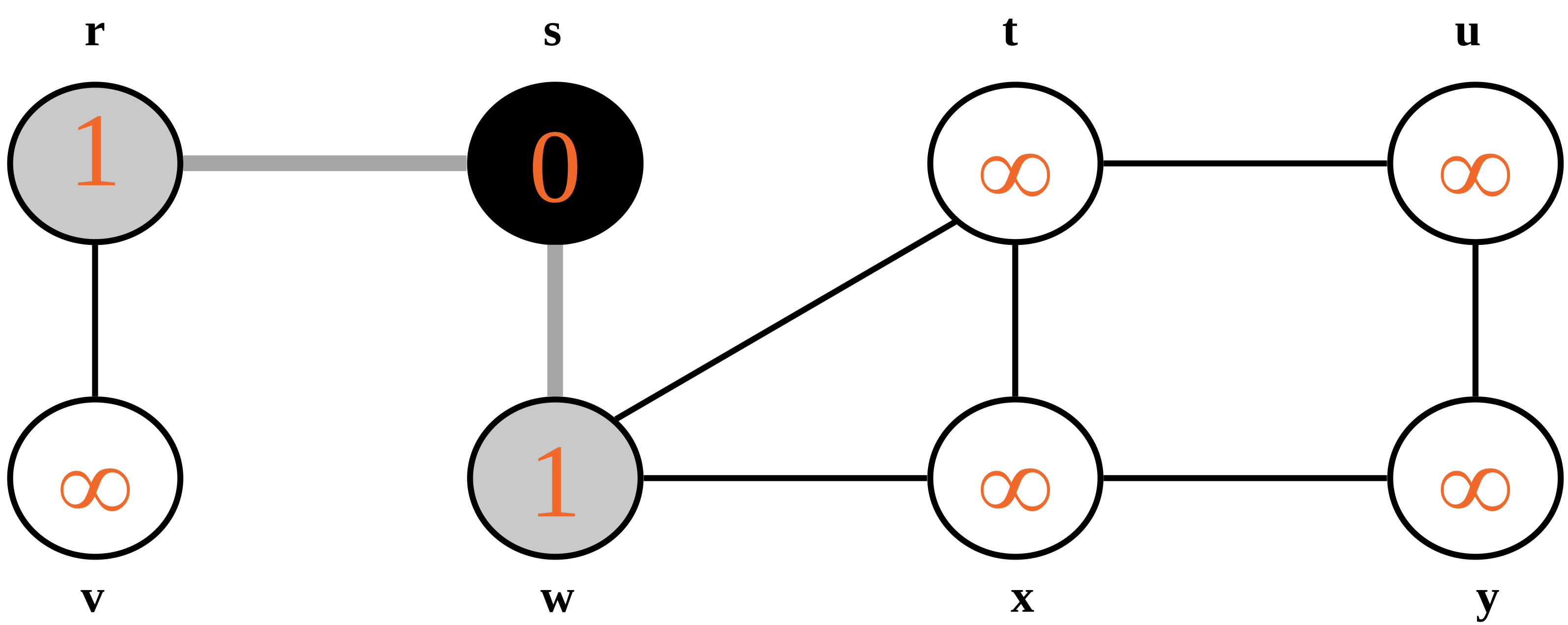
Breadth-First Search: Example



Breadth-First Search: Example

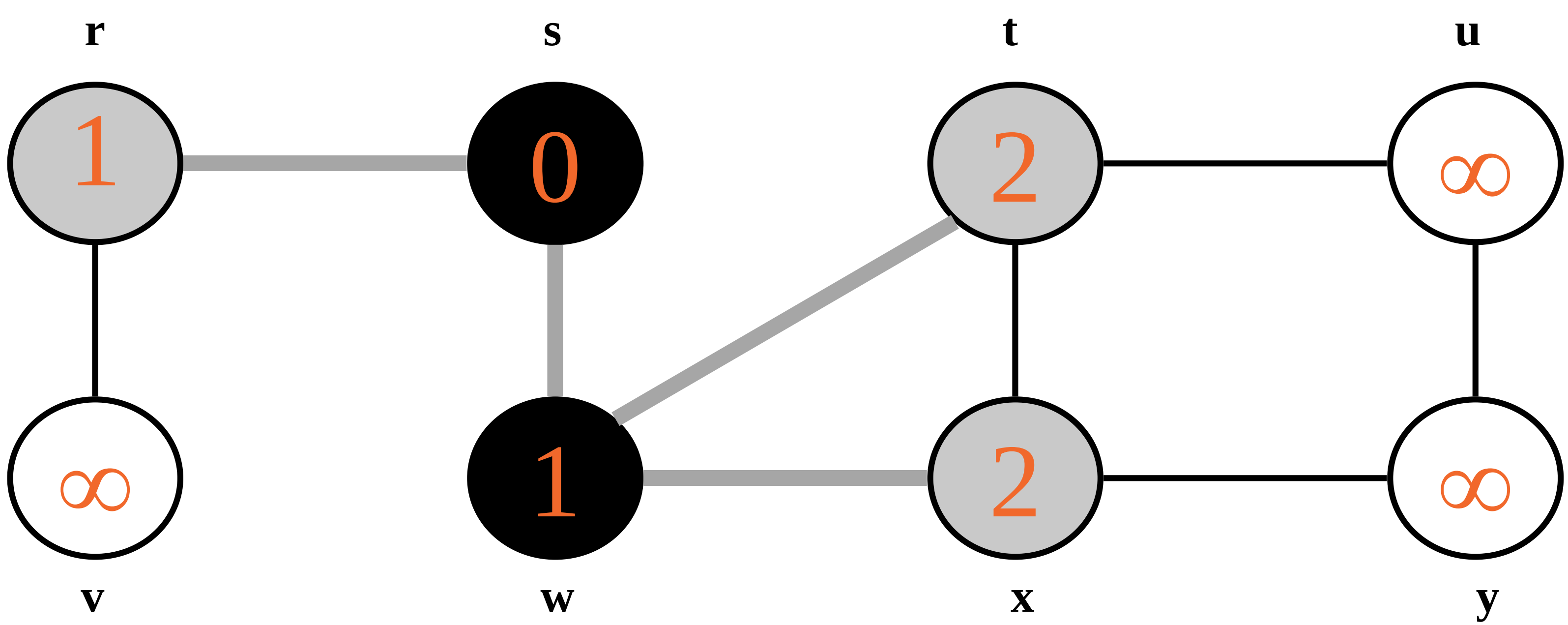


Breadth-First Search: Example



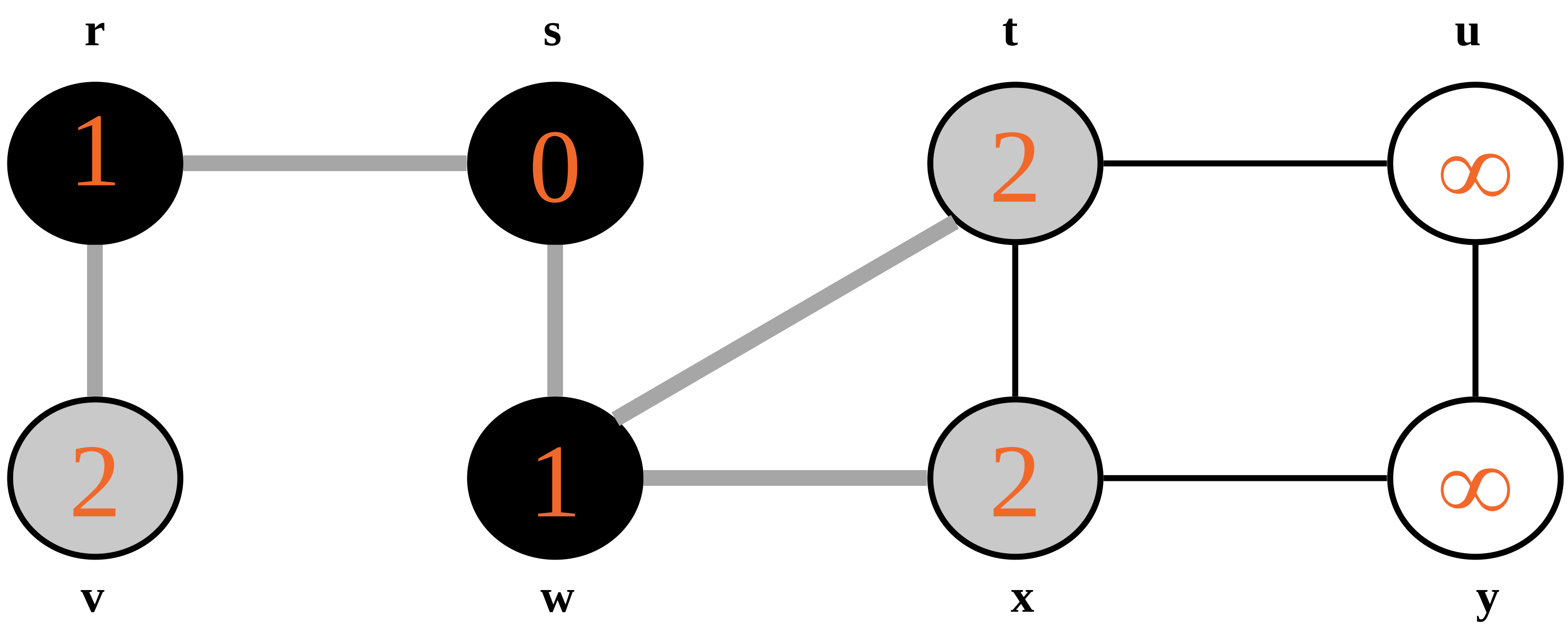
Q: | w | r |

Breadth-First Search: Example



Q: | r | t | x |

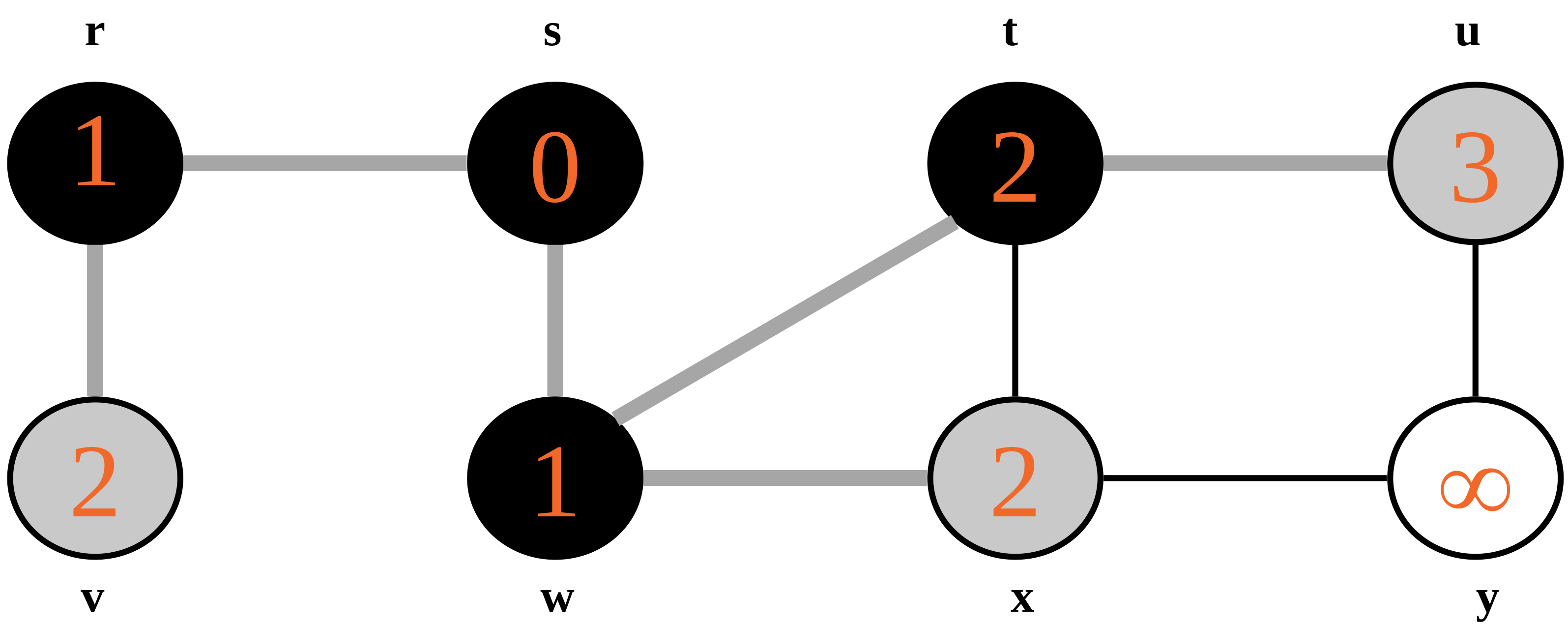
Breadth-First Search: Example



Q:

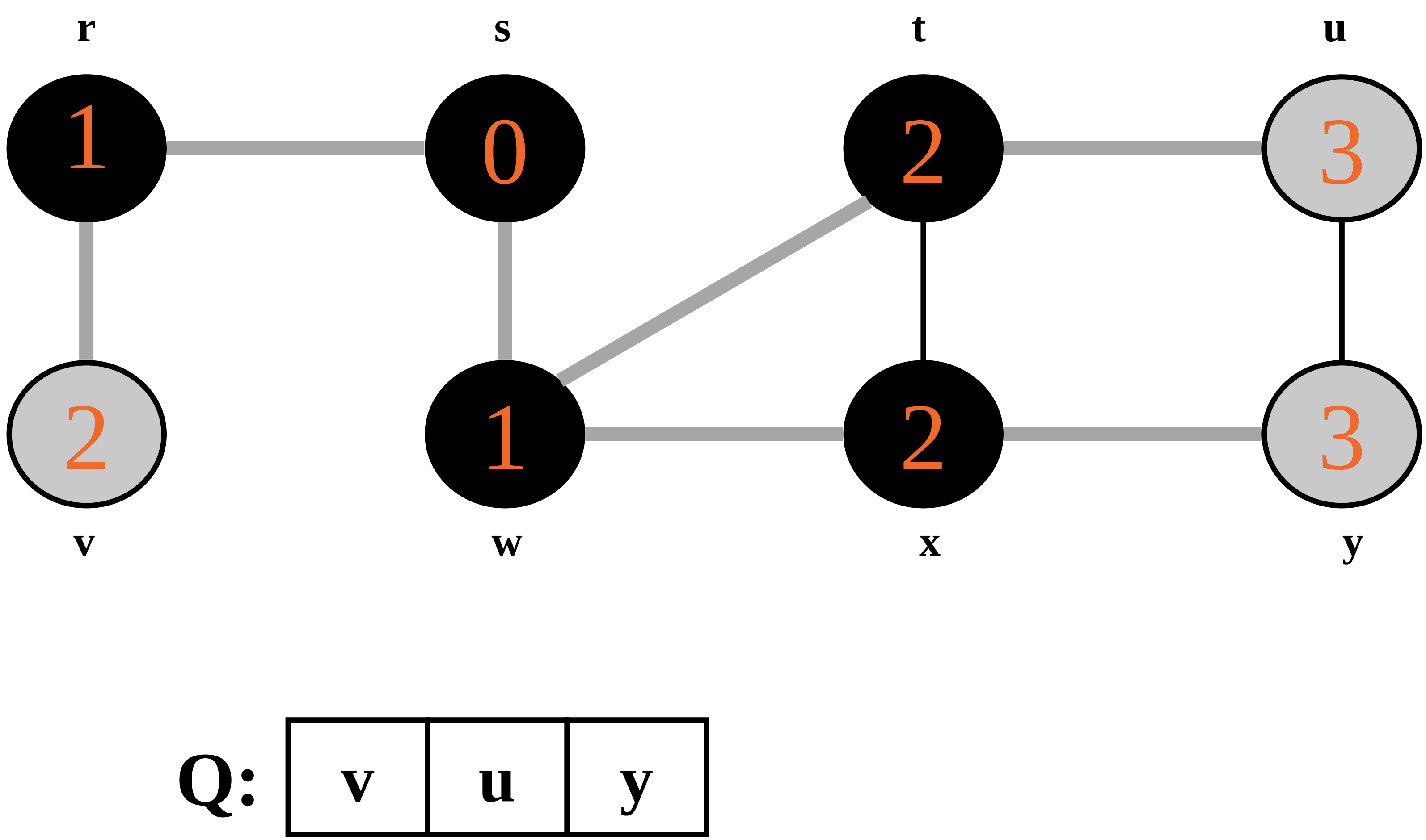
t	x	v

Breadth-First Search: Example

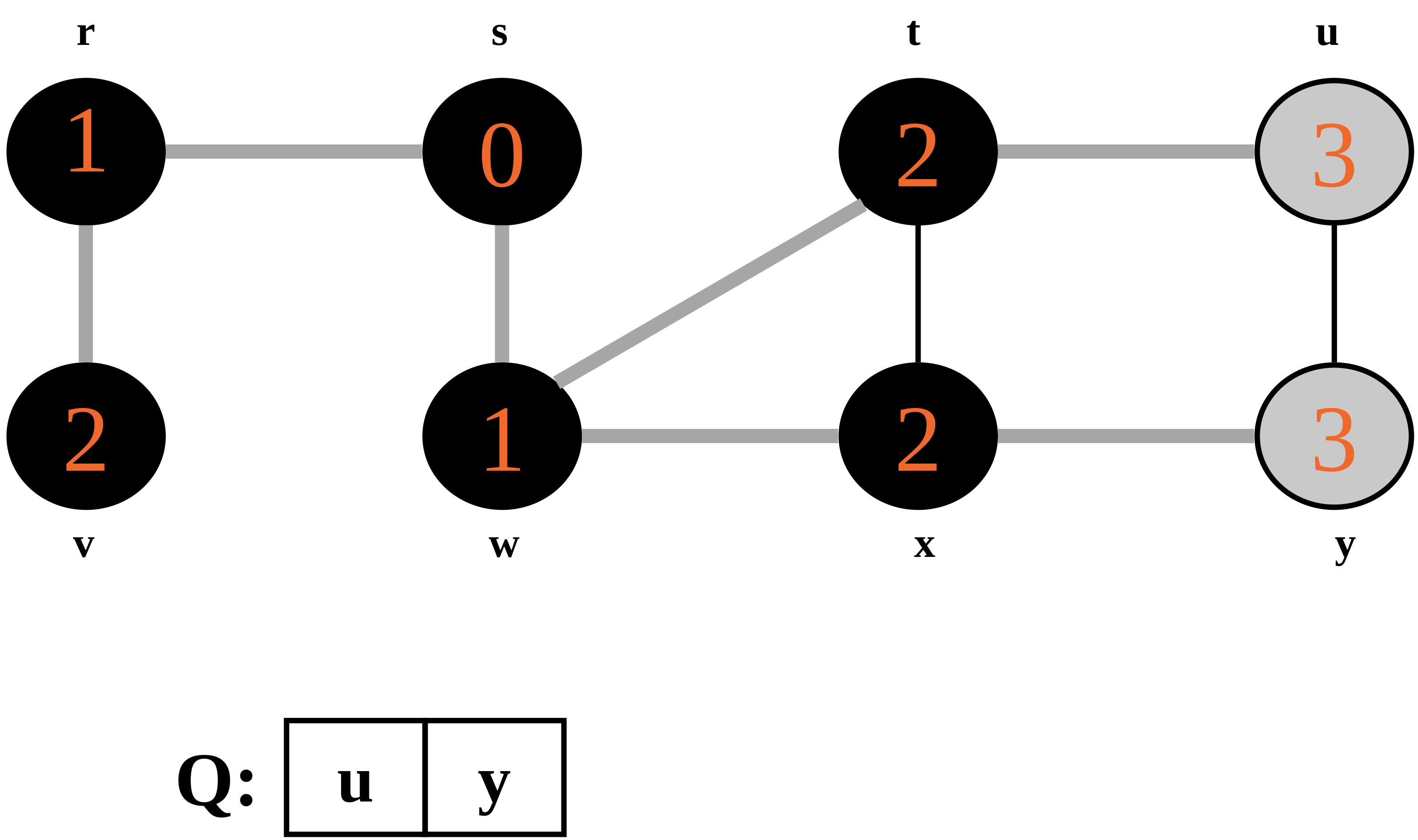


Q: | x | v | u |

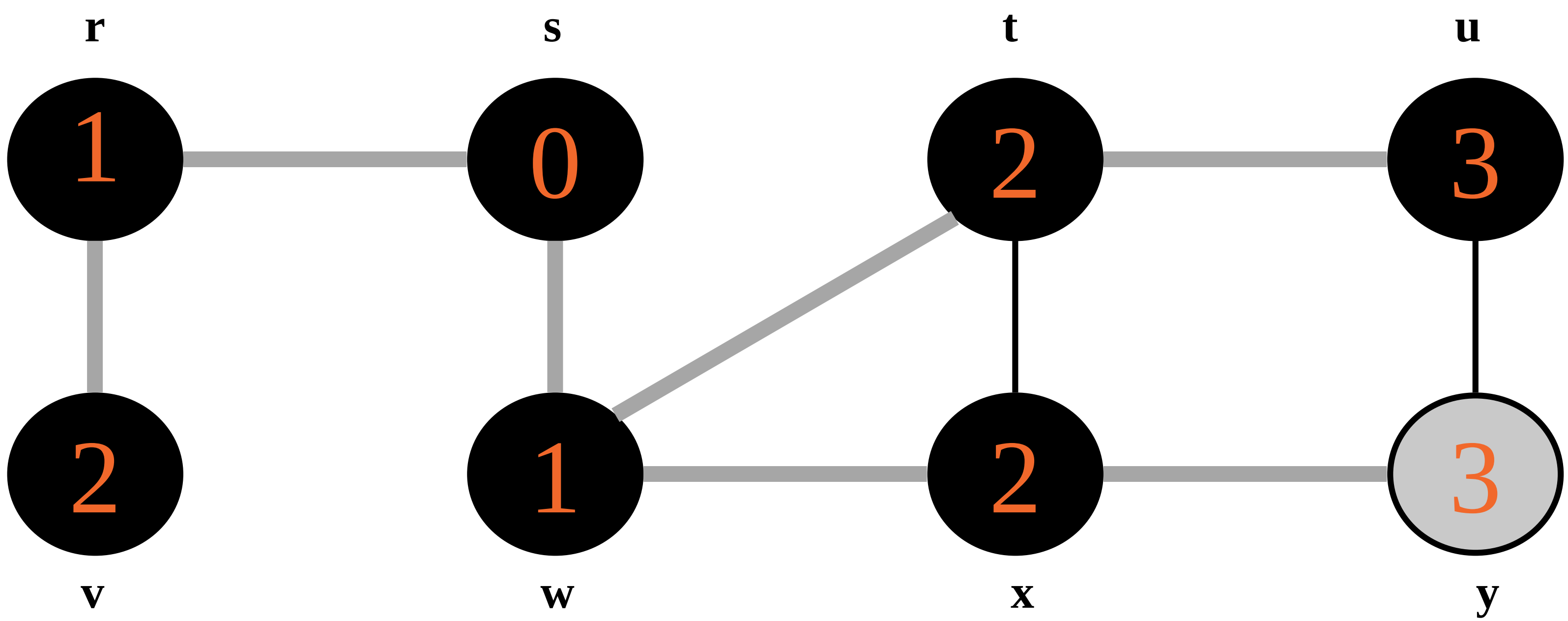
Breadth-First Search: Example



Breadth-First Search: Example

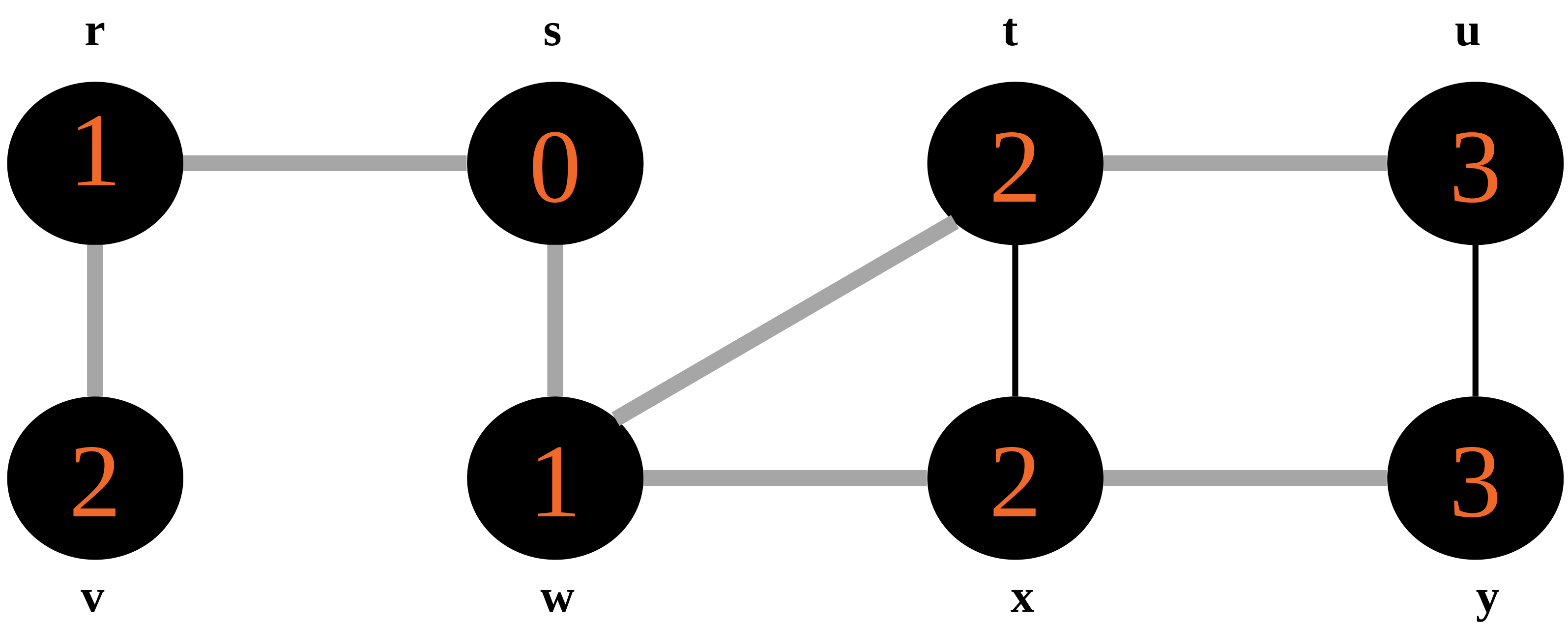


Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};              // Q is a queue initialize to s
    while (Q not empty) {
        u = DEQUEUE(Q);
        for each v ∈ G.Adj[u] {
            if (v.color == WHITE)
                v.color = GREY;
                v.d = u.d + 1;
                v.p = u;
                ENQUEUE(Q, v);
        }
        u.color = BLACK;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Analysis

- Each vertex is enqueued once and dequeued once : O(V)

- Each adjacency list is traversed once:

$$\sum_{u \in V} \deg(u) = O(E)$$

- Total: O(V+E)

BFS and shortest paths

Theorem: Let G=(V,E) be a directed or undirected graph, and suppose BFS is run on G starting from vertex s. During its execution BFS discovers every vertex v in V that is reachable from s. Let δ(s,v) denote the number of edges on the shortest path form s to v. Upon termination of BFS, d[v] = δ(s,v) for all v in V.

Depth-First Search

Depth-first search is another strategy for exploring a graph

- Explore "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges
- When all of *v*'s edges have been explored, backtrack to the vertex from which *v* was discovered
- recursive

Time stamps, color[u] and pred[u] as before

We store two time stamps:

- d[u] or u.d: the time vertex u is first discovered (discovery time)
- f[u] or u.f: the time we finish processing vertex u (finish time)

color[u] or u.color

- Undiscovered: white
- Discovered but not finished processing: gray
- Finished: black

pred[u] or u.π

- Pointer to the vertex that first discovered u

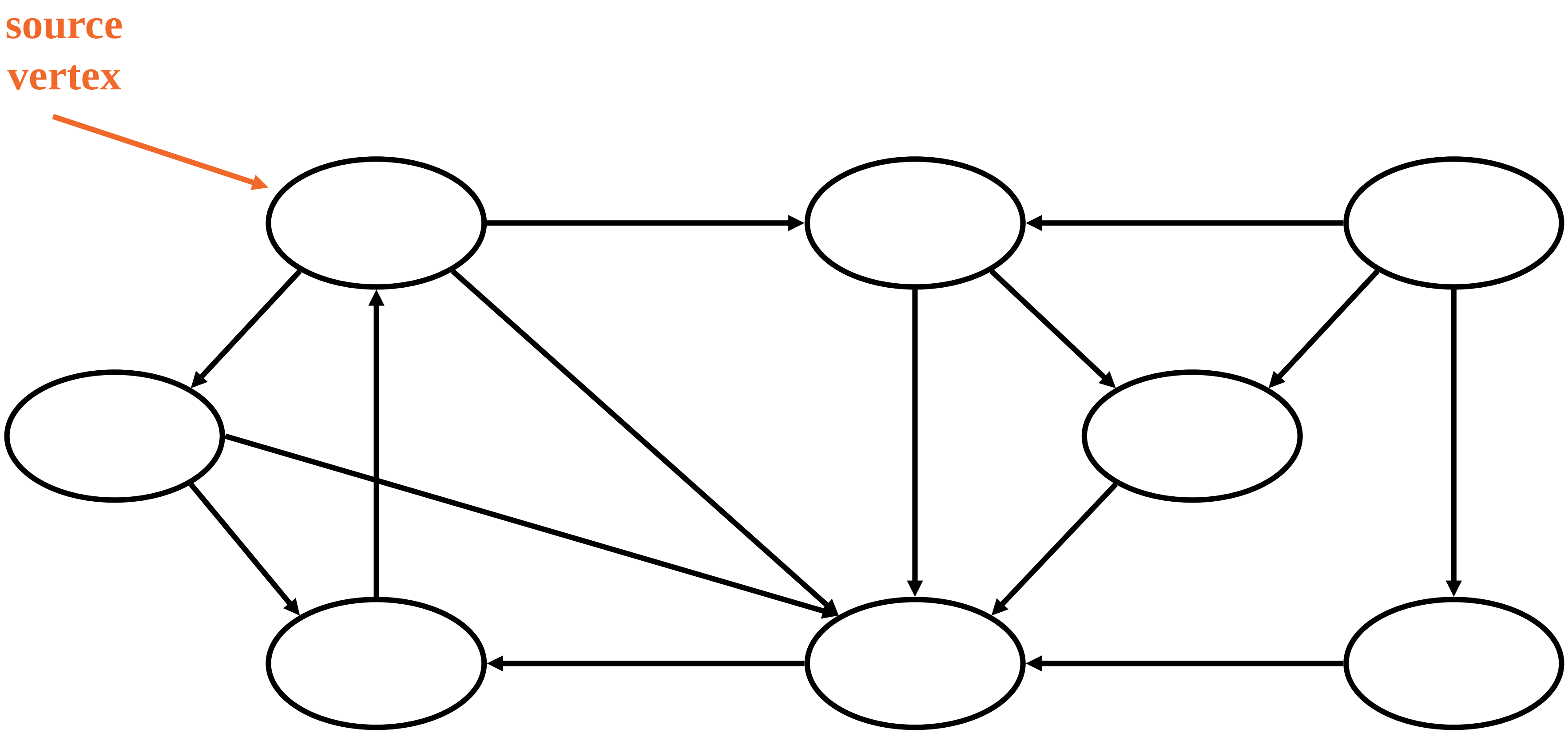
Depth-First Search: The Code

```
DFS(G)
{
   for each vertex u ∈ G.V
   {
      u.color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G.V
   {
      if (u.color == WHITE)
         DFS_Visit(G,u);
   }
}
```

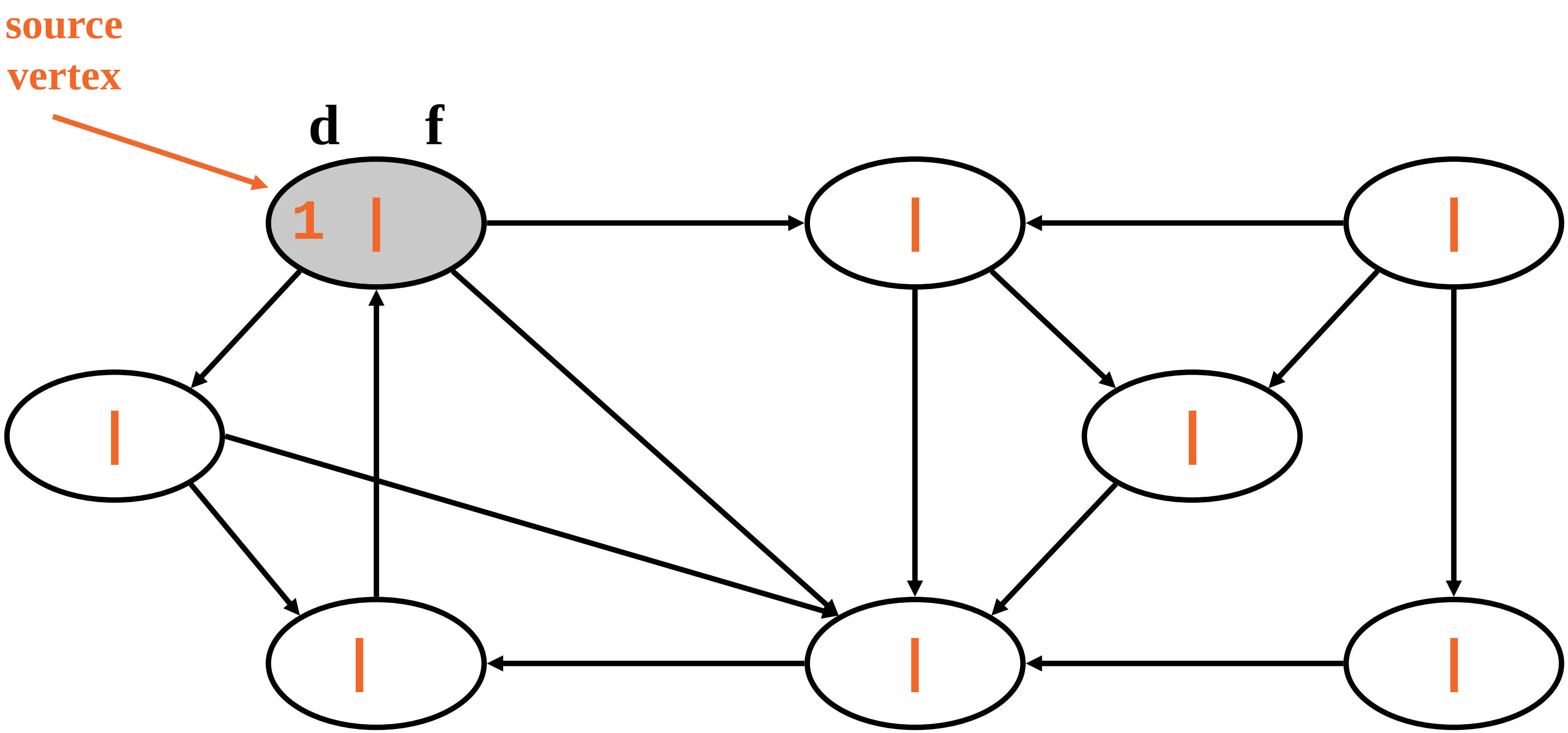
```
DFS_Visit(G, u)
{
   u.color = GREY;
   time = time+1;
   u.d = time;
   for each v ∈ G.Adj[u]
   {
      if (v.color == WHITE)
         DFS_Visit(G,v);
   }
   u.color = BLACK;
   time = time+1;
   u.f = time;
}
```

Running time: $\Theta(V+E) = \Theta(V^2)$ because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times

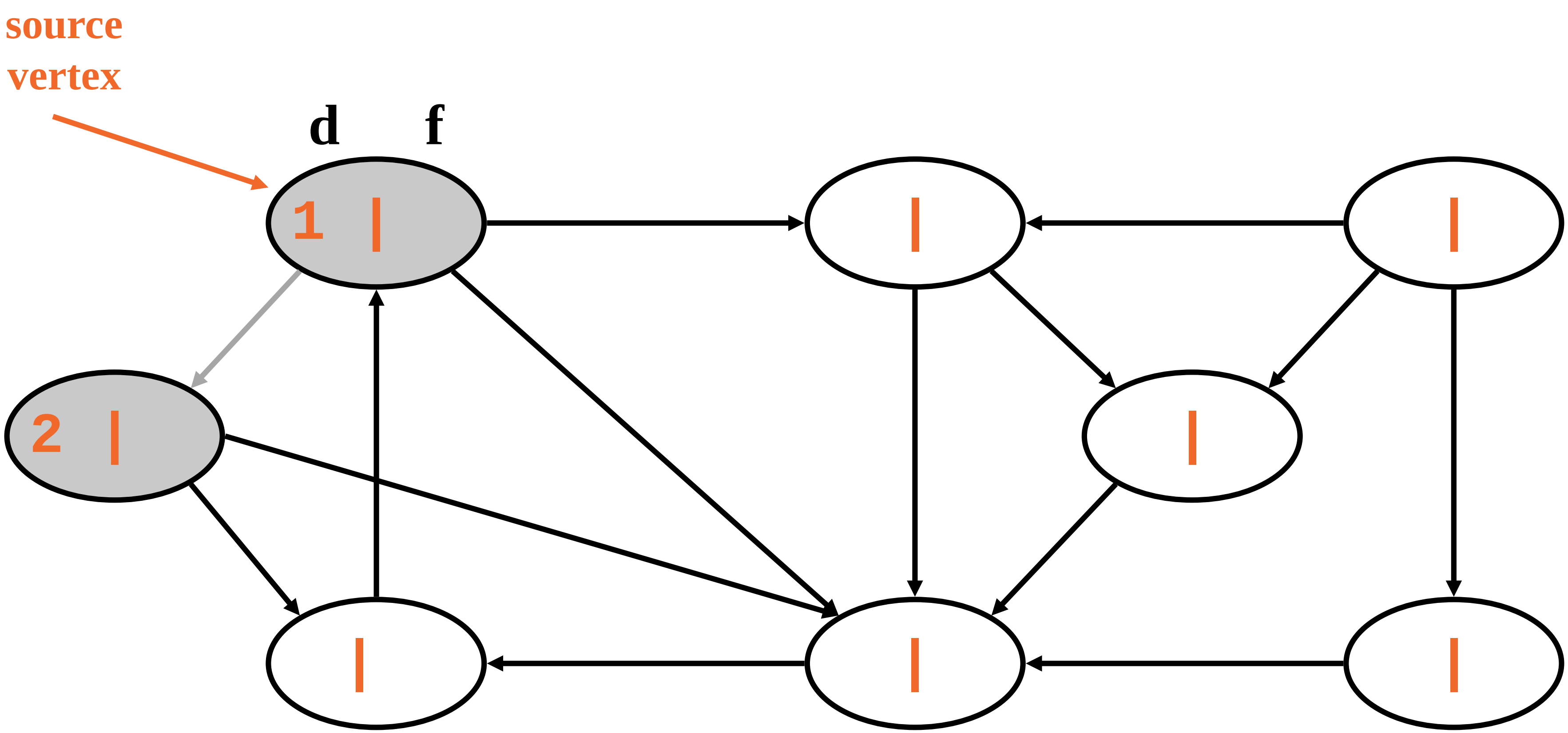
DFS Example



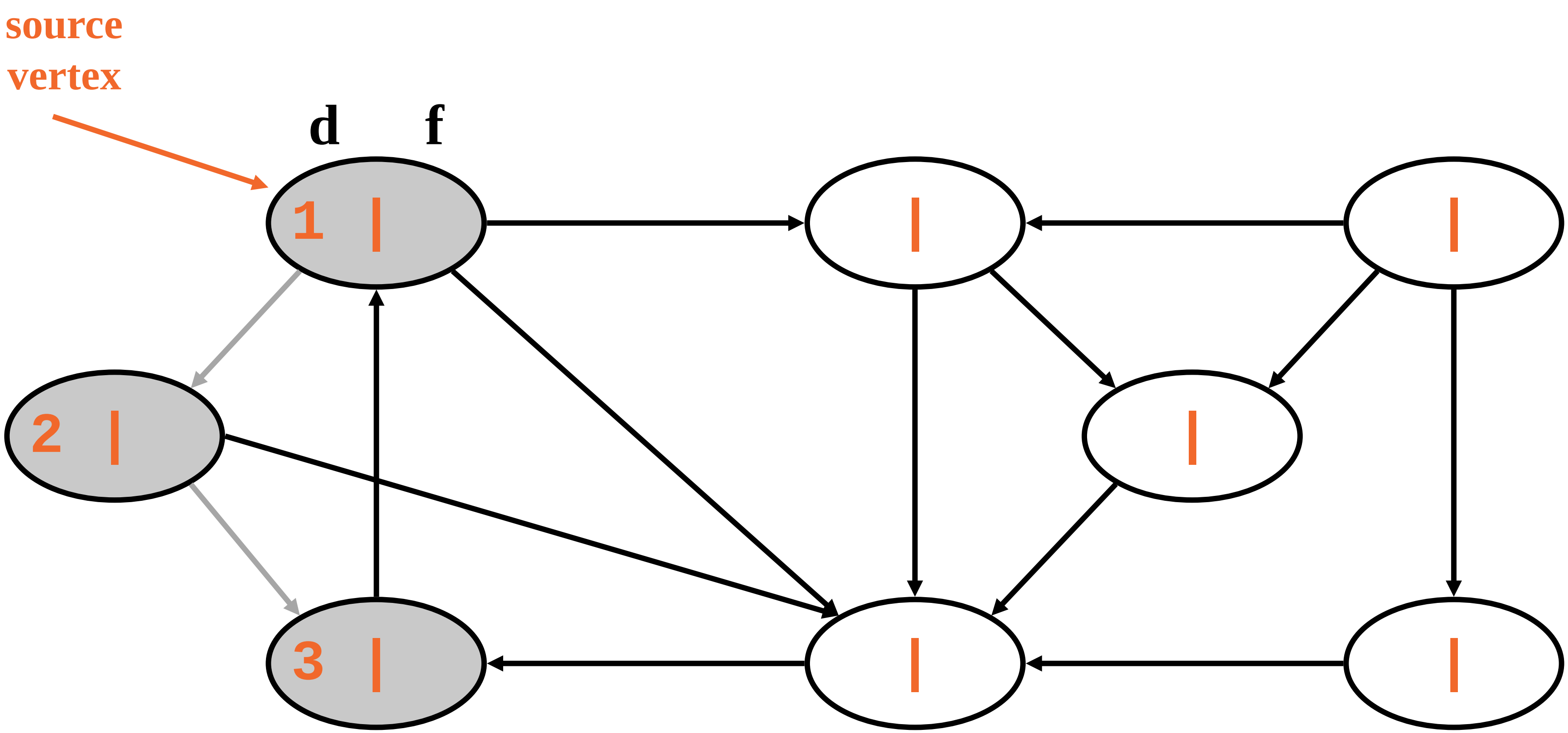
DFS Example



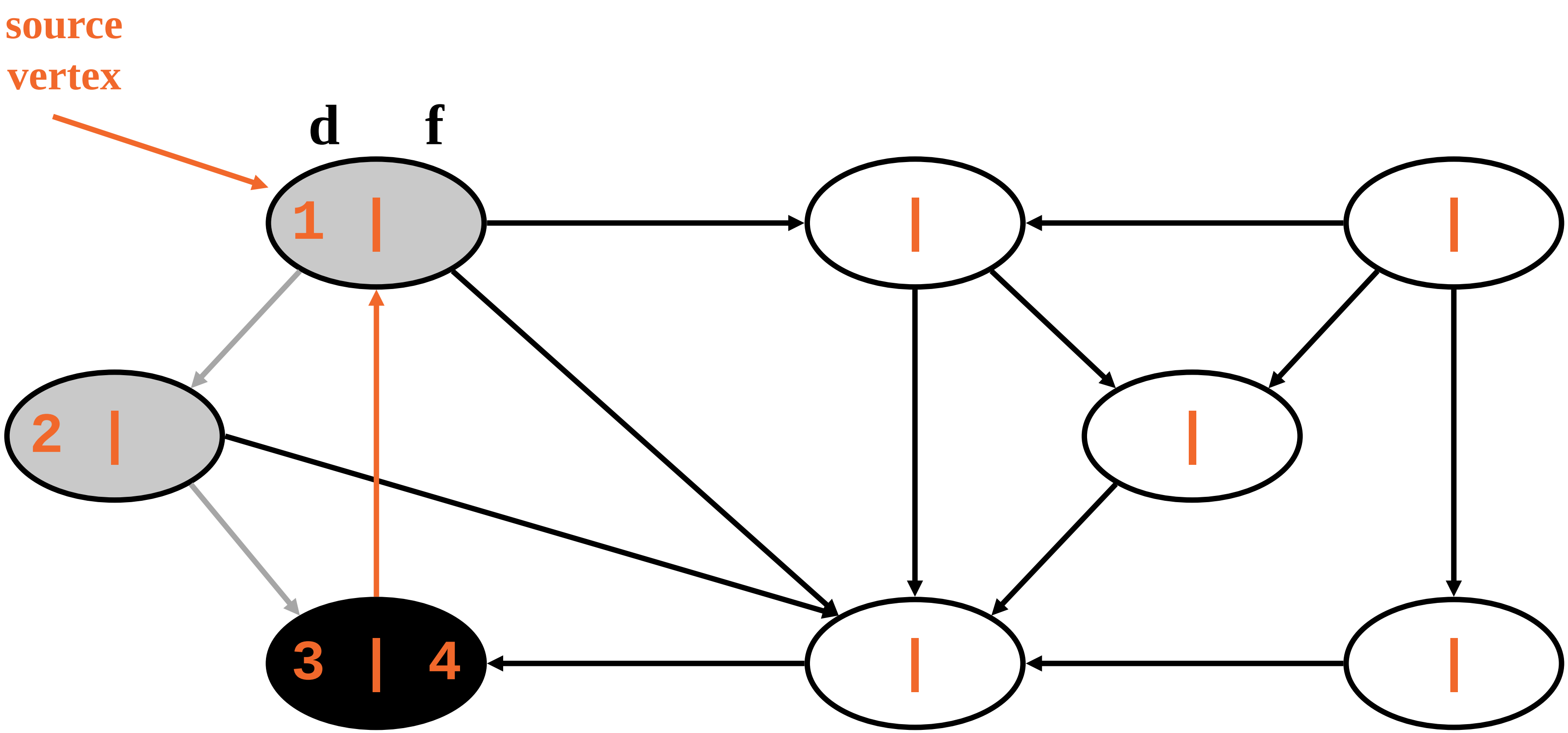
DFS Example



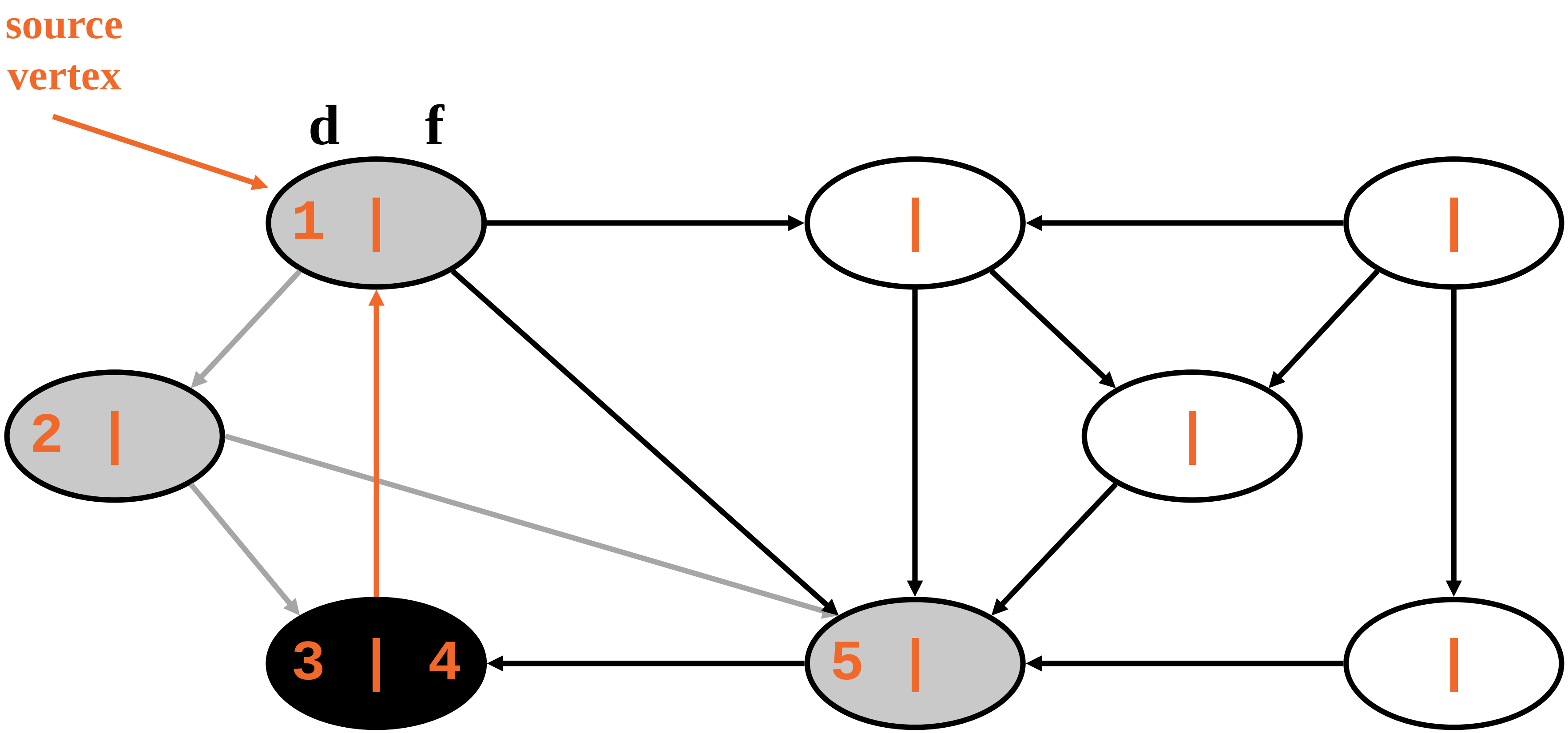
DFS Example



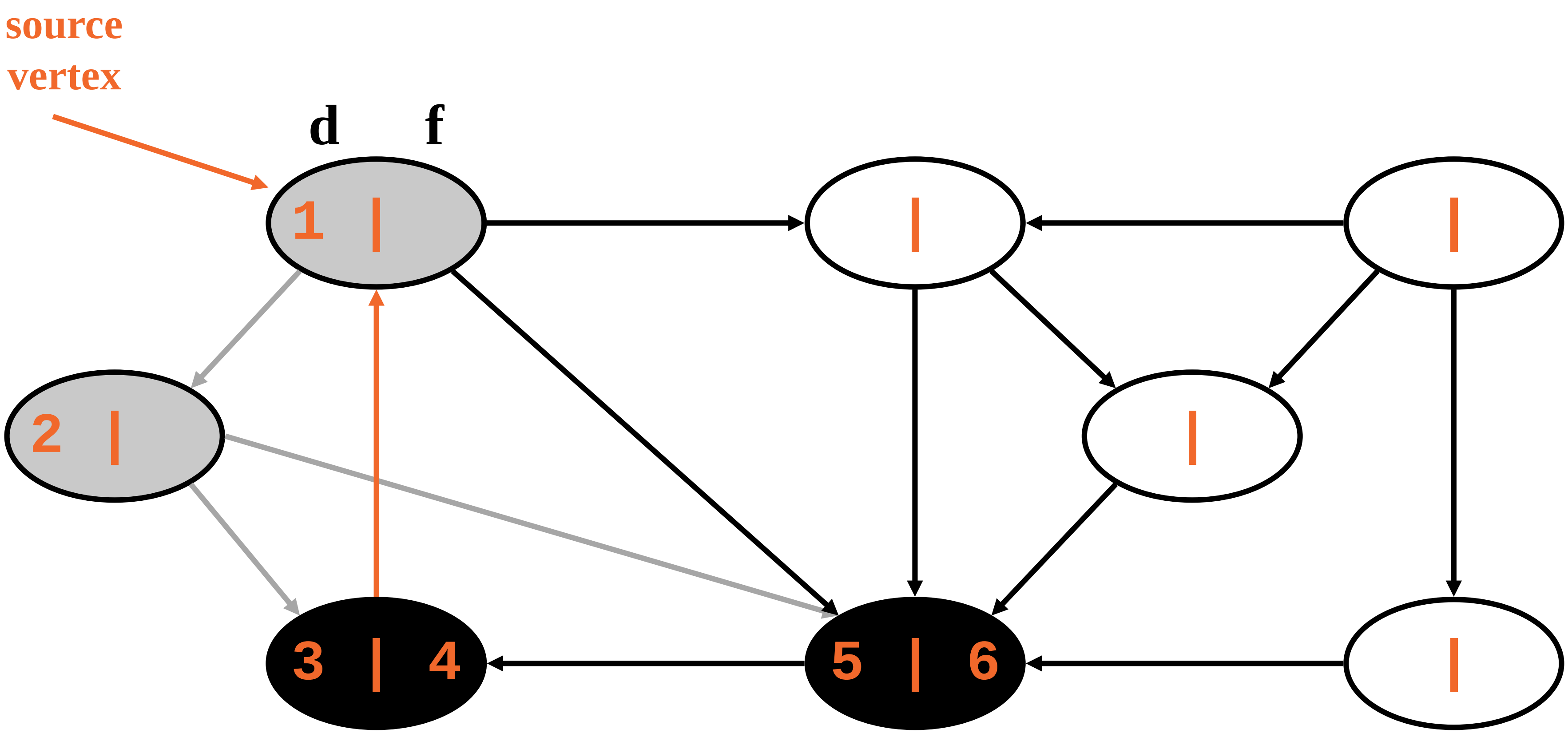
DFS Example



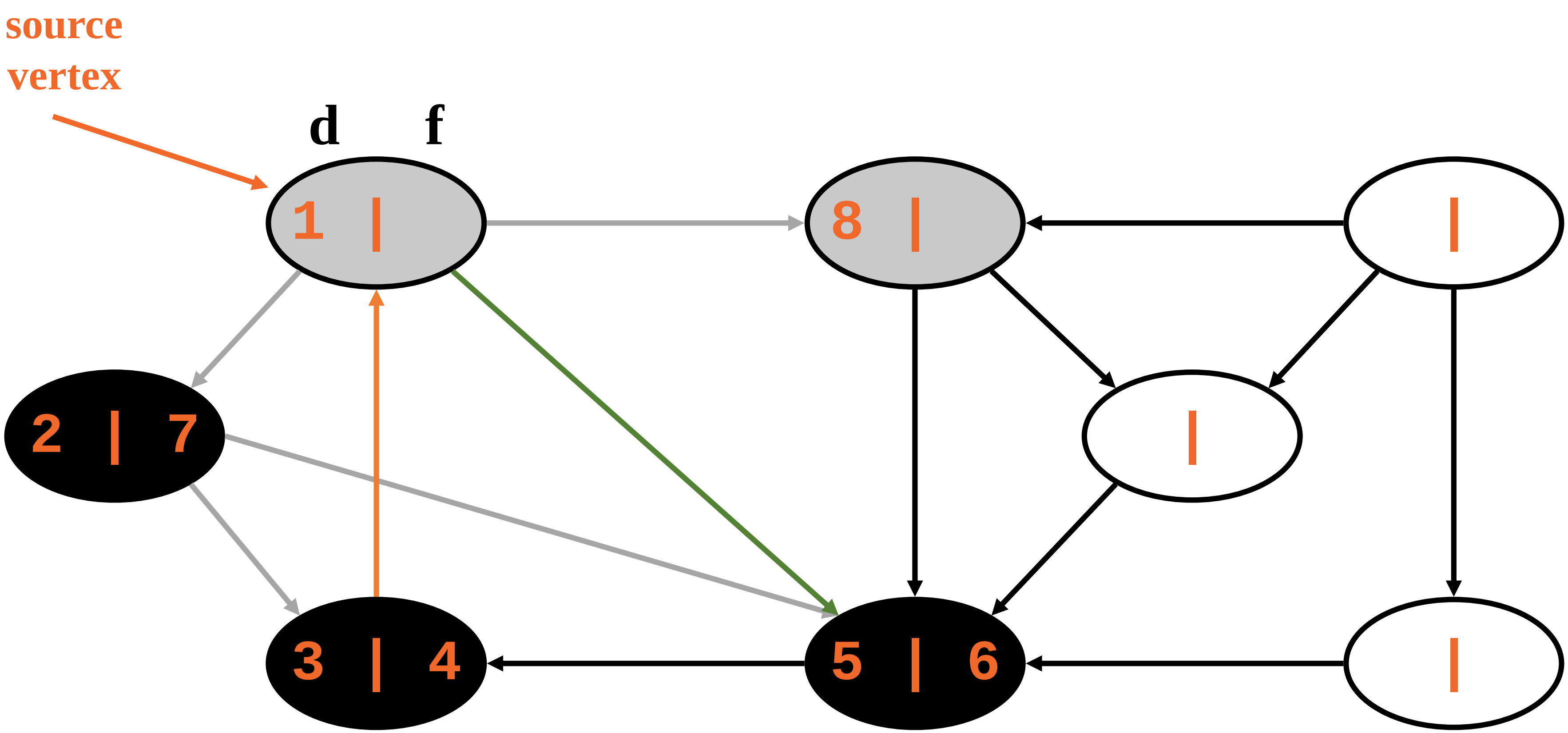
DFS Example



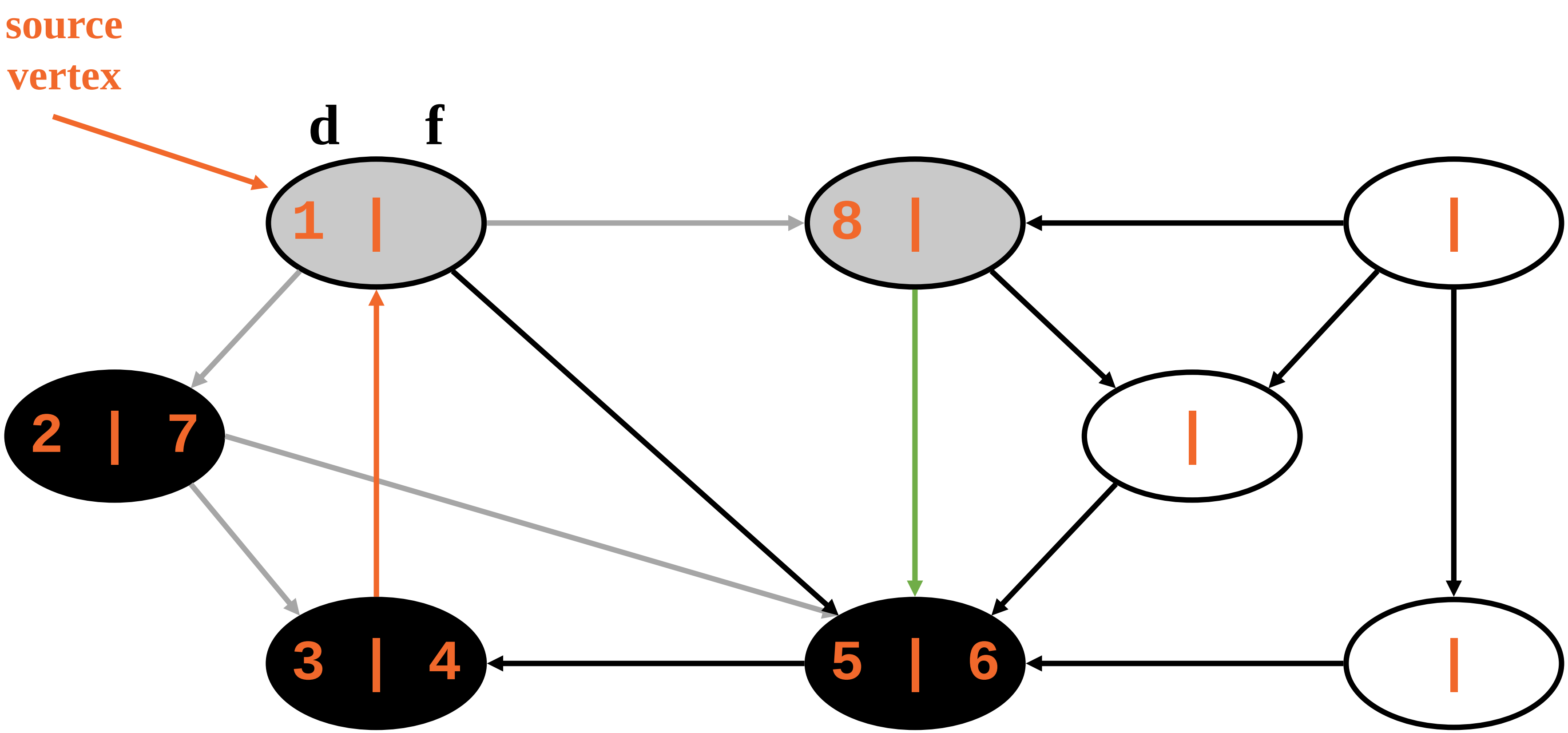
DFS Example



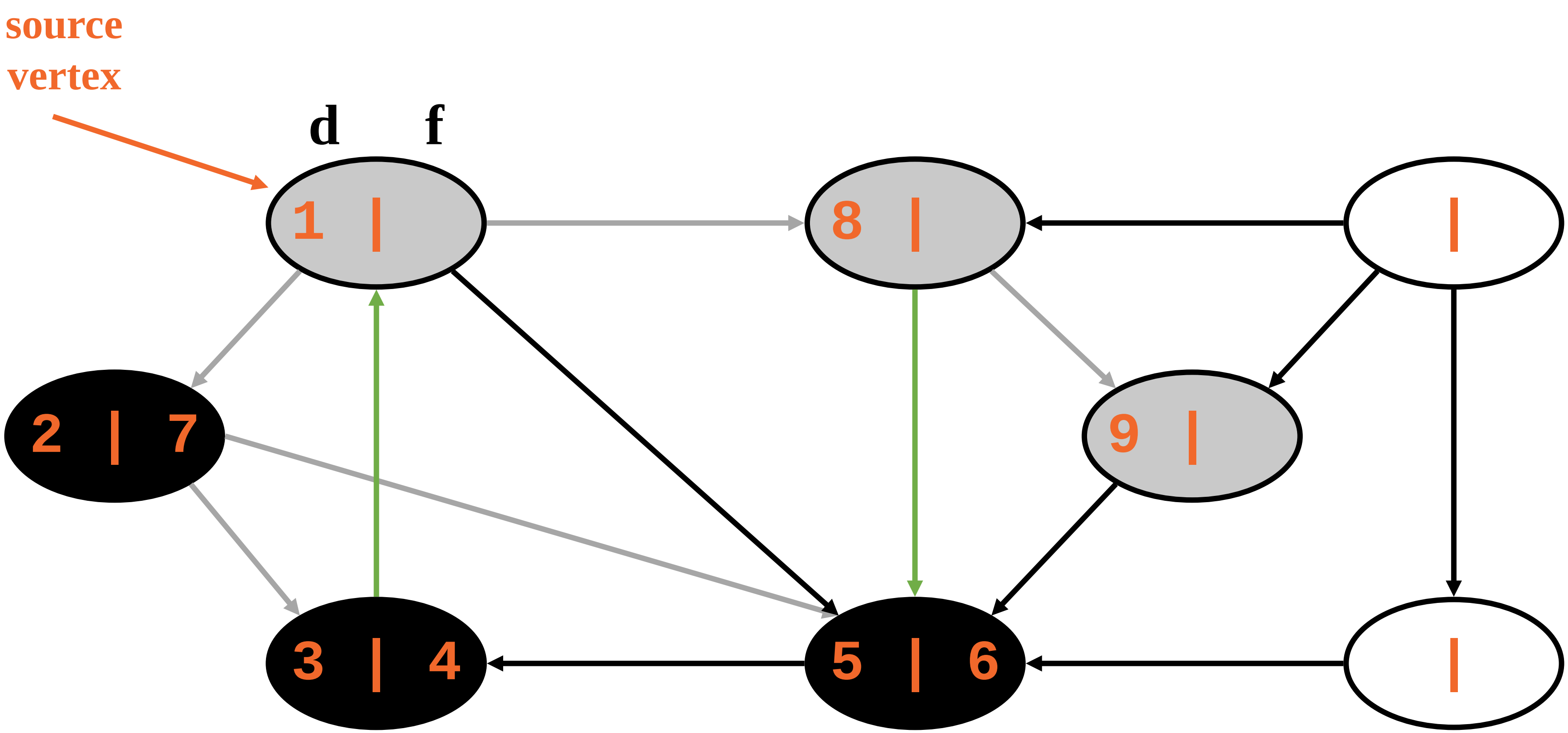
DFS Example



DFS Example

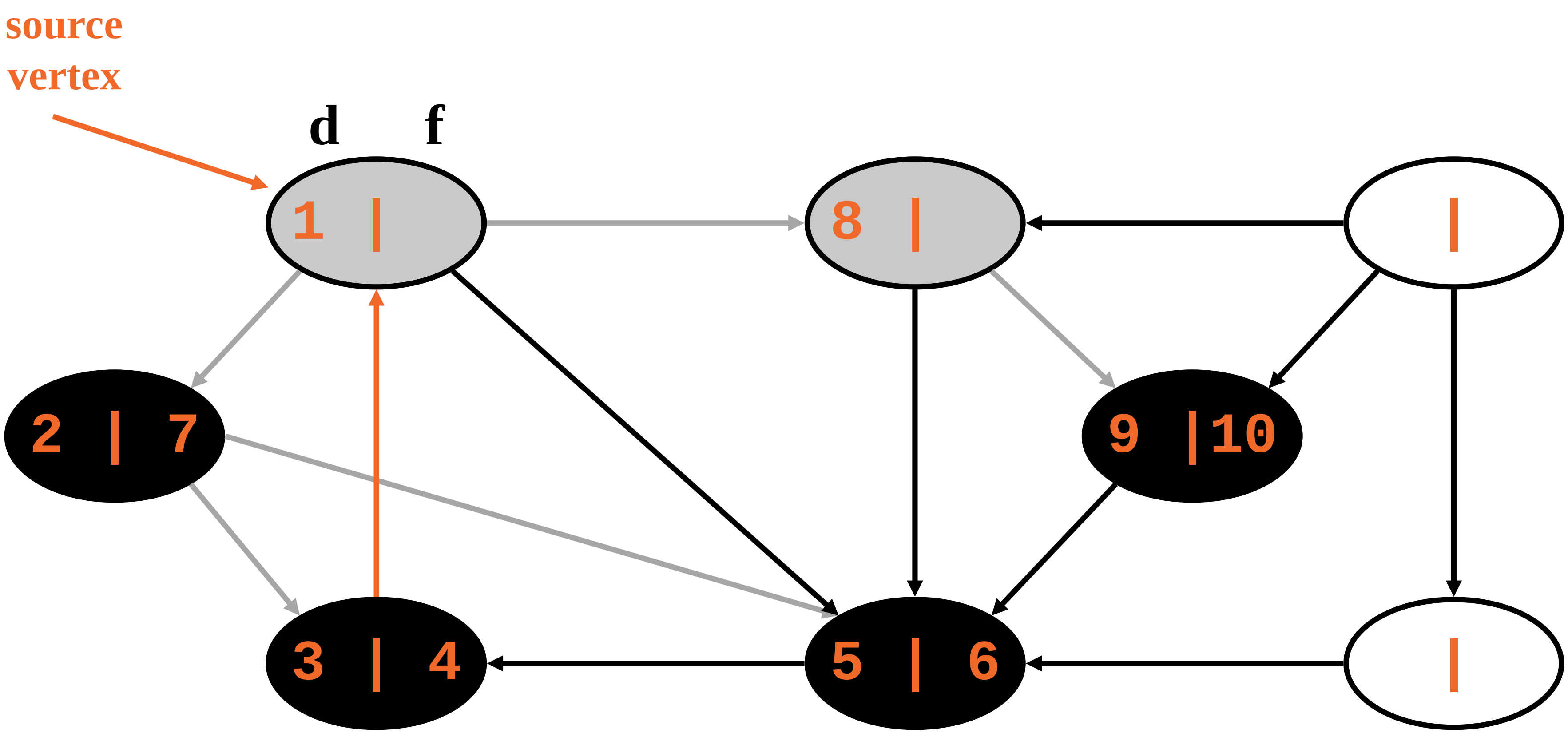


DFS Example

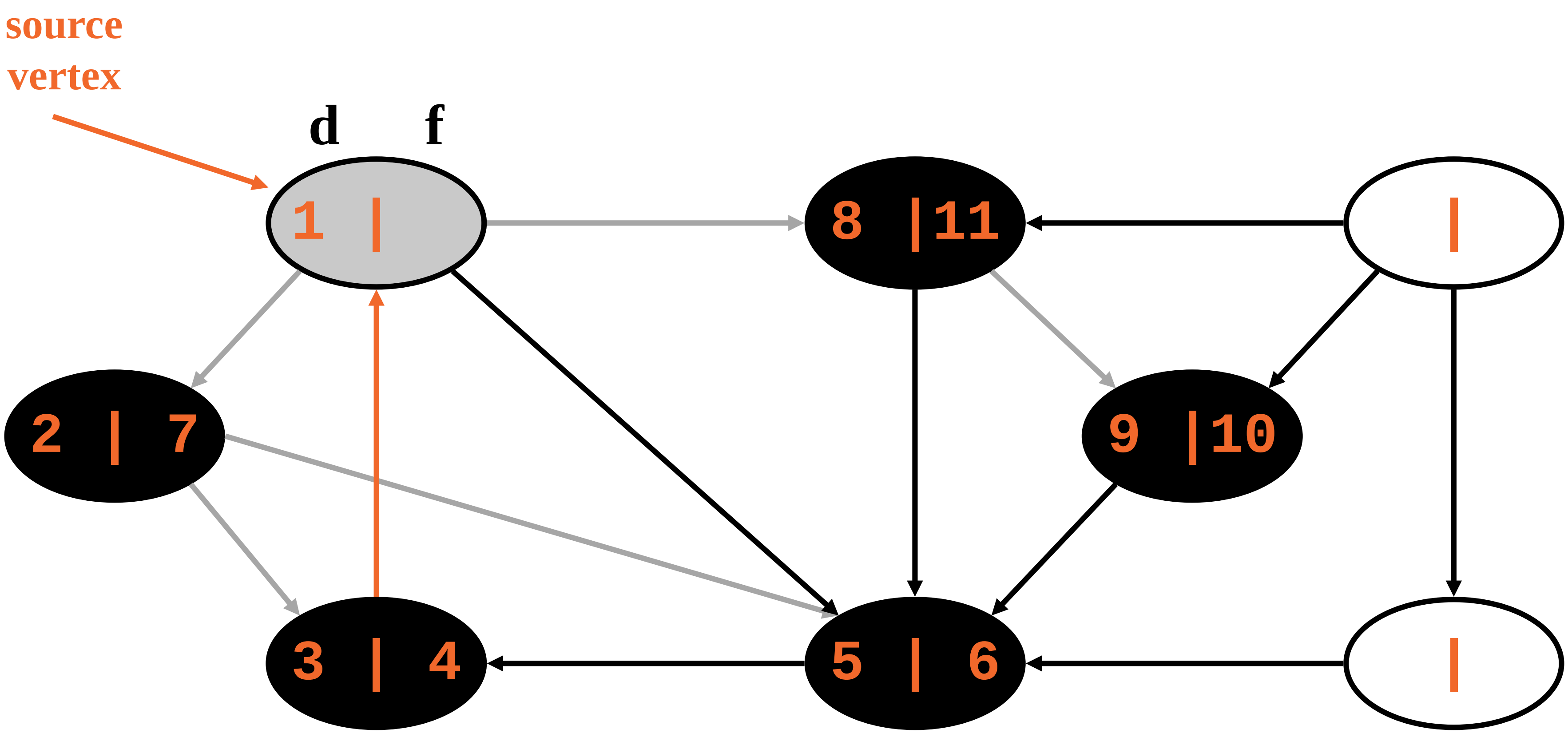


What is the structure of the grey vertices?
What do they represent?

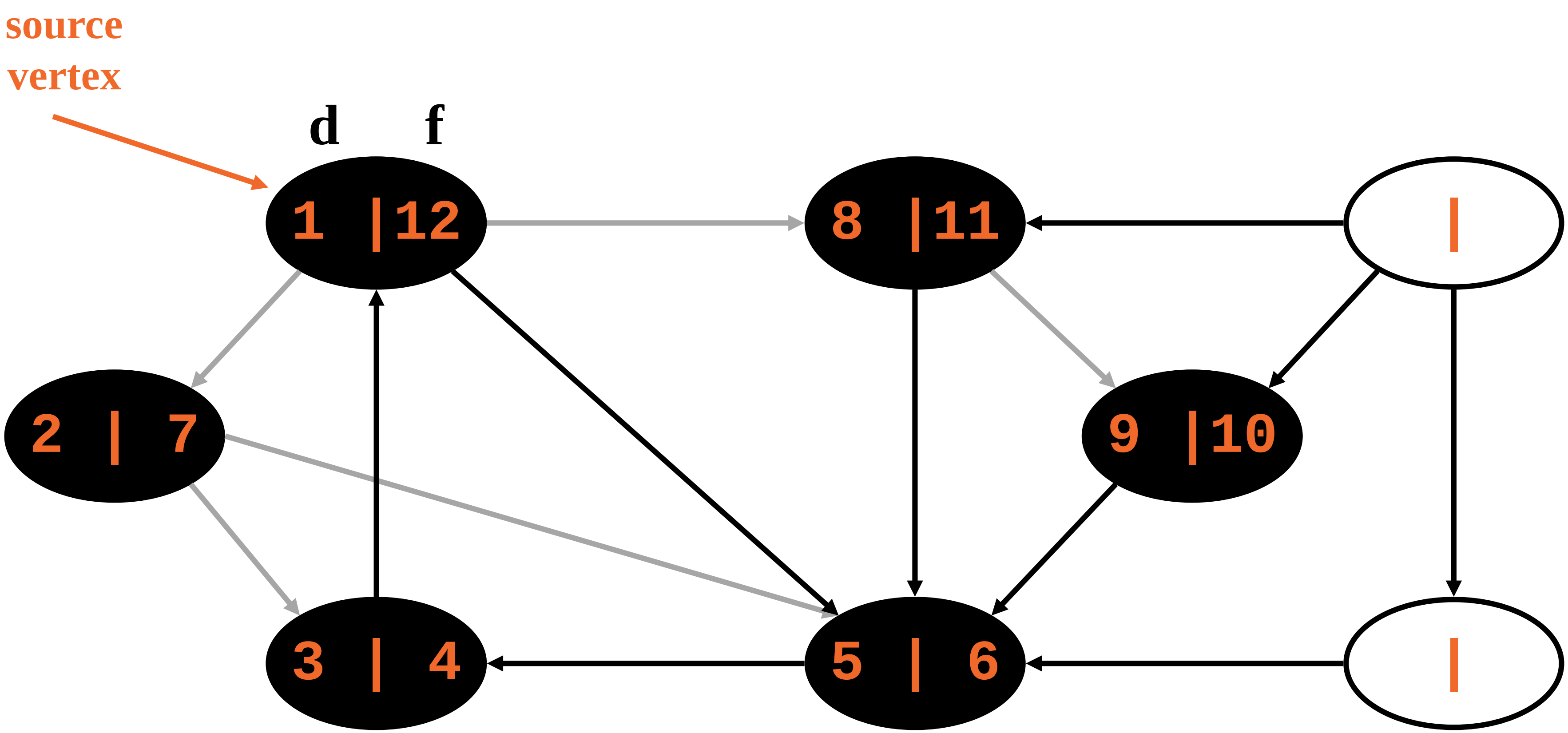
DFS Example



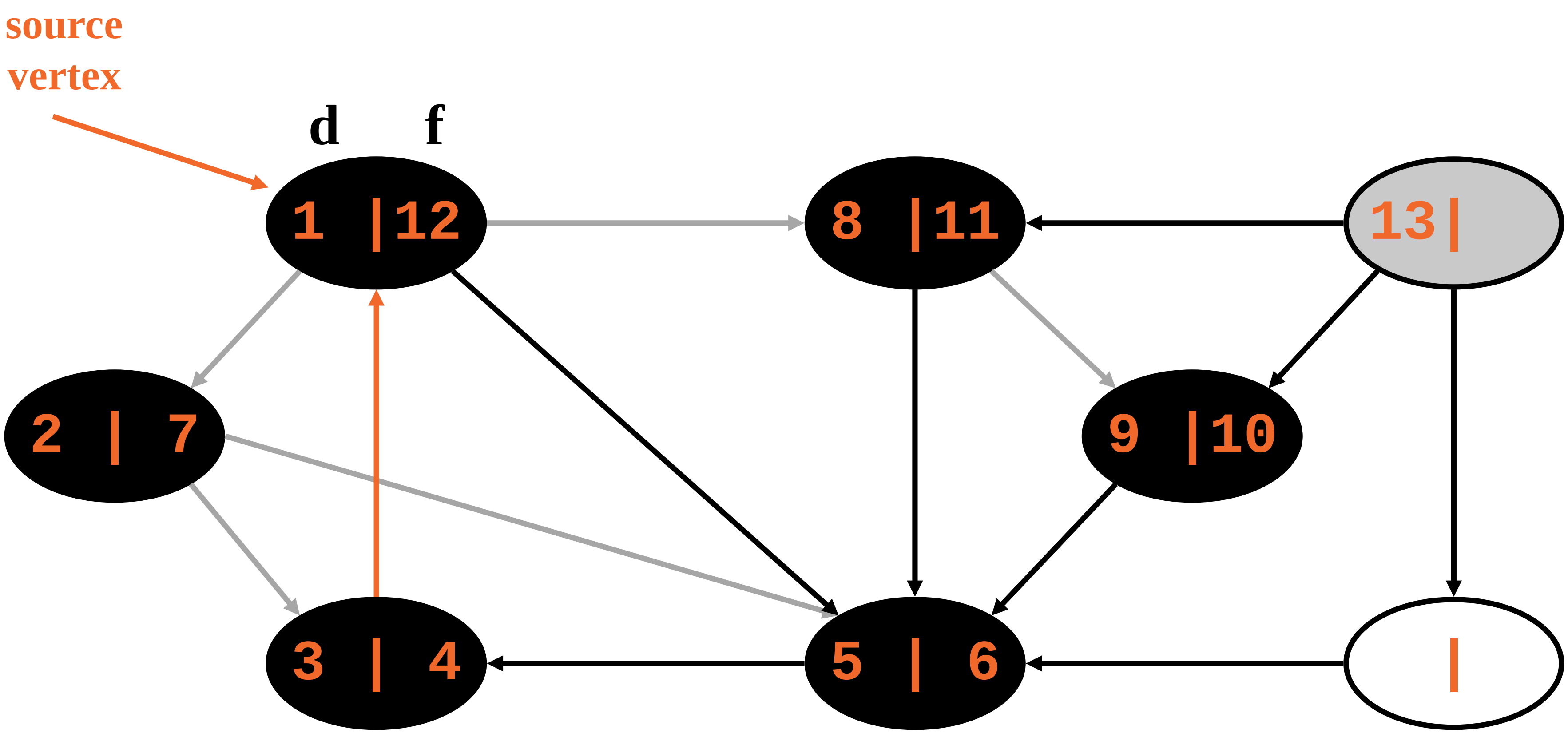
DFS Example



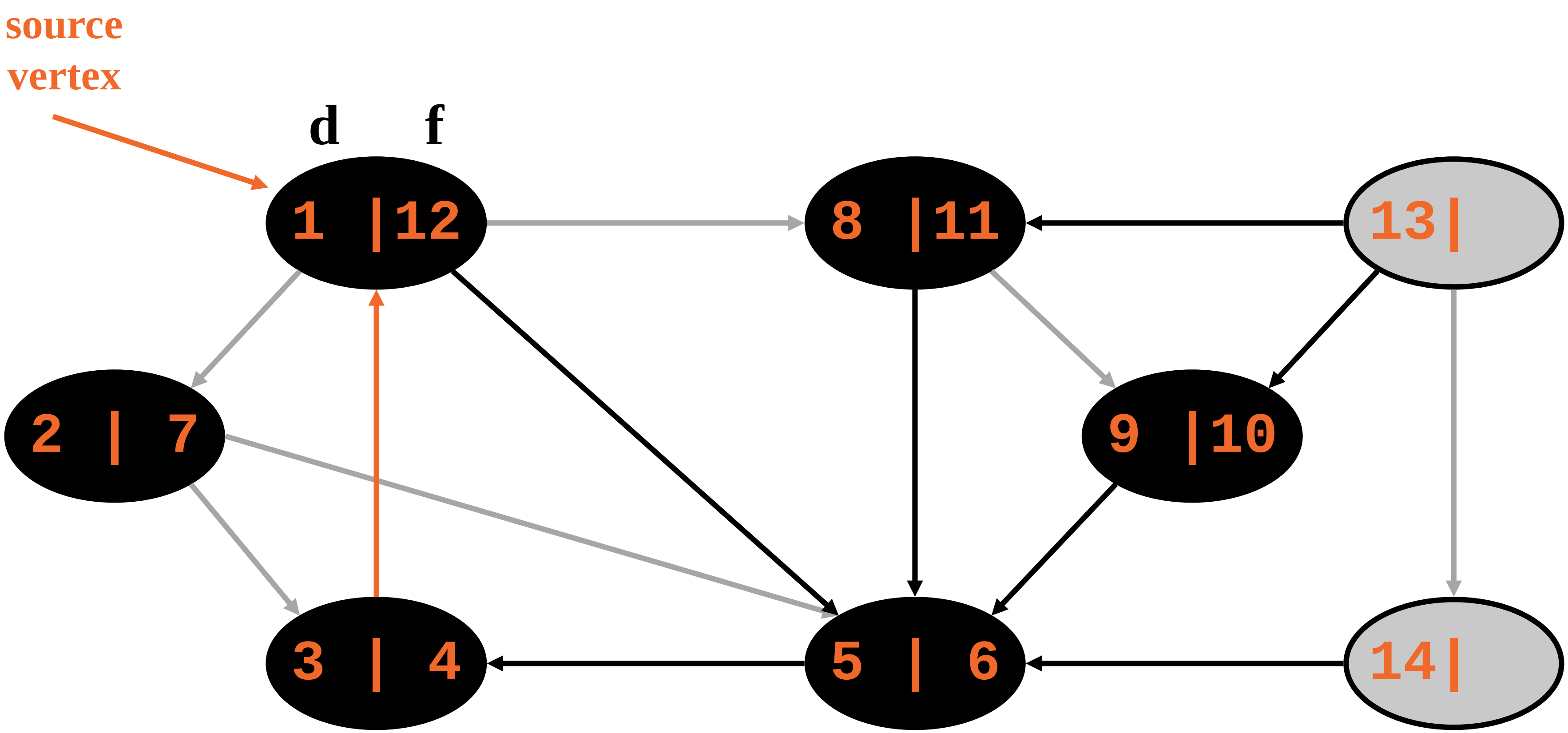
DFS Example



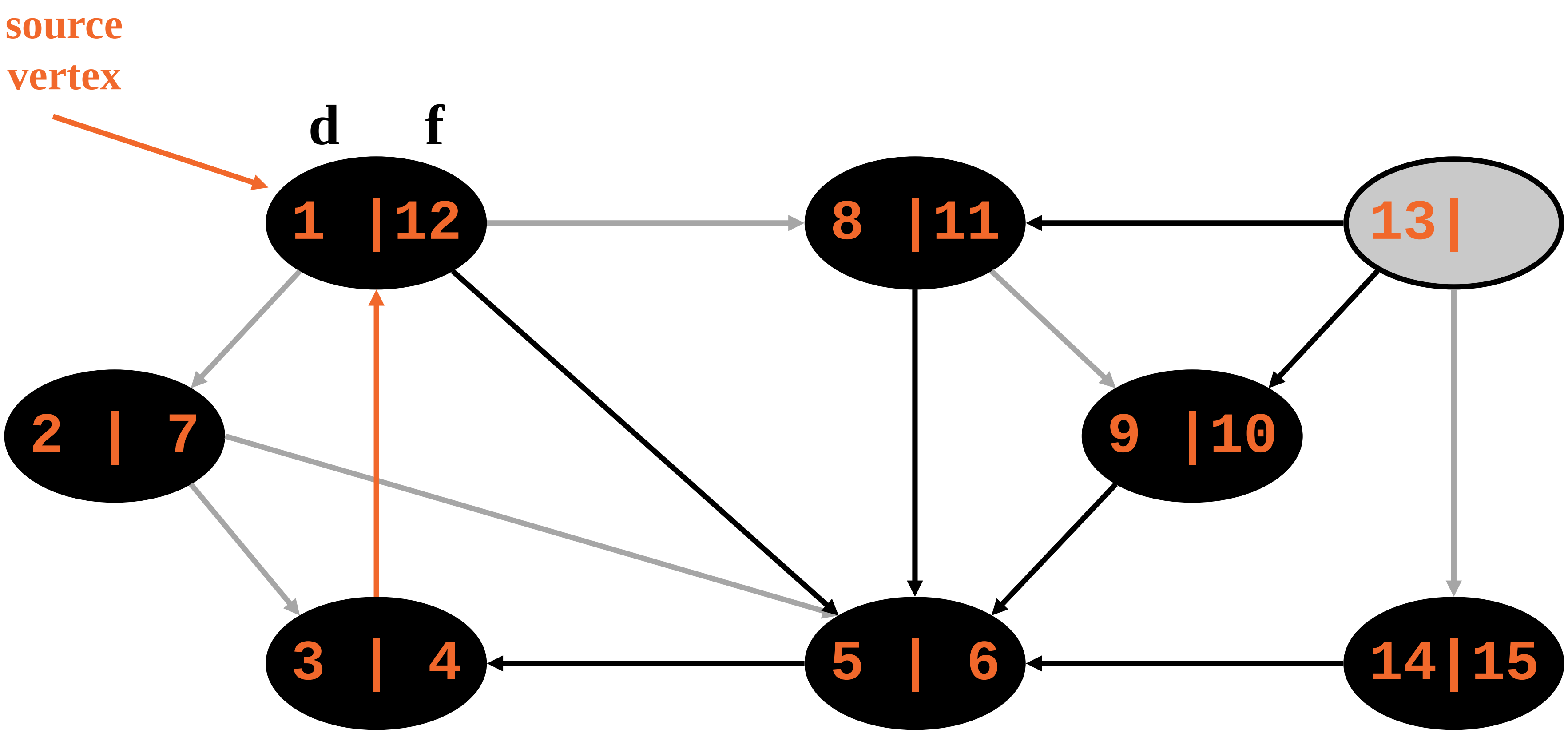
DFS Example



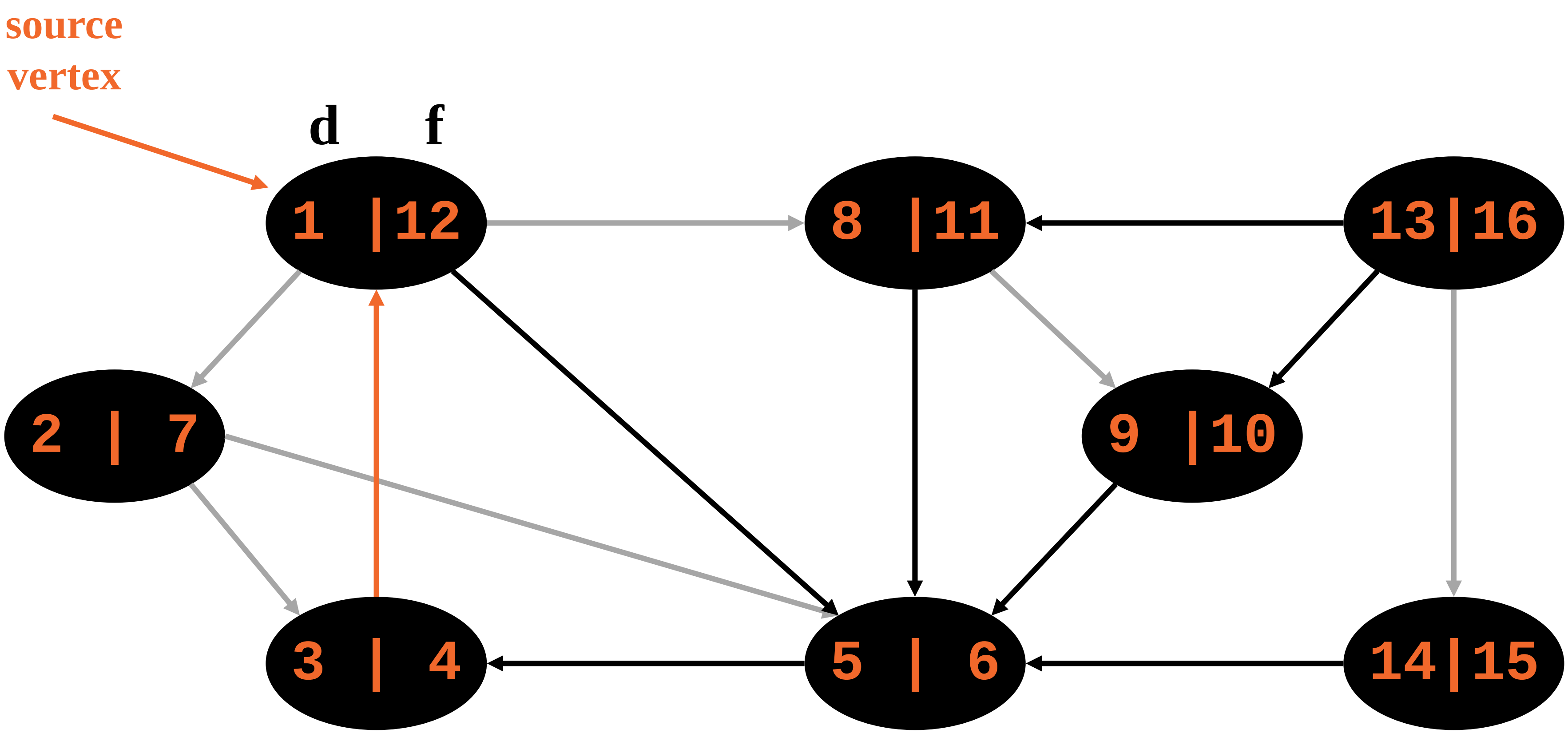
DFS Example



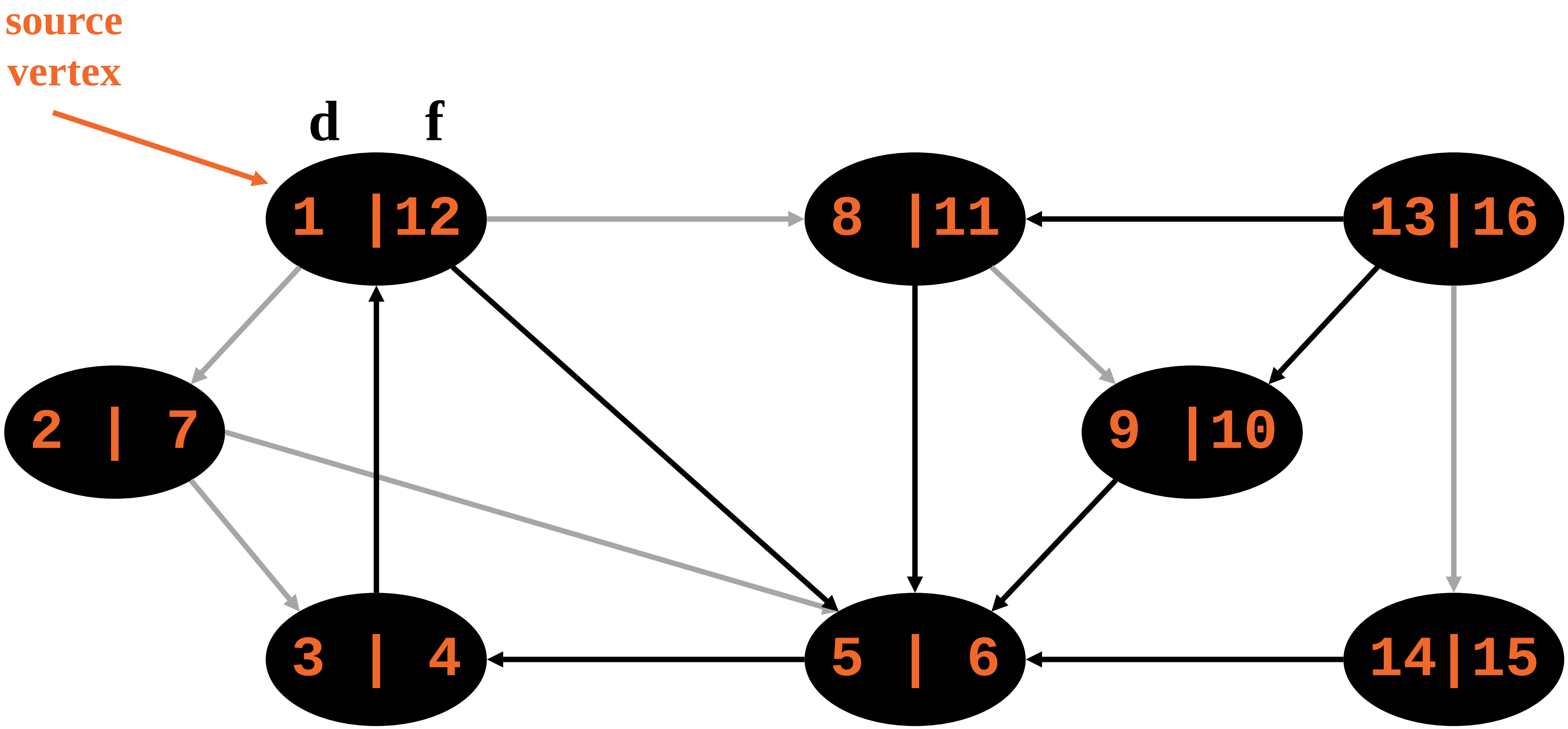
DFS Example



DFS Example

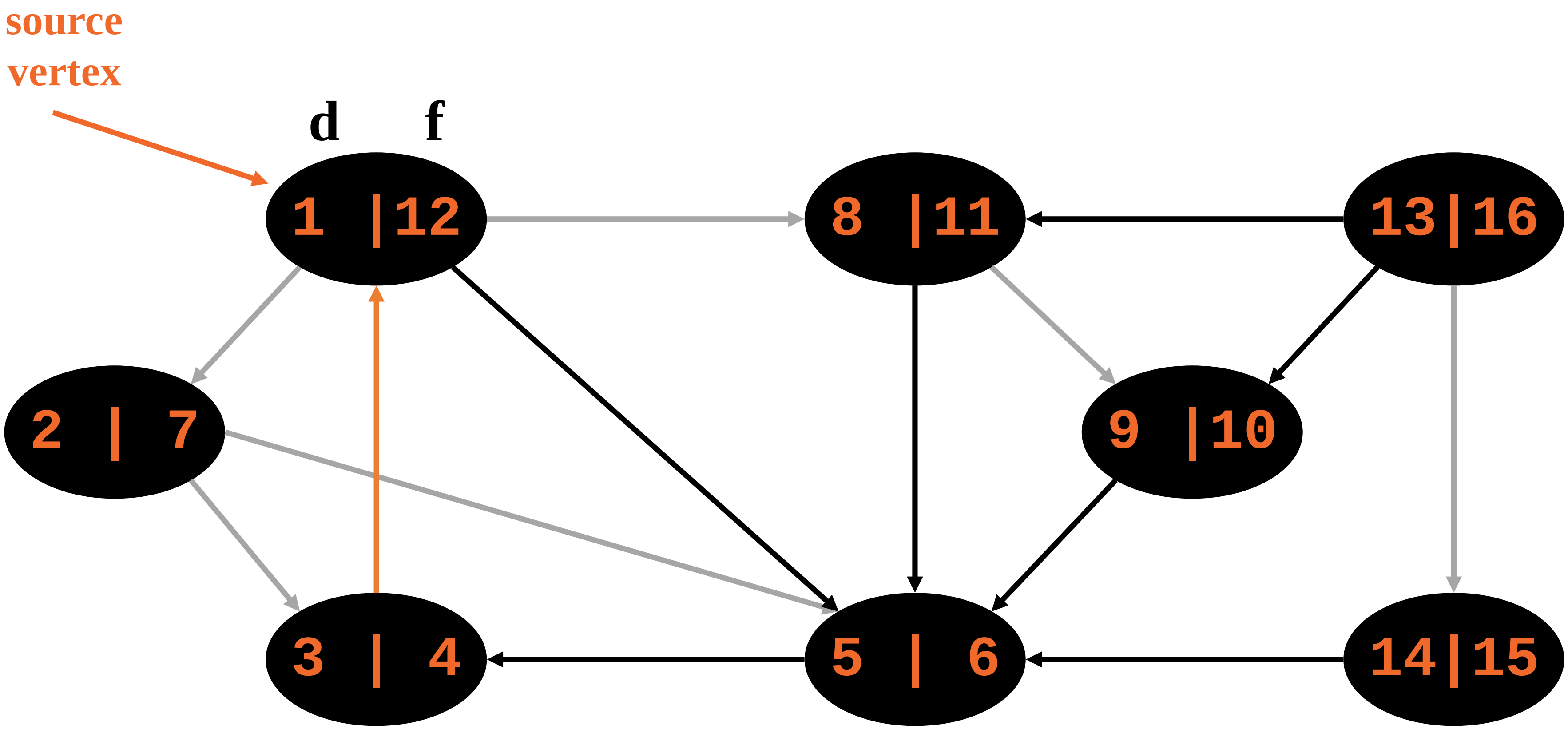


DFS Example



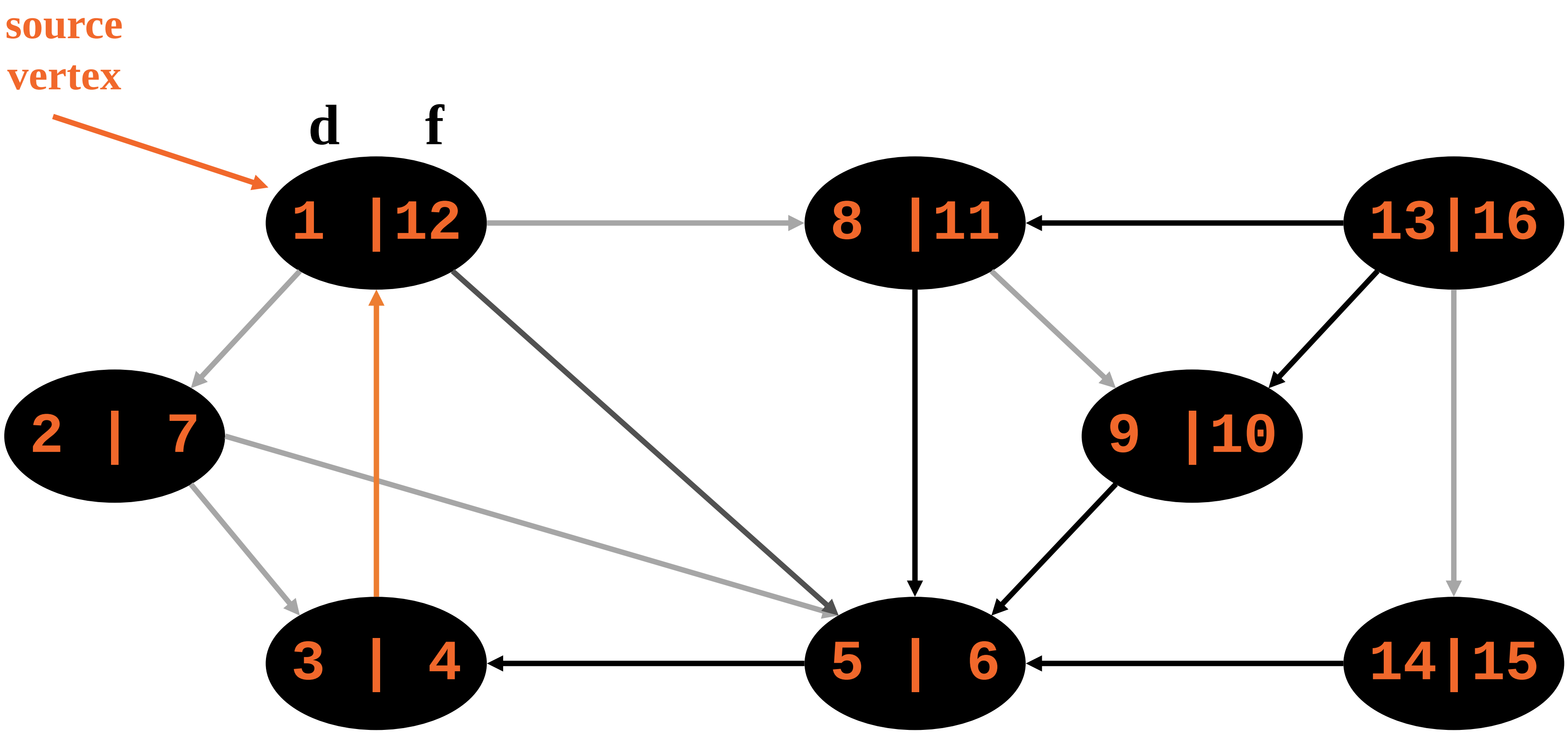
Tree edges

DFS Example



Tree edges   Back edges

DFS Example

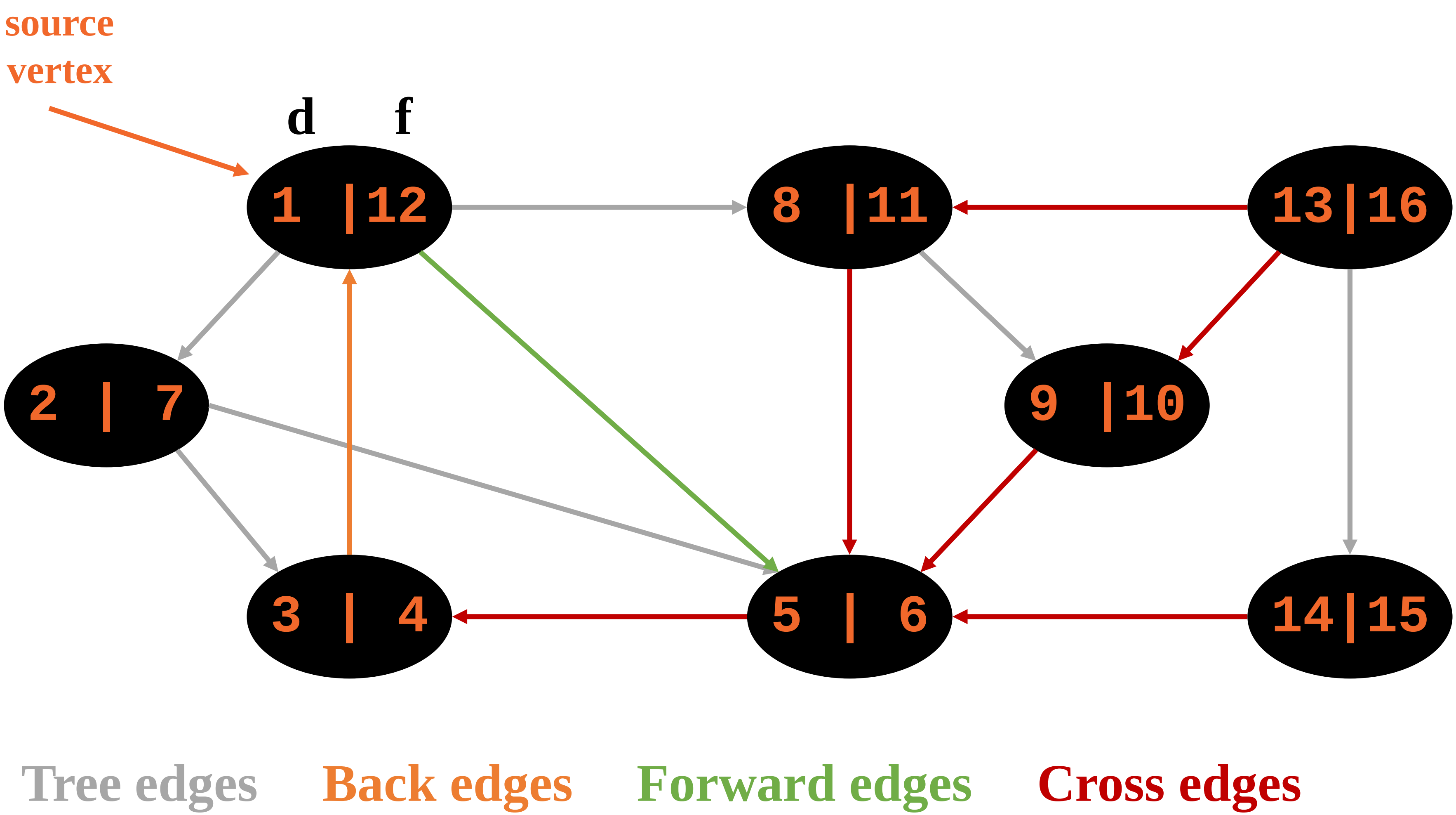


Tree edges Back edges Forward edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don’t distinguish forward & cross

DFS Example


source
vertex
d
f
1 |12
8 |11
13|16
2 | 7
9 |10
3 | 4
5 | 6
14|15
Tree edges
Back edges
Forward edges
Cross edges

DFS And Graph Cycles

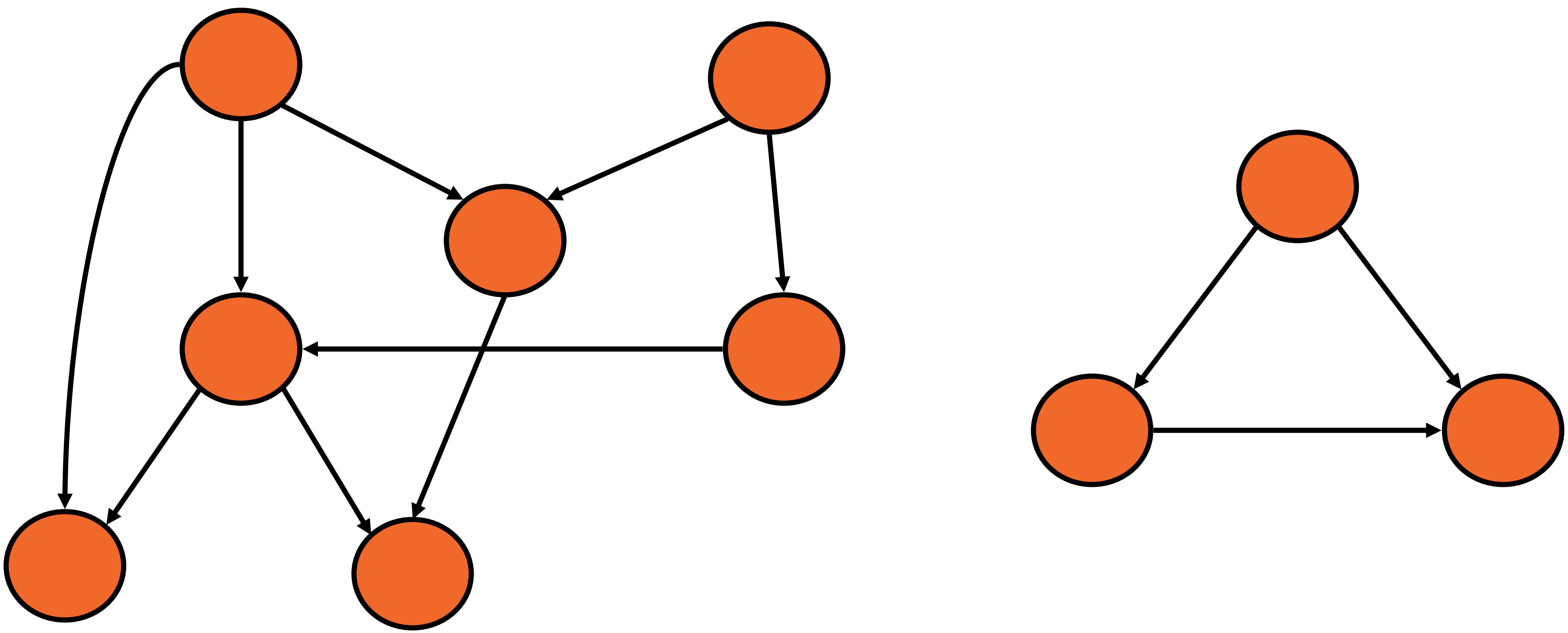
- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
  - If acyclic, no back edges (because a back edge implies a cycle
  - If no back edges, acyclic
    - No back edges implies only tree edges Only tree edges implies we have a tree or a forest
    - Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

DFS And Cycles

- $\Theta(V+E)$
- We can actually determine if cycles exist in $\Theta(V)$ time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

Directed Acyclic Graphs

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:
- directed graph G is acyclic iff a DFS of G yields no back edges:

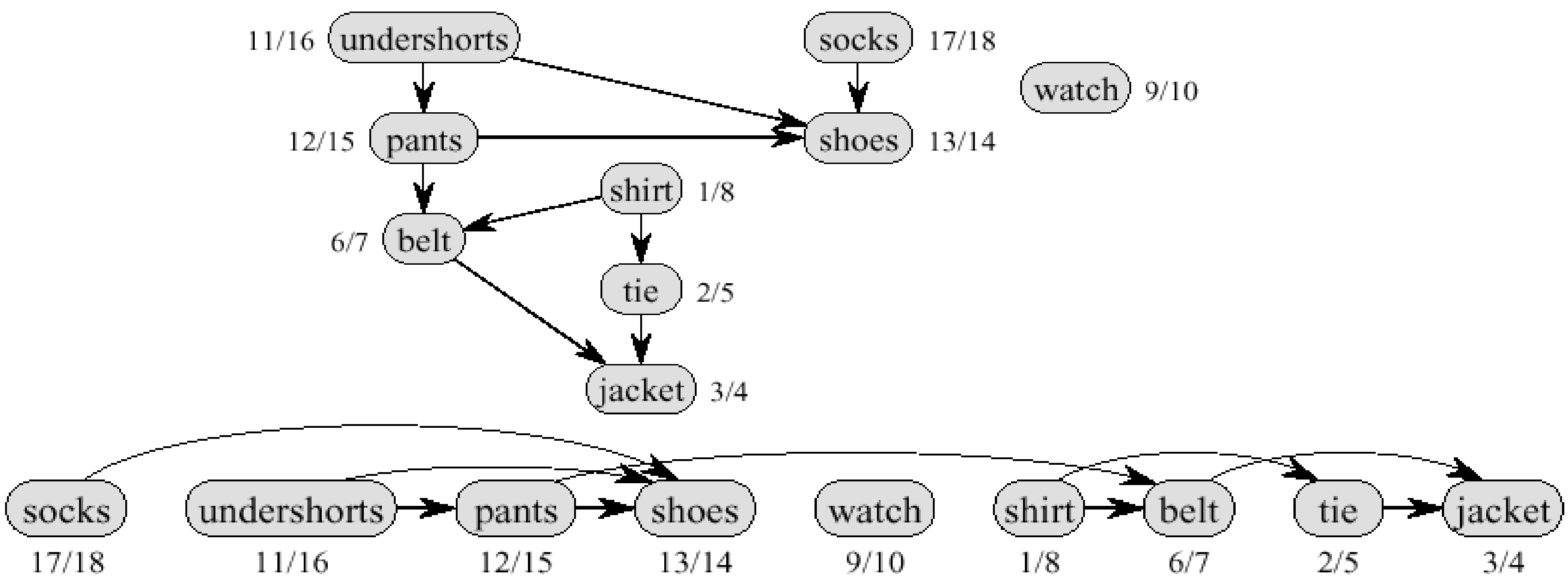


Topological Sort

- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex u comes before vertex v if edge $(u, v) \in G$
- Real-world example: getting dressed

Topological Sort Example

- Precedence relations: an edge from *x* to *y* means one must be done with *x* before one can do *y*
- Intuition: can schedule task only when all of its subtasks have been scheduled



Topological Sort Algorithm

```
Topological-Sort()
{
   Run DFS
   When a vertex is finished, output it
   Vertices are output in reverse topological
     order
}
```

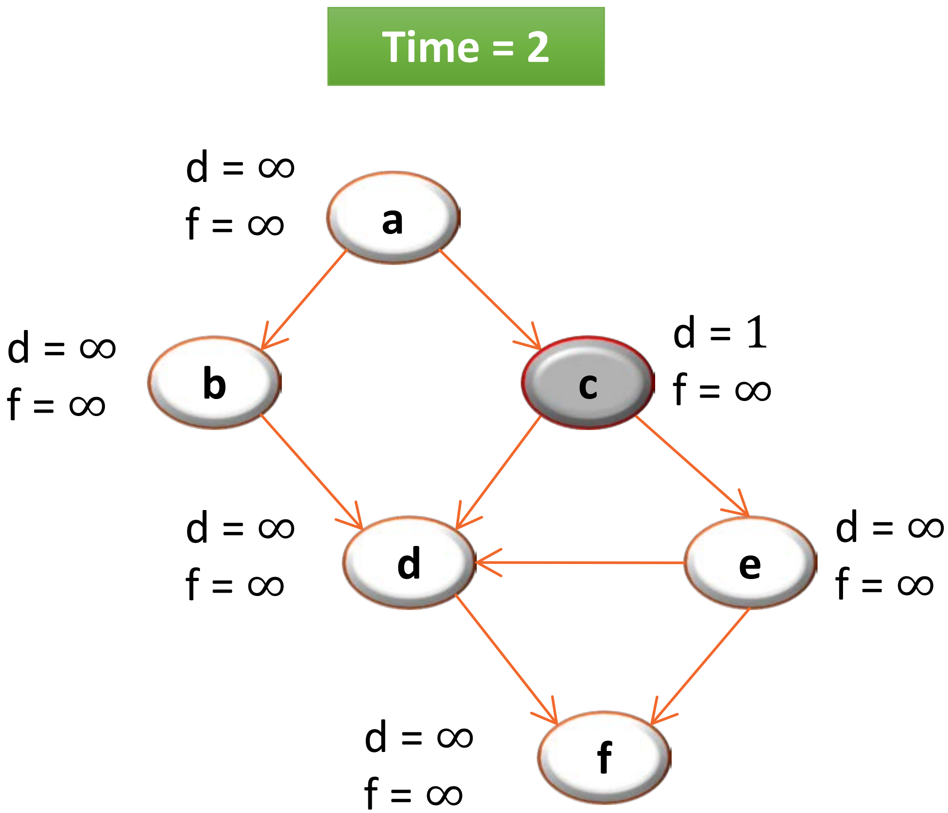
- Time: $\Theta(V+E)$

Topological Example

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

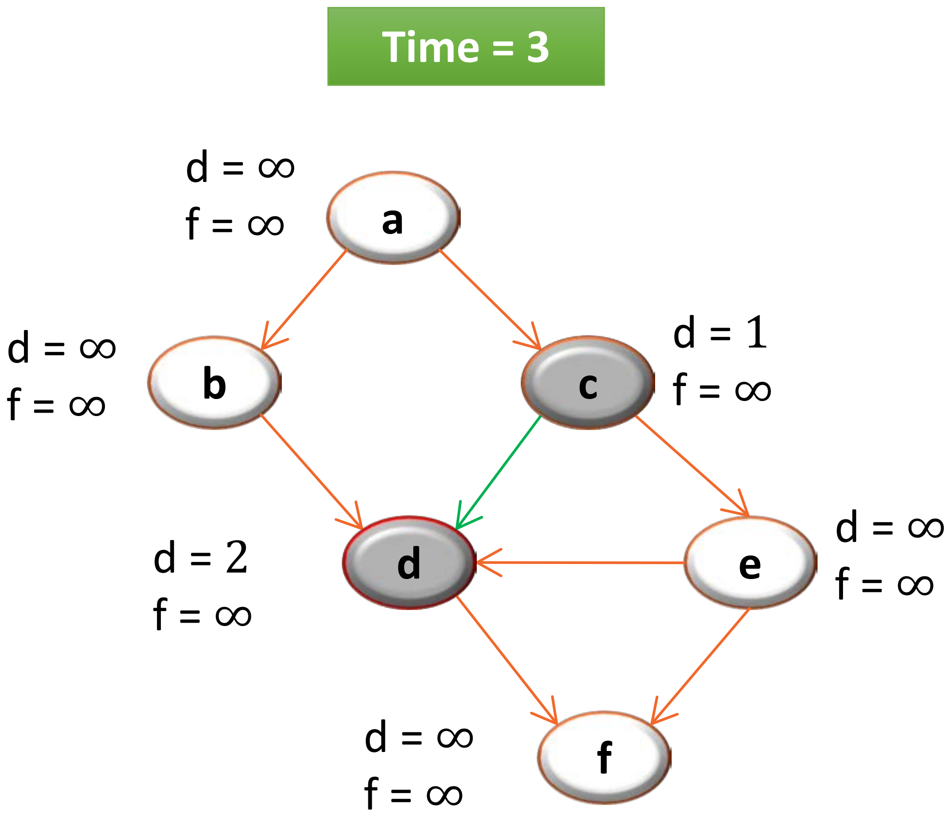
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let’s say we start the DFS from the vertex **c**

Next we discover the vertex **d**

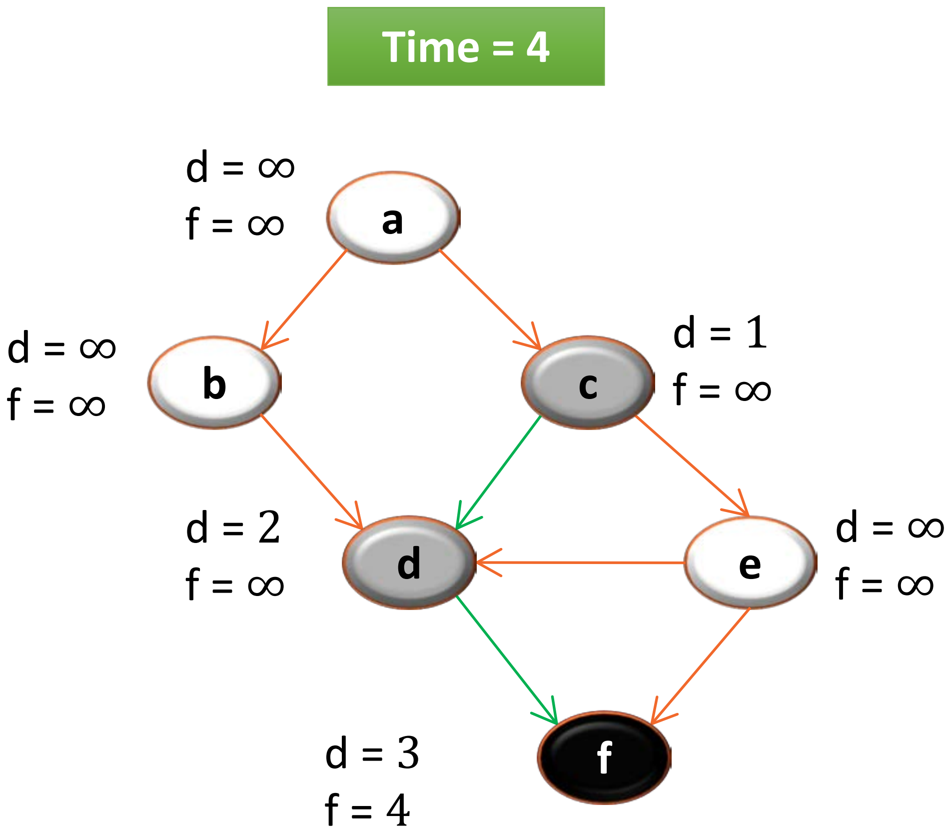
Topological sort

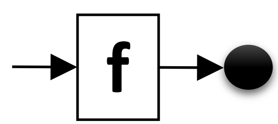
1) Call DFS(**G**) to compute the finishing times **f[v]**

2) as each vertex is finished, insert it onto the front of a linked list

Next we discover the vertex **f**

f is done, move back to **d**

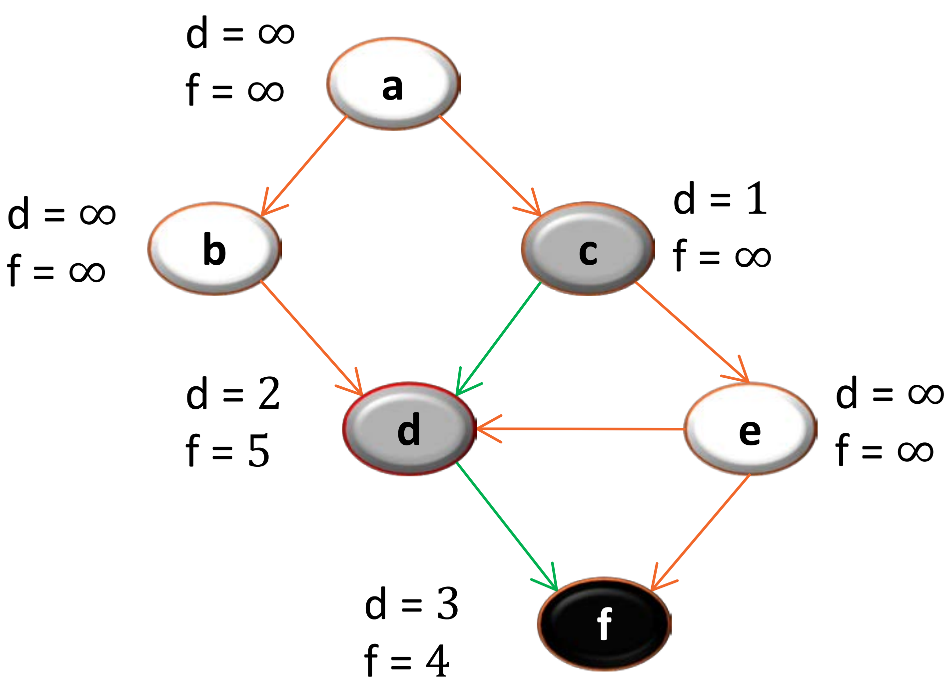




Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 5



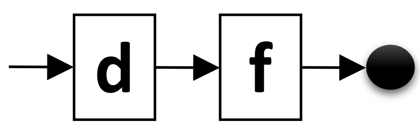
Let’s say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

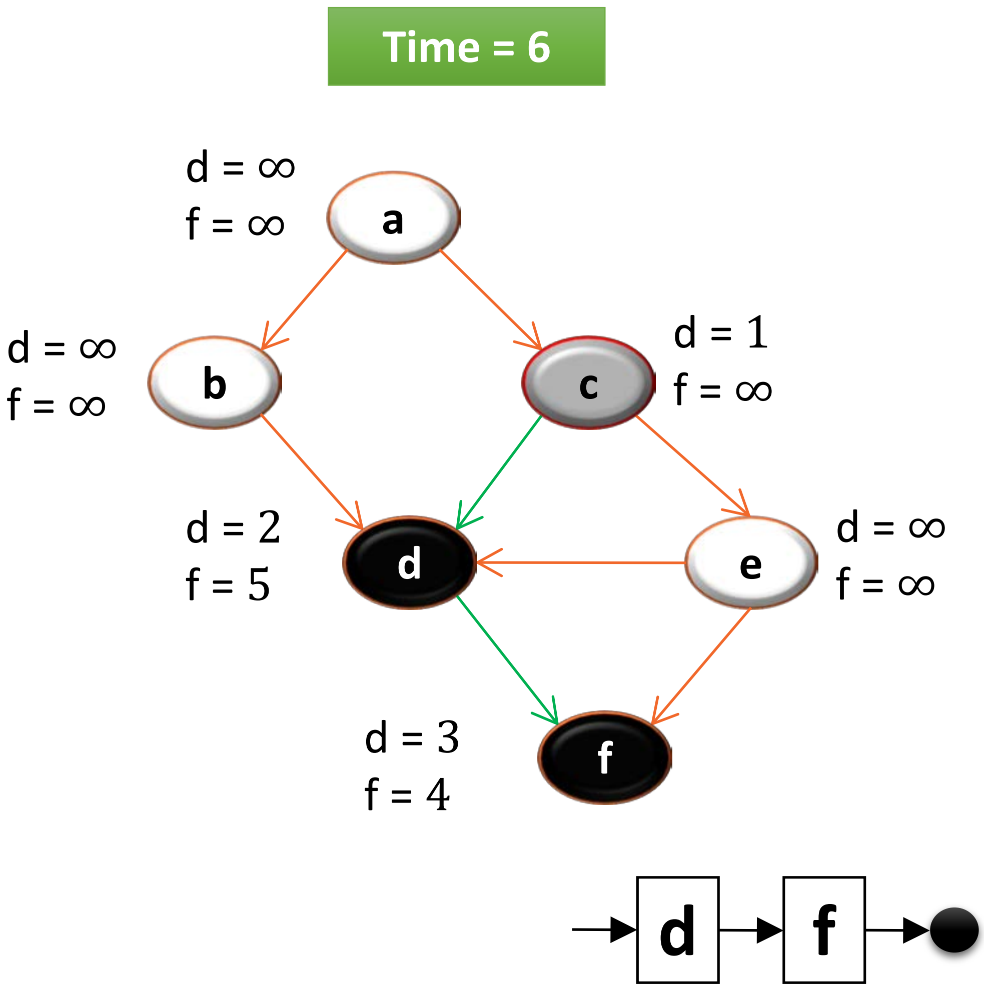
f is done, move back to **d**

d is done, move back to **c**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**



Let’s say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

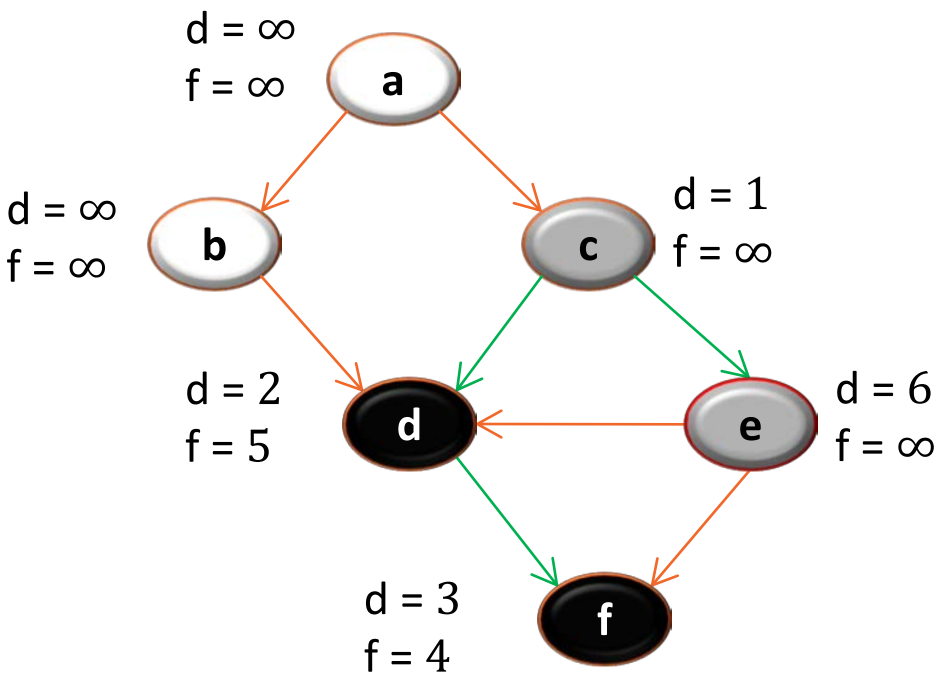
d is done, move back to **c**

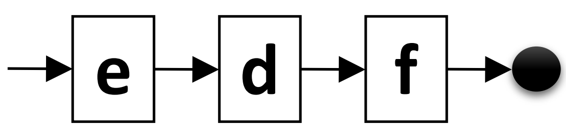
Next we discover the vertex **e**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**

Time = 7





Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Both edges from **e** are **cross edges**

d is done, move back to **c**

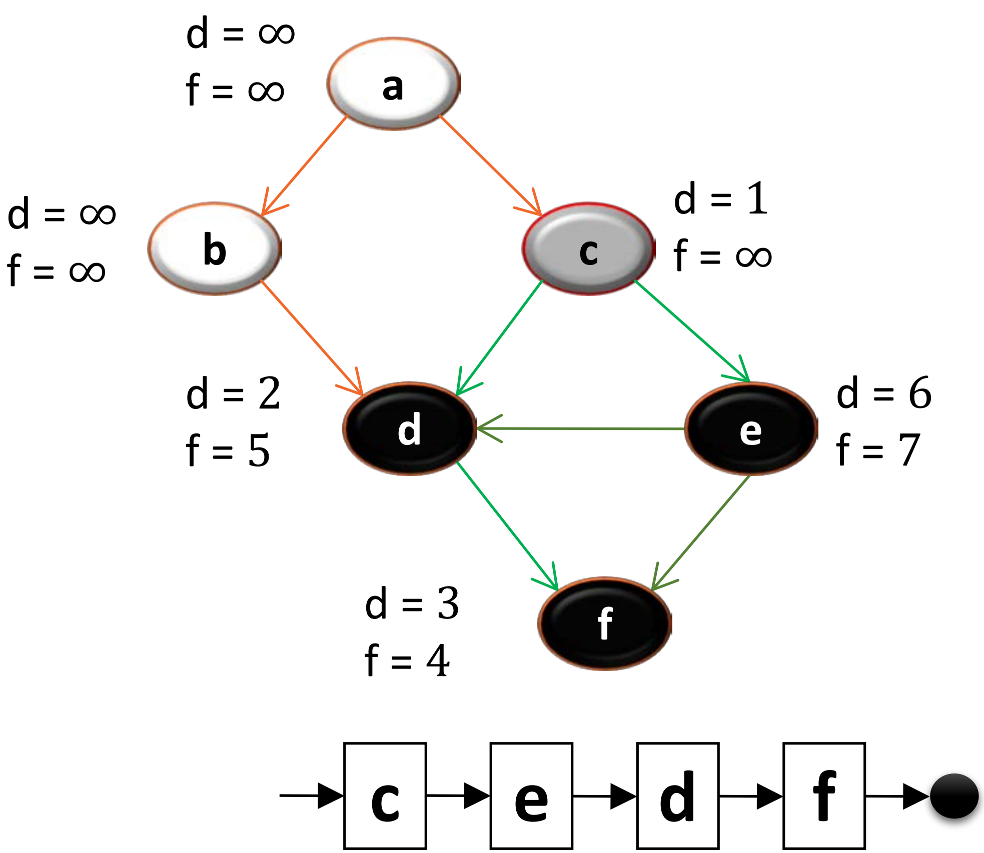
Next we discover the vertex **e**

e is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**

Time = 8



Let’s say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

d is done, move back to **c**

Next we discover the vertex **e**

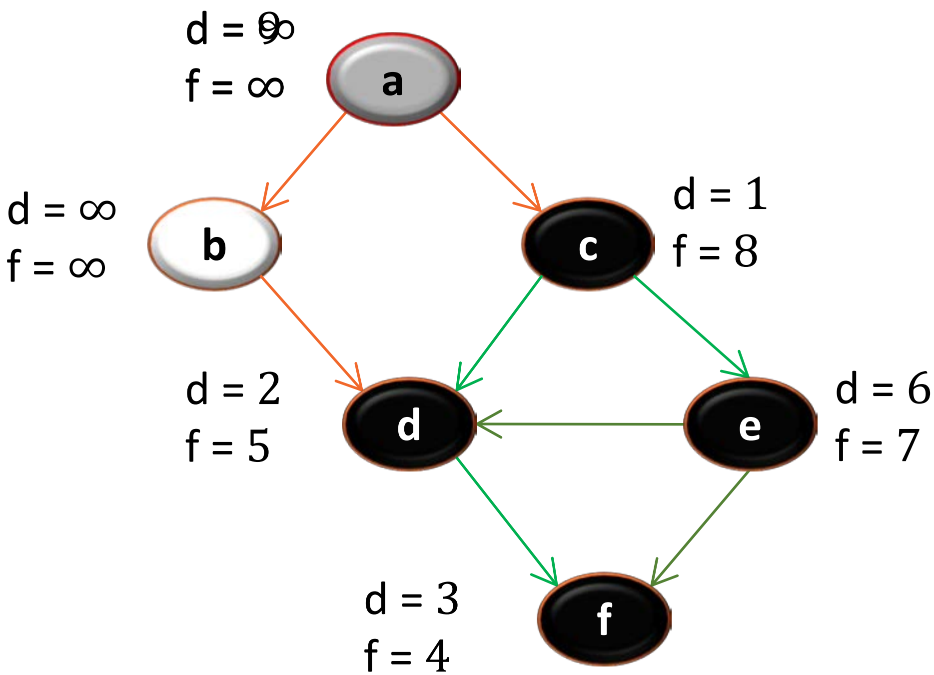
e is done, move back to **c**

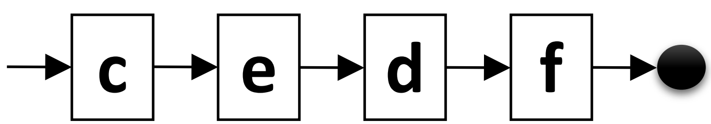
c is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 10





Let’s now call DFS visit from the vertex **a**

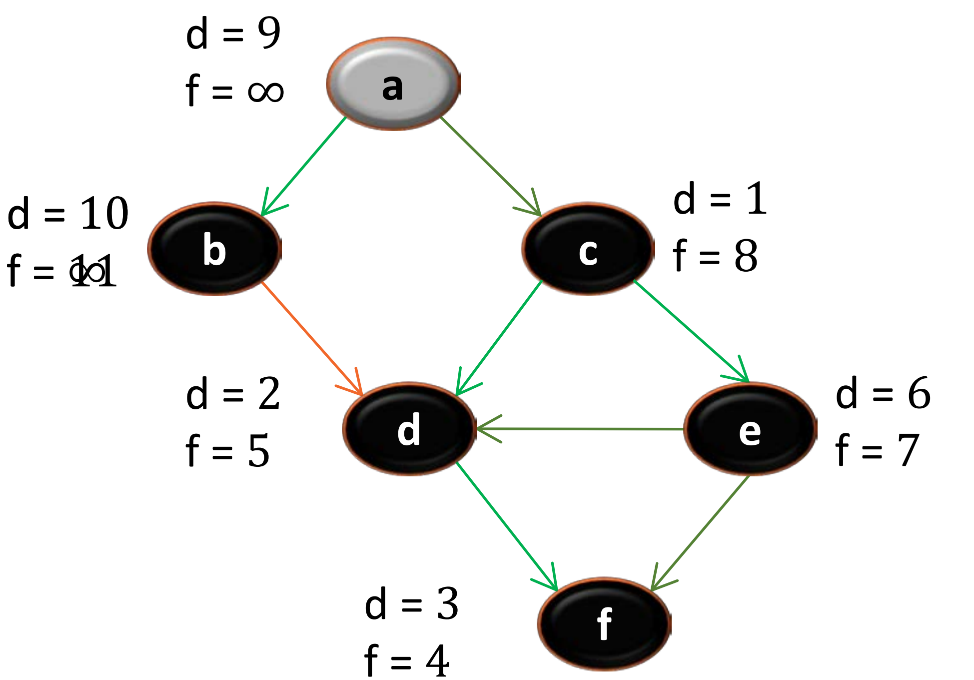
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

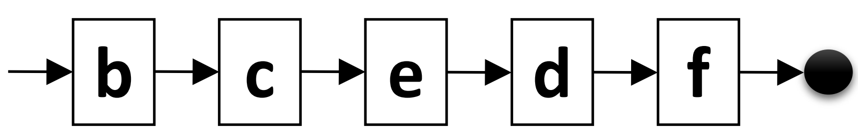
Next we discover the vertex **b**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 11





Let’s now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

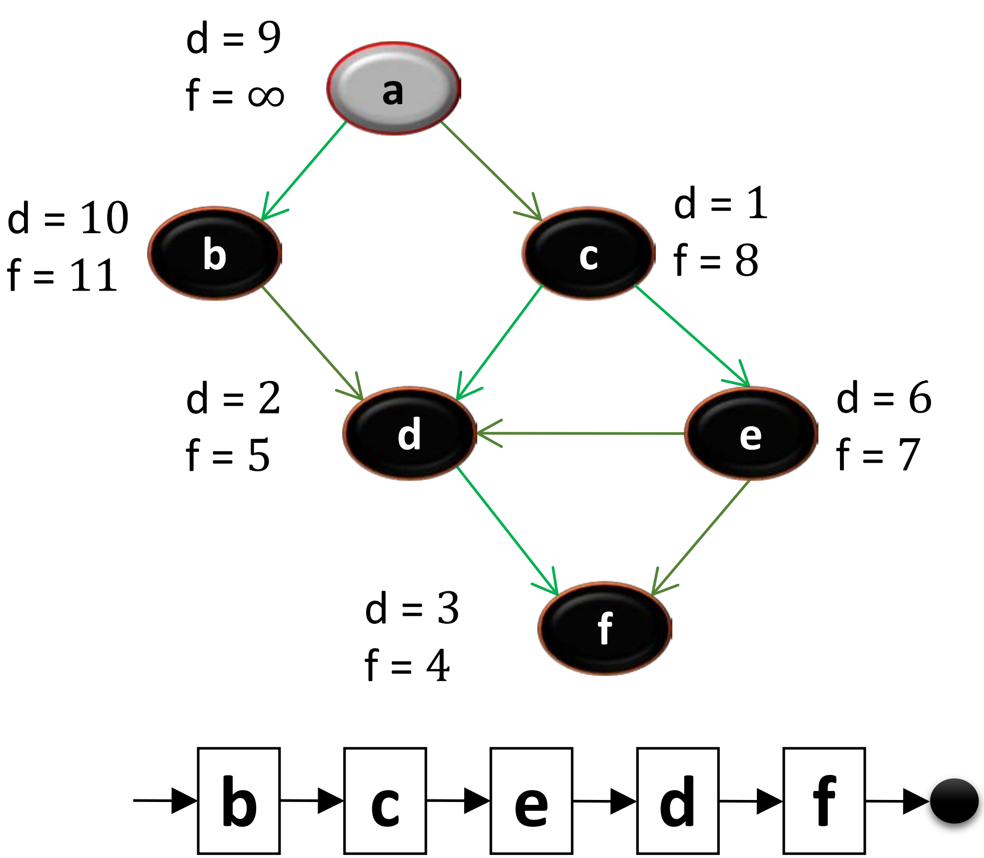
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 12



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

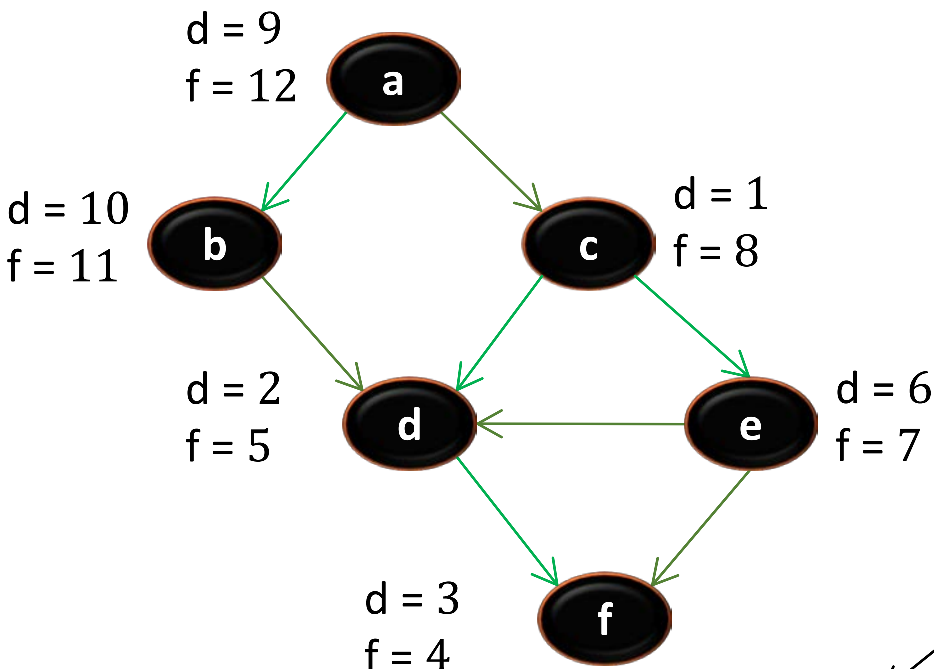
Next we discover the vertex **b**

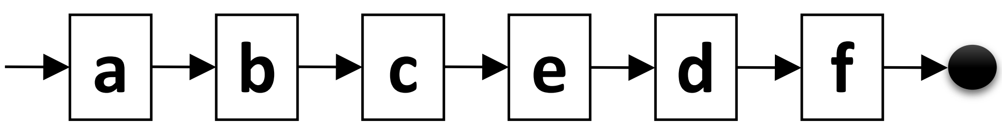
b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

Time = 13





1) Call DFS(**G**) to compute the finishing times **f**[**v**]

WE HAVE THE RESULT!

3) return the linked list of vertices

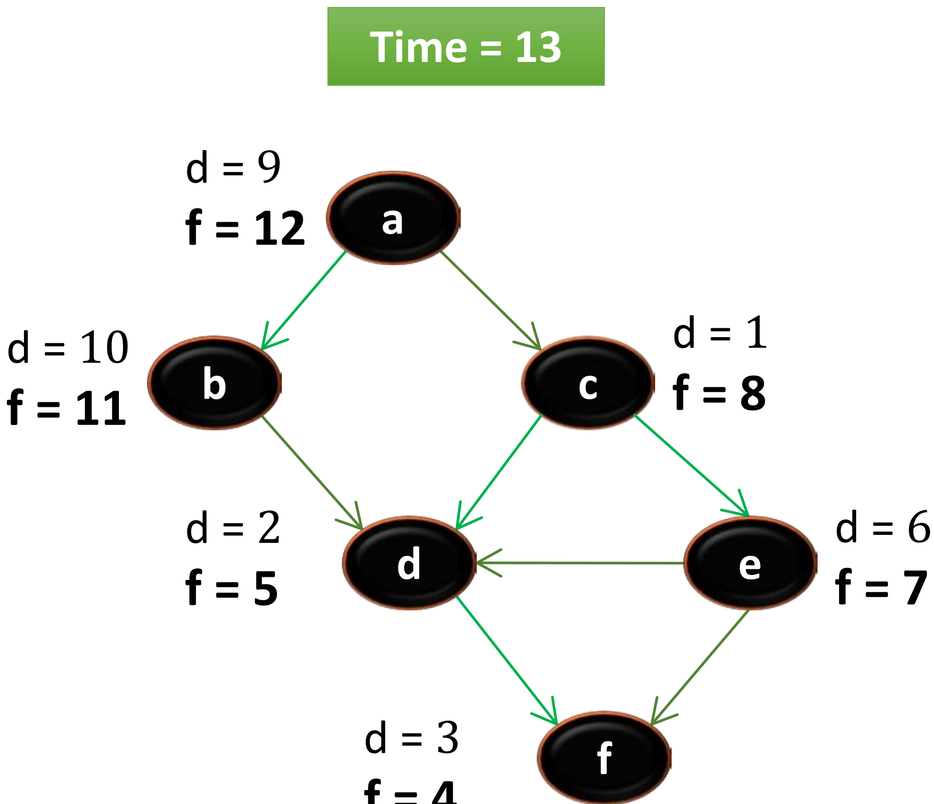
(**a**,**c**) is a cross edge

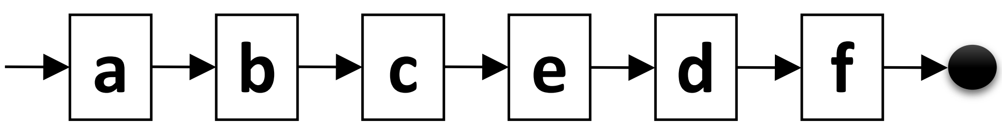
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort





The linked list is sorted in **decreasing** order of finishing times **f**[]

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „**from left to right**“

Graph Algorithms

Part 2 MST

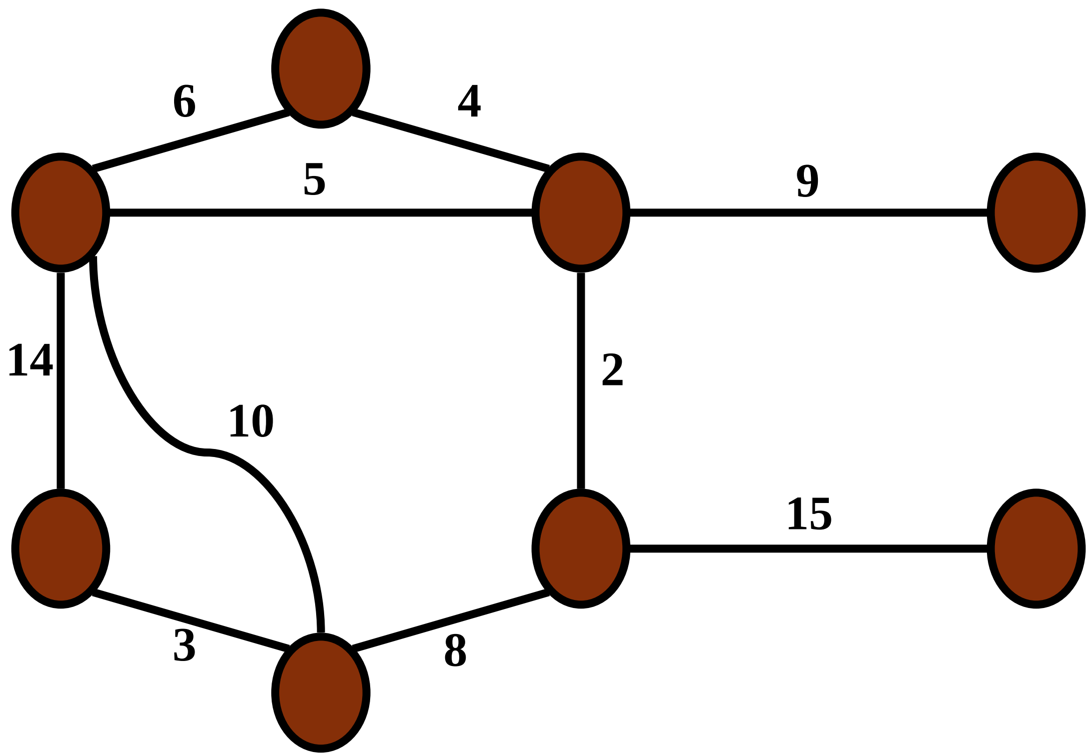
CS 325

Ch 23 Spanning Trees

- Weighted Graphs
- Minimum Spanning Trees
  - Greedy Choice Theorem
  - Kruskal's Algorithm
  - Prim's Algorithm

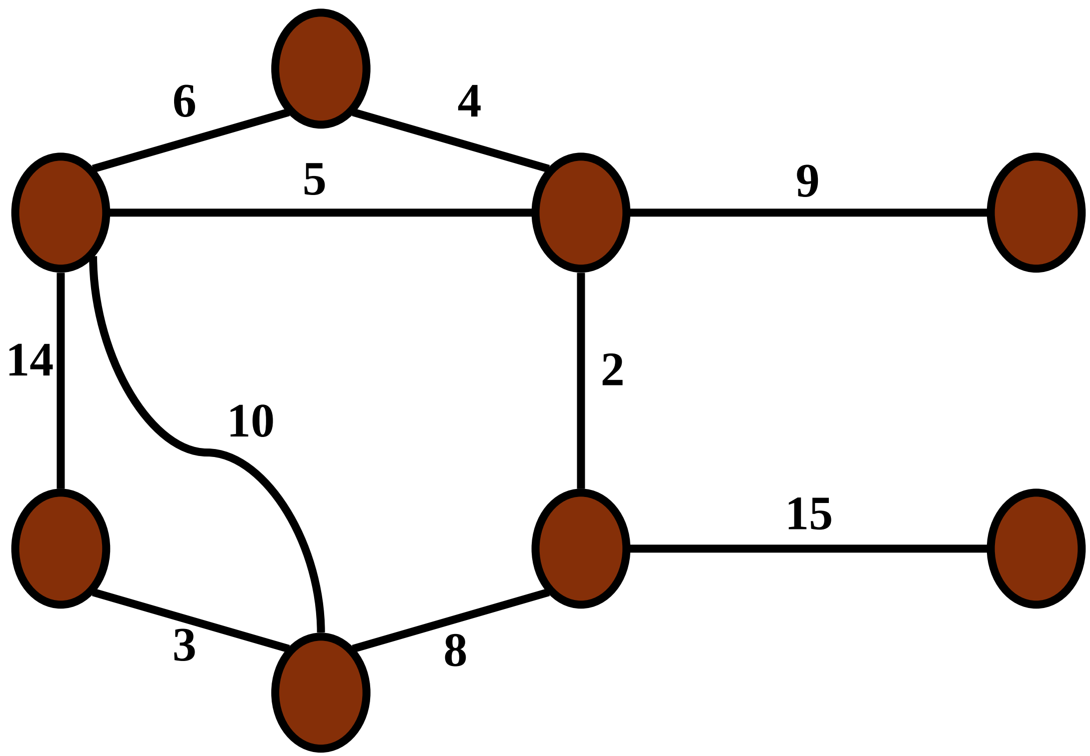
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



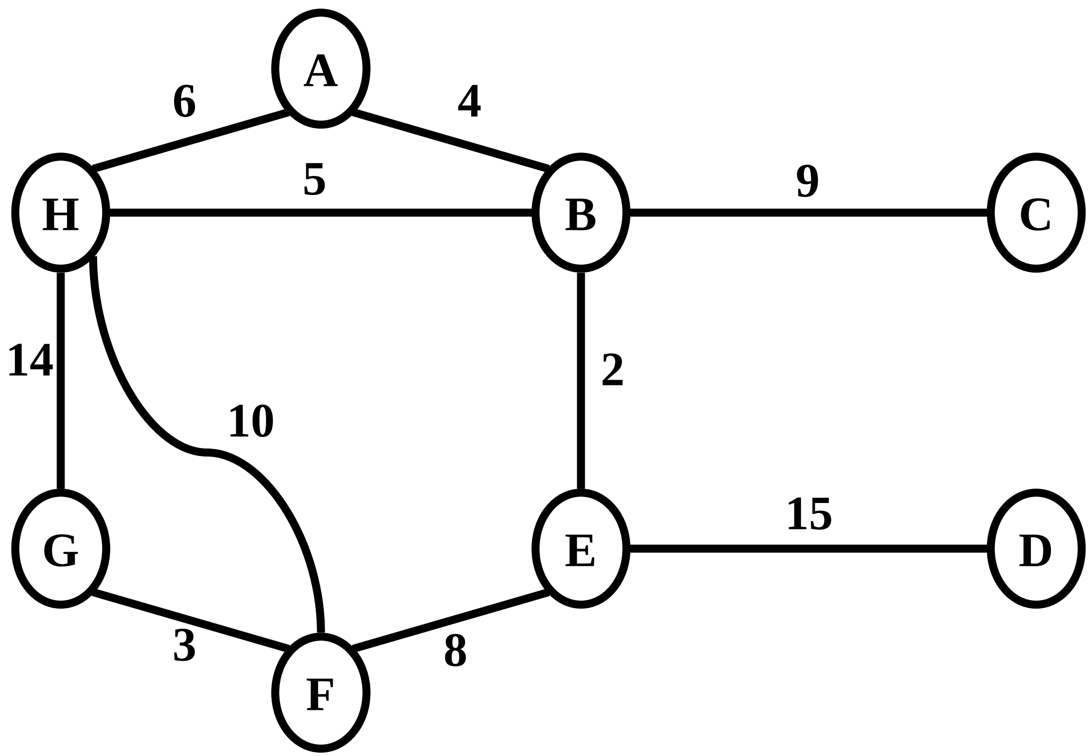
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



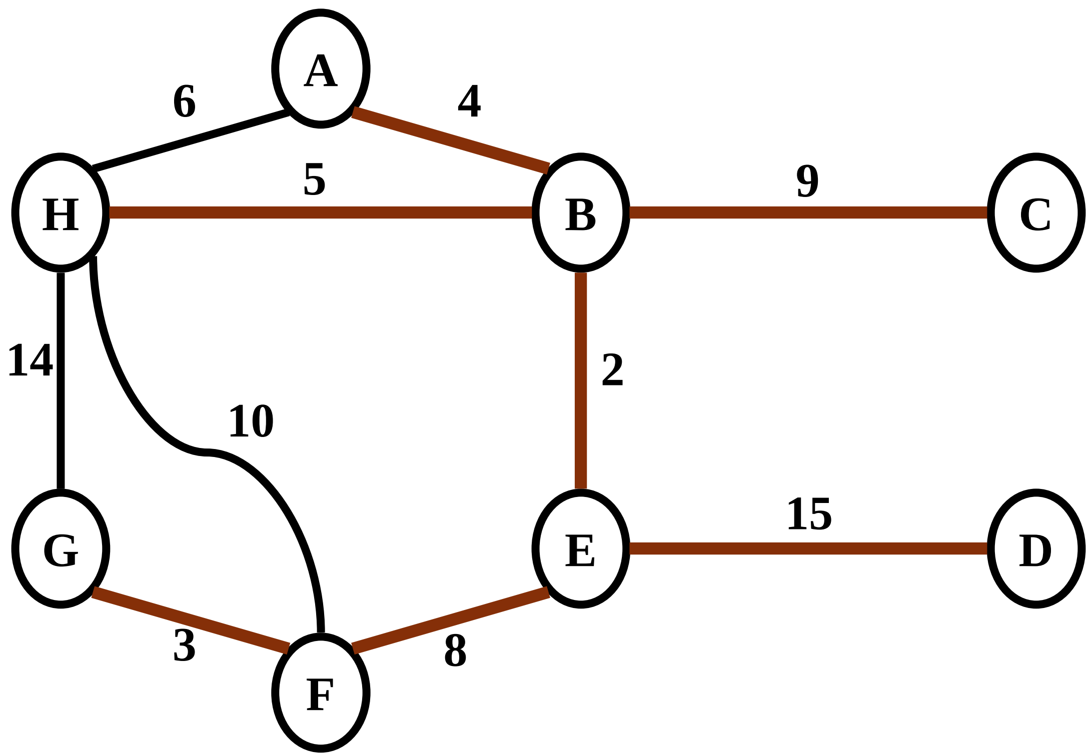
Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



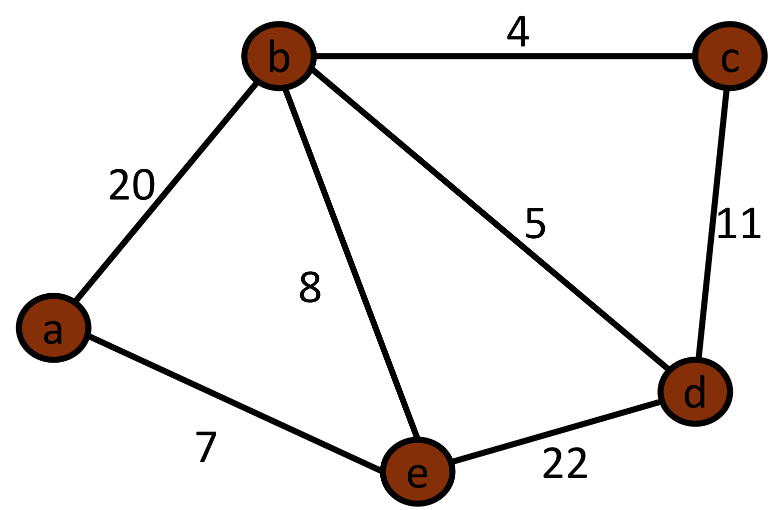
Minimum Spanning Tree

- Answer:



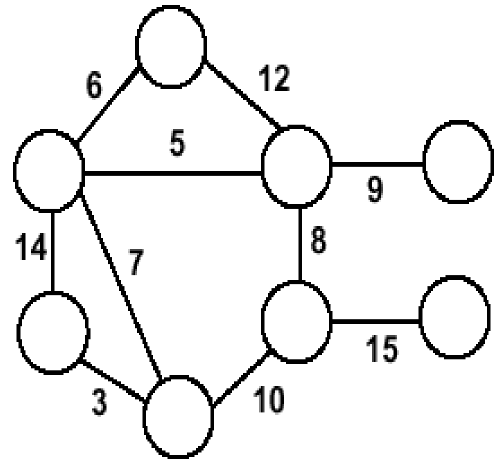
Greedy Algorithms for Minimum Spanning Tree

- [Prim] Extend a tree by including the cheapest out going edge
- [Kruskal] Add the cheapest edge that joins disjoint components



Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- Weight function W: $E \rightarrow R$ (assigning cost or length or other values to edges)

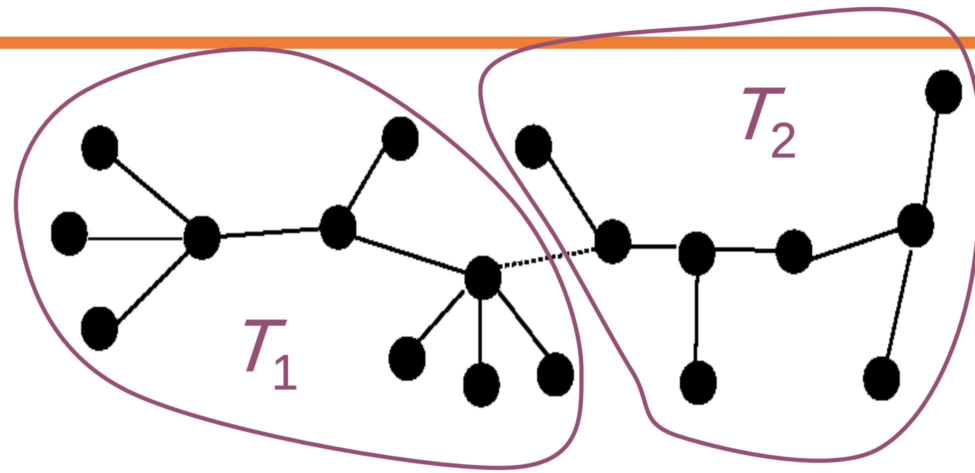


- Spanning tree: tree that connects all the vertices (above?)
- Minimum spanning tree: tree that connects all the vertices and minimizes

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Optimal Substructure

- MST T



- Removing the edge (u,v) partitions T into T_1 and T_2

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

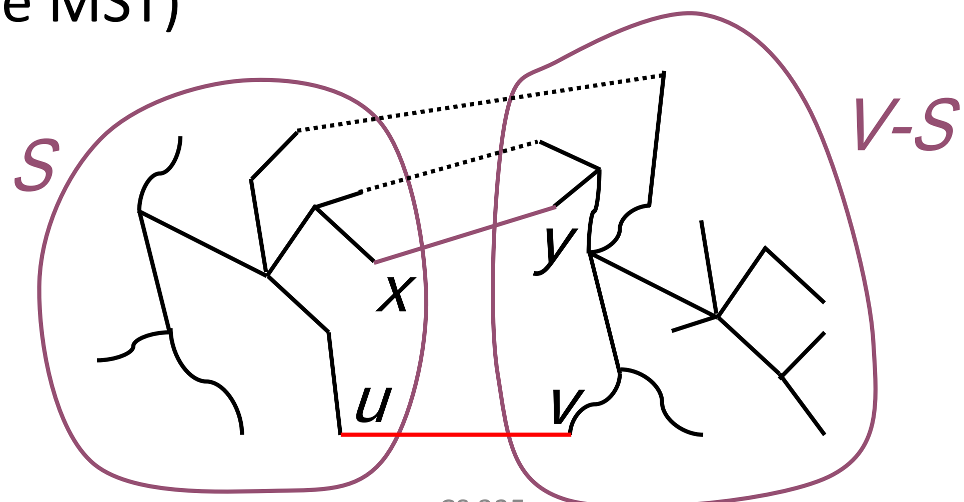
- We claim that T_1 is the MST of $G_1=(V_1,E_1)$, the subgraph of G induced by vertices in T_1
- Also, T_2 is the MST of G_2

Greedy Choice

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution
- Theorem
  - Let $G=(V, E)$, and let $S \subseteq V$ and
  - let (u,v) be min-weight edge in G connecting S to $V - S$
  - Then $(u,v) \in T$ – some MST of G

Greedy Choice (2)

- Proof
  - suppose $(u,v) \notin T$
  - look at path from u to v in T
  - swap (x, y) – the first edge on path from u to v in T that crosses from S to $V - S$
  - this improves T – contradiction (T supposed to be MST)



Prim’s Algorithm

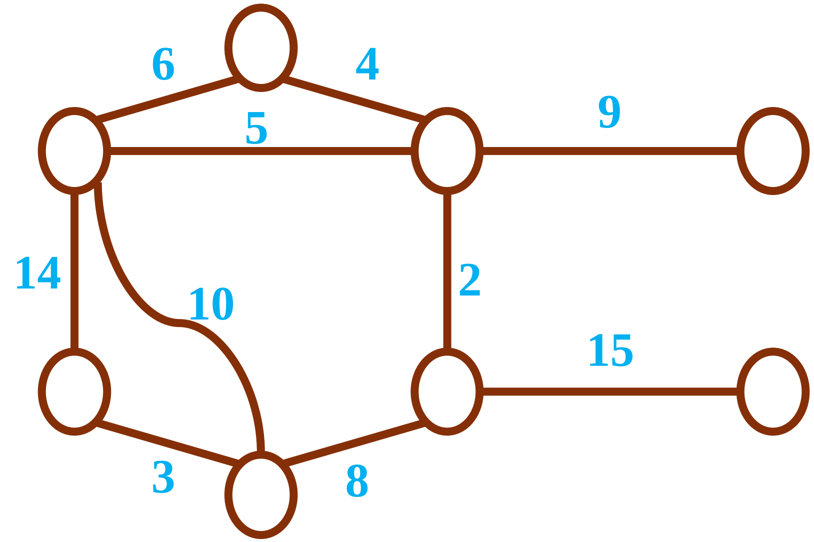
- Vertex based algorithm
- Grows one tree T, **one vertex at a time**
- A cloud covering the portion of T already computed
- Label the vertices *v* outside the cloud with *key*[*v*] – the minimum weigth of an edge connecting *v* to a vertex in the cloud, *key*[*v*] = ∞, if no such edge exists

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Prim's Algorithm

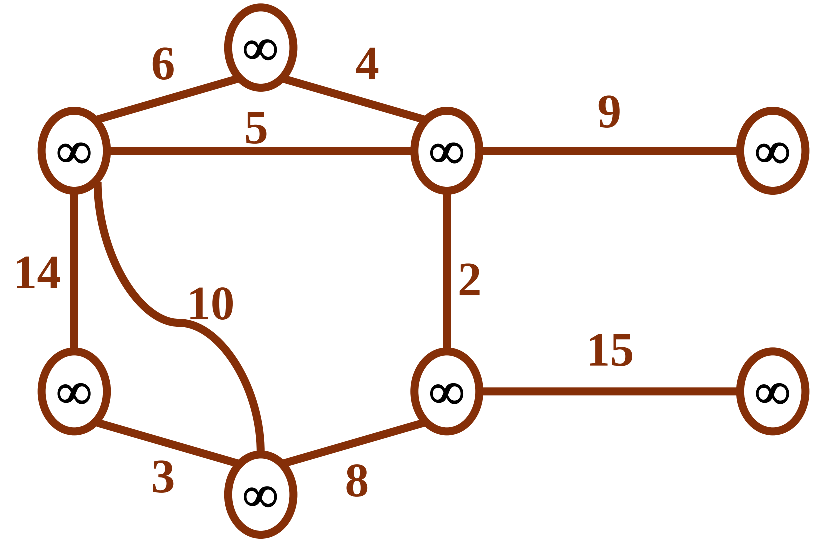
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim’s Algorithm

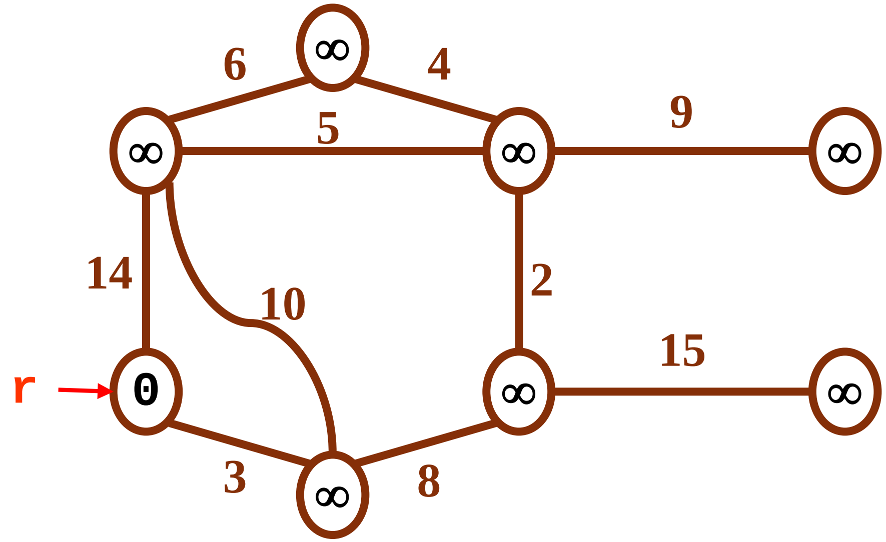
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

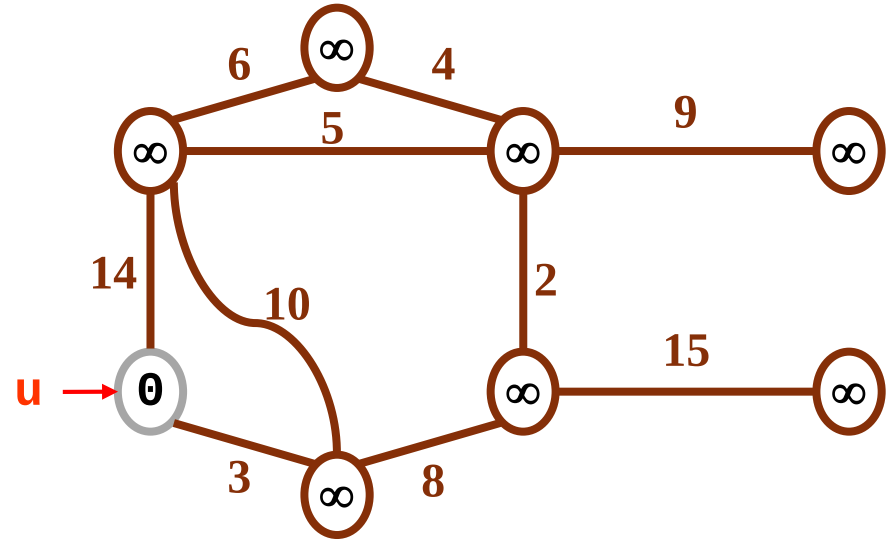
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Pick a start vertex r

Prim's Algorithm

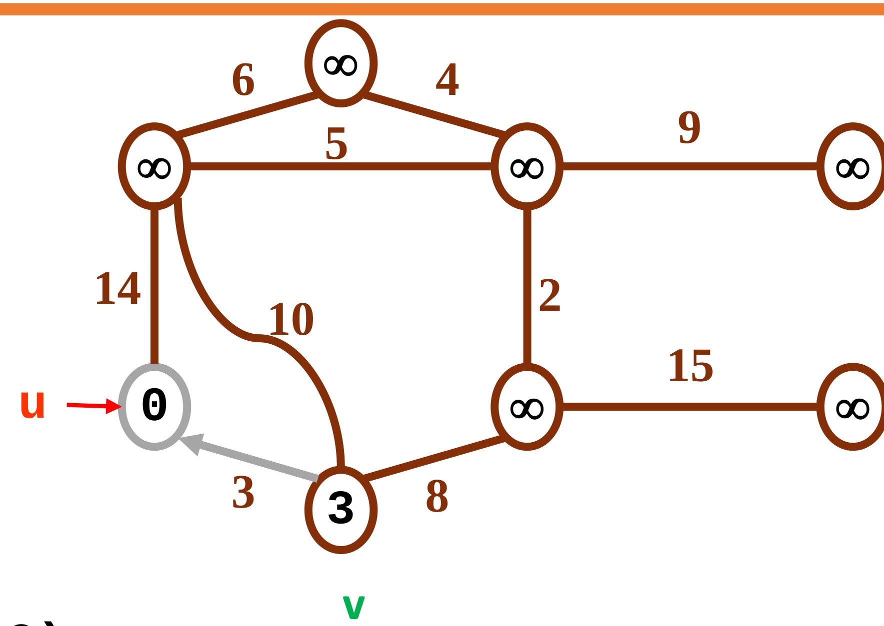
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Grey vertices have been removed from Q

Prim's Algorithm

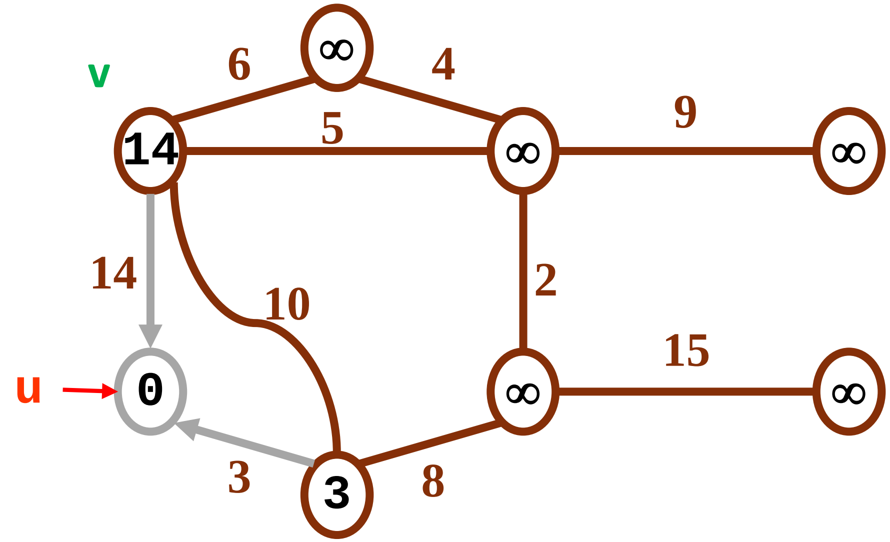
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Grey arrows indicate parent pointers

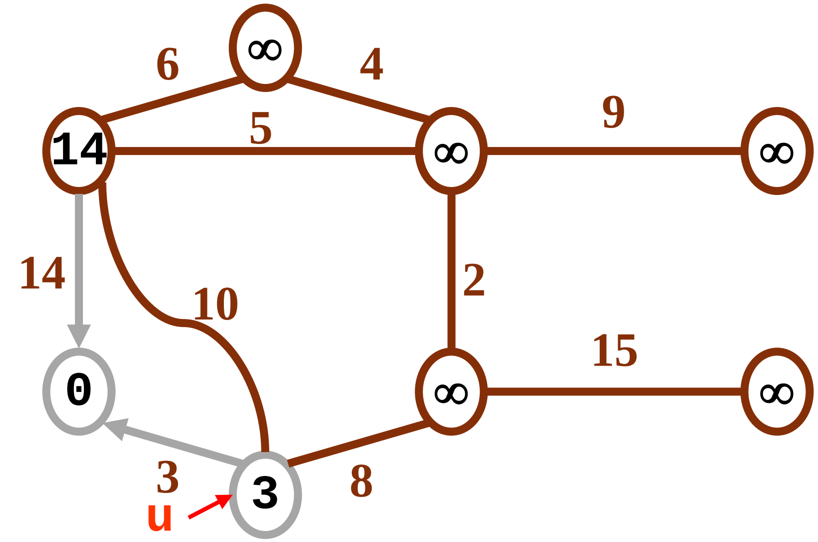
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



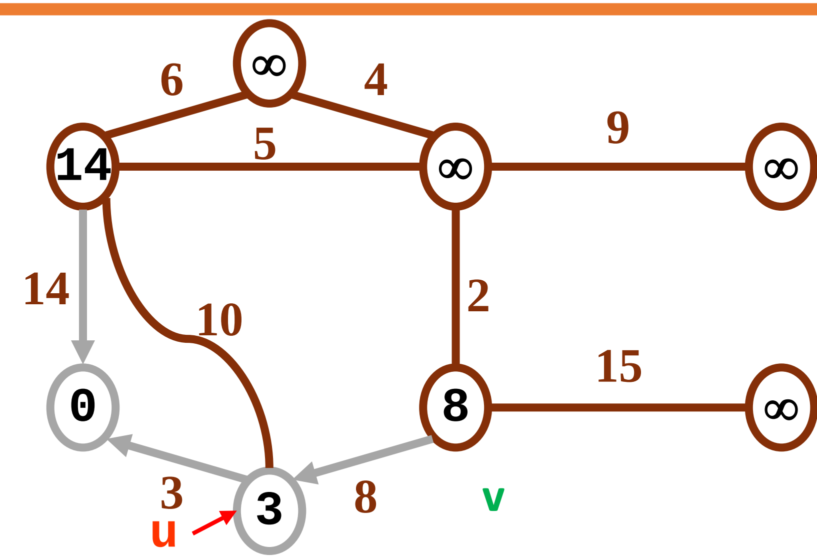
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



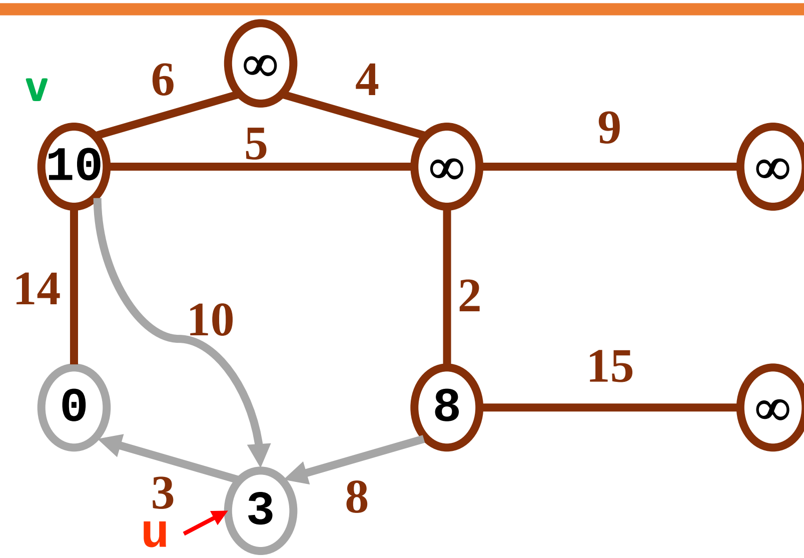
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



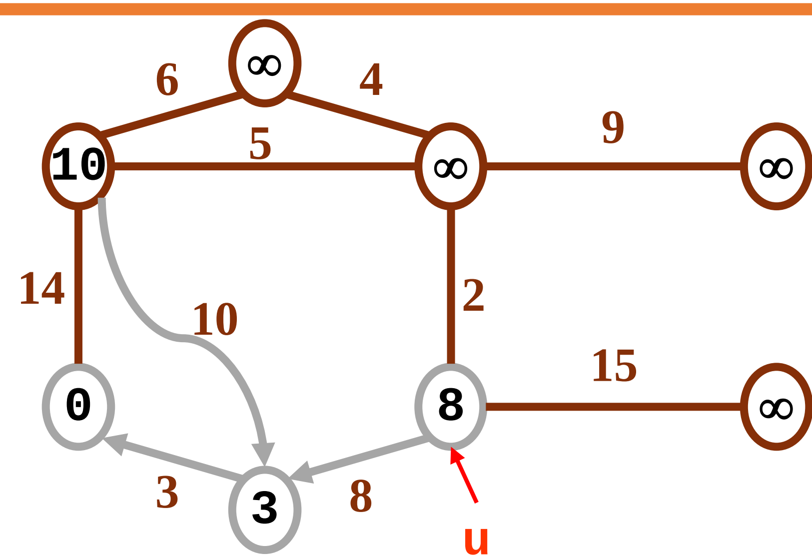
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



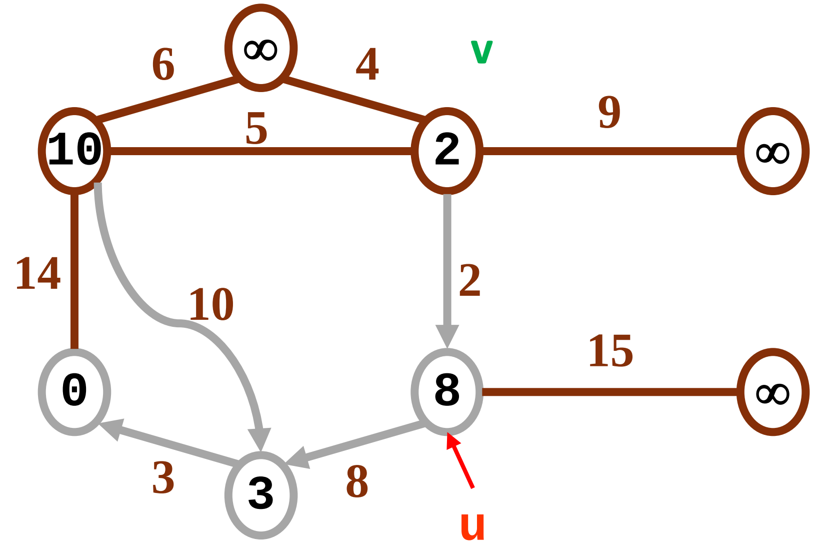
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



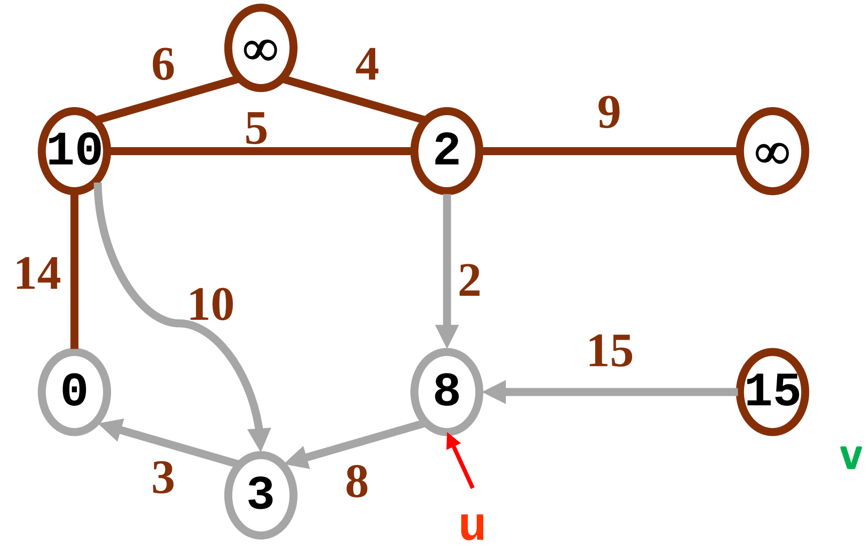
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



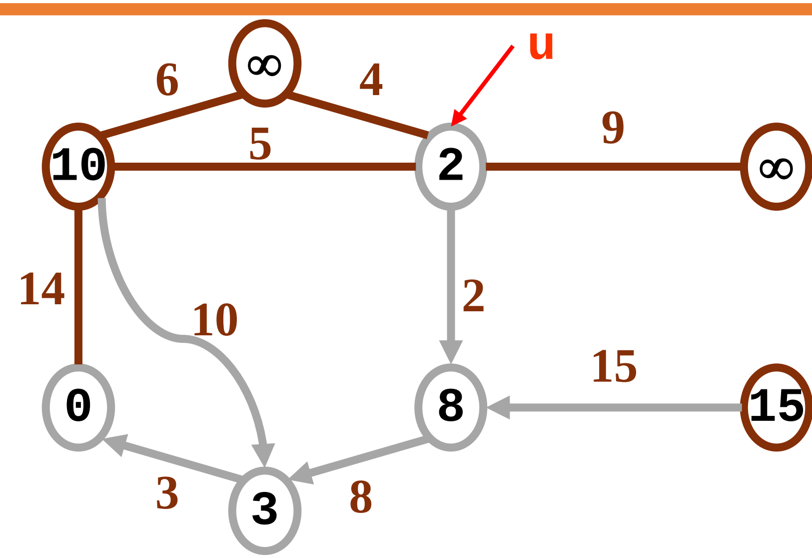
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



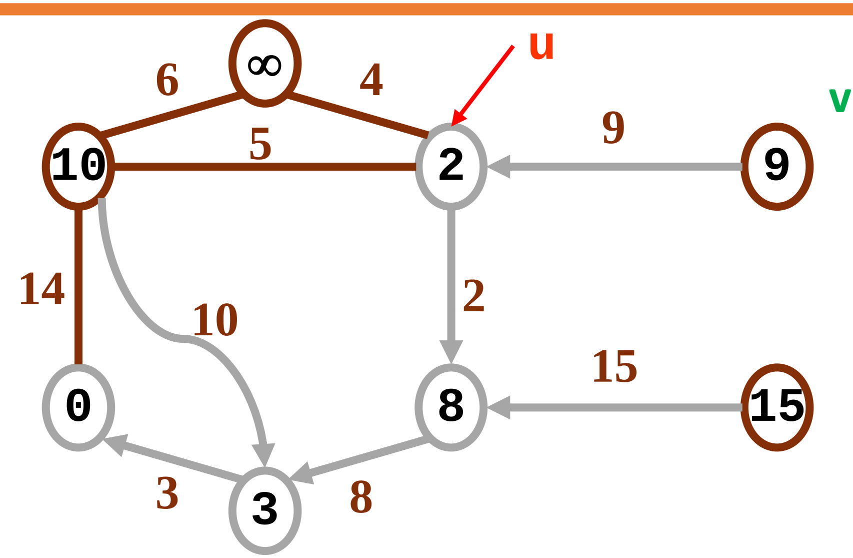
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



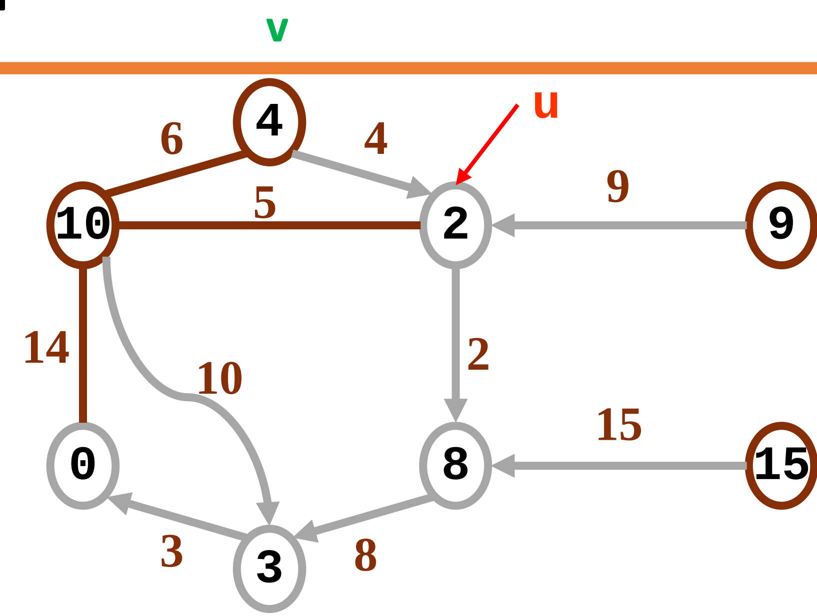
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



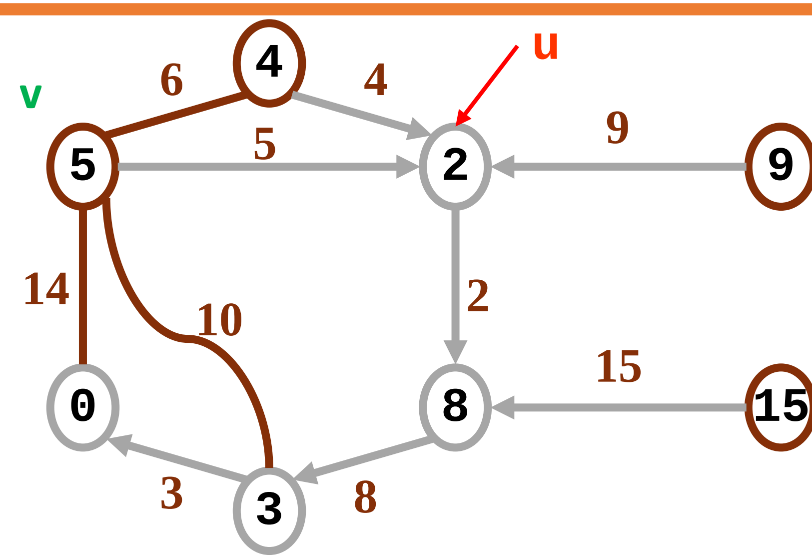
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



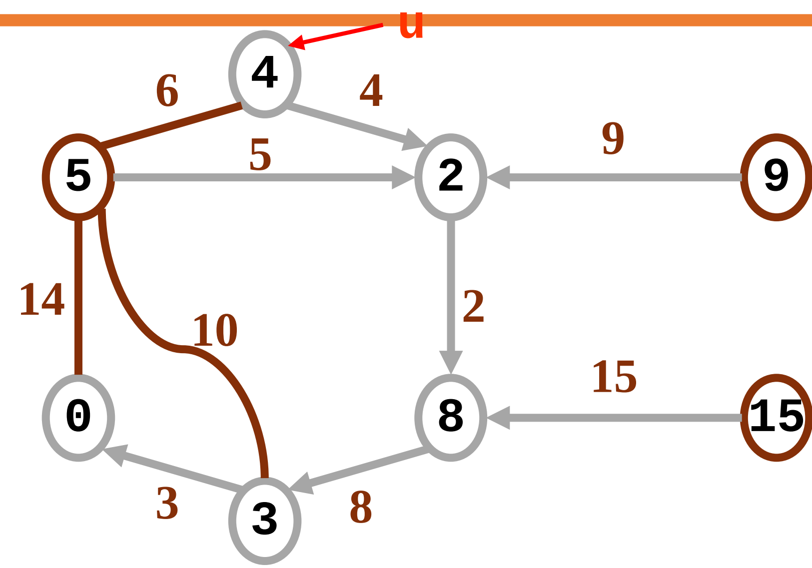
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

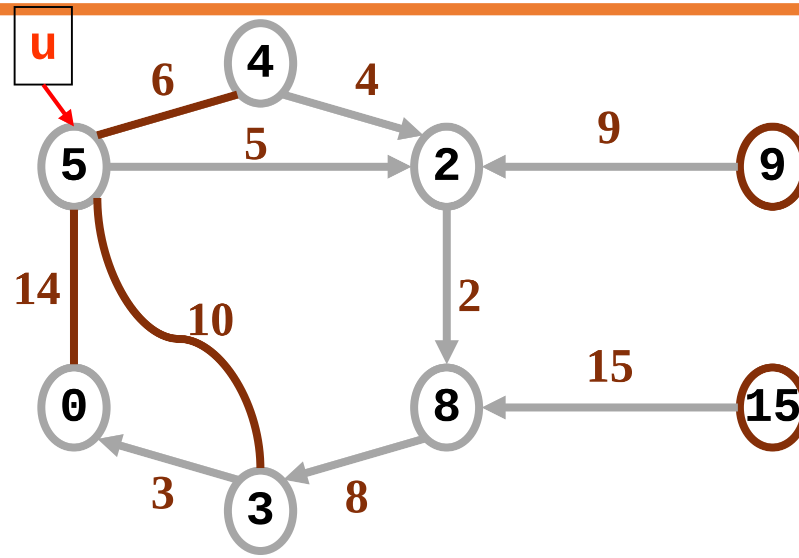


Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



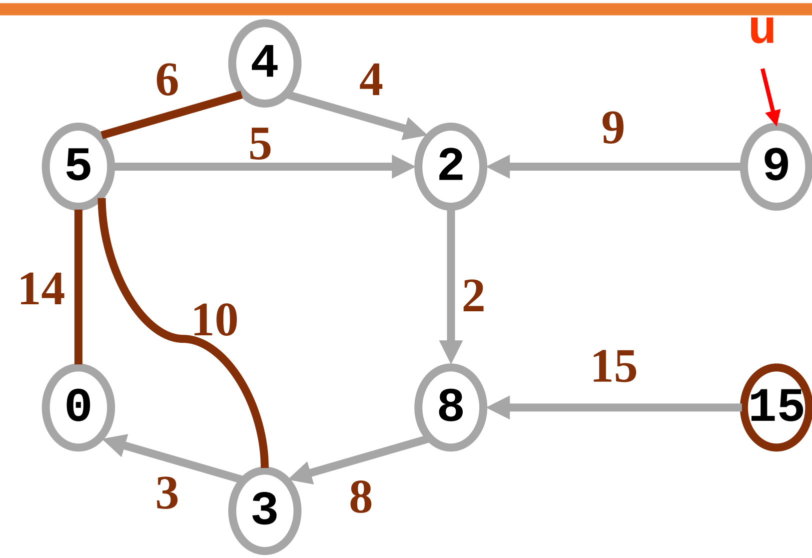
Prim's Algorithm




```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

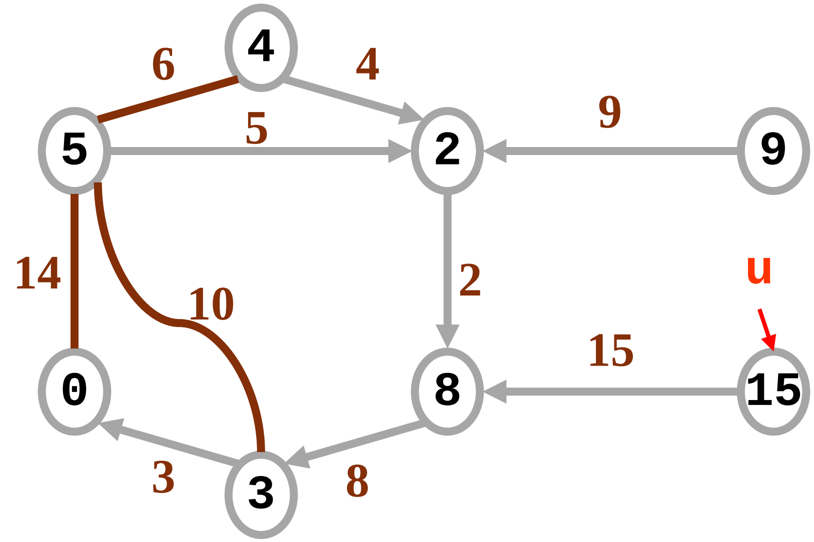
Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Analysis of Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Analysis of Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

Analysis of Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?

How often is DecreaseKey() called?

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?

A: Depends on queue

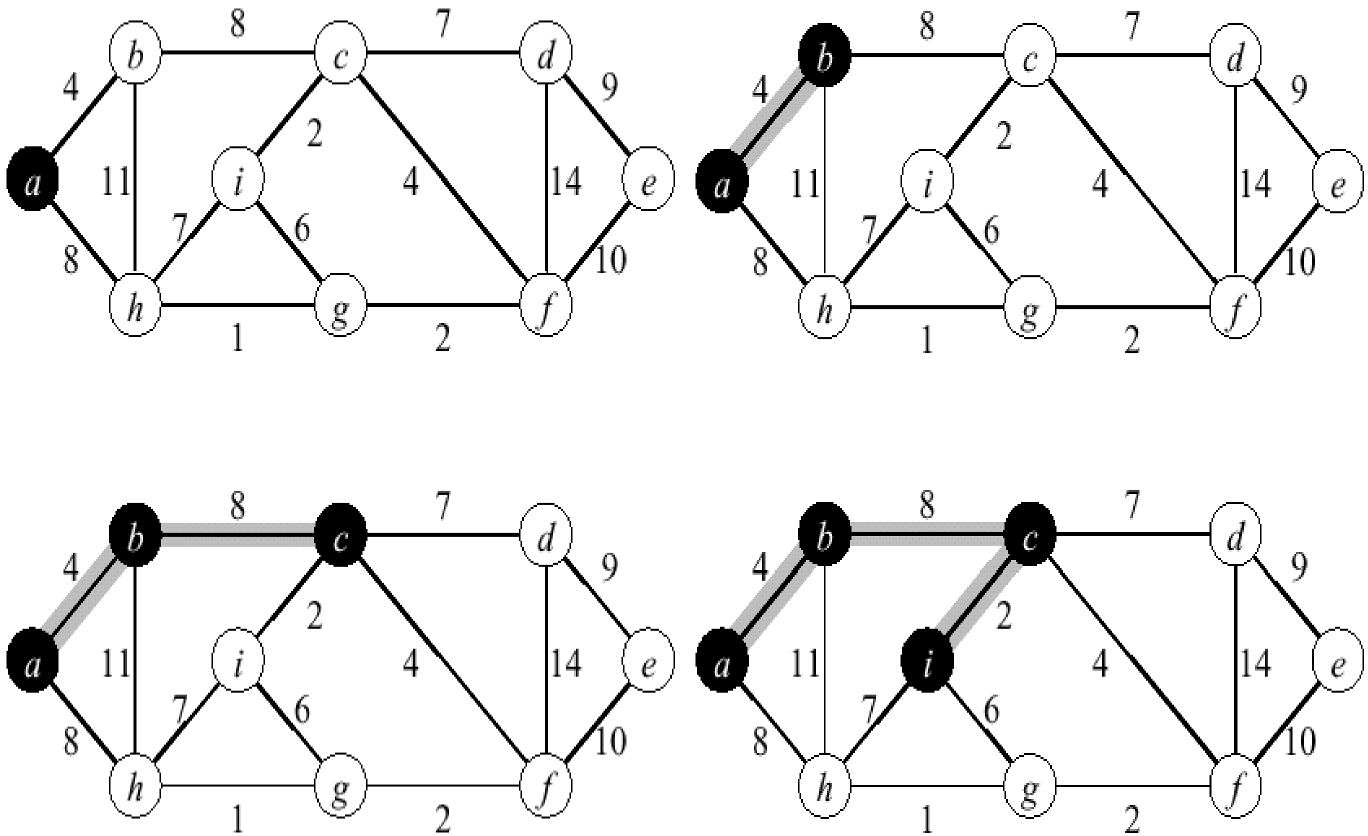
- **binary heap: O(E lg V)**
- **Fibonacci heap: O(V lg V + E)**

Prim's Running Time

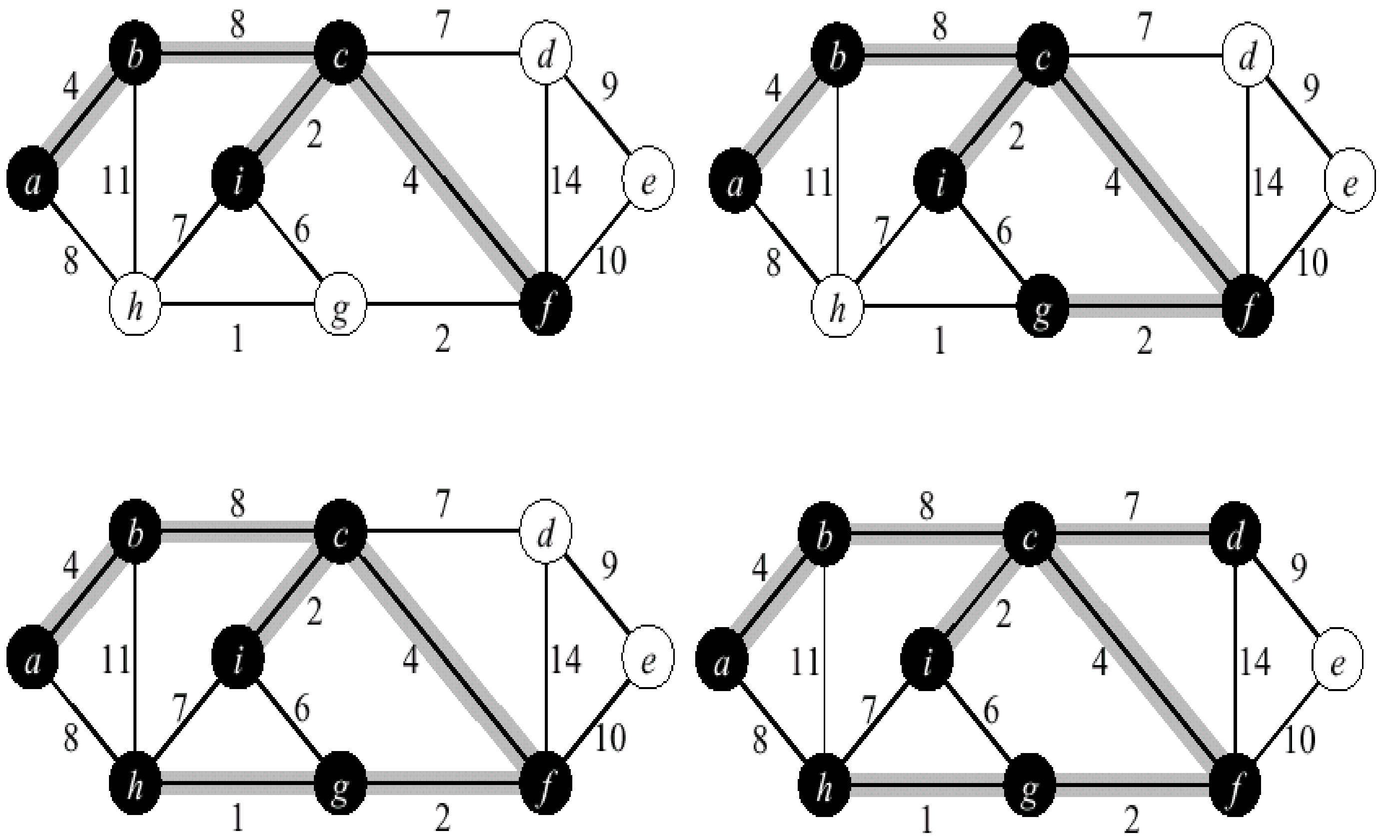
- Time = $|V|\,T(\text{ExtractMin}) + O(E)\,T(\text{ModifyKey})$
- Time = $O(V \lg V + E \lg V) = O(E \lg V)$

Q	T(ExtractMin)	T(DecreaseKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ amortized	$O(V \lg V + E)$

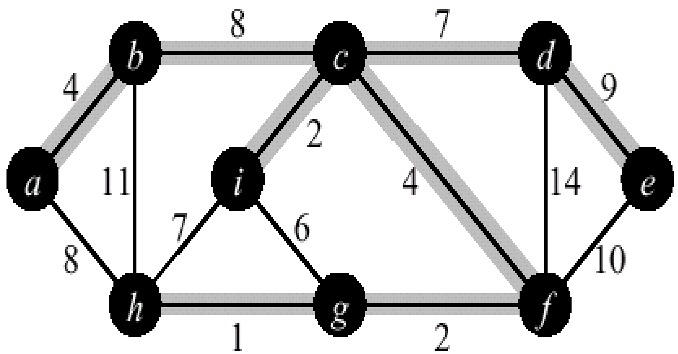
Prim’s Example (2)



Prim’s Example (2)



Prim’s Example (2)



Kruskal's Algorithm

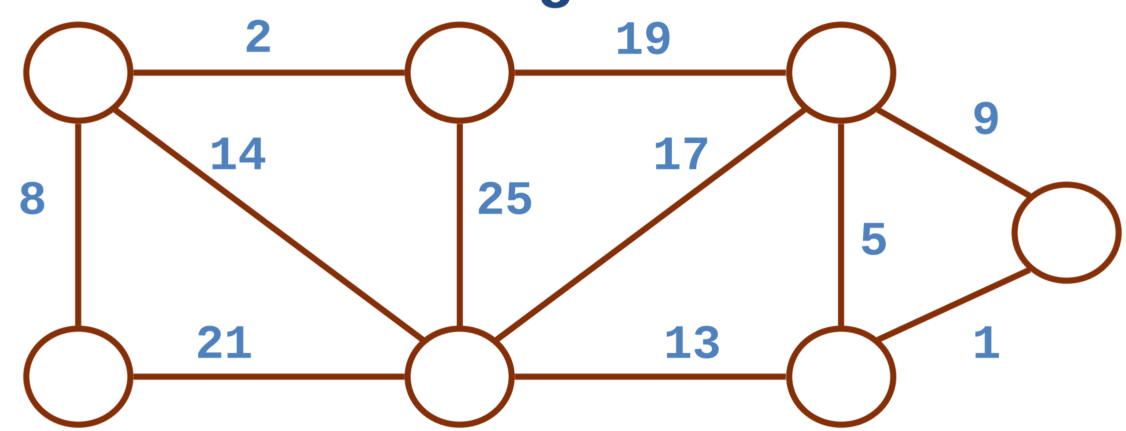
- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains *A* – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees
- We need an Abstract Data Type (ADT) that maintains a partition, i.e.,a collection of disjoint sets
  - MakeSet(S,x): $S \leftarrow S \cup \{\{x\}\}$
  - Union(S_i,S_j): $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(S, x): returns unique $S_i \in S$, where $x \in S_i$

Kruskal’s Algorithm

```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

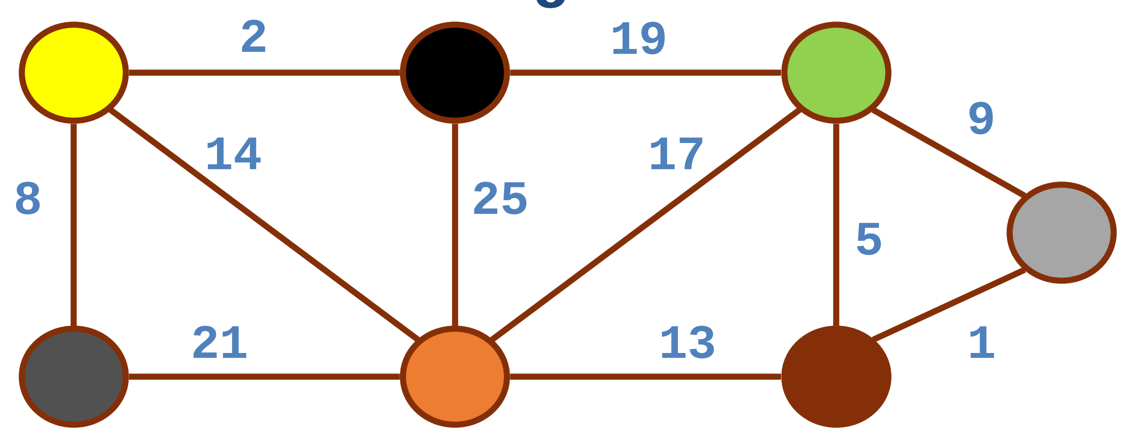
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

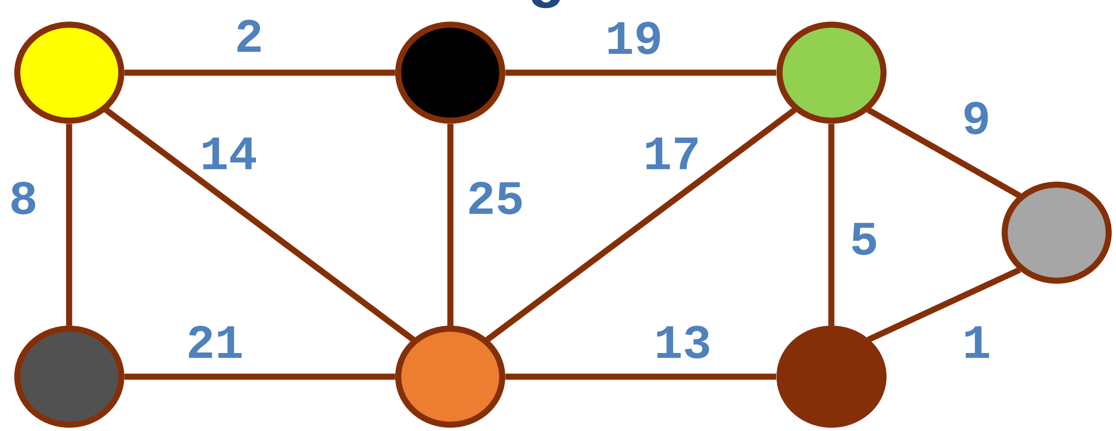
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

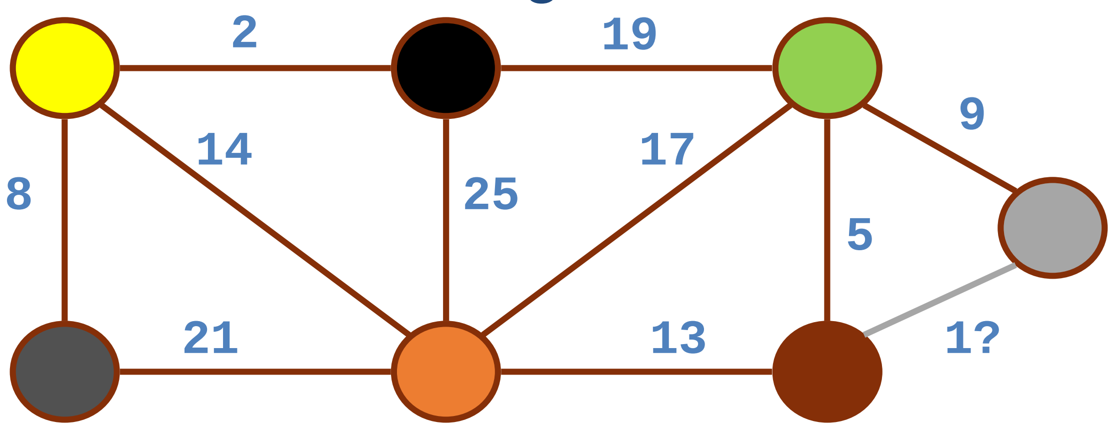
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

Run the algorithm:

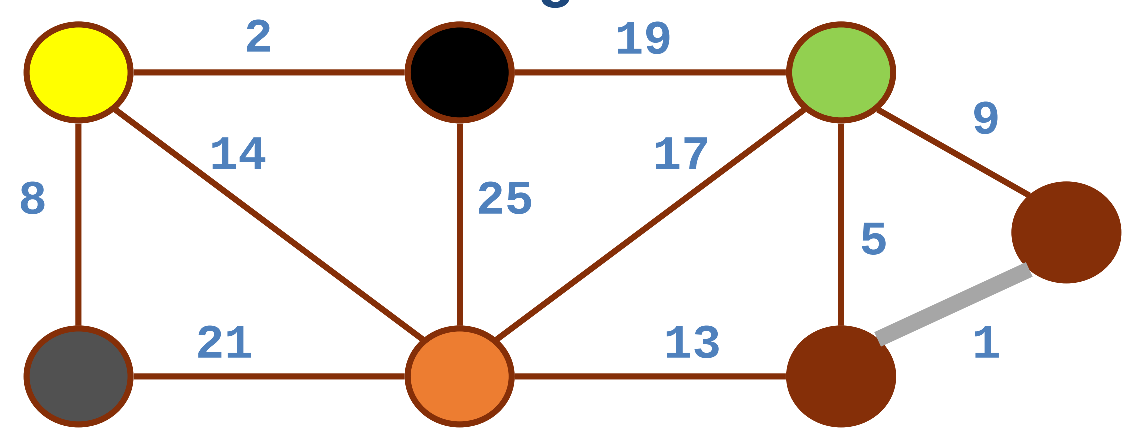



```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

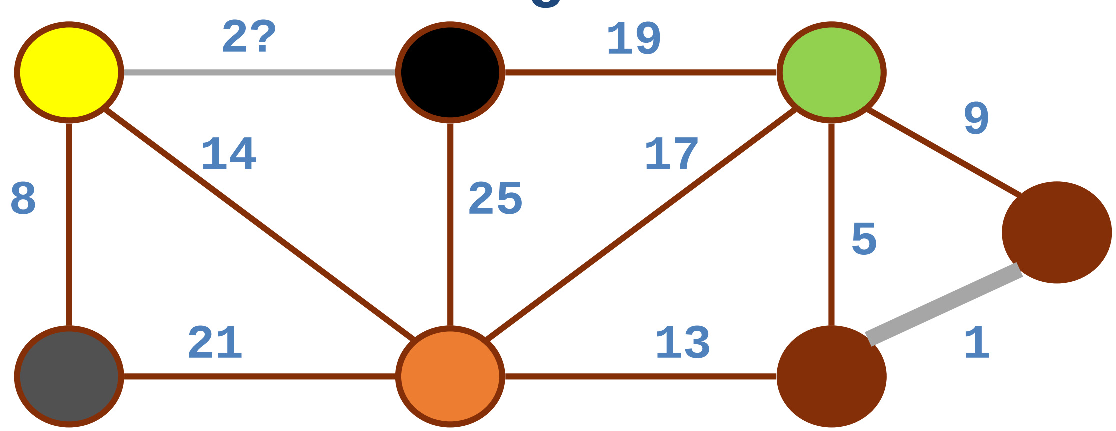
```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

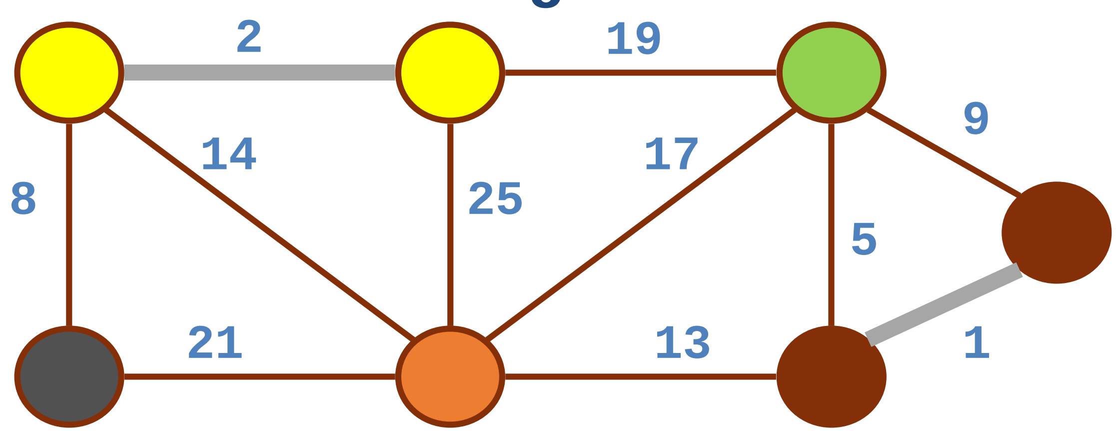
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

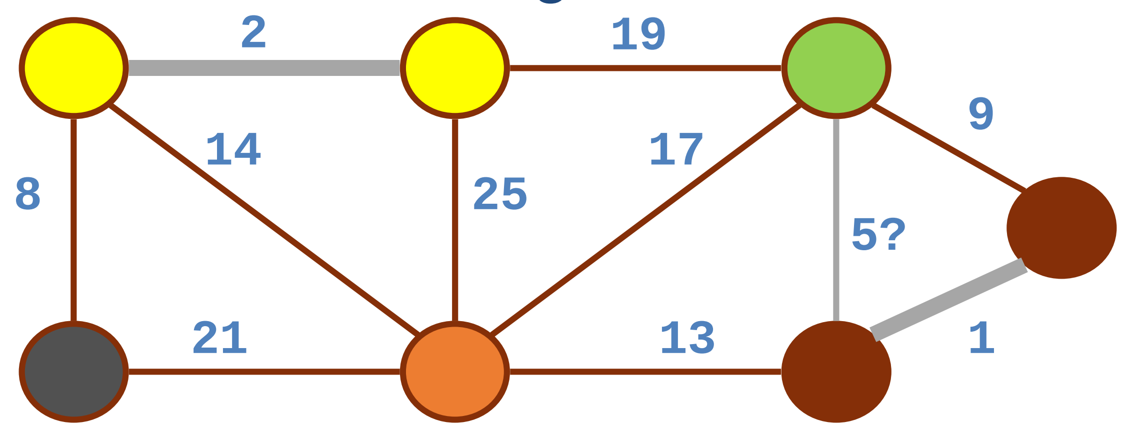
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

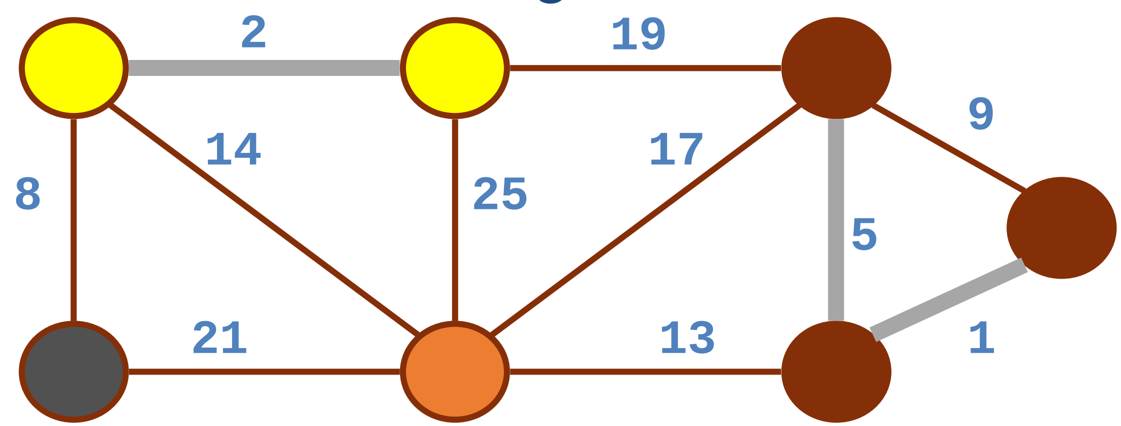
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

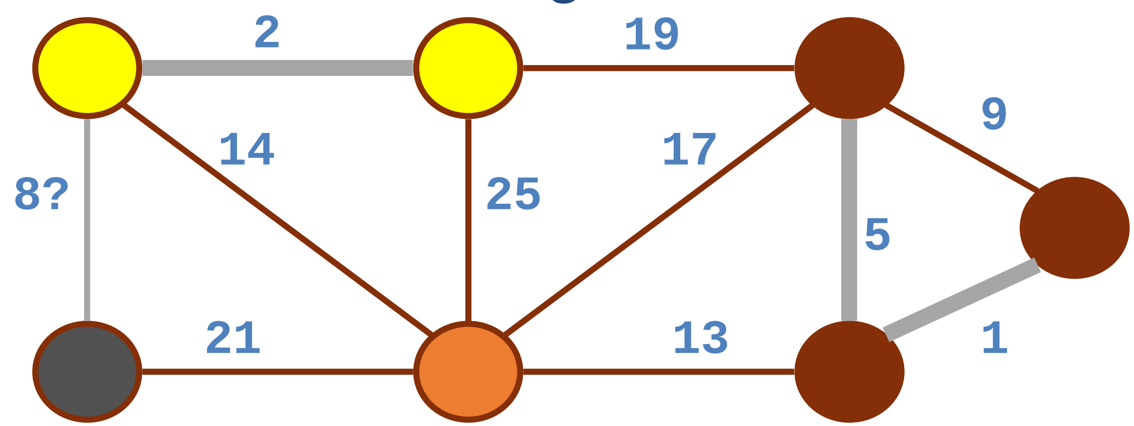
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

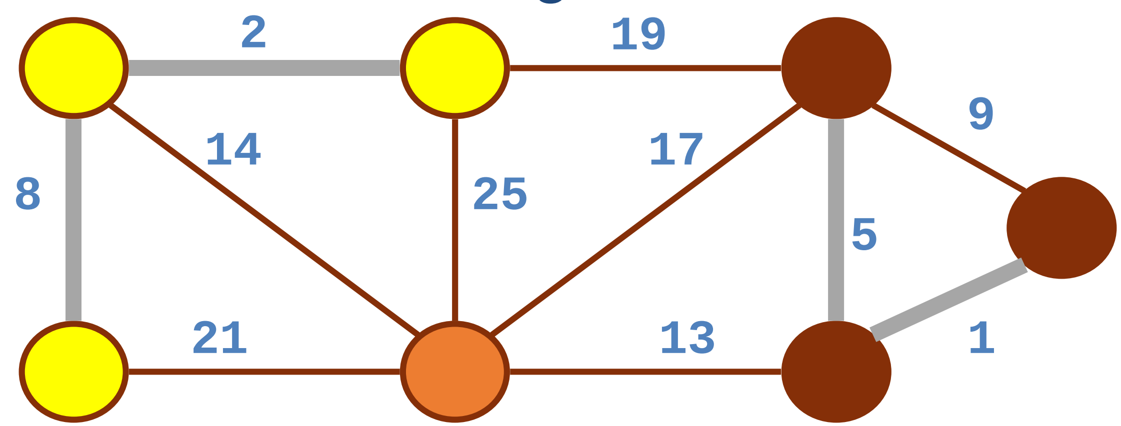
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

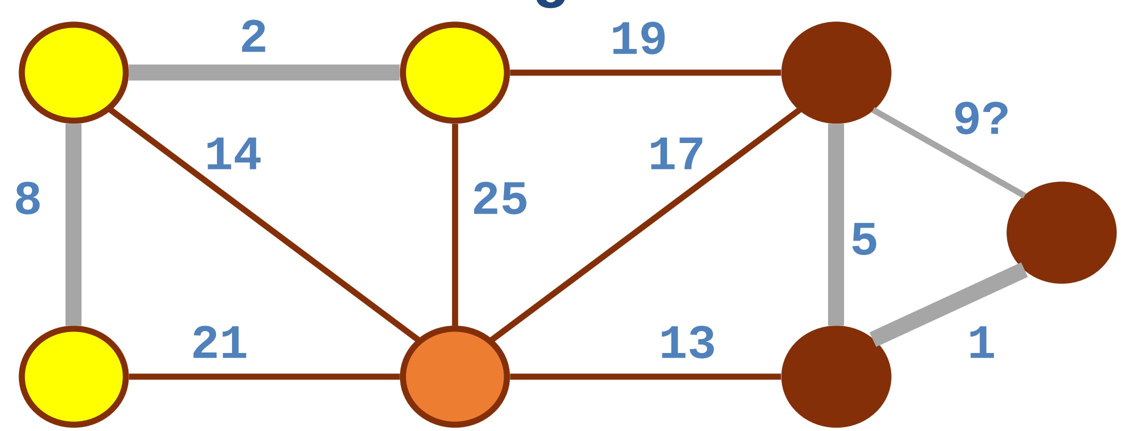
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

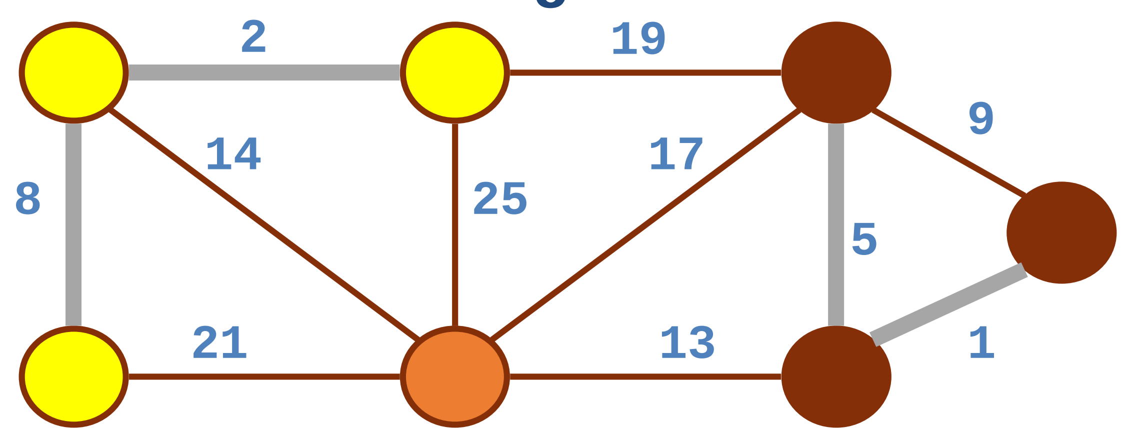



```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

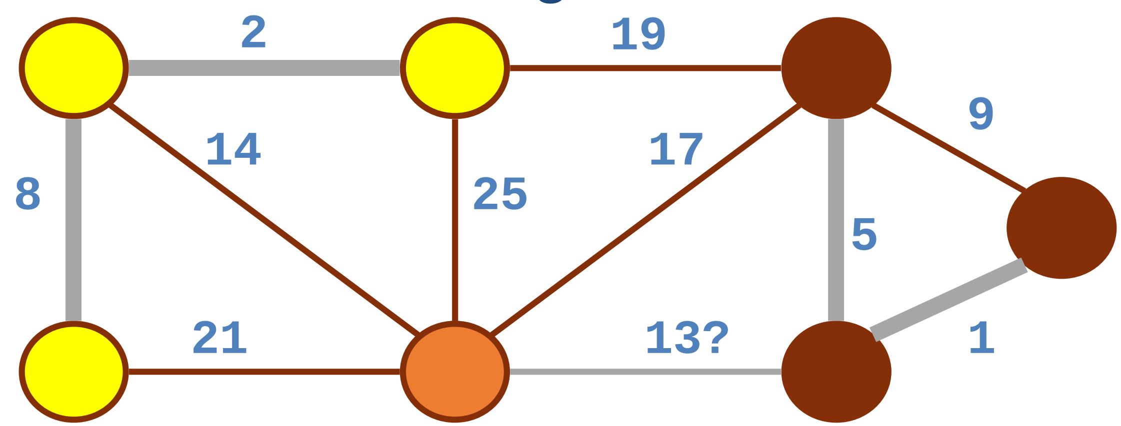
```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

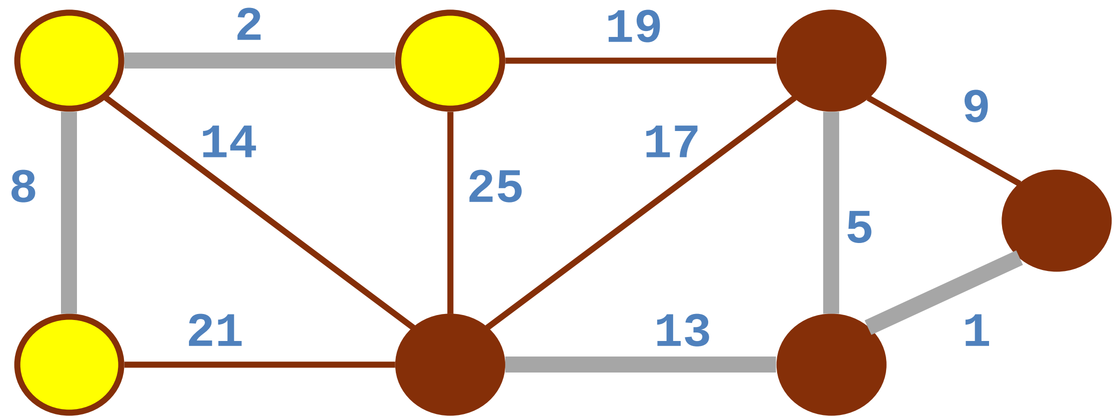
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

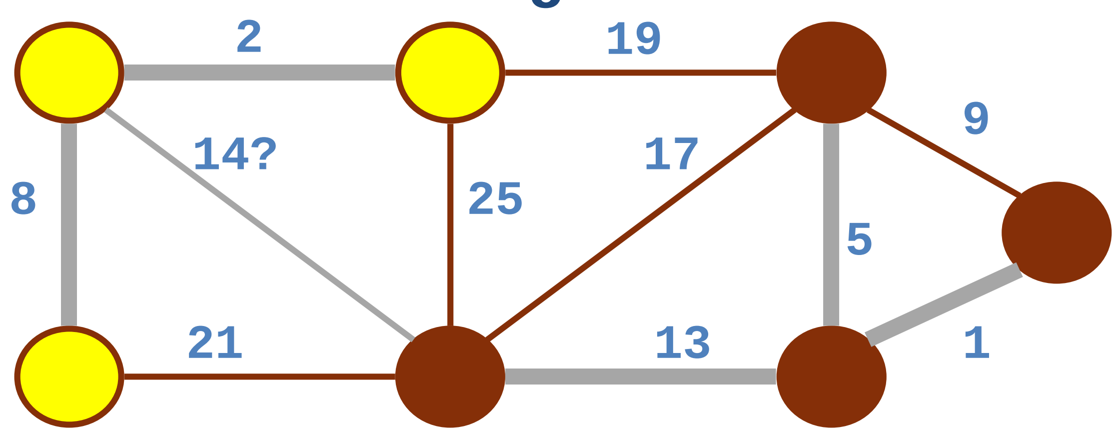
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

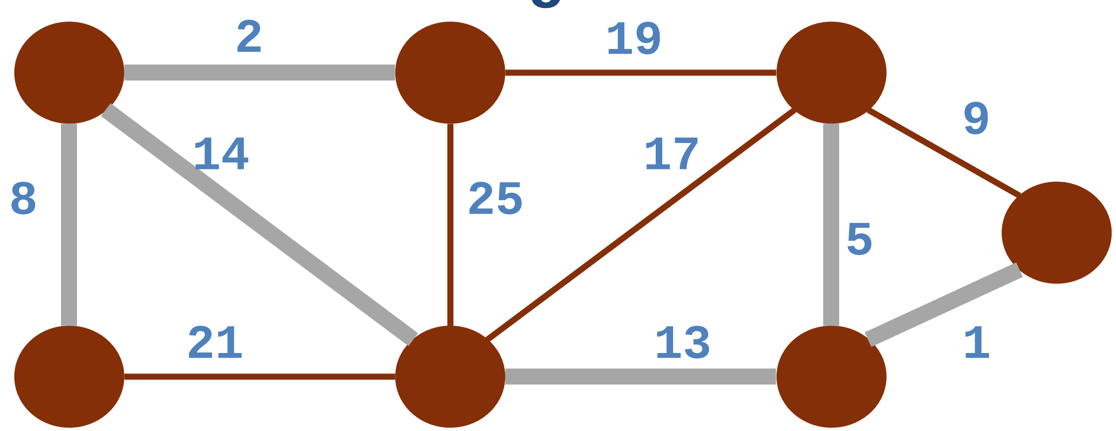
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

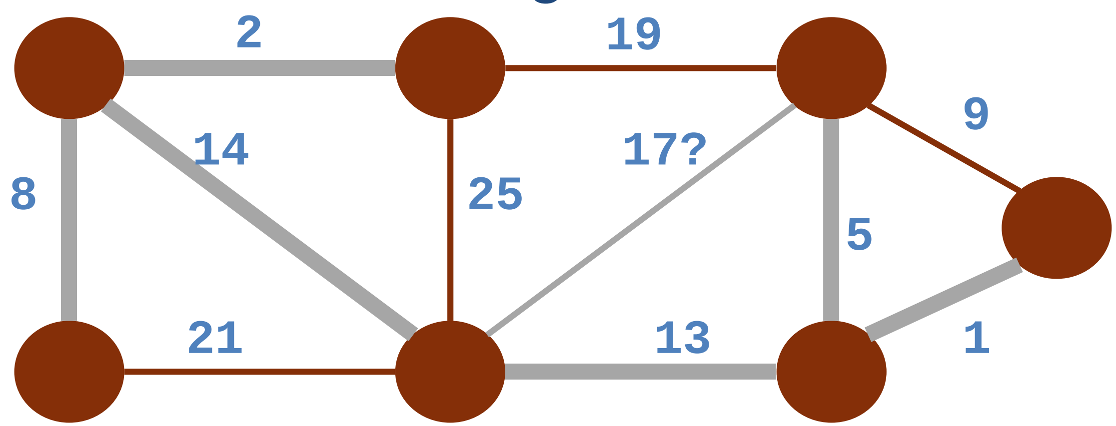
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

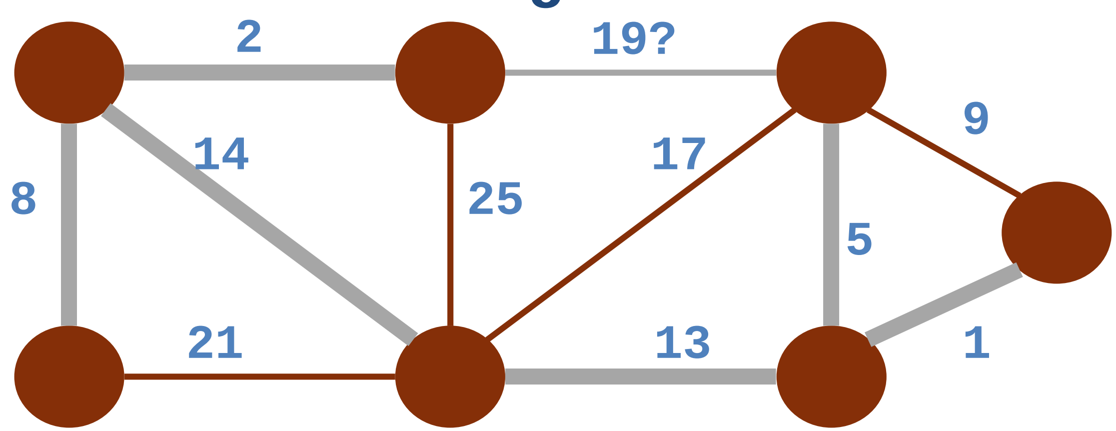
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

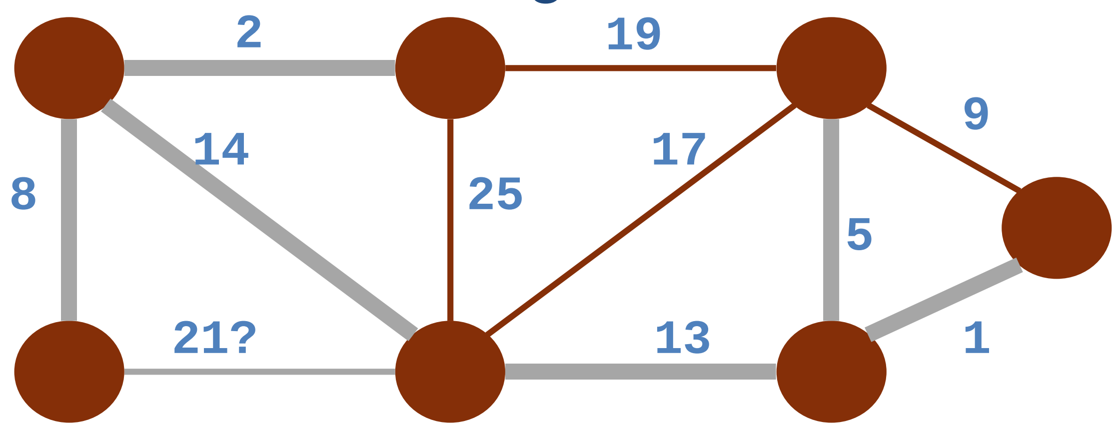
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

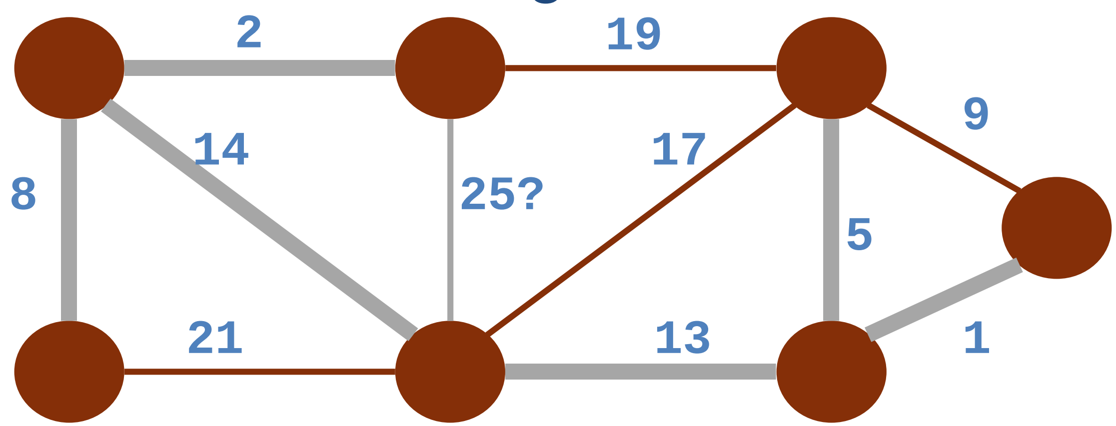
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

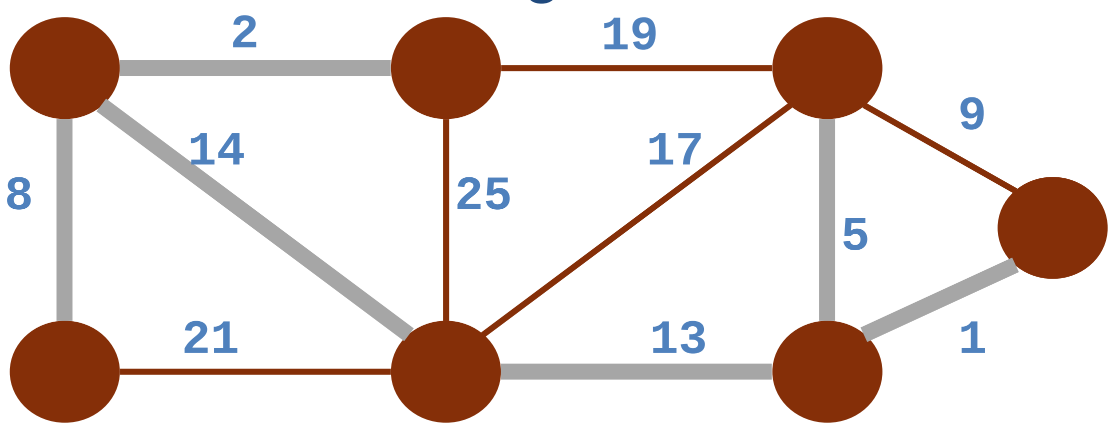
Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

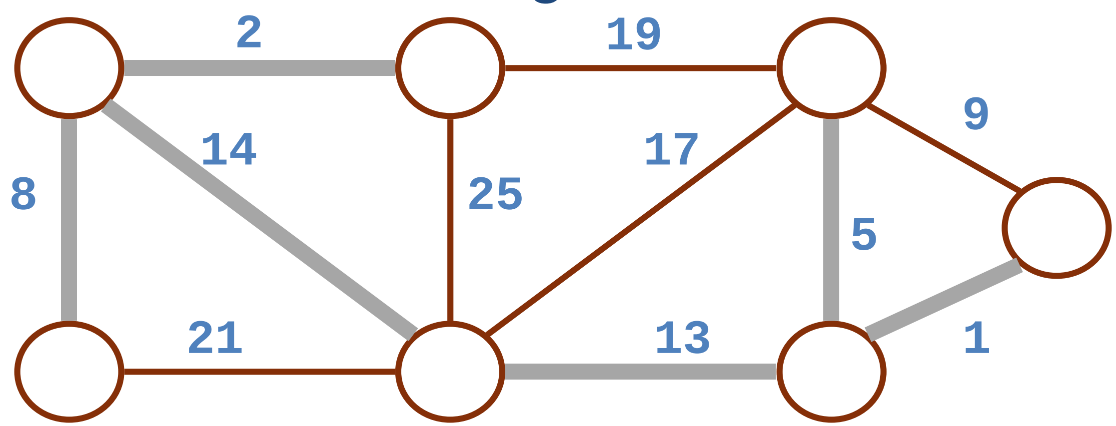
Run the algorithm:




```
Kruskal()
{
   A = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         A = A U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:




```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
    - Assume algorithm is wrong: result is not an MST
    - Then algorithm adds a wrong edge at some point
    - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
    - But algorithm chooses lowest weight edge at each step. Contradiction
- Again, important to be comfortable with cut and paste arguments

Kruskal's Algorithm

What will affect the running time?

```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
    A = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort

O(V) MakeSet() calls

O(E) FindSet() calls

O(E) Union() calls

Kruskal's Algorithm: Running Time

- To summarize:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(E) Union()'s
- Upshot:
  - Best disjoint-set union algorithm makes above 3 operations take $O(E \cdot \alpha(E,V))$, α almost constant
  - Overall thus O(E lg E) = O(E lgV) since $E < V^2$

Kruskal Running Time

- Initialization $O(V)$ time
- Sorting the edges $\Theta(E \lg E) = \Theta(E \lg V)$
- $O(E)$ calls to FindSet
- Union costs
  - Let *t(v)* – the number of times *v* is moved to a new cluster
  - Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: $t(v) \leq \log V$
  - Total time spent doing Union $\sum_{v \in V} t(v) \leq |V| \log |V|$
- Total time: $O(E \lg V)$

Graph Algorithms

Part 3 Shortest Path

CS 325

Shortest-Path Problems

- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
  - Unweighted shortest-paths – BFS.

Shortest Paths: Applications

- Flying times between cities
- Distance between street corners
- Cost of doing an activity
  - Vertices are states
  - Edge weights are costs of moving between states

Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function W: $E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

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

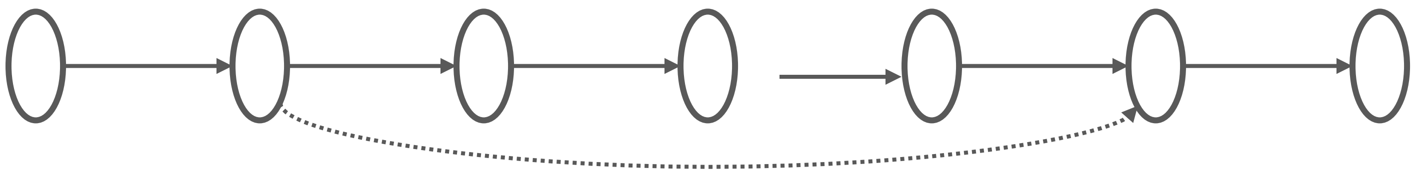
- Shortest path = a path of the minimum weight

Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" = minimum weight
  - Weight of path is sum of edges

Shortest Path Properties

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

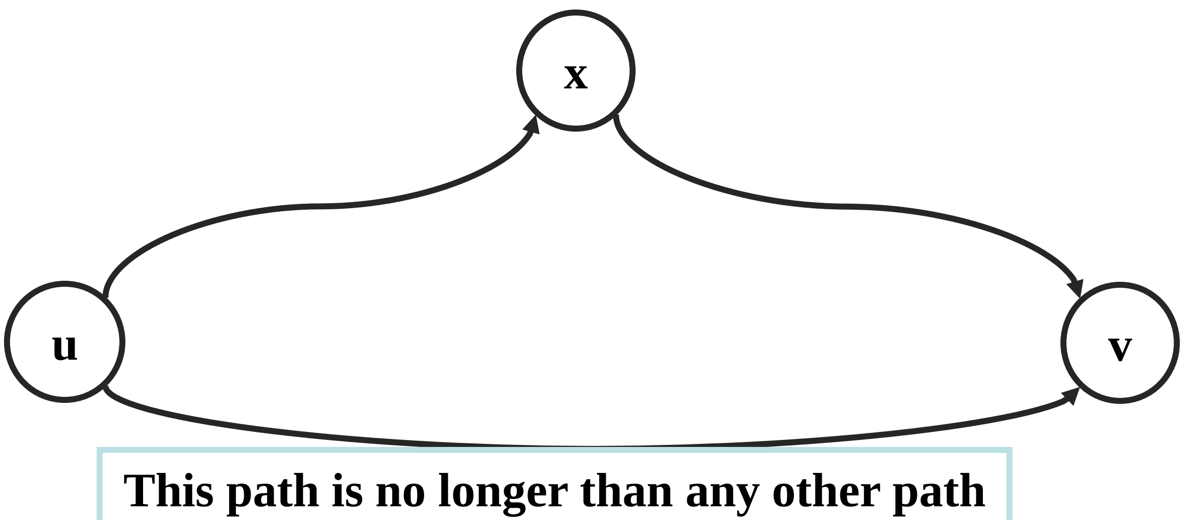


Proof: suppose some subpath is not a shortest path

- 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

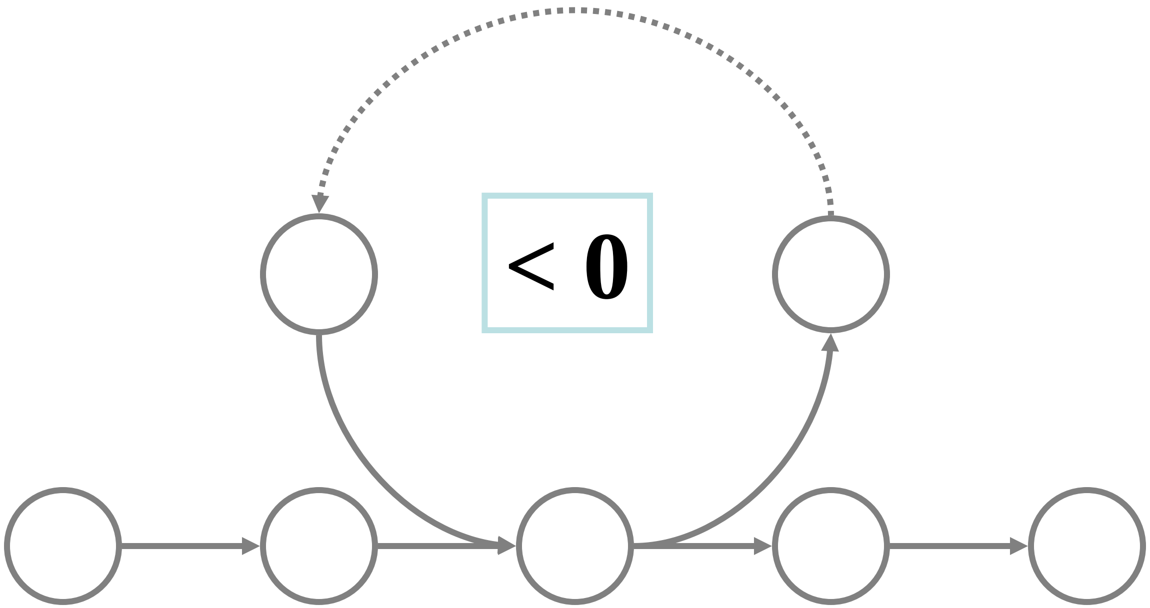
Shortest Path Properties

- Define $\delta(u,v)$ to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$



Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist



Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small “lengths” would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)

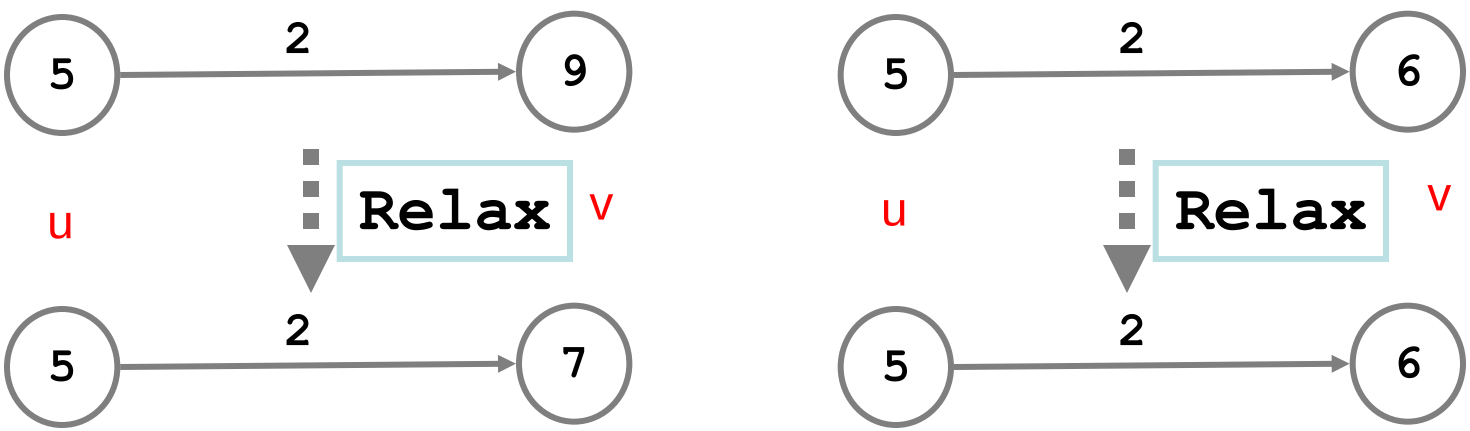
  Any shortest-path in graph G can be no longer than n–1 edges, where n is the number of vertices

Relaxation

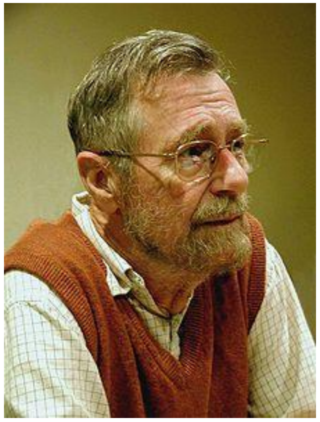
A key technique in shortest path algorithms is *relaxation*

Idea: for all *v*, maintain upper bound d[*v*] on δ(s,v)

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



Edsger Dijkstra



"Computer Science is no more about computers than astronomy is about telescopes."

- May 11, 1930 – August 6, 2002

- Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science.

- The Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000

- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.

Dijkstra's algorithm

<u>Dijkstra's algorithm</u> - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

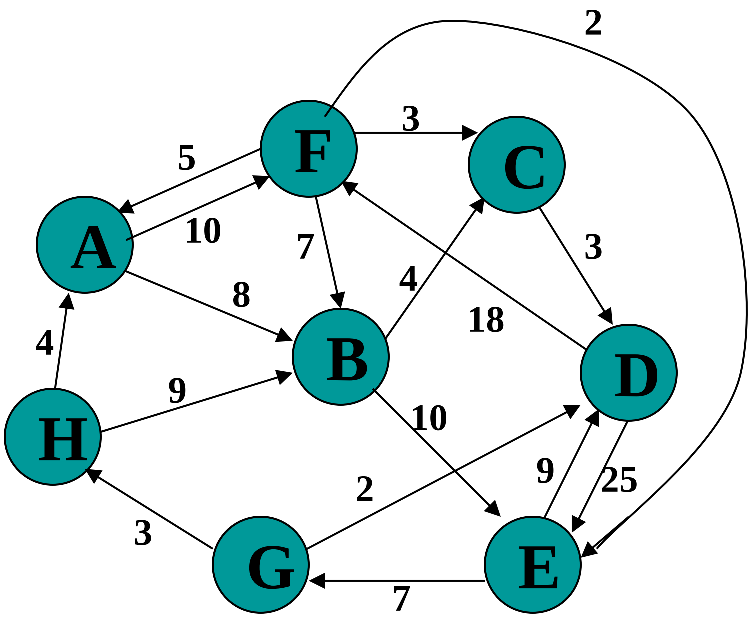
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's Algorithm - SSSP-Dijkstra

Dijkstra(G, w, s)
    InitializeSingleSource(G, s)
    $S \leftarrow \emptyset$
    $Q \leftarrow V[G]$
    while $Q \neq 0$ **do**
        u ← *ExtractMin*(Q)
        $S \leftarrow S \cup \{u\}$
        for $v \in$ *Adj*[u] **do**
            Relax(u,v,w)

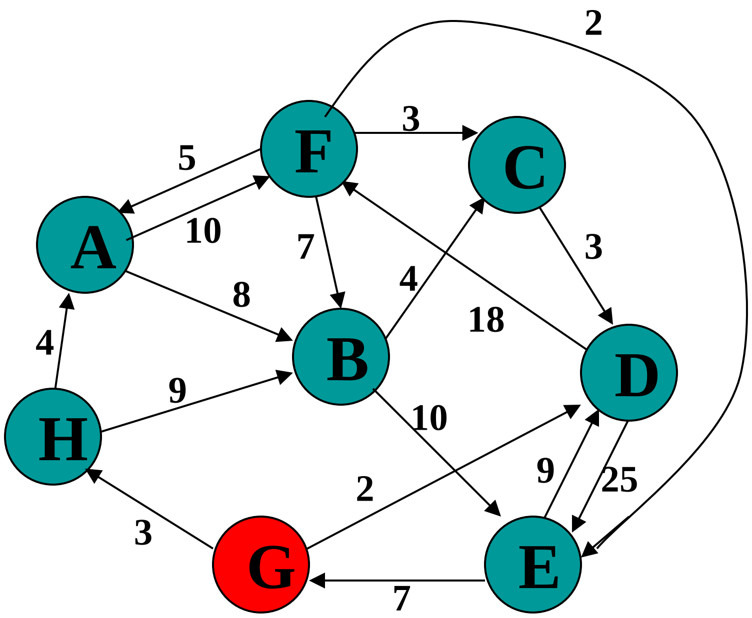
InitializeSingleSource(G, s)
    for $v \in V[G]$ **do**
        $d[v] \leftarrow \infty$
        $p[v] \leftarrow 0$
    $d[s] \leftarrow 0$

Example



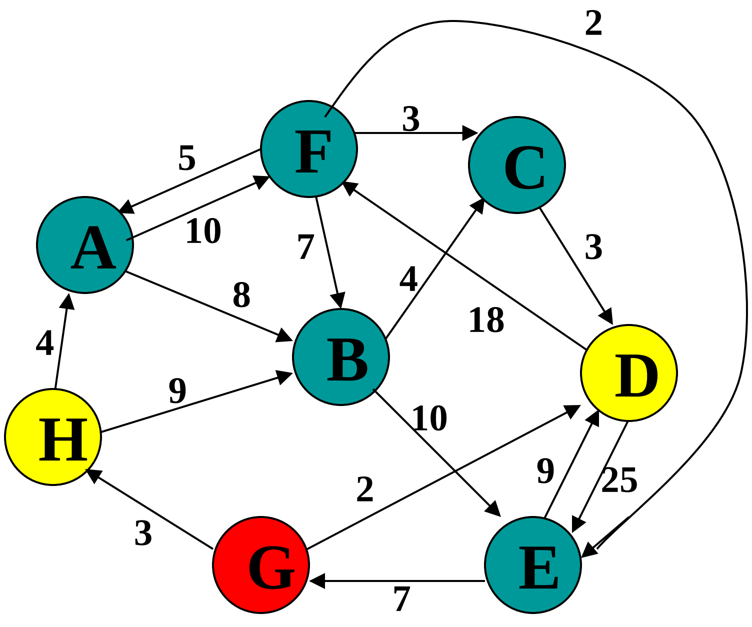
Initialize array

	S	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–



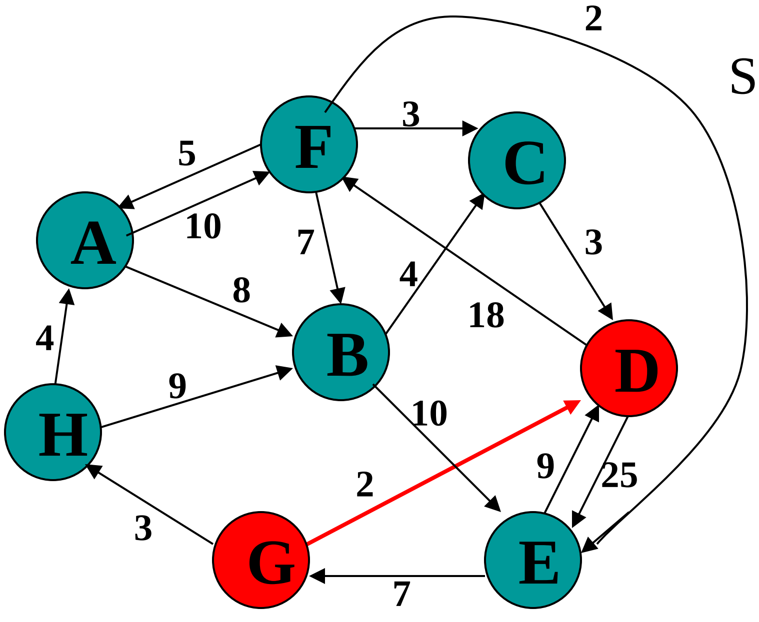
Start with G

	S	d_v	p_v
A			
B			
C			
D			
E			
F			
G	T	0	–
H			



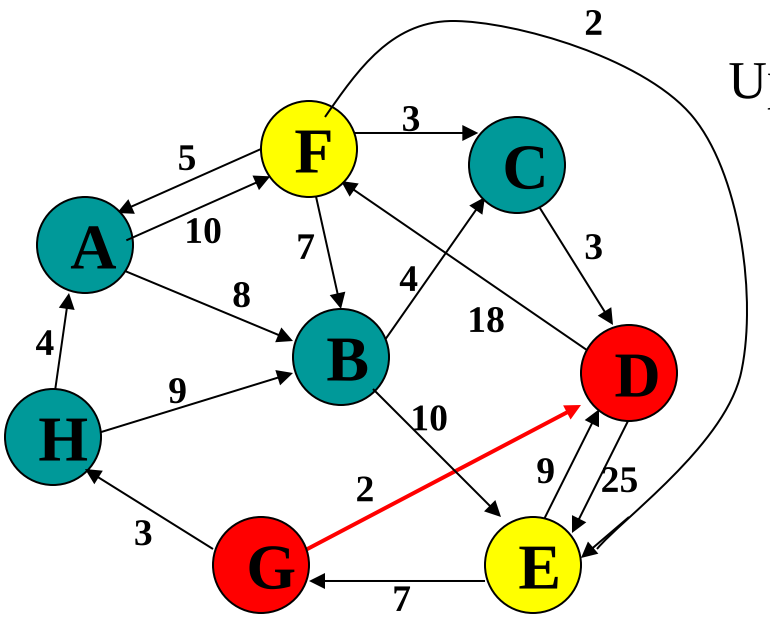
Update unselected nodes

	S	d_v	p_v
A			
B			
C			
D		2	G
E			
F			
G	T	0	–
H		3	G



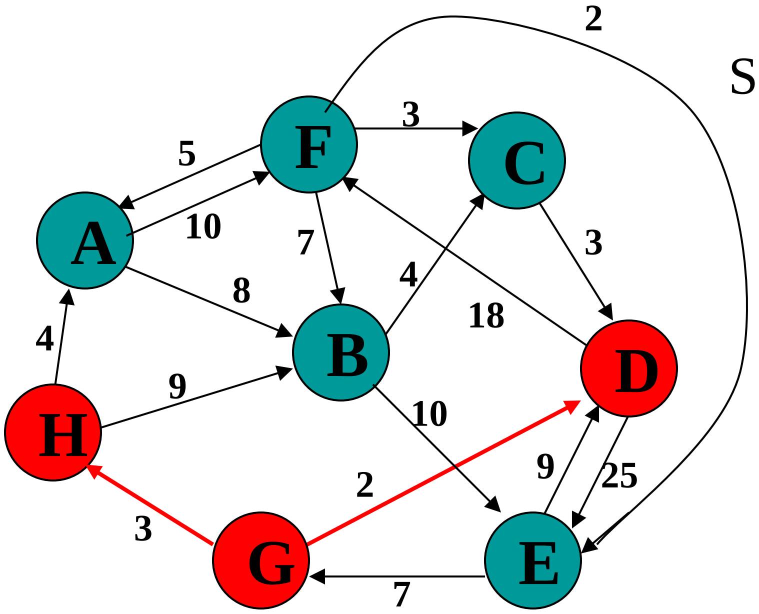
Select minimum distance

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E			
F			
G	T	0	–
H		3	G



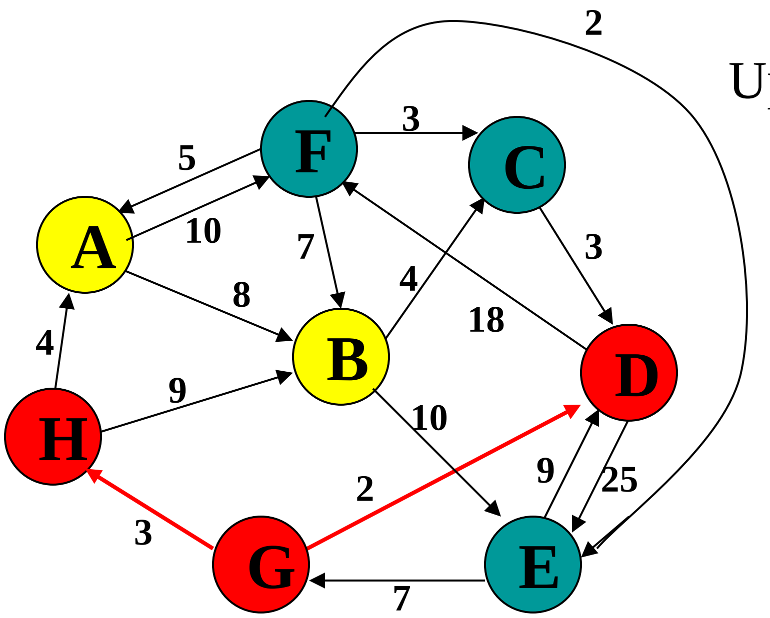
Update unselected nodes

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H		3	G



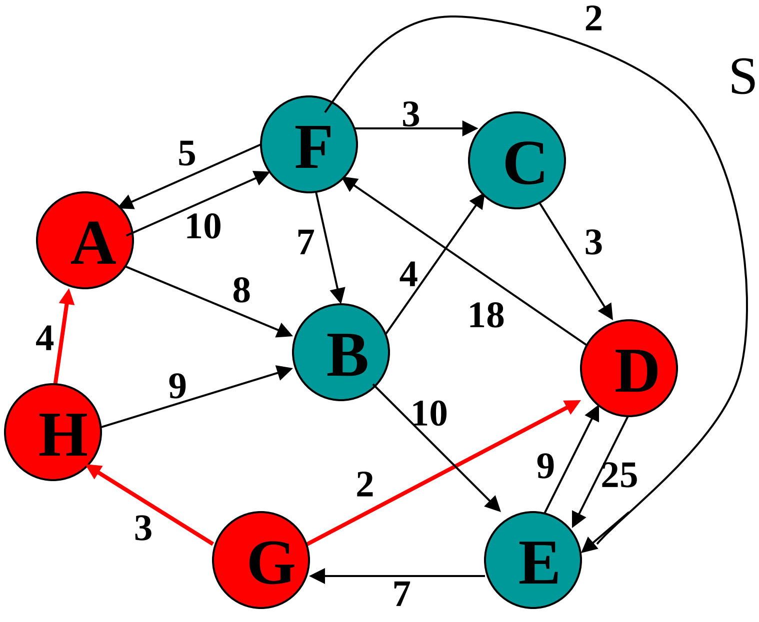
Select minimum distance

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



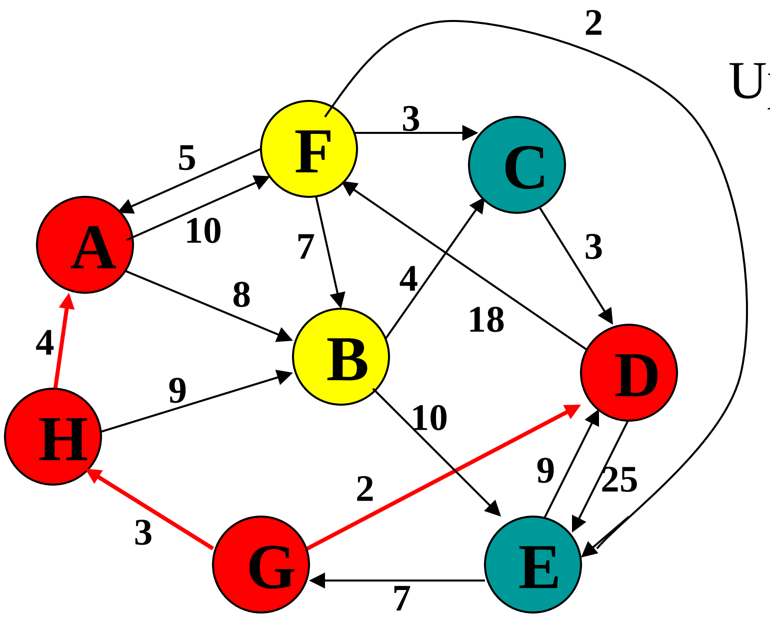
Update unselected nodes

	S	d_v	p_v
A		7	H
B		12	H
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



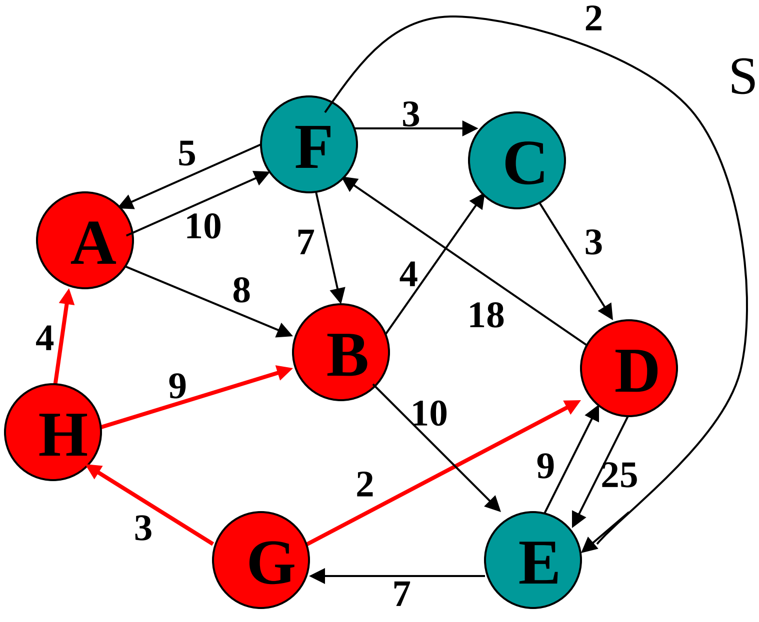
Select minimum distance

	S	d_v	p_v
A	T	7	H
B		12	H
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



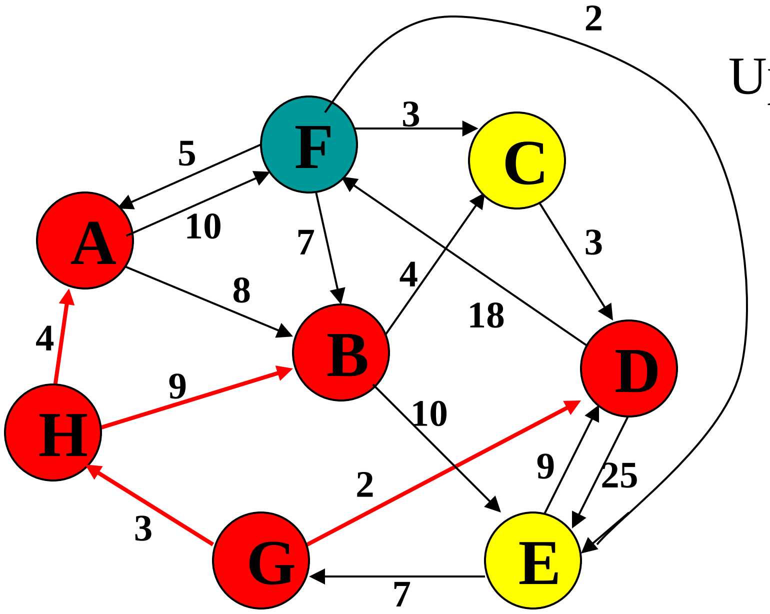
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B		12	H
C			
D	T	2	G
E		27	D
F		17	A
G	T	0	–
H	T	3	G



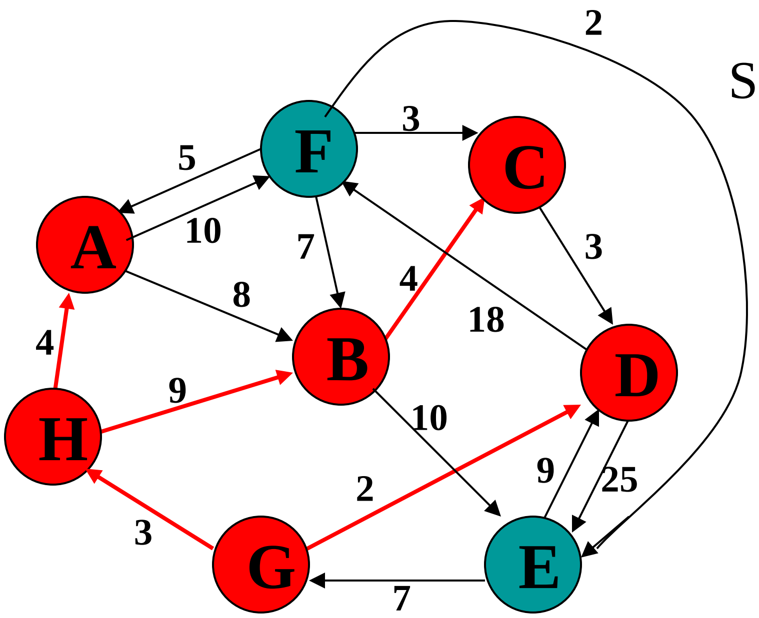
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C			
D	T	2	G
E		27	D
F		17	A
G	T	0	–
H	T	3	G



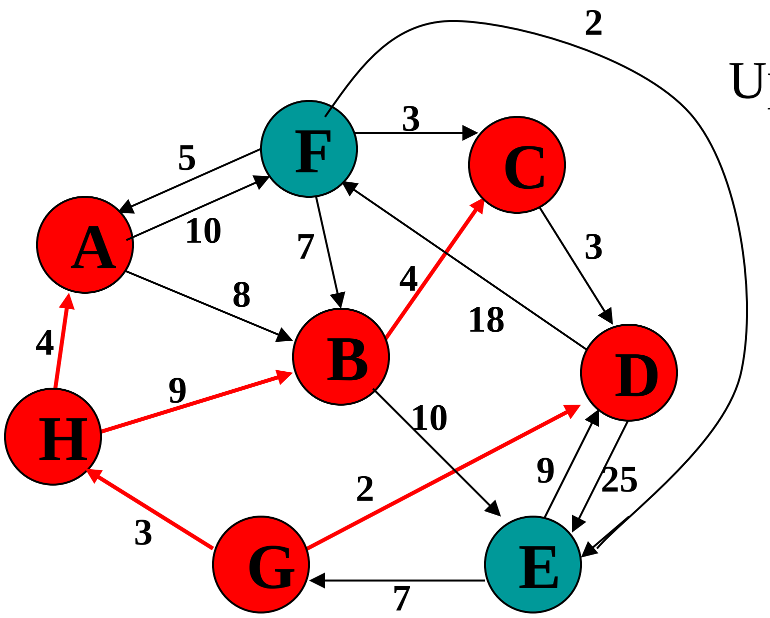
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C		16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



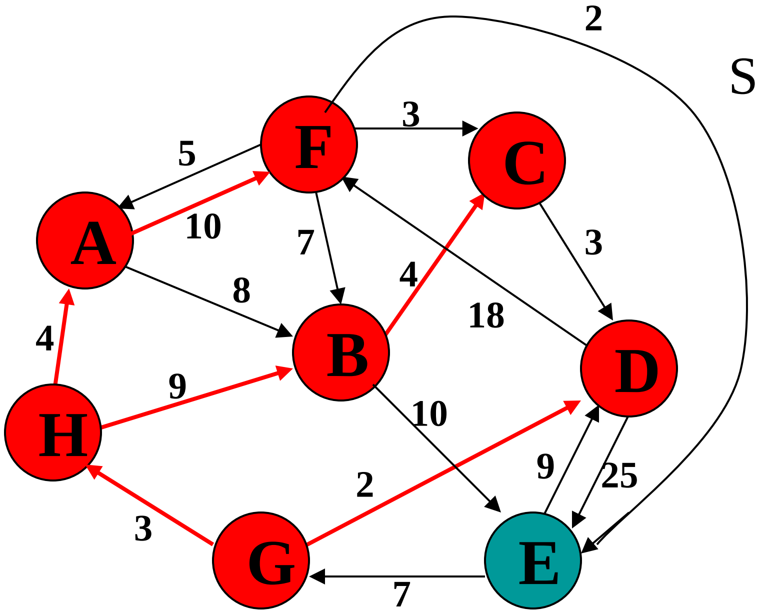
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



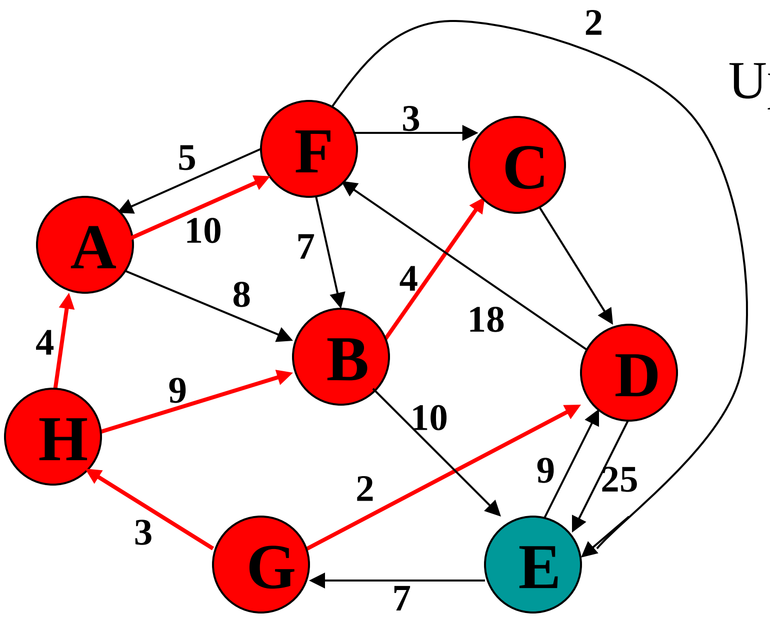
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



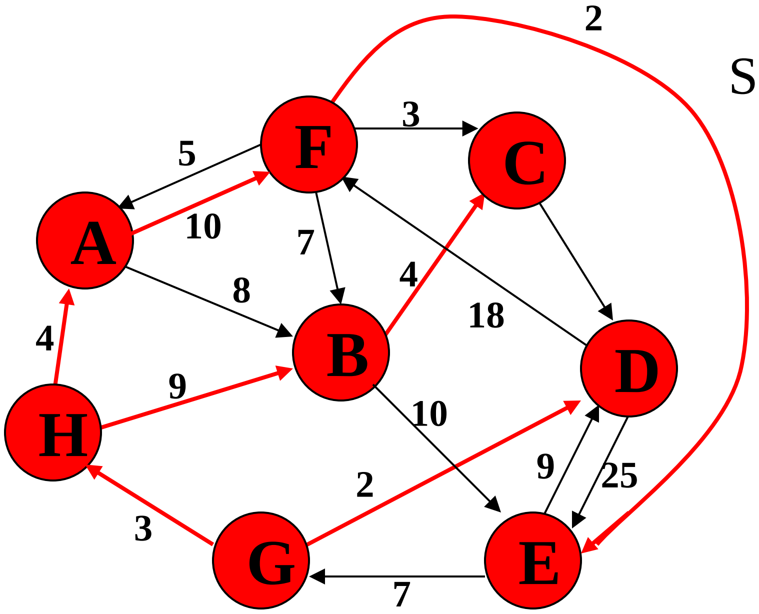
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F	T	17	A
G	T	0	–
H	T	3	G



Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		19	F
F	T	17	A
G	T	0	–
H	T	3	G



Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E	T	19	F
F	T	17	A
G	T	0	–
H	T	3	G

Done

Order of Complexity

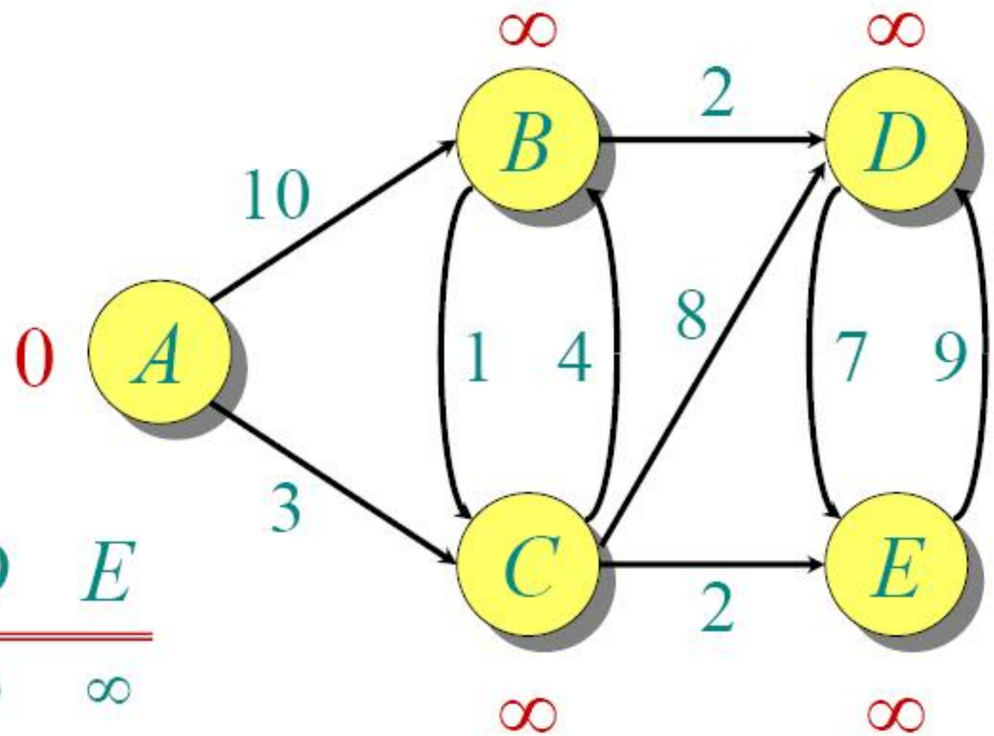
- Analysis
  - findMin() takes O(V) time
  - outer loop iterates (V-1) times
  - ➔ $O(V^2)$ time
- Optimal for dense graphs, i.e., $|E| = O(V^2)$
- Suboptimal for sparse graphs, i.e., $|E| = O(V)$

All-Pairs Shortest Paths

- One option: run Dijktra's algorithm |V| times → $O(V^3)$ time
- There is a more efficient $O(V^3)$ time algorithm

Dijkstra Another Example

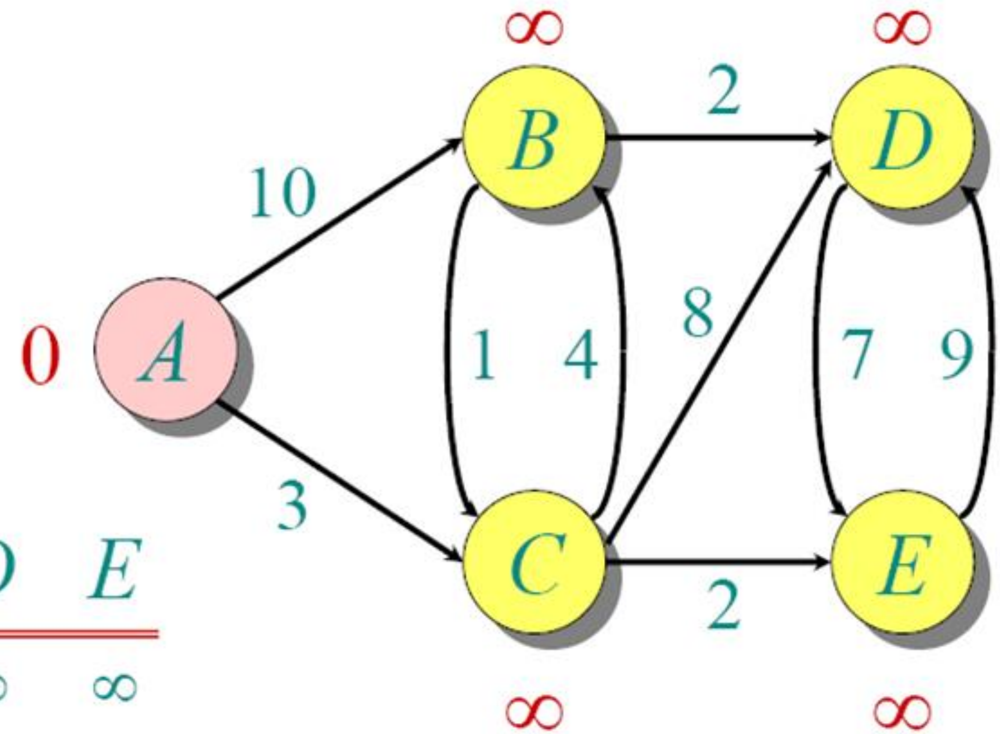
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

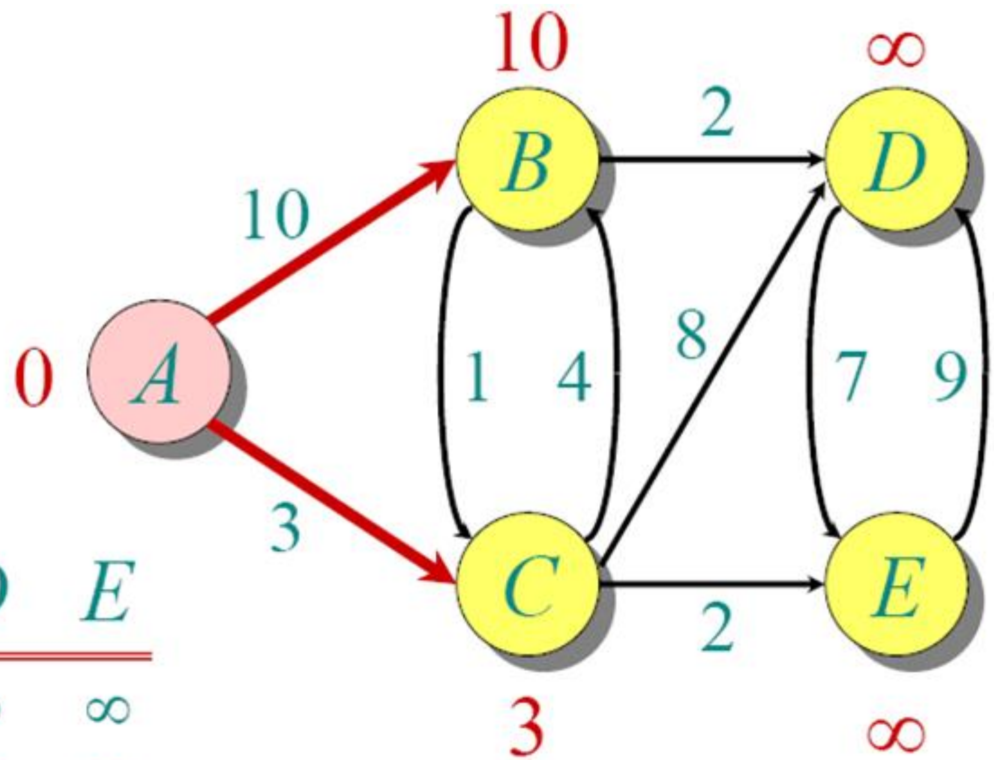
S: {}

Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

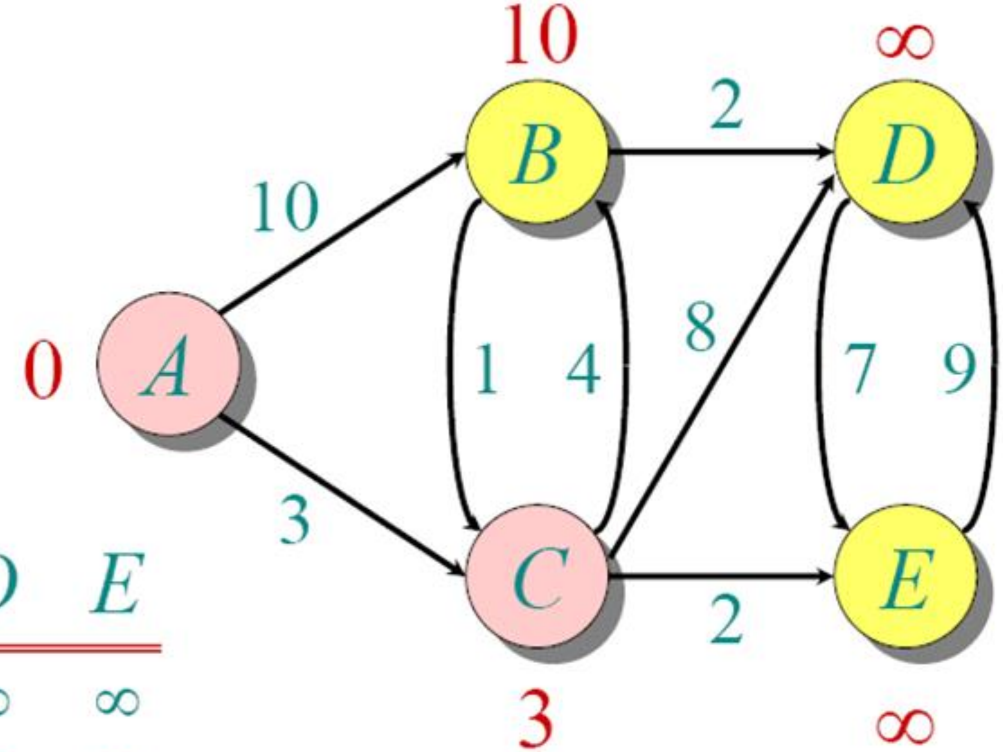
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

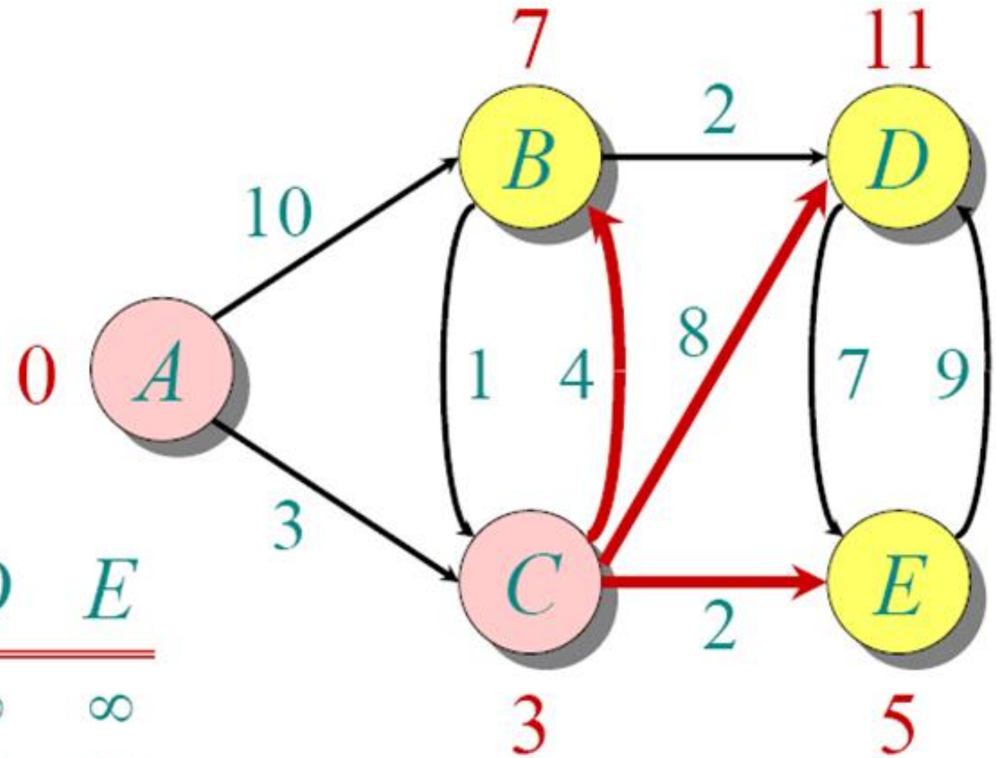
Dijkstra Another Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞

S: $\{A, C\}$

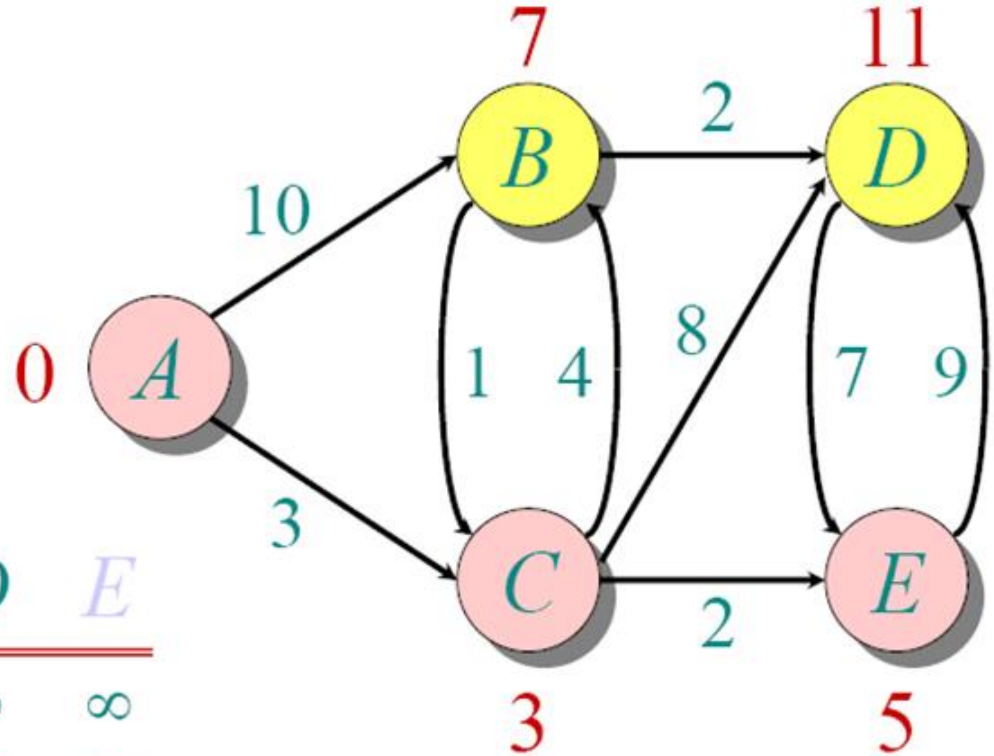
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

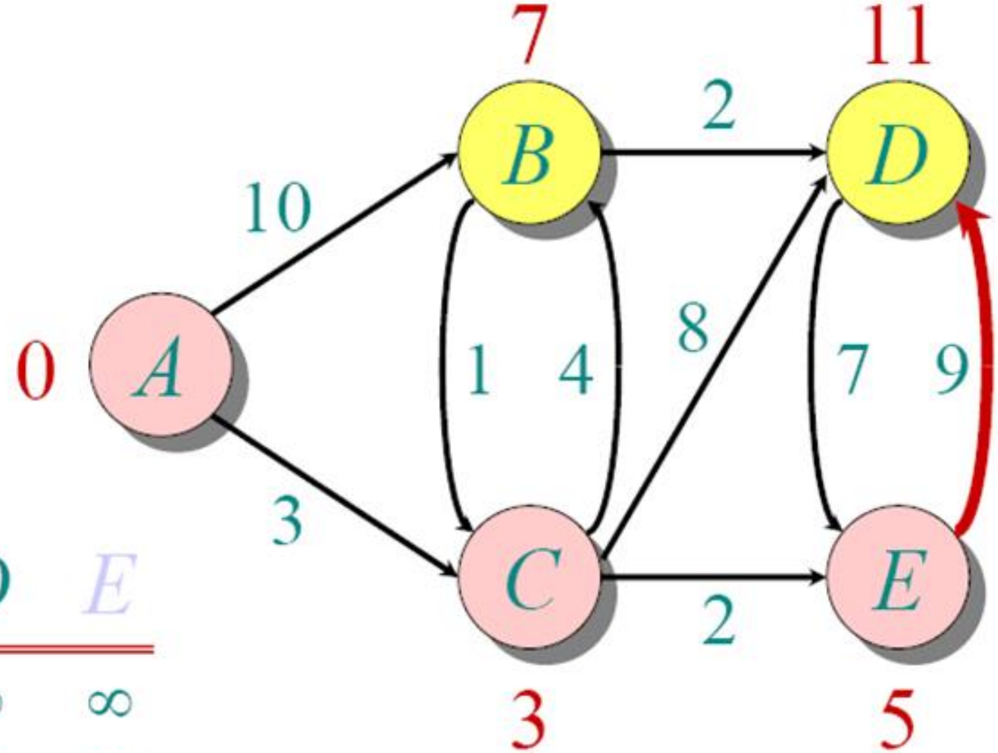
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

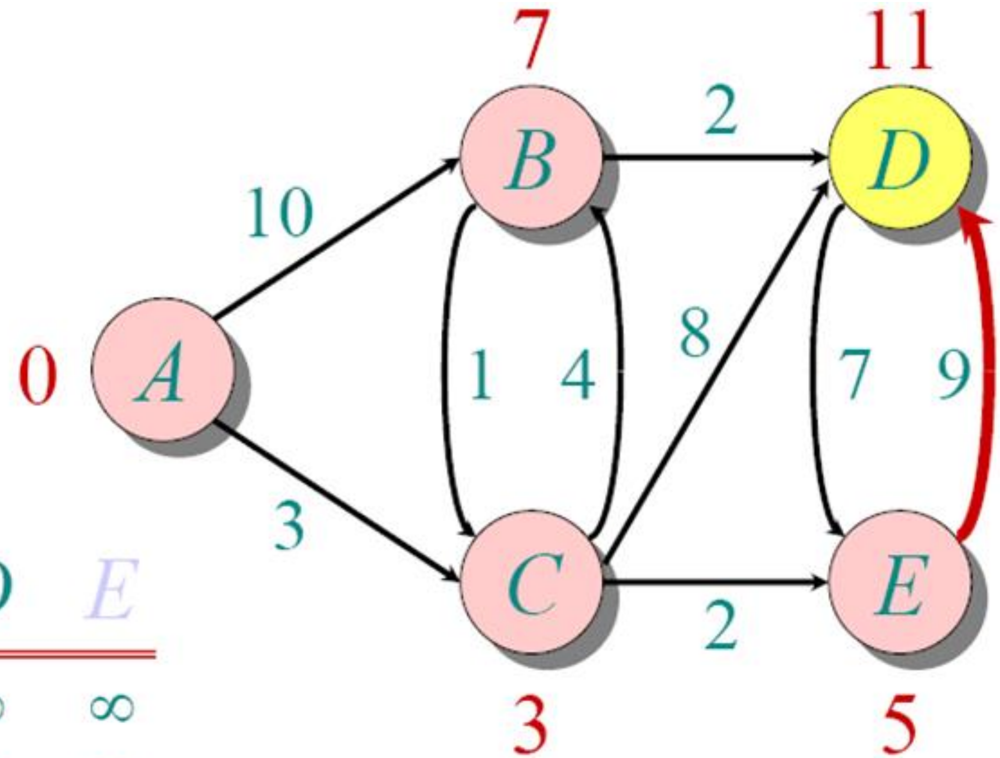
Dijkstra Another Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		7		11	

S: $\{A, C, E\}$

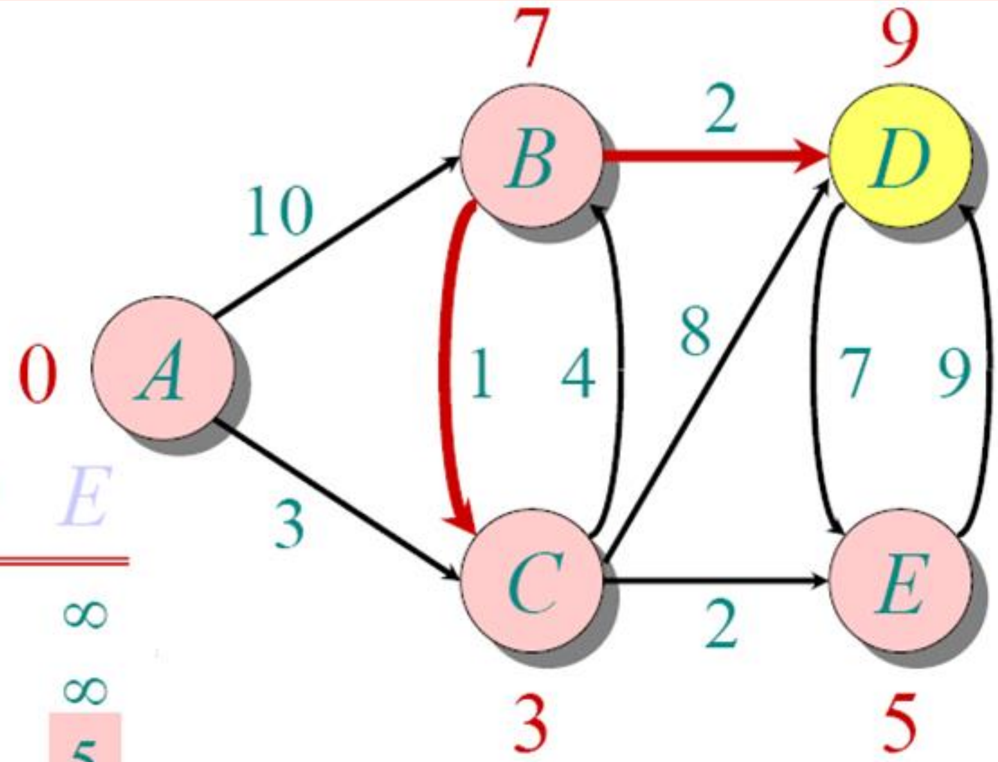
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Dijkstra Another Example

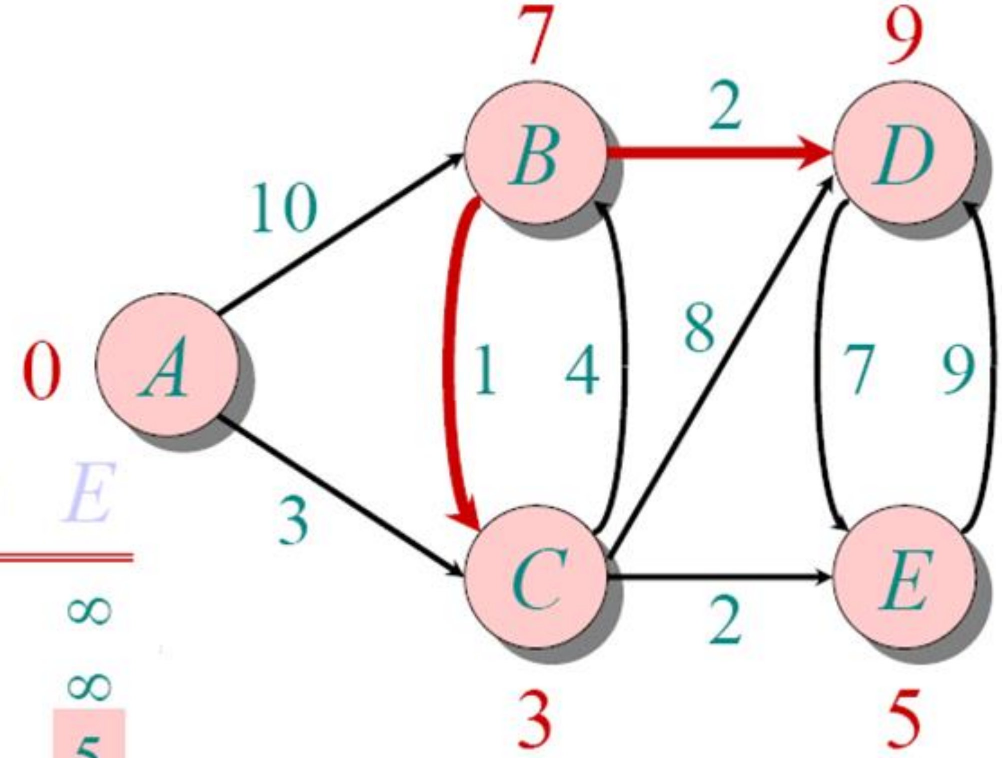


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra Another Example

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	



S: { *A*, *C*, *E*, *B*, *D* }

Dijkstra's - Complexity

SSSP-Dijkstra(**G, w, s**)
  InitializeSingleSource(G, s)
  S $\leftarrow \varnothing$
  Q $\leftarrow$ V[G]
  while Q $\neq$ 0 **do**
    u $\leftarrow$ *ExtractMin*(Q)
    S $\leftarrow$ S $\cup$ {u}
    for u $\in$ *Adj*[u] **do**
      Relax(u,v,w)

executed Θ(V) times

Θ(E) times in total

InitializeSingleSource(**G, s**)
for v $\in$ V[G] **do**
  d[v] $\leftarrow \infty$
  p[v] $\leftarrow$ 0
d[s] $\leftarrow$ 0

Θ(V)

Relax(**u, v, w)**
if d[v] > d[u] + w(u,v) **then**
  d[v] $\leftarrow$ d[u] + w(u,v)
  p[v] $\leftarrow$ u

Θ(1) ?

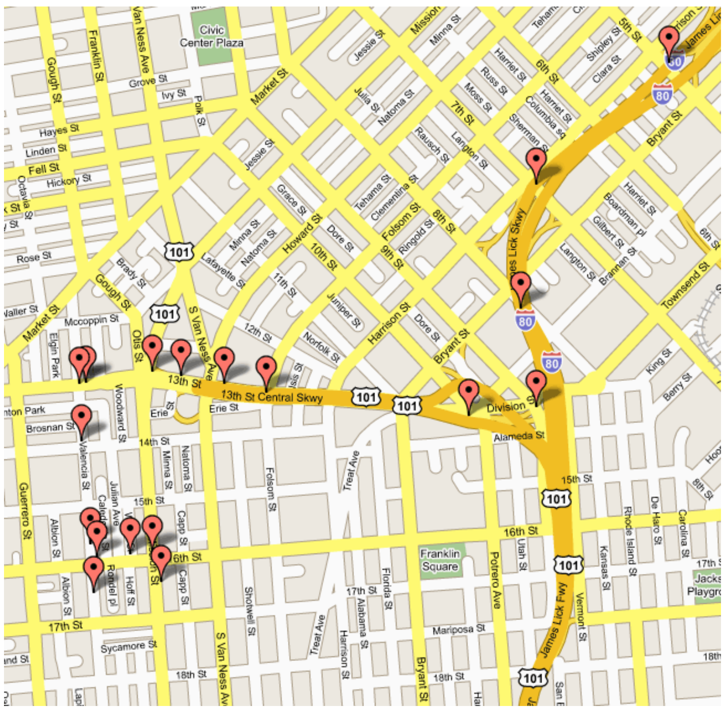
Dijkstra's Running Time

- Extract-Min executed $|V|$ time
- Decrease-Key (Relax) executed $|E|$ time
- Time = $|V|\ T_{\text{Extract-Min}} + |E|\ T_{\text{Decrease-Key}}$
- T depends on different Q implementations

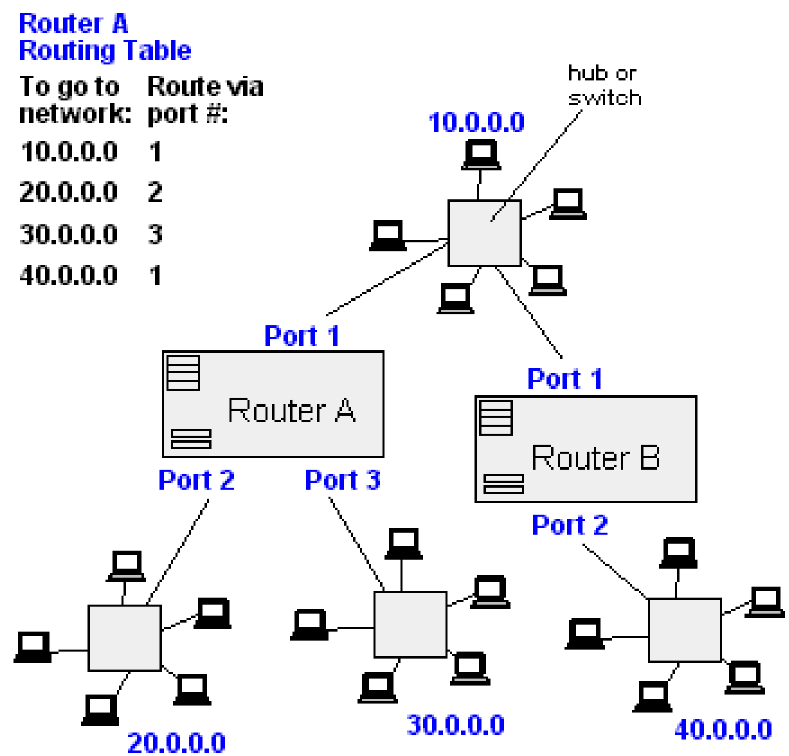
Prority Queue	Extract-Min	Decrease-Key	Total
array	O(V)	O(1)	O(V ^{2})
binary heap	O(lg V)	O(lg V)	O(E lg V)
Fibonacci heap	O(lg V)	O(1) (amort.)	O(V lgV + E)

Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

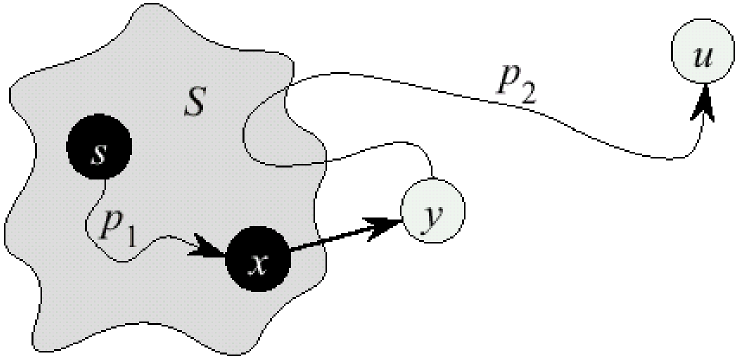


Dijkstra's Algorithm - Summary

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use Q, priority queue keyed by $d[v]$ (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some d decreases)
- Basic idea
  - maintain a set S of solved vertices
  - at each step select "closest" vertex u, add it to S, and relax all edges from u

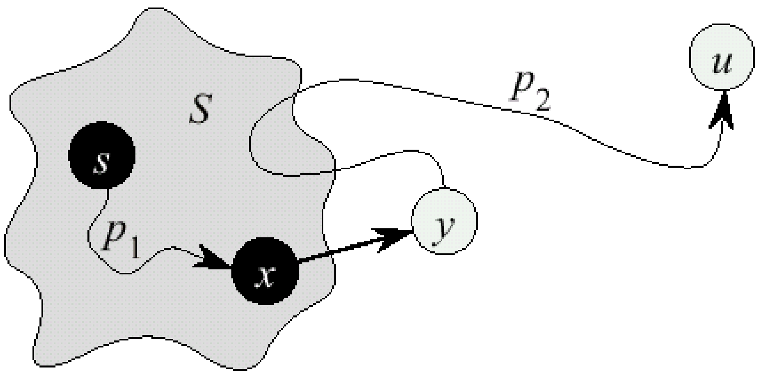
Dijkstra's Correctness

- We will prove that **whenever u is added to S**, $d[u] = \delta(s,u)$, i.e., that d is minimum, and that equality is maintained thereafter
- Proof
  – Note that $\forall v$, $d[v] \geq \delta(s,v)$
  – Let u be the first **vertex picked** such that there is a shorter path than $d[u]$, i.e., that $\Rightarrow d[u] > \delta(s,u)$
  – We will show that this assumption leads to a contradiction



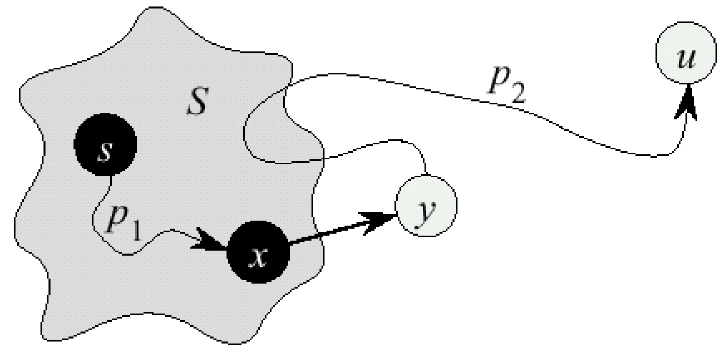
Dijkstra Correctness (2)

- Let y be the first vertex $\in V - S$ on the actual shortest path from s to u, then it must be that $d[y] = \delta(s,y)$ because
  - $d[x]$ 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 $d[y]$ the correct value



Dijkstra Correctness (3)

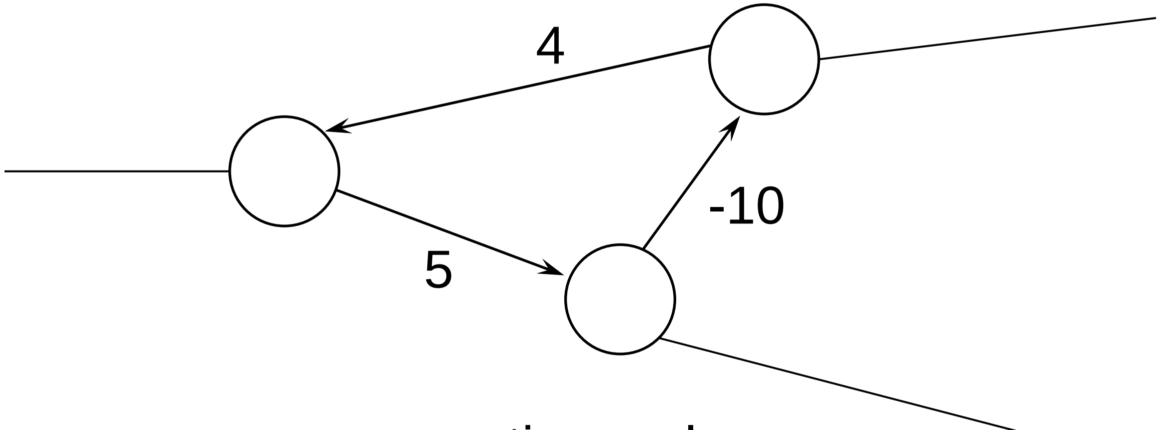
$$
\begin{aligned}
d[u] &> \delta(s,u) && \text{(initial assumption)}\\
&= \delta(s,y)+\delta(y,u) && \text{(optimal substructure)}\\
&= d[y]+\delta(y,u) && \text{(correctness of } d[y]\text{)}\\
&\geq d[y] && \text{(no negative weights)}
\end{aligned}
$$



- But $d[u] > d[y] \Rightarrow$ algorithm would have chosen y (from the PQ) to process next, not $u \Rightarrow$ Contradiction
- Thus d[u] = $\delta(s,u)$ at time of insertion of u into S, and Dijkstra's algorithm is correct

The Bellman-Ford Algorithm

- Handles negative edge weights
- Detects negative cycles
- Is slower than Dijkstra



a negative cycle

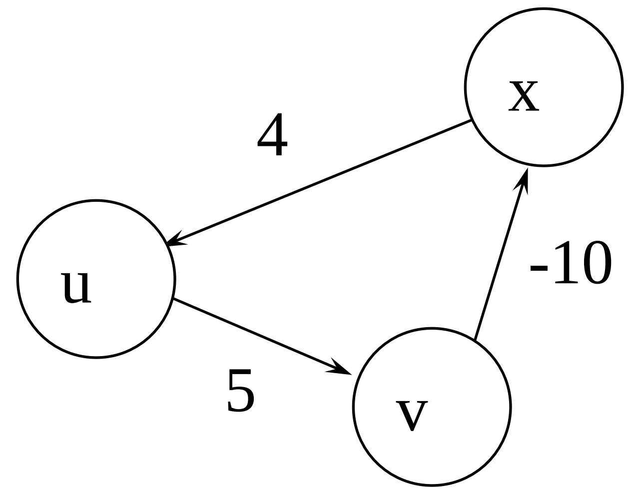
Bellman-Ford: Idea

- Repeatedly update d for all pairs of vertices connected by an edge.
- **Theorem:** If u and v are two vertices with an edge from u to v, and $s \Rightarrow u \rightarrow v$ is a shortest path, and $d[u] = \delta(s,u)$,
- then $d[u]+w(u,v)$ is the length of a shortest path to v.
- **Proof:** Since $s \Rightarrow u \rightarrow v$ is a shortest path, its length is $\delta(s,u) + w(u,v) = \mathrm{d}[u] + w(u,v)$. ■

Why Bellman-Ford Works

- On the first pass, we find δ(s,u) for all vertices whose shortest paths have one edge.
- On the second pass, the d[*u*] values computed for the one-edge-away vertices are correct (= δ(s,u)), so they are used to compute the correct d values for vertices whose shortest paths have two edges.
- Since no shortest path can have more than |V[G]|-1 edges, after that many passes all d values are correct.
- Note: all vertices not reachable from s will have their original values of infinity. (Same, by the way, for Dijkstra).

Negative Cycle Detection



- What if there is a negative-weight cycle reachable from s?
- Assume: $d[u] \le d[x]+4$
- $d[v] \le d[u]+5$
- $d[x] \le d[v]-10$
- Adding:
- $d[u]+d[v]+d[x] \le d[x]+d[u]+d[v]-1$
- Because it's a cycle, vertices on left are same as those on right. Thus we get $0 \le -1$; a contradiction. So for at least one edge (u,v),
- $d[v] > d[u] + w(u,v)$
- This is exactly what Bellman-Ford checks for.

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 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 (d[v] > d[u] + 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 (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 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 (d[v] > d[u] + w(u,v))
            return "no solution";
```

What will be the running time?

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

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 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 (d[v] > d[u] + w(u,v))
           return "no solution";
```

What will be the running time?

A: O(VE)

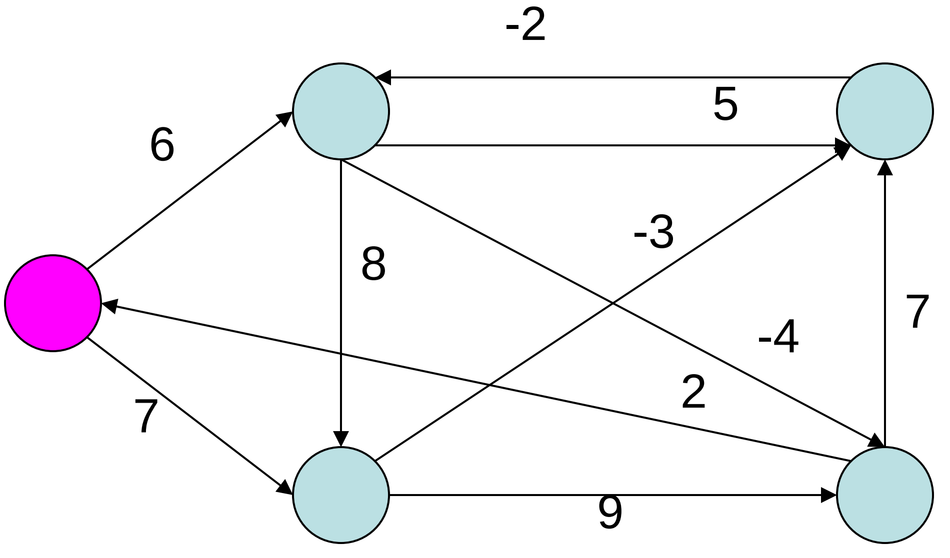
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

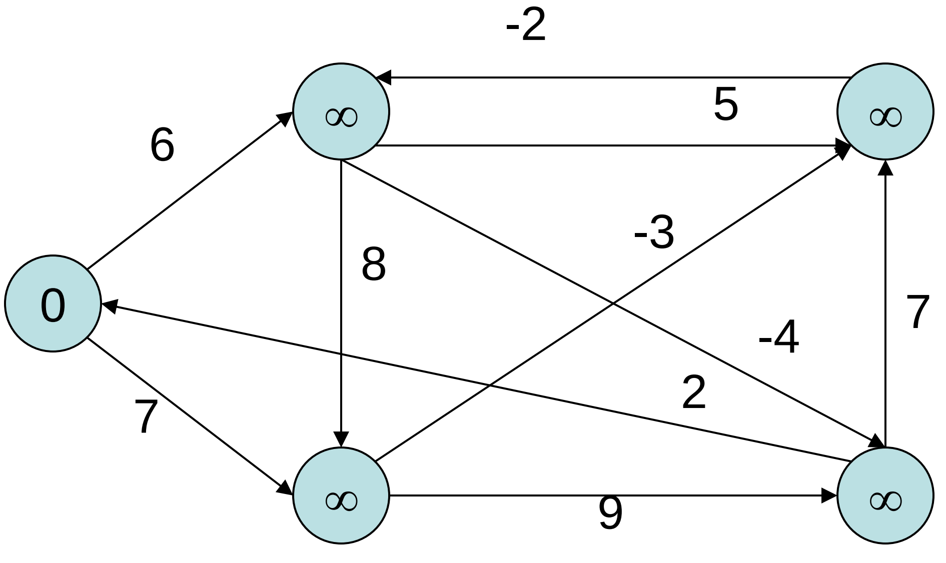
```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return “no solution”;
```

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

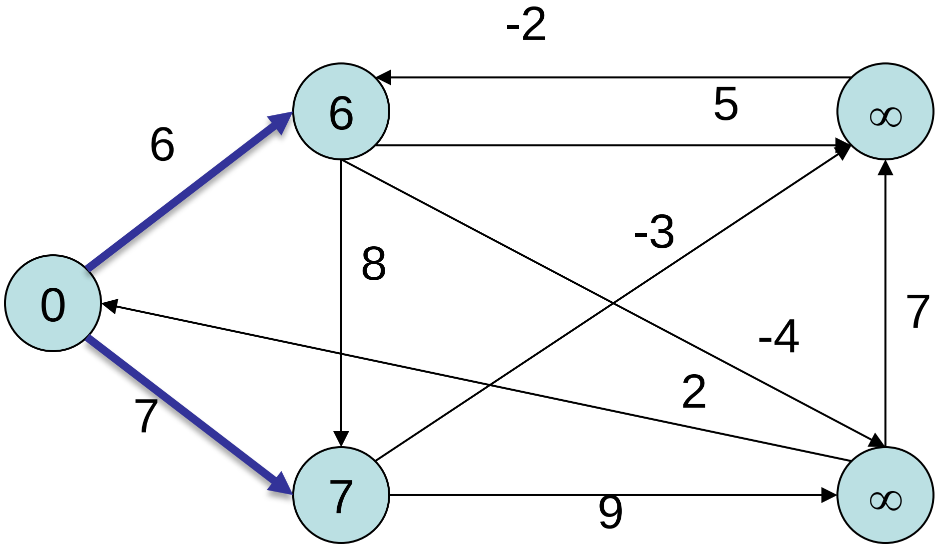
Bellman-Ford Algorithm - Example



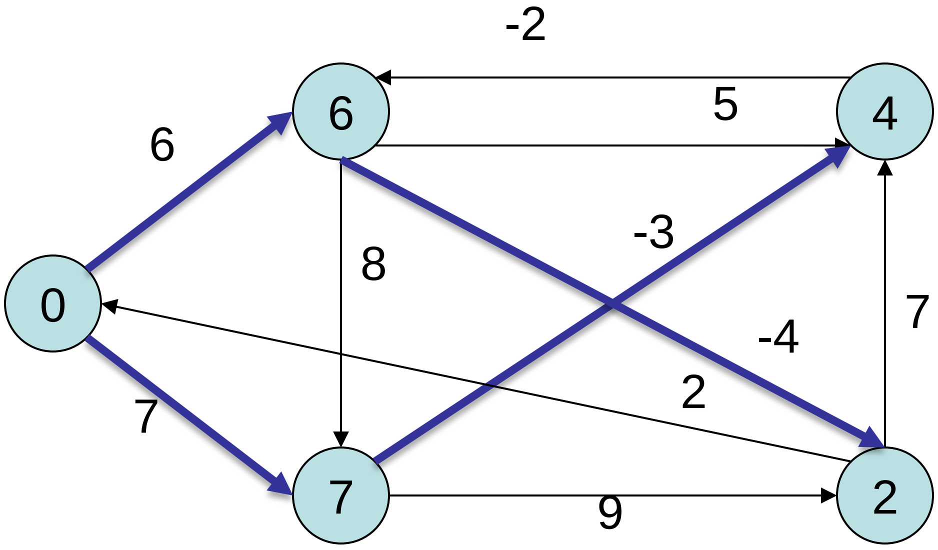
Bellman-Ford Algorithm - Example



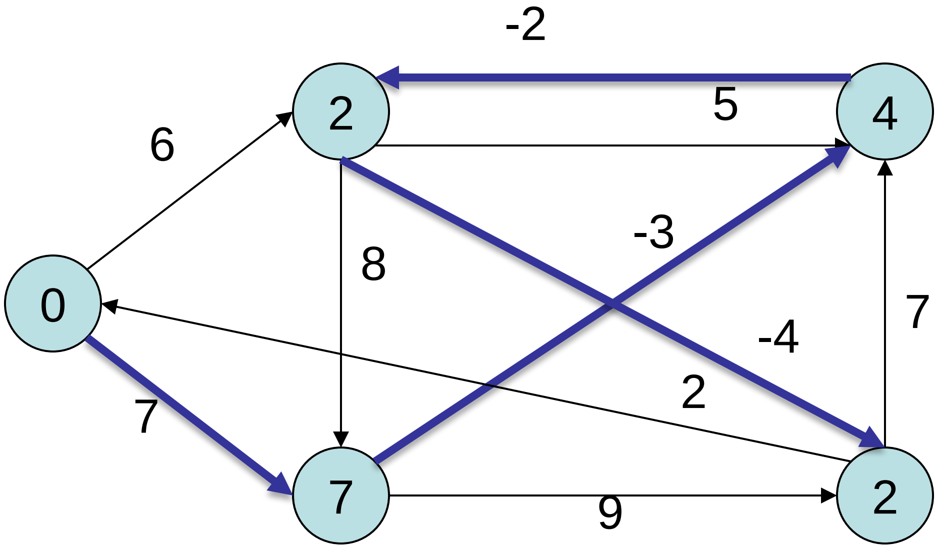
Bellman-Ford Algorithm - Example



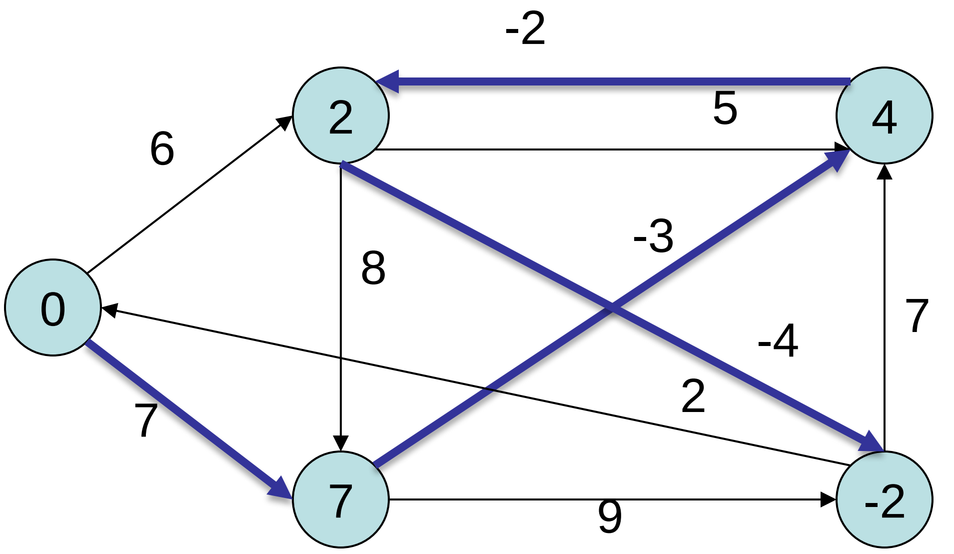
Bellman-Ford Algorithm - Example



Bellman-Ford Algorithm - Example



Bellman-Ford Algorithm - Example



Bellman-Ford - Complexity

BellmanFord(**graph** (G,w), s)

    InitializeSingleSource(G, s)

executed Θ(V) times   **for** *i* = 1 **to** |V| – 1 **do**

        for *(u,v)* ∈ E **do**

Θ(E)   *Relax*(u,v,w)   Θ(1)

    for *(u,v)* ∈ E **do**

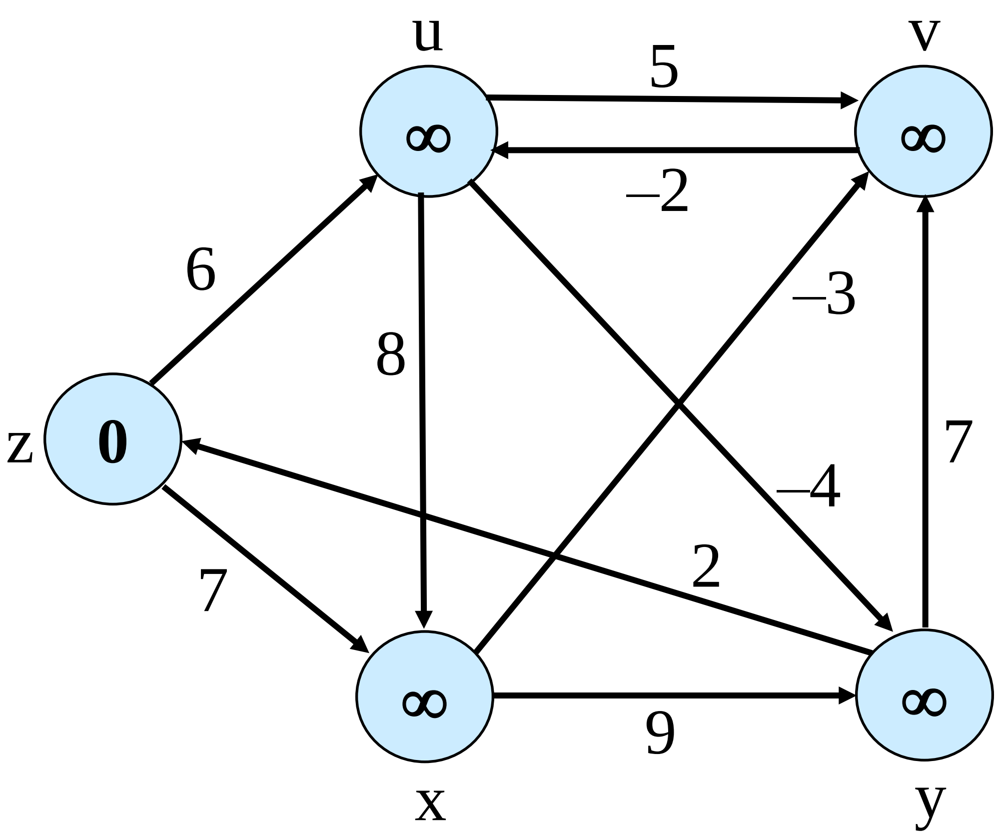
Θ(E)   **if** *d[v]* > *d[u]* + *w(u,v)* **then**

            return false

    return true

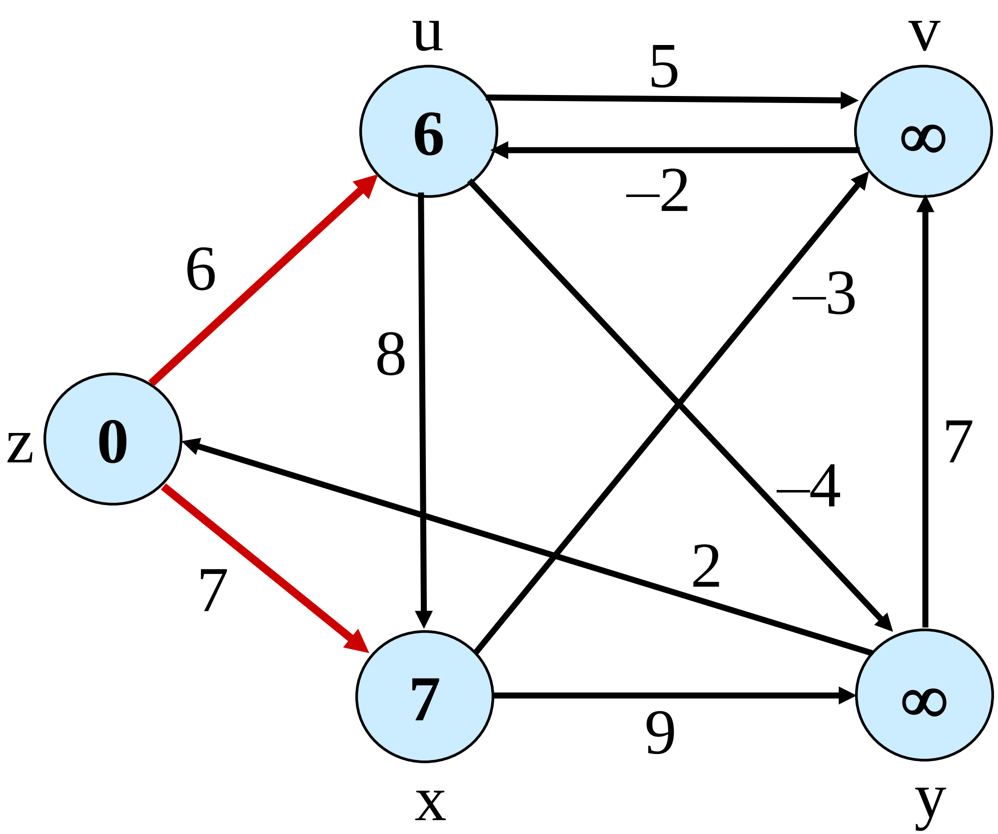
So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Example



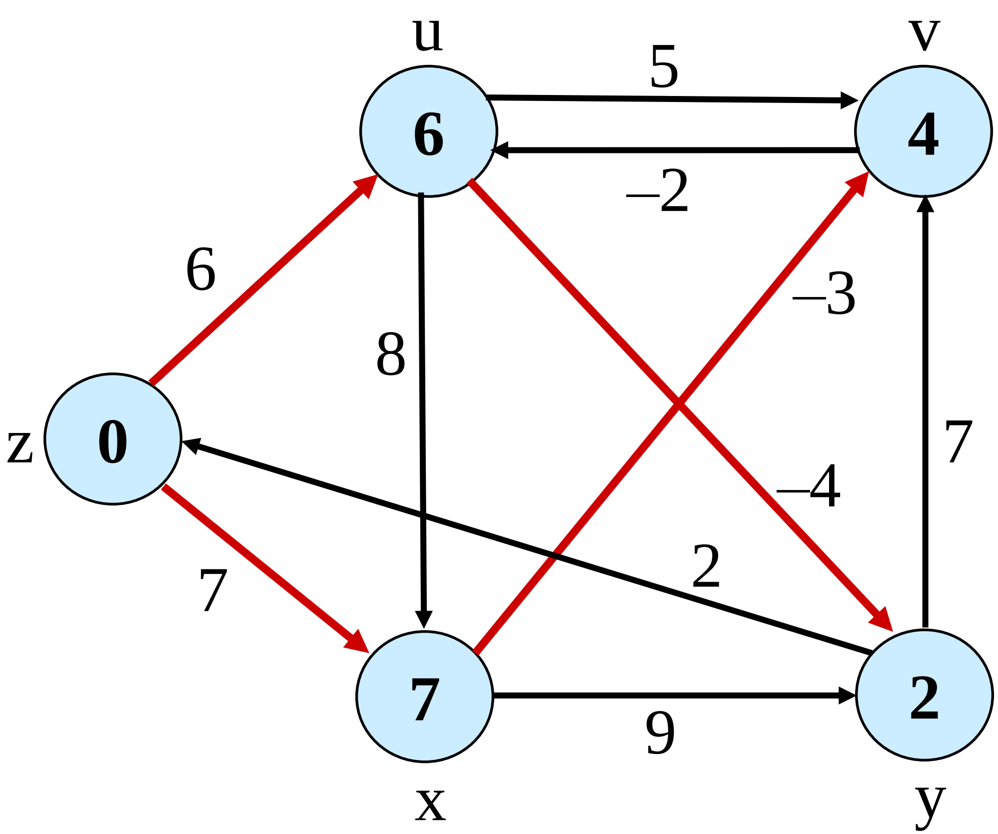
So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Example

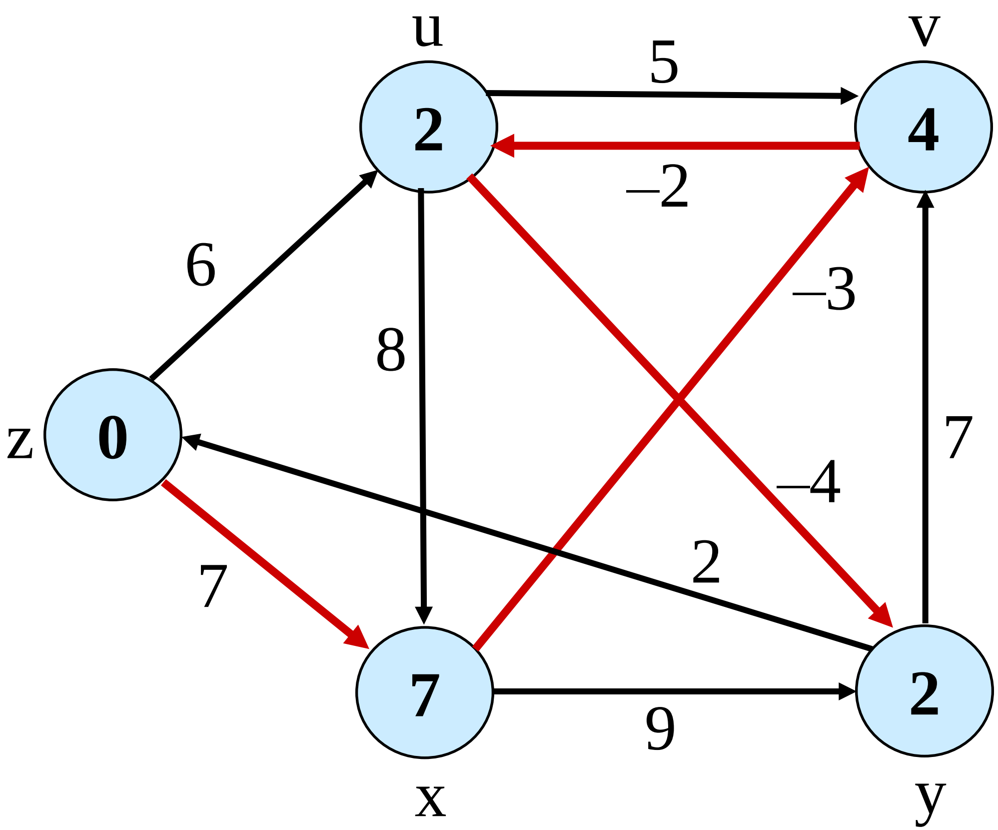


So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

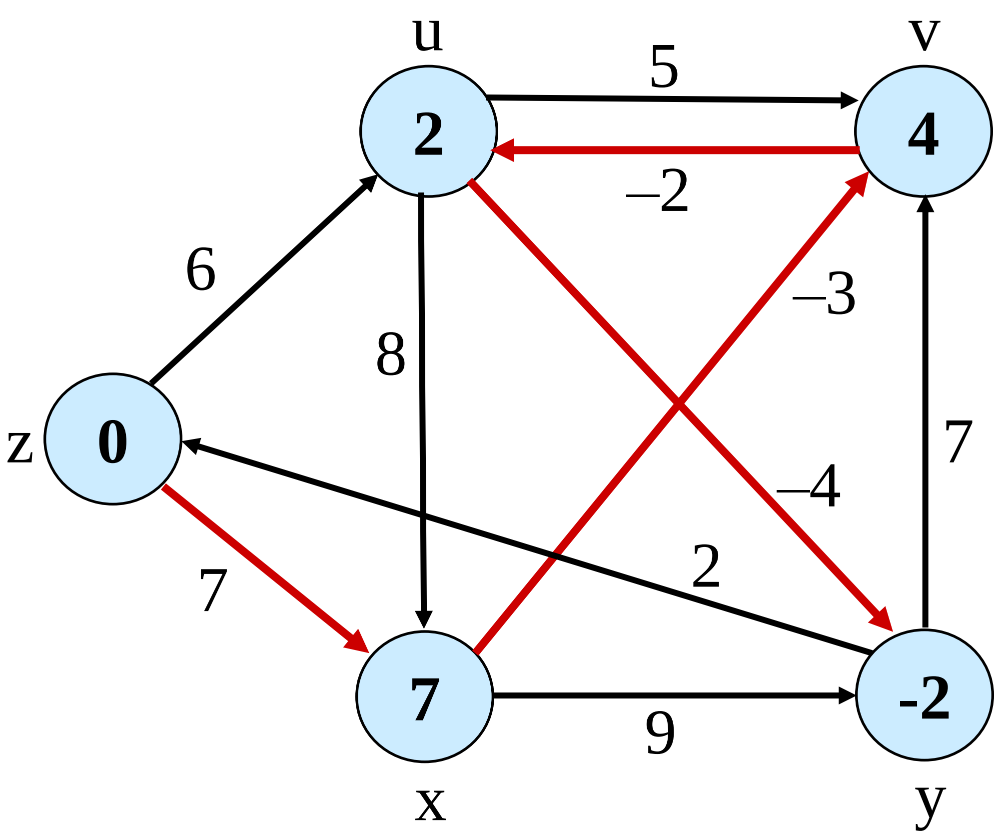
Example



Example



Example



Correctness of Bellman-Ford

- Let $\delta_i(s,u)$ denote the length of path from *s* to *u*, that is shortest among all paths, that contain at most *i* edges
- Prove by induction that $d[u] = \delta_i(s,u)$ after the *i*-th iteration of Bellman-Ford
  - Base case (i=0) trivial
  - Inductive step (say $d[u] = \delta_{i-1}(s,u)$):
    - Either $\delta_i(s,u) = \delta_{i-1}(s,u)$
    - Or $\delta_i(s,u) = \delta_{i-1}(s,z) + w(z,u)$
    - In an iteration we try to relax each edge ((z,u) also), so we will catch both cases, thus $d[u] = \delta_i(s,u)$

Correctness of Bellman-Ford

- After n-1 iterations, $d[u] = \delta_{n-1}(s,u)$, for each vertex u.
- If there is still some edge to relax in the graph, then there is a vertex u, such that $\delta_n(s,u) < \delta_{n-1}(s,u)$. But there are only n vertices in G – we have a cycle, and it must be negative.
- Otherwise, $d[u]= \delta_{n-1}(s,u) = \delta(s,u)$, for all u, since any shortest path will have at most n-1 edges

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 verts in that order, we will do each path in forward order (will never relax edges out of vert before doing all edges into vert).
    - Thus: just one pass. *What will be the running time?*

Shortest Paths in DAGs - SP-DAG

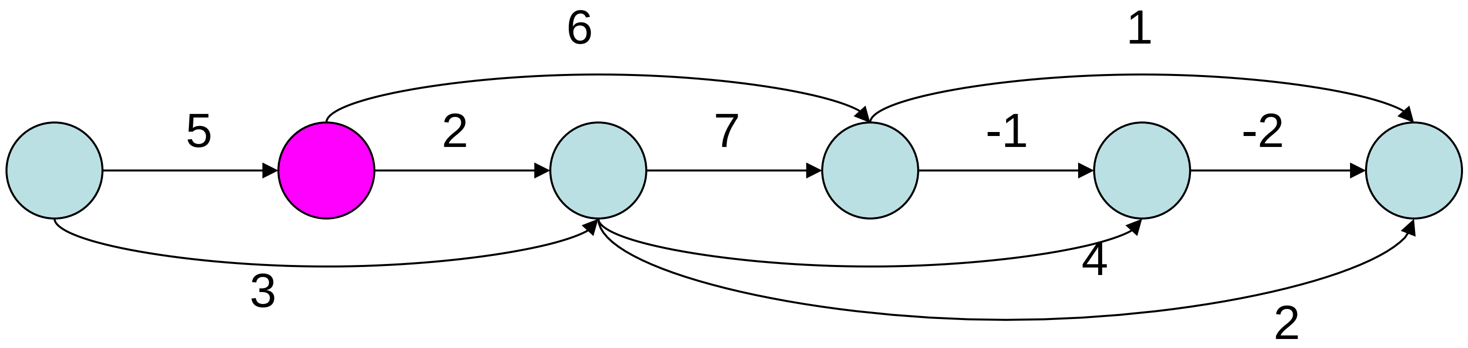
SP-DAG(**graph** (G,w), **vertex** s)

topologically sort vertices of G

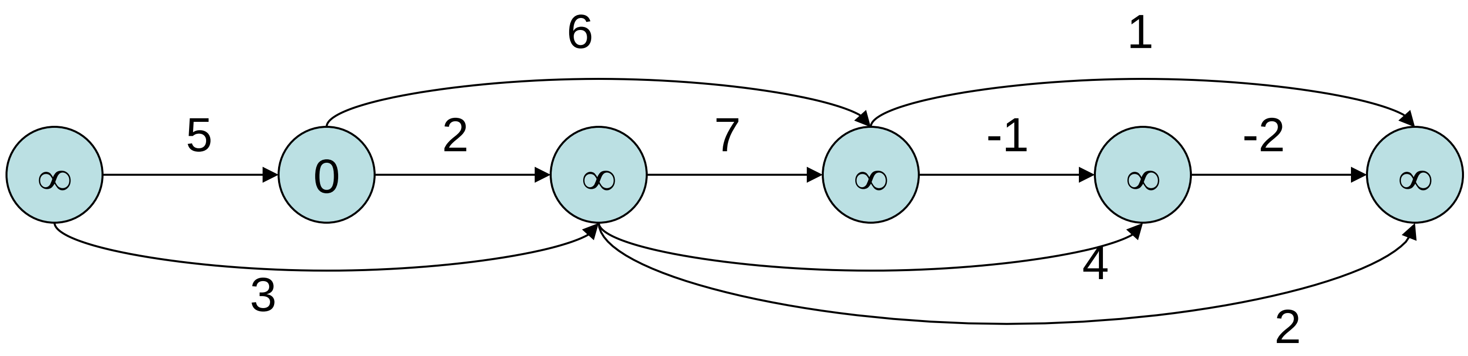
InitializeSingleSource(G, s)

for each vertex u taken in topologically sorted order **do**
    for each vertex $v \in$ *Adj*[u] **do**
        Relax(u,v,w)

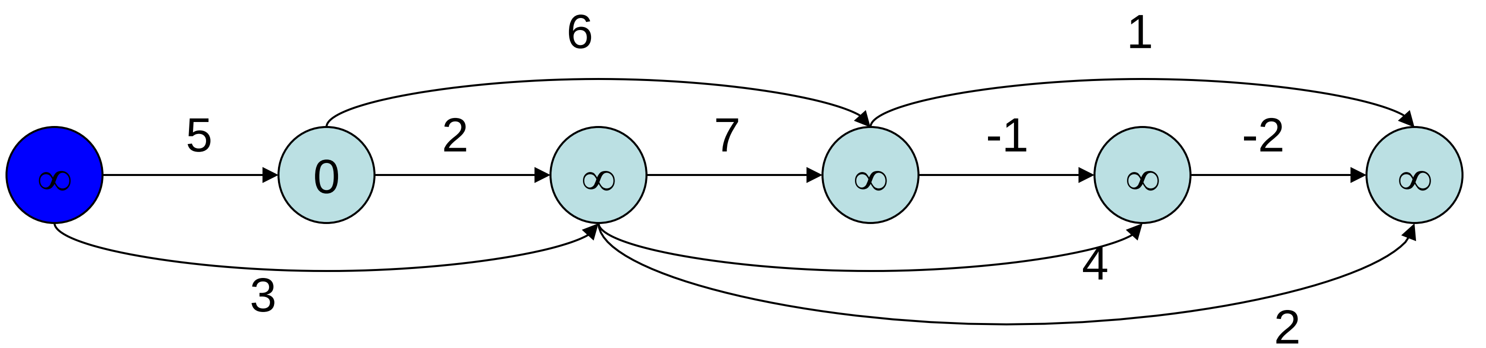
Shortest Paths in DAGs - Example



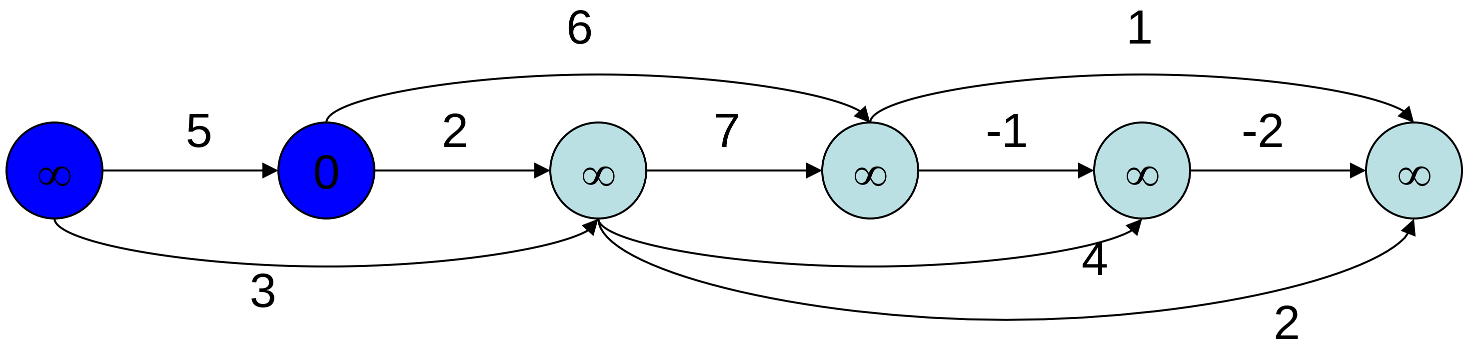
Shortest Paths in DAGs - Example



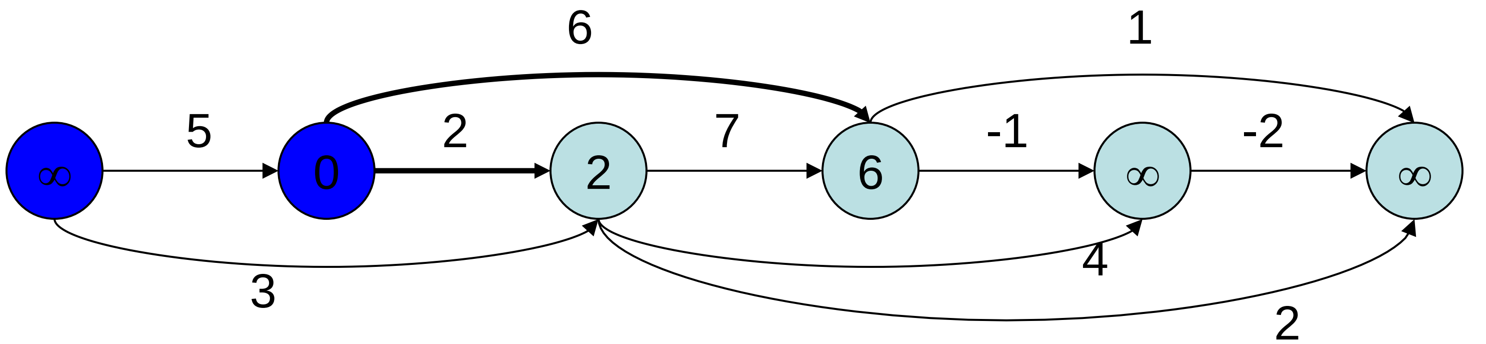
Shortest Paths in DAGs - Example



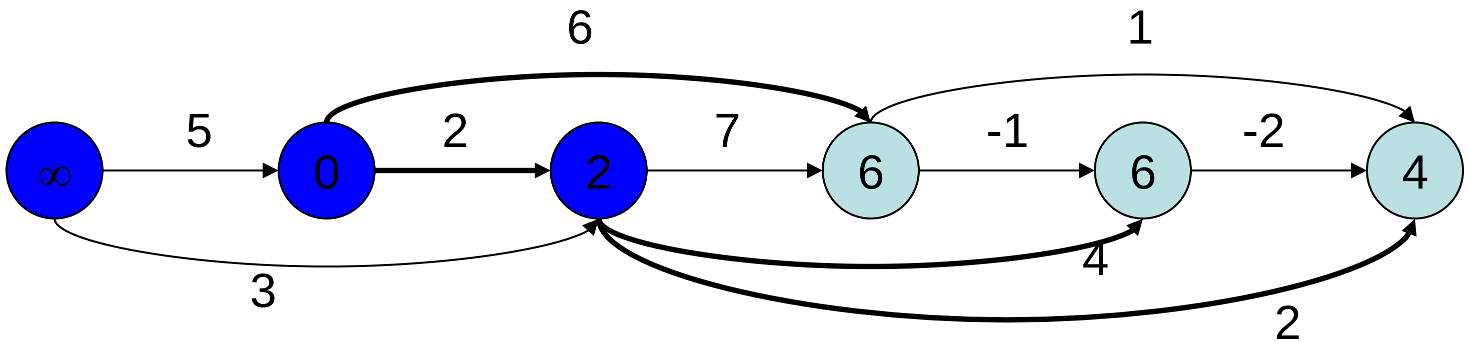
Shortest Paths in DAGs - Example



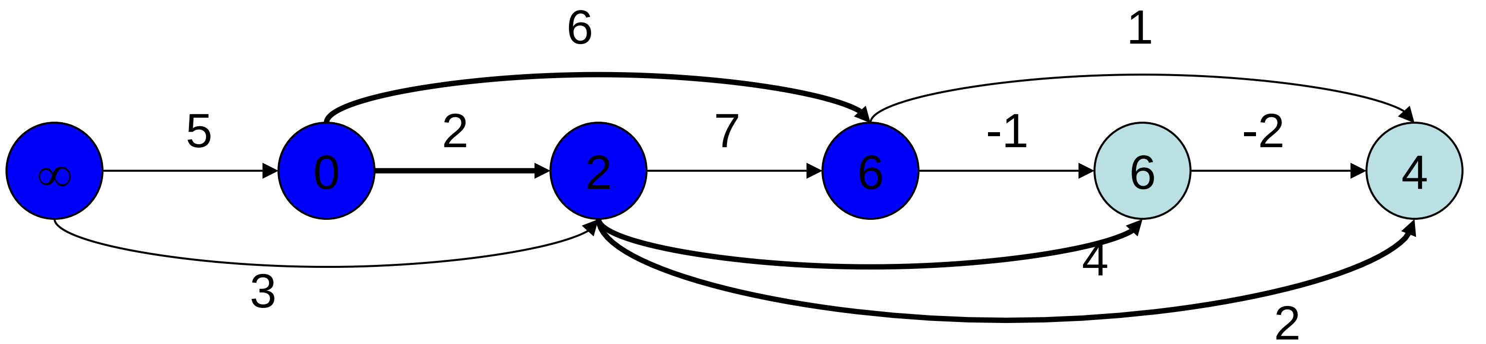
Shortest Paths in DAGs - Example



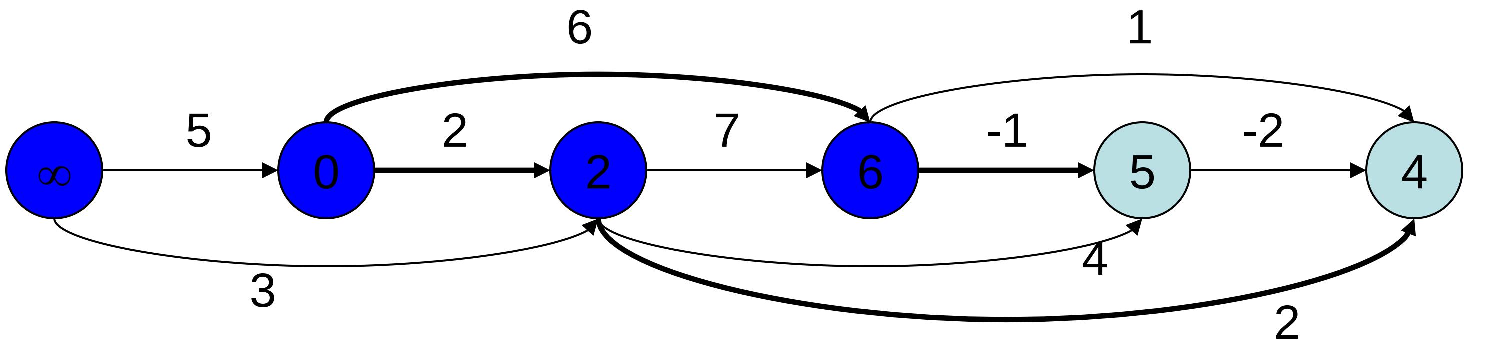
Shortest Paths in DAGs - Example



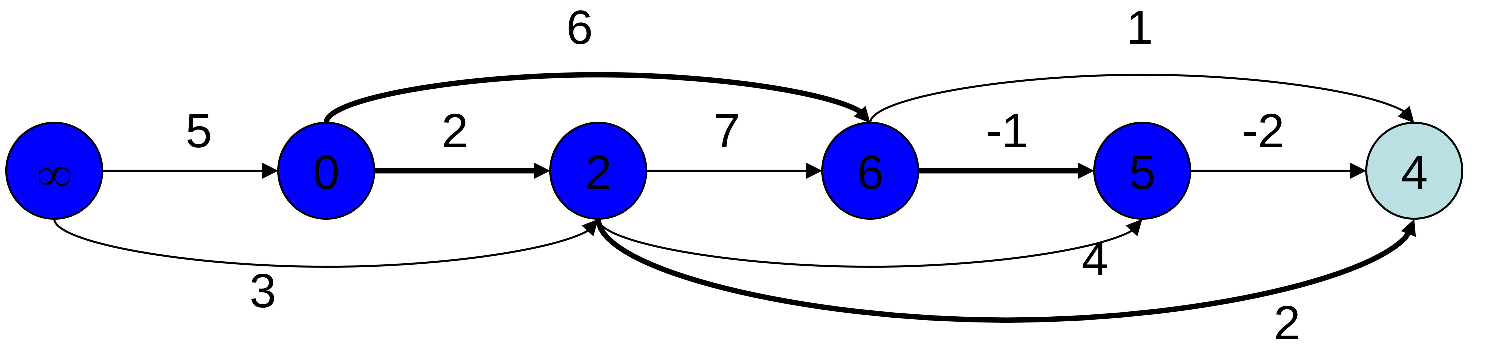
Shortest Paths in DAGs - Example



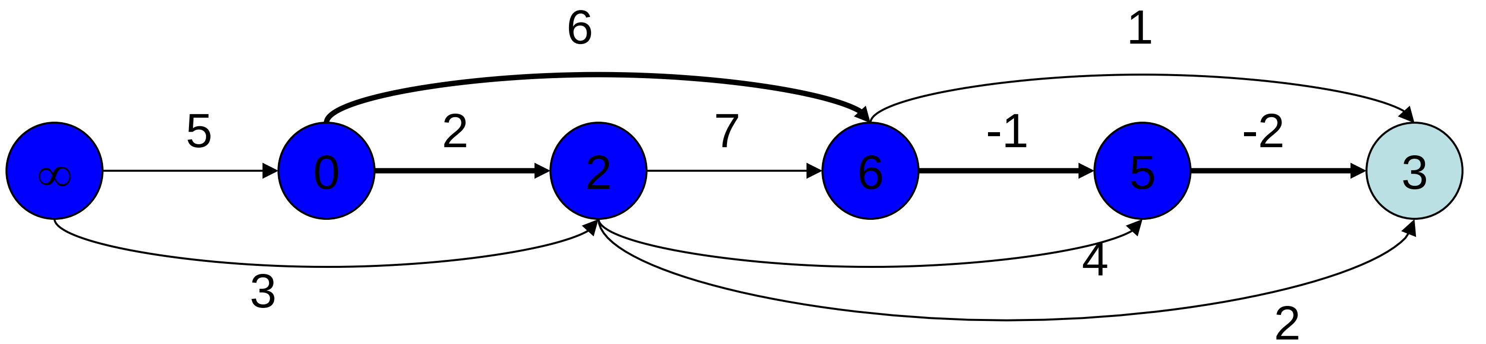
Shortest Paths in DAGs - Example



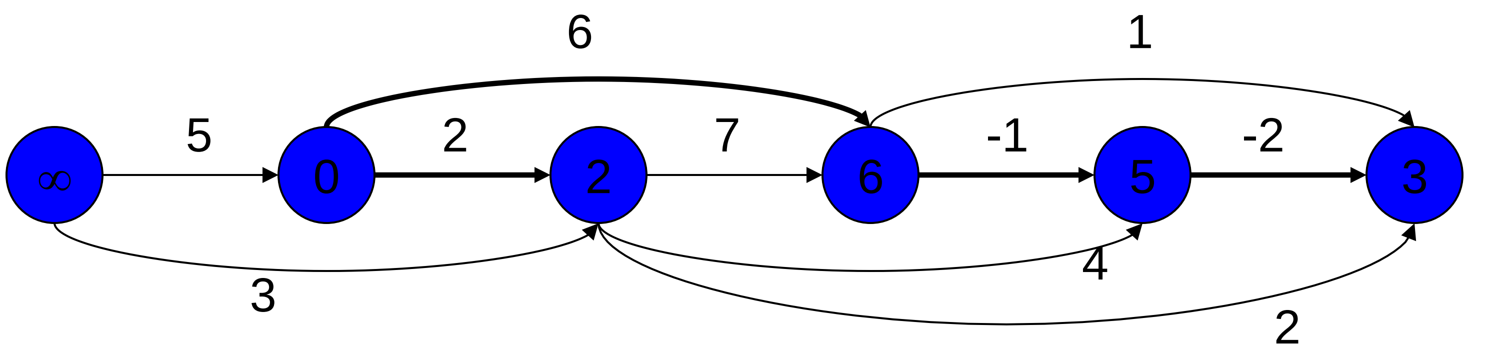
Shortest Paths in DAGs - Example



Shortest Paths in DAGs - Example



Shortest Paths in DAGs - Example



SP in a DAGs - Complexity

SP-DAG(**graph** (G,w), s)

topologically sort vertices of G

InitializeSingleSource(G, s)

for each vertex u taken in topologically sorted order **do**
  for each vertex $v \in$ *Adj*[u] **do**
    Relax(u,v,w)

$T(V,E) = \Theta(V + E) + \Theta(V) + \Theta(V) + E\ \Theta(1) = \Theta(V + E)$