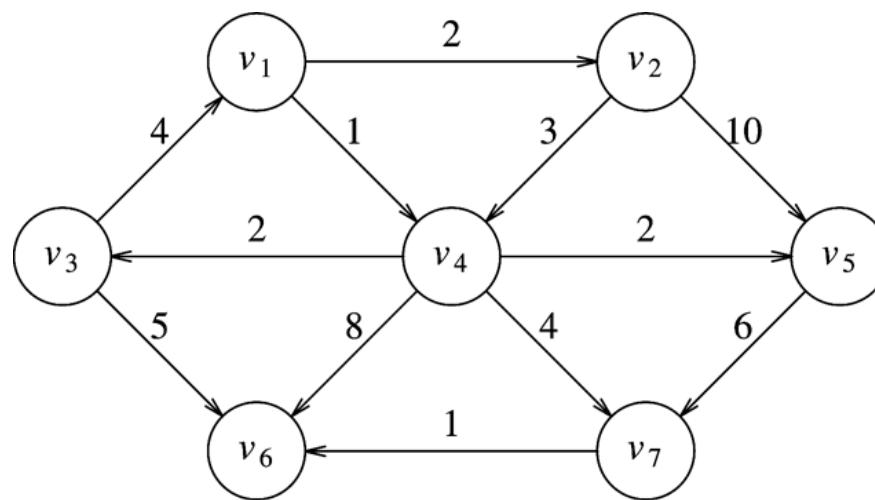


# Graphs



# Graph

- A Graph  $G = (V, E)$
- A set  $V$  of vertices and a set  $E$  edges
- A graph is a way representing connections/relationships between pairs of objects in  $V$ .
- Each edge is a pair  $(v, w)$ , where  $v, w \in V$
- Edges are either directed or undirected
  - Directed referred to as ordered
    - Directed graphs are called Digraphs
    - Directed graphs with no cycles(acyclic) are called DAGs
  - Undirected referred to as unordered

# Graph Terminology

- An edge  $(v, v)$  is a **loop**
- Vertices can have incoming and outgoing edges
  - **indegree** – number of incoming edges of a vertex  $v$
  - **outdegree** – number of outgoing edges of a vertex  $v$
- A path in a graph is a sequence of vertices  $w_1, w_2, w_3, \dots, w_N$  such that  $(w_i, w_{i+1}) \in E$  for  $1 \leq i < N$ 
  - **simple path**: path where all vertices distinct except possibly first/last.
  - **path length**:  $N-1$  for  $N$  vertices
- Edges can have an associated cost: **weight**.

# Graph Properties

- An **undirected** graph is connected if there is a path from every vertex to every other vertex
- A **directed** graph that is connected is called strongly connected.
- **Weakly connected** - If directed graph is not strongly connected, but removing direction makes graph connected
- **Complete graph** – edge between every pair of vertices.

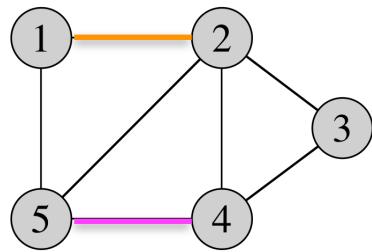
# Graph Representations

- **Adjacency matrix** –  $\Theta(|V|^2)$  matrix, **1** if edge between two vertices, **0** otherwise.
  - Good for dense graphs.
  - Wasted space if not dense.  $|E| = \Theta(|V|^2)$
  - If graph is sparse, adjacency list is better.
- **Adjacency list** – for each vertex, store a list of vertices that share an edge

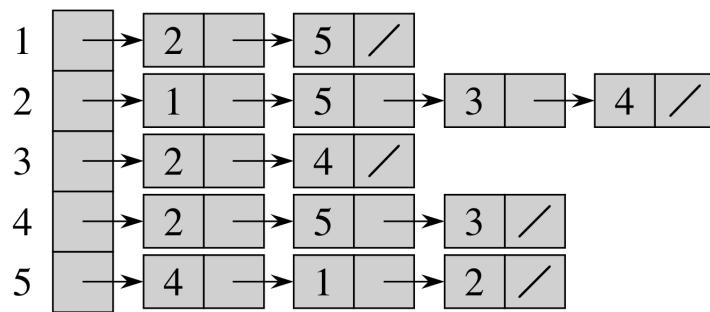
# Adjacency list

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

# Undirected Graph



Graph

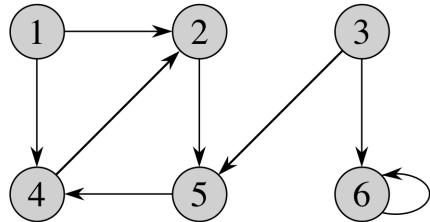


Adjacency List

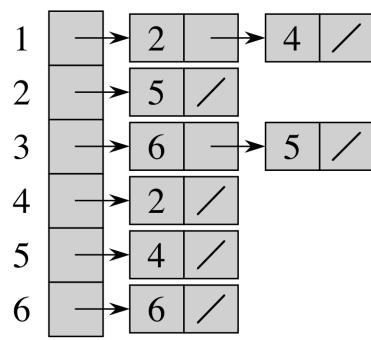
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix

# Directed Graph



Graph

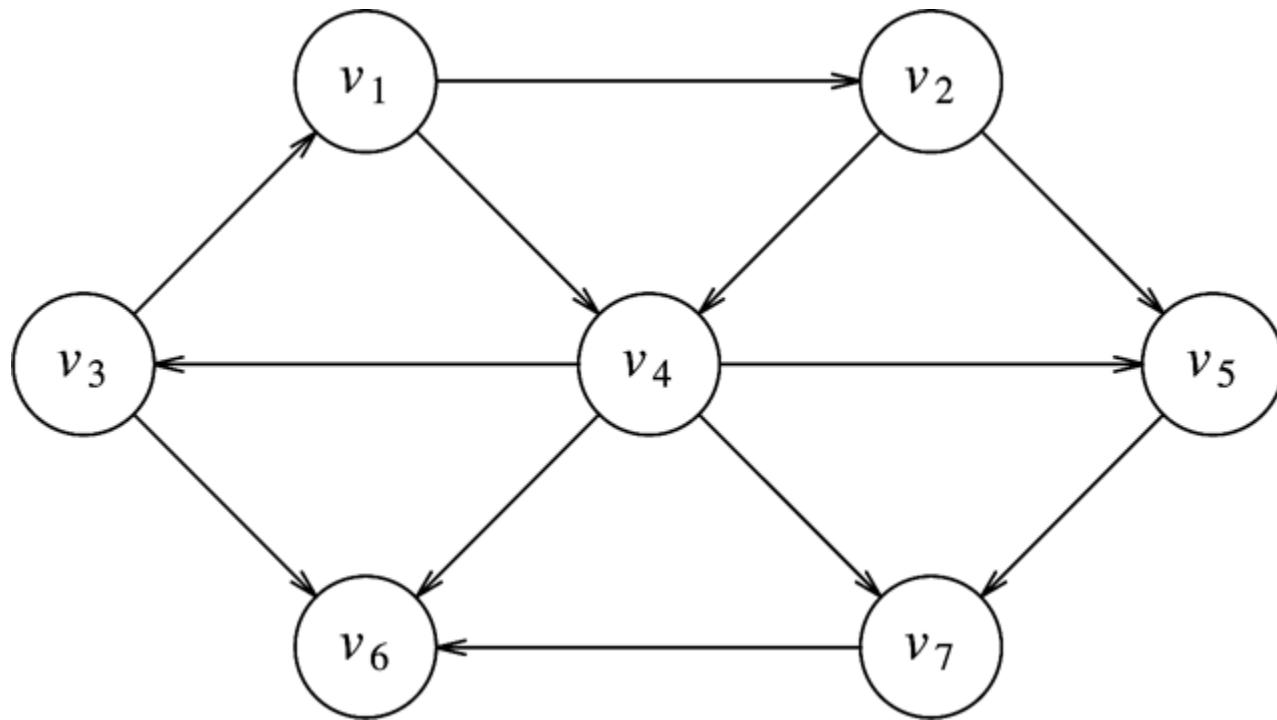


Adjacency List

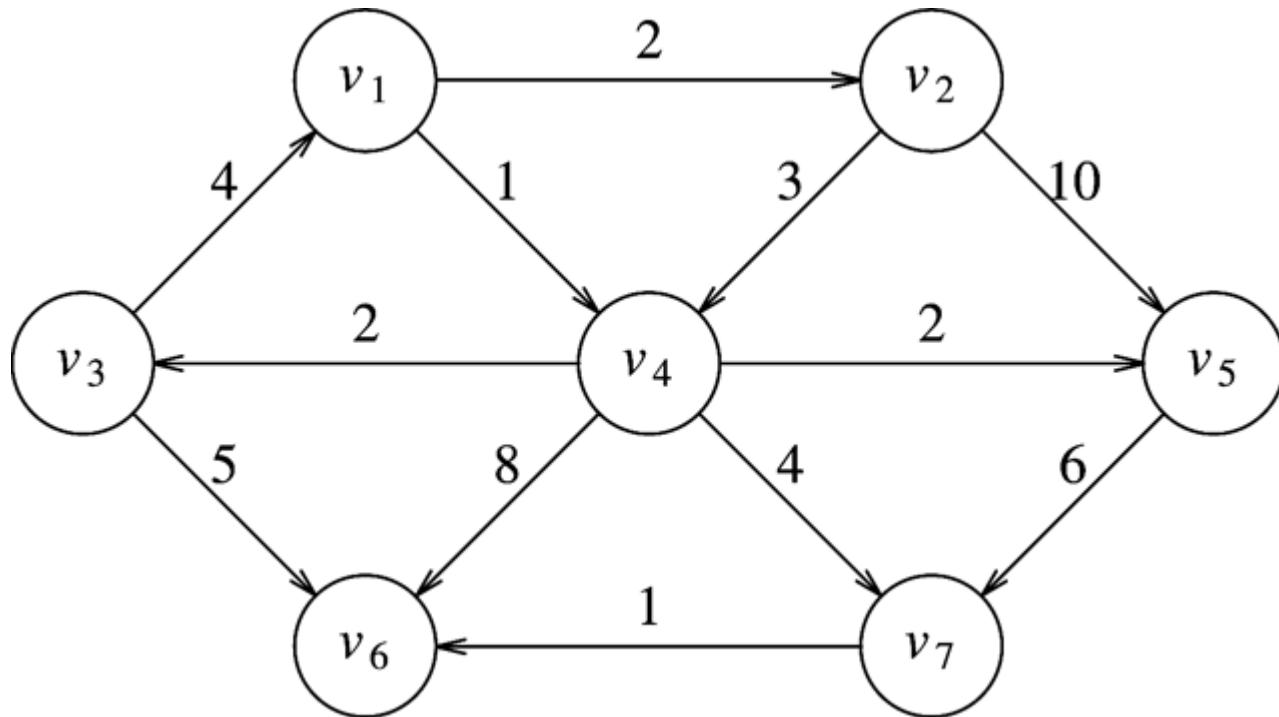
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix

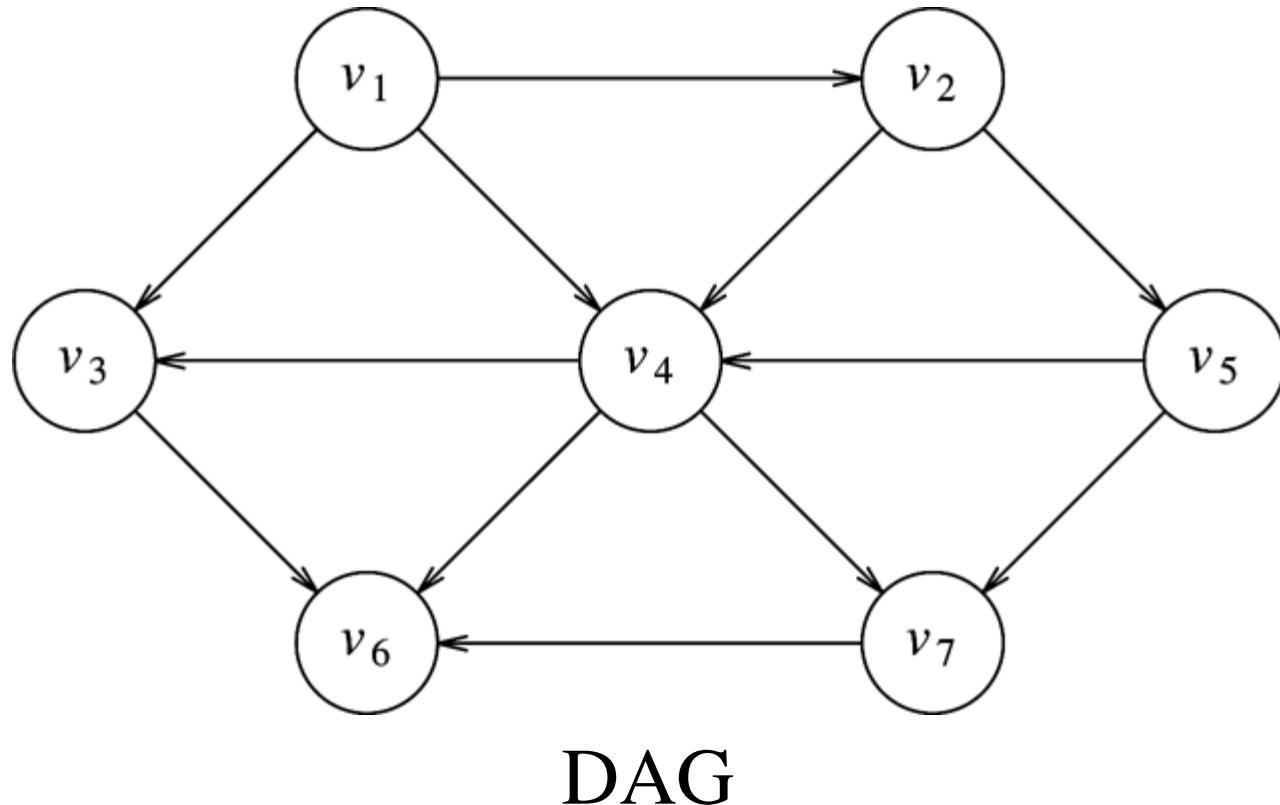
# Directed Graph - unweighted



# Directed Graph - weighted



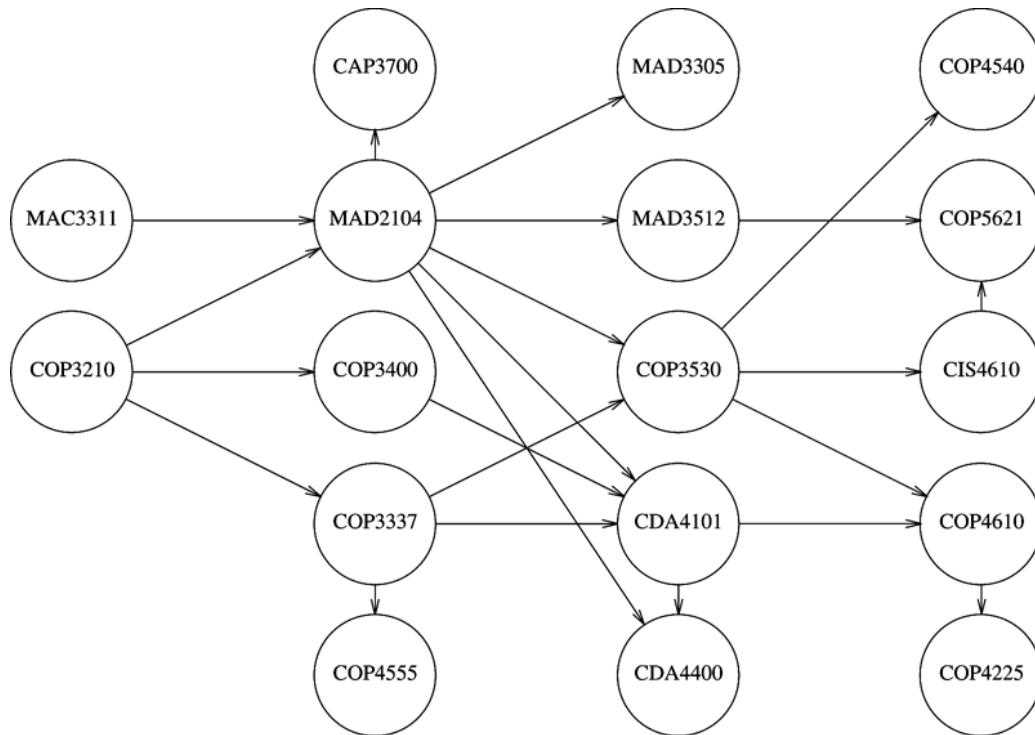
# Directed Acyclic Graph



# Topological Sort

- Linear ordering of vertices for a DAG (Directed Acyclic Graph).
- Every vertex comes before all vertices to which it has outgoing edges.
- Every DAG has at least 1 topological sort
  - **Hamiltonian path** – every vertex is visited exactly once
    - 1 indicates a Hamiltonian path
    - 2 or more, NO Hamiltonian path
- Topological Sort - used in scheduling jobs or tasks

# Scheduling



# Topological Sort

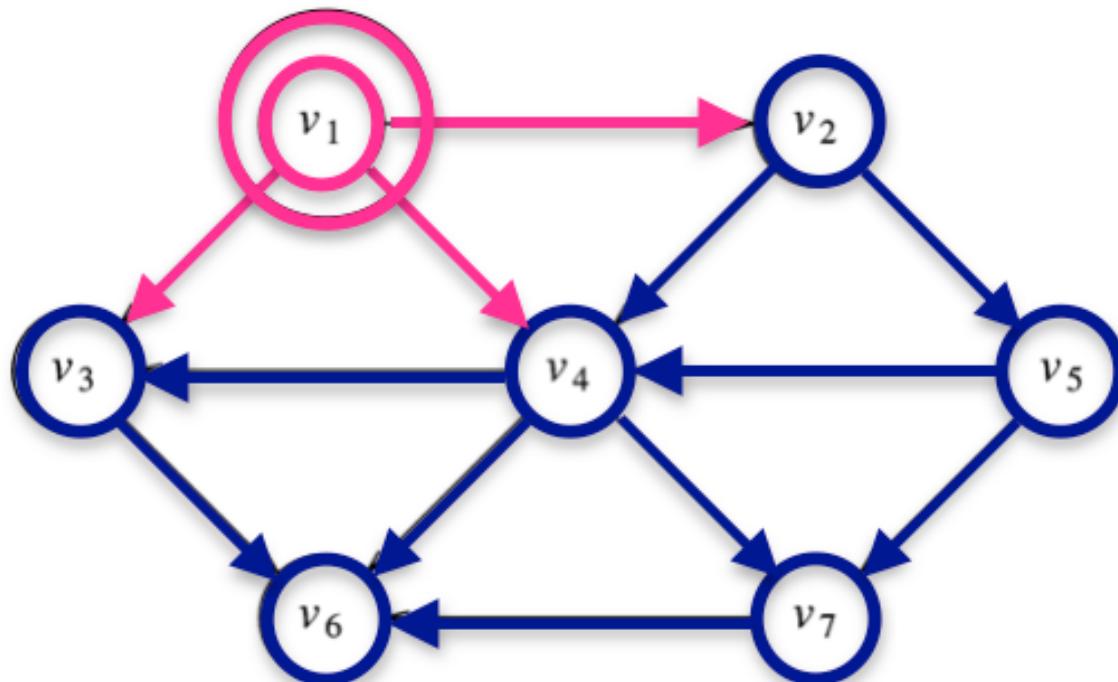
- **Simple algorithm:**

1. Find an initial vertex  $v$  with no incoming edges
2. Print  $v$
3. Remove  $v$  from graph
4. Remove  $v$ 's edges from graph, updating affected vertices
5. Iterate (steps 1-4) over the remaining graph

**Running Time :**  $O(|V|^2)$

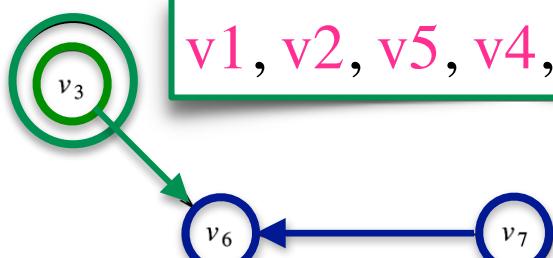
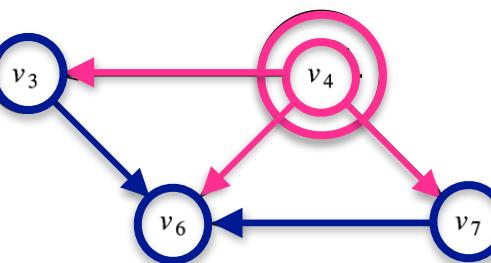
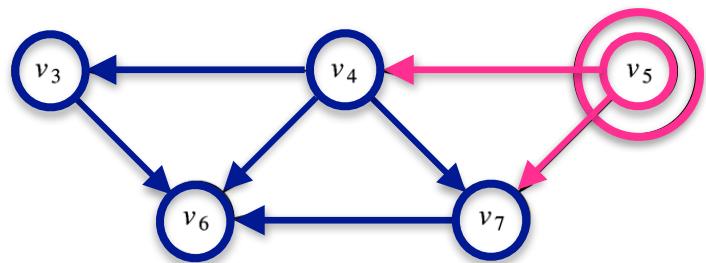
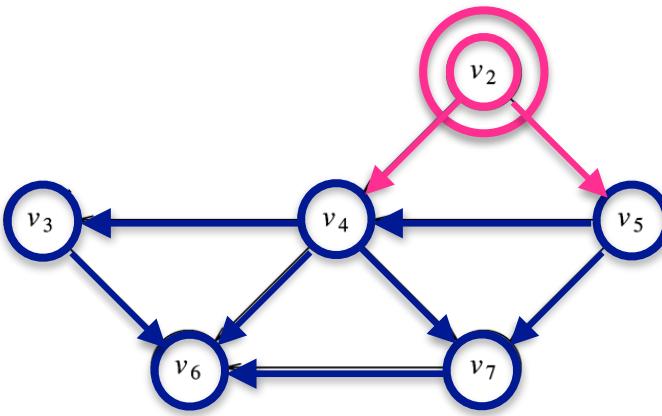
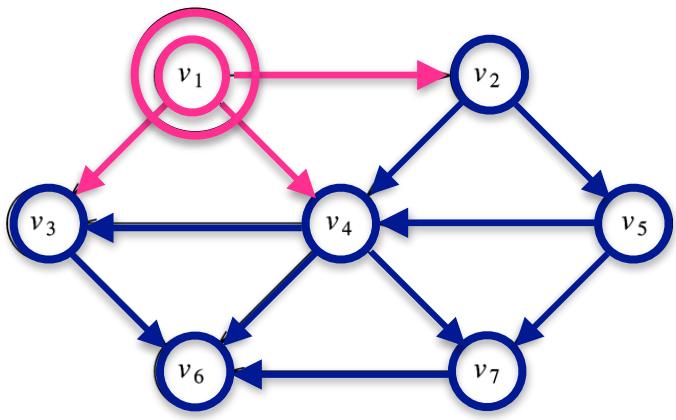
- Finding a vertex with in-degree zero, linear scan of  $V$
  - There are  $|V|$  calls to do this
- Speed up:
    - Use a Queue to store vertices with in-degree zero
    - Only have to remove the front of Queue, skip linear scan of  $V$
    - **Running Time:**  $O(|E| + |V|)$

# Topological Sort



v1, v2, v5, v4, v3, v7, v6

v1, v2, v5, v4, v7, v3, v6

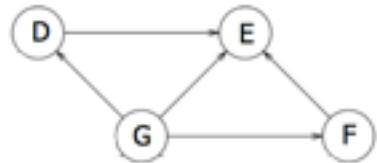
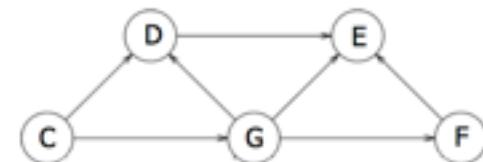
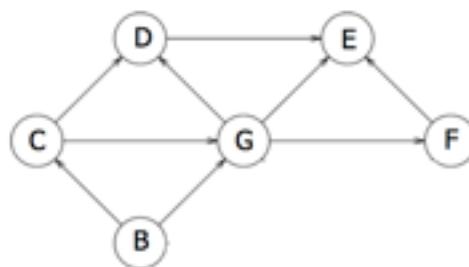
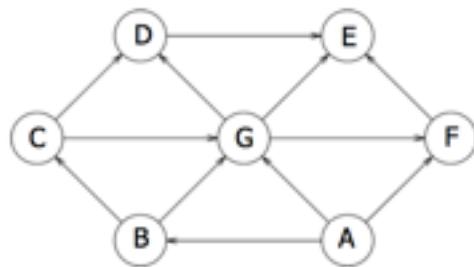


v1, v2, v5, v4, v3, v7, v6

v1, v2, v5, v4, v7, v3, v6



# Topological Sort

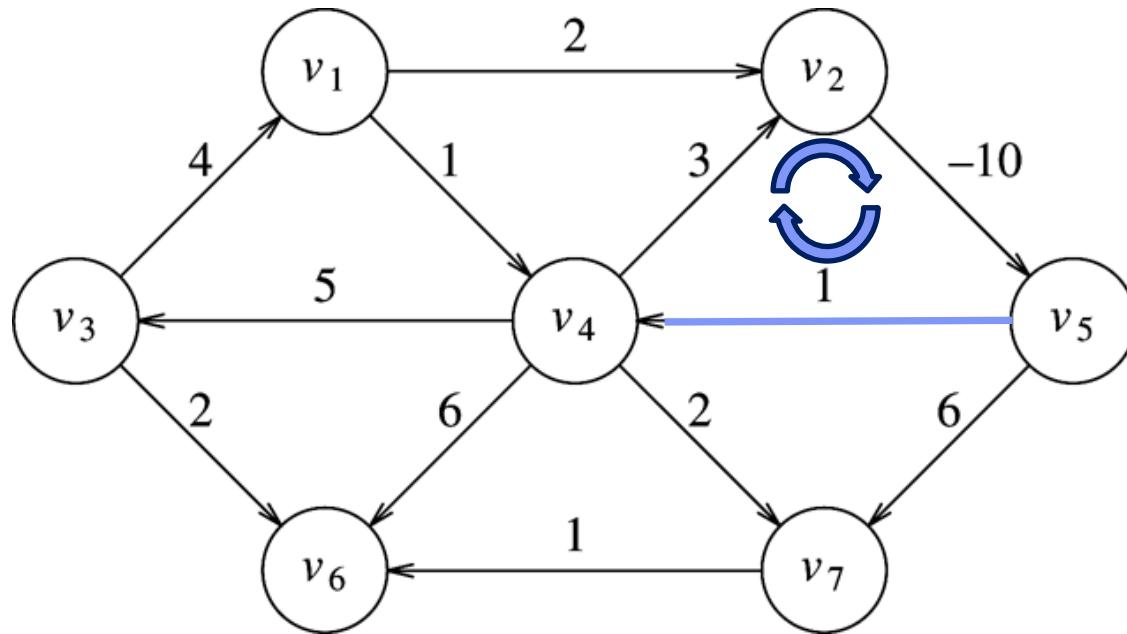


A, B, C, G, D, E, F

# Single-Source Shortest Path

- **Problem:** find the path between two vertices that minimizes the number of edges or minimizes the sum of the weights of the edges.
  - Unweighted Graphs – number of edges
  - Weighted Graphs – sum of the costs
- **BFS** – solves for unweighted graphs
- **Dijkstra's** – solves for directed graph with positive weights
- **Bellman Ford** – solves for graphs that can include negative weights

# Negative Cost Cycle



Shortest Path **NOT** defined

$\text{BFS}(V, E, s)$

**for** each  $u \in V - \{s\}$

$u.d = \infty$

$s.d = 0$

$Q = \emptyset$

    ENQUEUE( $Q, s$ )

**while**  $Q \neq \emptyset$

$u = \text{DEQUEUE}(Q)$

**for** each  $v \in G.\text{Adj}[u]$

**if**  $v.d == \infty$

$v.d = u.d + 1$

                ENQUEUE( $Q, v$ )

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

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

$v.d = \infty$

$v.\pi = \text{NIL}$

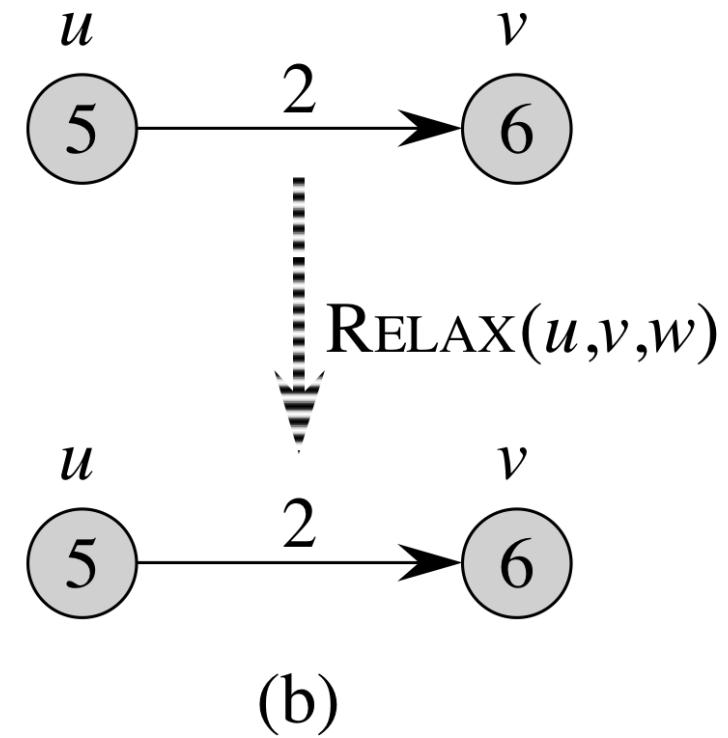
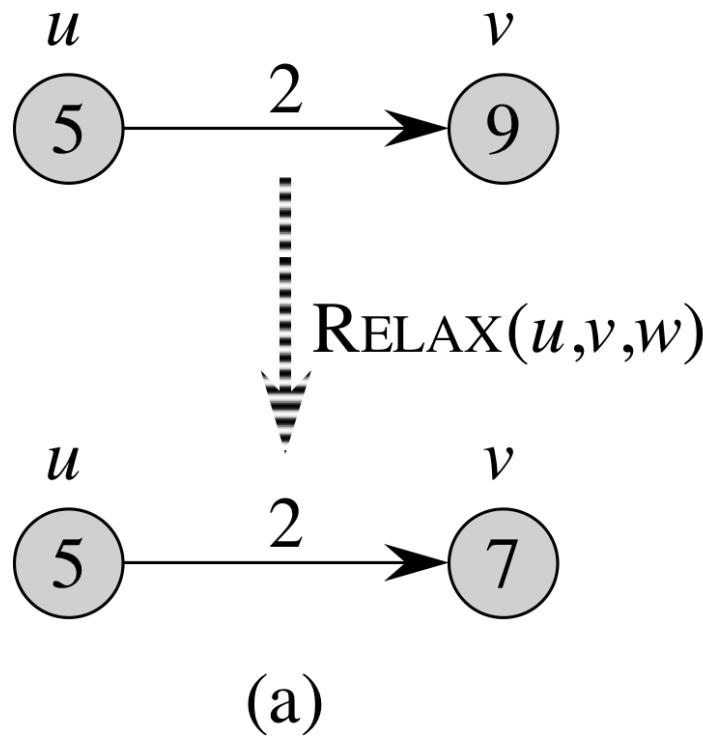
$s.d = 0$

Single-pair shortest path initialize distances step.

RELAX( $u, v, w$ )

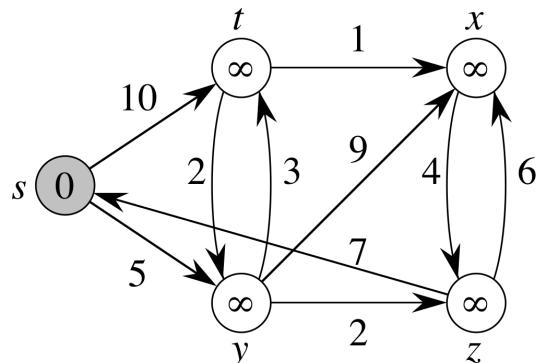
**if**  $v.d > u.d + w(u, v)$   
 $v.d = u.d + w(u, v)$   
 $v.\pi = u$

Single-pair shortest path update distance step.

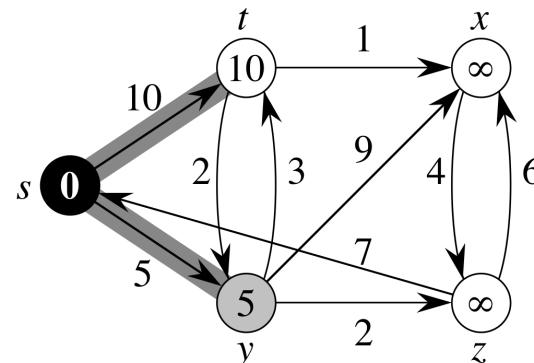


**if**  $v.d < u.d + w_{uv}$   
**then relax**

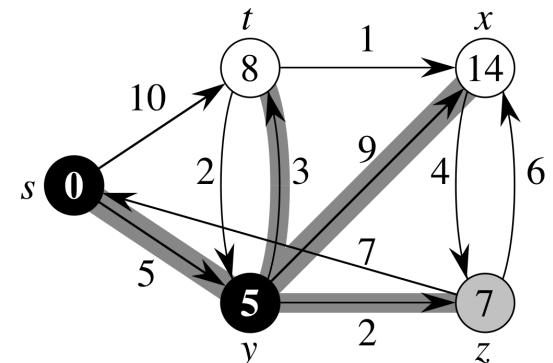
# Dijkstra's - Example



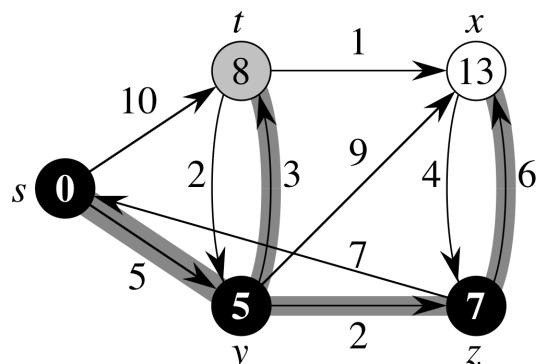
(a)



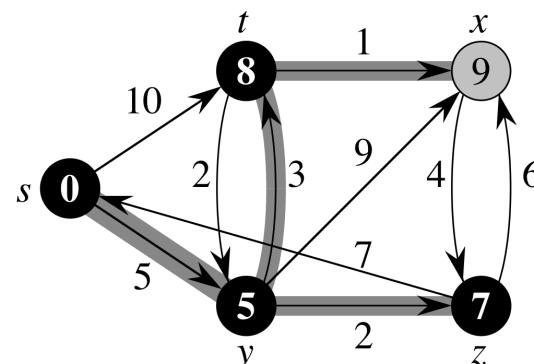
(b)



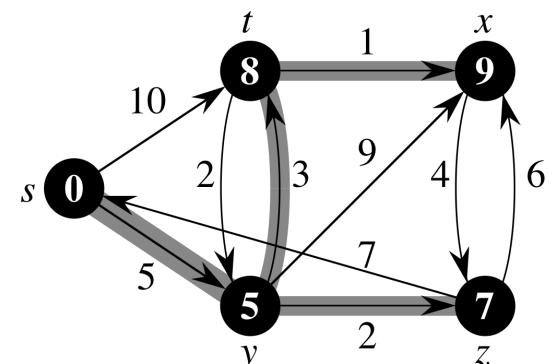
(c)



(d)

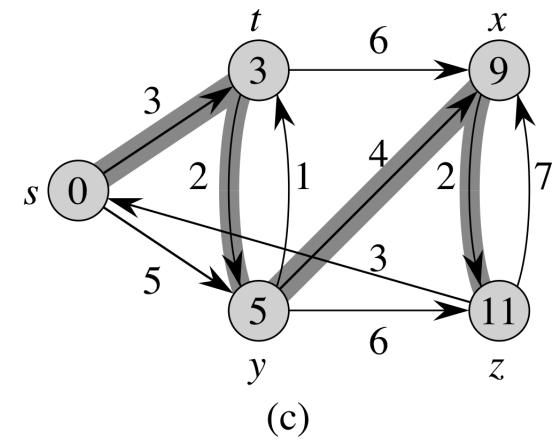
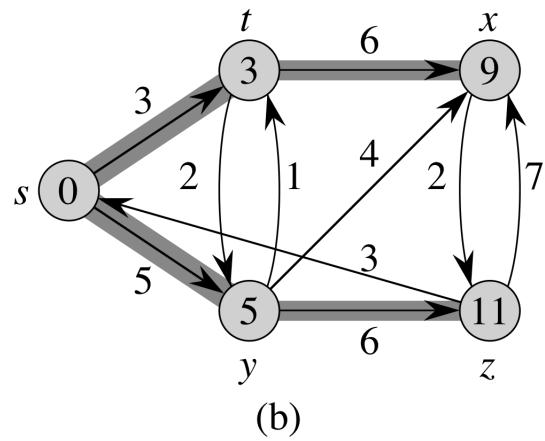
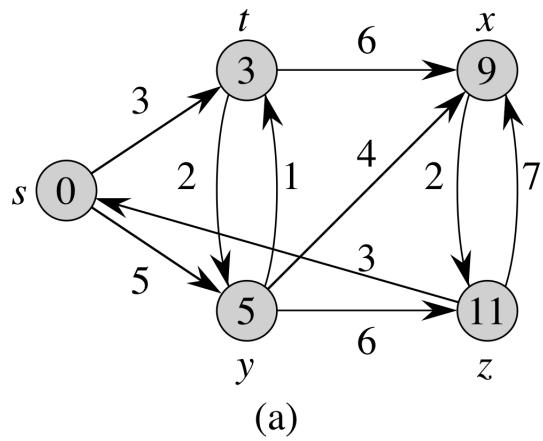


(e)



(f)

# Non-Unique Shortest Path



**BELLMAN-FORD**( $G, w, s$ )

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

**for**  $i = 1$  to  $|G.V| - 1$

**for** each edge  $(u, v) \in G.E$

        RELAX( $u, v, w$ )

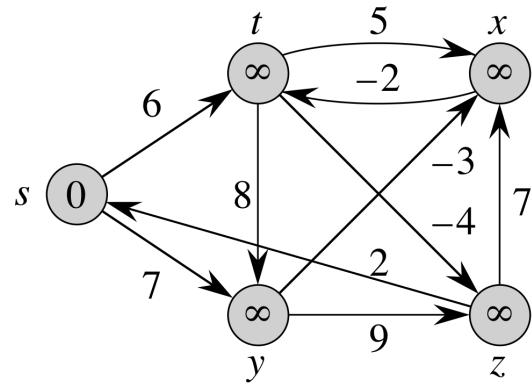
**for** each edge  $(u, v) \in G.E$

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

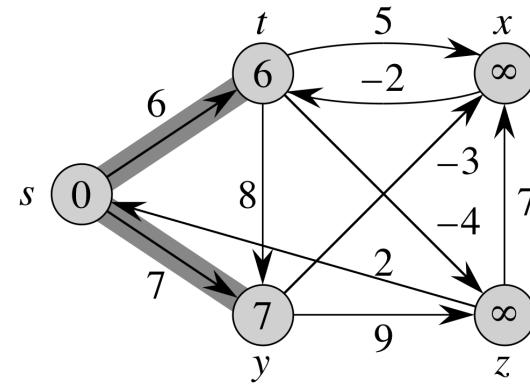
**return** FALSE

**return** TRUE

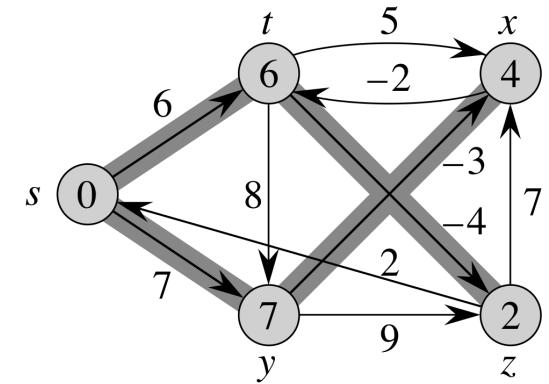
# Bellman Ford - Example



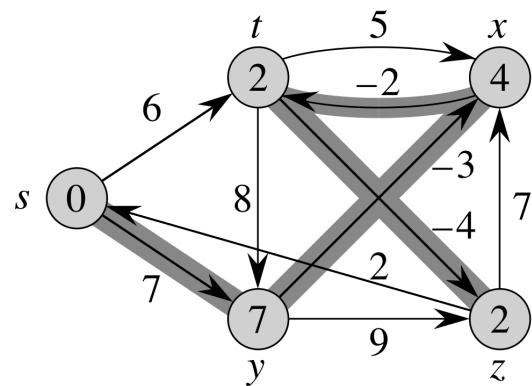
(a)



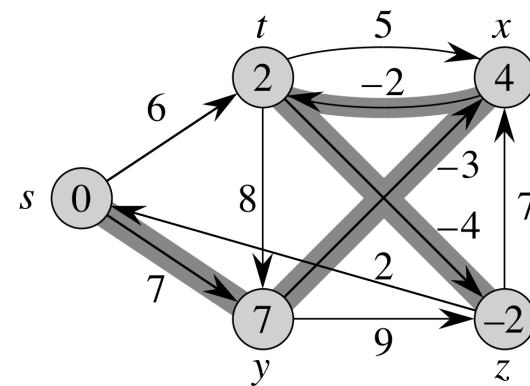
(b)



(c)



(d)



(e)

# BFS - Breadth First Search

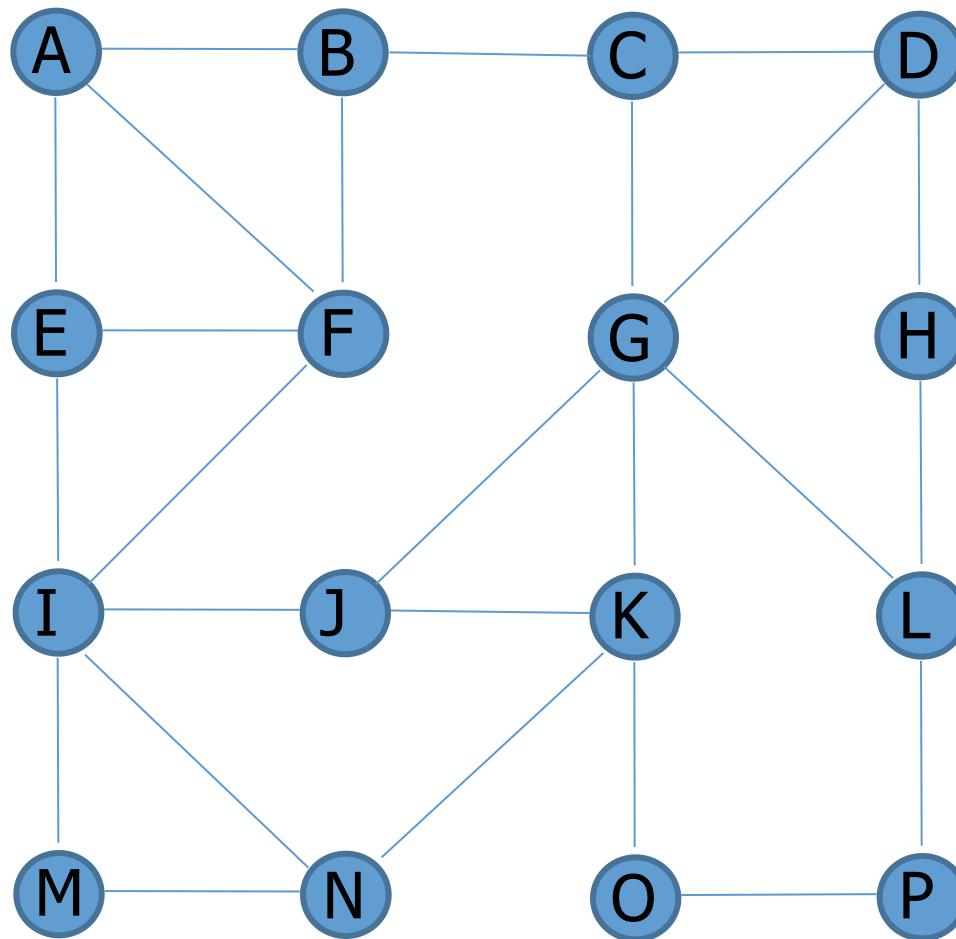
**Process vertices in layers:**

- $s$  – *source vertex*
- $d$  – *destination vertex*
- Vertices 1 edge away from vertex  $s$ , then 2 edges away from  $s$ , etc.

**Each vertex  $d$  holds information:**

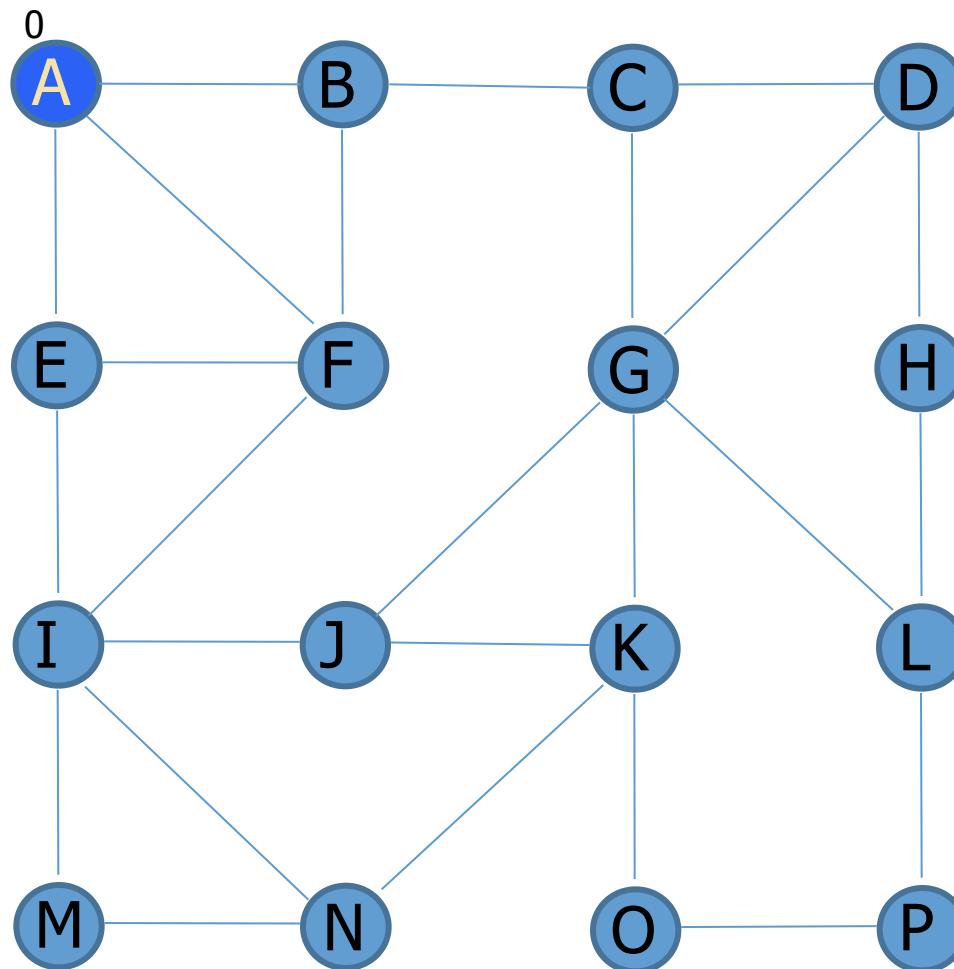
- $Distance(s,d)$ , number of edges in path from  $s$  to  $d$
- $Parent(d)$ , for retrieving path

# BFS



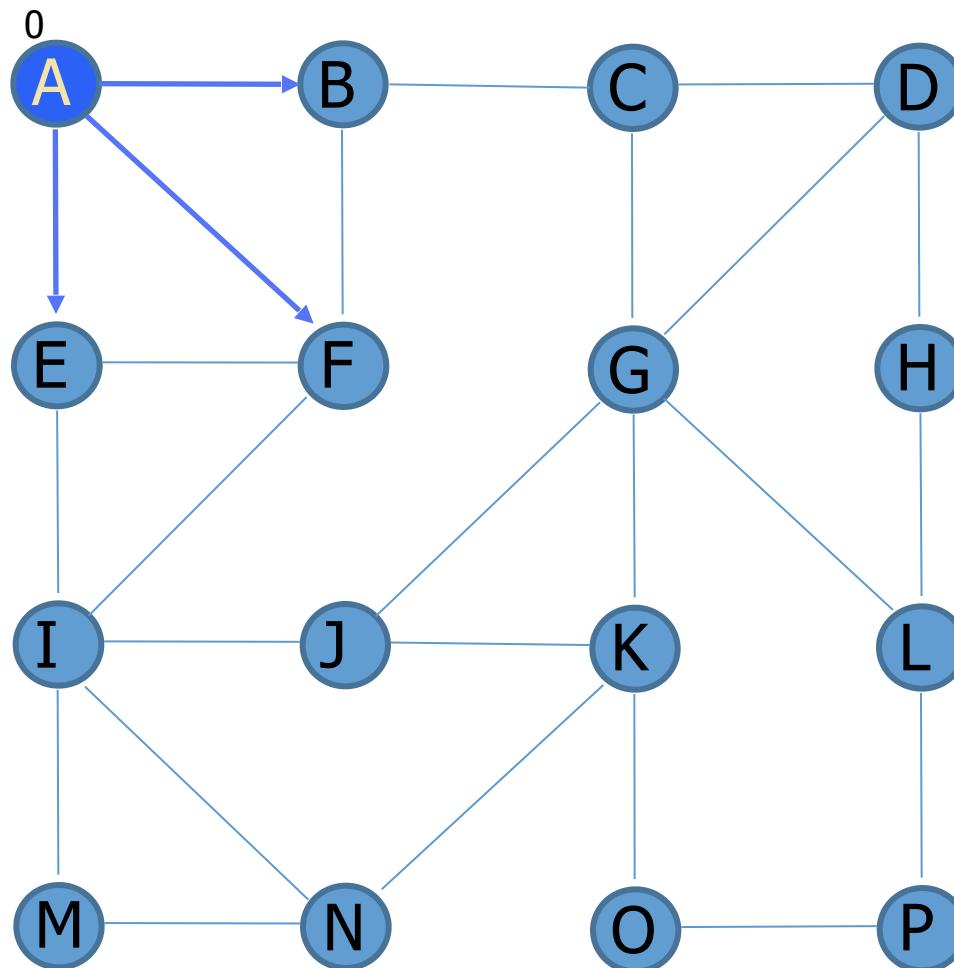
# BFS

A visited

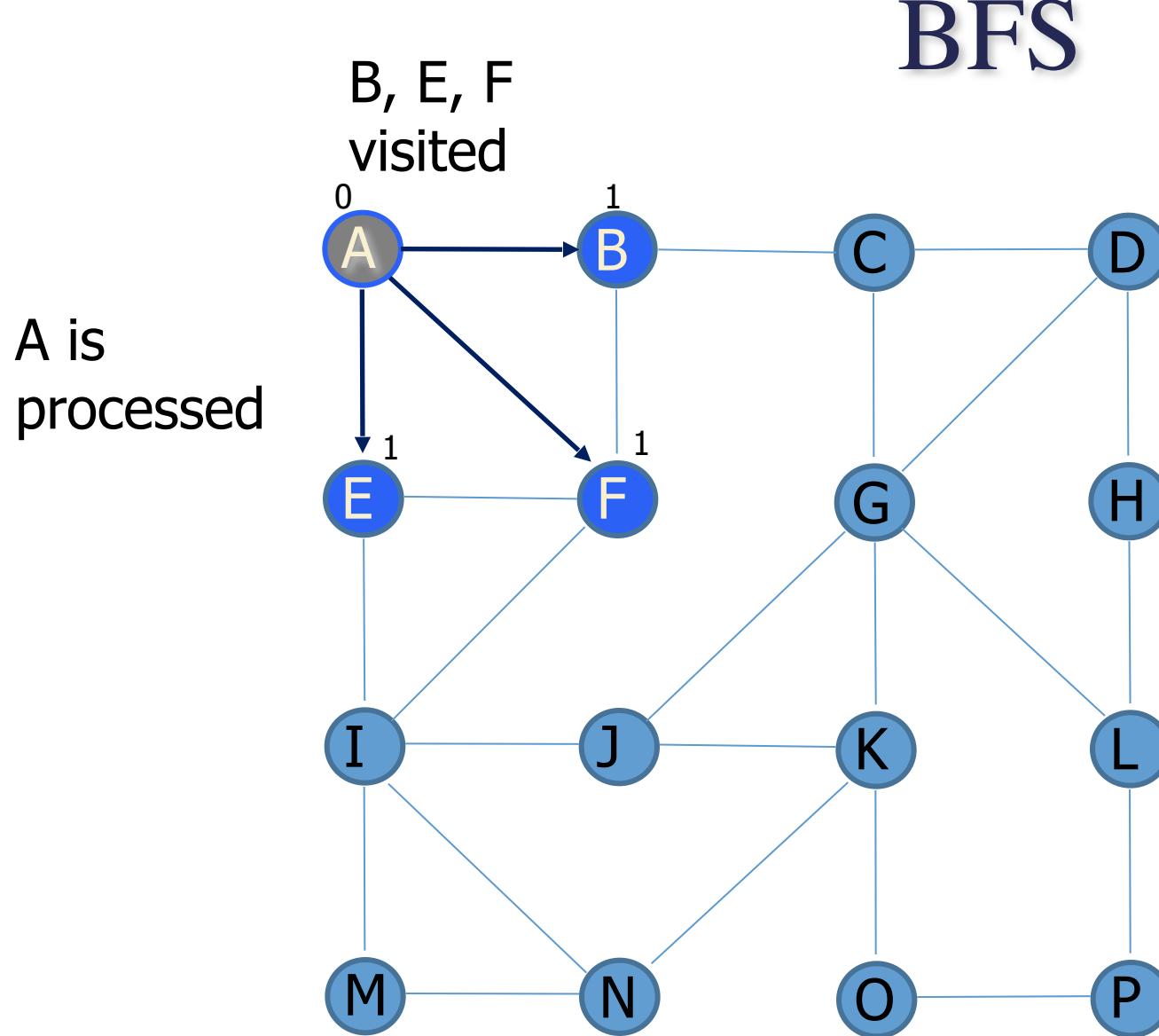


# BFS

Explore  
A's  
neighbors

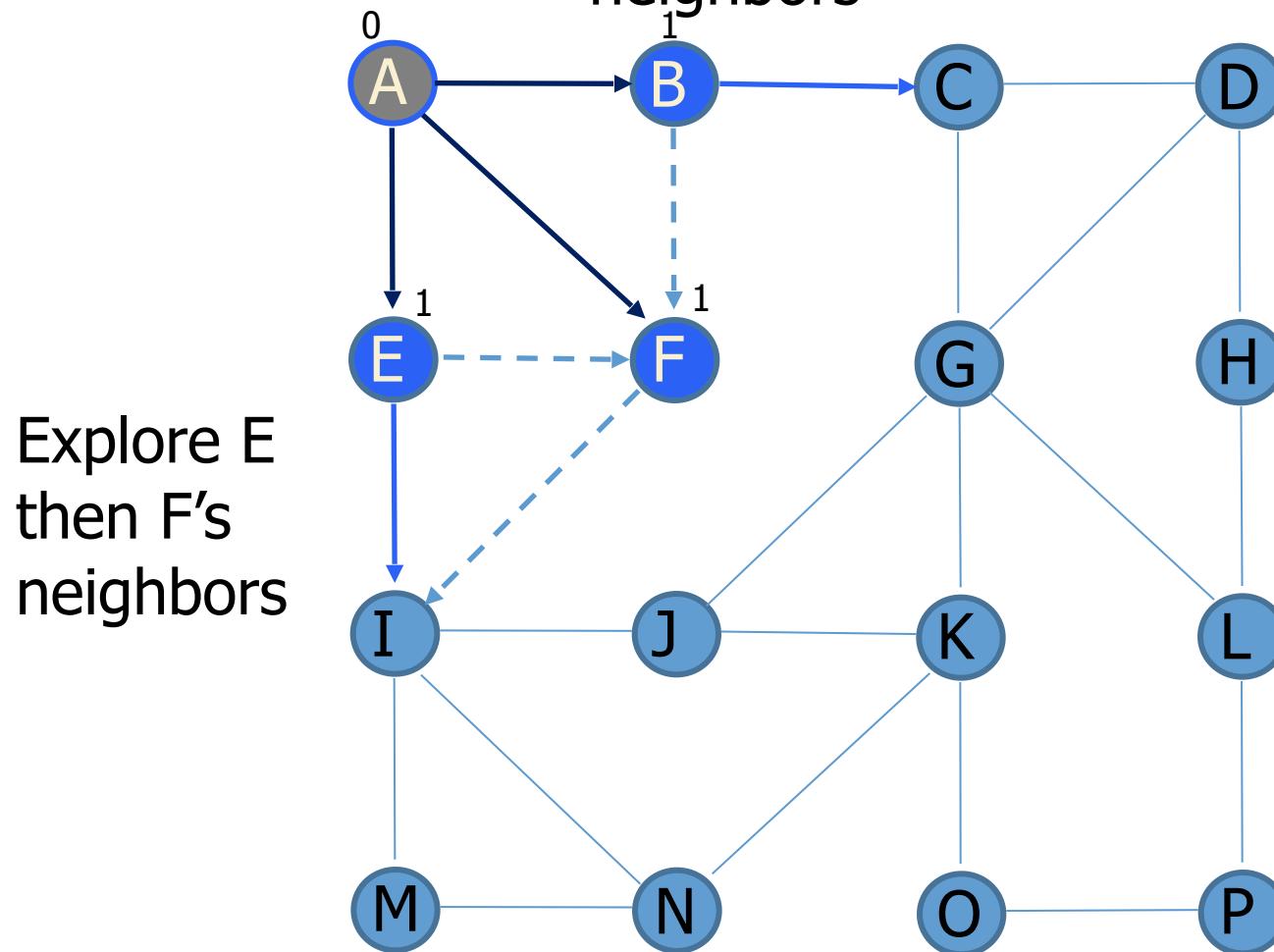


# BFS



# BFS

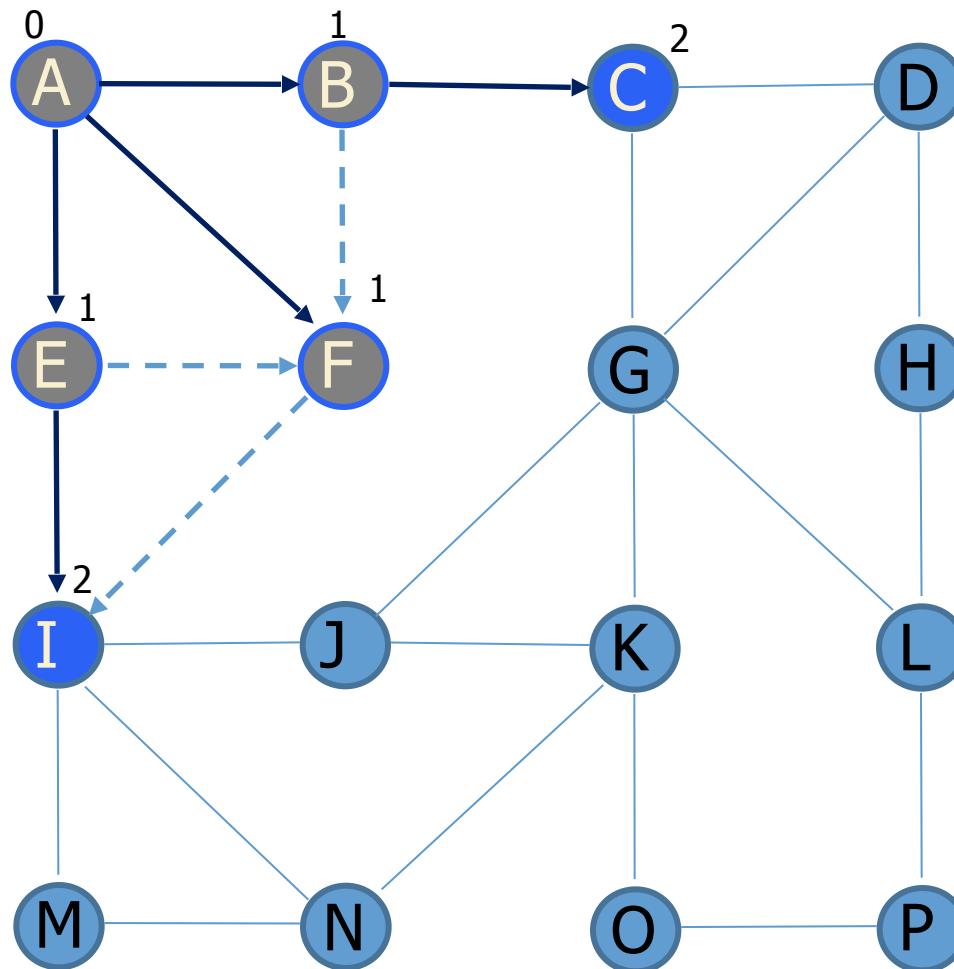
Explore B's  
neighbors



Explore E  
then F's  
neighbors

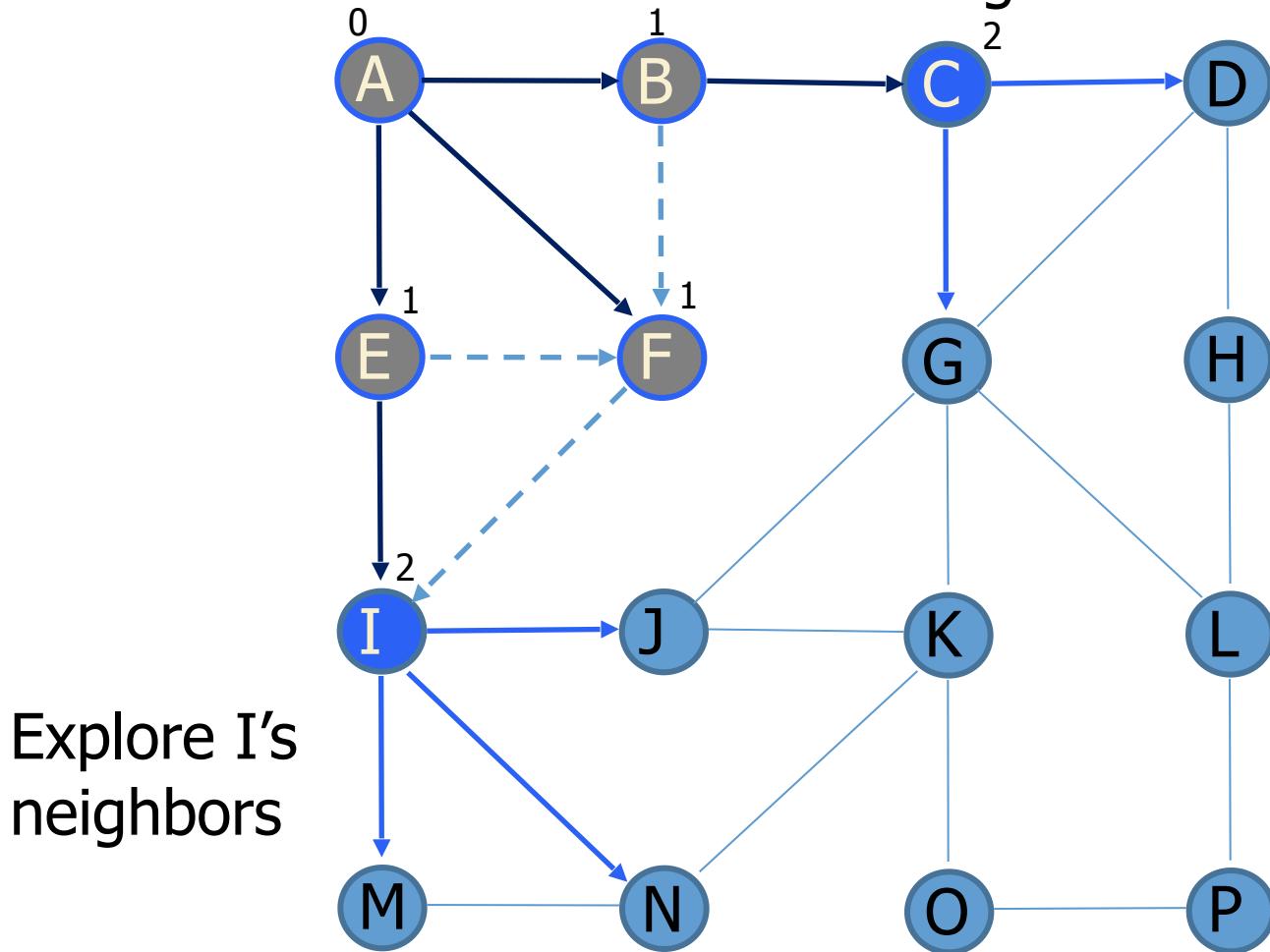
# BFS

C, I  
visited  
B,E,F are  
processed



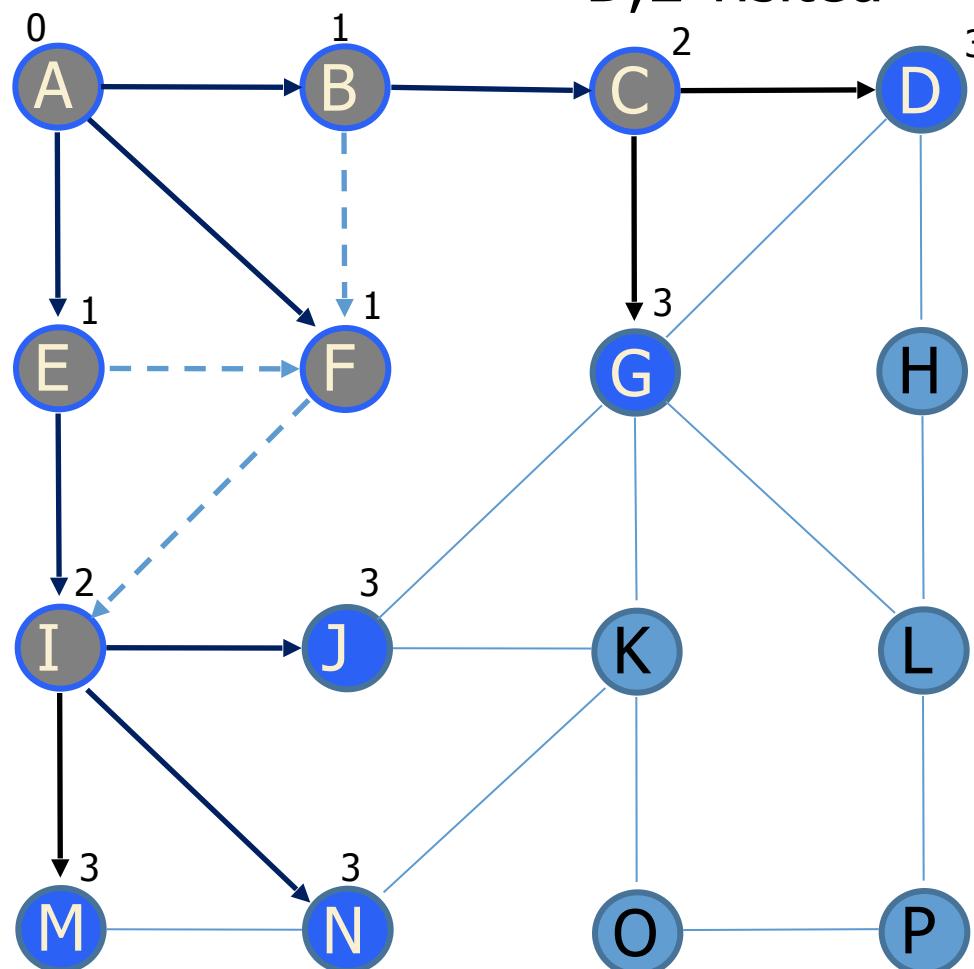
# BFS

Explore C's  
neighbors



# BFS

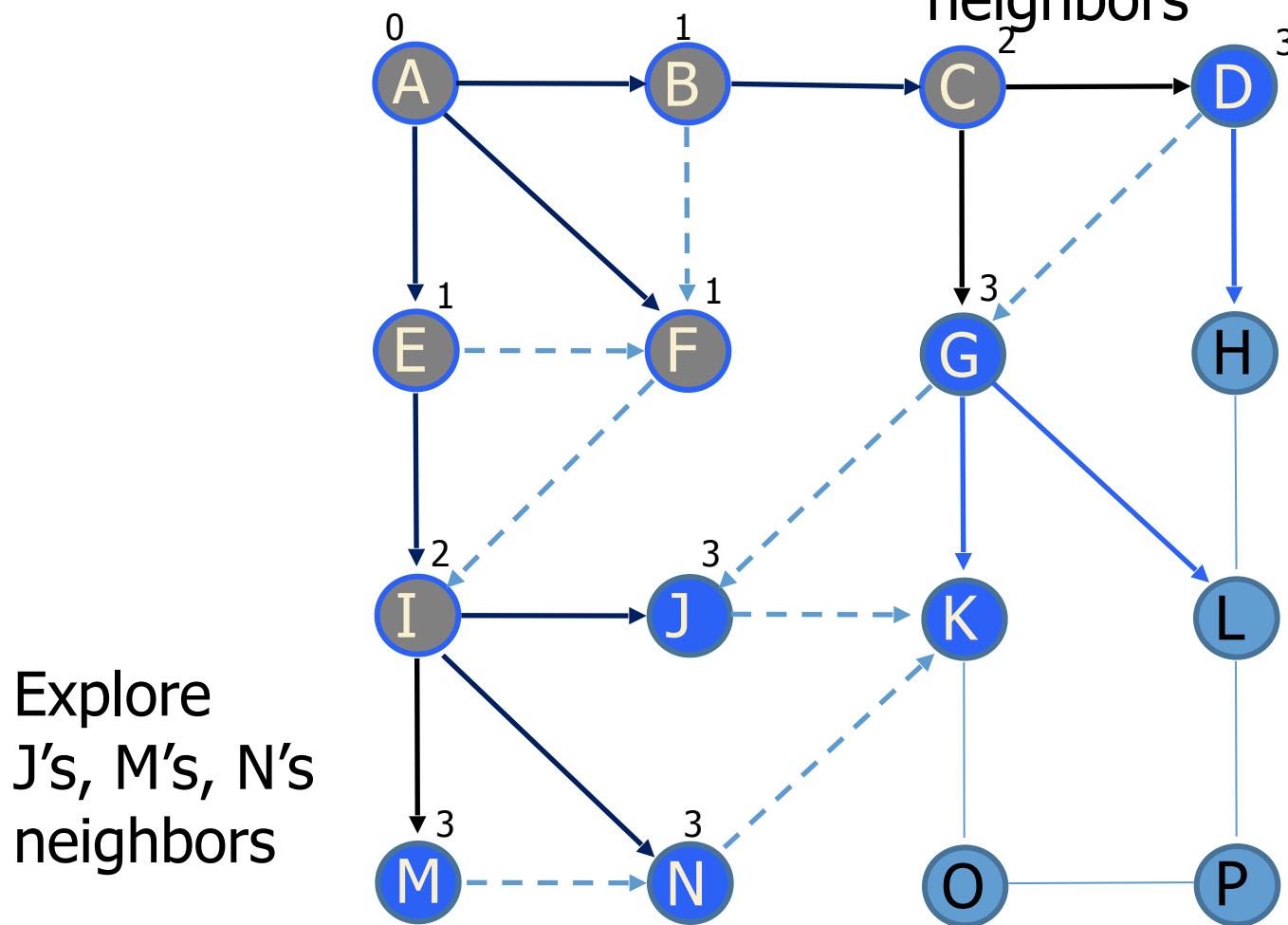
C processed  
D,E visited



I processed  
J,M,N  
visited

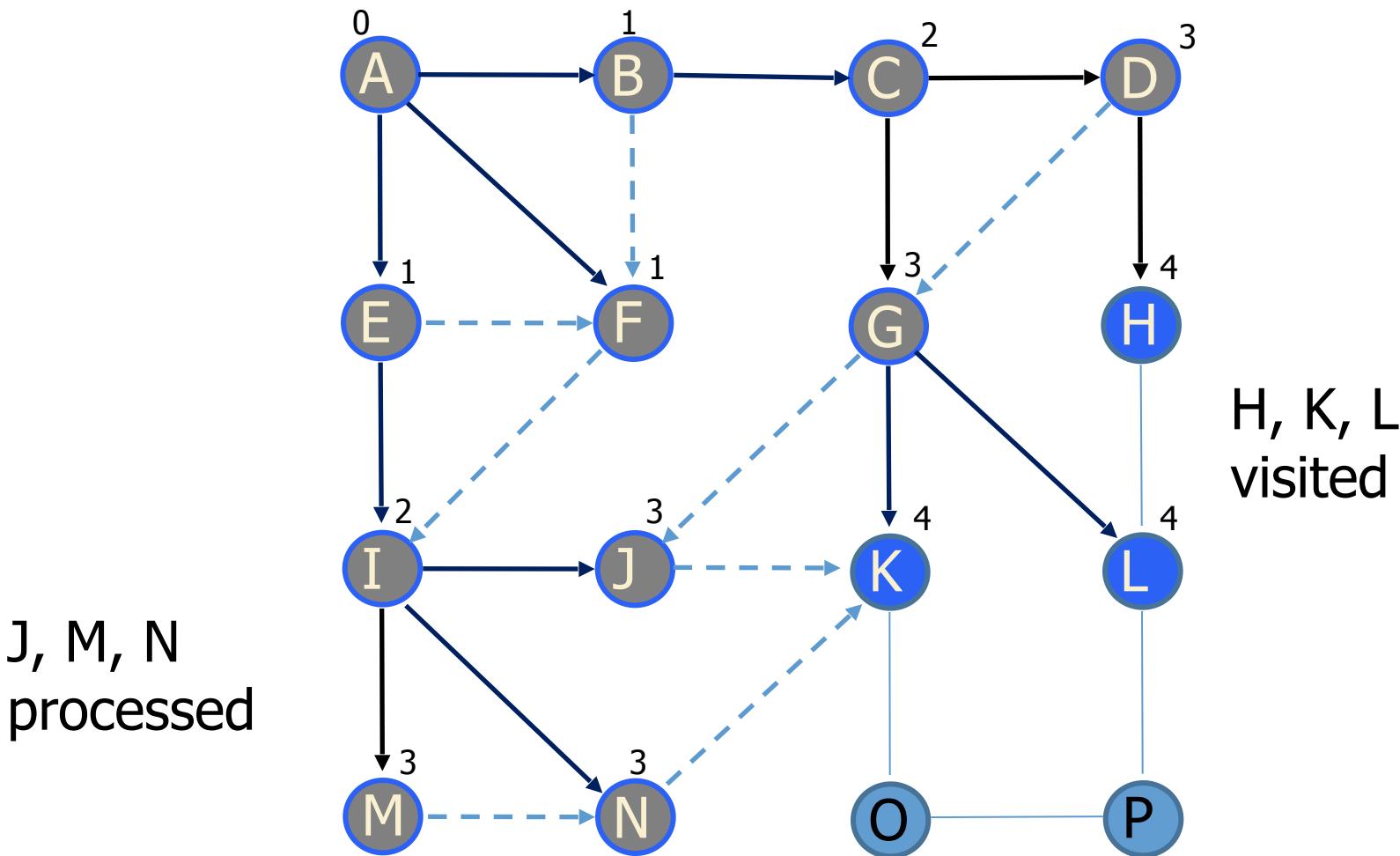
# BFS

Explore  
D's, G's  
neighbors



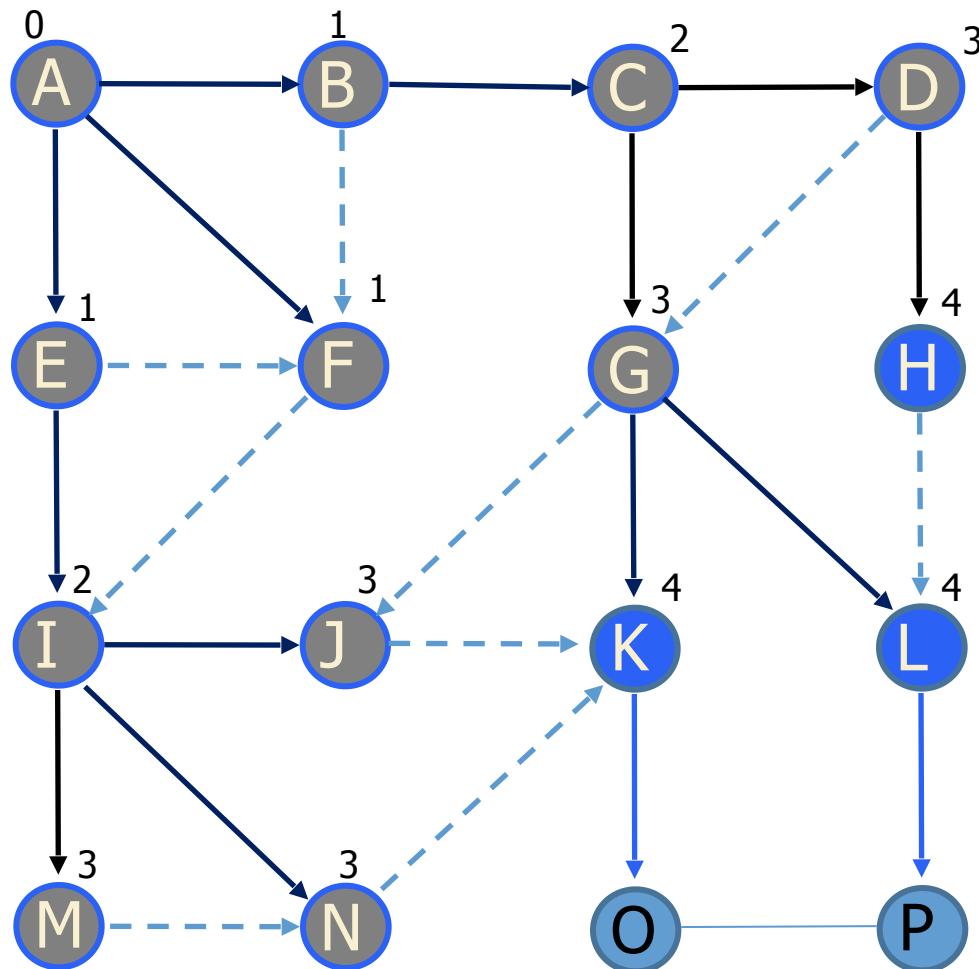
# BFS

D, G  
processed



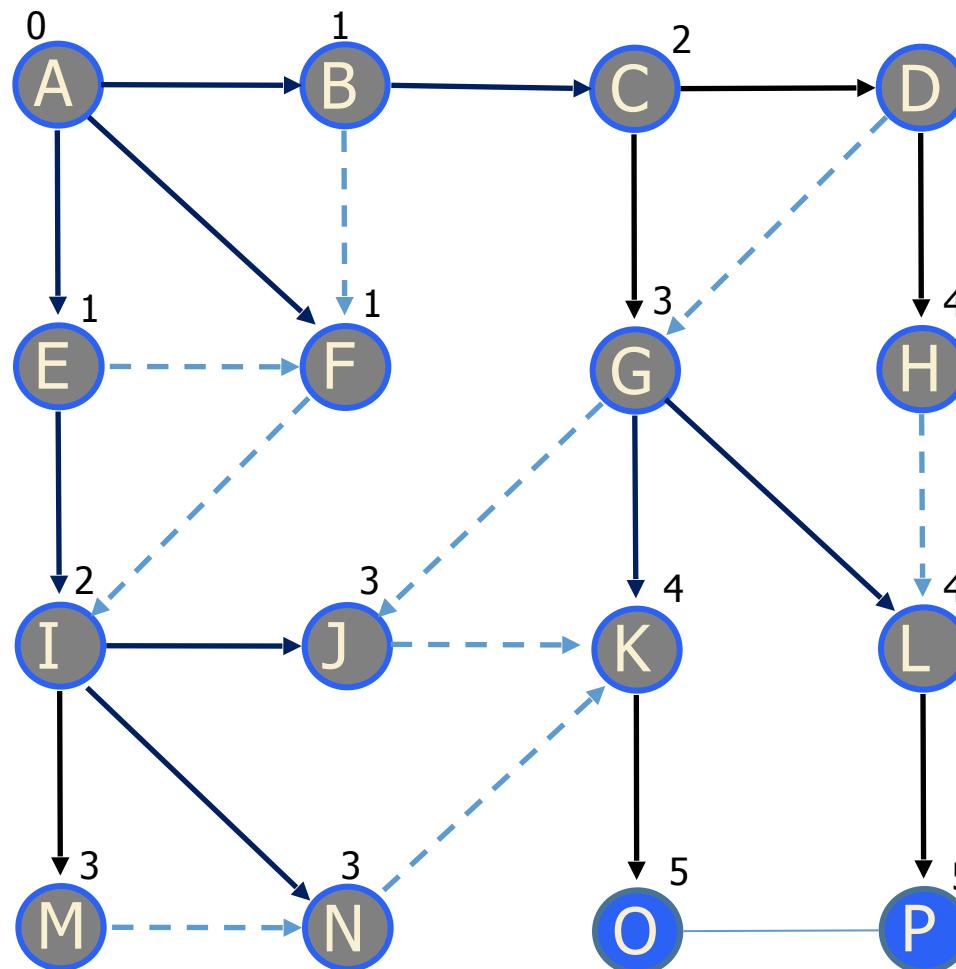
J, M, N  
processed

# BFS



Explore  
H, K, L  
neighbors

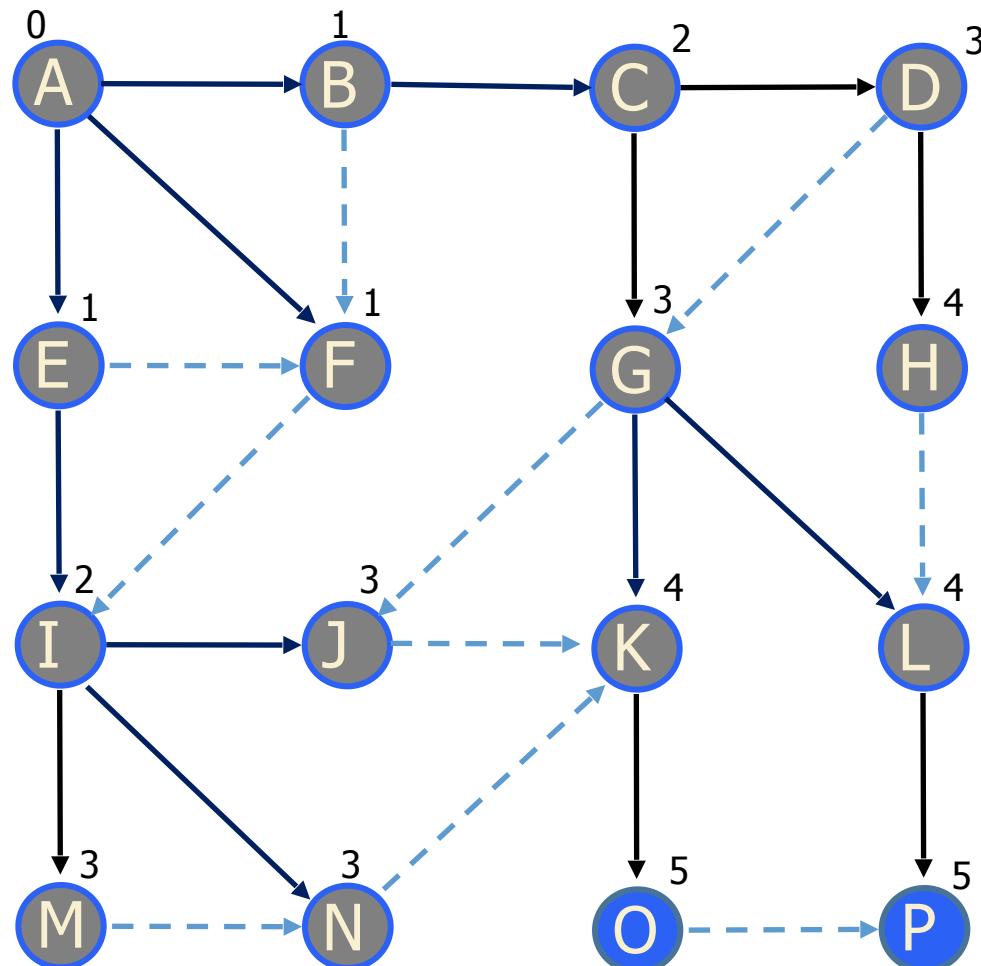
# BFS



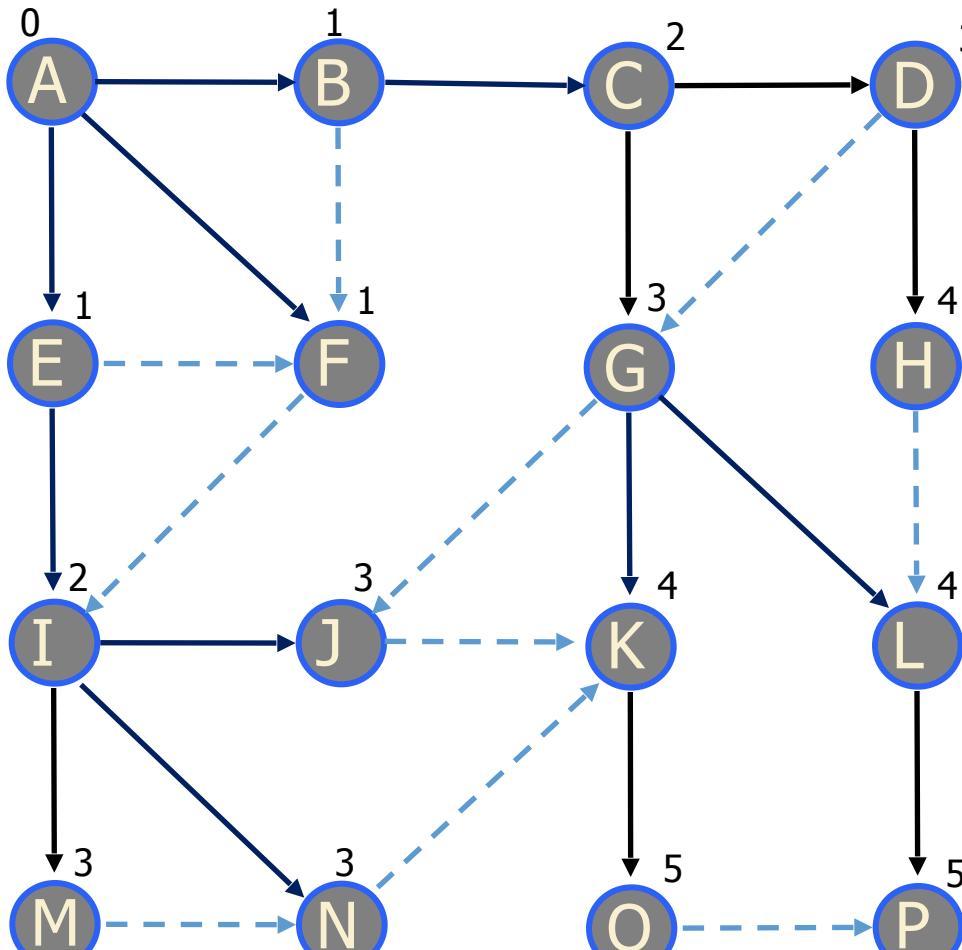
H, K, L  
processed

O, P  
visited

# BFS

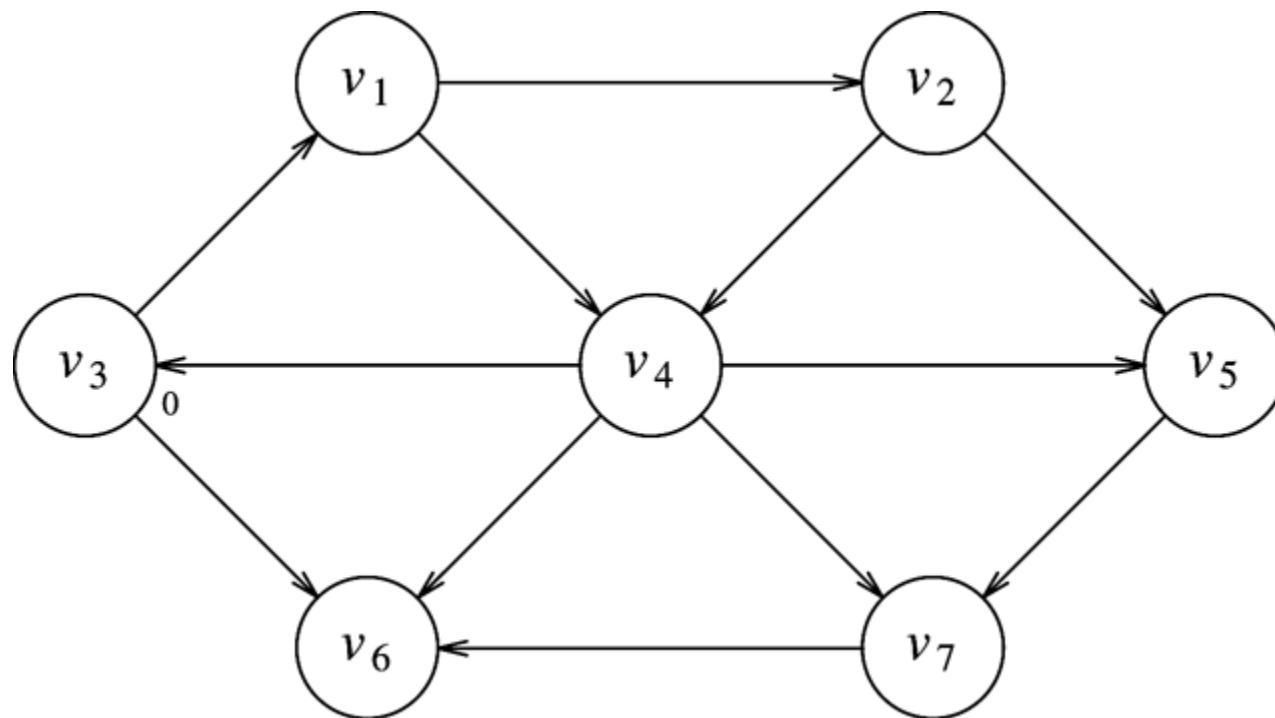


# BFS

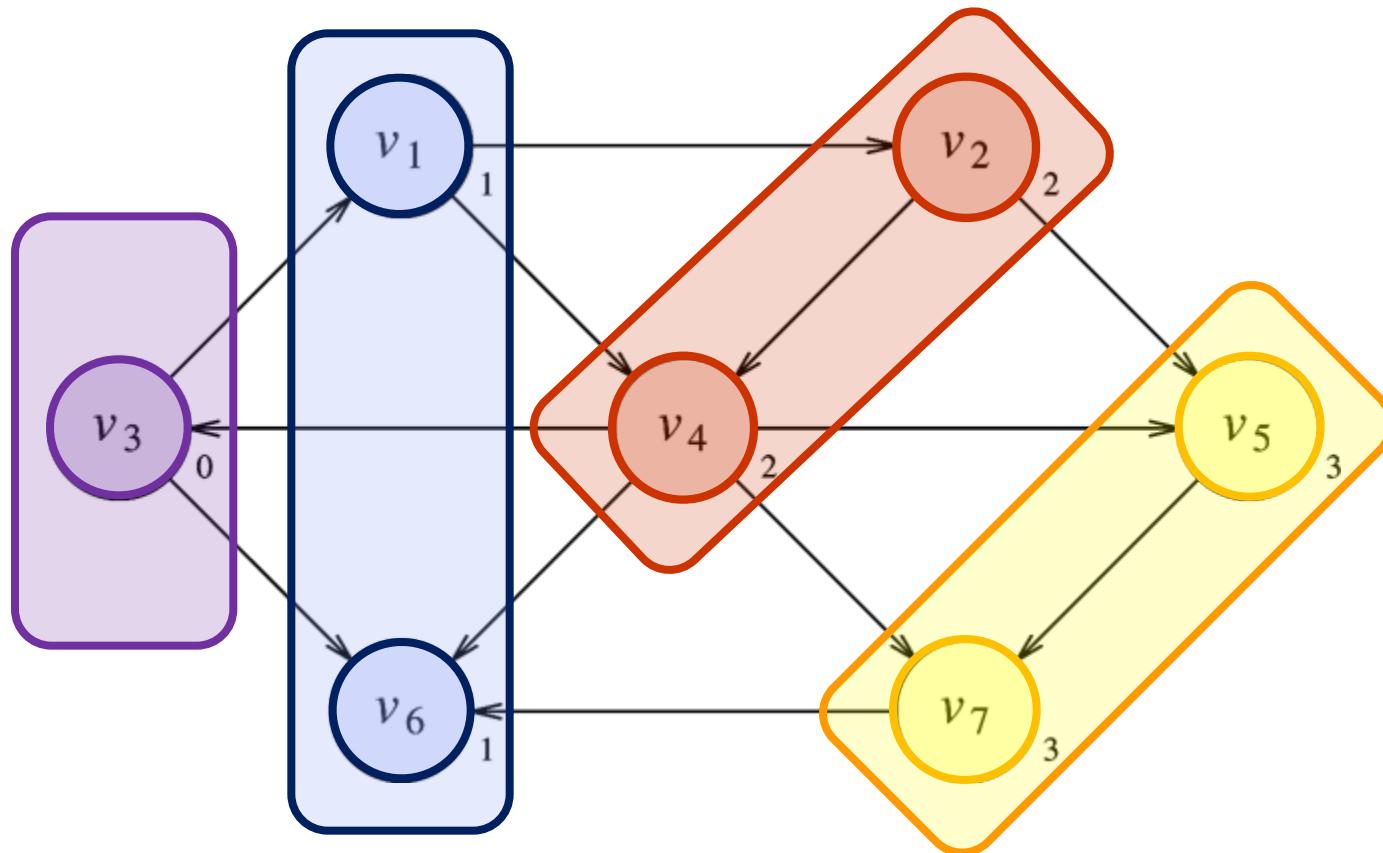


Running Time =  $O(|E| + |V|)$

# BFS – In Class Exercise

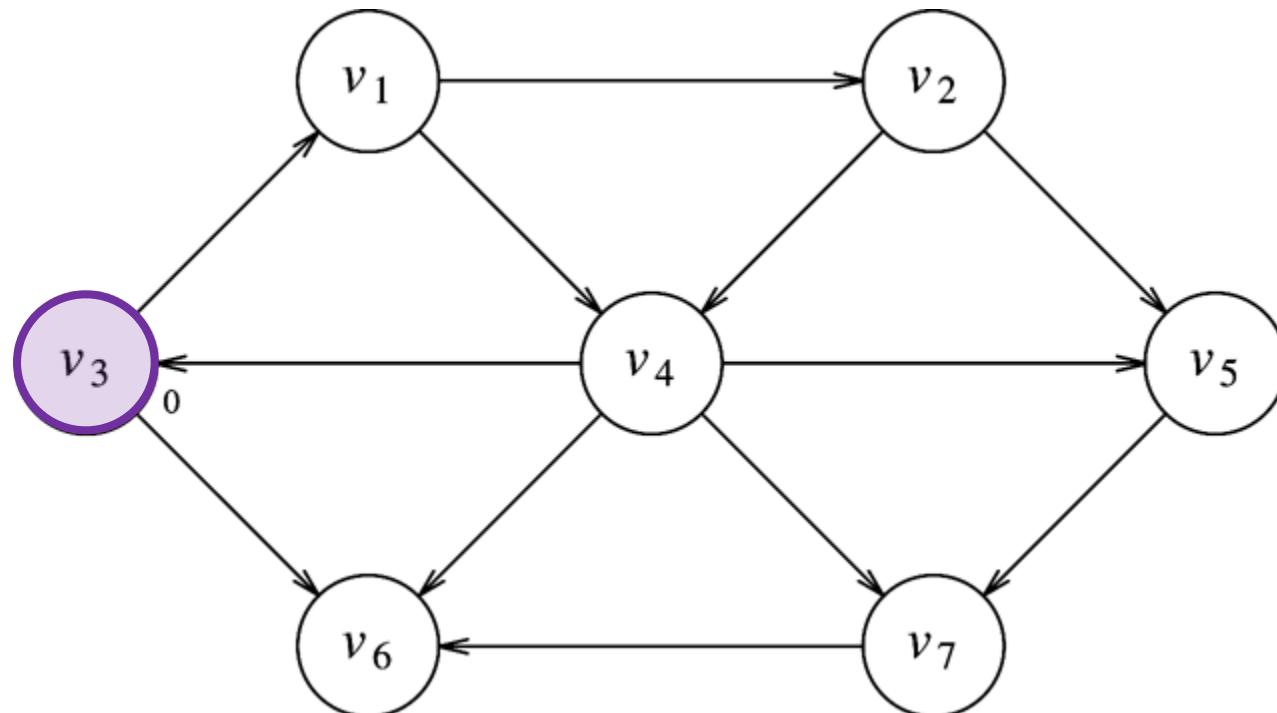


# BFS – In Class Exercise

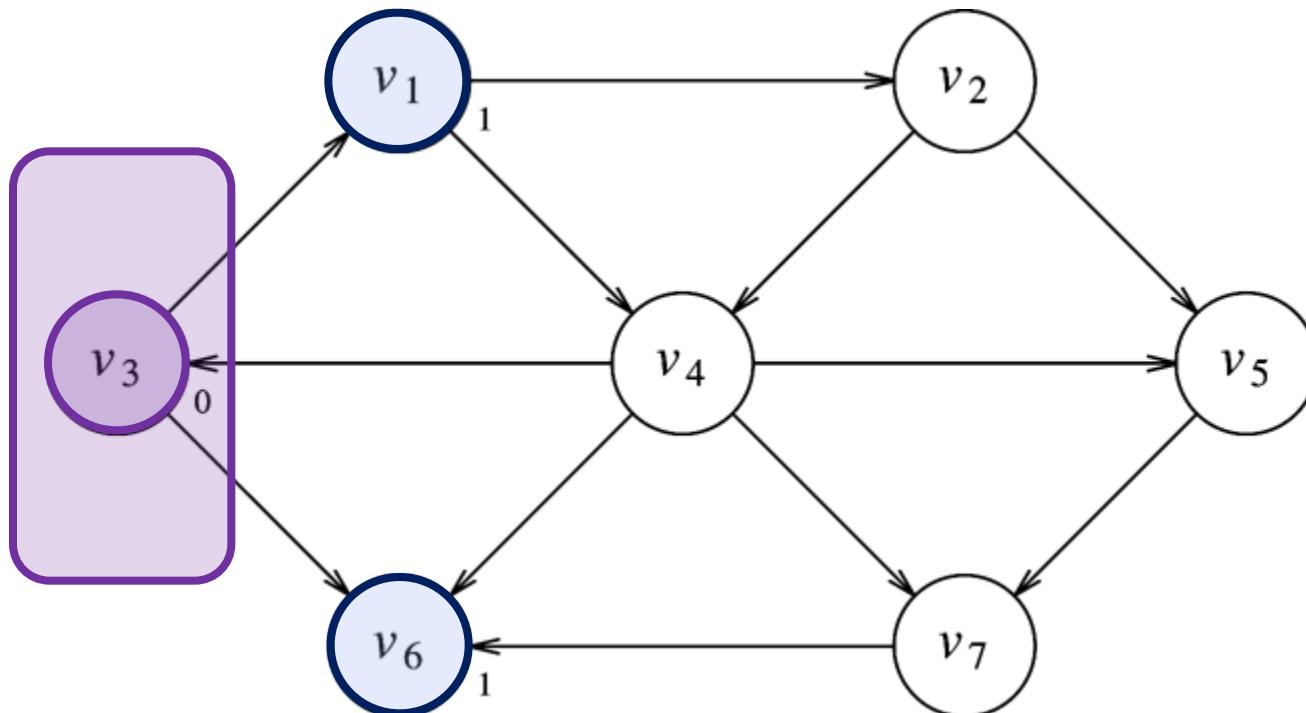


Running Time =  $O(|E| + |V|)$ , every V is explored, all E in worst case.

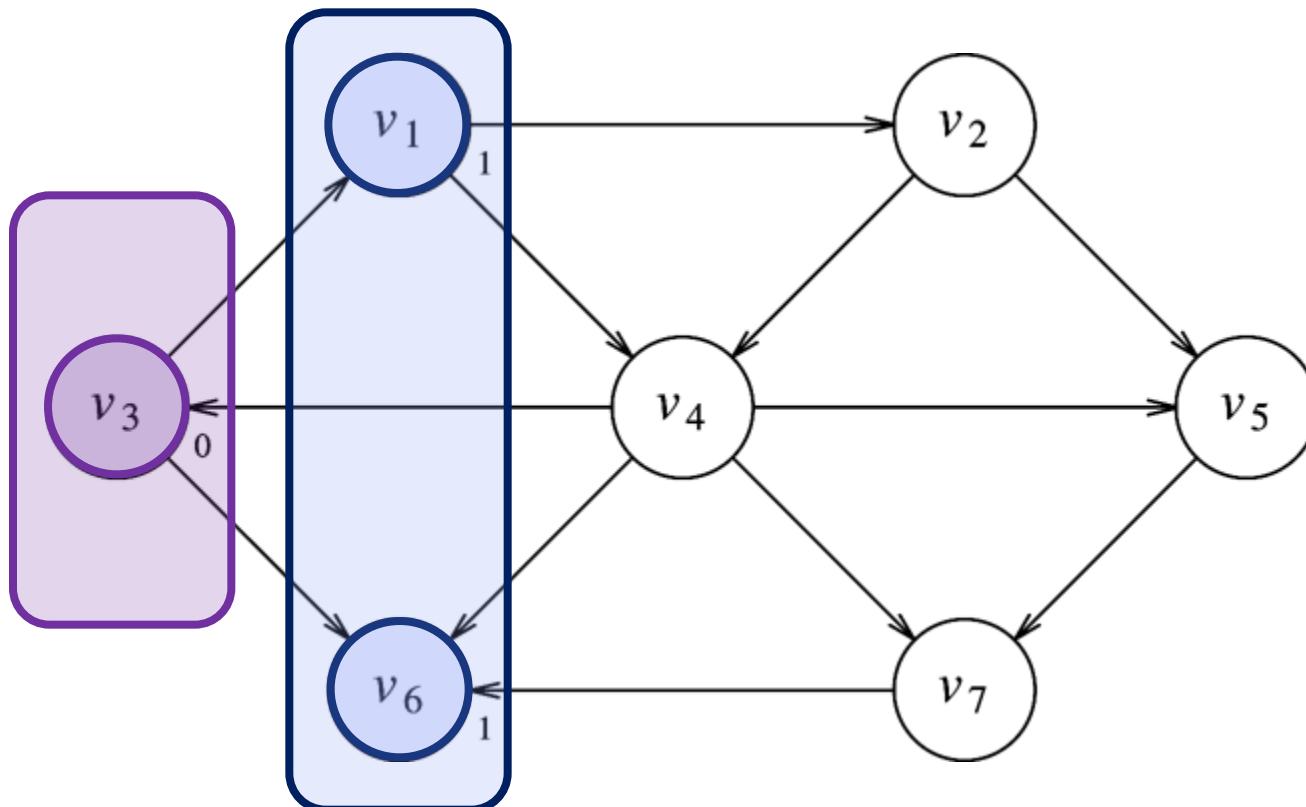
# BFS – In Class Exercise



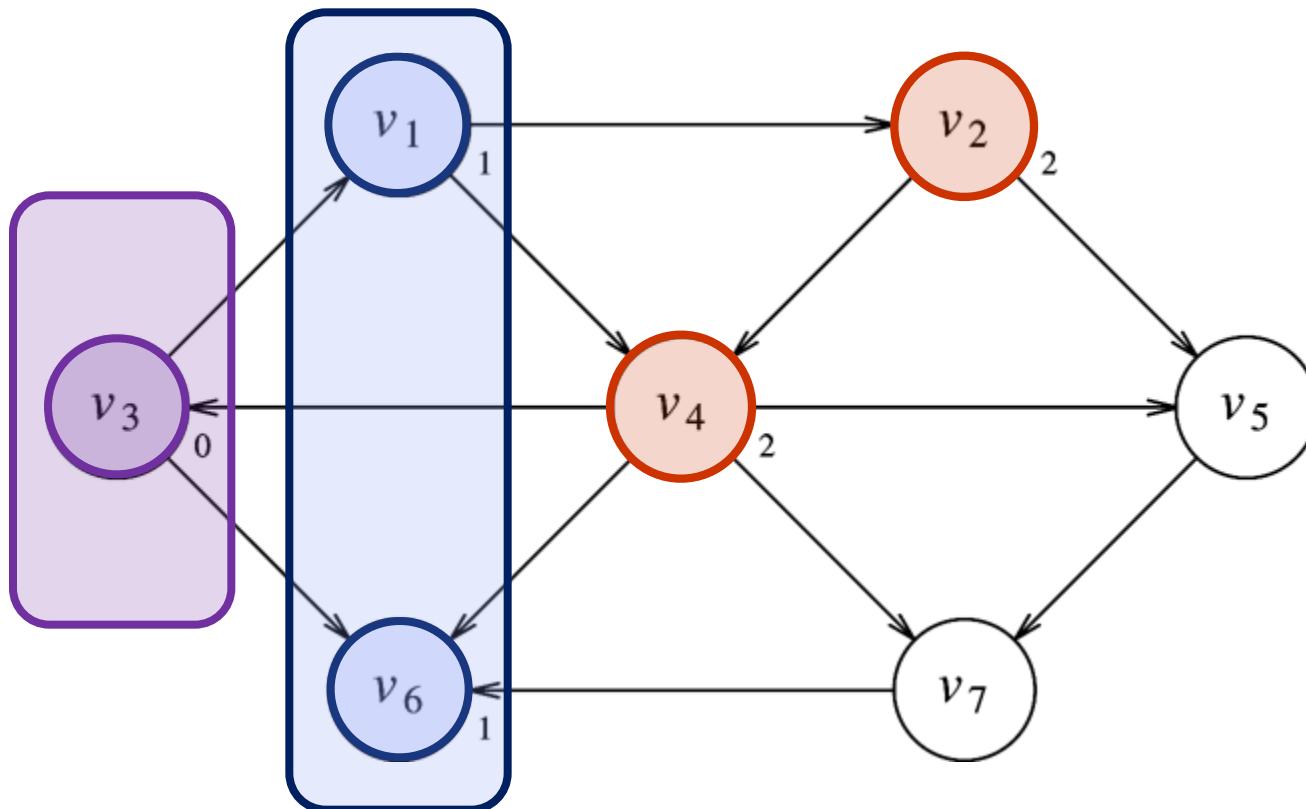
# BFS – In Class Exercise



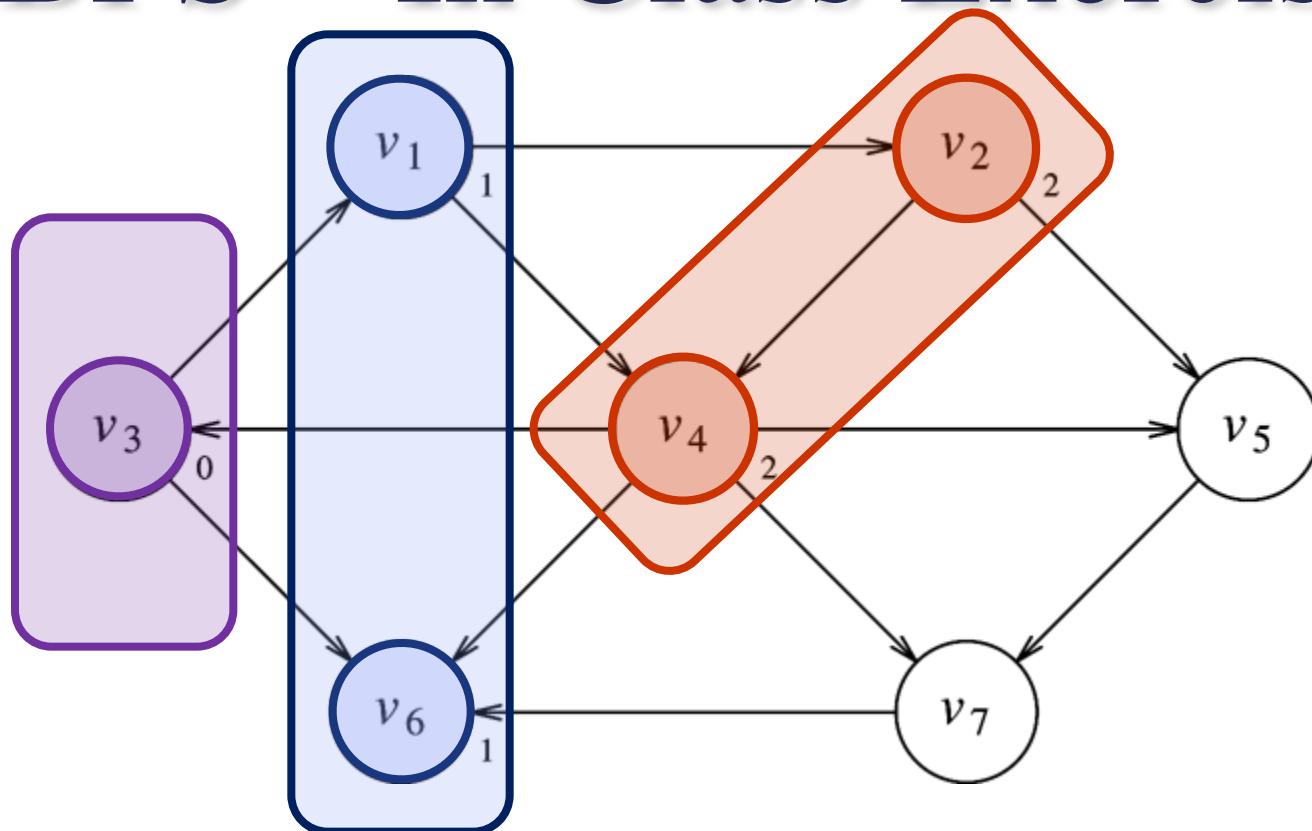
# BFS – In Class Exercise



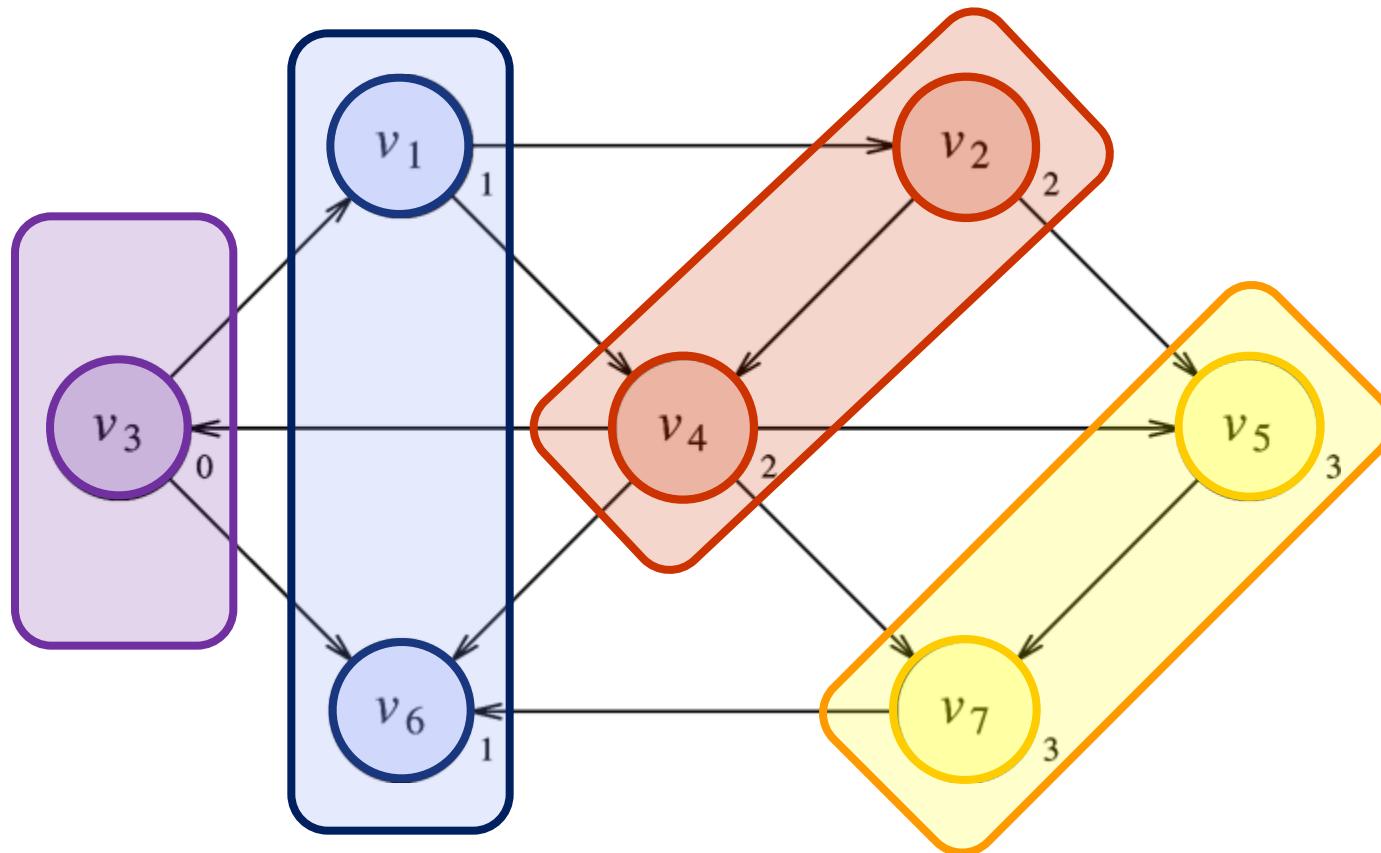
# BFS – In Class Exercise



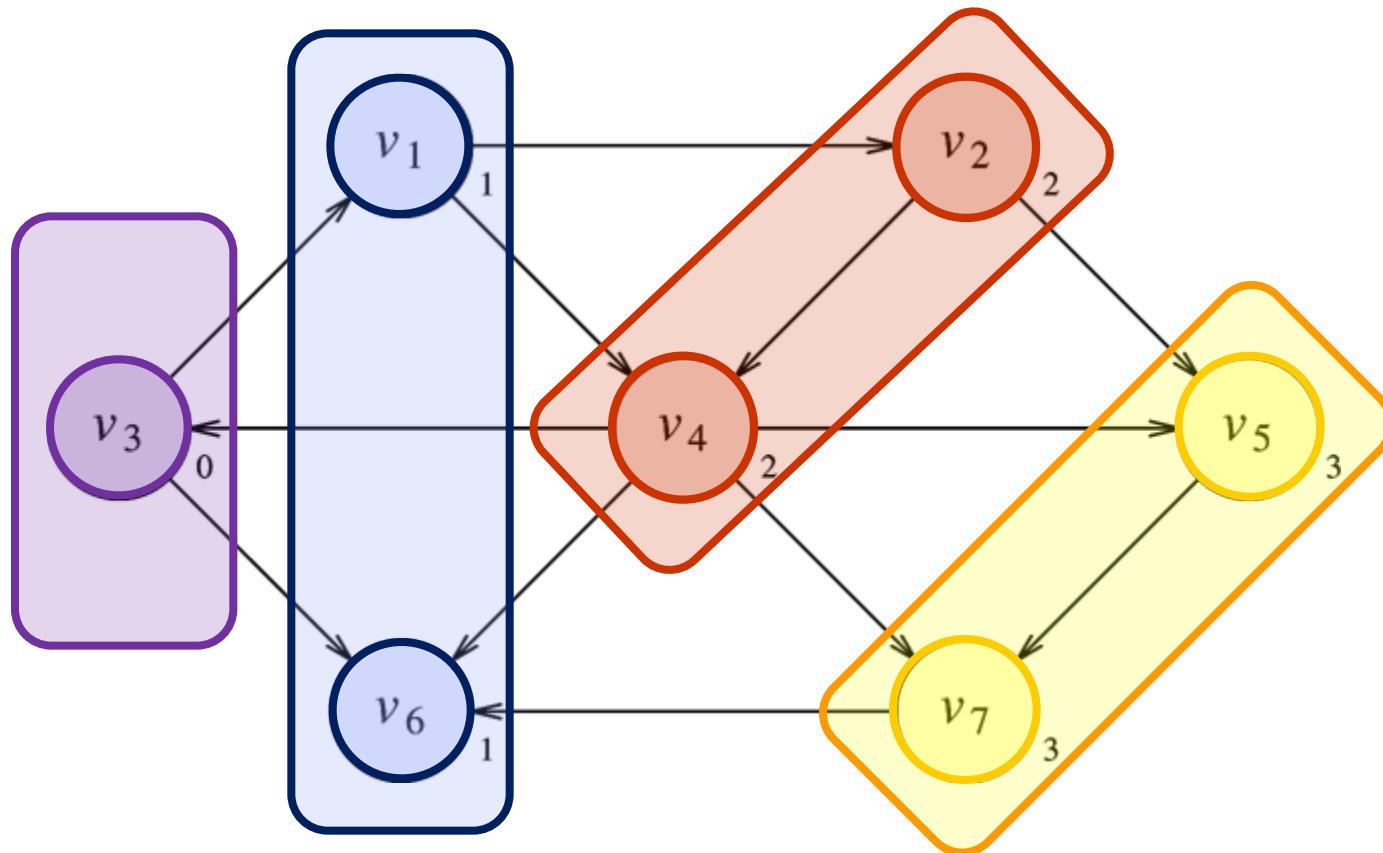
# BFS – In Class Exercise



# BFS – In Class Exercise



# BFS – In Class Exercise



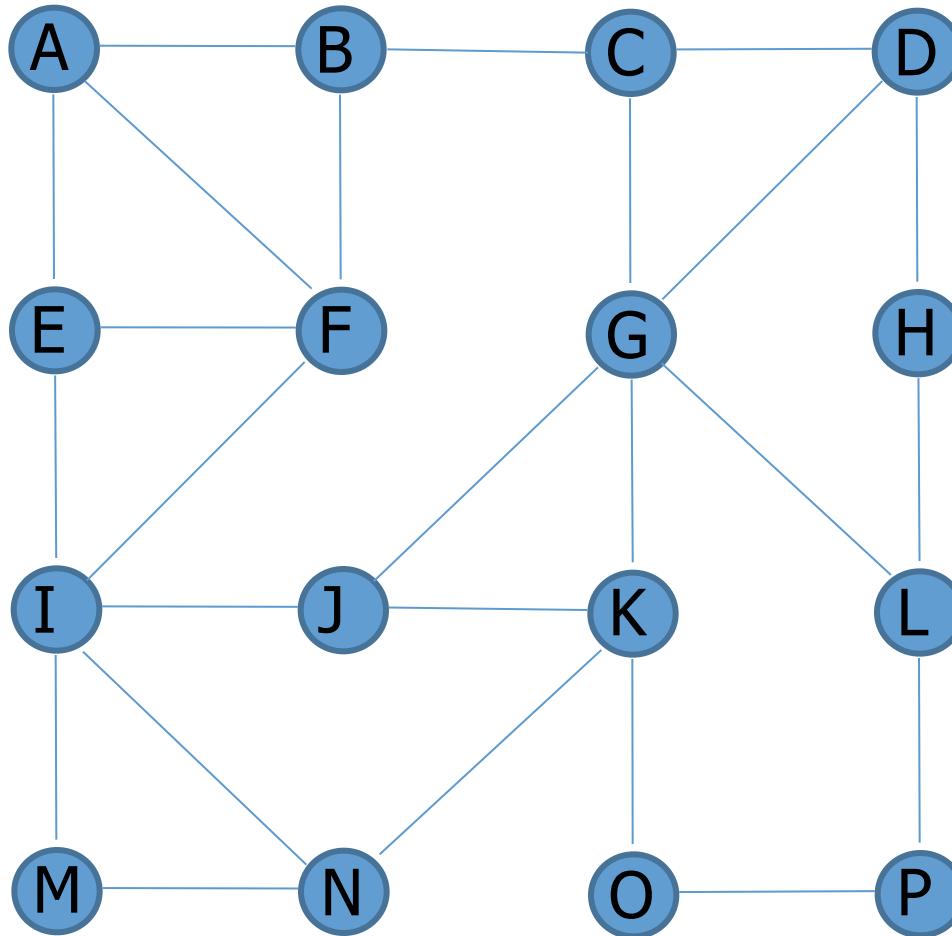
$$\text{Running Time} = O(|E| + |V|)$$

# DFS - Depth First Search

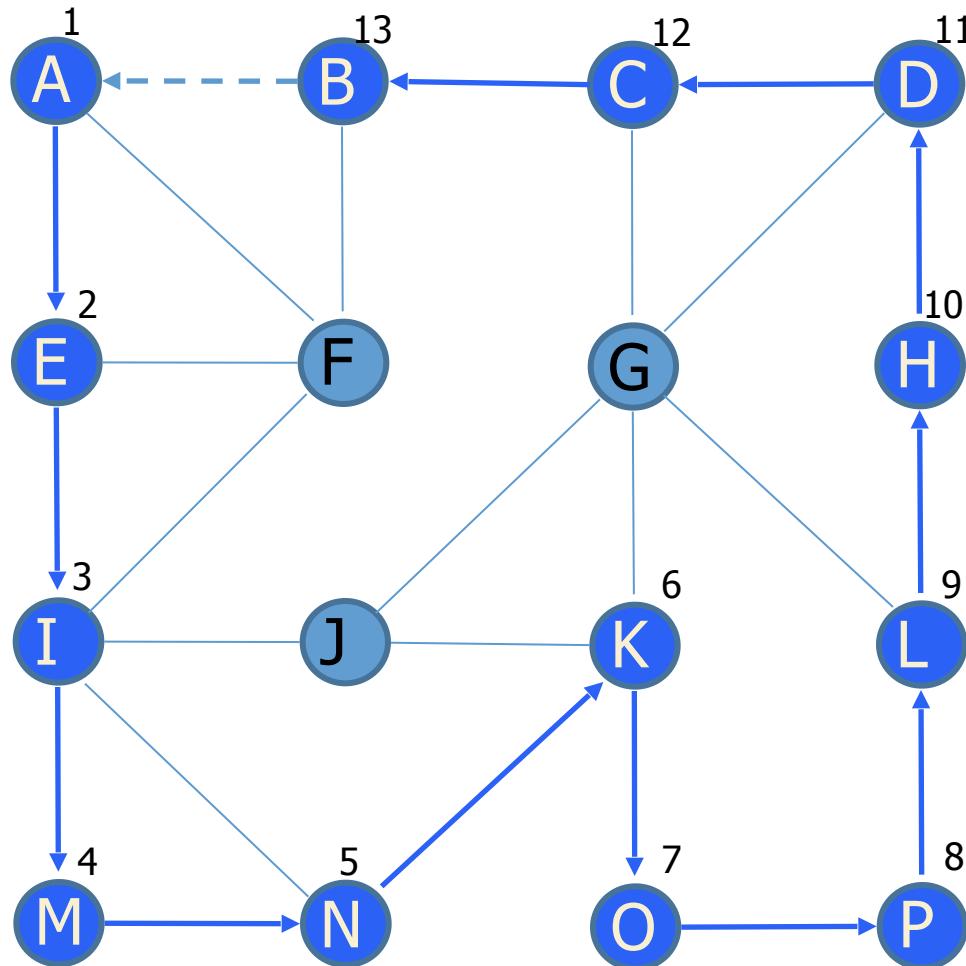
Generalization of a preorder traversal.

1. Start at an arbitrary vertex  $s$ .
2. Traverse (visit) as many descendants of  $s$  as possible without visiting a previously visited vertex.
3. Once a previously visited vertex is encountered, back out until a vertex  $v_i$  with unexplored edges is found.
4. Depth first search on  $v_i$ .
5. Repeat (steps 3-4) until all vertices visited.

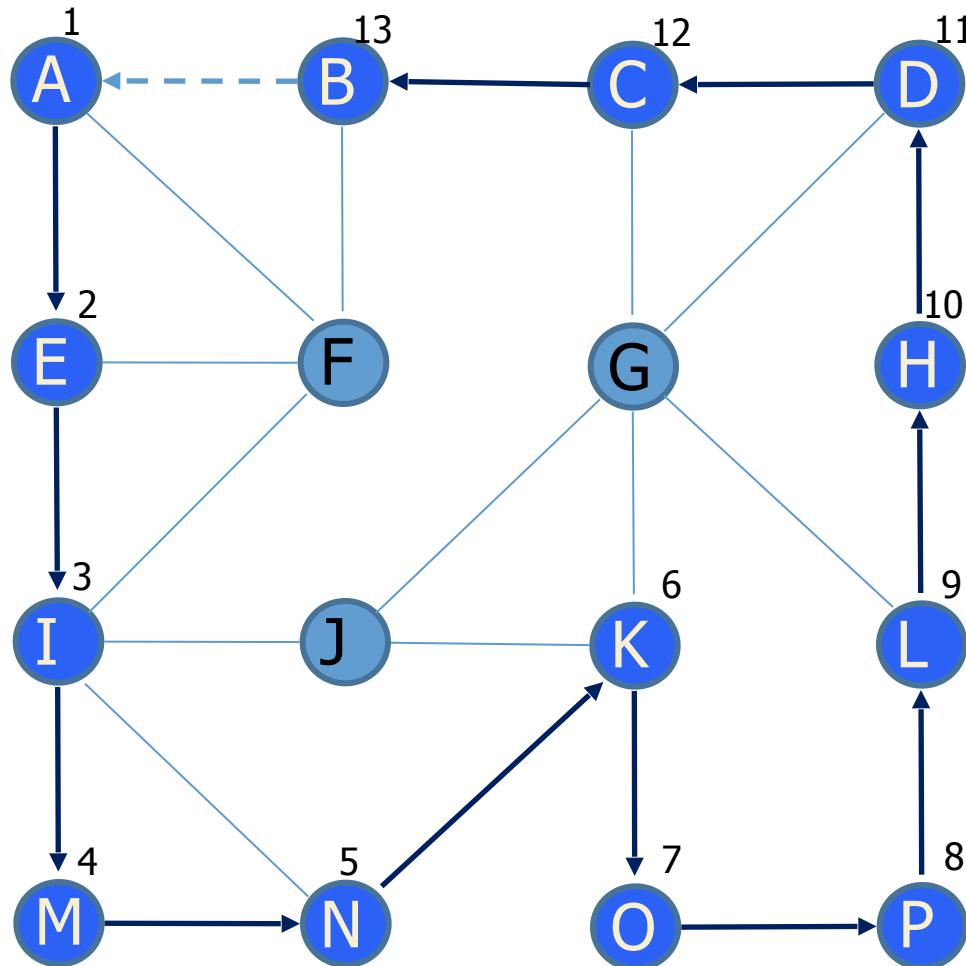
# DFS - Depth First Search



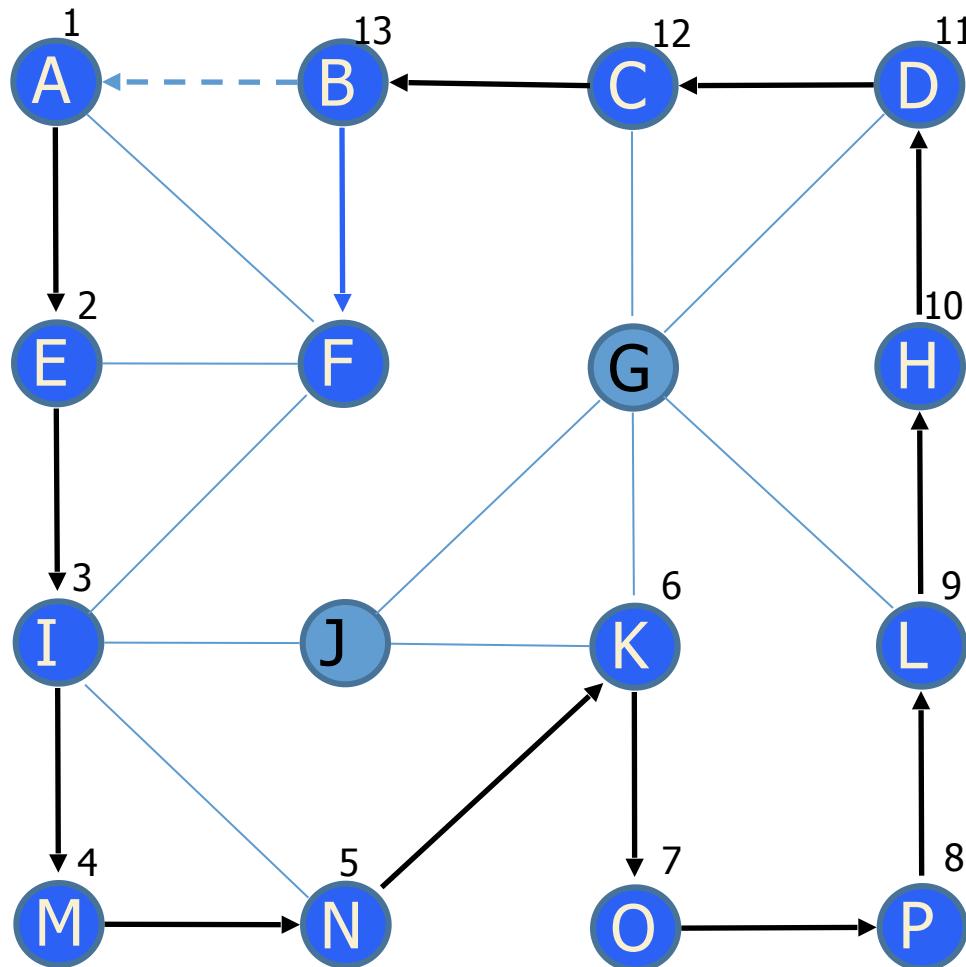
# DFS - Depth First Search



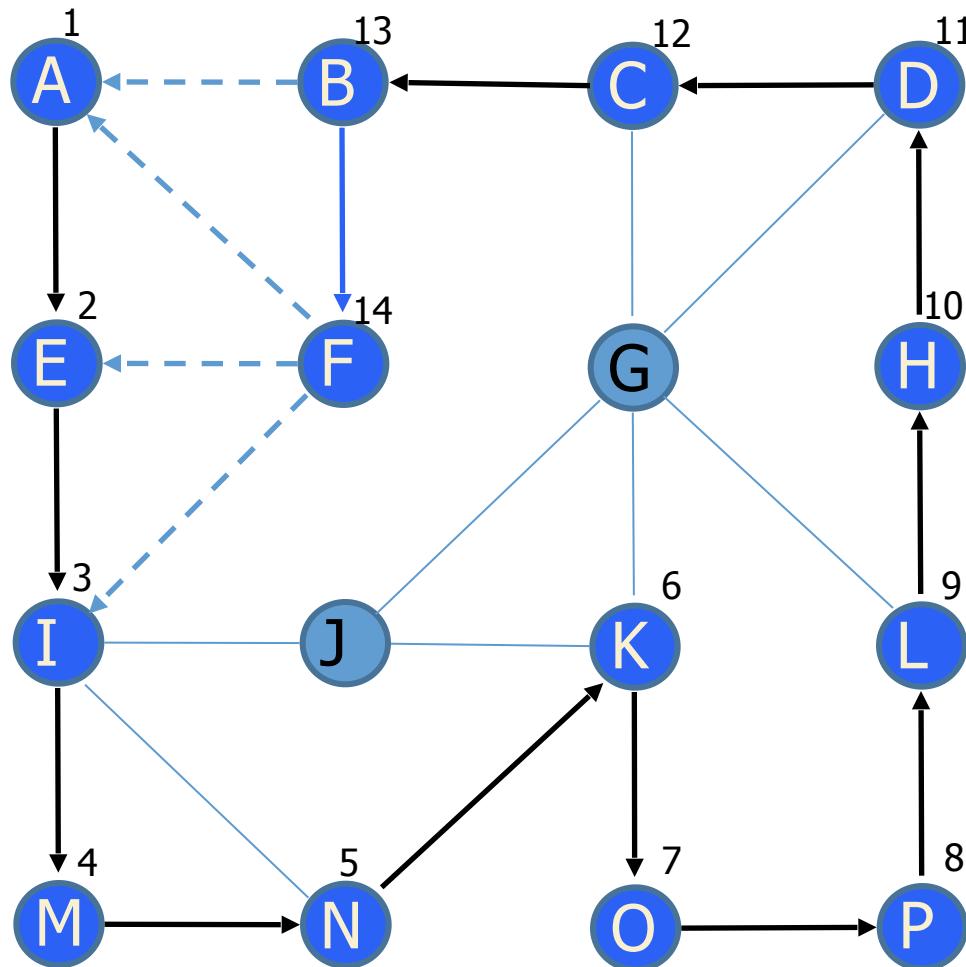
# DFS - Depth First Search



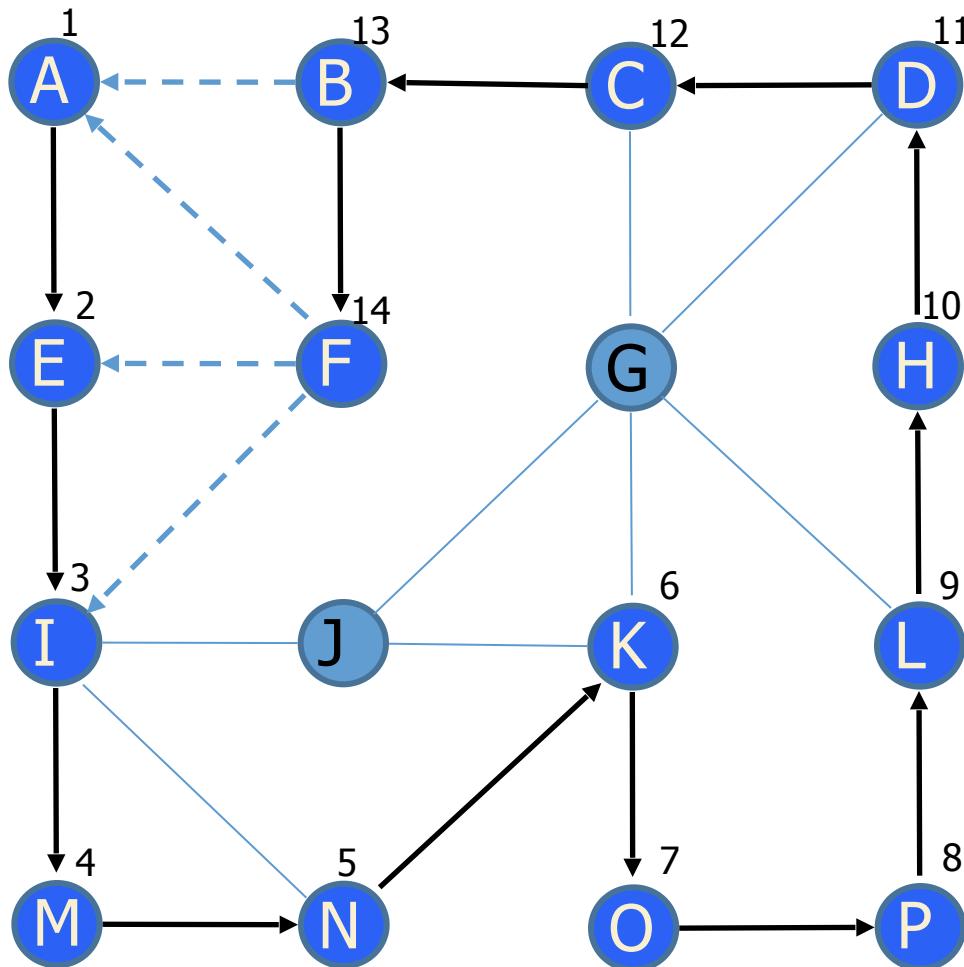
# DFS - Depth First Search



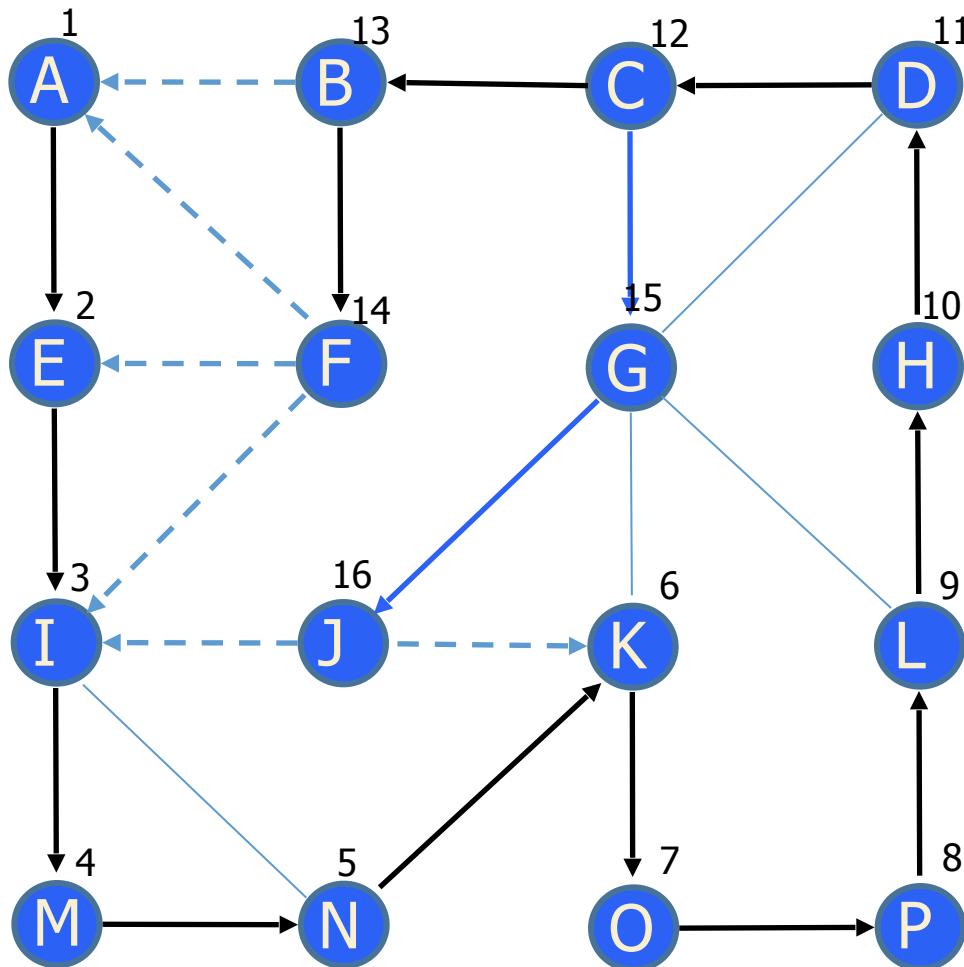
# DFS - Depth First Search



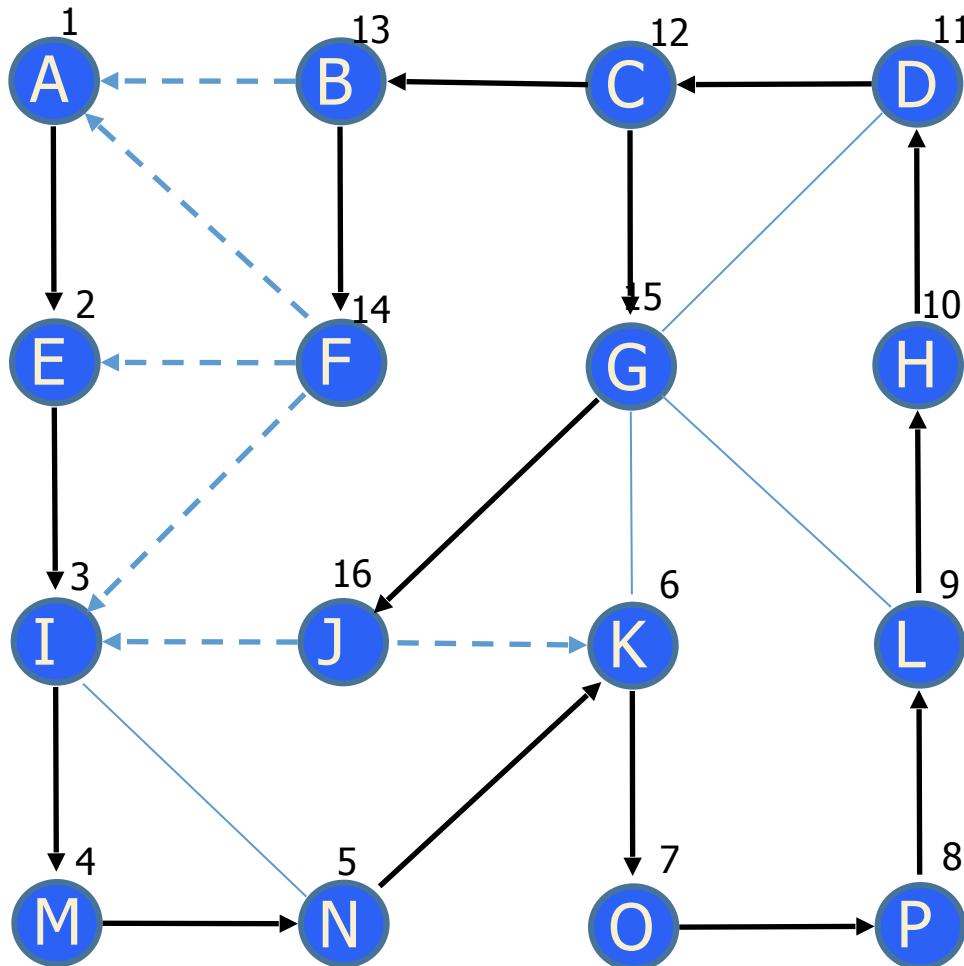
# DFS - Depth First Search



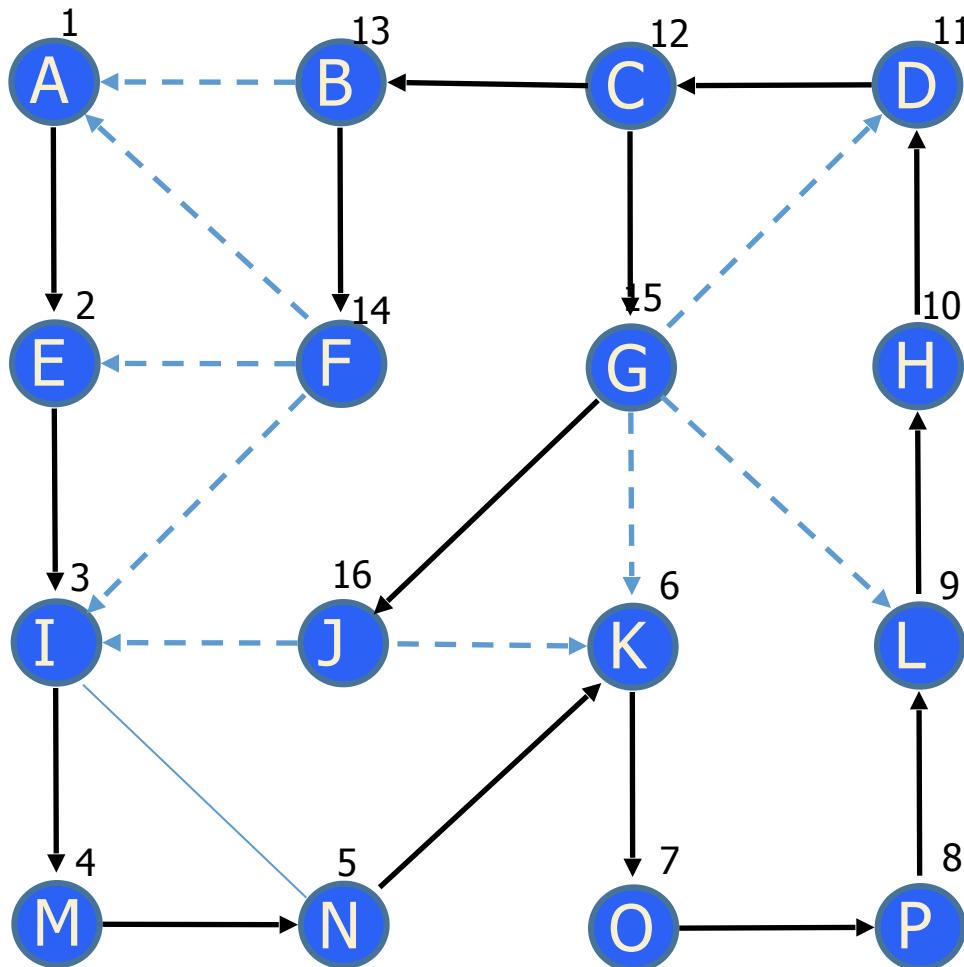
# DFS - Depth First Search



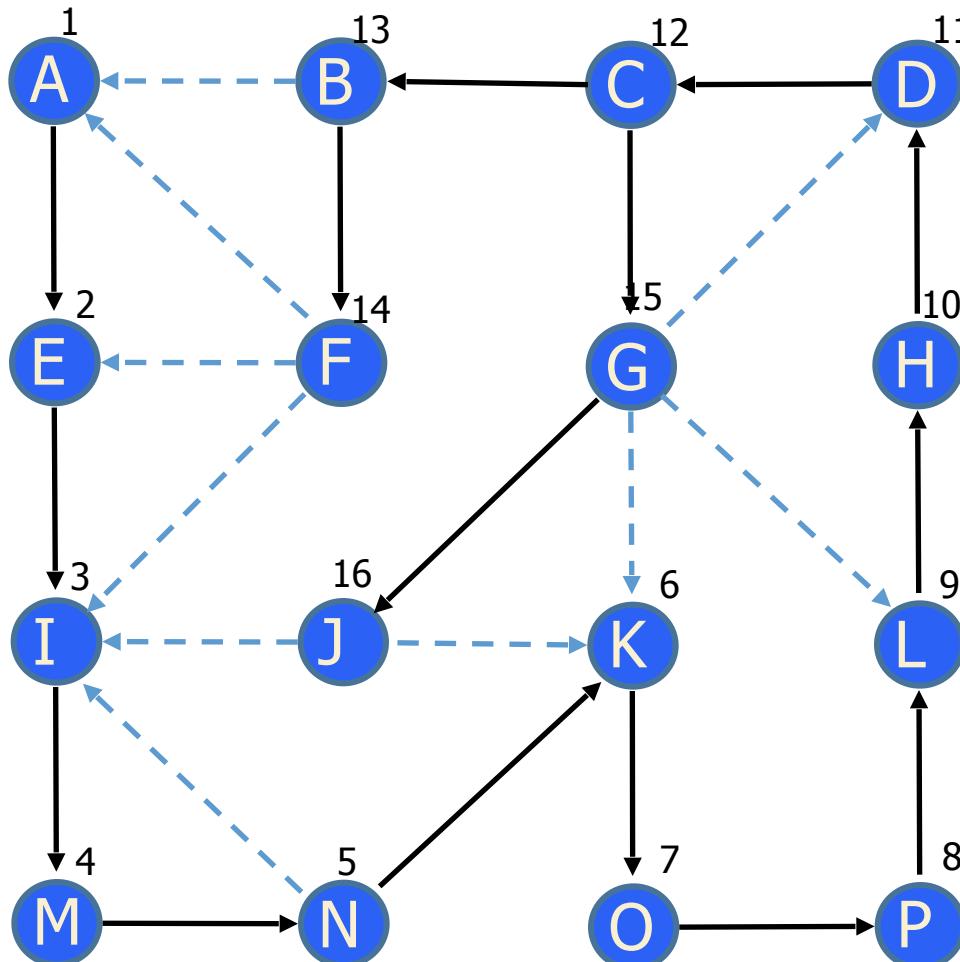
# DFS - Depth First Search



# DFS - Depth First Search

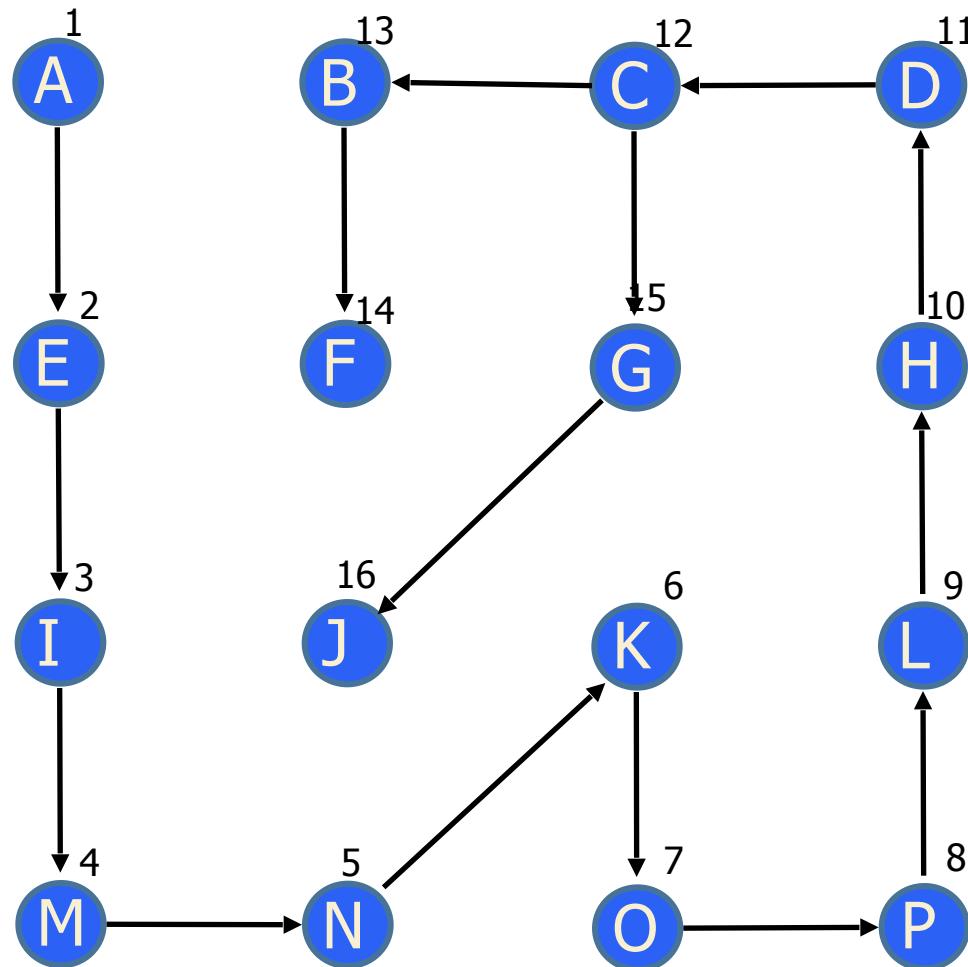


# DFS - Depth First Search

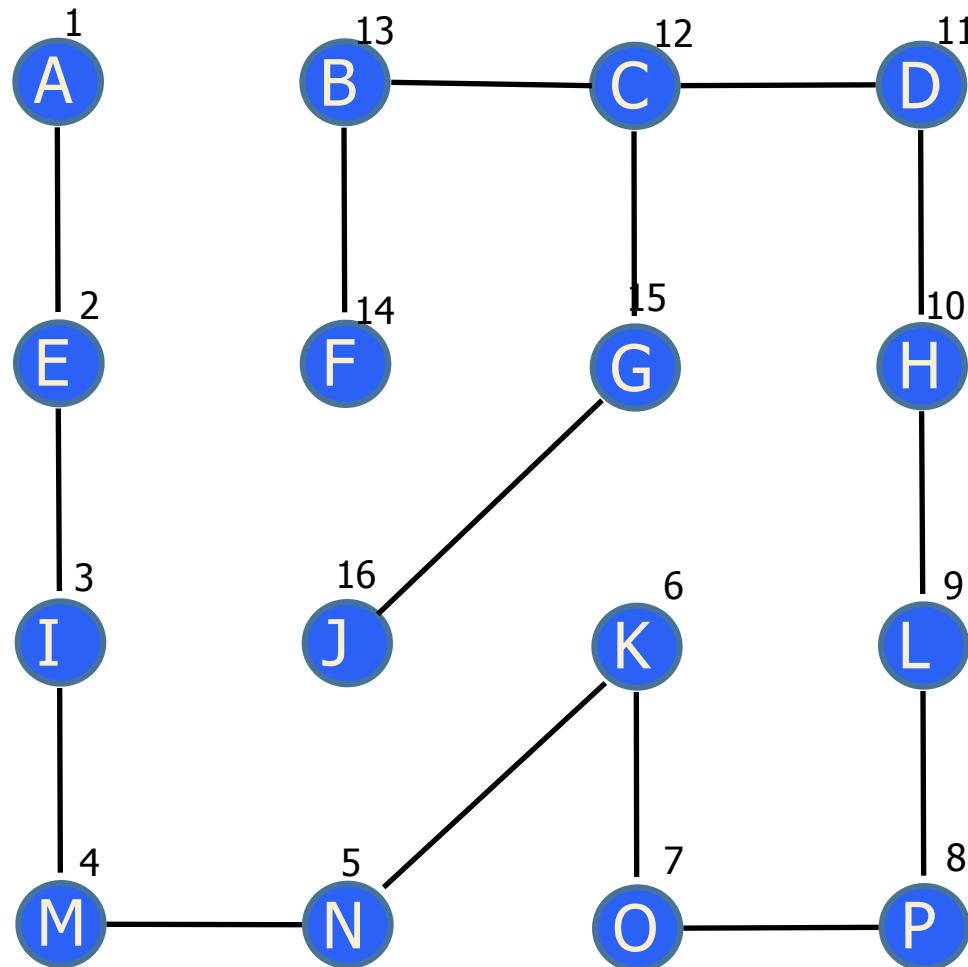


Running Time =  $O(|E| + |V|)$

# Spanning Tree



# Spanning Tree



# MST – Minimum Spanning Tree

- **Spanning Tree** – Given an undirected, connected graph  $G$ , a spanning tree  $T$  is a connected acyclic subgraph (tree) that contains all vertices of the graph.
- **Minimum Spanning Tree** – the spanning tree of the smallest weight, where the weight is the sum of the weights on all  $T$ 's edges.
- **MST Problem** – Find the minimum spanning tree for a given weighted connected graph.
- Not always a unique MST.

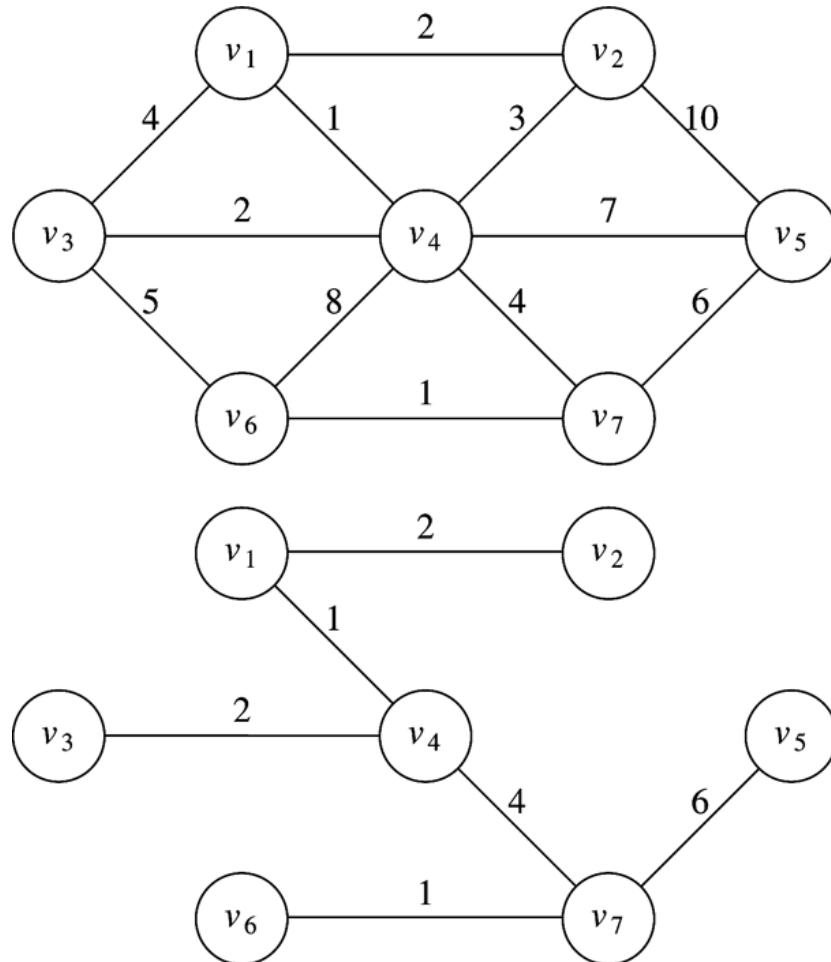
# MST

- Useful for minimizing the amount of wiring needed for
  - Phone lines
  - Cable lines
- Used in wireless sensor networks
- Used in network routing algorithms

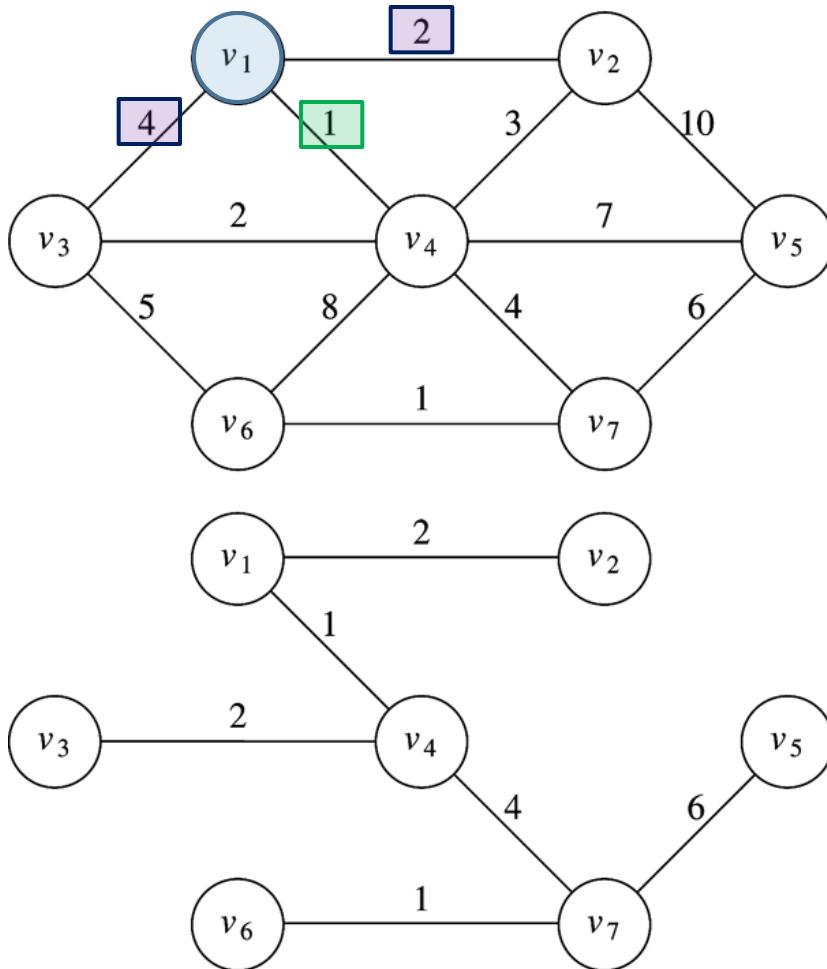


Copyright notice: Google 2005, The GeoInformation Group 2006

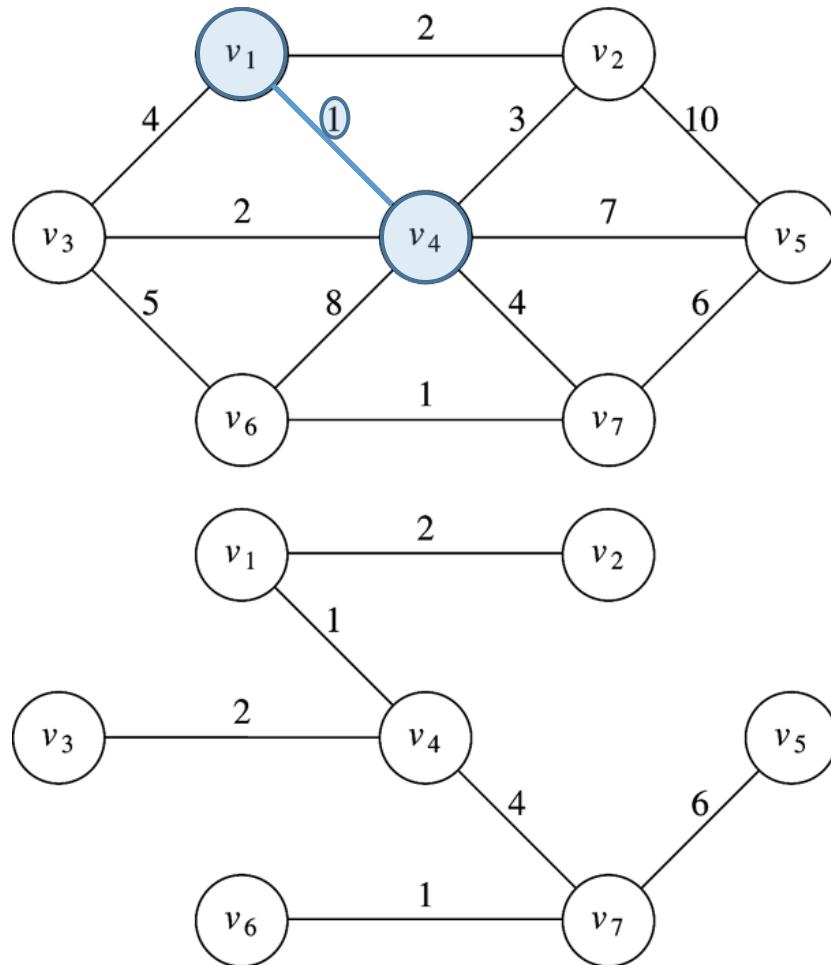
# Prim's Algorithm



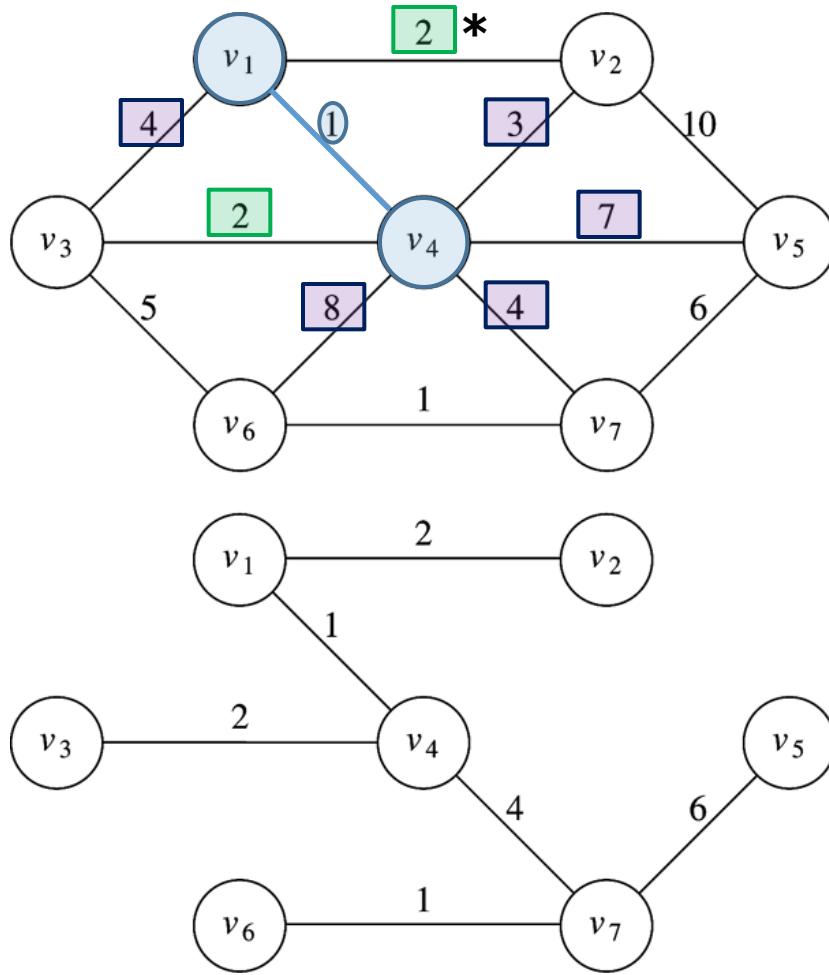
# Prim's Algorithm



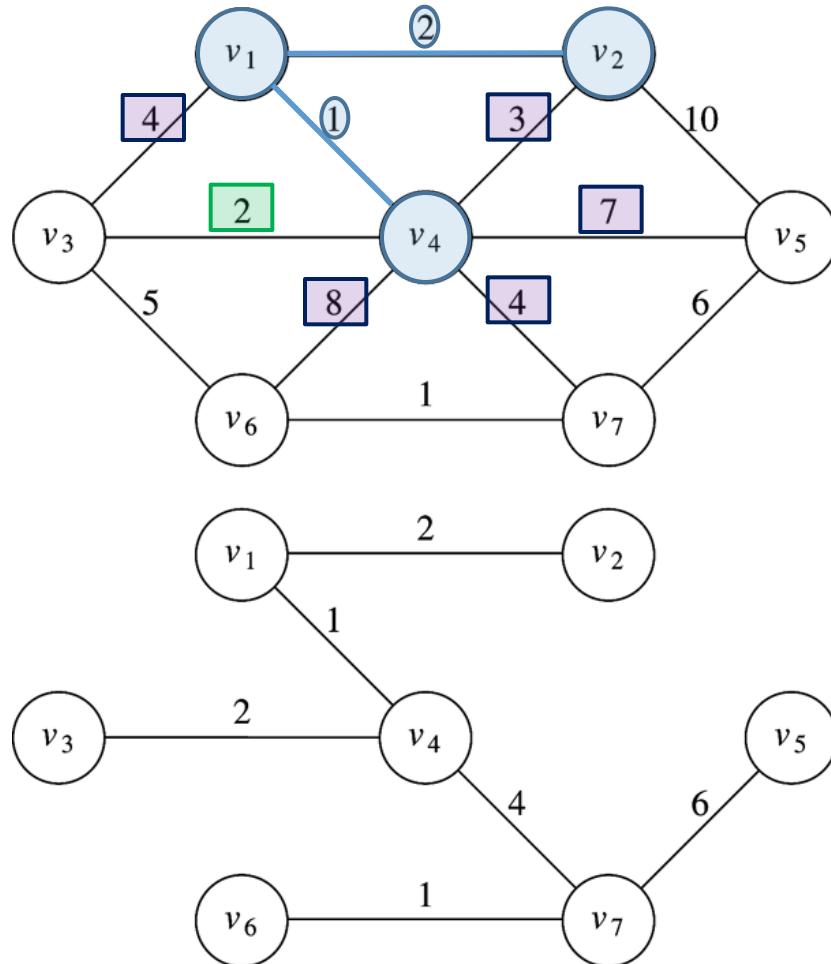
# Prim's Algorithm



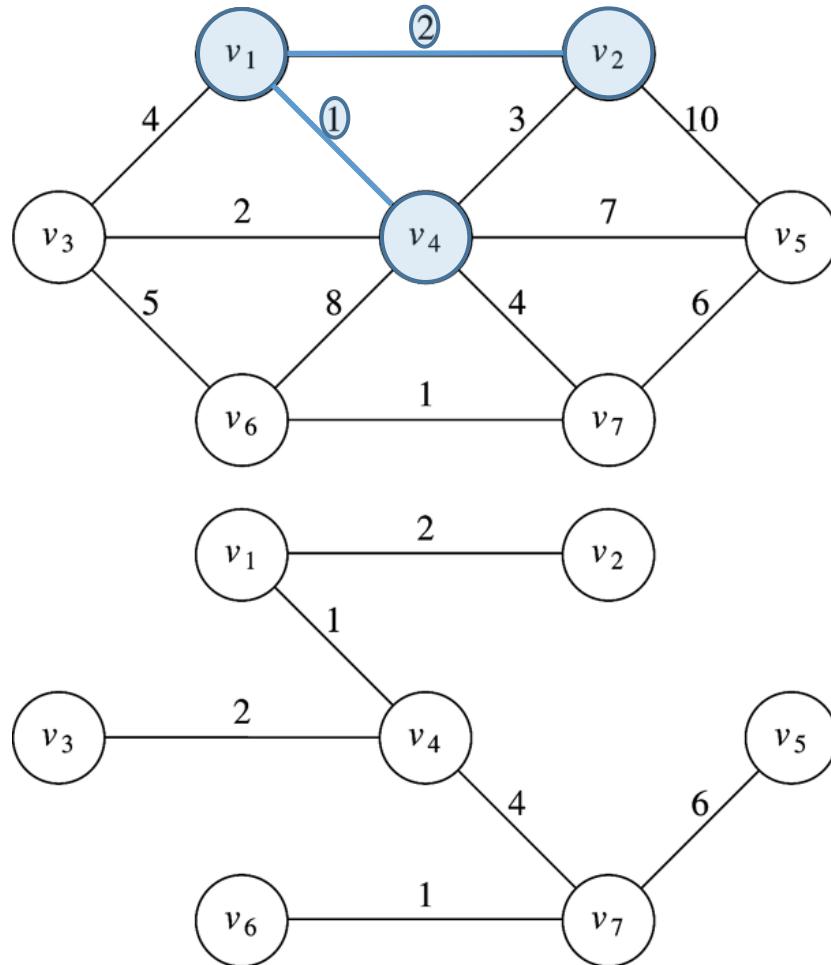
# Prim's Algorithm



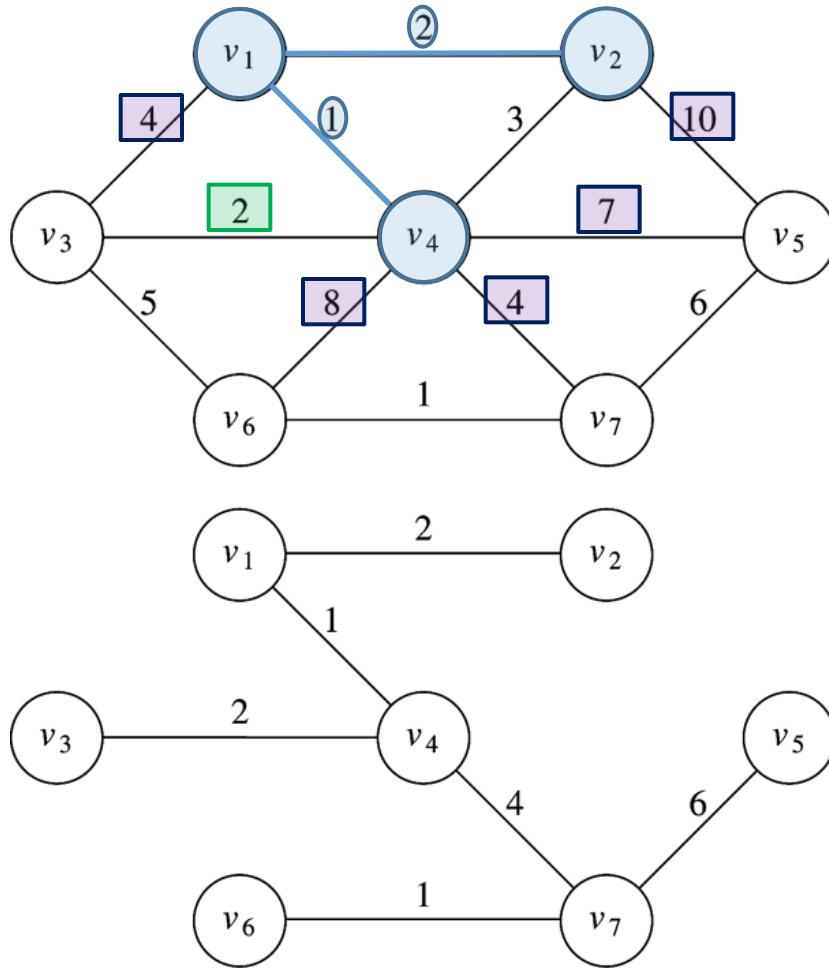
# Prim's Algorithm



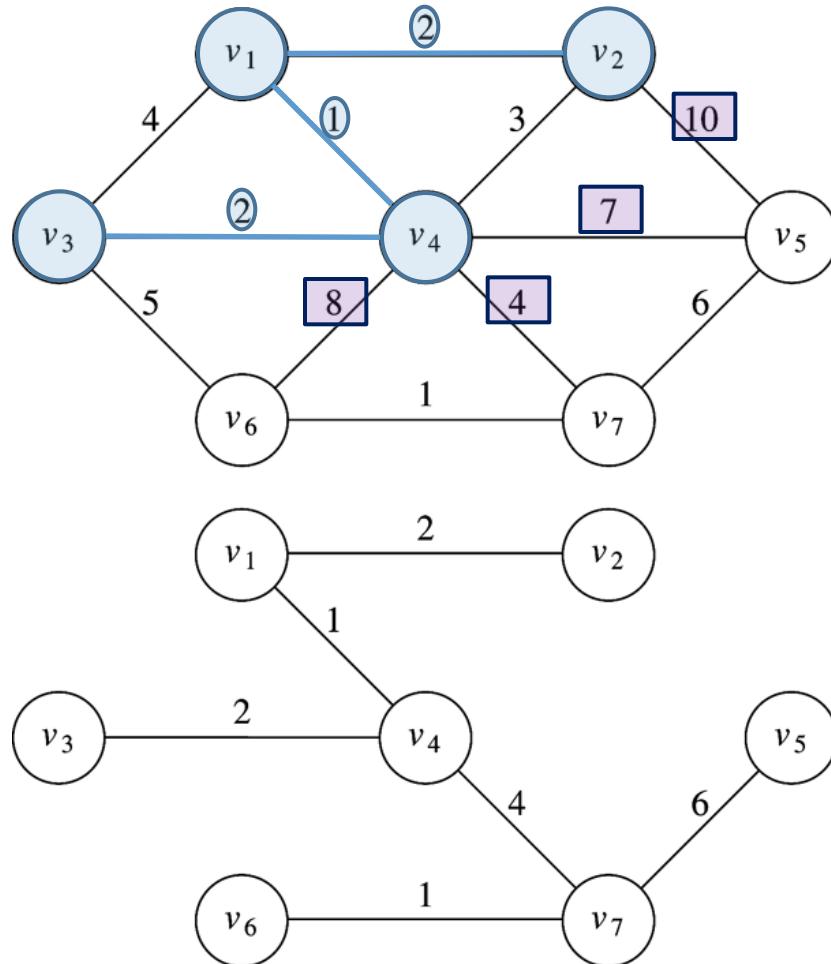
# Prim's Algorithm



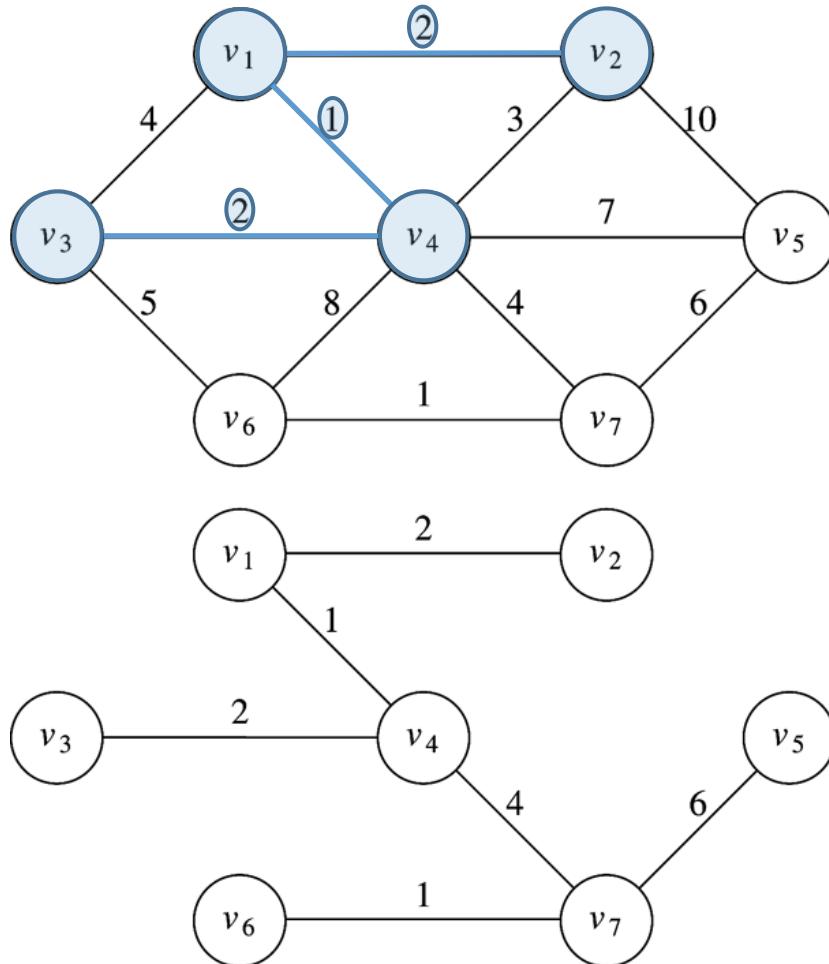
# Prim's Algorithm



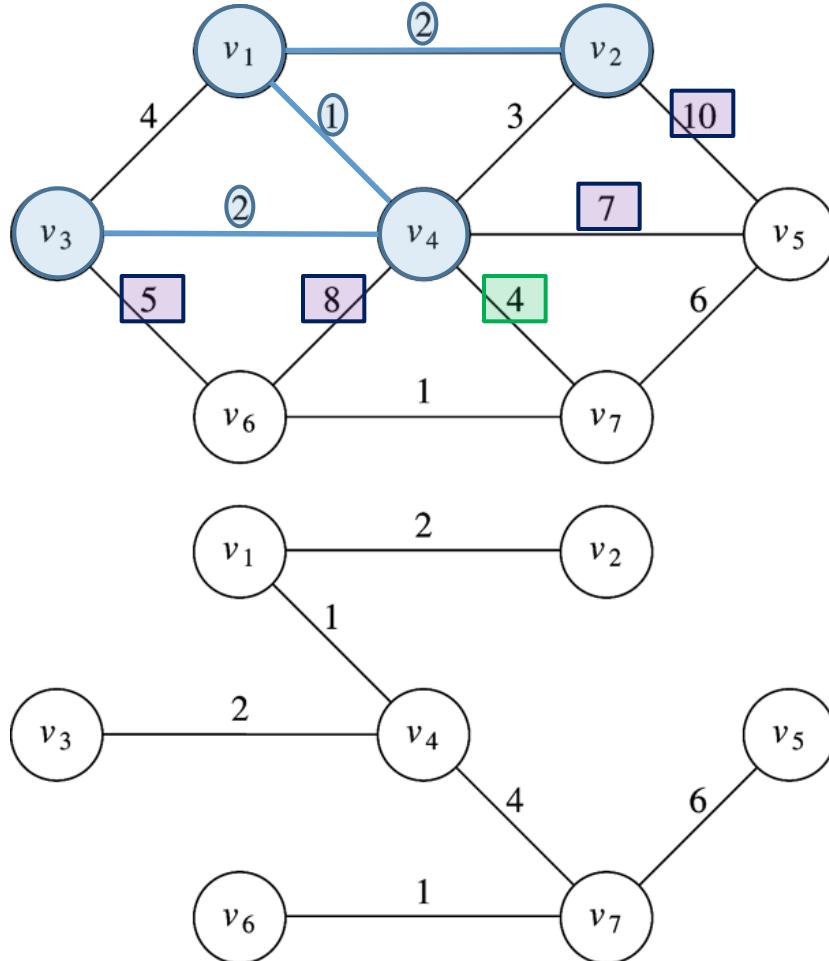
# Prim's Algorithm



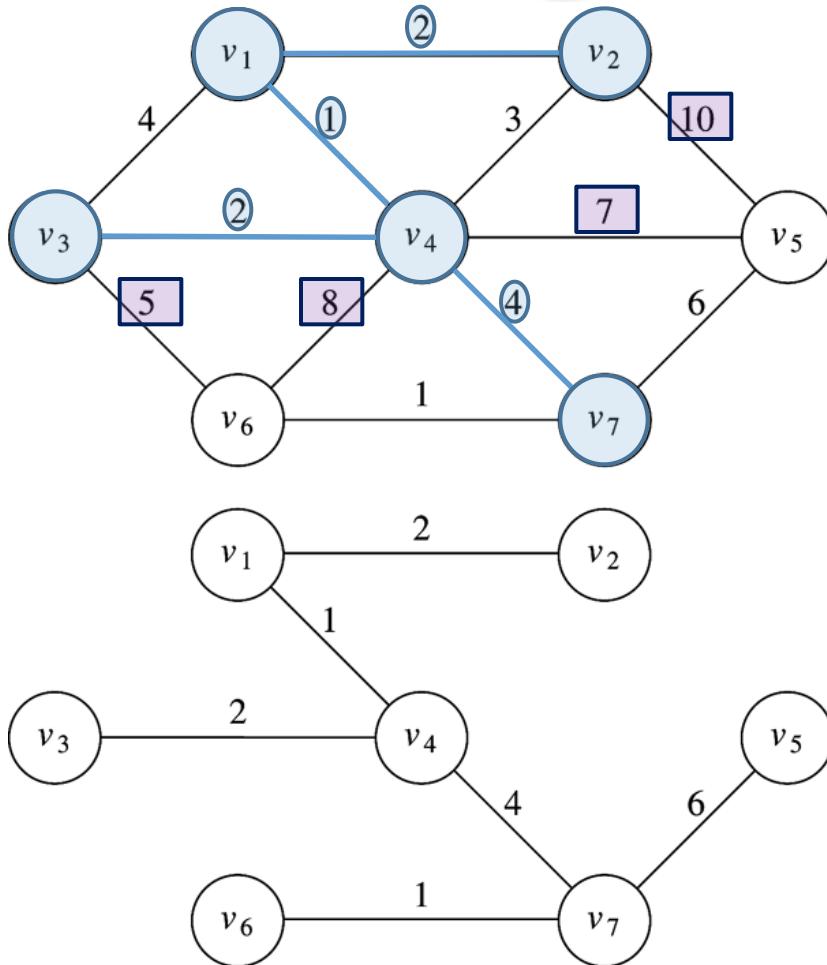
# Prim's Algorithm



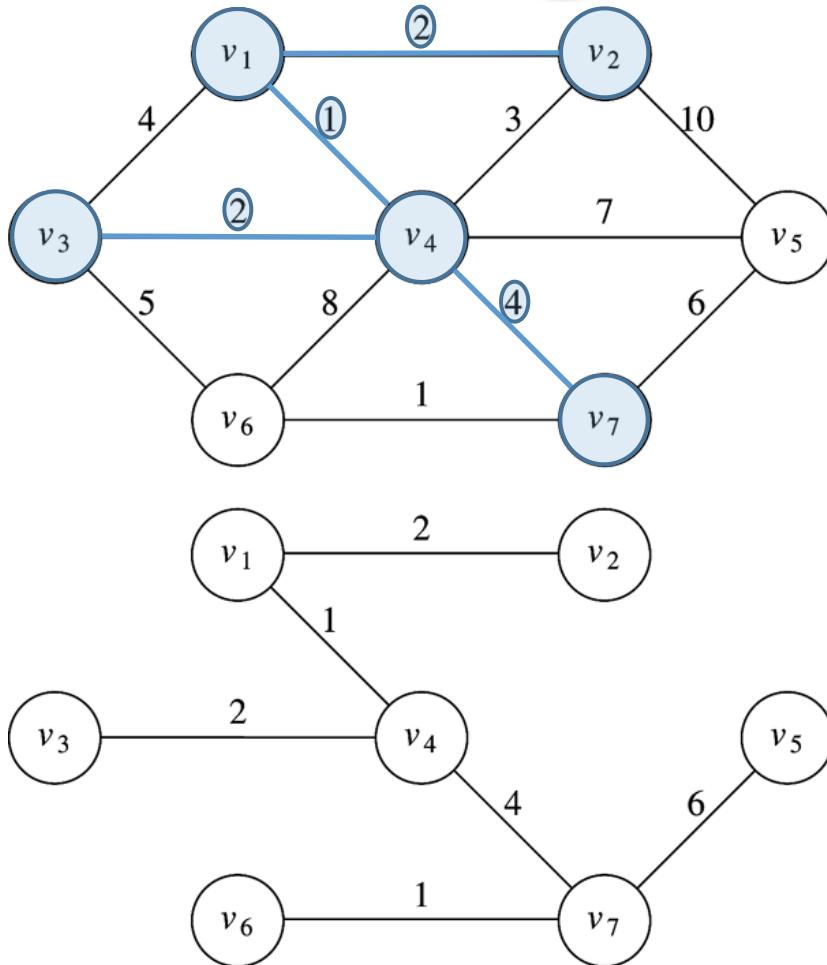
# Prim's Algorithm



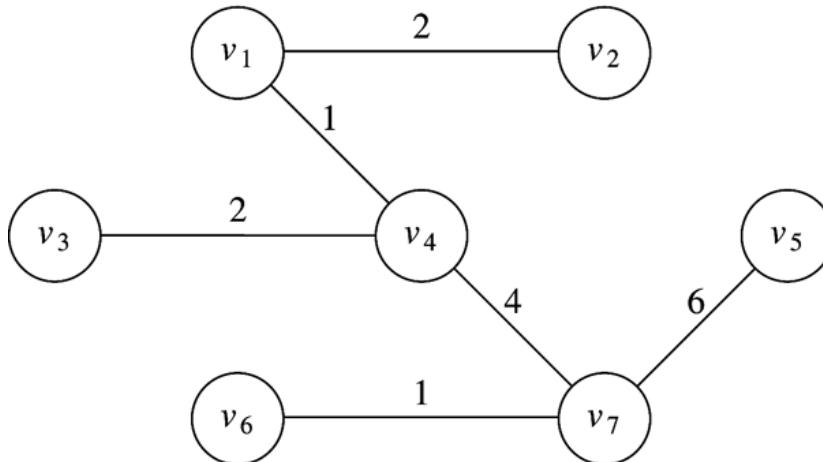
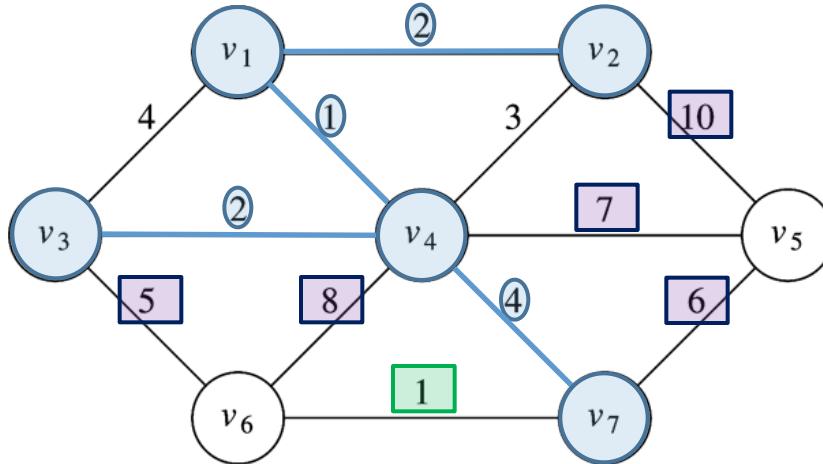
# Prim's Algorithm



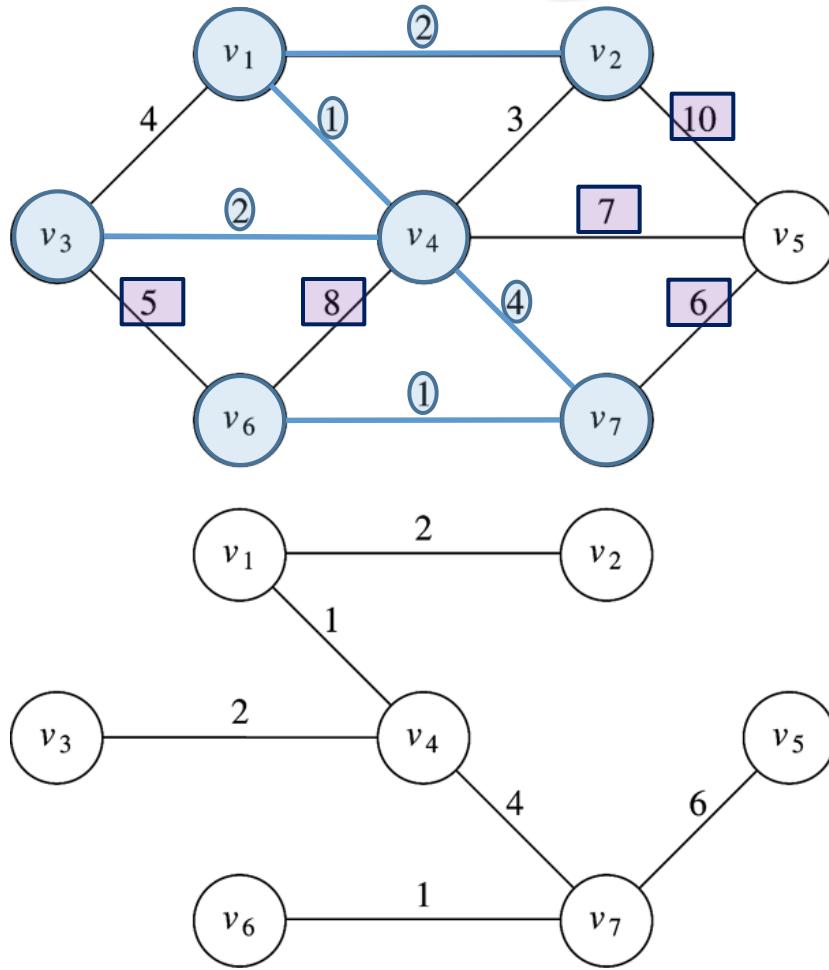
# Prim's Algorithm



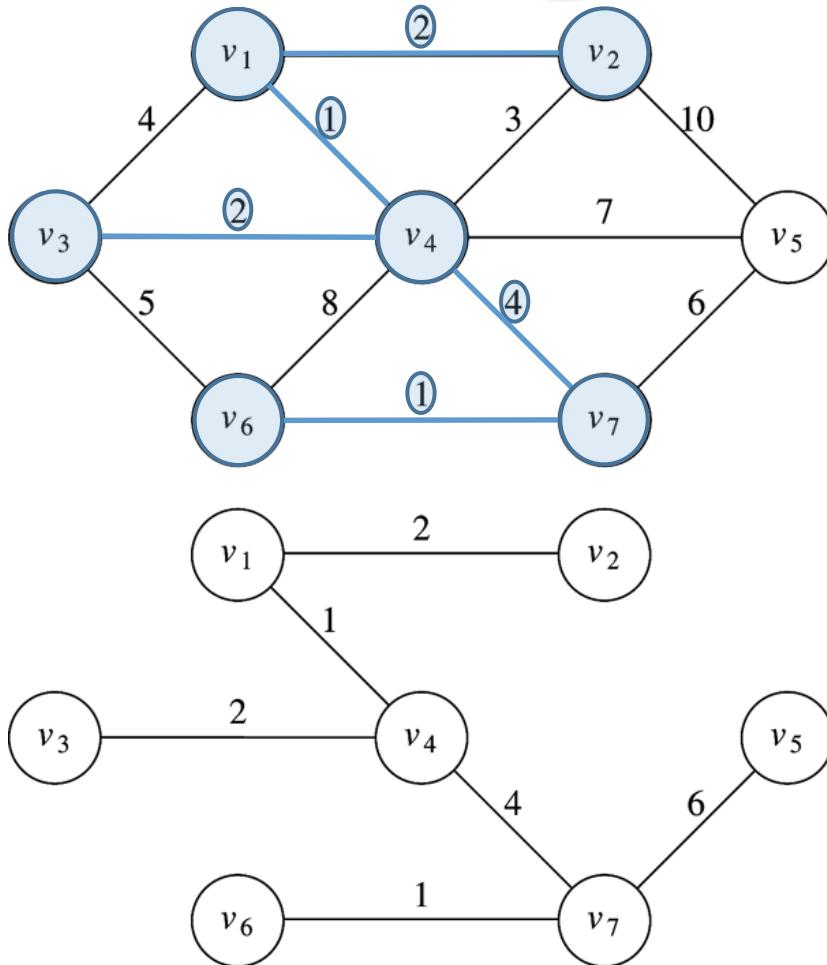
# Prim's Algorithm



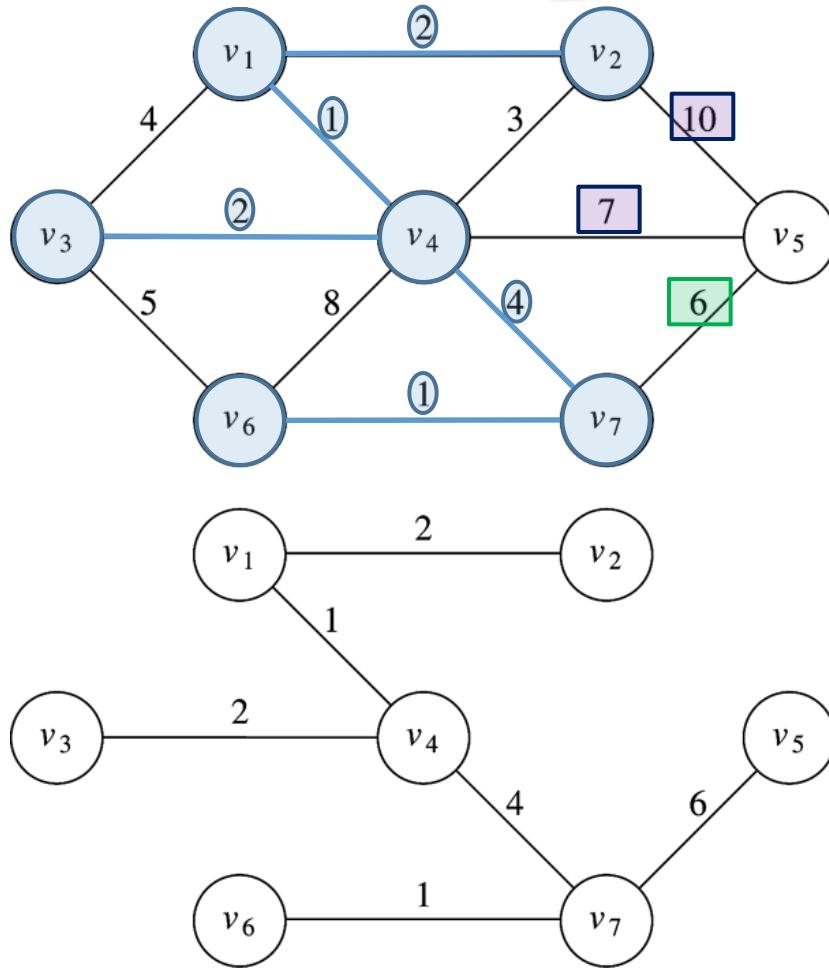
# Prim's Algorithm



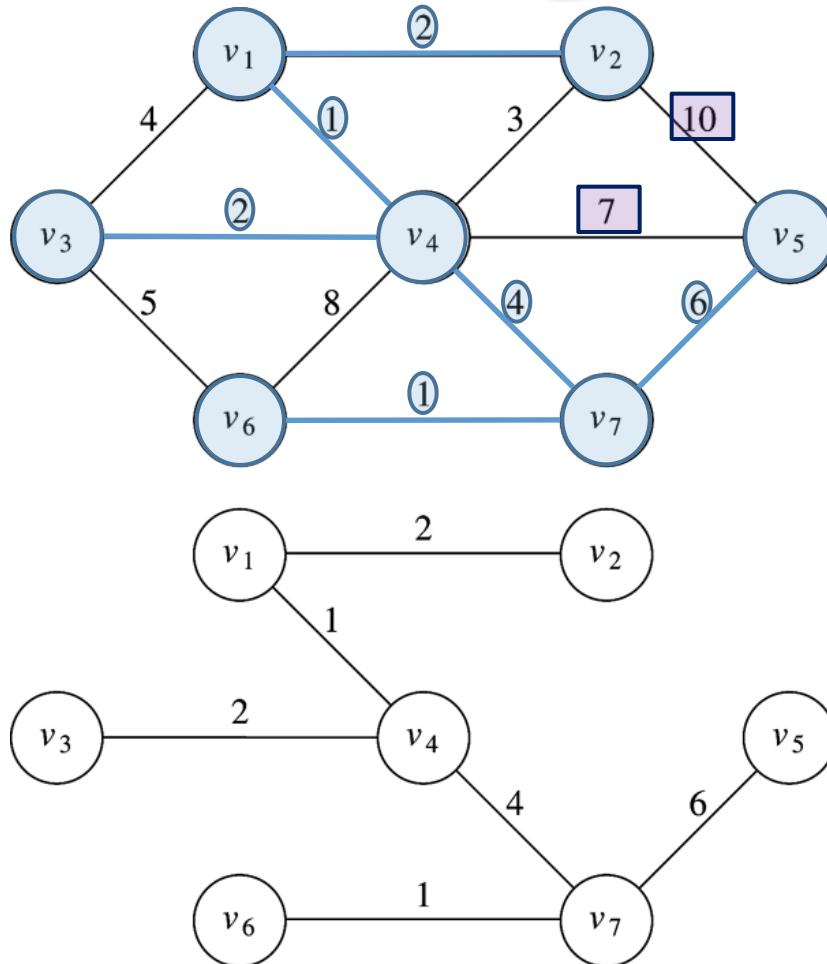
# Prim's Algorithm



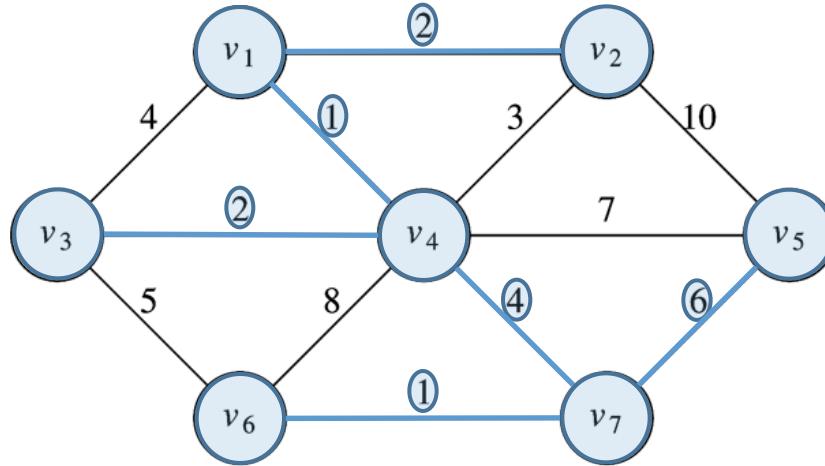
# Prim's Algorithm



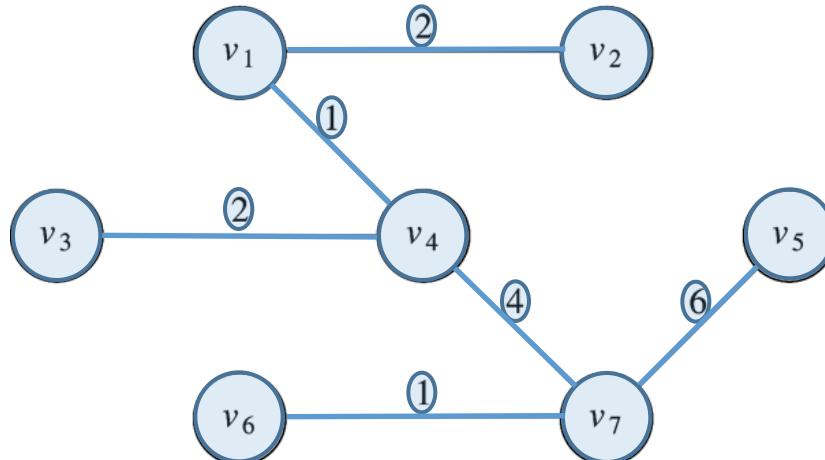
# Prim's Algorithm



# Prim's Algorithm



- $O(|V|^2)$   
Linear scan to  
find min
- $O(|E|\log|V|)$   
Binary heaps



# Directed Graph - Weighted

