EECS 114:
Engineering Data Structures and Algorithms
Lecture 10

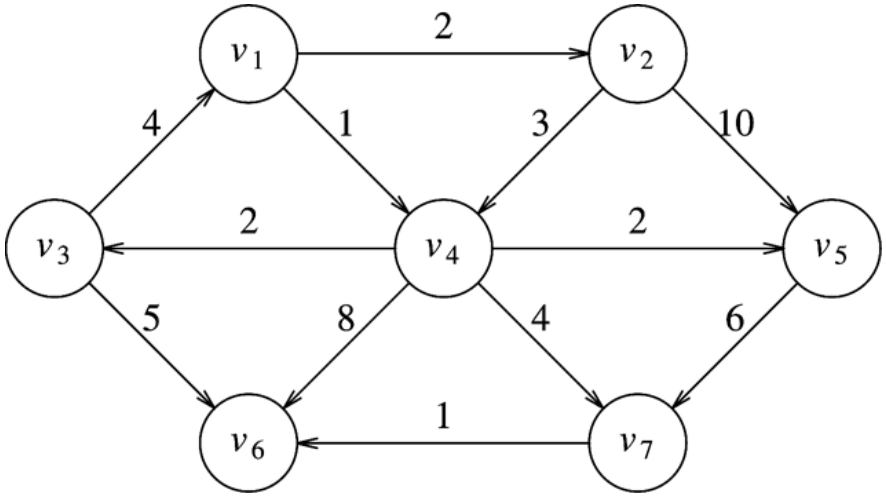
Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

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

Graphs



Graph

- 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, …, w_N such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$
  - A **simple path** is a path where all vertices are distinct except possibly first/last
  - **path length:** $N-1$ for N vertices
- Edges can have an associated cost called the **weight**.

Graphs - Definitions

- 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 – $|V|^2$ matrix, with **1** if there is an edge between two vertices, **0** otherwise.
  - Good for dense graphs.
  - Wasted space if not dense.
  - If graph is sparse, adjacency list is better.

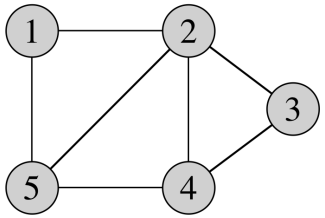
$$|E| = \Theta(|V|^2)$$

- Adjacency list – for each vertex, store a list of vertices that share an edge
  - Example…

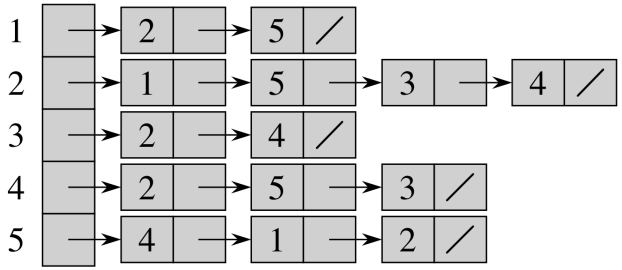
Adjacency list

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

Undirected Graph



(a)

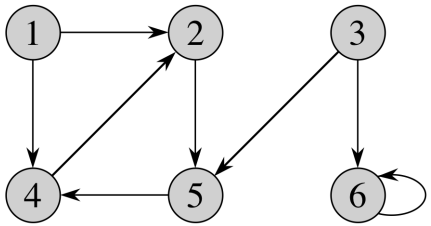


(b)

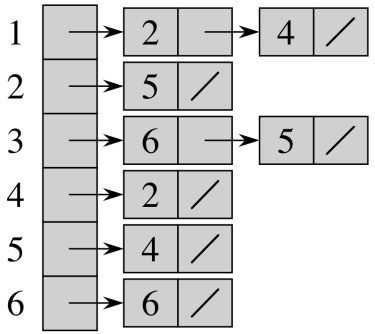
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Directed Graph



(a)

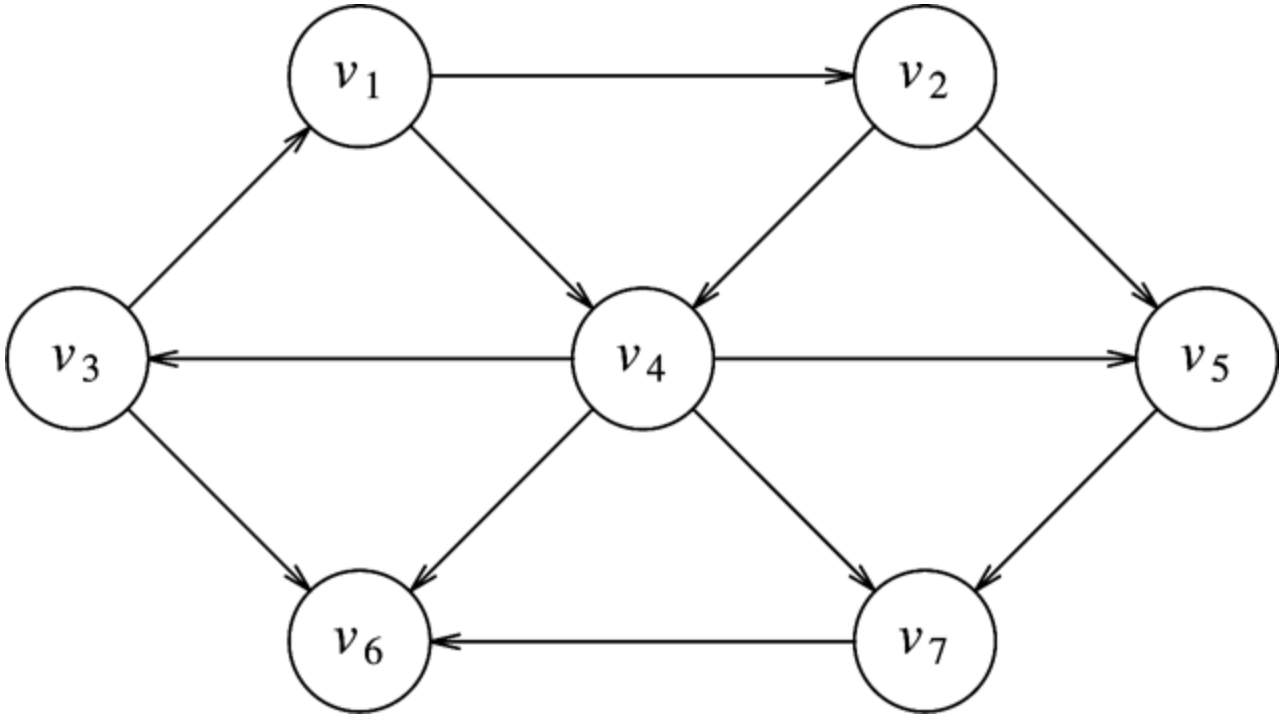


(b)

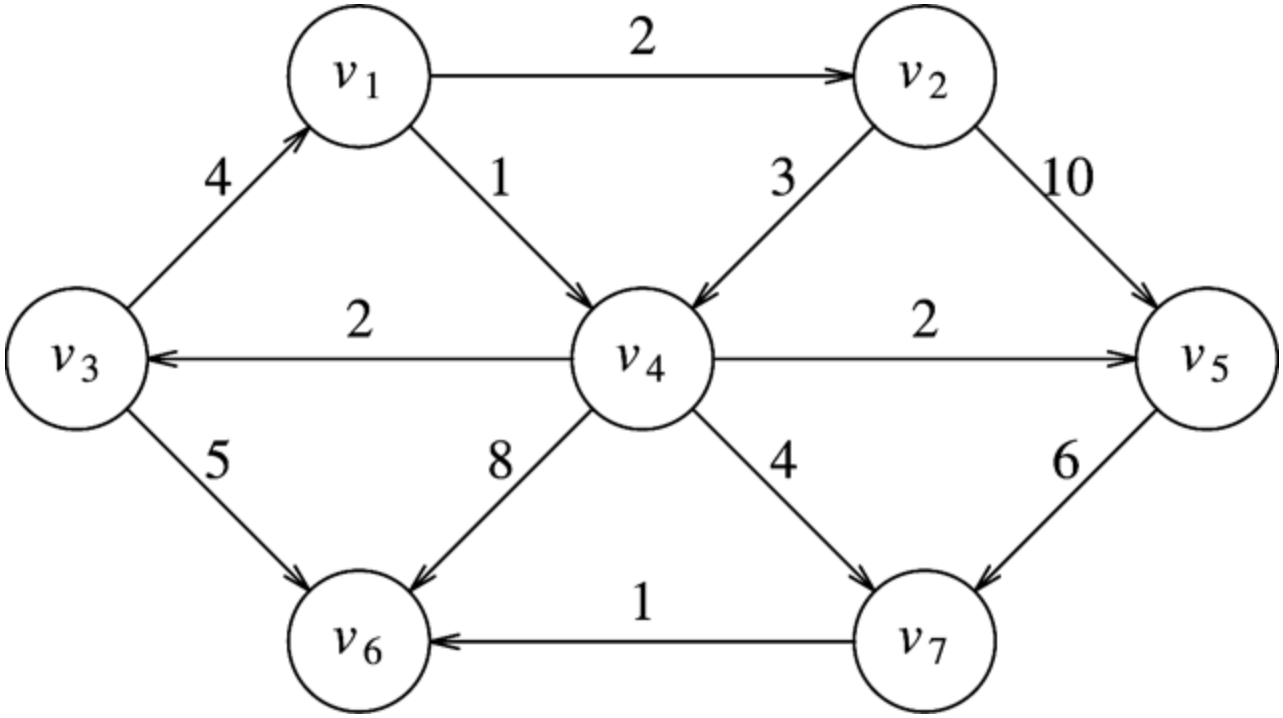
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

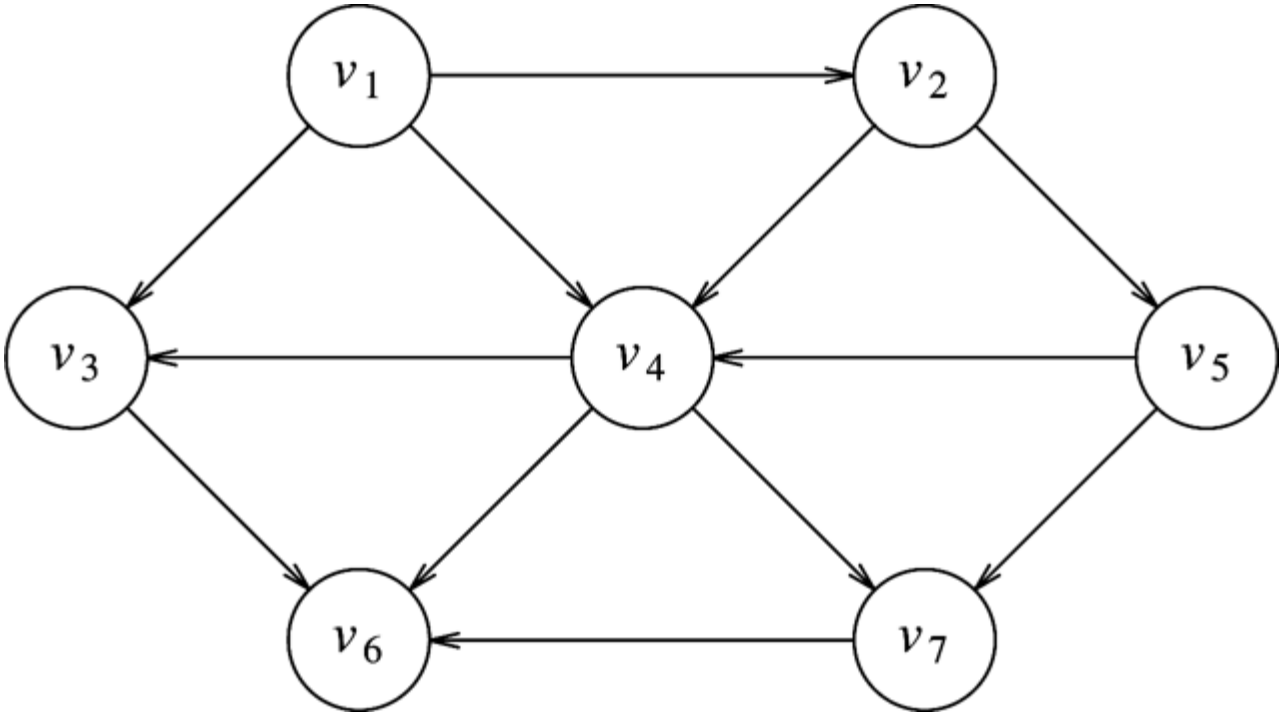
Directed Graph – Unweighted



Directed Graph - Weighted



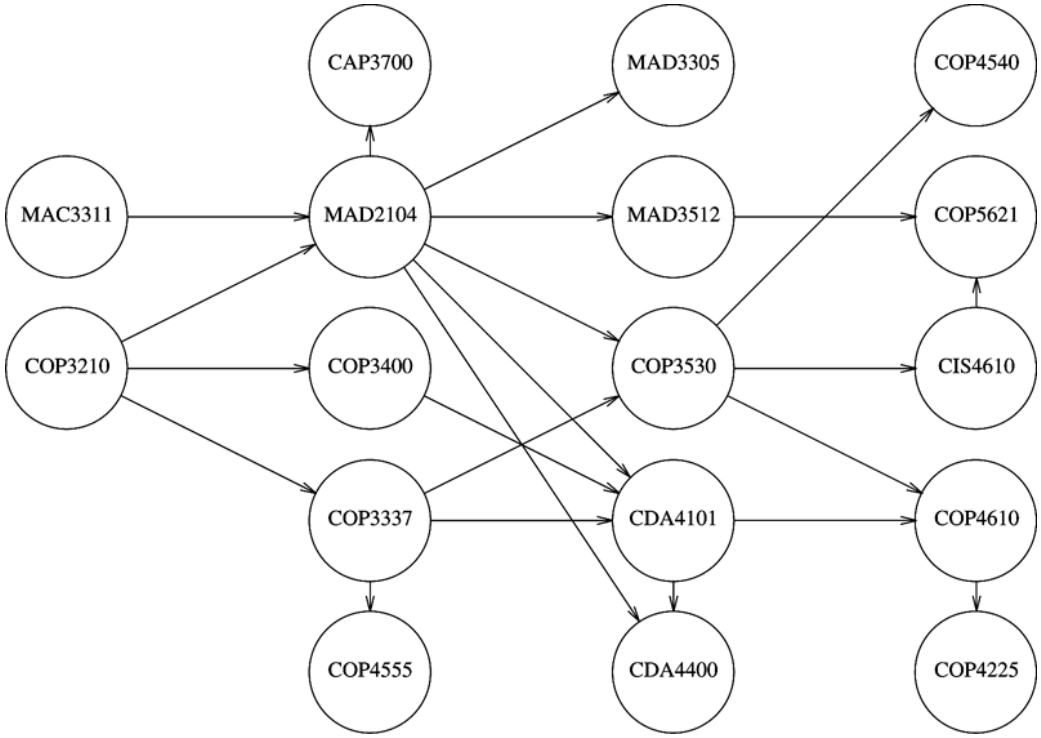
Directed Acyclic Graph



Topological Sort

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

Scheduling

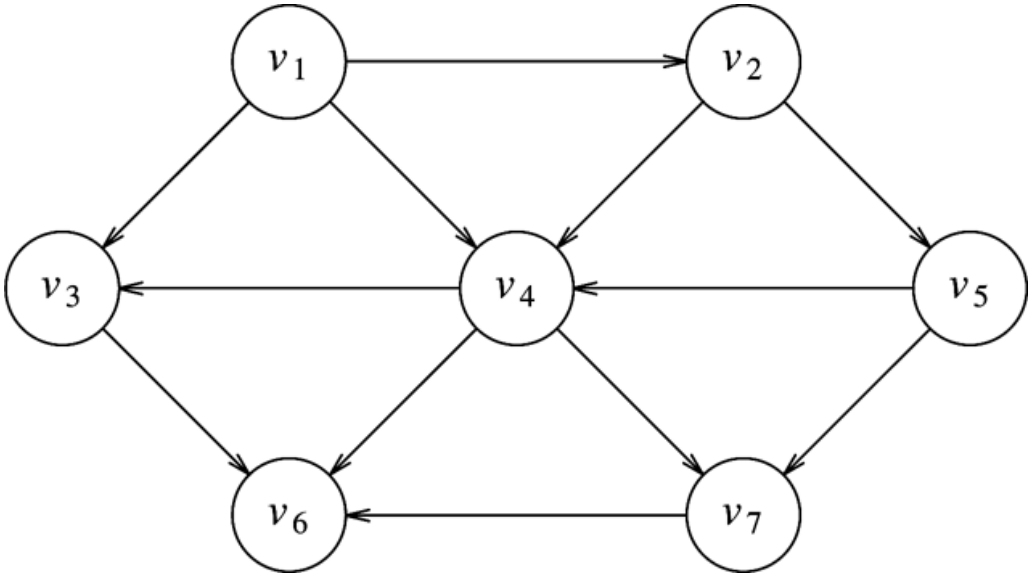

CAP3700
MAD3305
COP4540
MAC3311
MAD2104
MAD3512
COP5621
COP3210
COP3400
COP3530
CIS4610
COP3337
CDA4101
COP4610
COP4555
CDA4400
COP4225

Topological Sort

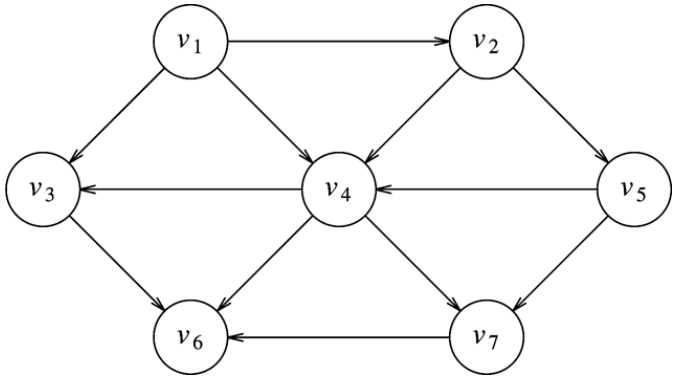
- **Simple algorithm:**
  1. Find an initial vertex v with no incoming edges
  2. Print v
  3. Remove v from graph
  4. Remove $v's$ edges from graph, updating effected vertices
  5. Iterate(steps 1-4) over the remaining graph

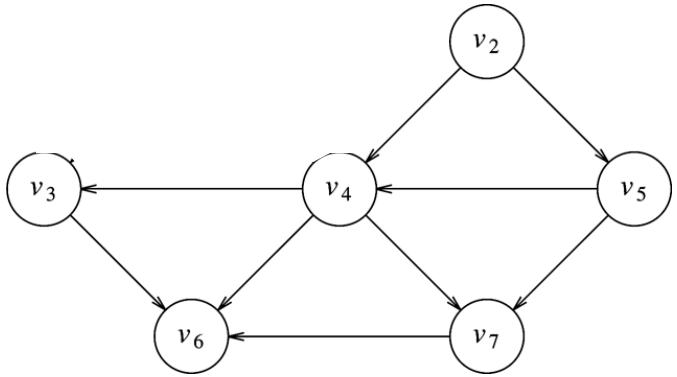
  Running Time : $O(|V|^2)$
  - Finding a vertex with in-degree zero, linear scan of V
  - There are $|V|$ calls to do this
- Speed up:
  - Use a Queue to store vertices with in-degree zero
  - Only have to remove the front of Queue, skip linear scan of V
  - **Running Time:** $O(|E| + |V|)$

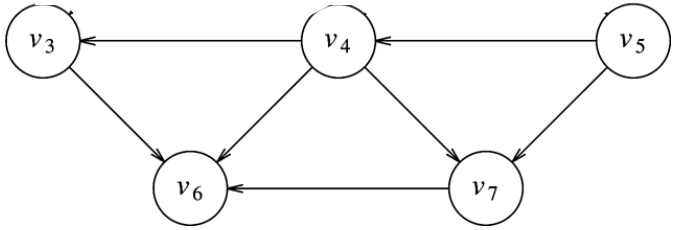
Topological Sort

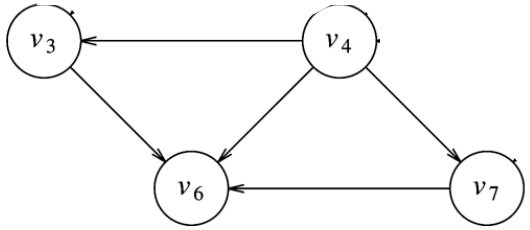


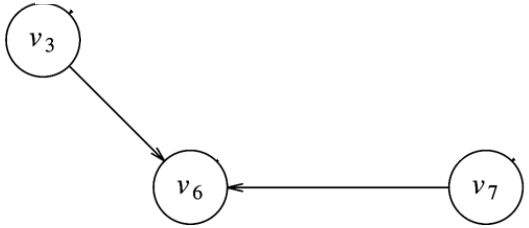
v1, v2, v5, v4, v3, v7, v6

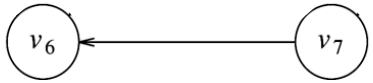










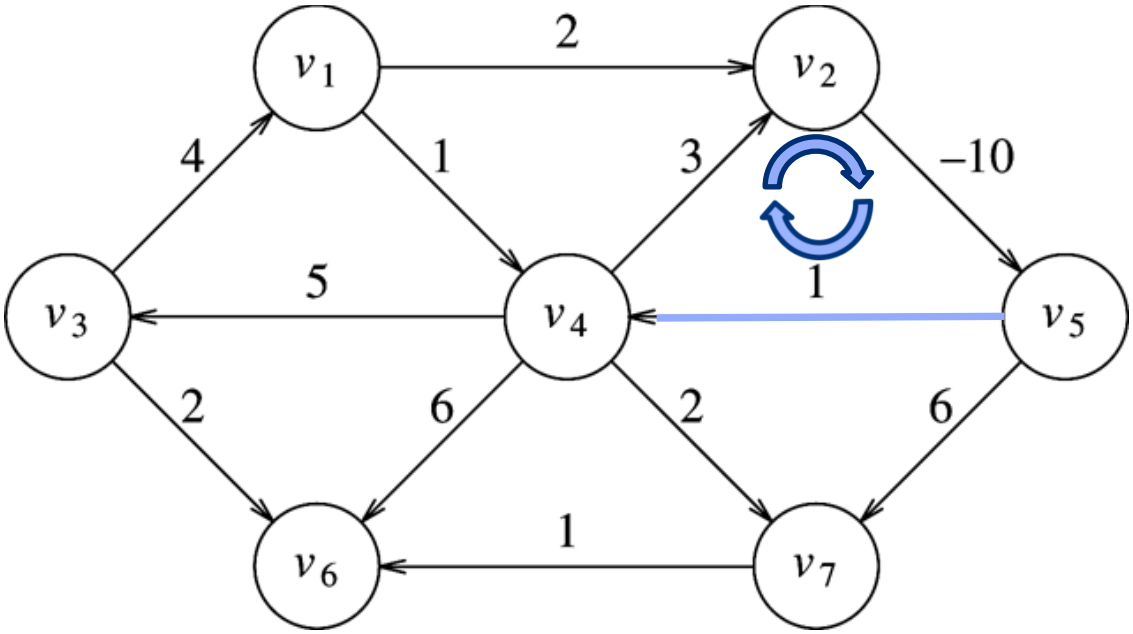


v1, v2, v5, v4, v3, v7, v6

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
- Referred to as *single-pair shortest path* problem
- Used in network routing, flight scheduling, everyday life.
- **B**readth **F**irst **S**earch – Find shortest path in unweighted graph.

Negative Cost Cycle

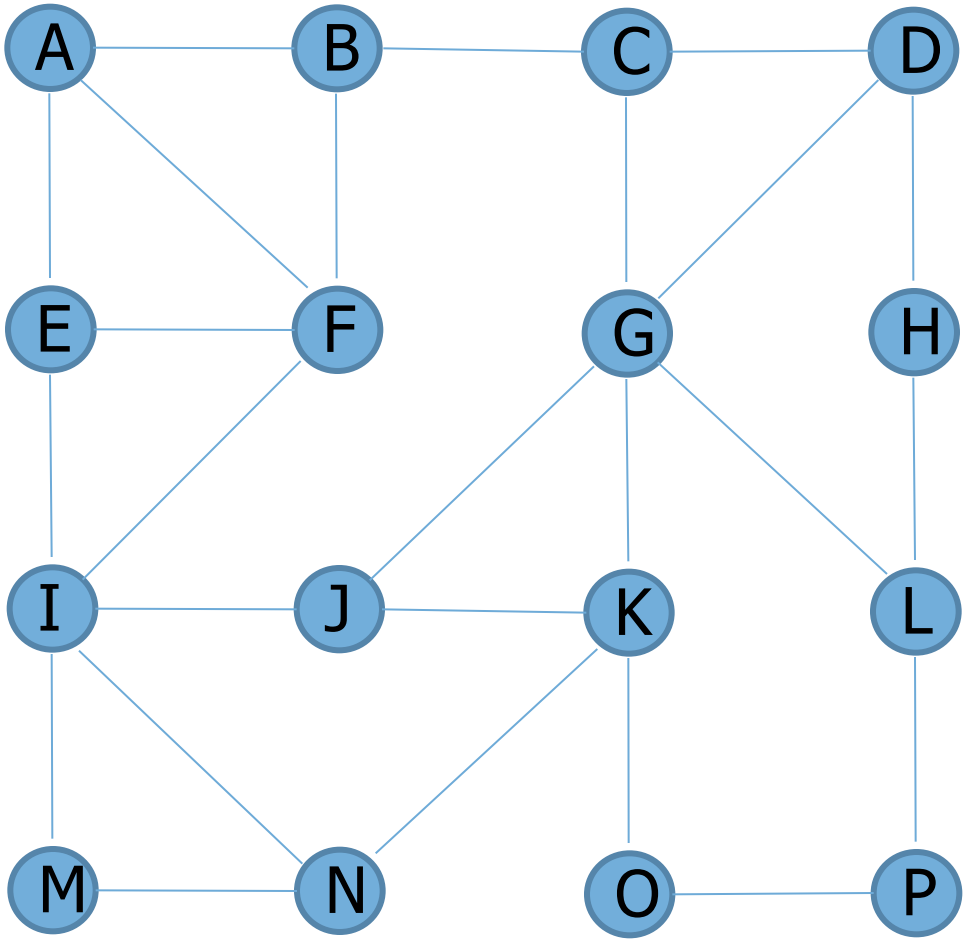


Shortest Path NOT defined

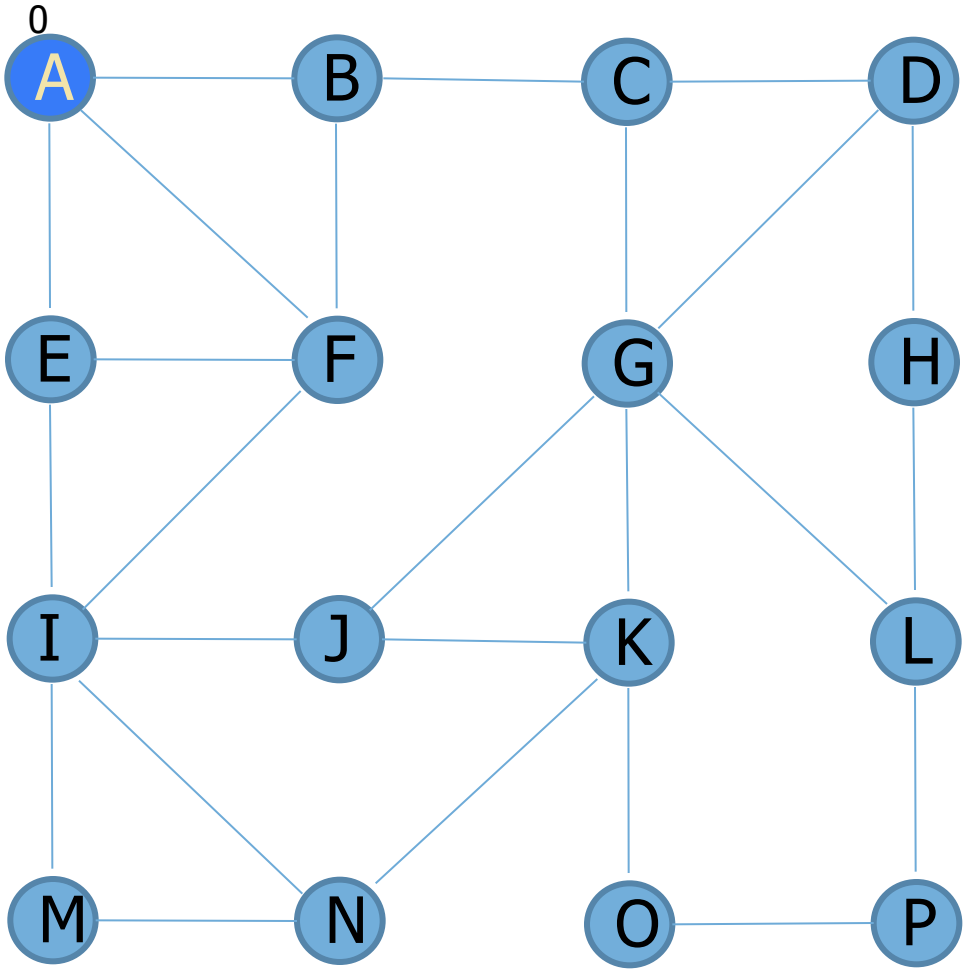
BFS - Breadth First Search

- 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

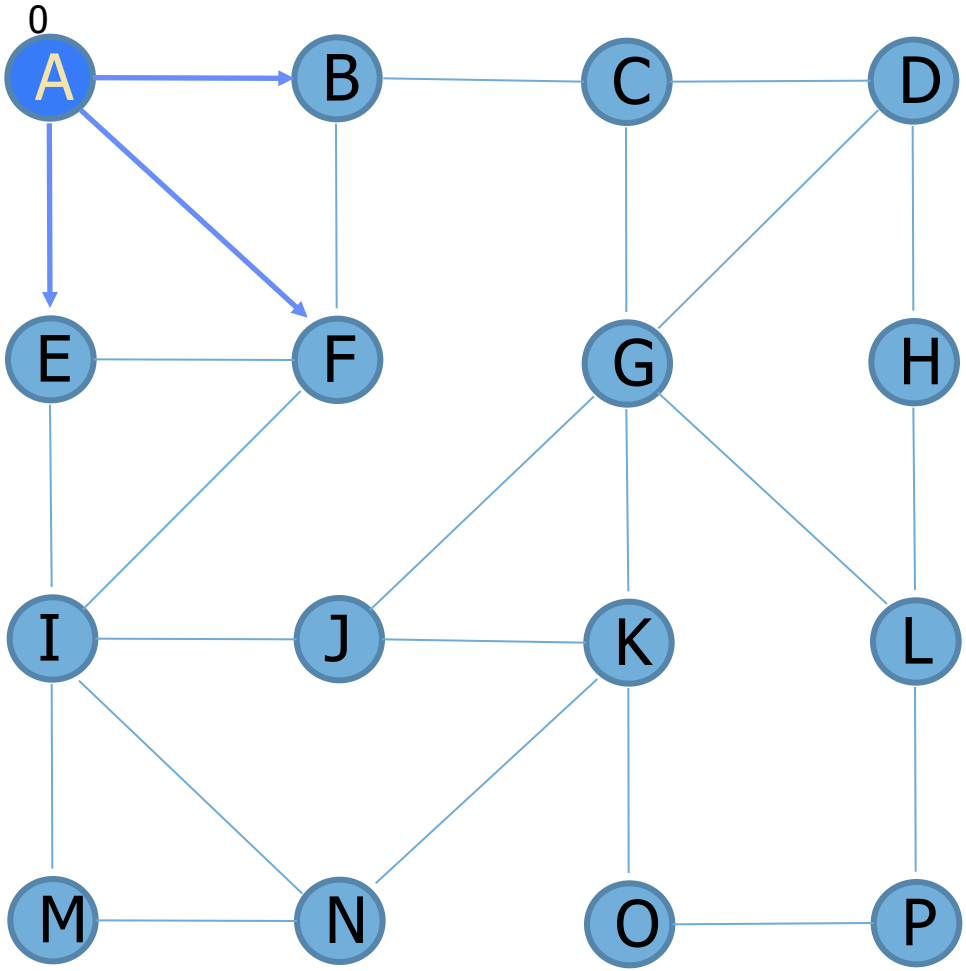
Breadth First Search



A visited

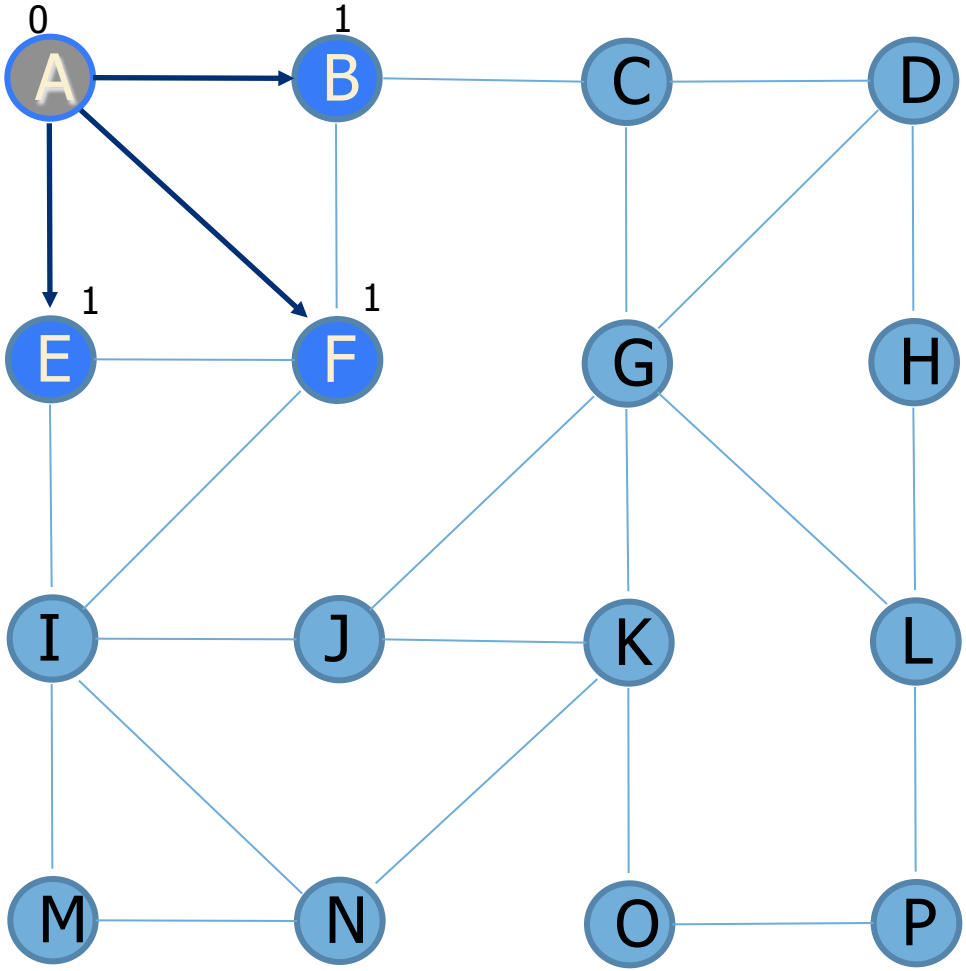


Explore A's neighbors



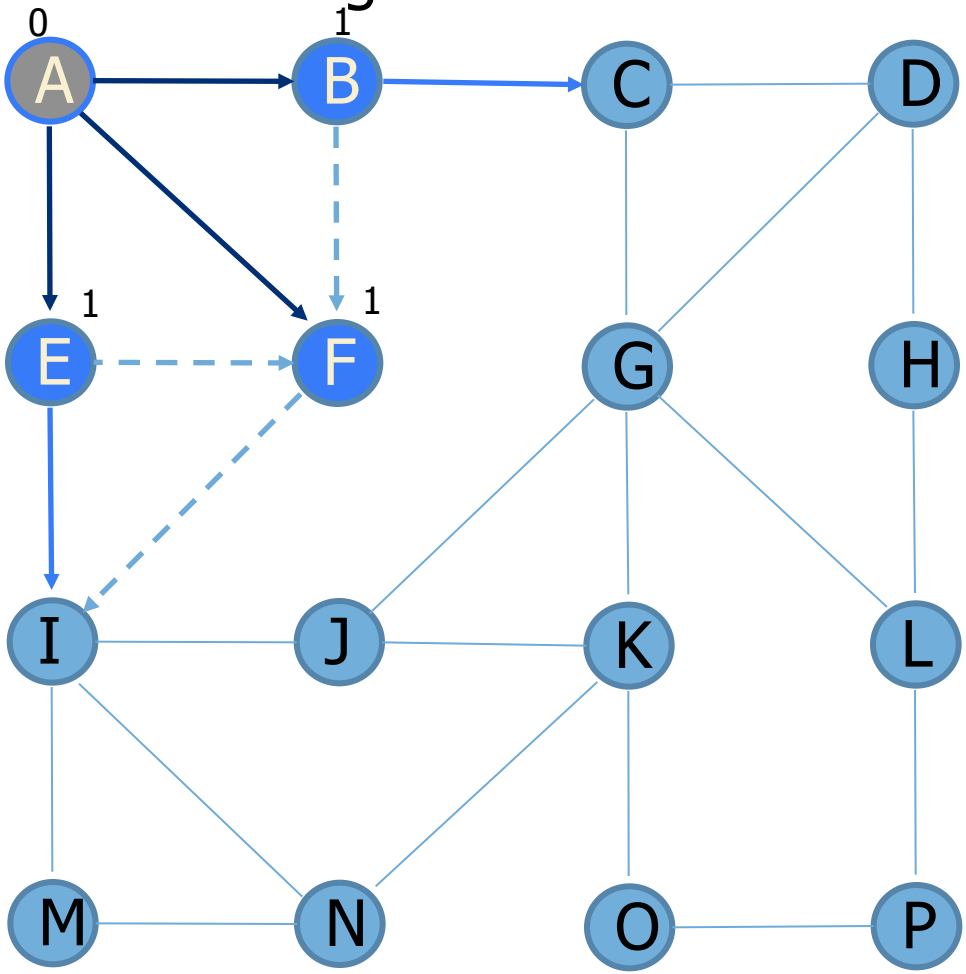
A is processed

B, E, F visited



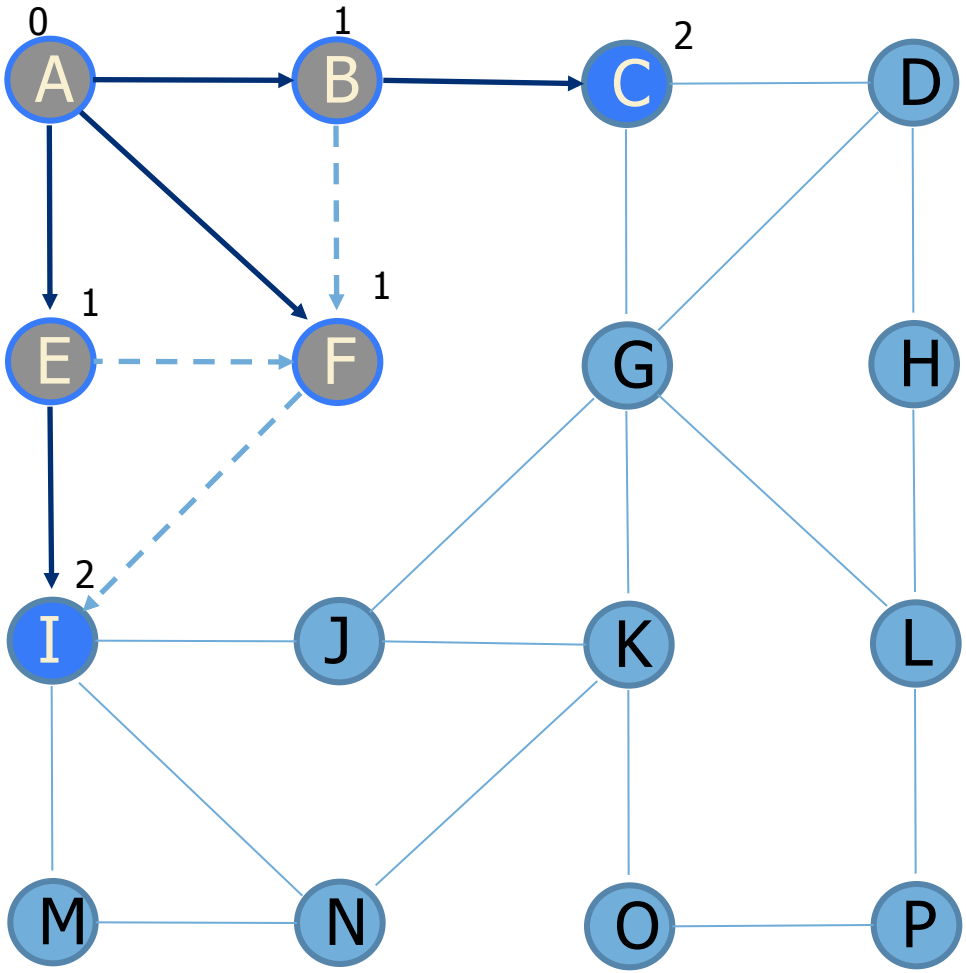
Explore B's neighbors

Explore E then F's neighbors

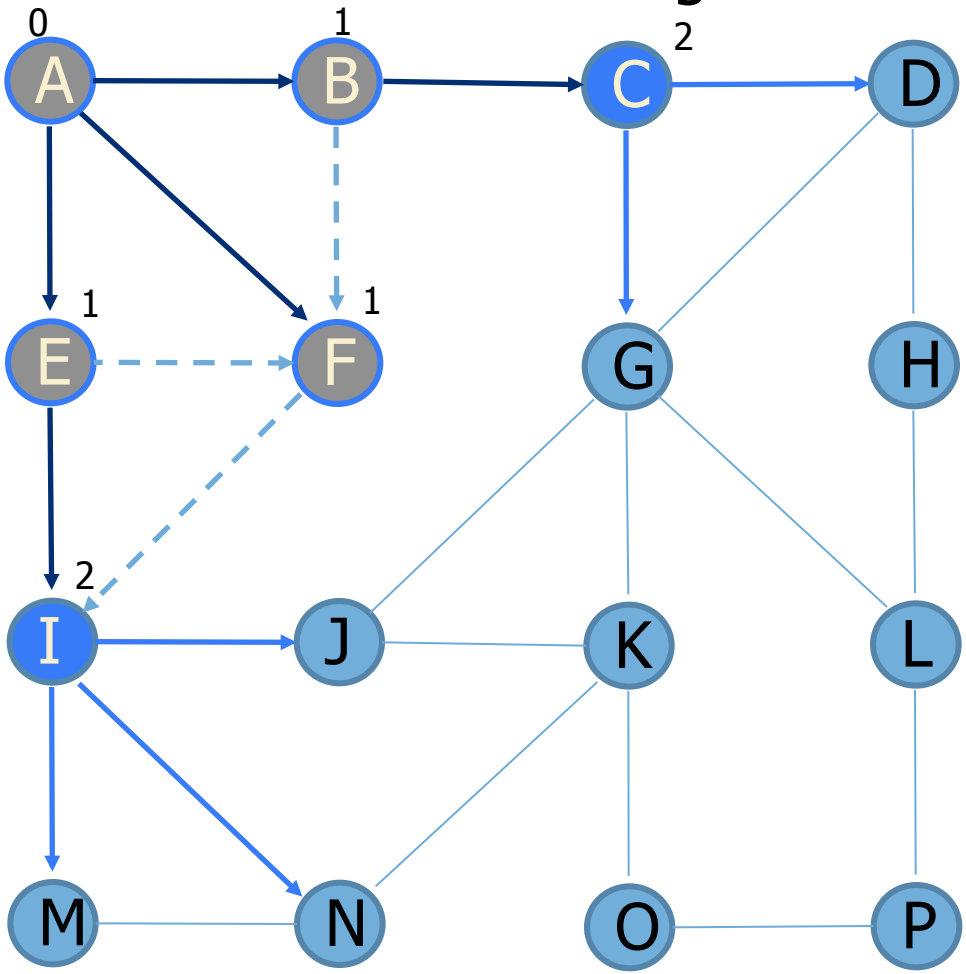


C, I visited

B,E,F are processed

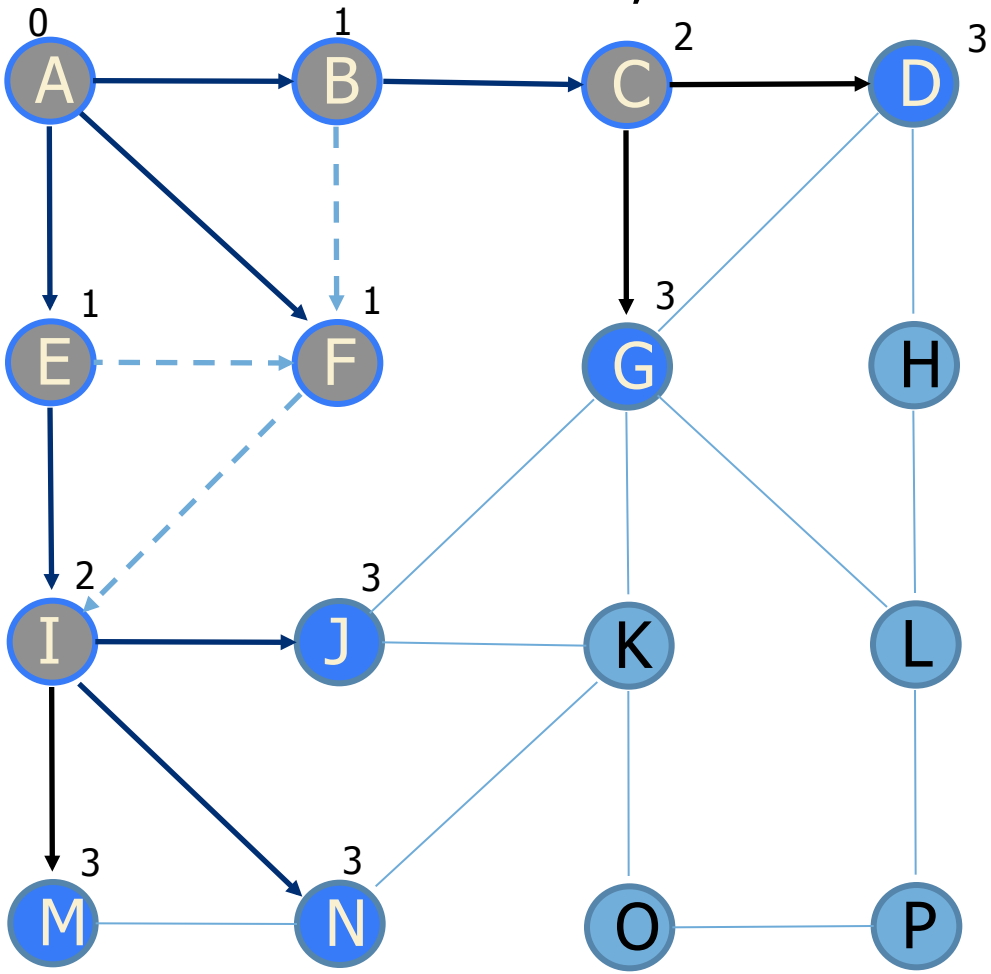


Explore C's neighbors



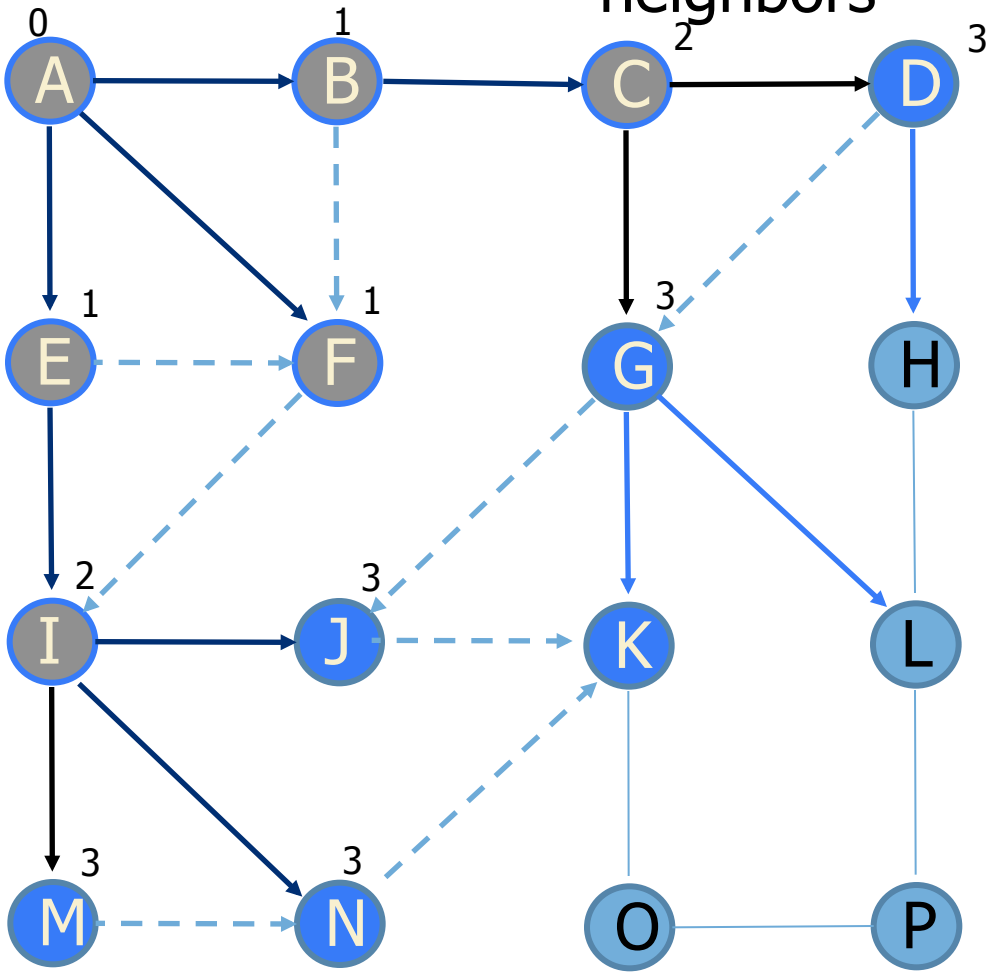
Explore I's neighbors

C processed
D,E visited



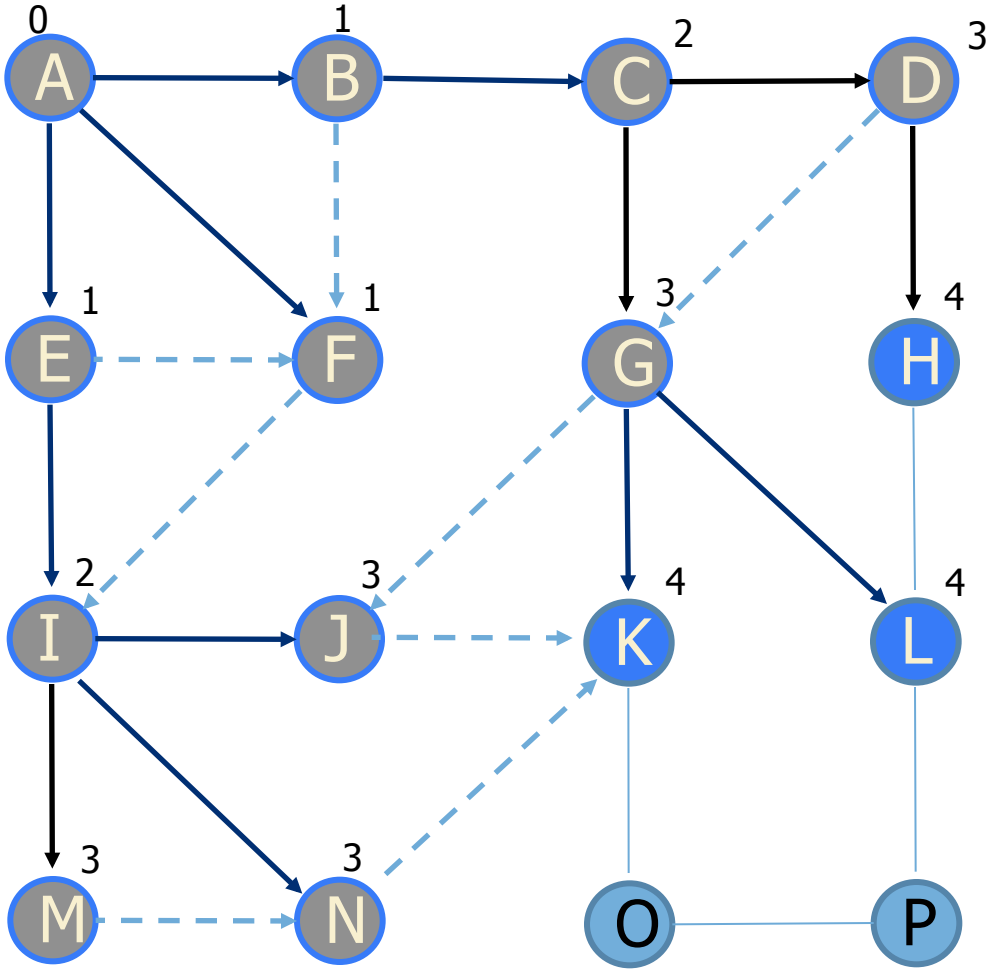
I processed
J,M,N
visited

Explore
D's,G's
neighbors



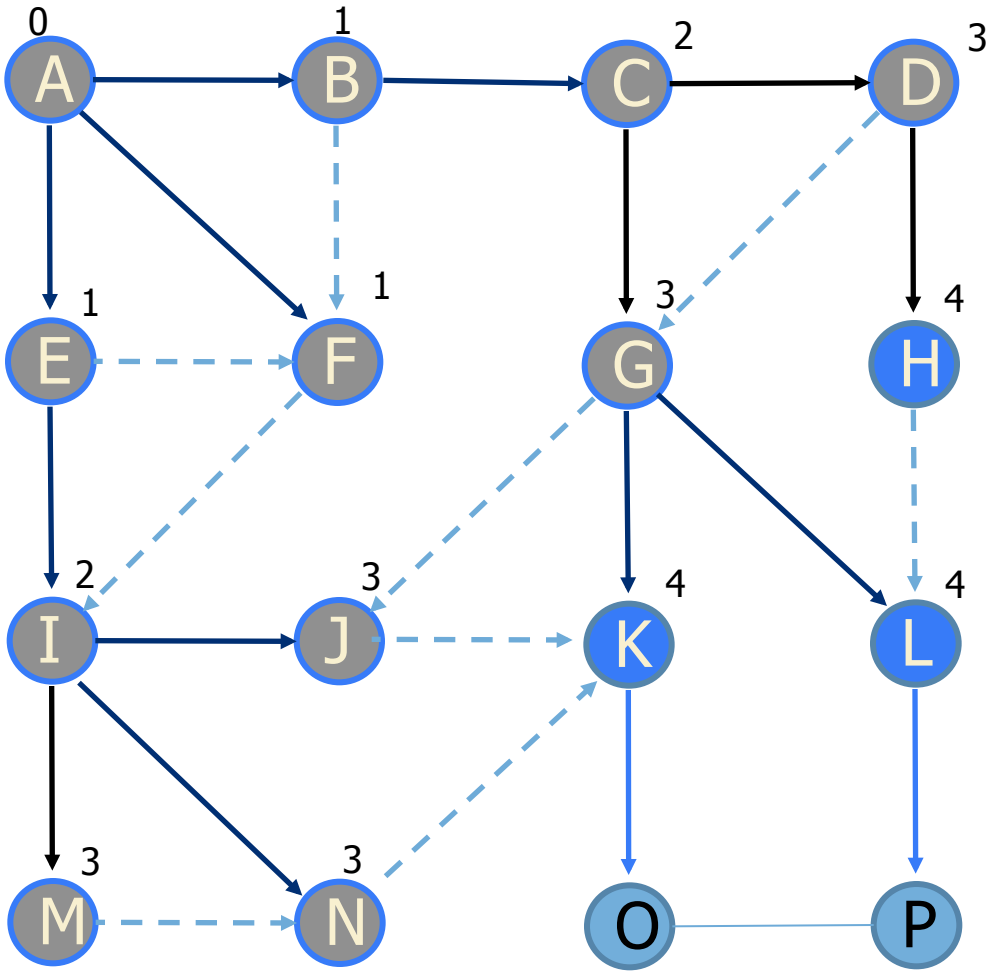
Explore
J's, M's,
N's
neighbors

D, G
processed

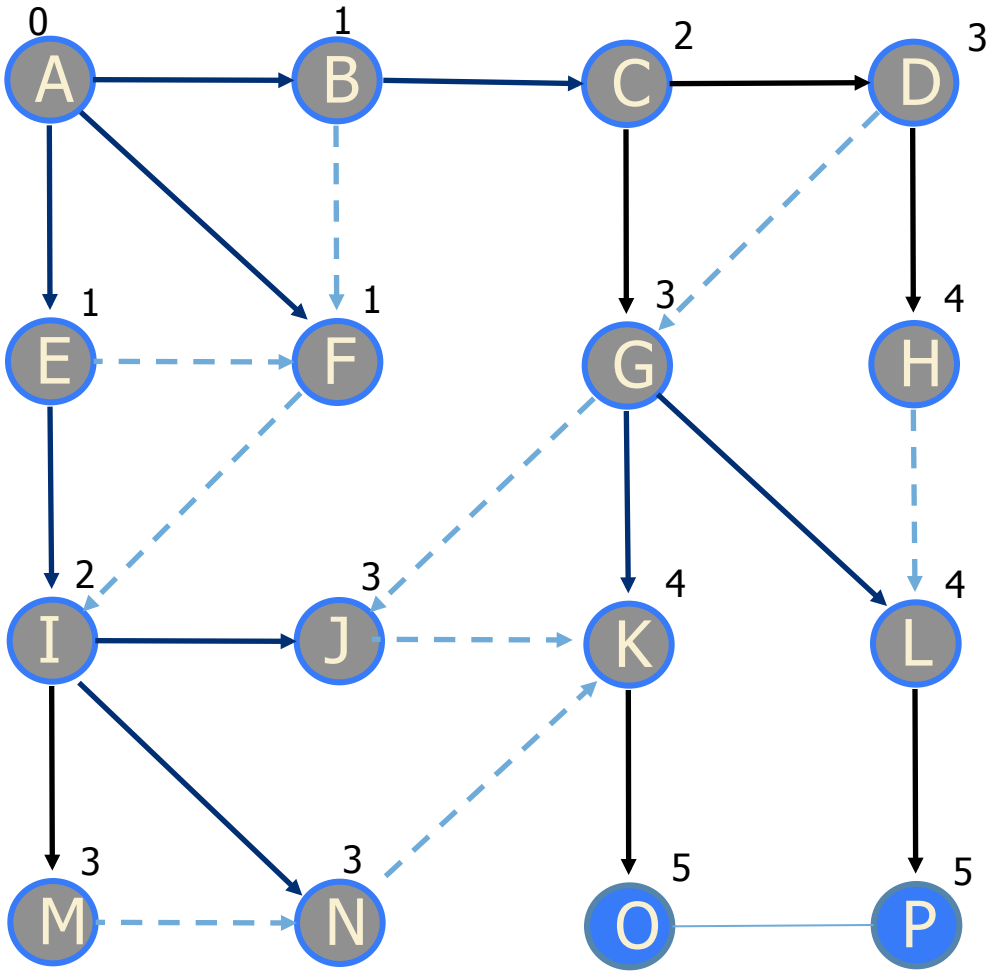


H, K, L
visited

J, M, N
processed

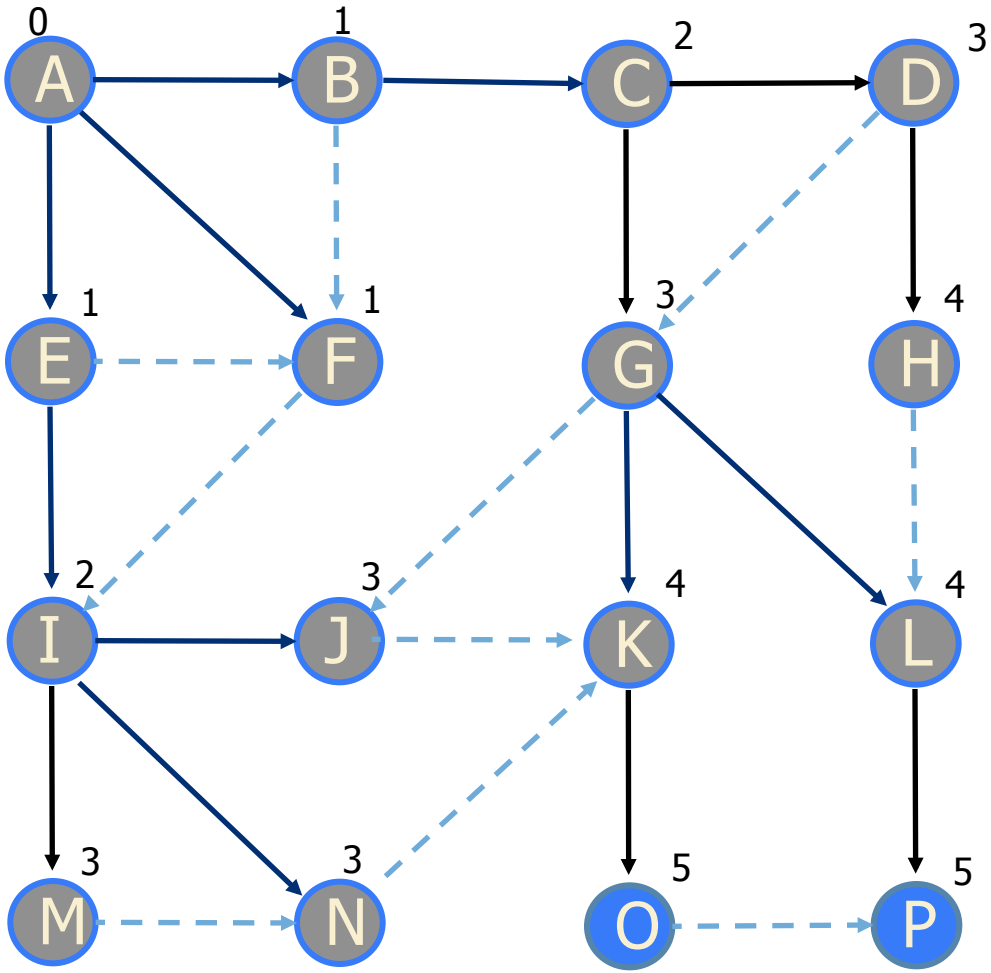


Explore
H, K, L
neighbors



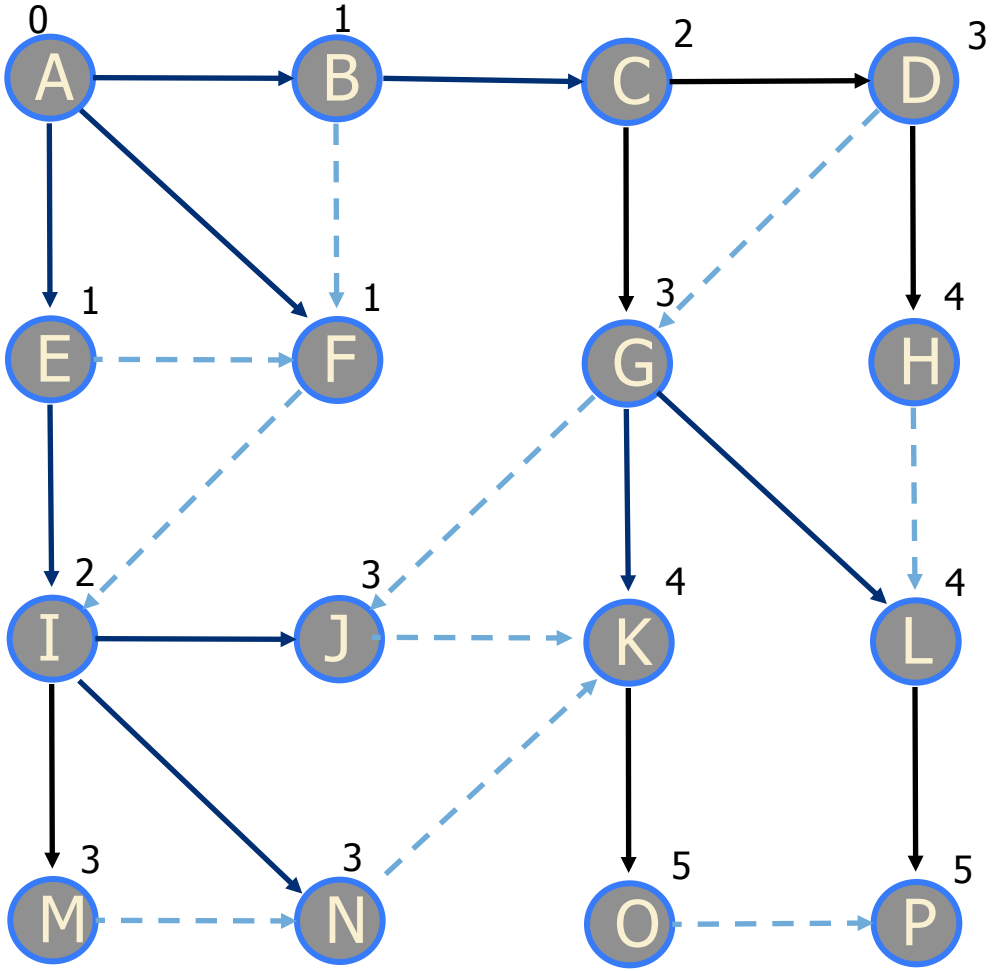
H, K, L processed

O,P visited



Explore O's neighbor

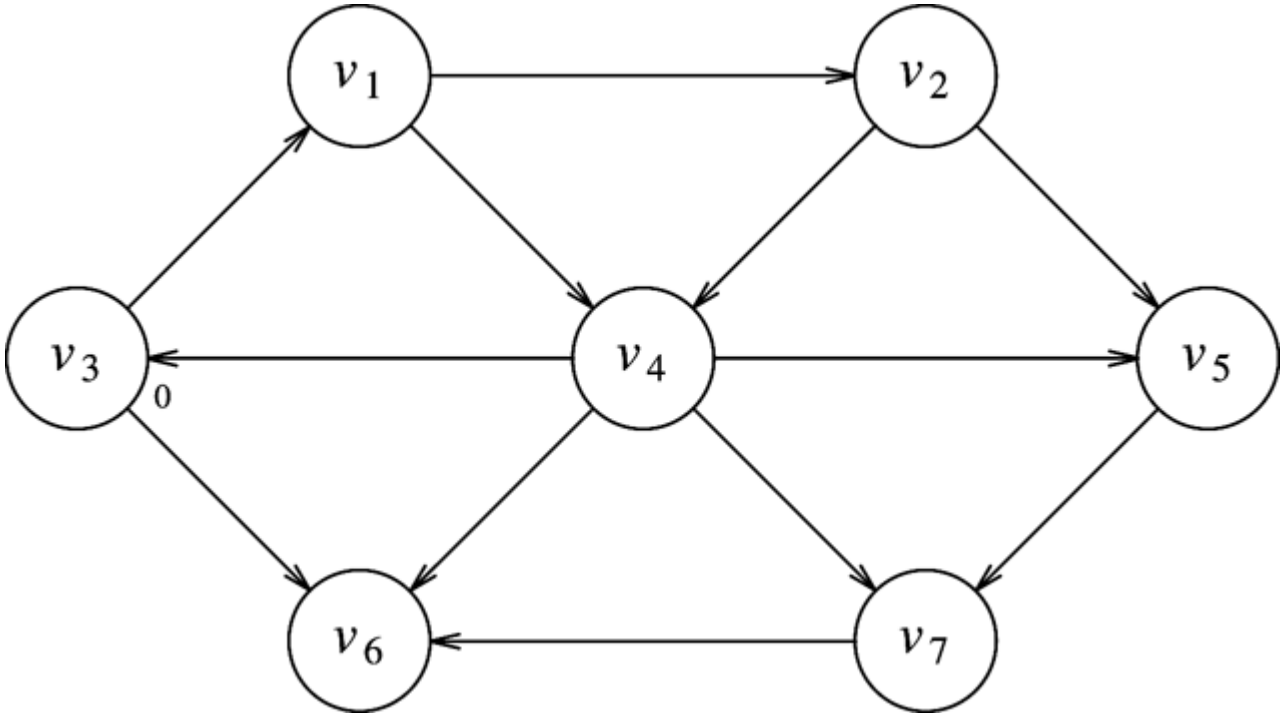
BFS



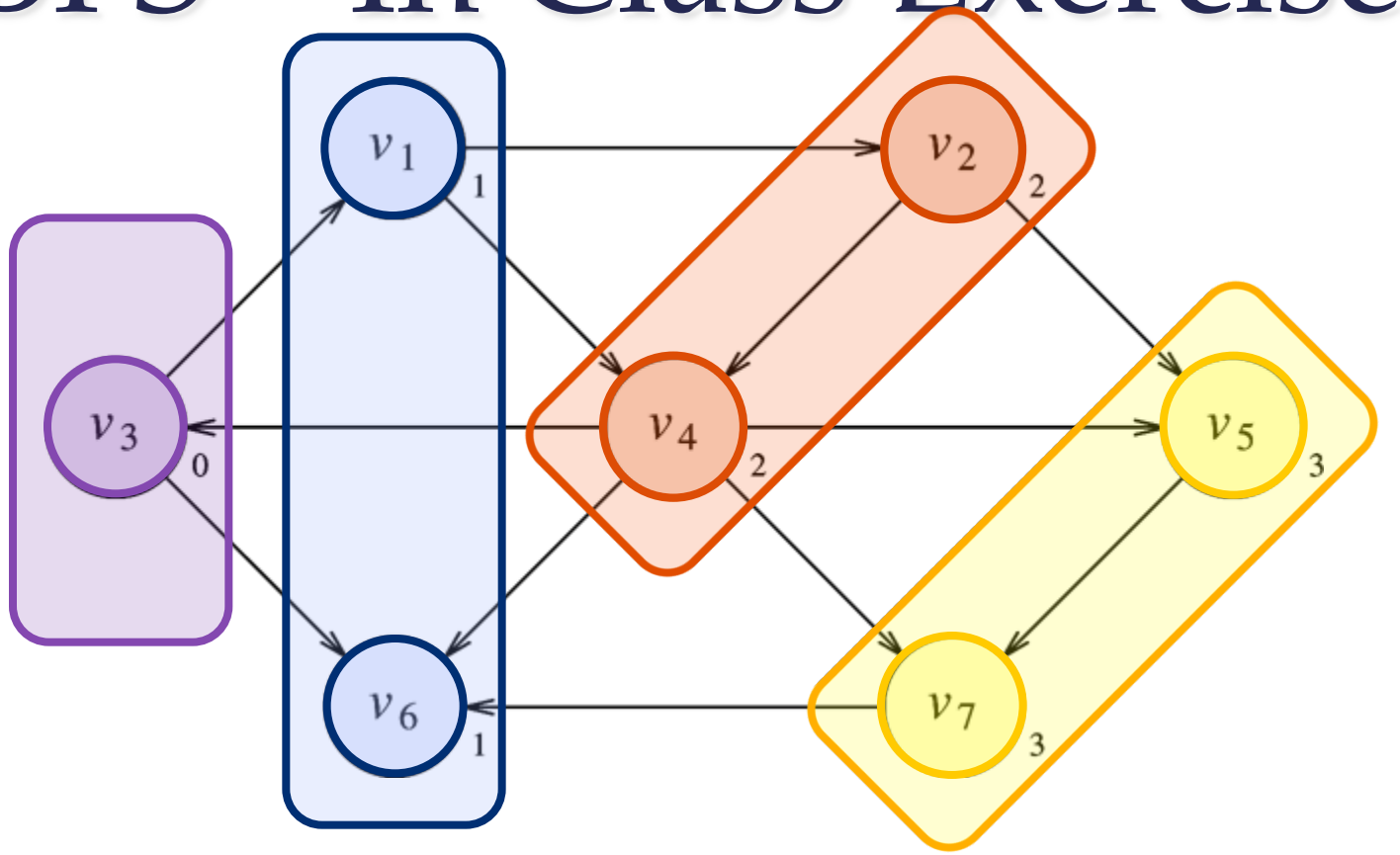
Done

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

BFS – In Class Exercise

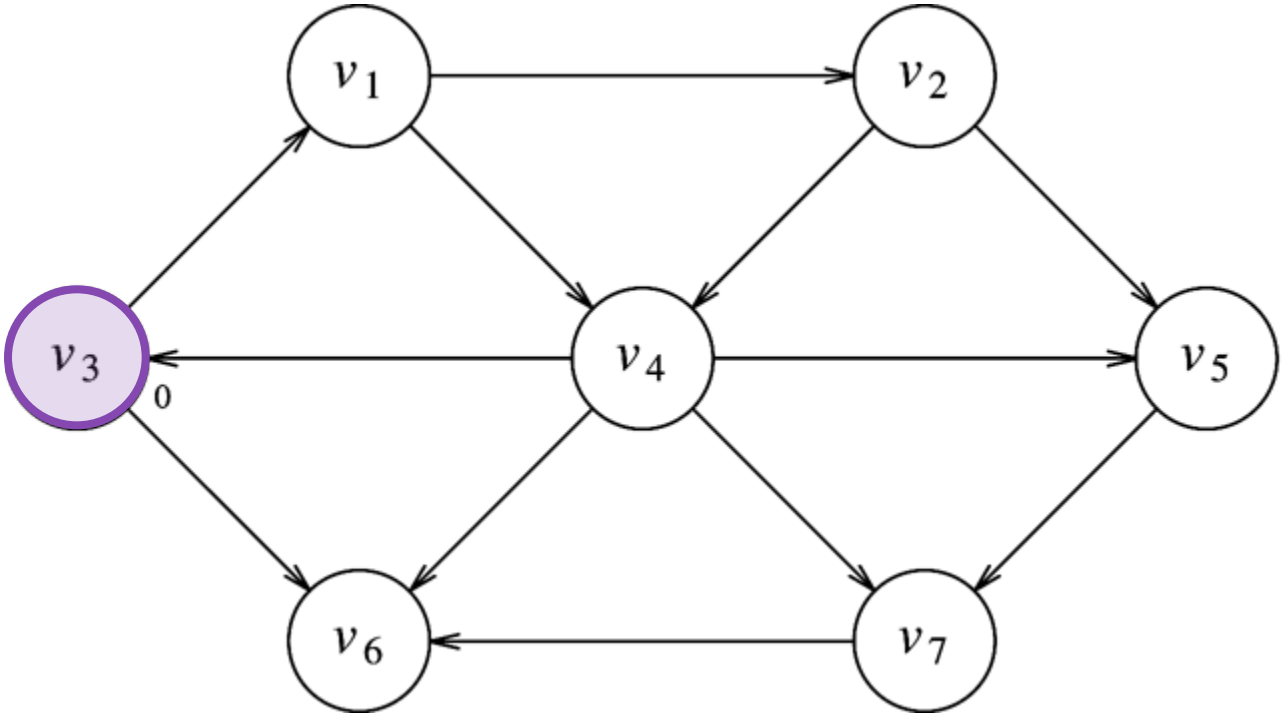


BFS – In Class Exercise

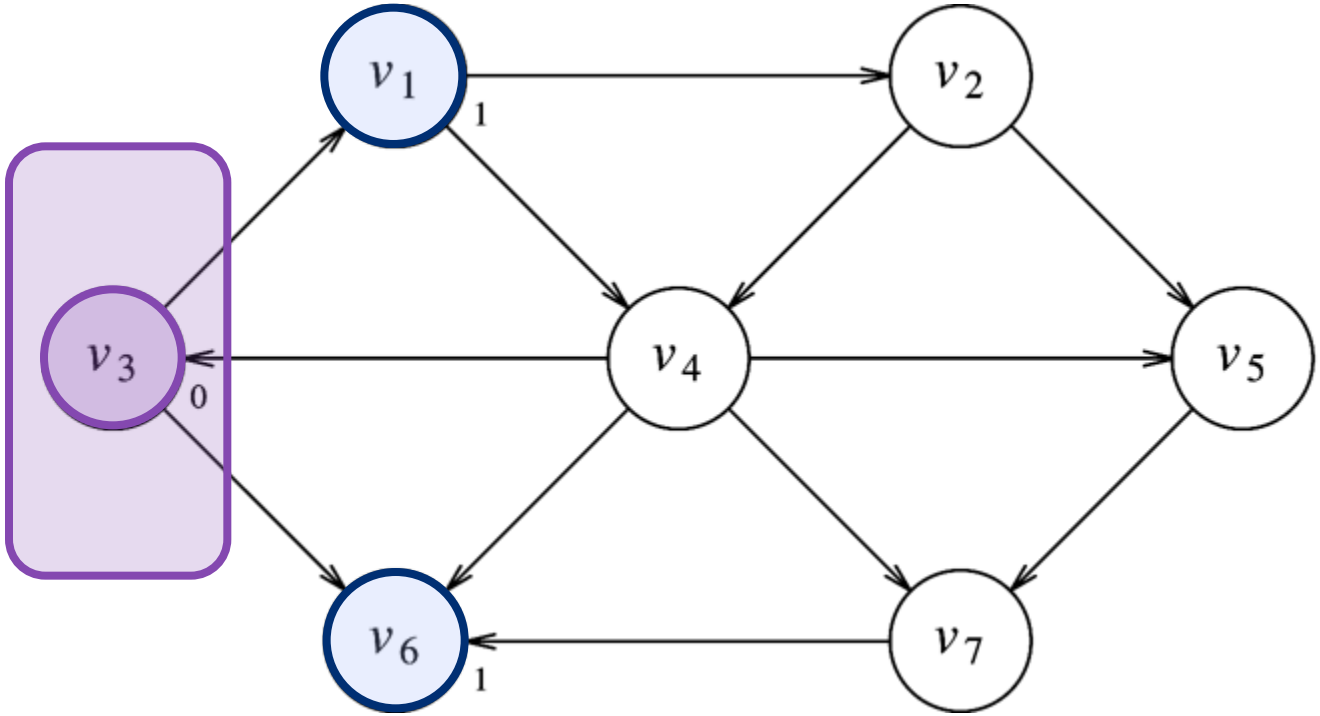


Running Time = $O(|E| + |V|)$, every V is explored, all E in worst case.

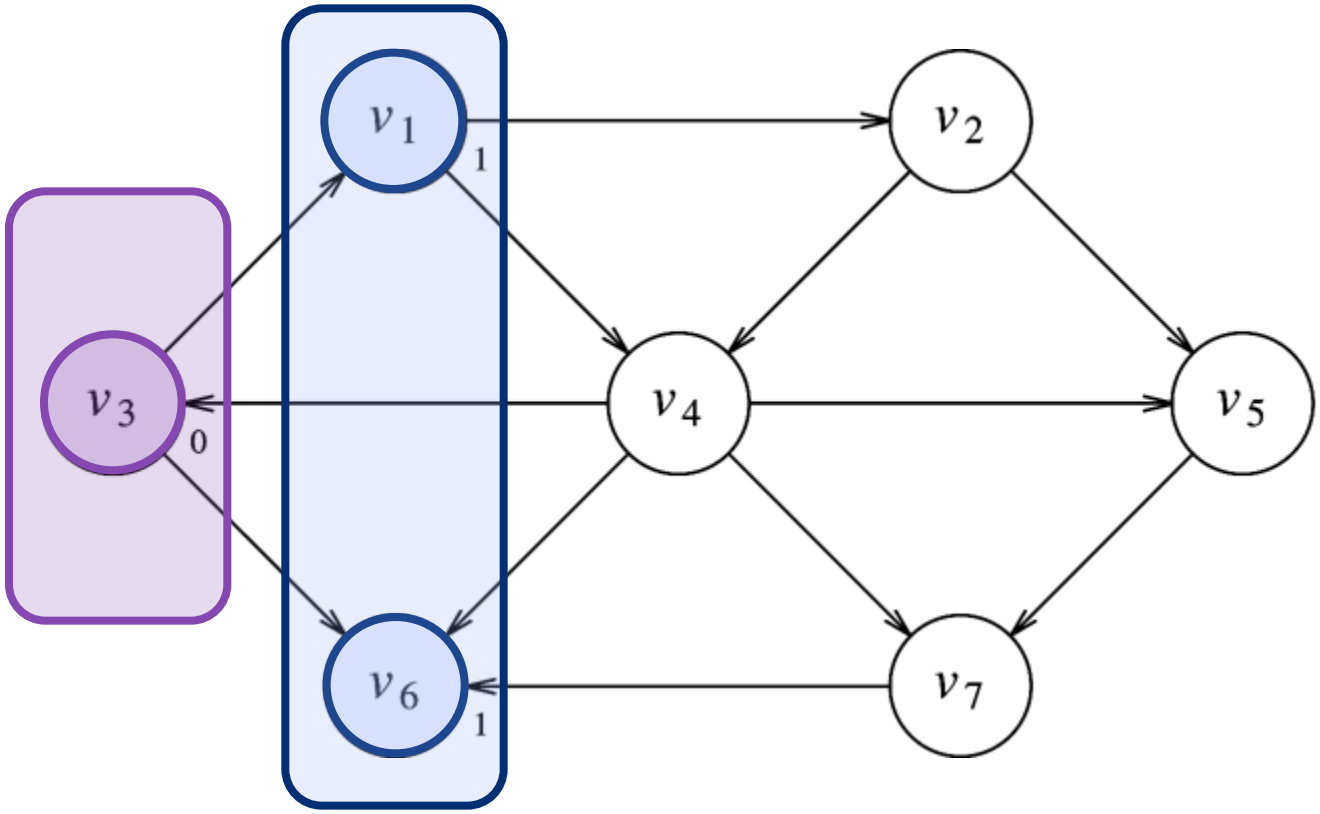
BFS – In Class Exercise



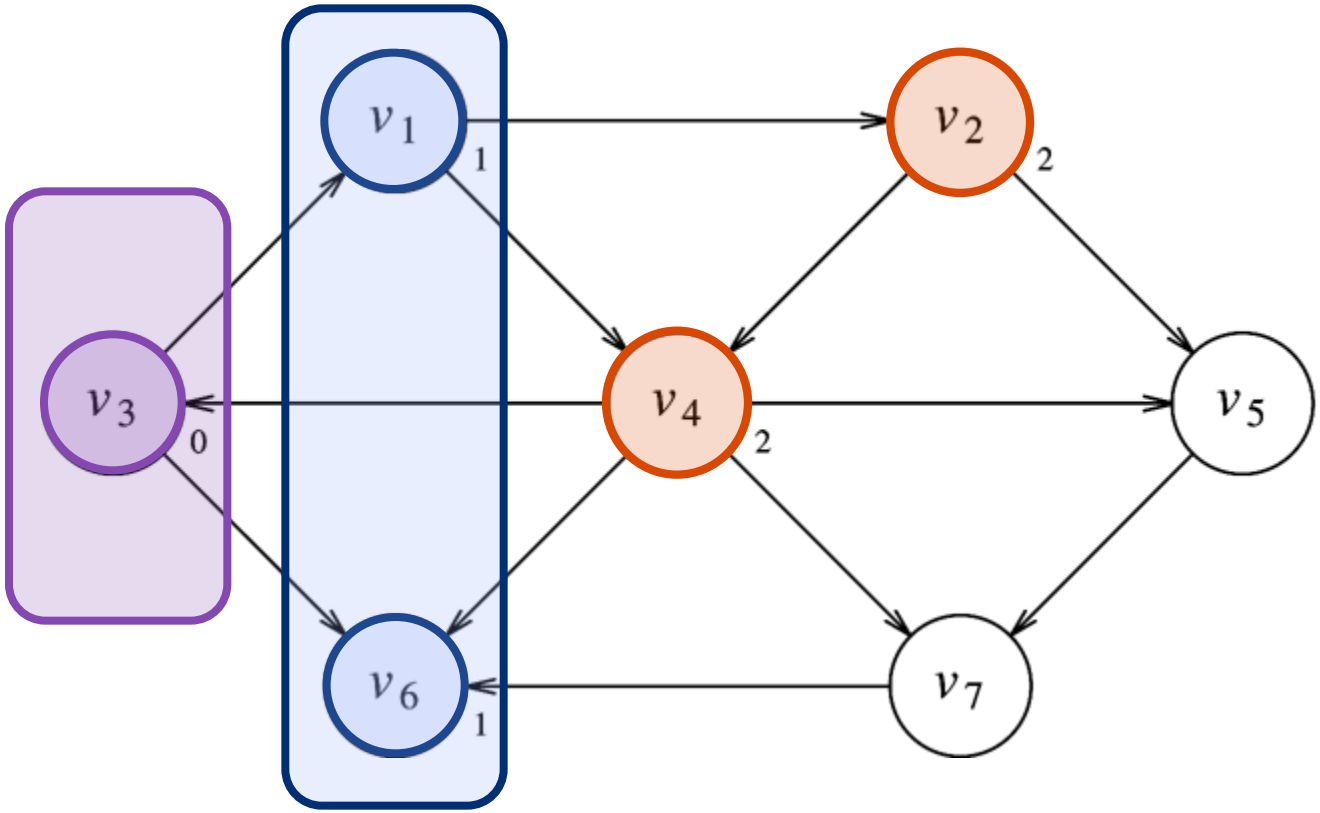
BFS – In Class Exercise



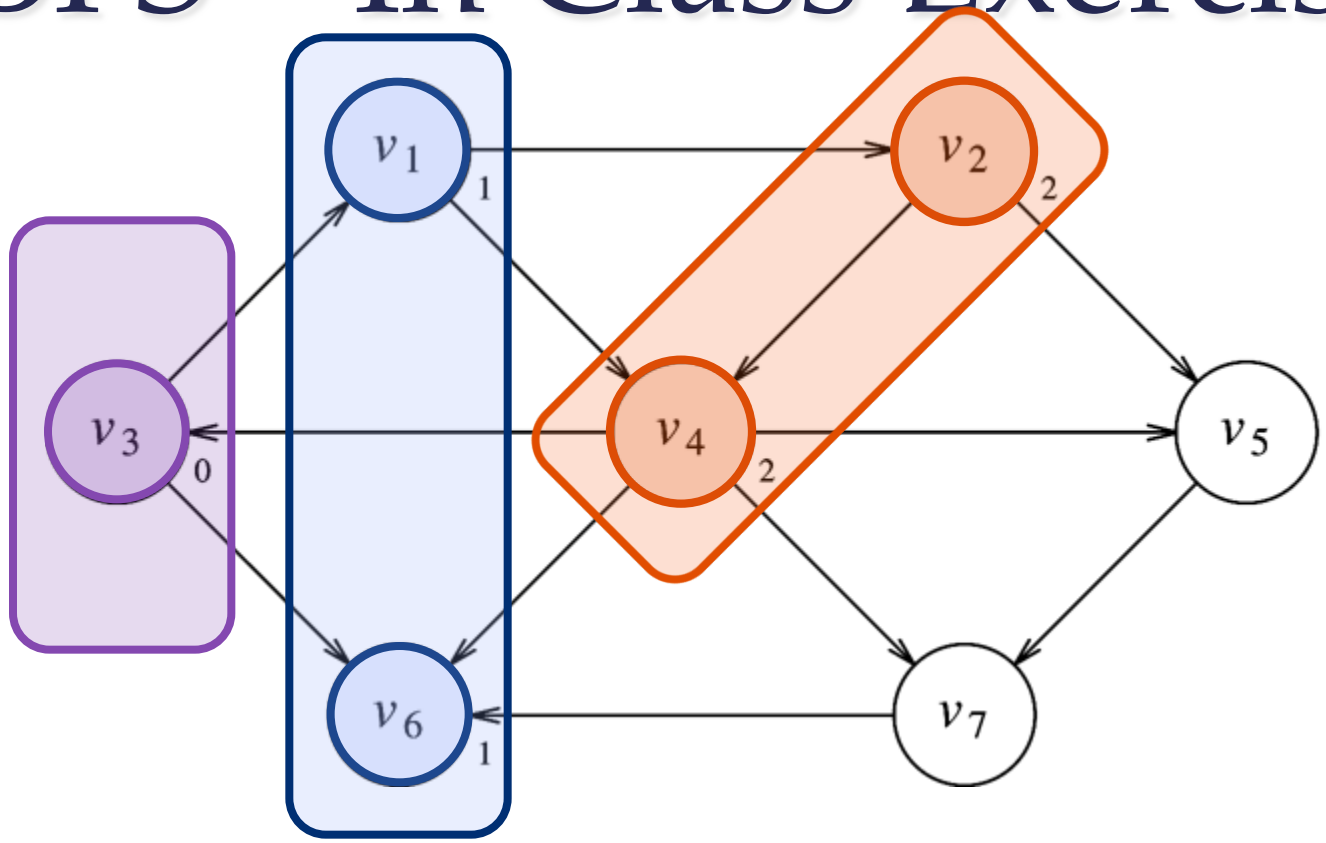
BFS – In Class Exercise



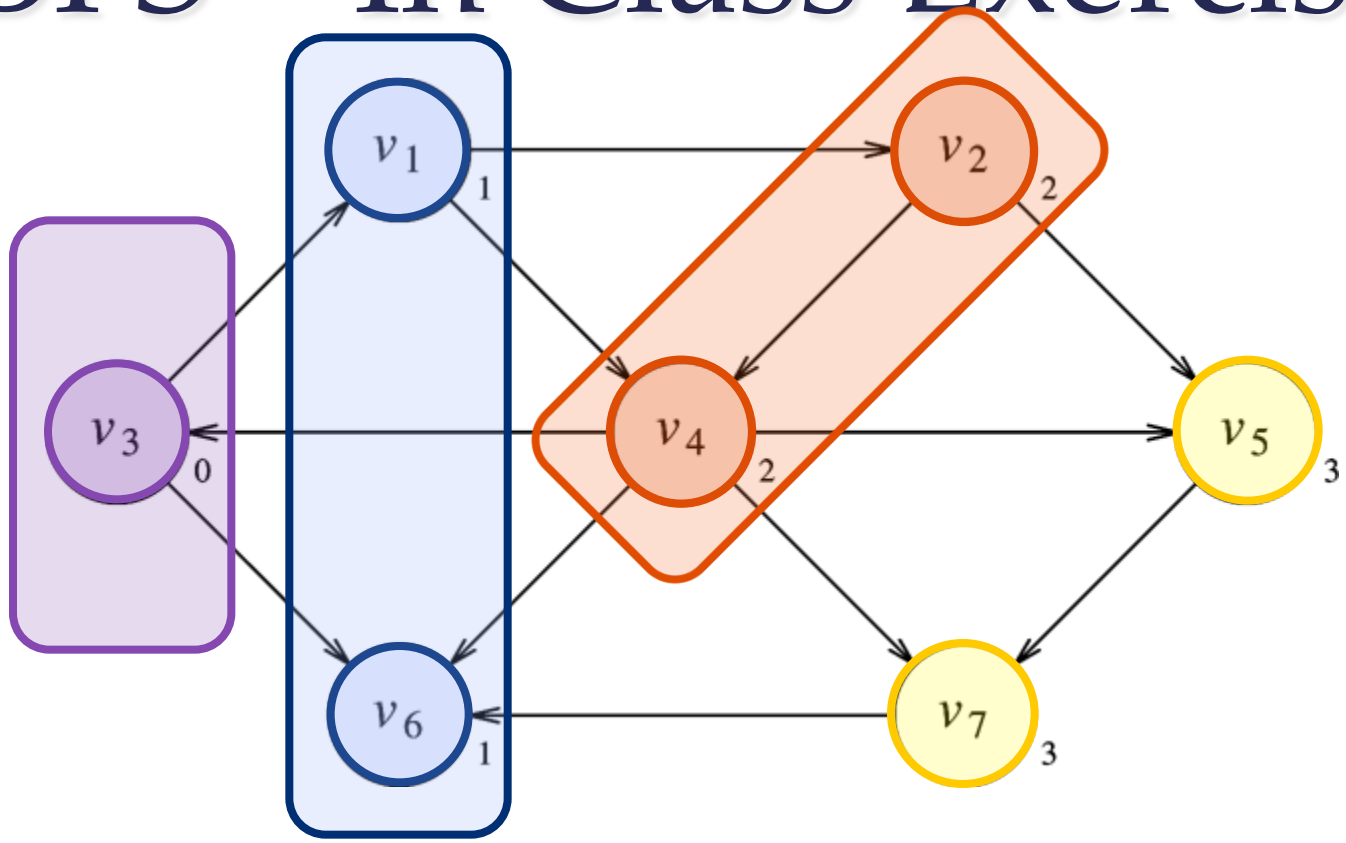
BFS – In Class Exercise



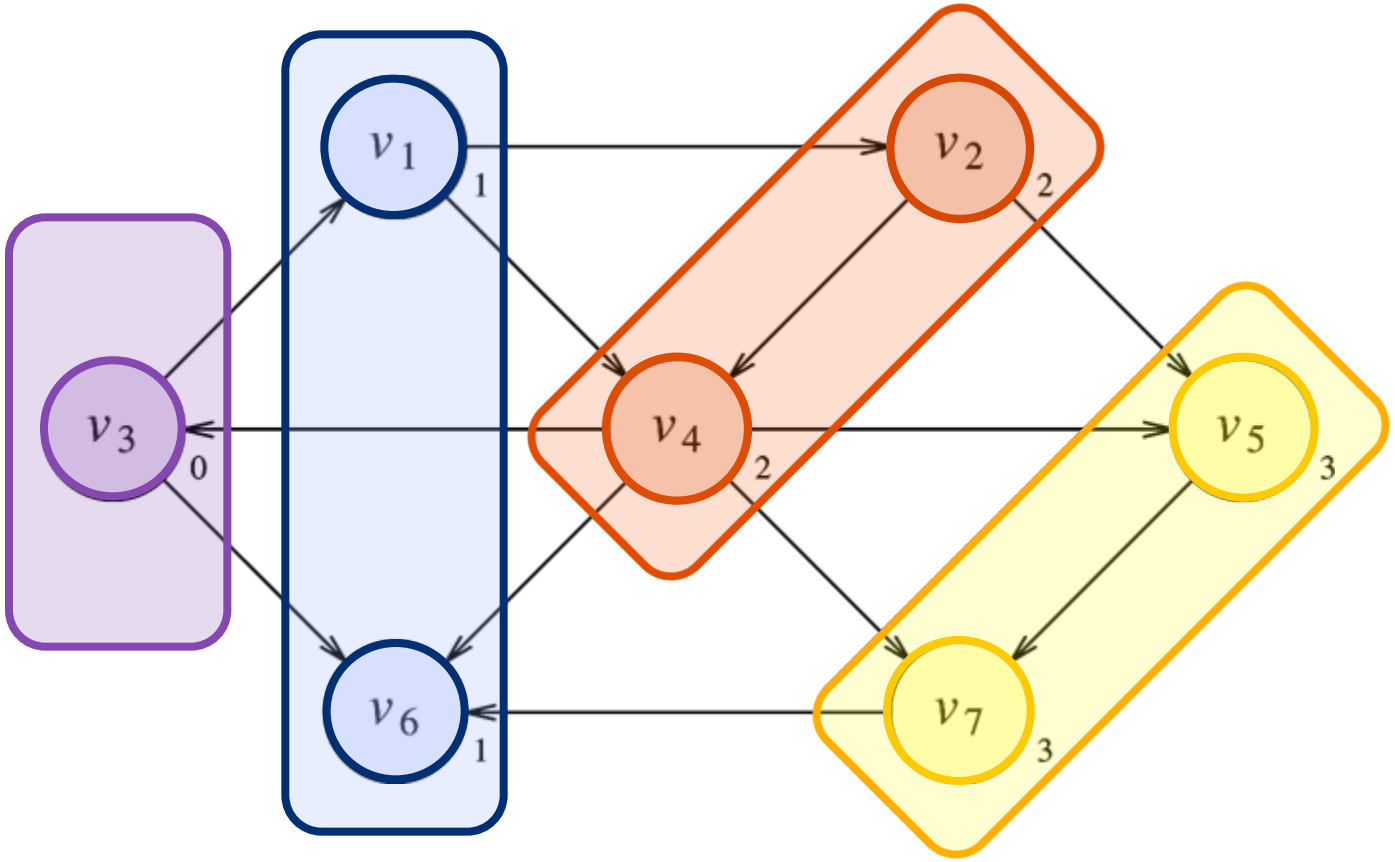
BFS – In Class Exercise



BFS – In Class Exercise

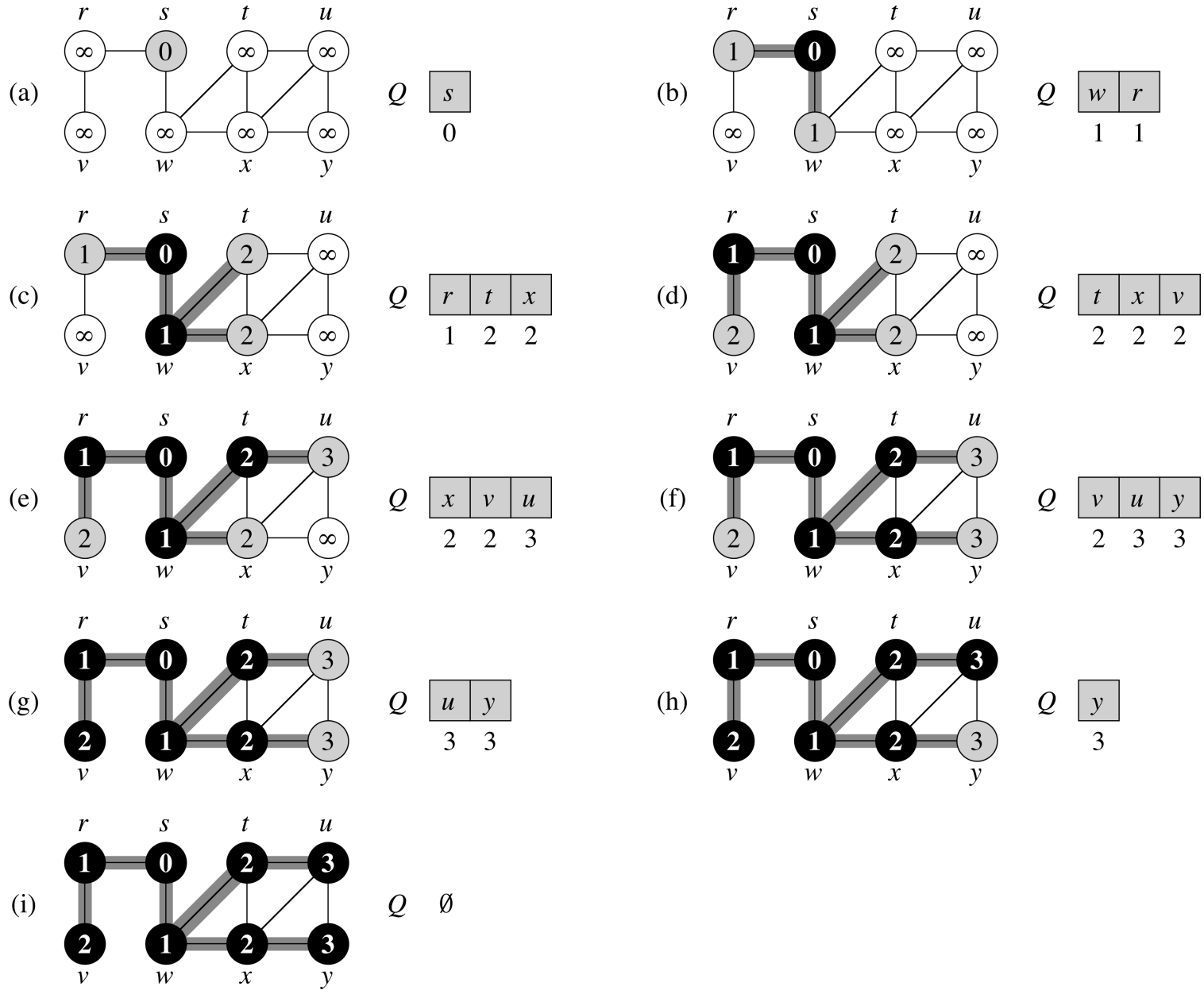


BFS – In Class Exercise



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

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q
s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q
w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
∅

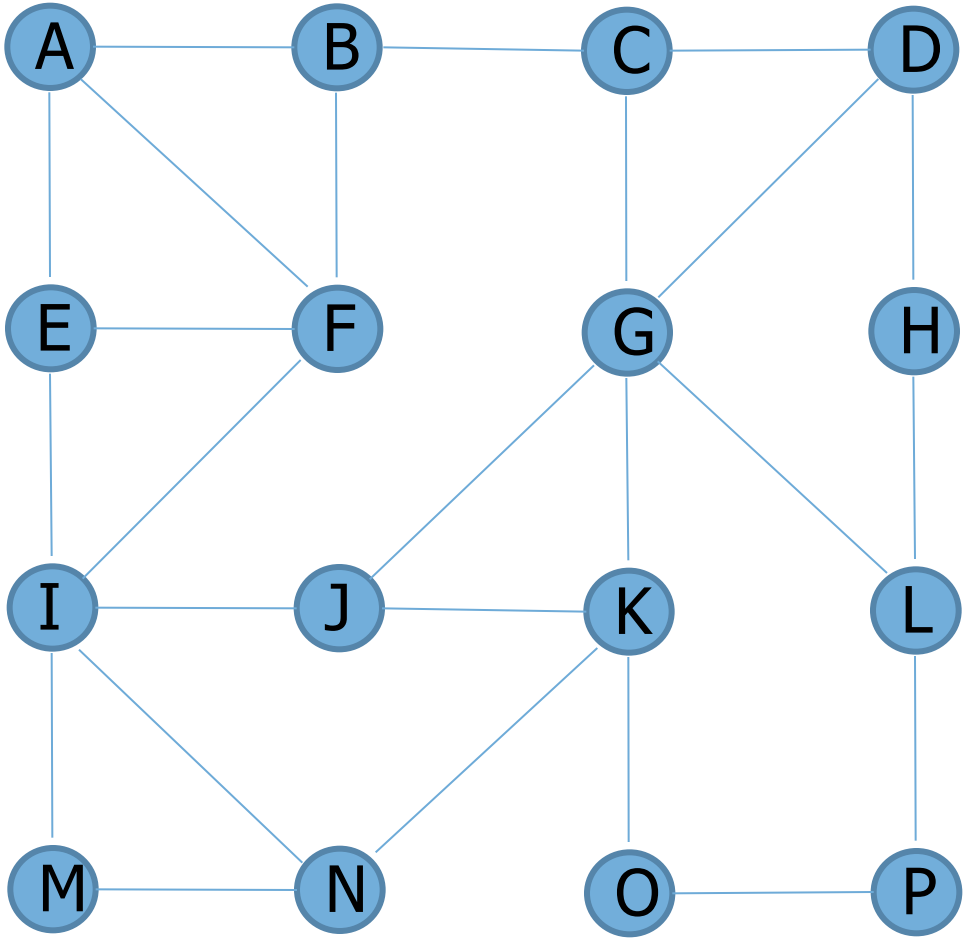
```
BFS(V, E, s)
  for each u ∈ V − {s}
        u.d = ∞
  s.d = 0
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
              if v.d == ∞
                    v.d = u.d + 1
                    ENQUEUE(Q, v)
```

DFS - Depth First Search

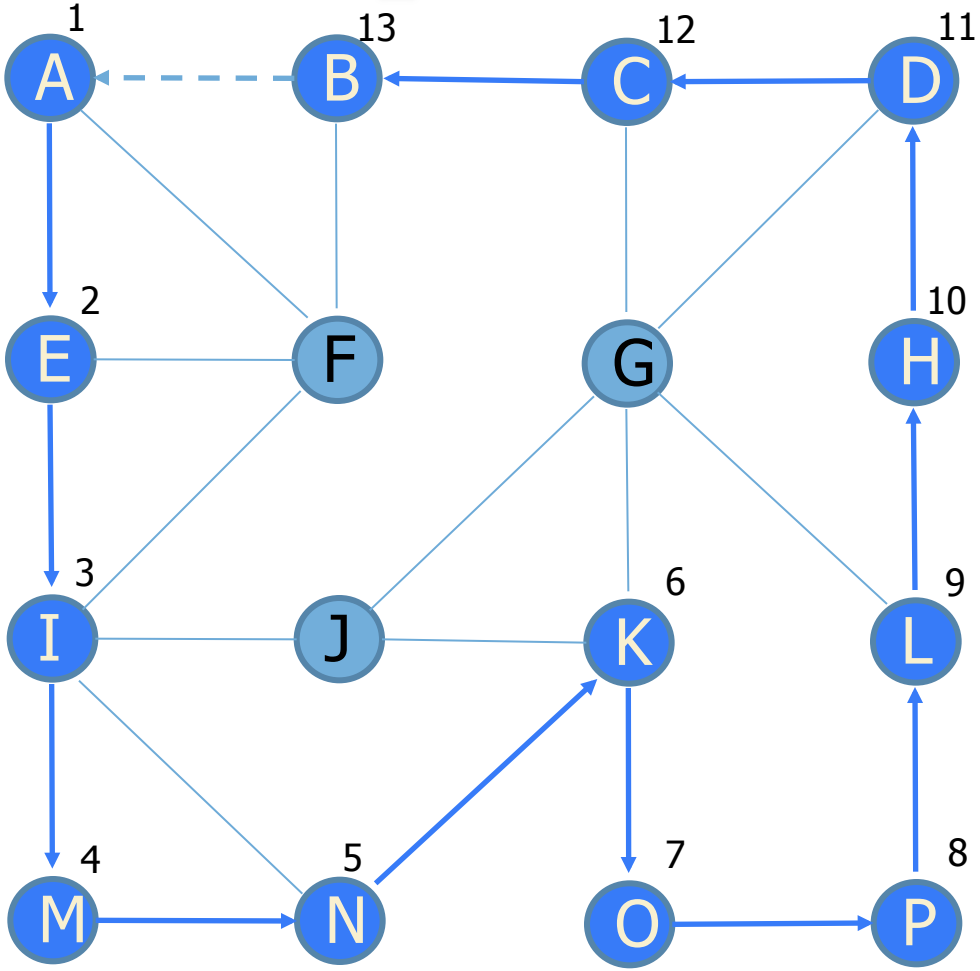
Generalization of a preorder traversal.

1. Start at an arbitrary vertex s.
2. Traverse (visit) as many descendents 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.

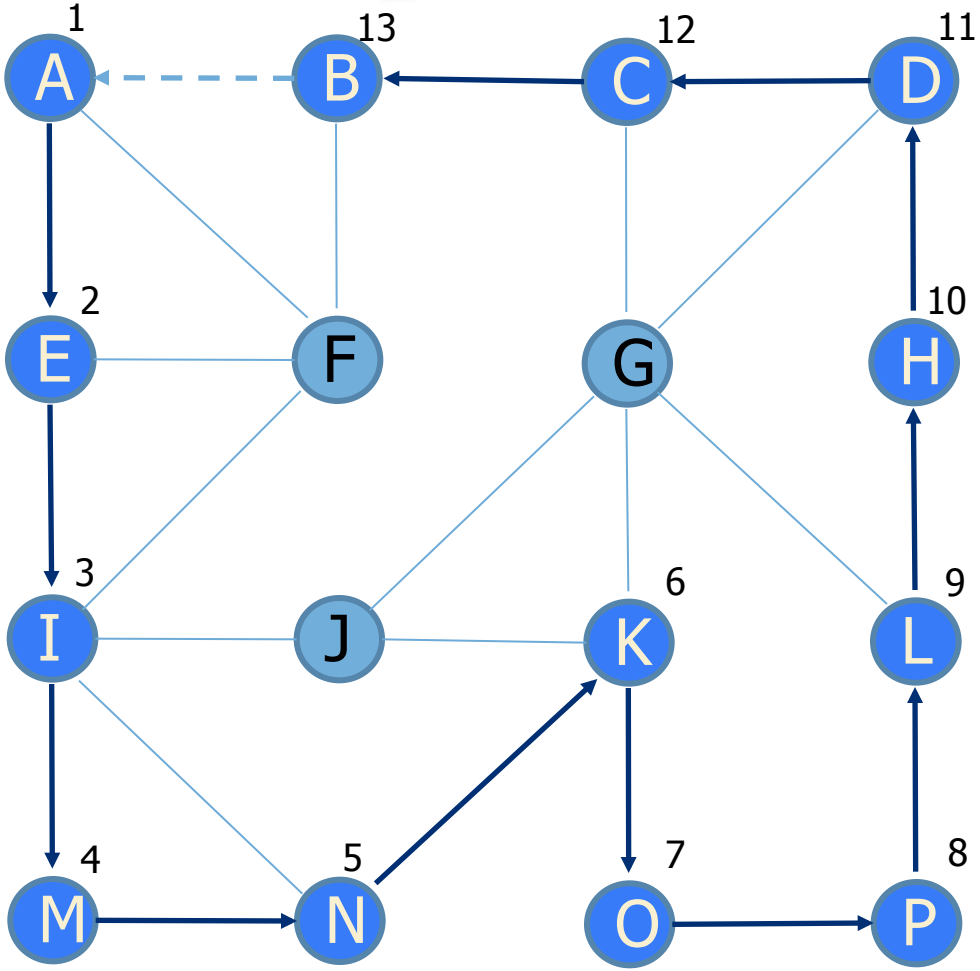
DFS - Depth First Search



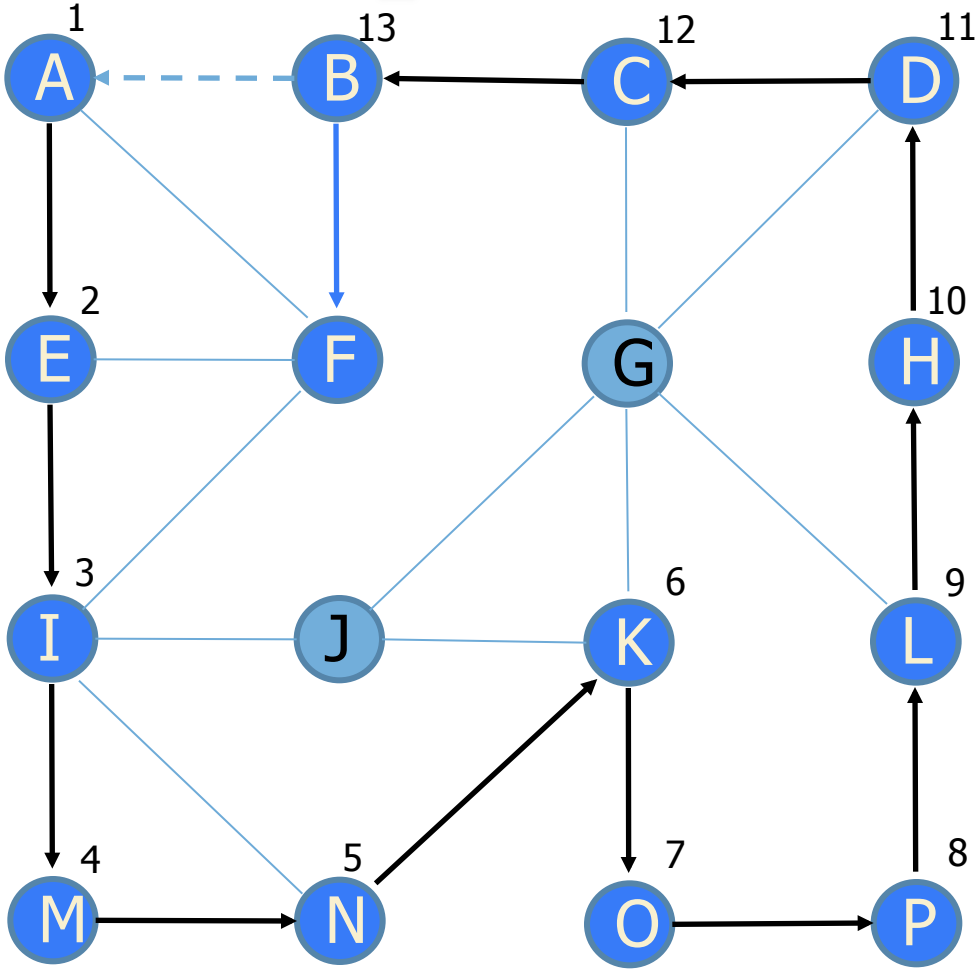
DFS - Depth First Search



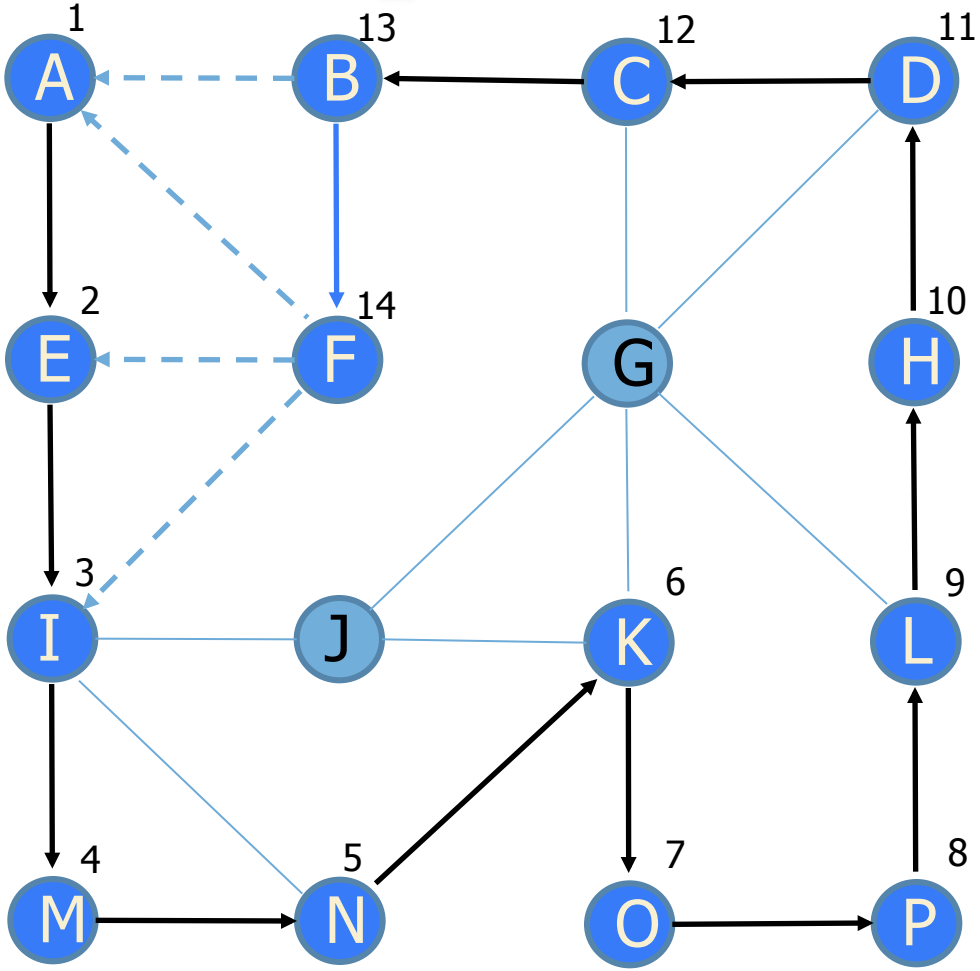
DFS - Depth First Search



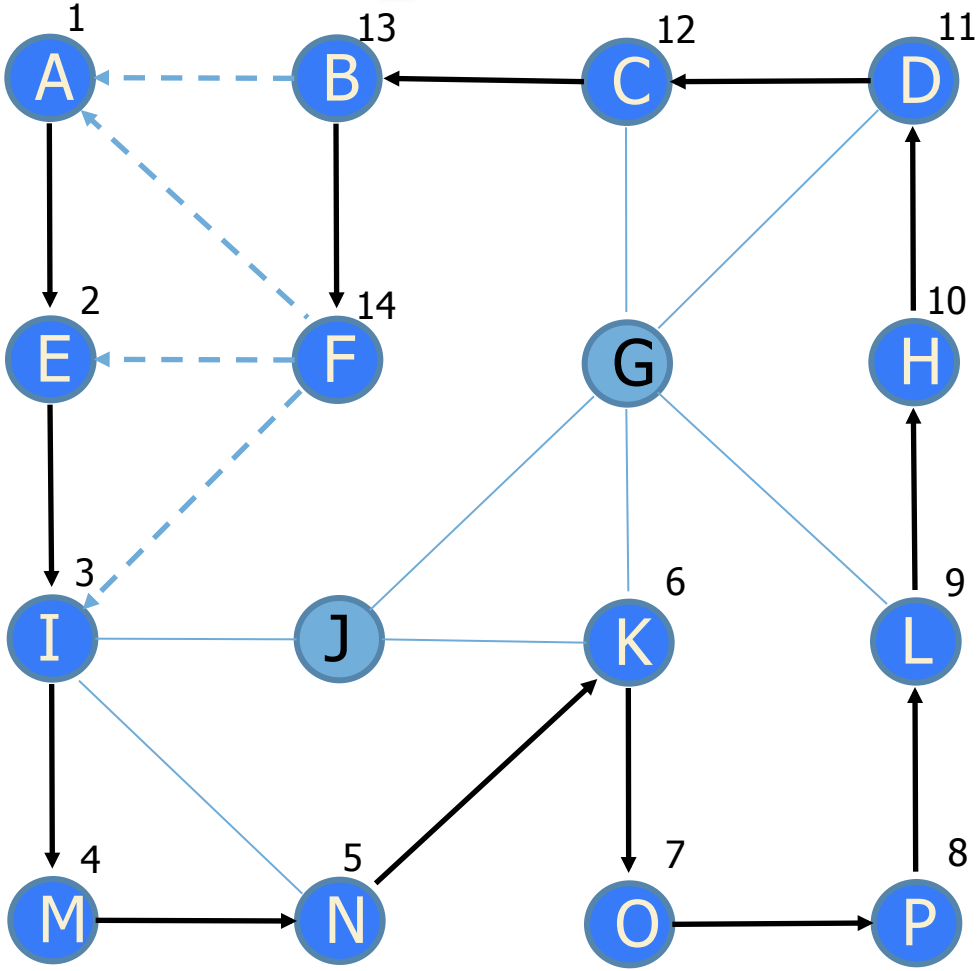
DFS - Depth First Search



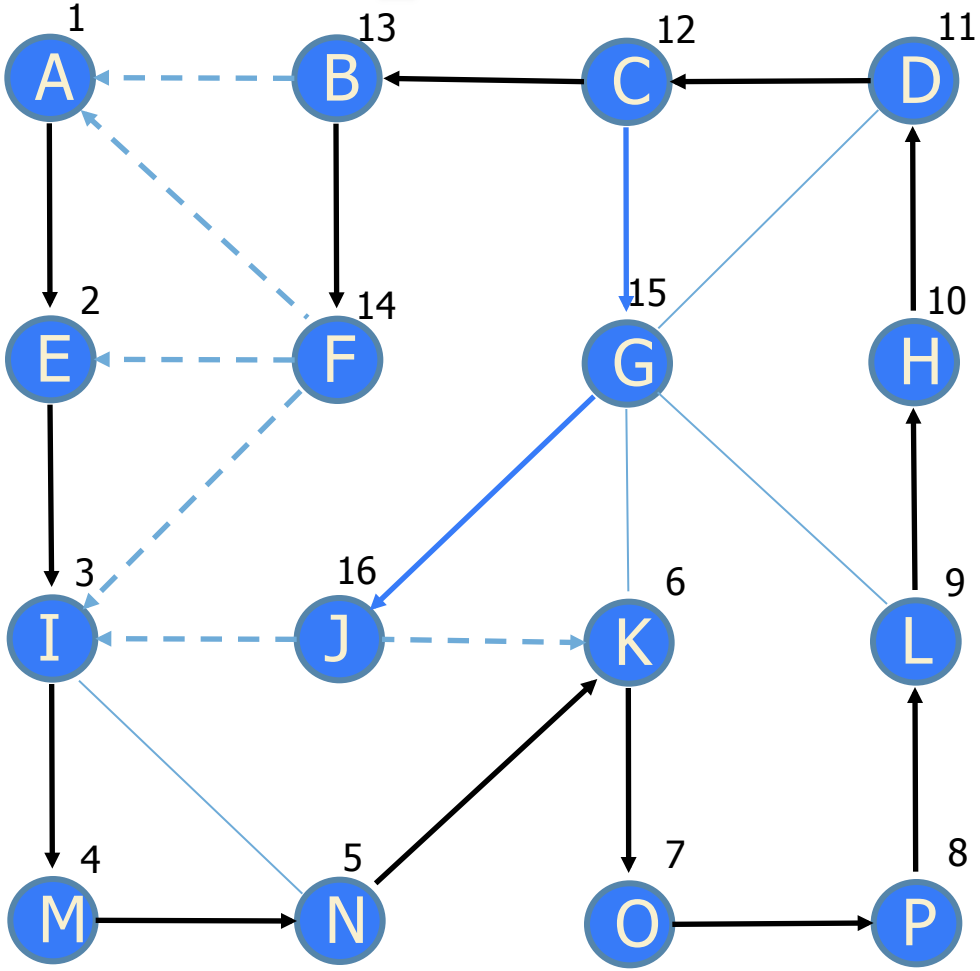
DFS - Depth First Search



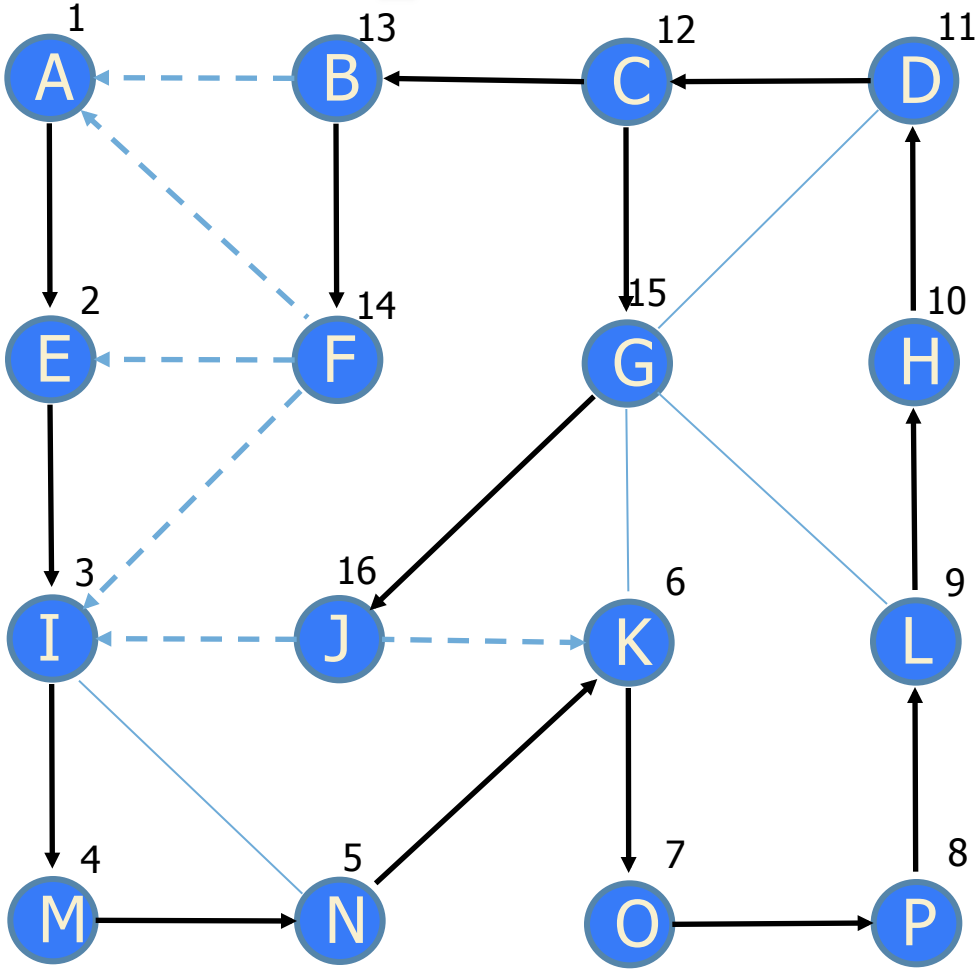
DFS - Depth First Search



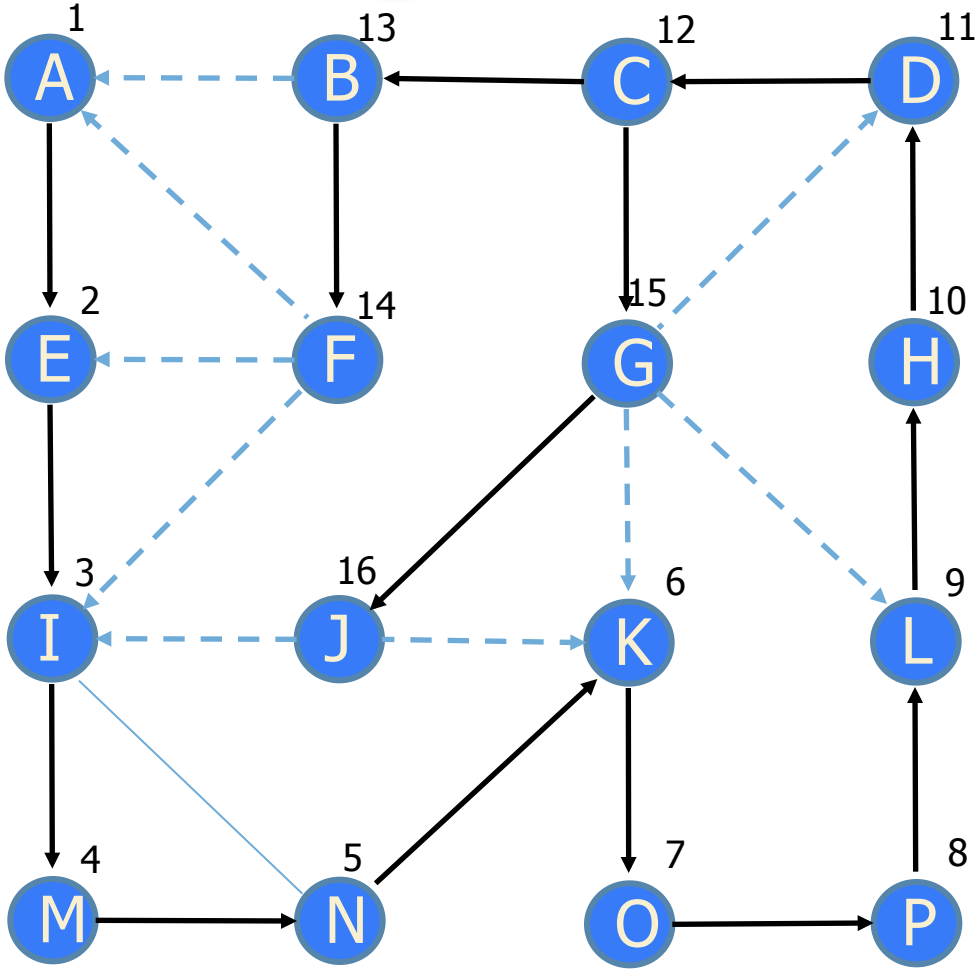
DFS - Depth First Search



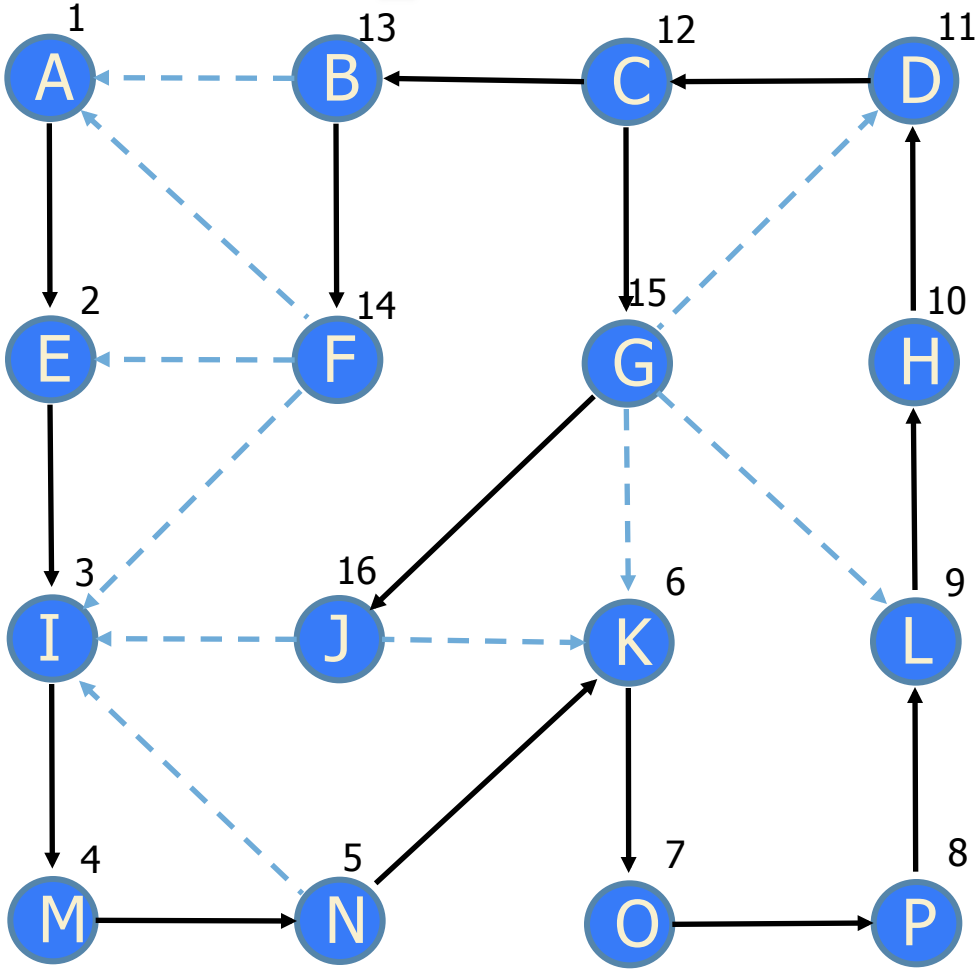
DFS - Depth First Search



DFS - Depth First Search



DFS - Depth First Search

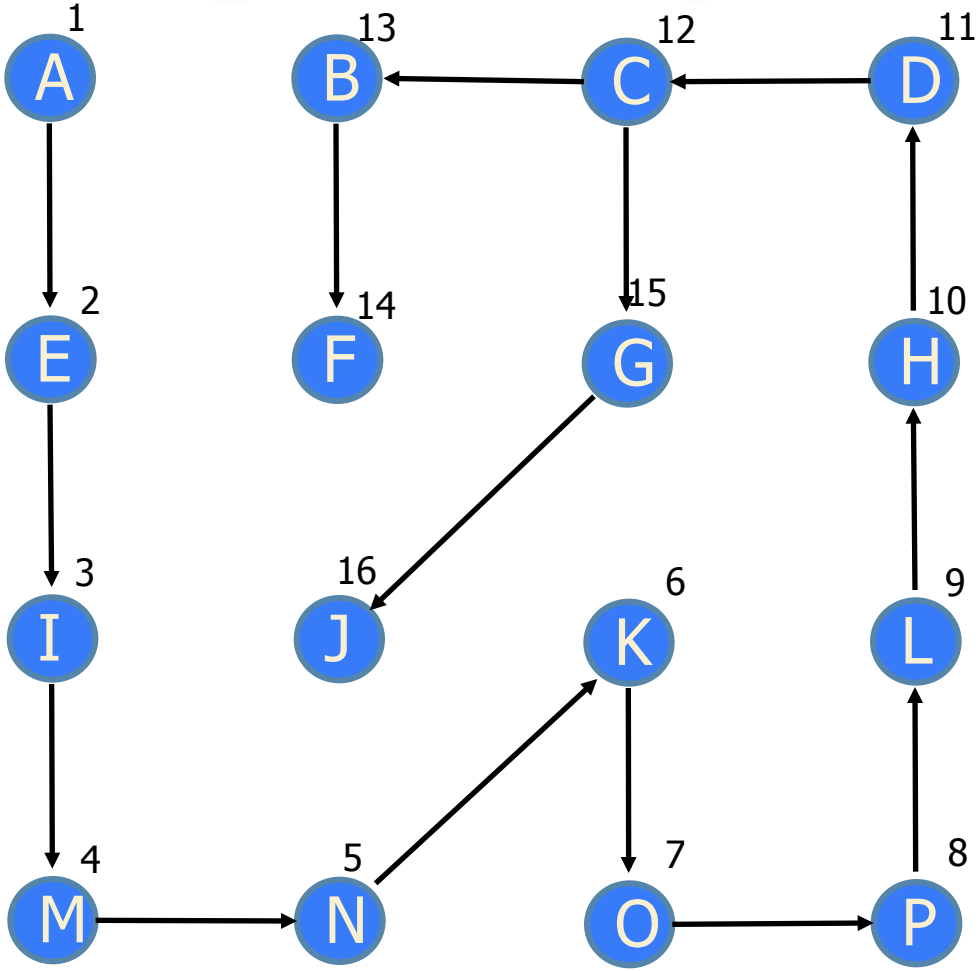


Running Time = $O(|E| + |V|)$

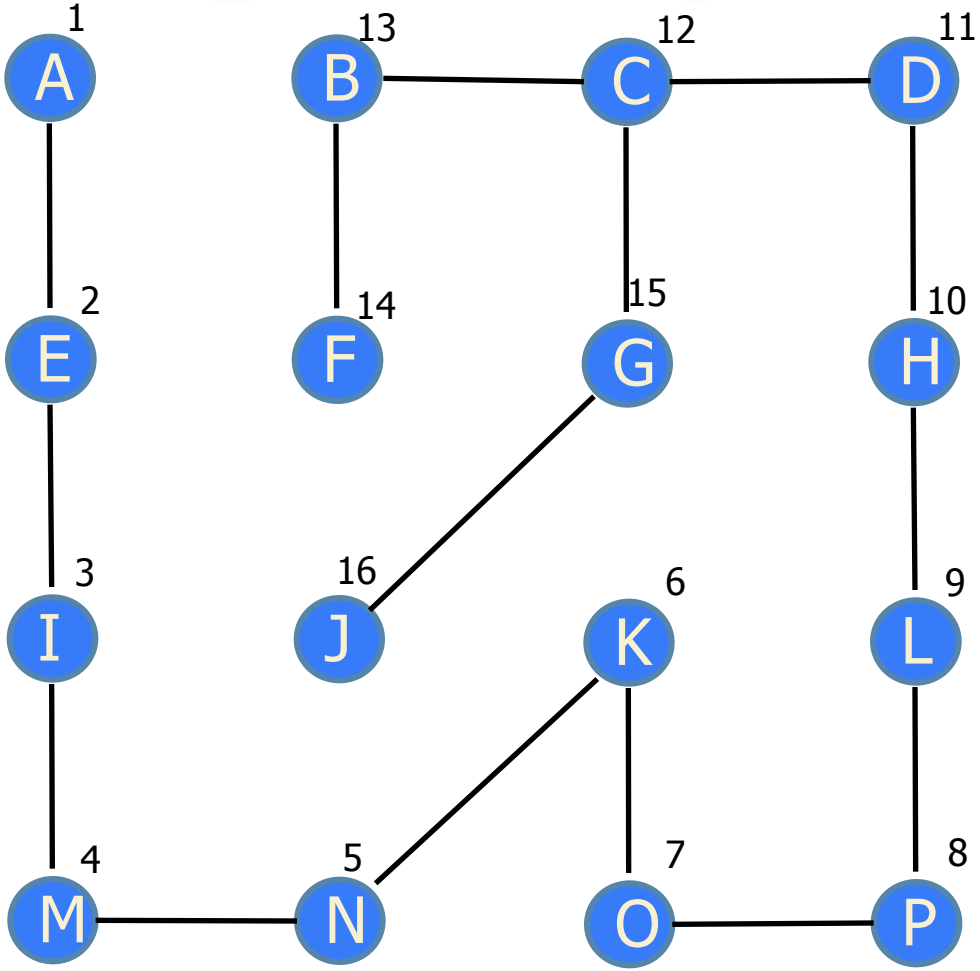
```
DFS(G)
    for each u ∈ G.V
        u.color = WHITE
    time = 0
    for each u ∈ G.V
        if u.color == WHITE
            DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                          // discover u
  for each v ∈ G.Adj[u]                   // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                              // finish u
```

Spanning Tree



Spanning Tree



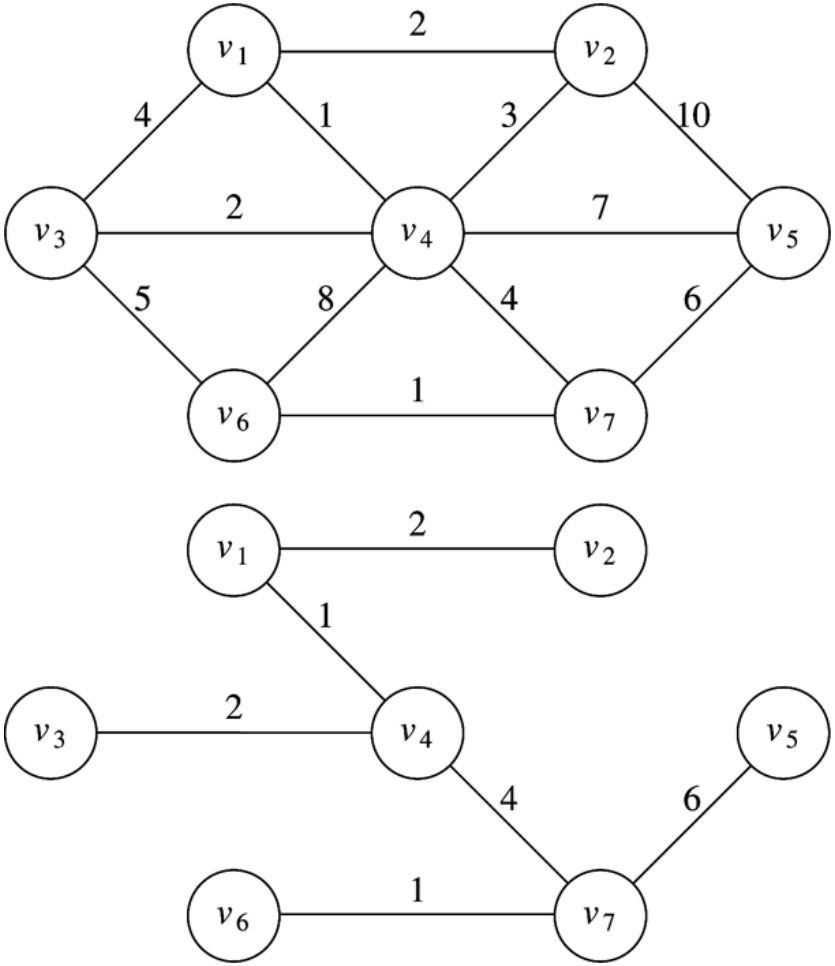
MST – Minimum Spanning Tree

- 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.

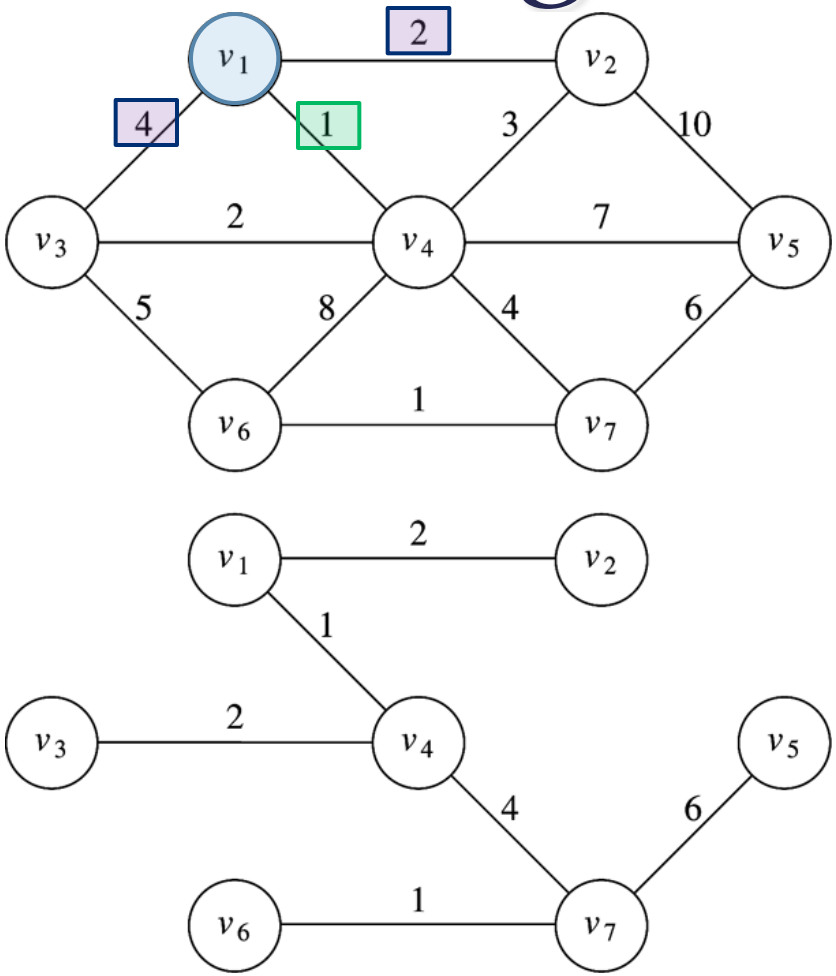
MST

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

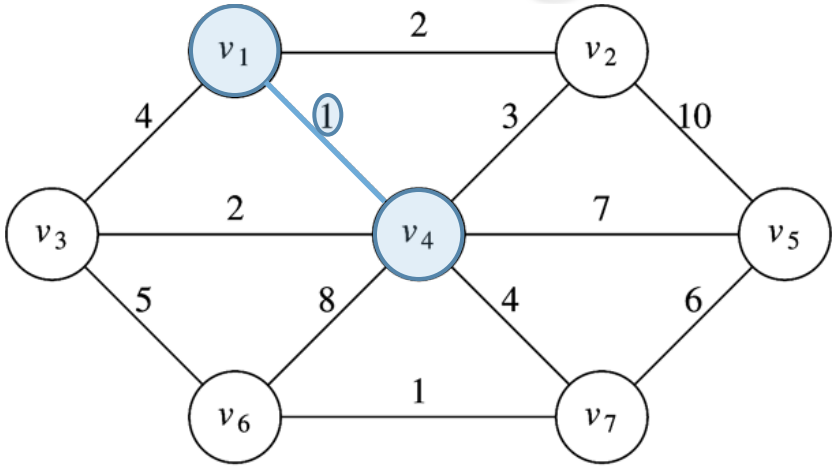
Prim's Algorithm

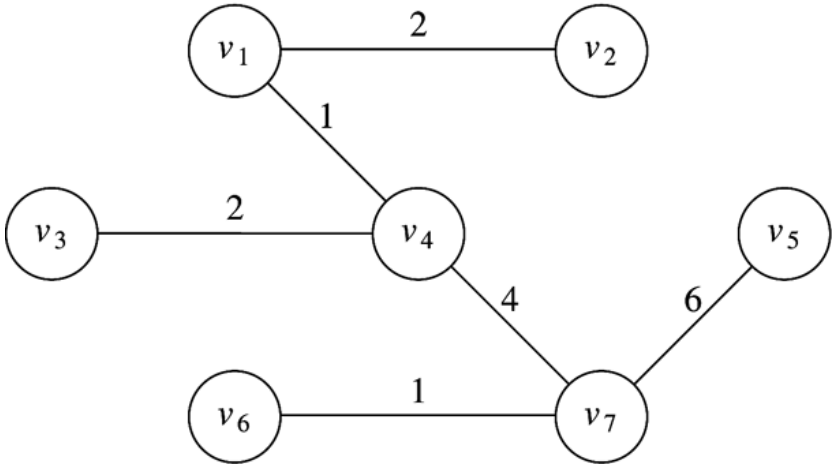


Prim's Algorithm

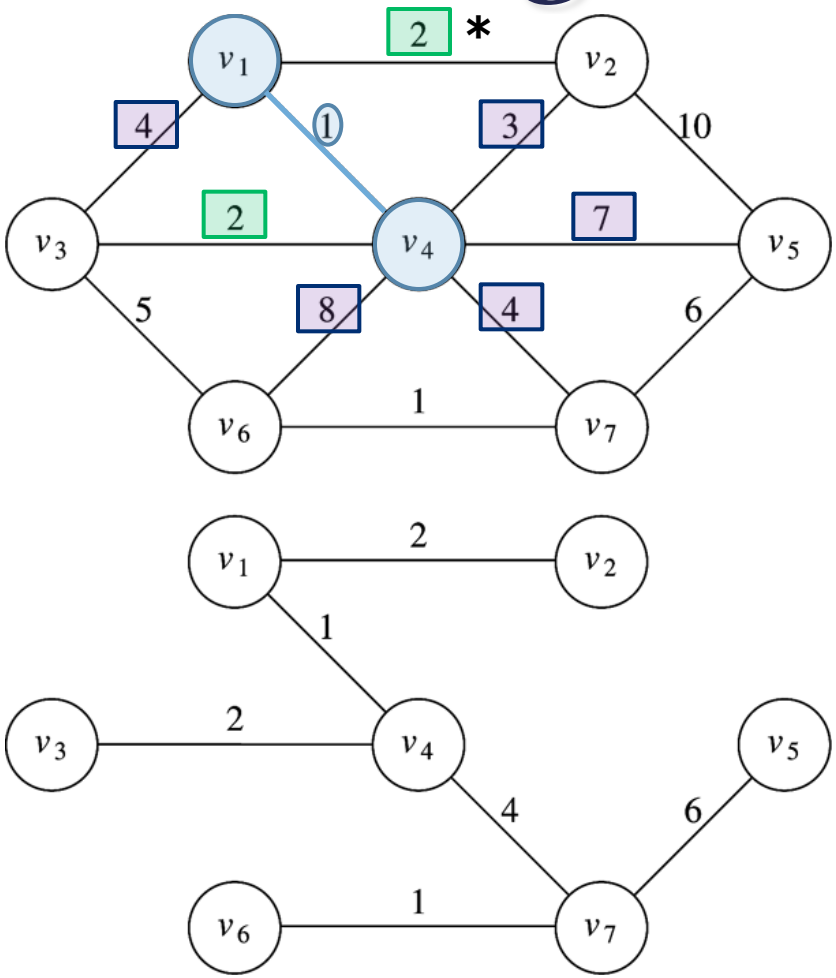


Prim's Algorithm

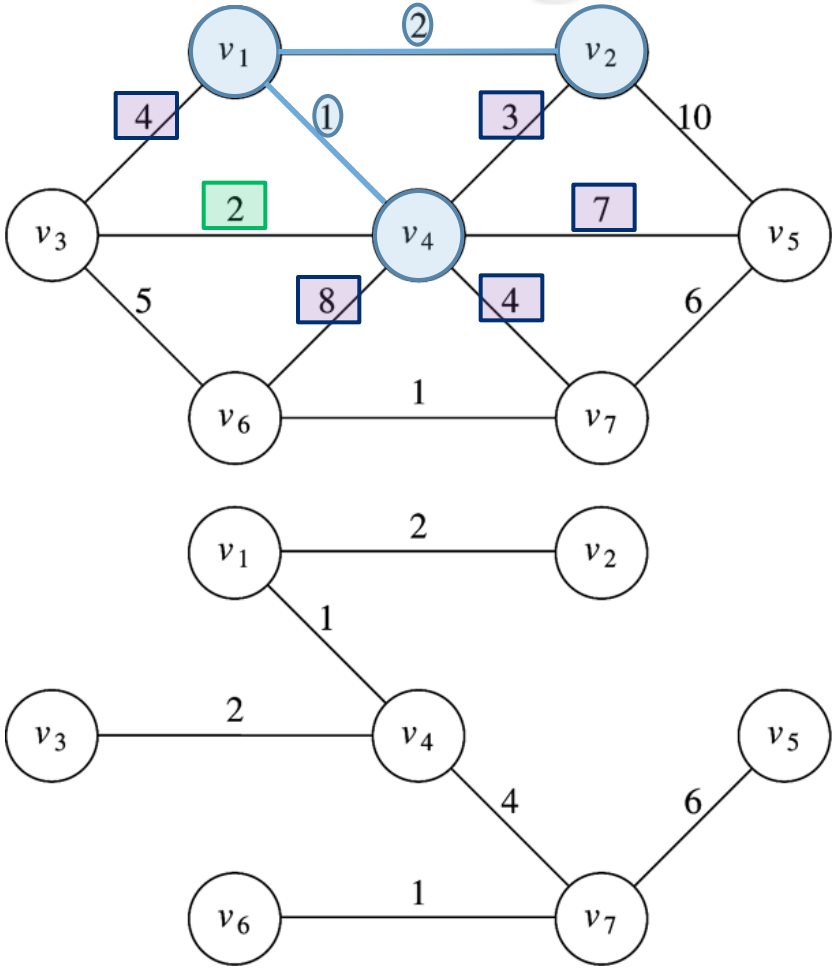




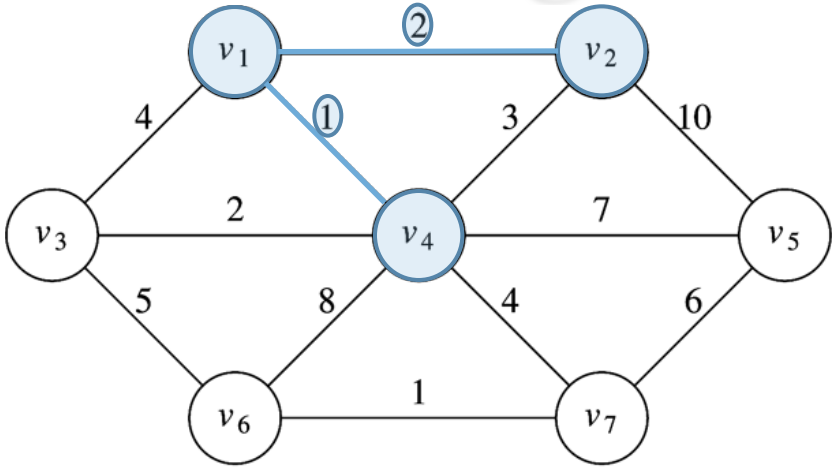
Prim's Algorithm

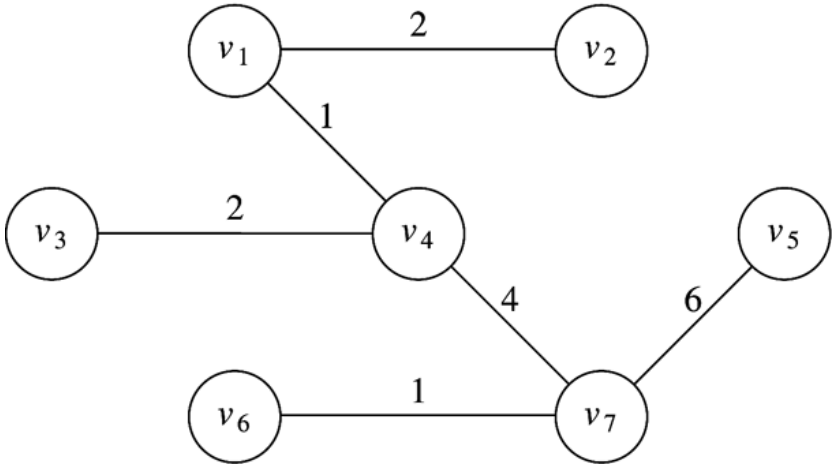


Prim's Algorithm

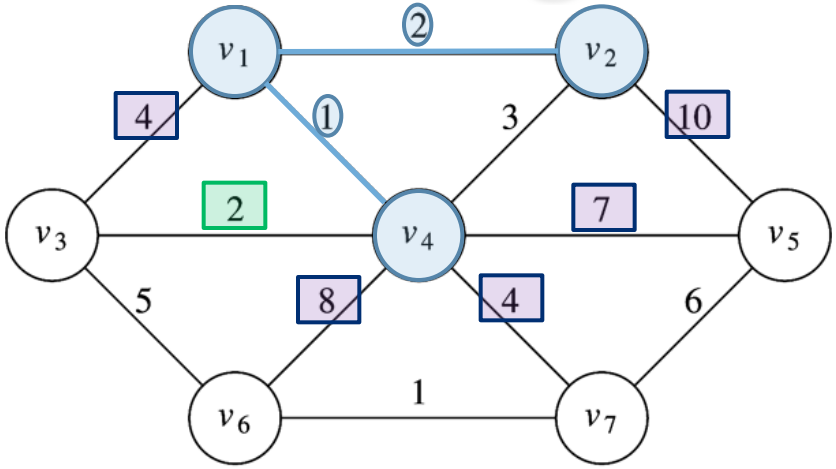


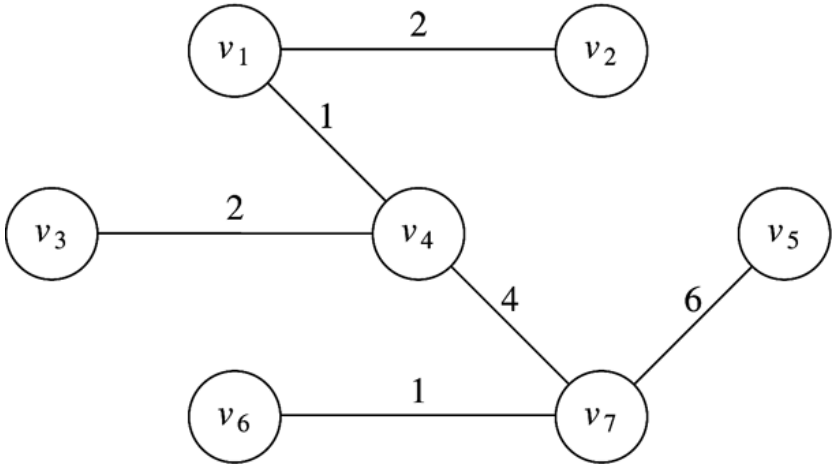
Prim's Algorithm



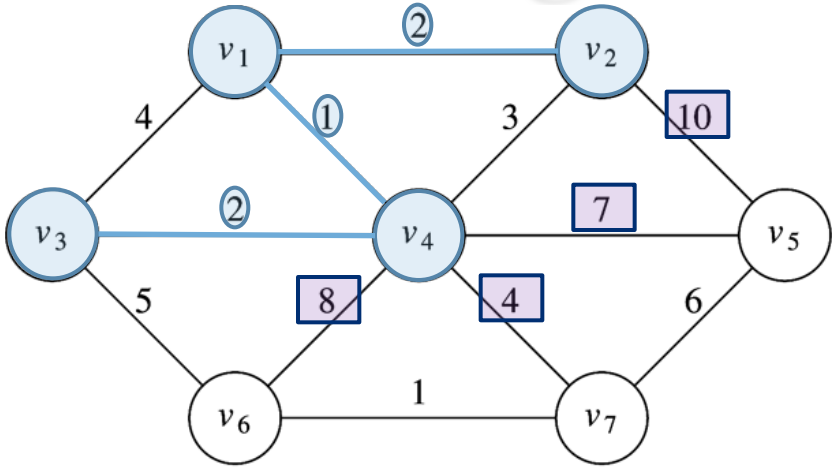


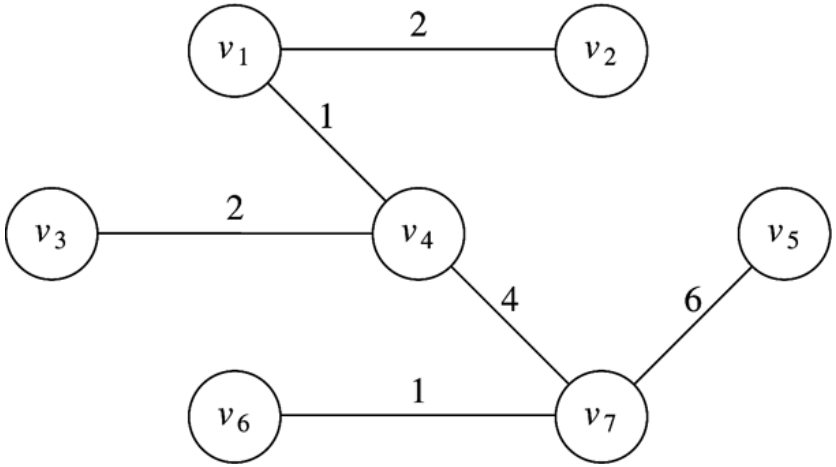
Prim's Algorithm



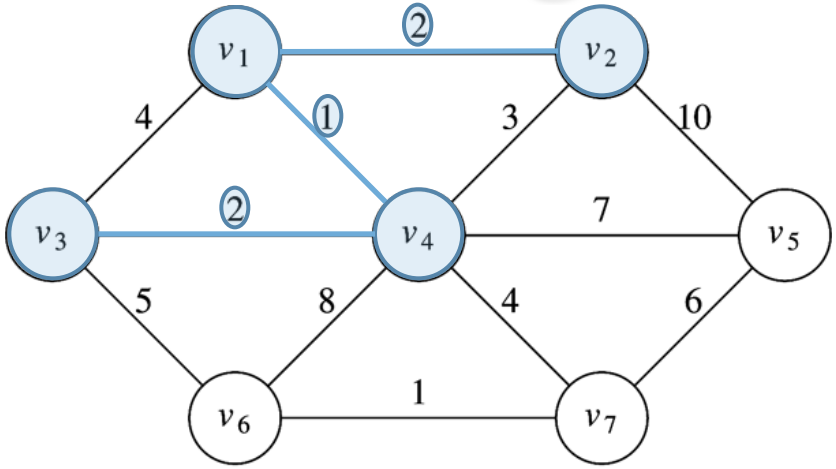


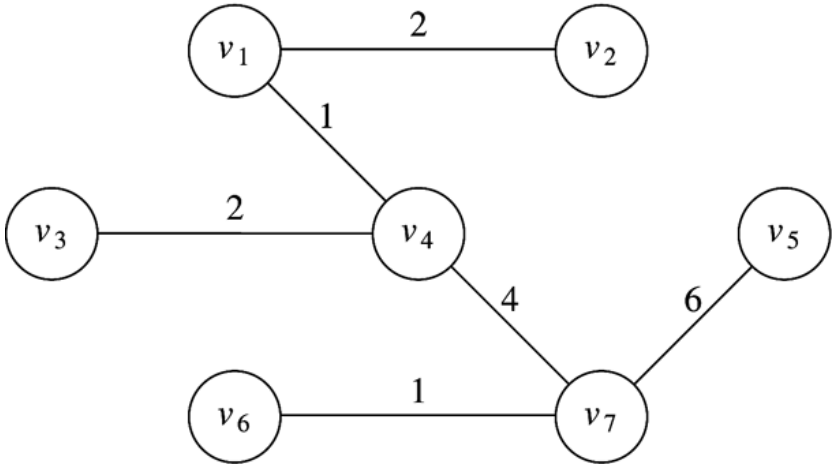
Prim's Algorithm



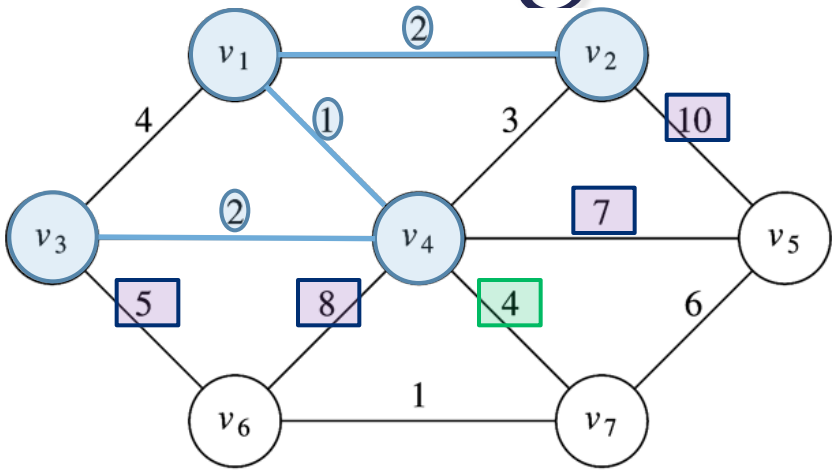


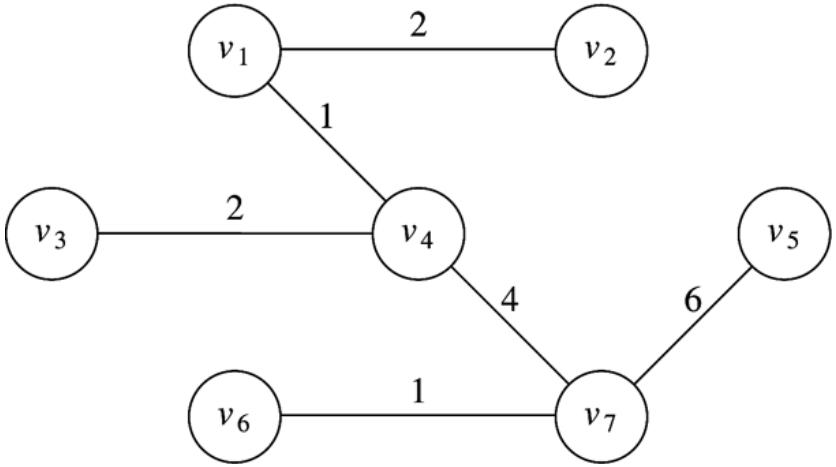
Prim's Algorithm



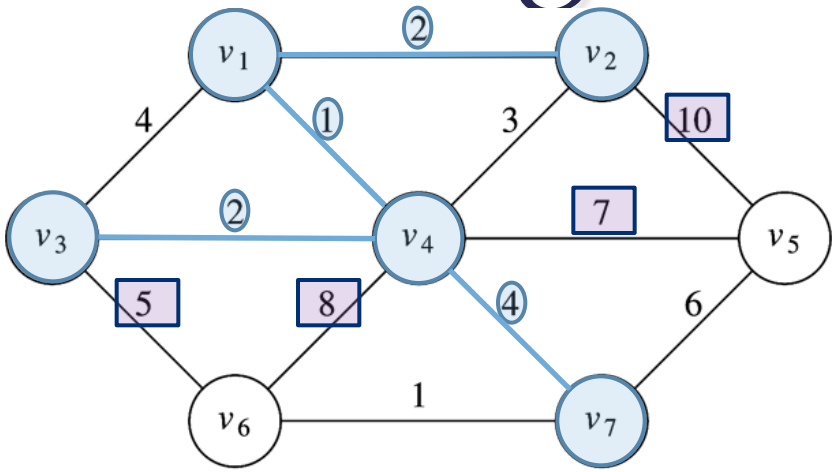


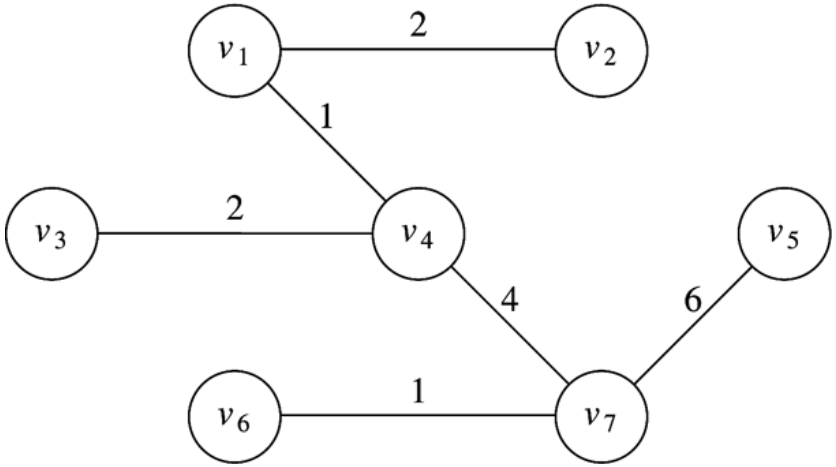
Prim's Algorithm



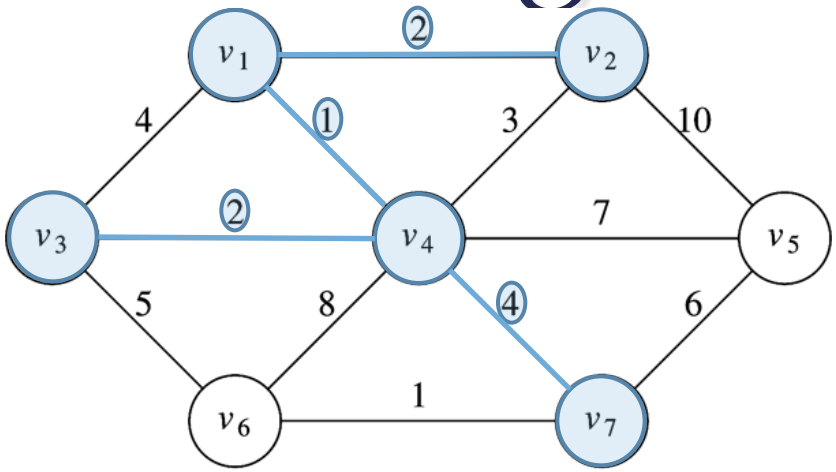


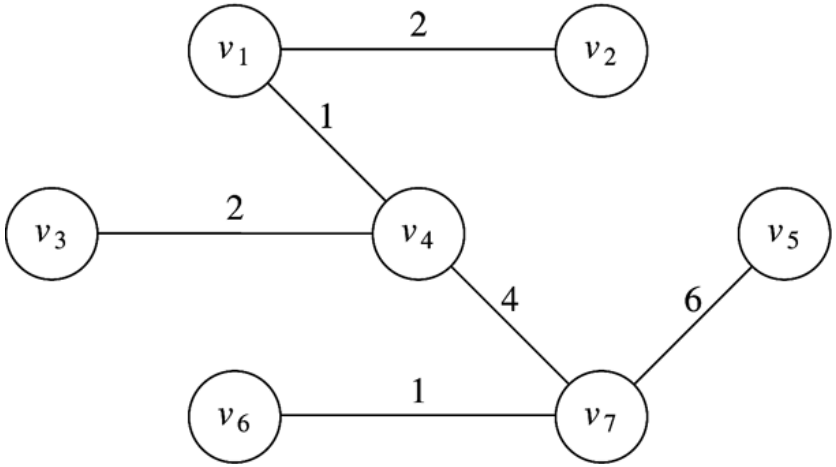
Prim's Algorithm



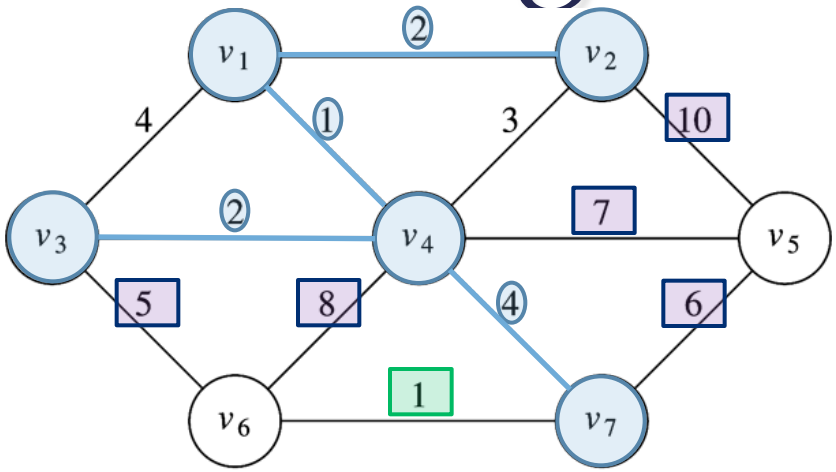


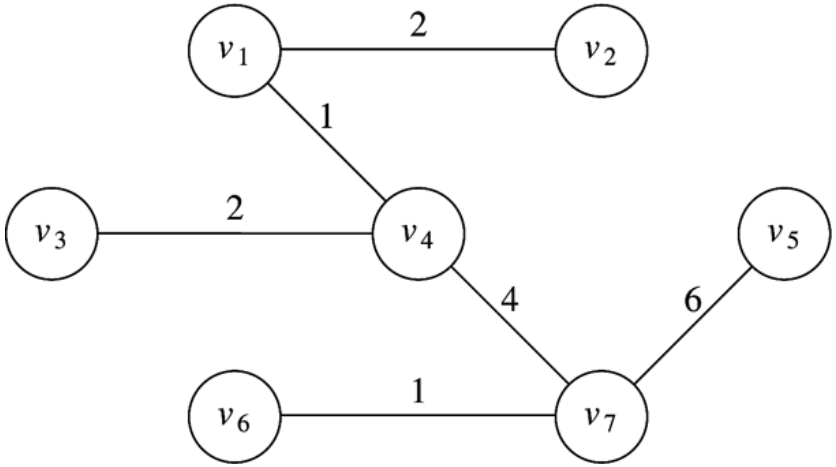
Prim's Algorithm



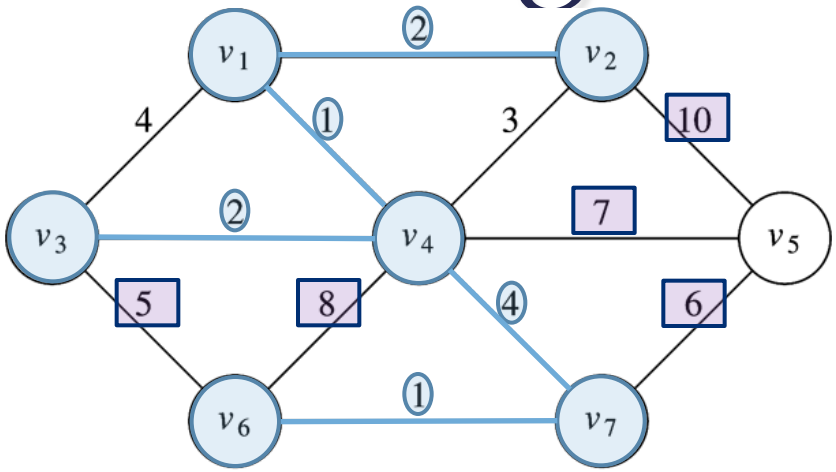


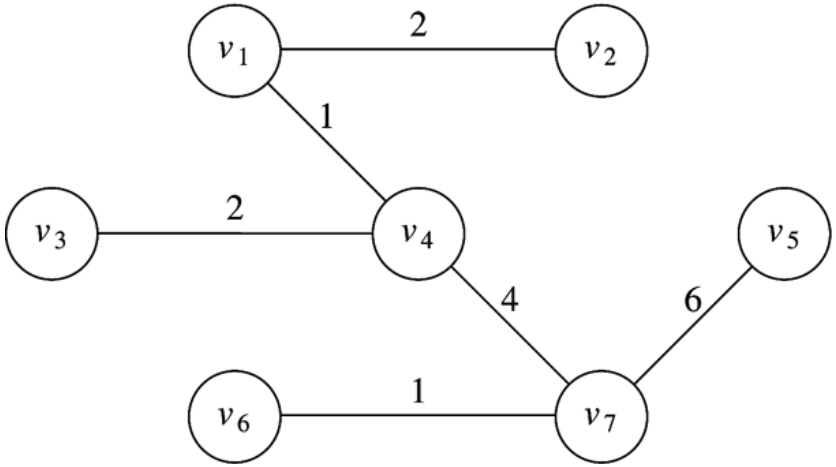
Prim's Algorithm



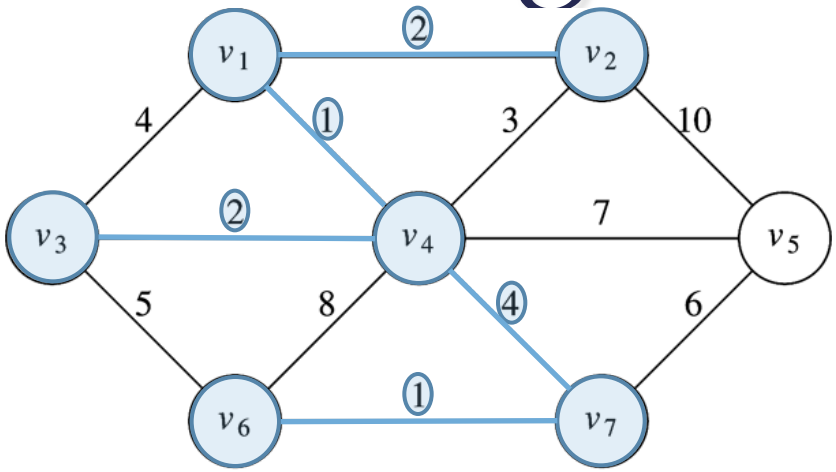


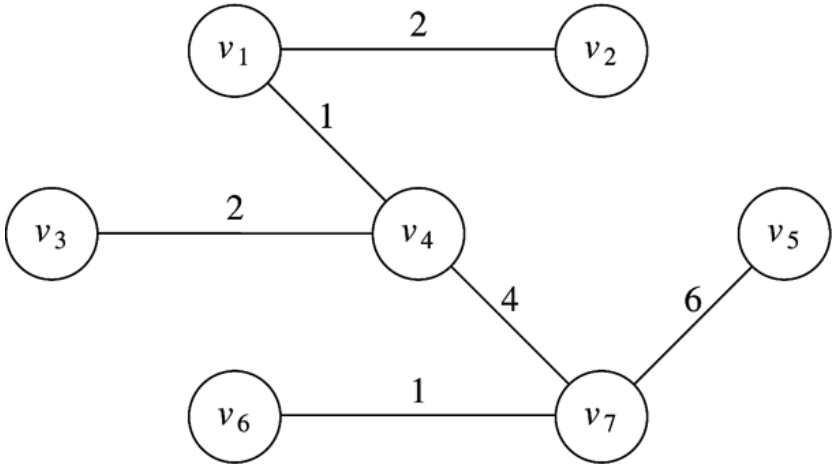
Prim's Algorithm



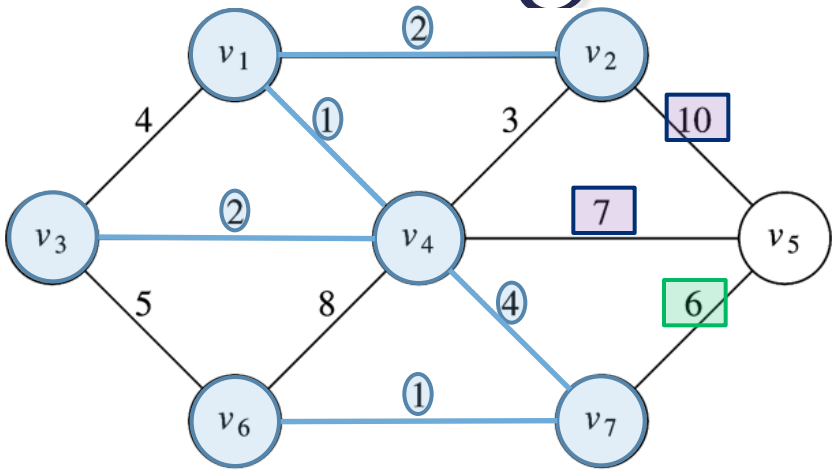


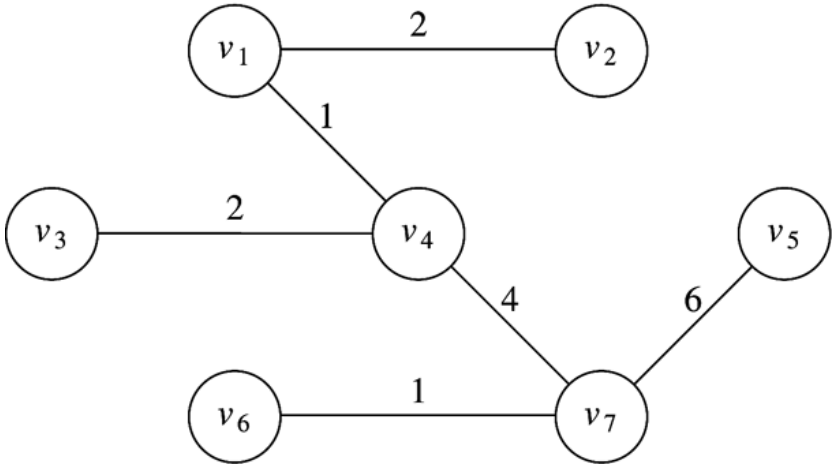
Prim's Algorithm



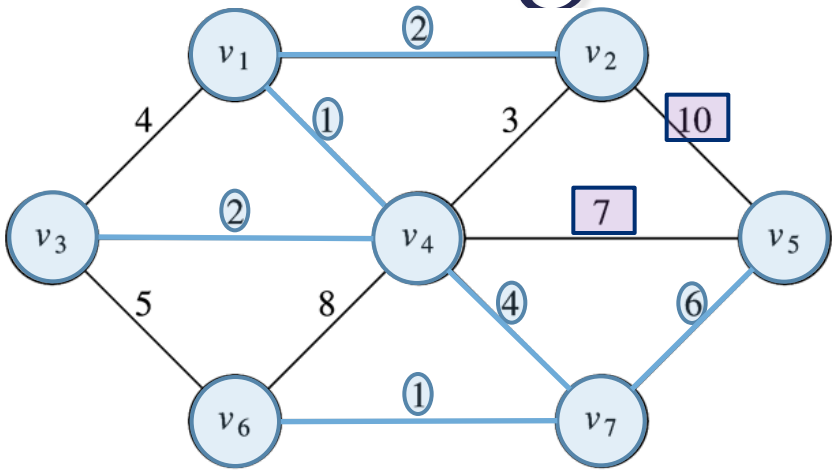


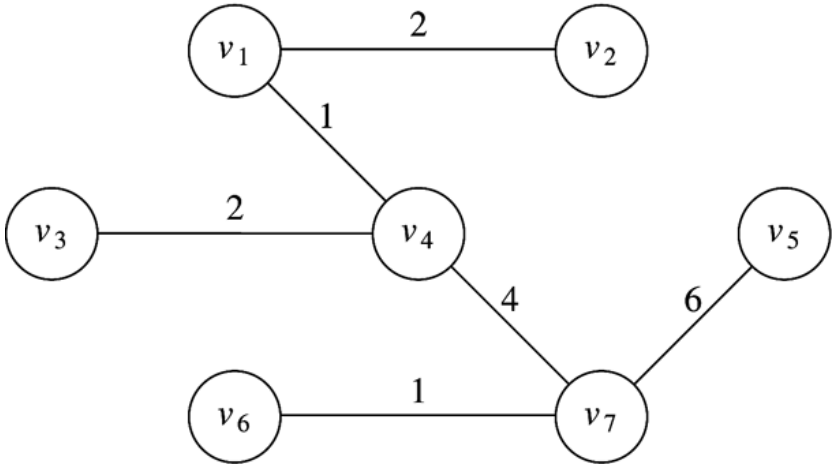
Prim's Algorithm





Prim's Algorithm





Prim's Algorithm