

# 15CSE201 : Data Structures and Algorithms

## Lecture 14 :Graphs

Ritwik M

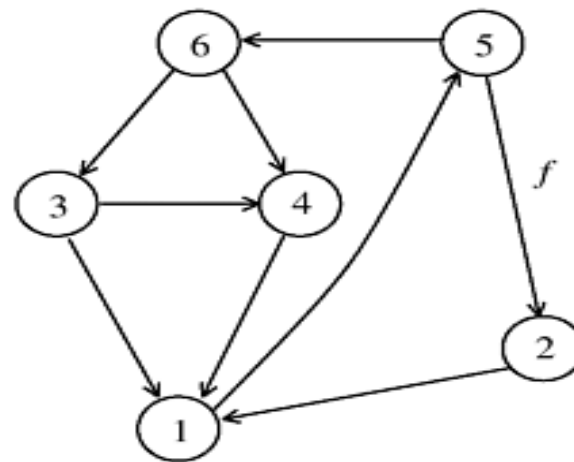
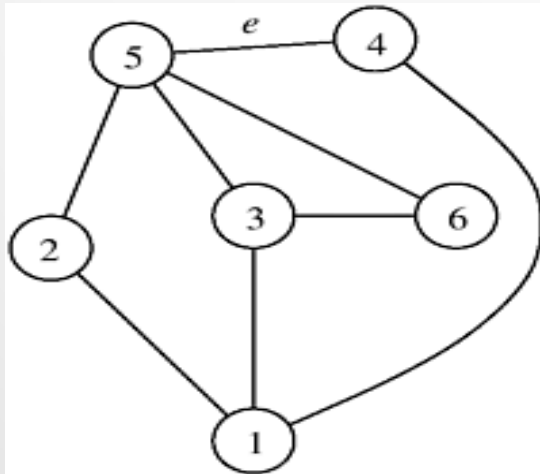
Based on the reference materials by Prof. Goodrich and Dr. Vidhya Balasubramanian

# Recap

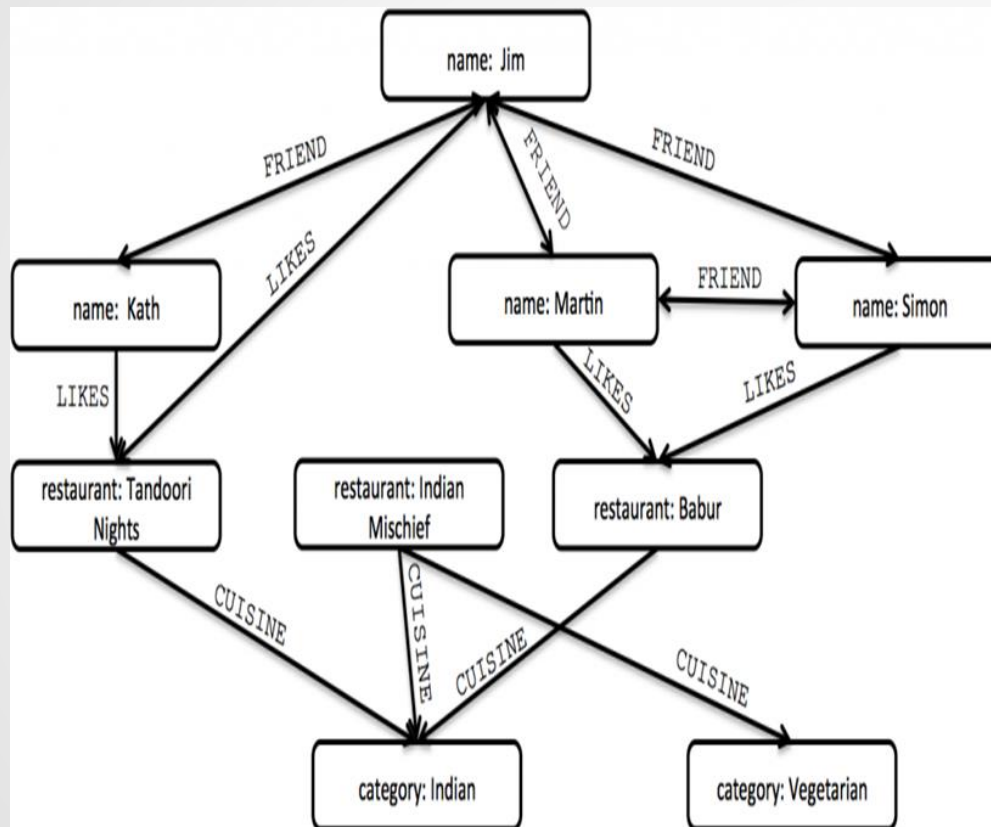
- Linear Data Structures
  - Stack, Queue, Vectors, Linked List
- Non-Linear Data Structures
  - Trees
    - Binary Trees, BST, Height balanced trees, multi-way search trees
    - Heaps

# Graphs

- A graph  $G = (V, E)$  is a set of vertices  $V$ , and a collection of edges  $E$  which is a subset of  $V \times V$
- A way of representing connections between pairs of objects from some set  $V$ 
  - Edges can be either directed or undirected

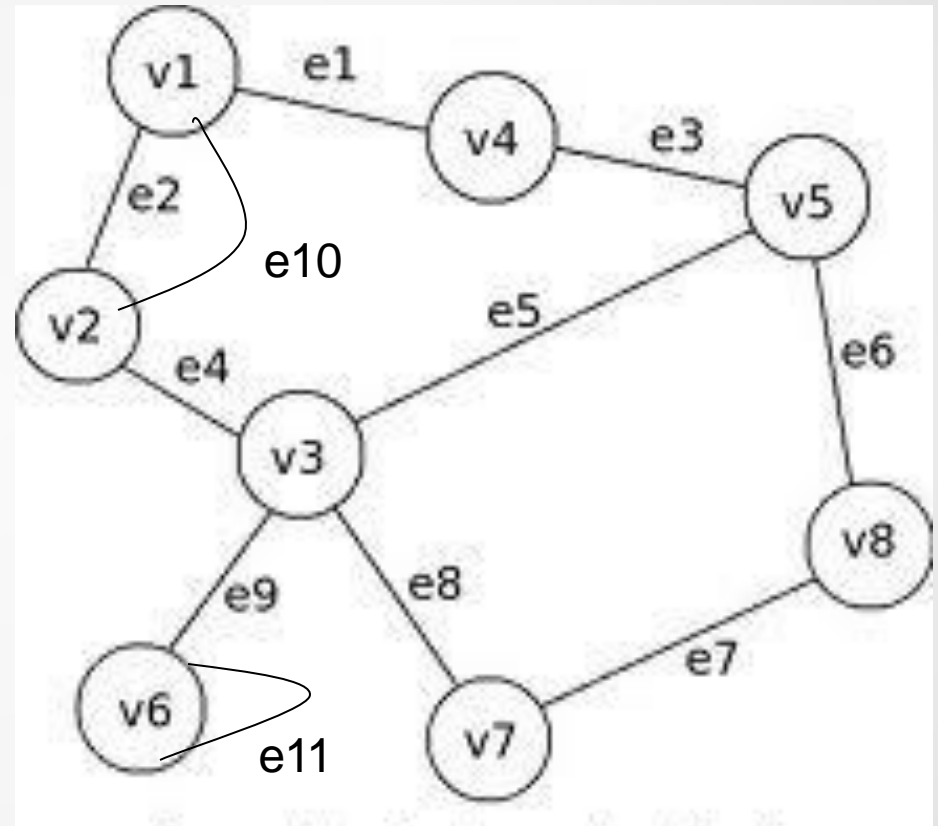


# Examples



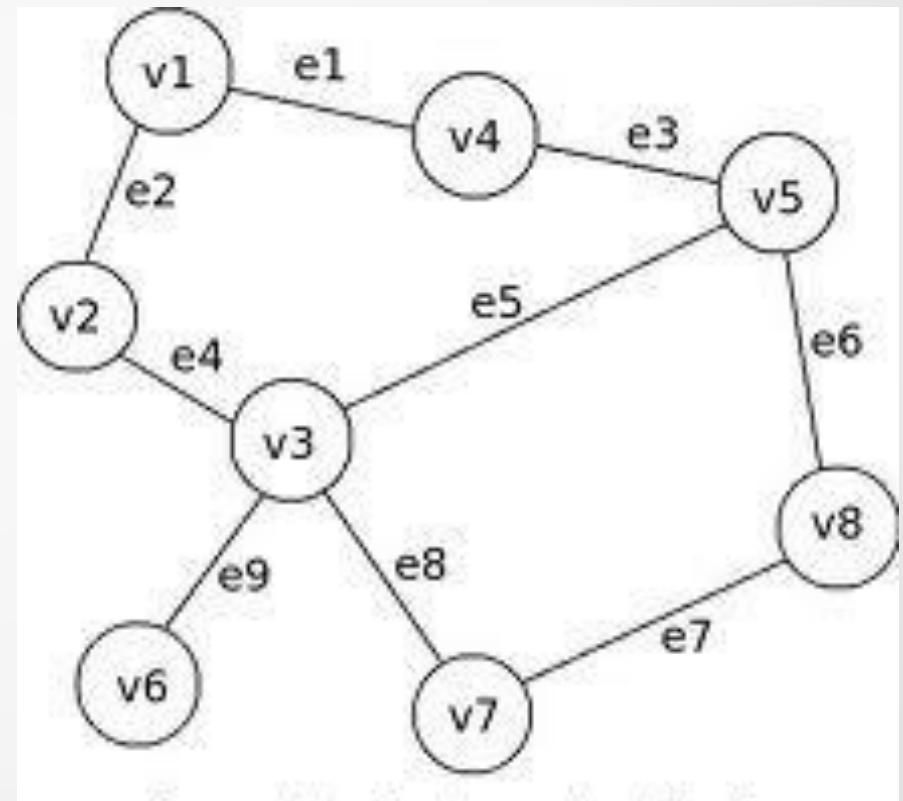
# Definitions

- Vertices (v)
- Edges (e)
- Adjacent vertices - v1, v2 adjacent
- Degree of vertex
  - is number of edges incident on it
    - $\deg(v3) = 4$
  - Here, vertices with loops are counted twice as the graph is undirected.
    - $\text{Deg}(v6) = 3$
- Parallel edges connect same set of vertices - e2 and e10
- Loop, edge connecting same vertex – e11



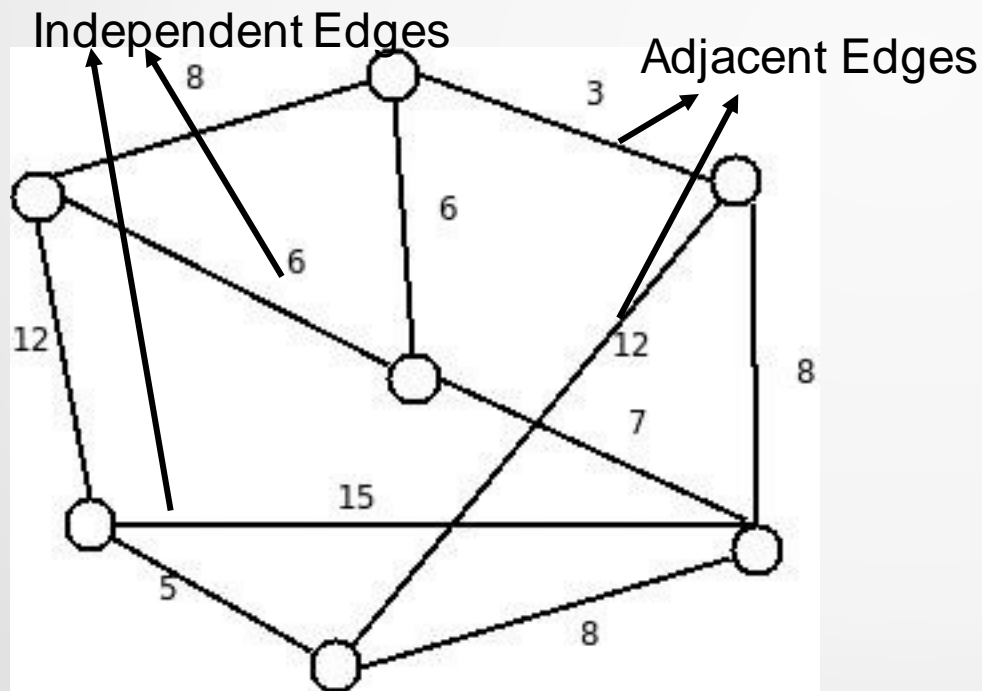
# Definitions Contd

- Path
  - A Sequence of alternating vertices and edges
  - E.g.:  $P1=(v1,v2,v3,v6)$
  - Simple path
    - Vertices and edges are distinct
      - e.g.  $P1$
    - $v1,v2,v3,v7,v3,v6$ , not simple
- Cycle
  - Path that ends at start node
  - $v2,v1,v4,v5,v3,v2$

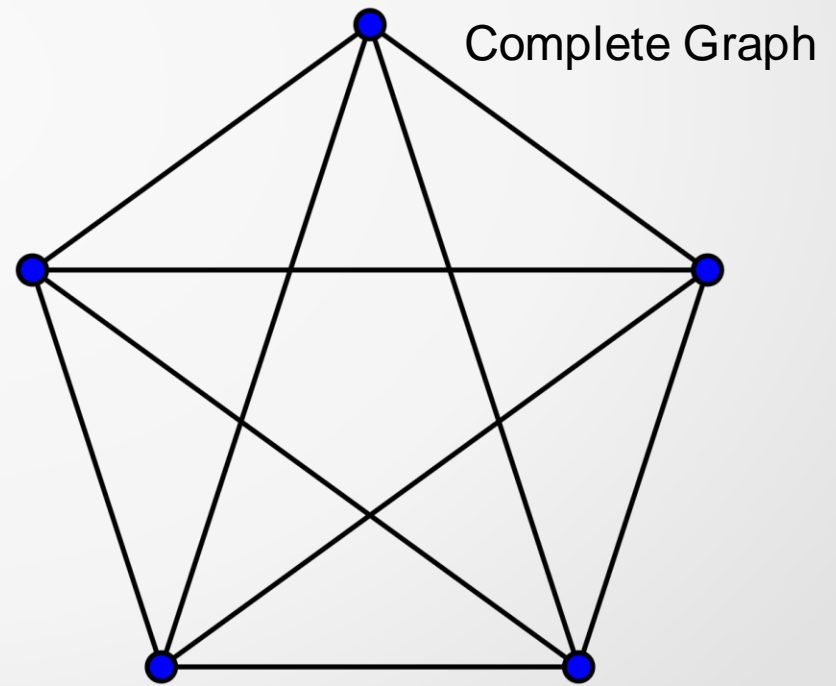


# Definitions Contd.

- Independent edges: don't have common vertices
- Complete graph: Every node adjacent to each other
- Simple Graph: Has no loops or parallel edges

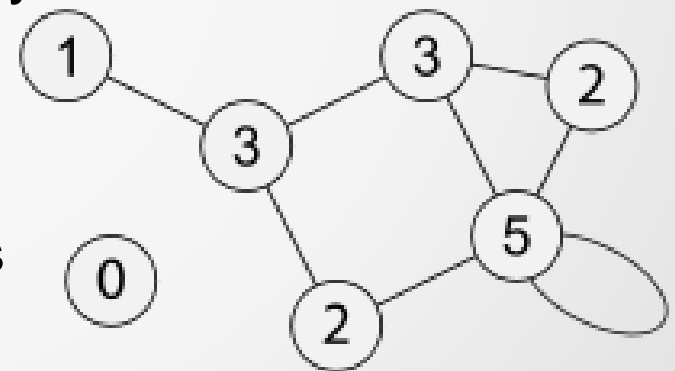


Simple Graph



# Definitions contd.

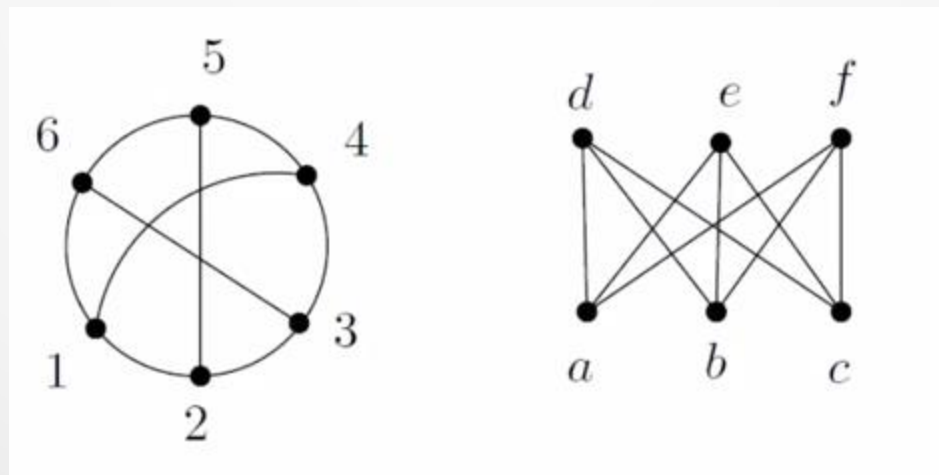
- Maximum degree of a Graph  $G$  denoted by  $\Delta(G)$ , and the minimum degree of a graph, denoted by  $\delta(G)$ , are the maximum and minimum degree of its vertices.
- Here  $\Delta(G) = 5$  and  $\delta(G) = 0$
- Number of vertices of odd degree is always even in a graph  $\rightarrow$  handshaking lemma
- In an undirected simple graph
  - $m \leq n(n-1)/2$ , where  $n$  is number of vertices
  - Proof: each vertex has degree at most  $(n-1)$



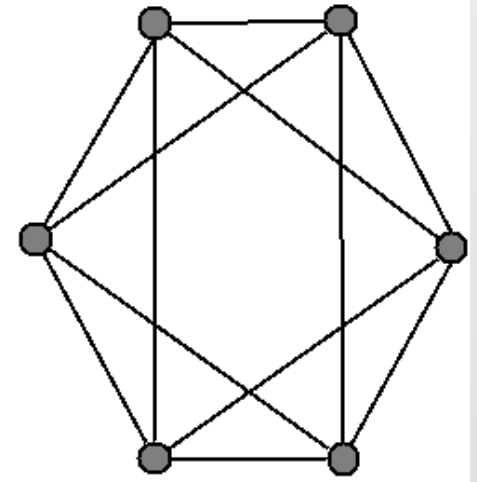
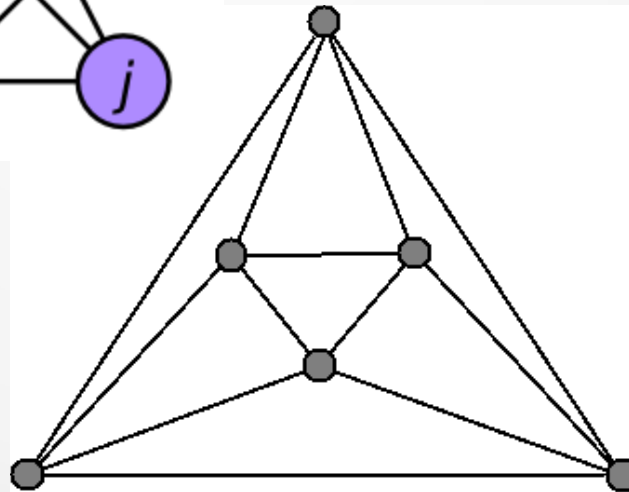
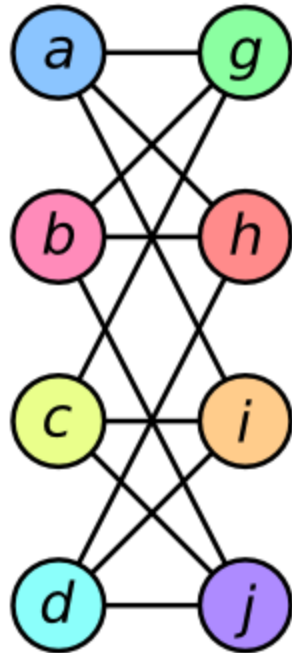
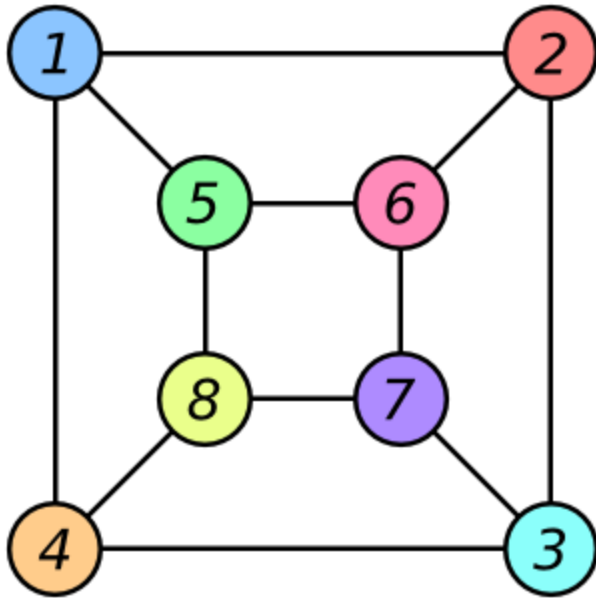


# Definitions contd.

- Planar Graphs: Graph whose edges intersect only at their end
- Graph Isomorphism
  - Two graphs  $G$  and  $H$  are isomorphic if
    - They have same number of vertices
    - A pair of nodes are adjacent in  $G$  iff a corresponding pair exists in  $H$



# Example of Isomorphism



# Graph ADT

- Accessor methods

- isVertex()
- incidentEdges(v)
- endVertices(e)
- isDirected(e)
- origin(e)
- destination(e)
- opposite(v, e)
- areAdjacent(v, w)

- Update Methods

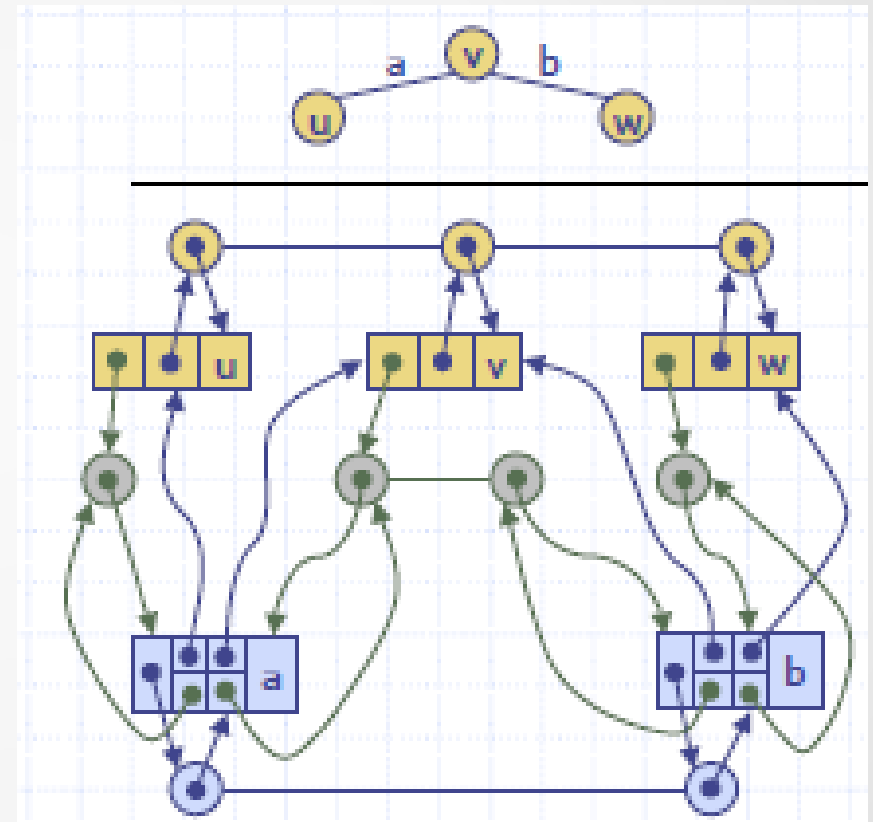
- insertVertex(o)
- insertEdge(v, w, o)
- insertDirectedEdge(v, w, o)
- removeVertex(v)
- removeEdge(e)

- Generic Methods

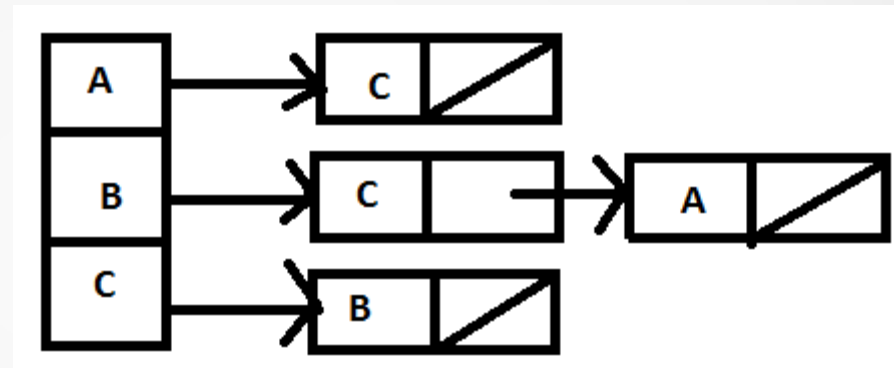
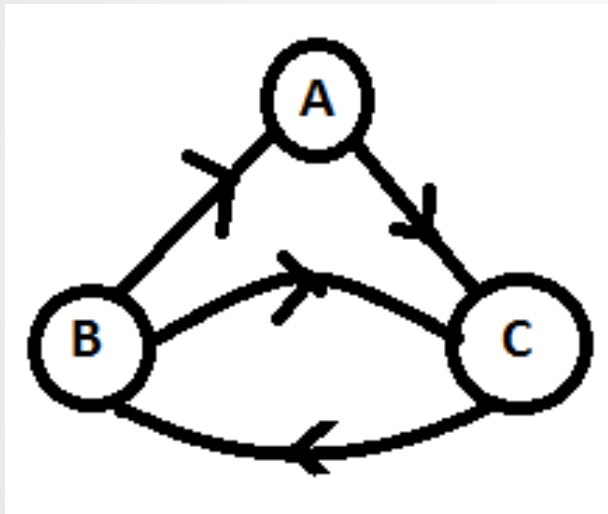
- numVertices()
- numEdges()
- vertices()
- edges()

# Graph Representations

- Adjacency list
  - Each vertex has incidence container
    - List of vertices incident on  $v$
  - Edge list
- Augmented list objects
  - references to associated positions in incidence sequences of end vertices

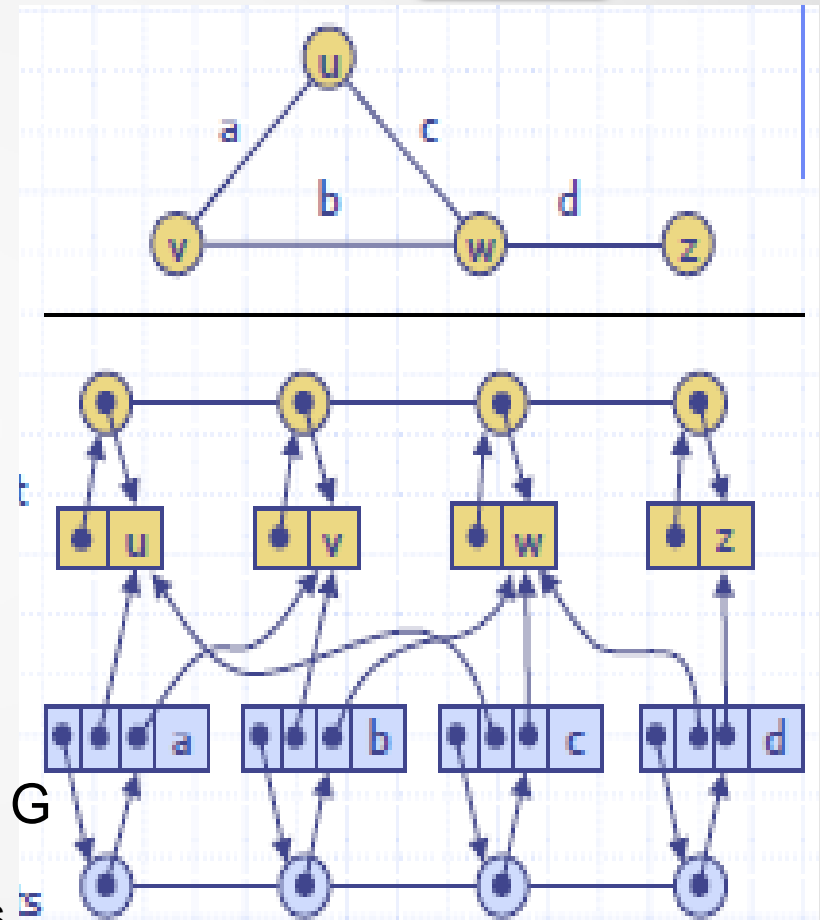


# Adjacency List Example



# Edge List Representation

- Vertex object
  - Element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex
  - destination vertex
  - reference to position in edge sequence
- Vertex sequence - Sequence of vertices in  $G$
- Edge sequence - Sequence of edge objects



# Adjacency Matrix

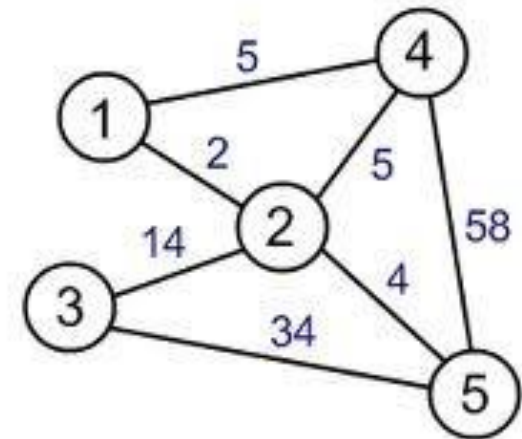
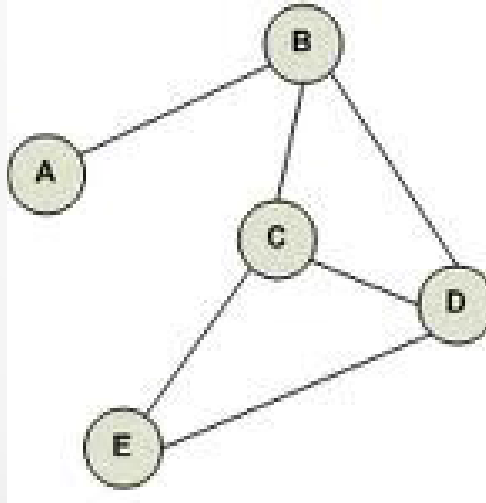
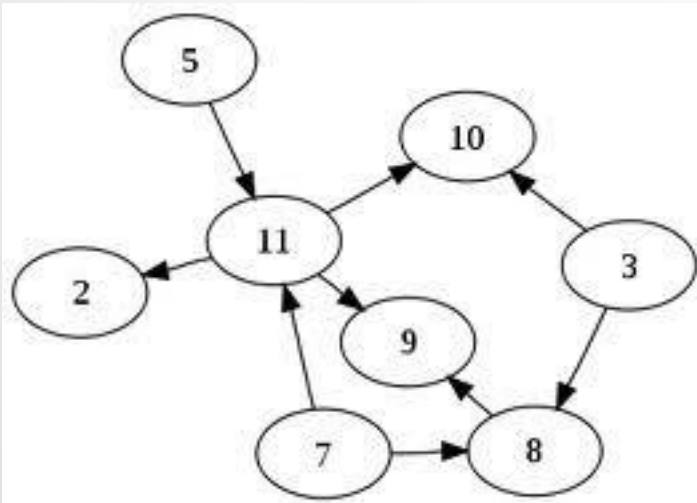
- Edge list
- 2D-array adjacency array  $A$  where
  - $A[i,j]$  is 1 if there is an edge between vertices  $i$  and  $j$
  - $A[i,j] = 0$ , if there is no edge incident on  $i$  and  $j$



	<b>u</b>	<b>v</b>	<b>w</b>
<b>u</b>	<b>0</b>	<b>1</b>	<b>0</b>
<b>v</b>	<b>1</b>	<b>0</b>	<b>1</b>
<b>w</b>	<b>0</b>	<b>1</b>	<b>0</b>

# Exercises

- Represent the following graphs using the three representations
- Suppose we represent a graph  $G$  with  $n$  vertices and  $m$  edges with edge list structure. Why does the function `insertVertex()` function take  $O(1)$  time, while `removeVertex()` function runs in  $O(n)$  time

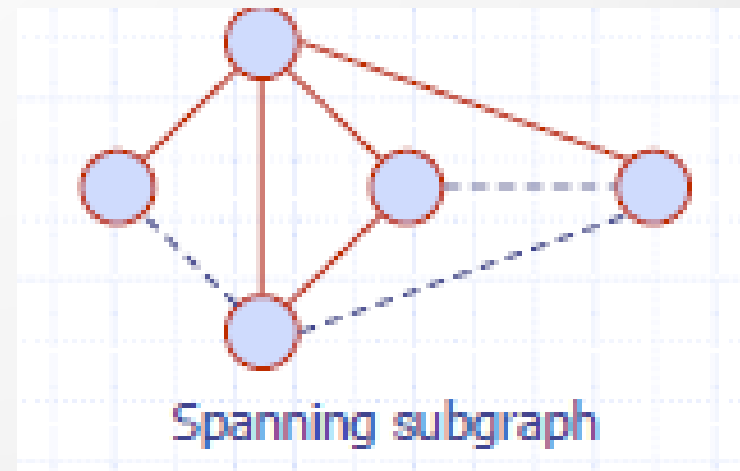
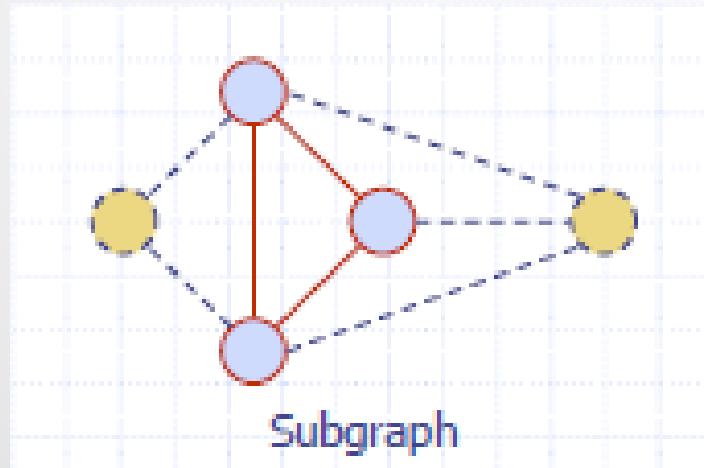




- Would you use the adjacency list structure or the adjacency matrix structure for the following cases? Justify.
  - The graph has 10,000 vertices and 20,000 edges and it is important to use as little space as possible
  - The graph has 10,000 vertices and 20,000,000 edges and it is important to use as little space as possible
  - You need to answer the query  $\text{areAdjacent}(v,w)$  as fast as possible, no matter how much space you use

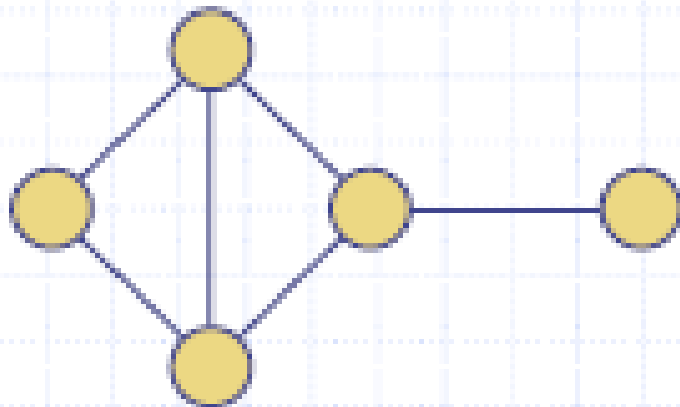
# Subgraphs

- 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  $G$

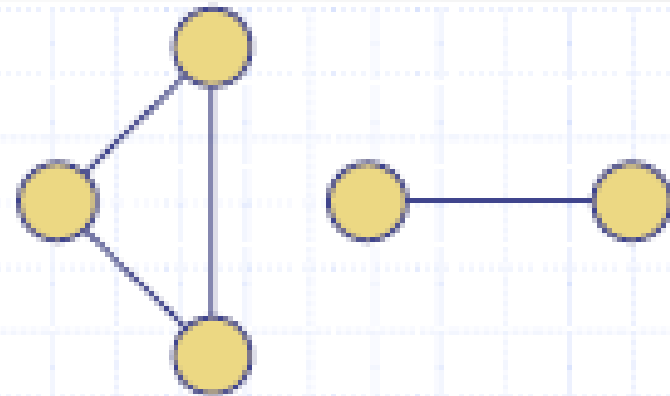


# Connectivity

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



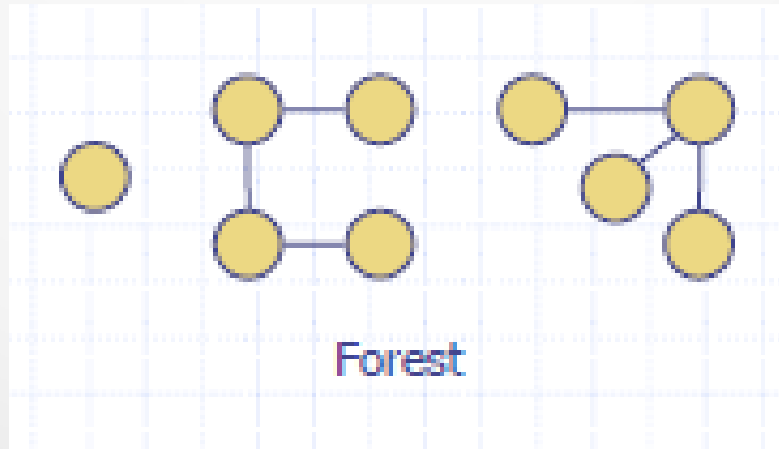
Connected graph



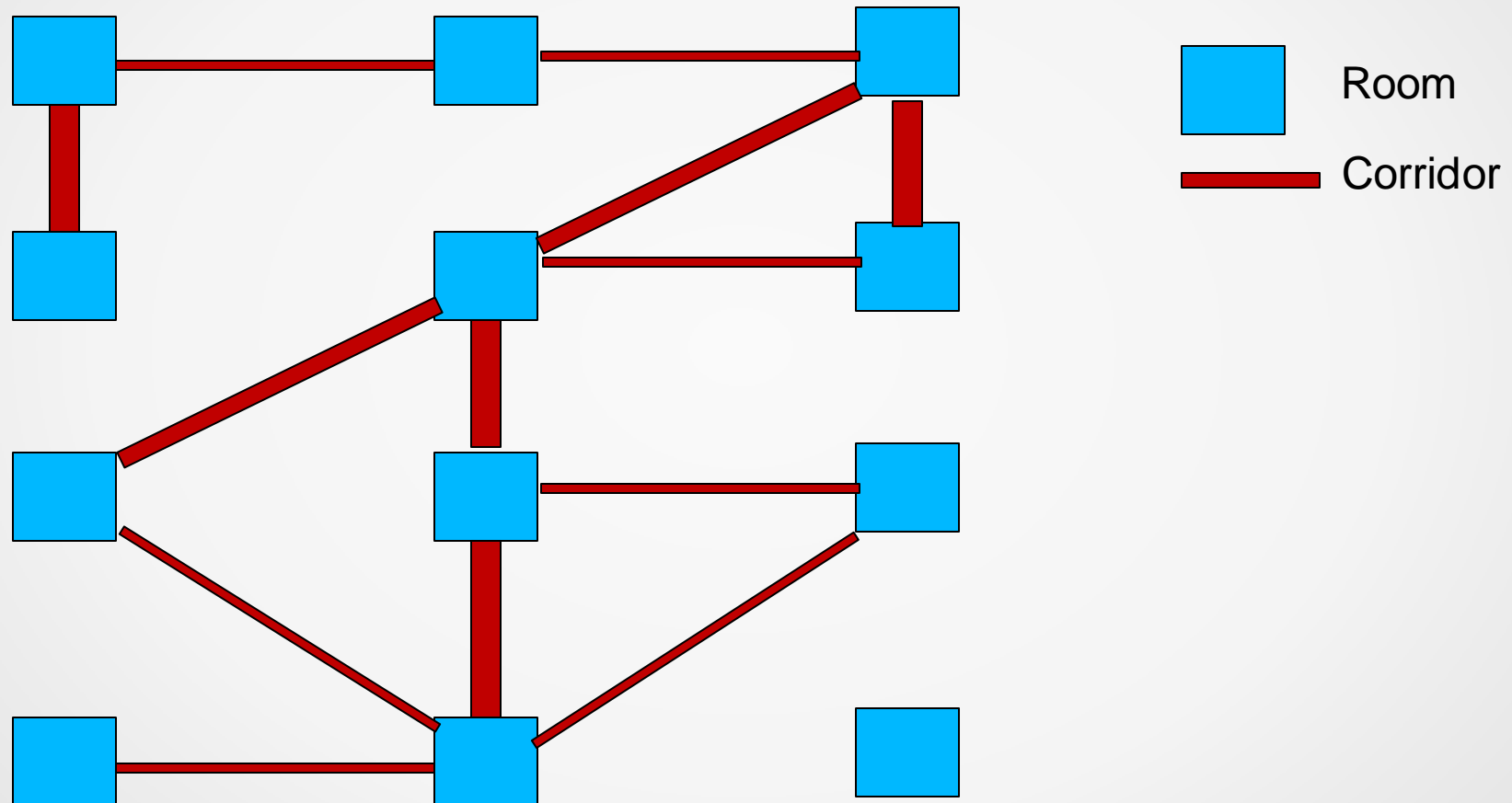
Non connected graph with two connected components

# Trees and Forests

- A tree is an undirected graph such that
  - T is connected
  - T has no cycles
  - Need not be rooted
- Forest is an undirected graph without cycles
  - Connected component of a forest are trees



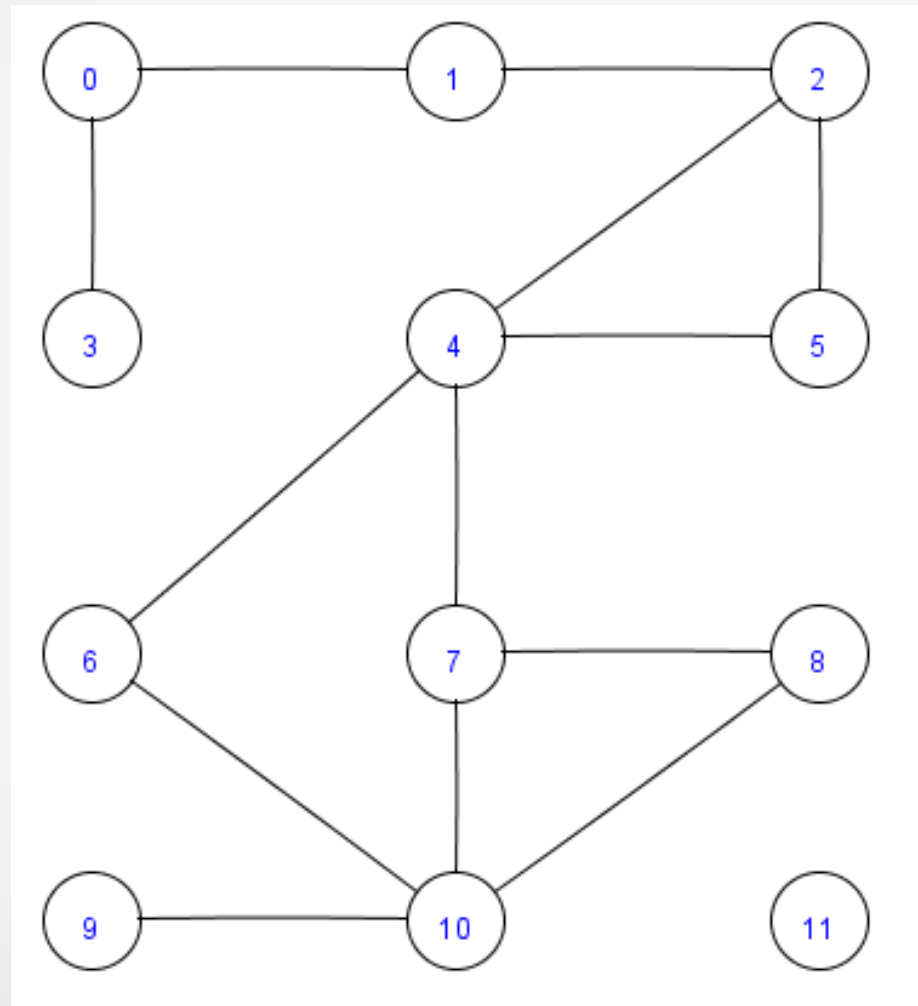
# Challenge: Try to visit all rooms without forming a cycle



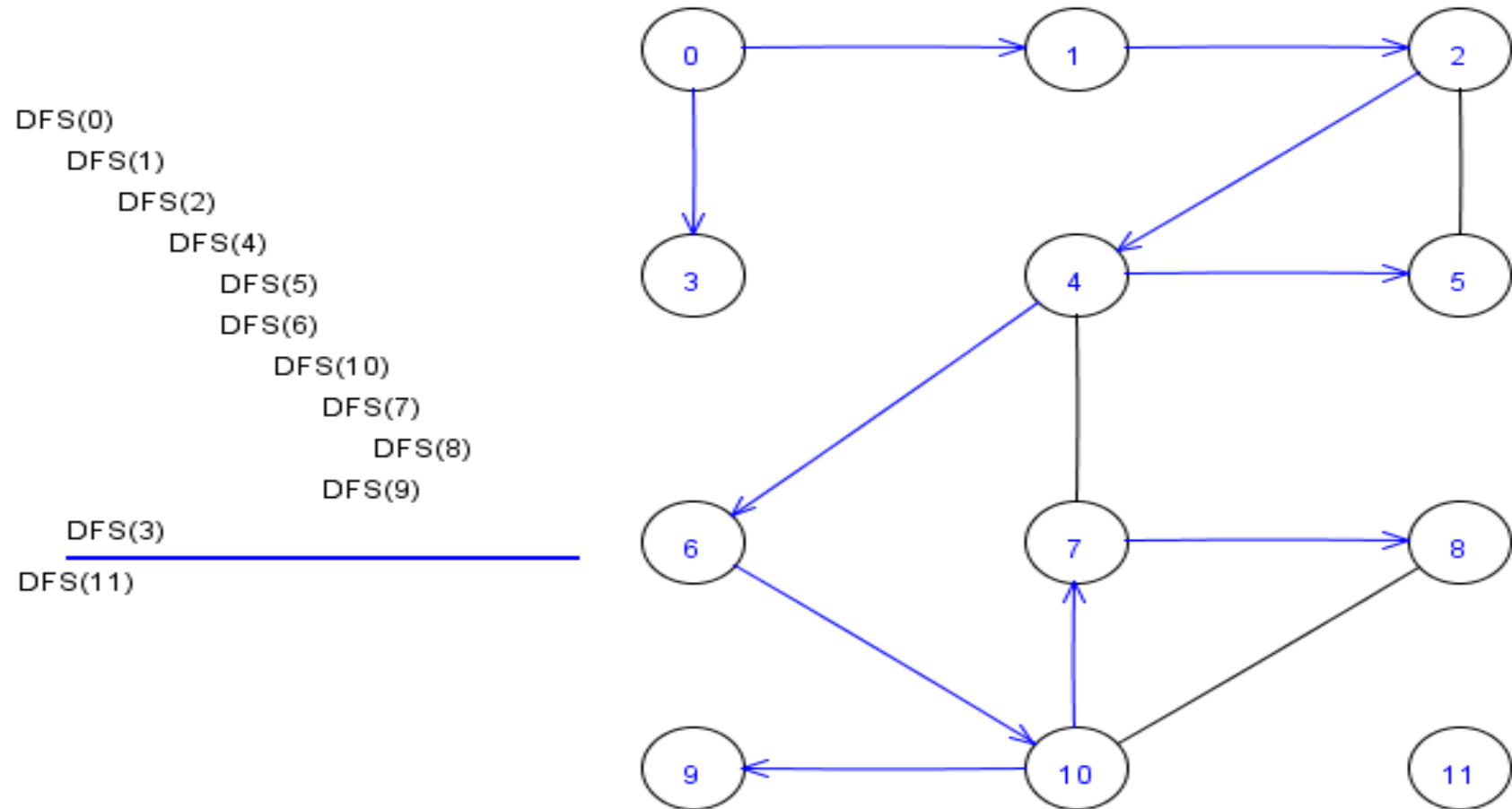
# Graph Traversal

- Depth first search
  - Recursive –  $O(m+n)$  algorithm
  - Start with some node  $v$ 
    - Of all neighbors of  $v$ , goto next  $w$  which is unexplored do DFS( $w$ )
    - If  $w$  explored, then mark edge as back edge
- Similar to a Maze traversal
  - Mark each intersection, corner and dead end (vertex) visited
  - Mark each corridor (edge ) traversed
  - Keep track of the path back to the entrance (start vertex) by means of a rope (recursion)

# DFS Example



# DFS Example Contd





# DFS

## Algorithm DFS( $G, v$ )

label  $v$  as discovered

for all edges from  $v$  to  $w$  in  $G.\text{adjacentEdges}(v)$  do

    if vertex  $w$  is not labeled as discovered then

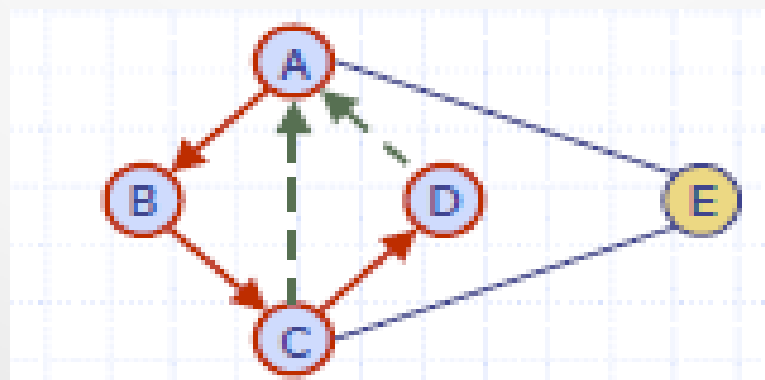
        recursively call DFS( $G, w$ )

# DFS Properties

- $\text{DFS}(G, v)$  visits all the vertices and edges in the connected component of  $v$
- The discovery edges labeled by  $\text{DFS}(G, v)$  form spanning tree of the connected component of  $v$
- Spanning tree is a tree which contains
  - All the nodes in the connected component
  - Subset of edges connecting all the nodes in the component such that no cycles are formed

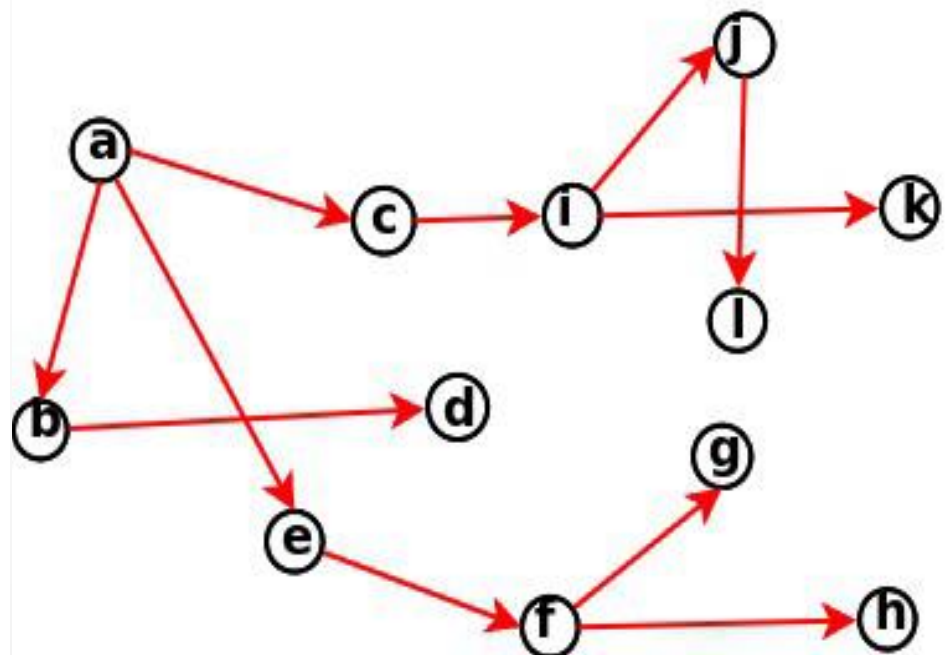
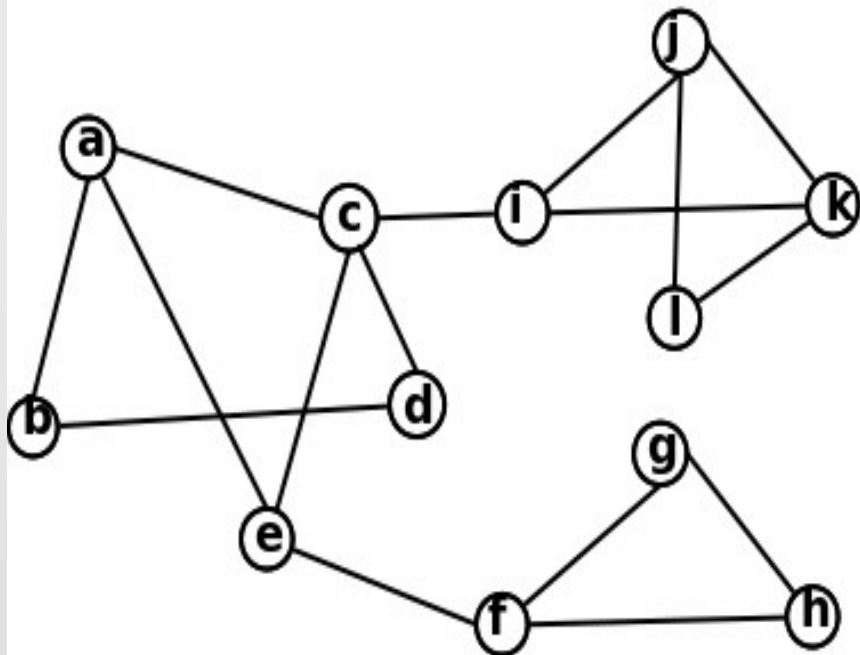
# DFS Applications

- Path finding
  - Can be used to find path between two vertices
- Cycle Finding
  - Usually a cycle is a sequence of forward edges with one backward edge
    - Backward edge indicates an already visited node



# Breadth First Traversal

- Discovery in levels, marks new nodes in levels
  - Cross edges connect to already discovered nodes
- $O(m+n)$



# BFS

Algorithm BFS( $v$ )

initialize container  $L_0$  to contain vertex  $v$  and  $i$  to 0

while  $L_i$  is not empty do

    for each vertex  $v$  in  $L_i$  do

        for each edge  $e$  incident on  $v$

            if  $e$  is unexplored

                let  $w$  be the other endpoint of  $e$

                If  $w$  is unexplored

                    label  $e$  as a discovery edge and insert  $w$  into  $L_{i+1}$

                else label  $e$  as cross edge

29       $i \leftarrow i+1$

# BFS

Algorithm BFS(v)

  //let Q be a queue

  Q.enqueue(v)

  label v as discovered

  while Q is not empty

$v \leftarrow Q.dequeue()$

    for all edges from v to w in G.adjacentEdges(v) do

      if w is not labeled as discovered

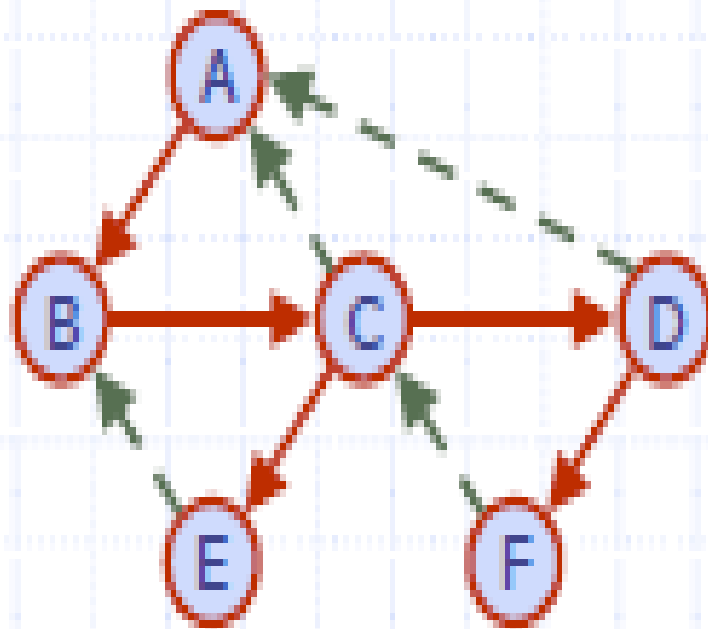
        Q.enqueue(w)

        label w as discovered

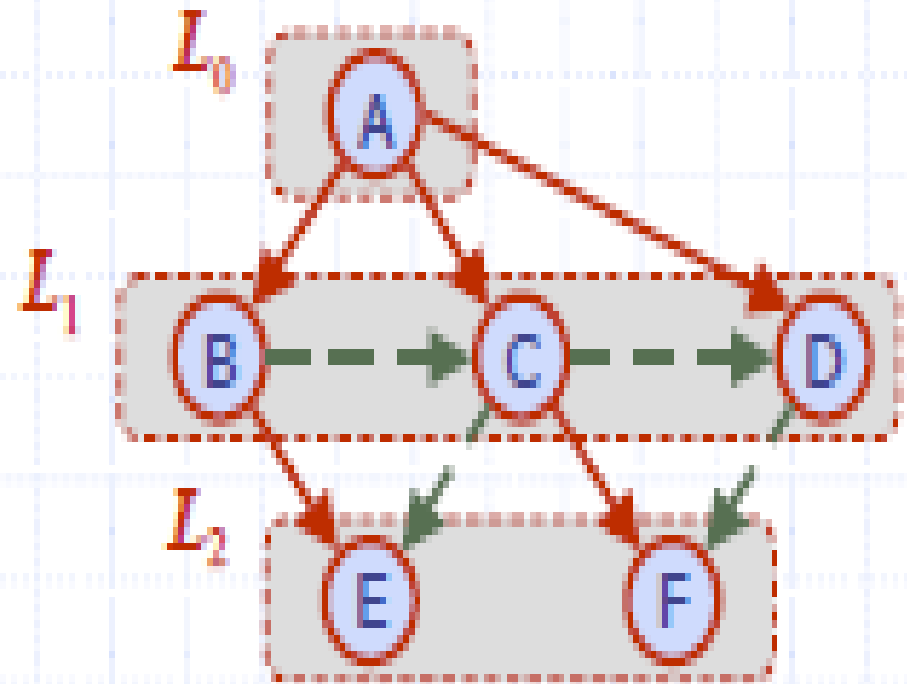
# Applications of BFS

- Finding the shortest path between two nodes  $u$  and  $v$  (with path length measured by number of edges)
- Find a simple cycle, if there is one
- Testing a graph for bipartiteness

# BFS vs DFS



DFS

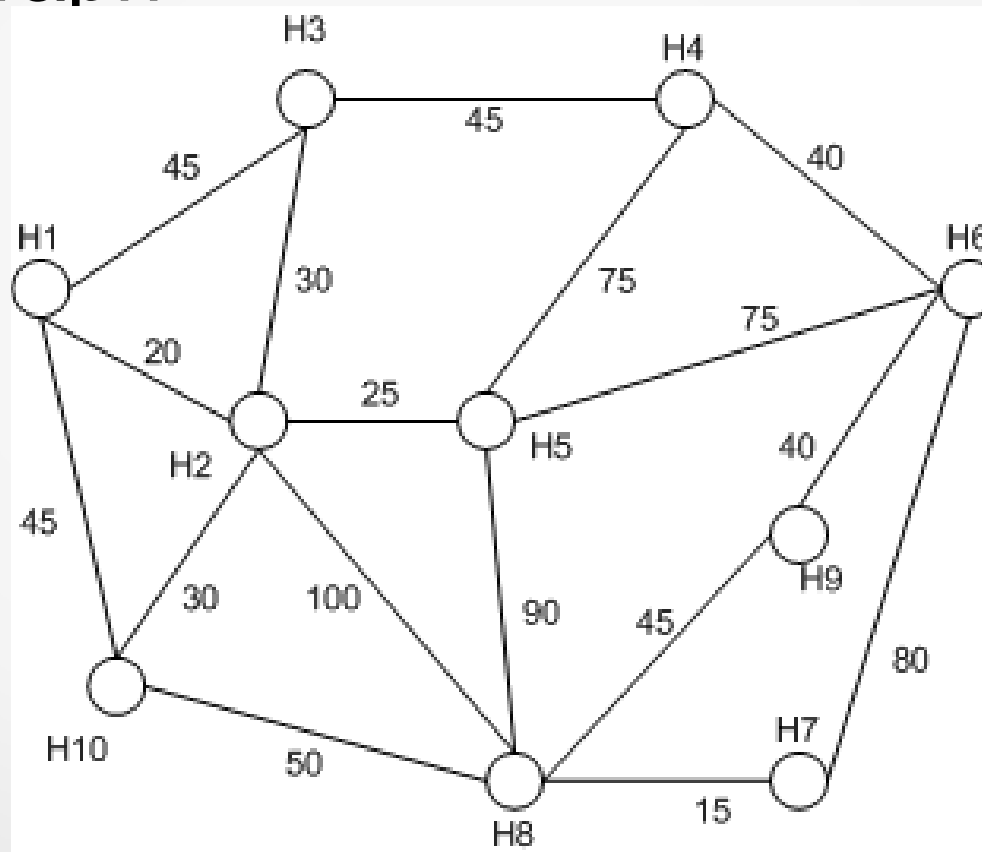


BFS



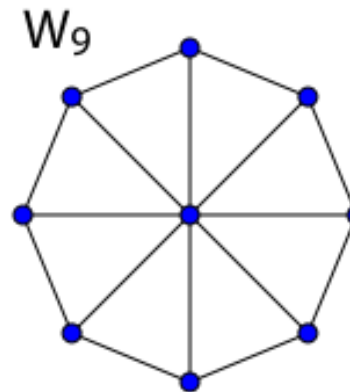
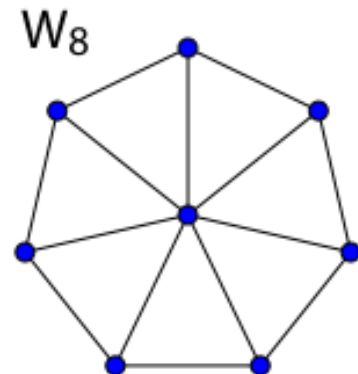
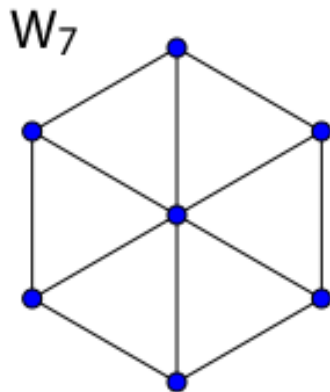
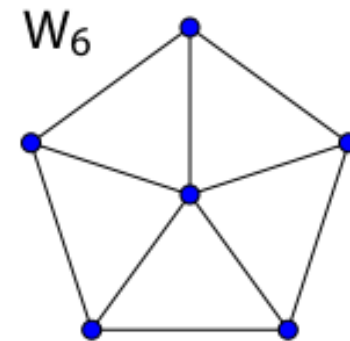
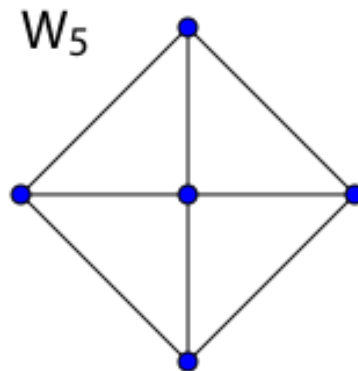
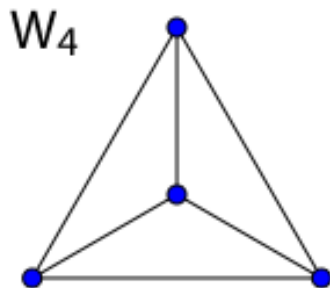
# Exercises

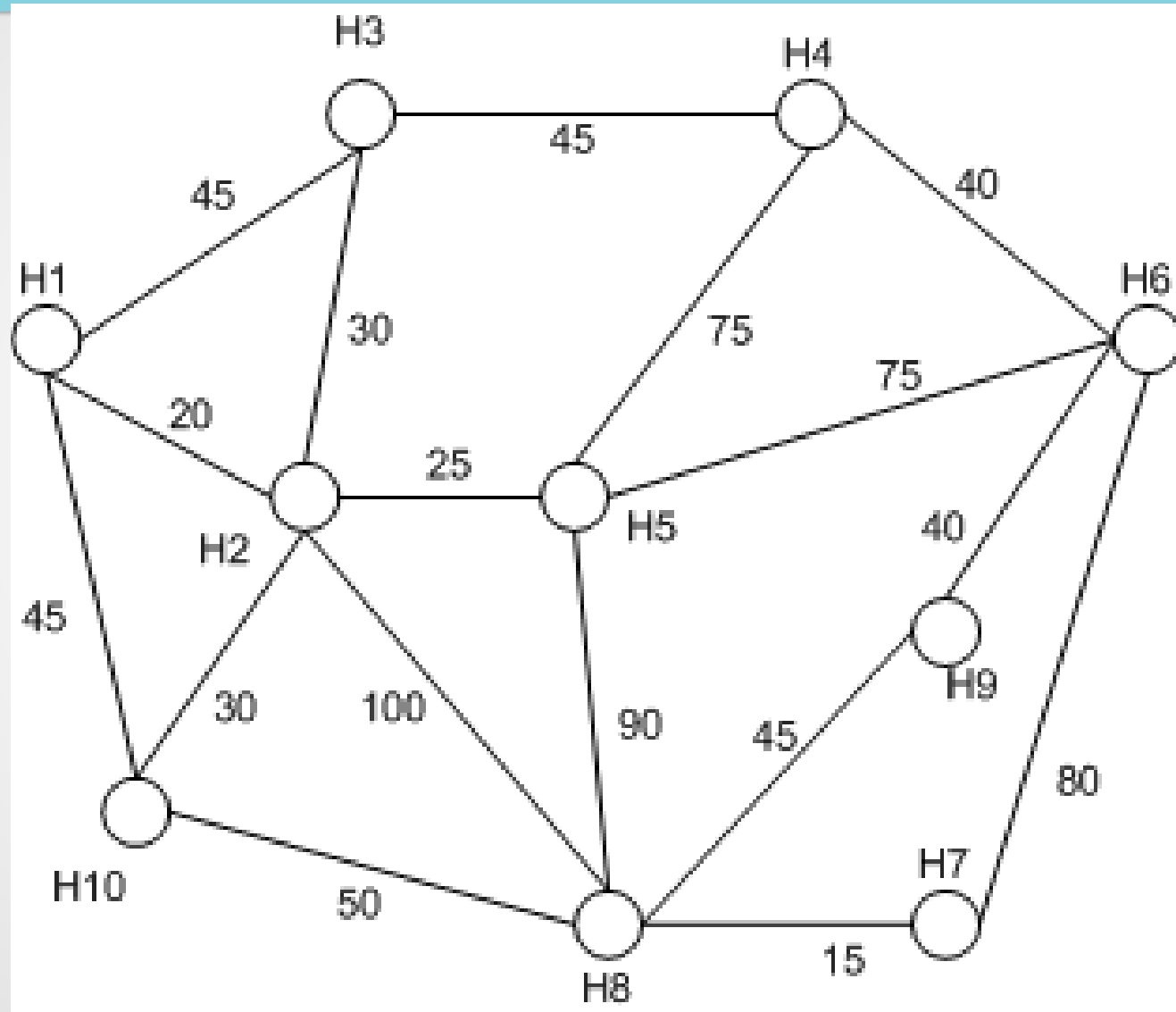
- Do the BFS and DFS traversal over the following graph



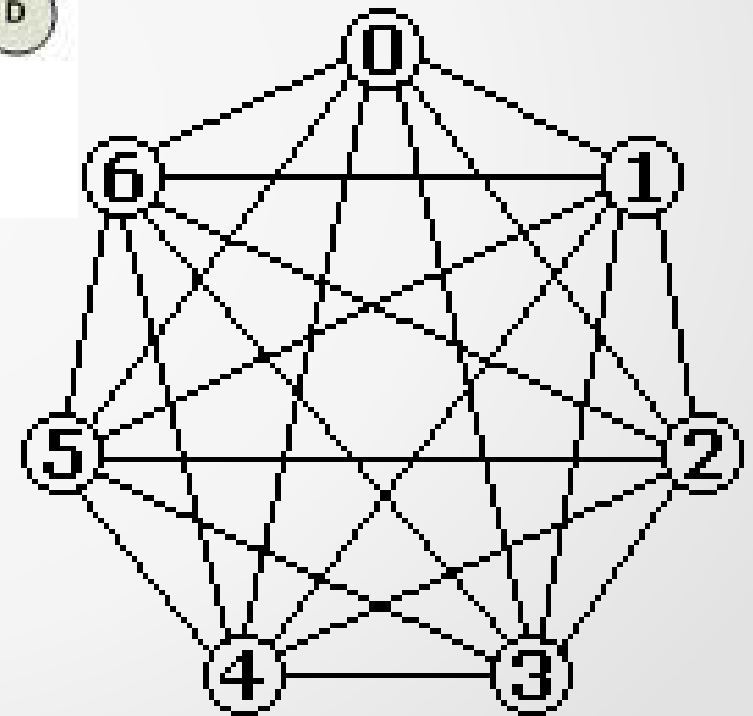
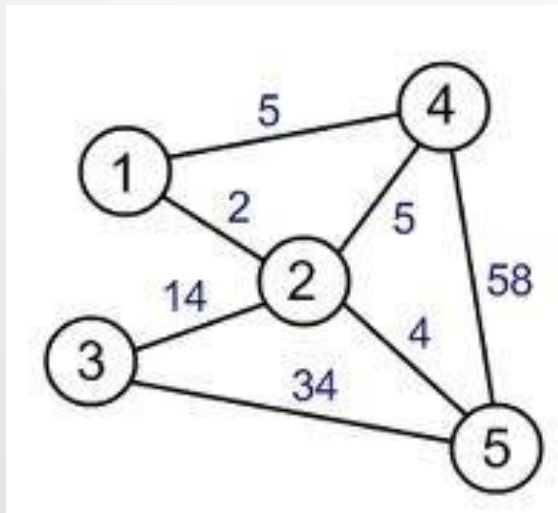
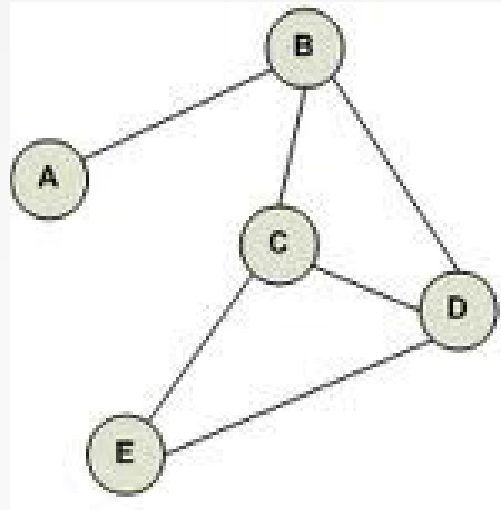
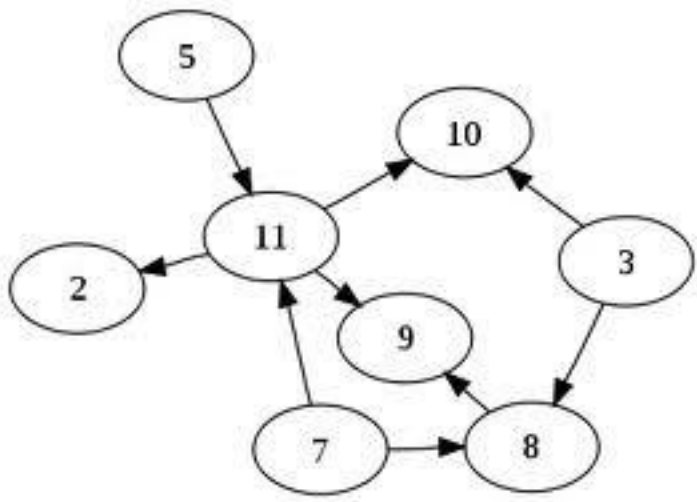
# Exercises

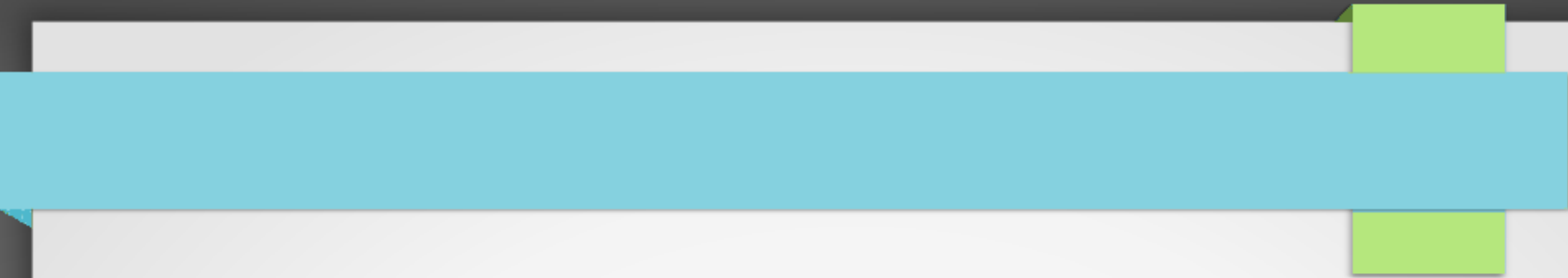
Illustrate all possible structures of traversal trees generated on a wheel graph  $W_n$

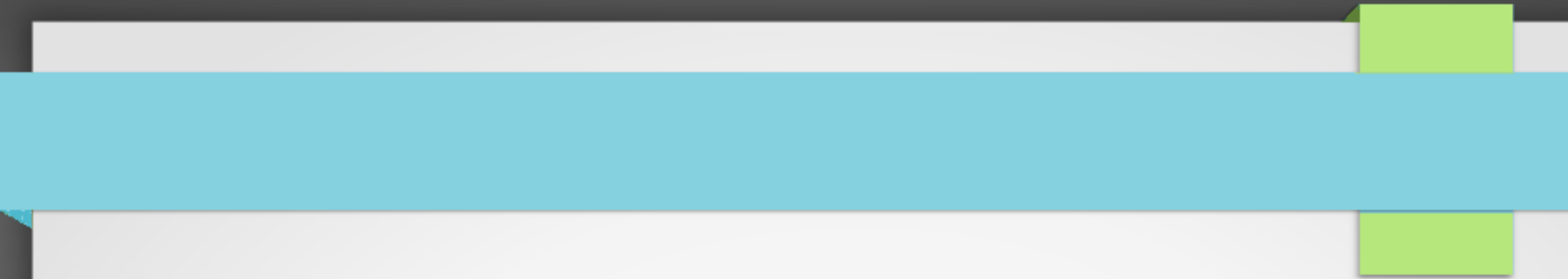




# Problem Solving

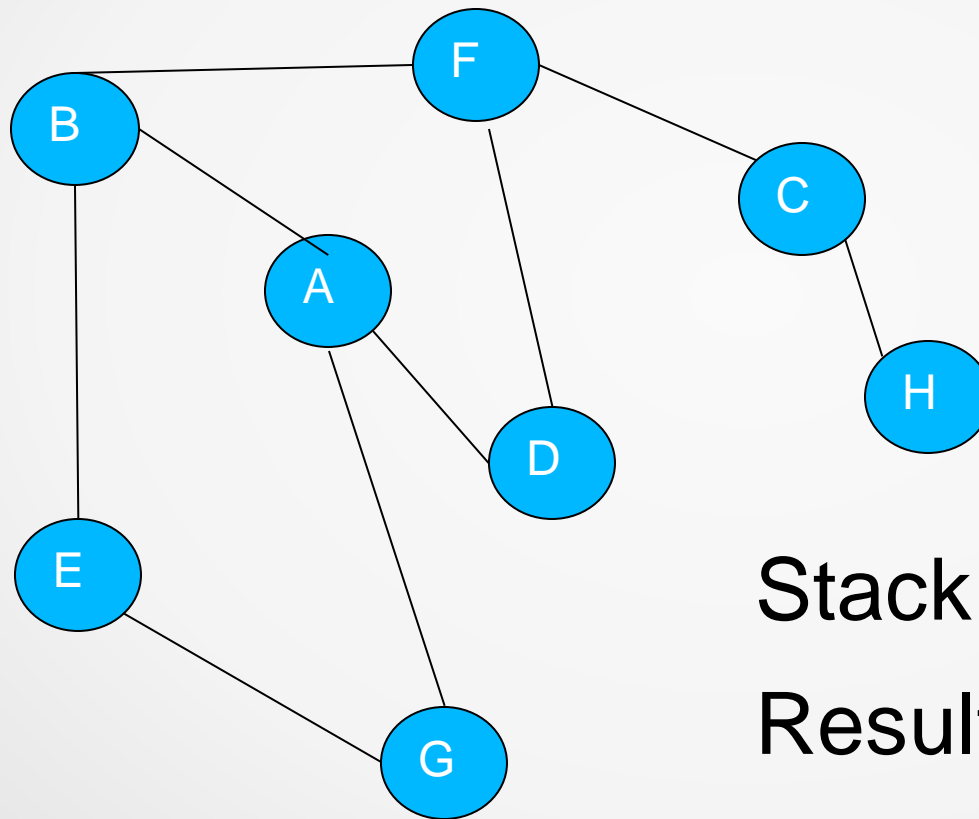


- 
1. Identify all the Adjacent vertices, parallel edges and loops
  2. Find the longest path in each graph and the longest cycle
  3. List out the independent edges and adjacent edges
  4. Draw the adjacency matrix and adjacency list representation for each graph

- 
5. Suppose we represent a graph  $G$  with  $n$  vertices and  $m$  edges with edge list structure. Why does the function `insertVertex()` function take  $O(1)$  time, while `removeVertex()` function runs in  $O(n)$  time
6. Identify 3 different subgraphs for each graph

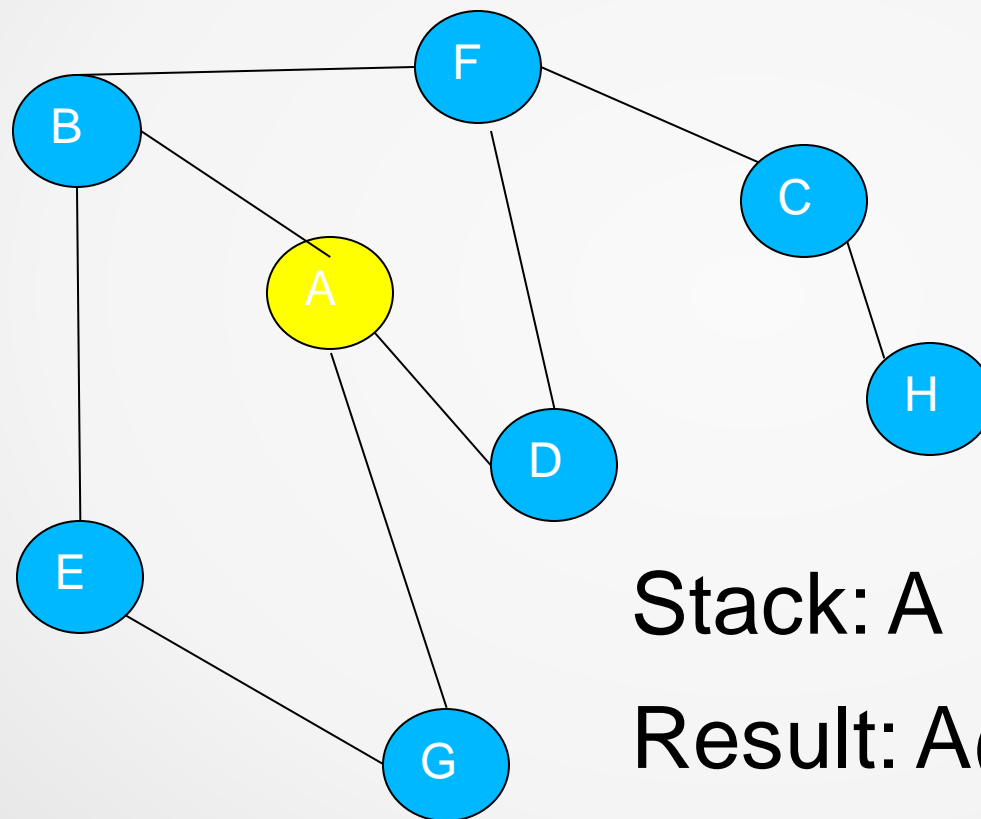
# DFS

- A non recursive way is to use stacks.



Stack :

Result:



Stack: A

Result: A(marked as visited)



# Problem Solving Session

- How are the DFS and BFS modified for a directed graph?
- Write an algorithm to find the longest path in a DAG, where the length of the path is measured by the number of edges that it contains.

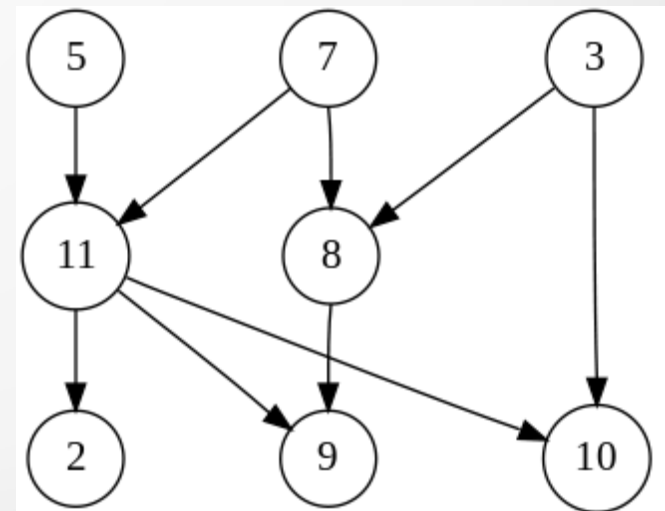
# Tutorial

- How are the DFS and BFS modified for a directed graph?
- Write an algorithm to find the longest path in a DAG, where the length of the path is measured by the number of edges that it contains.

# Digraphs

- A directed graph or digraph is a pair  $G=(V,E)$  of
  - a set  $V$ , whose elements are called vertices or nodes
  - a set  $A$  of ordered pairs of vertices, called arcs, directed edges, or arrows
- A directed acyclic graph (DAG) is a directed graph with no directed cycles

Note: A directed Graph  $G$  is acyclic iff a DFS( $G$ ) yields no back edges



# Digraph Applications

- Used to model several kinds of structures in math or science
- Scheduling
  - Edge (a, b) indicates task a must be completed before b
- Topological Ordering
- Data Processing Networks
  - data enters a processing element through its incoming edges and leaves the element through its outgoing edges
  - Bayesian Networks
  - Compilers
  - Circuit Design

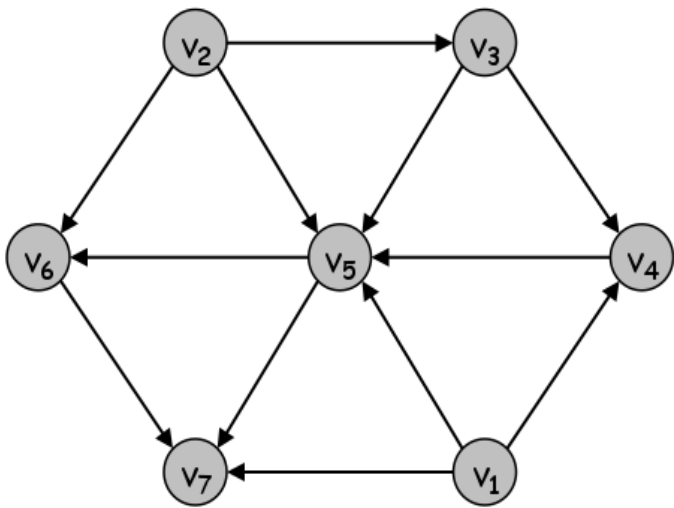
# Topological Ordering

- A DAG is a digraph that has no directed cycles and a topological ordering of a digraph is a numbering  $v_1, \dots, v_n$  of the vertices such that
  - for every edge  $(v_i, v_j)$ , we have  $i < j$
- Example
  - In task scheduling, a topological ordering is a task sequence that satisfies the precedence constraints
- Theorem
  - A digraph admits a topological ordering if and only if it is a DAG

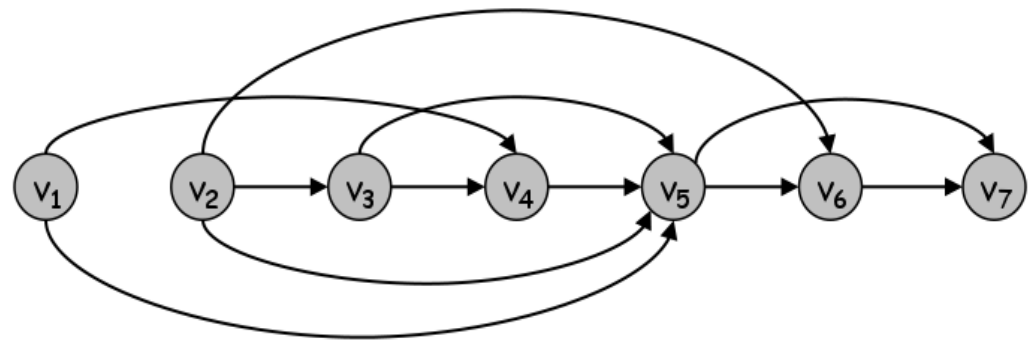
# Topological Ordering

- A DAG is a digraph that has no directed cycles and a topological ordering of a digraph is a numbering  $v_1, \dots, v_n$  of the vertices such that
  - for every edge  $(v_i, v_j)$ , we have  $i < j$
- Example
  - In task scheduling, a topological ordering is a task sequence that satisfies the precedence constraints
- Theorem
  - A digraph admits a topological ordering if and only if it is a DAG

# Topological Ordering Example



a DAG



a topological ordering

# Topological Sorting Algorithm

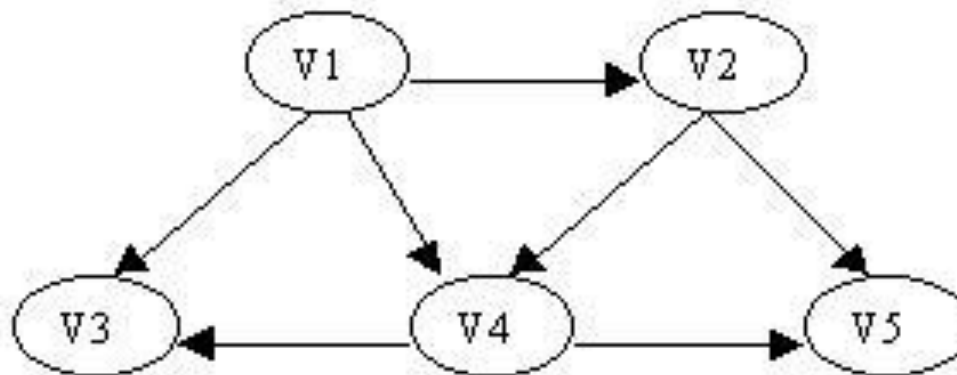
- Let  $S$  be an empty stack
- For each vertex  $u$  of  $G$ , set  $\text{incounter}(u)$  as the  $\text{indegree}(u)$ 
  - If  $\text{incounter}(u) = 0$ , push it in the stack
- Set  $i$  as 1
- If  $S$  is empty it has a directed cycle
- Pop the  $i$ th vertex  $u_i$  from the stack and increment  $i$ 
  - For each neighbor  $v$  of  $u_i$ , decrement  $\text{incounter}(v)$  by 1
  - If  $\text{incounter}(v) = 0$ , push it in stack
  - Do this while stack has elements in it



# Topological Sorting Algorithm Explained

- Compute the indegrees of all vertices
- Find a vertex  $U$  with indegree 0 and print it (store it in the ordering)
  - If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
- Remove  $U$  and all its edges  $(U,V)$  from the graph.
- Update the indegrees of the remaining vertices.
- Repeat steps 2 through 4 while there are vertices to be processed.

# Example



# Example

- Compute Indegrees:
  - $V1 = 0$
  - $V2 = 1$
  - $V3 = 2$
  - $V4 = 2$
  - $V5 = 2$
- Find a vertex with indegree 0:  $V1$
- Output  $V1$  , remove  $V1$  and update the indegrees:
  - Sorted :  $v1$
  - Remove edges  $(V1,V2)$   $(V1,V3)$   $(V1,V4)$
  - Update indegrees

# Example

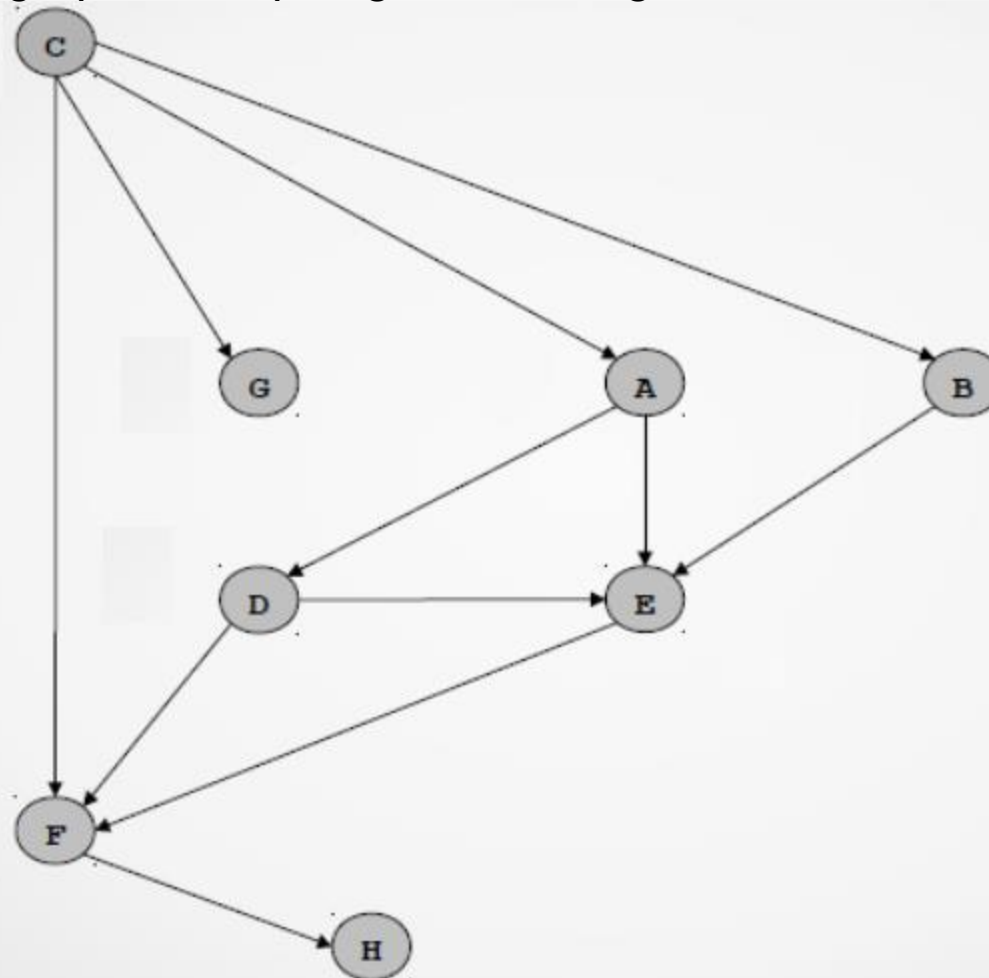
	Indegree					
Sorted →		V1	V1,V2	V1,V2,V4	V1,V2,V4,V3	V1,V2,V4,V3,V5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

# Analysis

- Runs in  $O(n+m)$  time –  $n$  vertices and  $m$  edges
  - Initial computation of indegree  $O(n+m)$
- Uses  $O(n)$  auxiliary space (stack).
- If some vertices are not numbered then there is a cycle
  - Any vertex on a directed cycle will not be visited
  - A vertex visited only when incounter is 0
  - Implies all its previous predecessors were previously visited

# Exercise

Convert the following graph into topological ordering



# Exercise

The root of a DAG is a vertex  $R$  such that every vertex of the DAG can be reached by a directed path from  $R$ . Write an algorithm that takes a directed graph as input and determines the root (if there is one) for the graph. The running time of your algorithm should be  $(|V| + |E|)$ .

# An Interesting Problem

A telecommunications company laying cable to a new neighborhood. It has to ensure that all the houses get cable connections. If it is constrained to bury the cable only along certain paths (e.g. along roads), some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper. Currency is an acceptable unit here.

If you are a member of this company, how would you solve this problem with minimal cost?

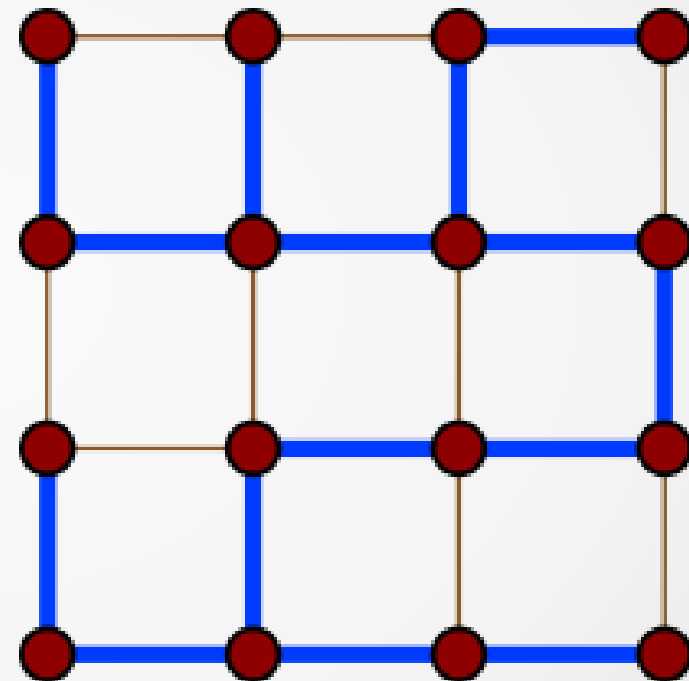
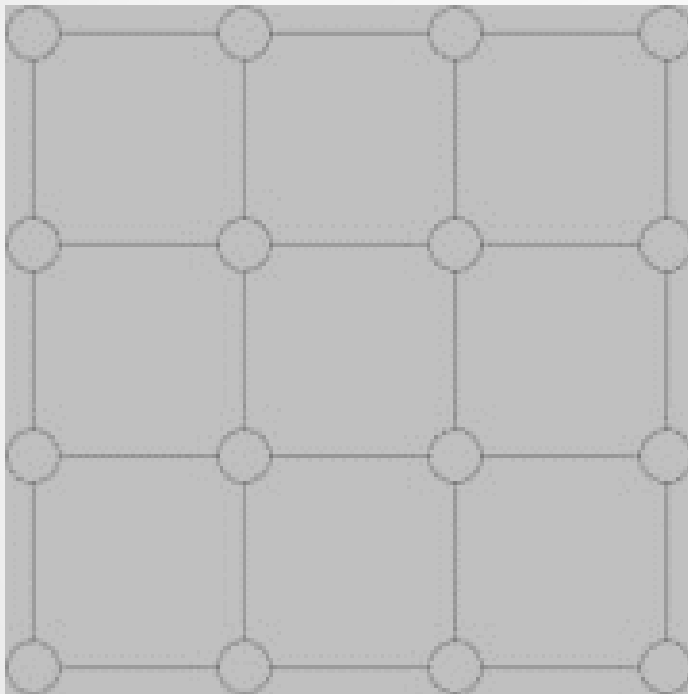


# Spanning Trees

- Given a connected, undirected graph, a spanning tree of that graph is a sub-graph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees
- We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree

# Exercise

Create a spanning tree for the following graph



Can you create more than one tree?

# Minimum Spanning Tree (MST)

- A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree

# Modelling the problem

- The company is constrained to bury the cable only along certain paths (e.g. along roads), then there would be a graph representing which points are connected by those paths.
- Some of those paths might be more expensive - these paths would be represented by edges with larger weights
- The weight of the edges can be the currency
- A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house
- A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable

# Solutions:

- There are different algorithms to find the minimum spanning tree from a graph.
- A few of the famous ones are:
  - Prim's algorithm
  - Kruskal's algorithm
  - Borůvka's algorithm

# Prim's Algorithm

- Algorithm  $\text{Prims}(G, v)$ 
  1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
  2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
  3. Repeat step 2 (until all vertices are in the tree).

## Pseudocode:

Input: A non-empty connected weighted graph with vertices  $V$  and edges  $E$  (the weights can be negative).

Initialize:  $V_{\text{new}} = \{x\}$ , where  $x$  is an arbitrary node (starting point) from  $V$ ,  $E_{\text{new}} = \{\}$

Repeat until  $V_{\text{new}} = V$ :

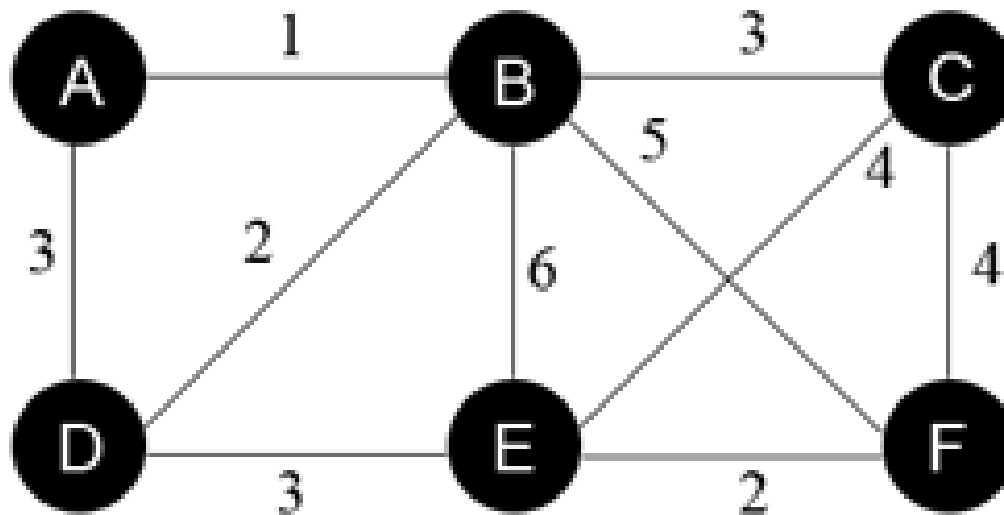
    Choose an edge  $\{u, v\}$  with minimal weight such that  $u$  is in  $V_{\text{new}}$  and  $v$  is not (if there are multiple edges with the same weight, any of them may be picked)

    Add  $v$  to  $V_{\text{new}}$ , and  $\{u, v\}$  to  $E_{\text{new}}$

Output:  $V_{\text{new}}$  and  $E_{\text{new}}$  describe a minimal spanning tree

# Visualization

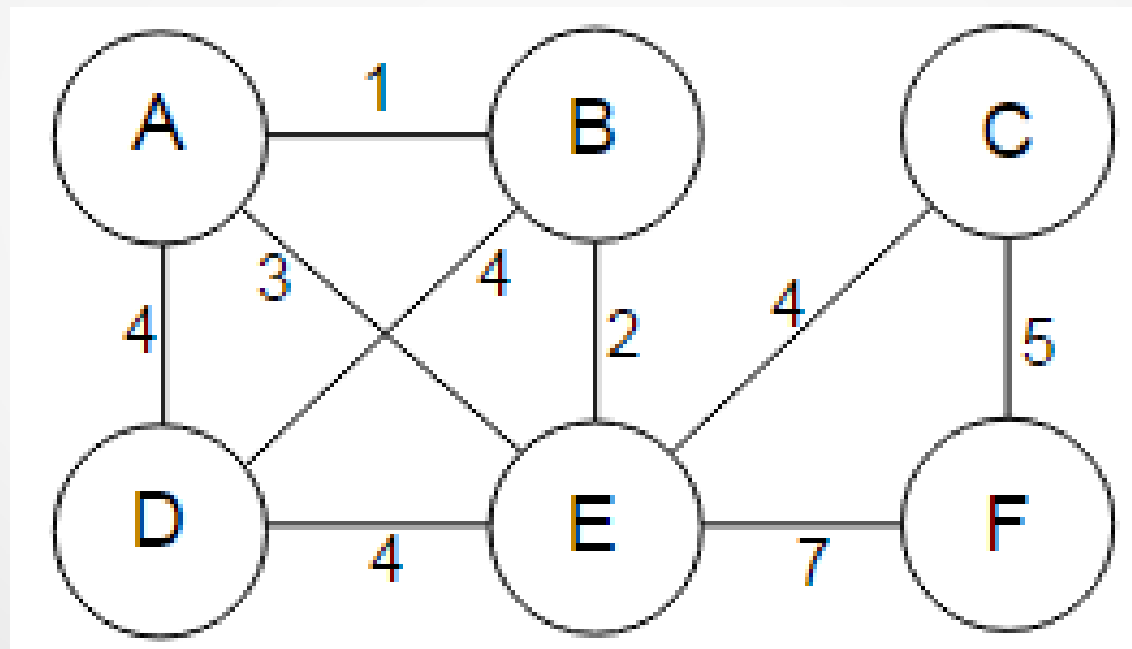
SET: { }





# Exercise

Perform prim's algorithm to find the minimum spanning tree for the following graph



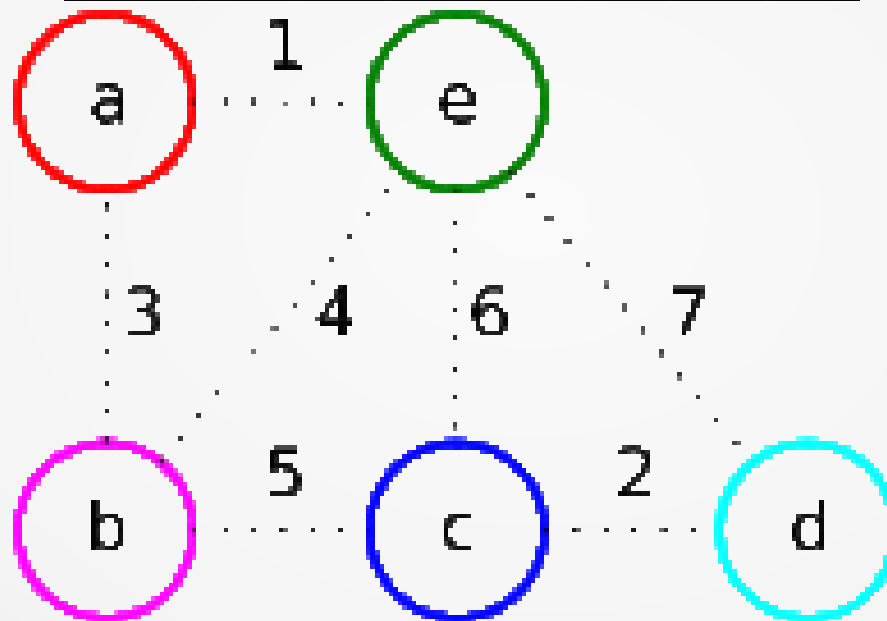
# Kruskal's Algorithm

## Description:

1. create a forest  $F$  (a set of trees), where each vertex in the graph is a separate tree
2. create a set  $S$  containing all the edges in the graph
3. while  $S$  is nonempty and  $F$  is not yet spanning
4.     remove an edge with minimum weight from  $S$
5.     if the removed edge connects two different trees then add it to the forest  $F$ , combining two trees into a single tree

# Kruskals Algorithm

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



# Kruskal's Algorithm

## Pseudocode:

KRUSKAL( $G$ ):

1  $A = \emptyset$

2 foreach  $v \in G.V$ :

3   MAKE-SET( $v$ )

4 foreach  $(u, v)$  ordered by  $\text{weight}(u, v)$ , increasing:

5   if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ):

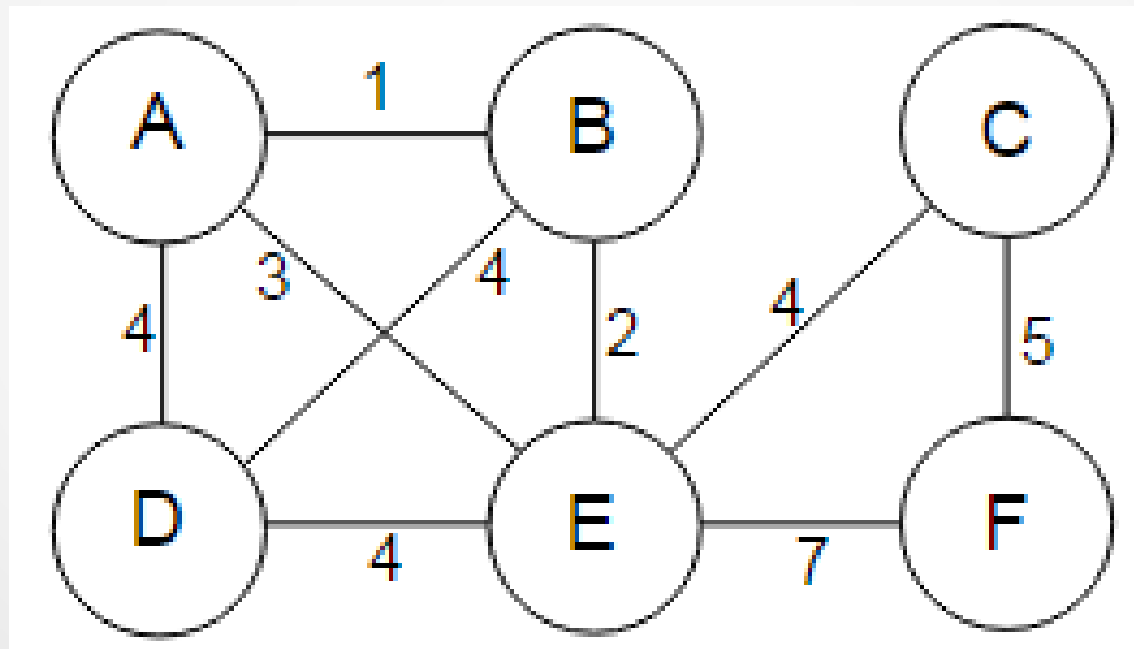
6        $A = A \cup \{(u, v)\}$

7       UNION( $u, v$ )

8 return  $A$

# Exercise

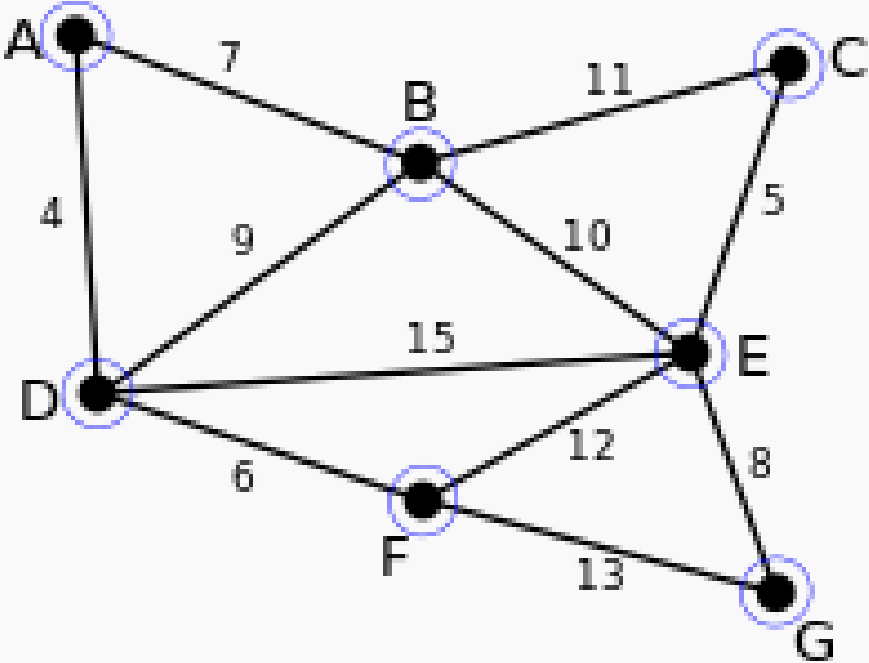
Perform Kruskal's algorithm to find the minimum spanning tree for the following graph.



# Borůvka's algorithm

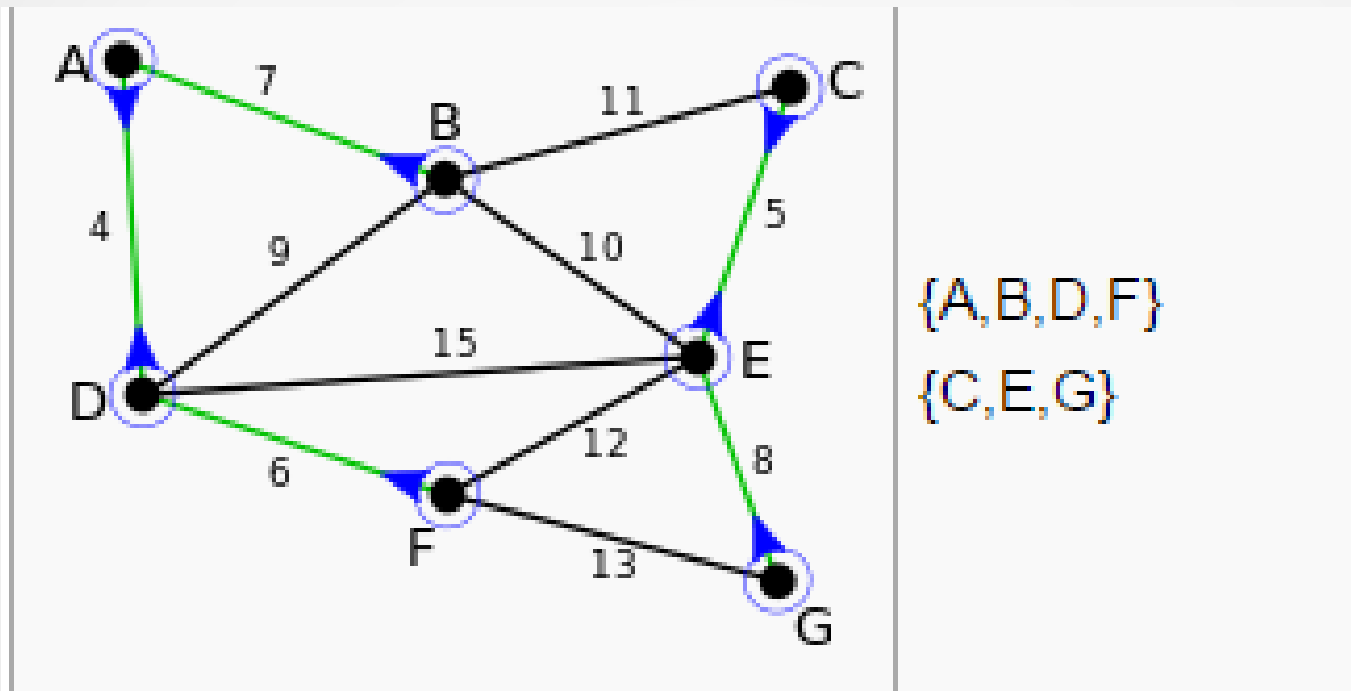
```
boruvkaAlgorithm(graph, weight)
  SpanningTree tree //spanning tree
  components = graph.vertices //at the beginning every
                                vertex is a component
  while components.size > 1 do
    for each component in components
      edge e = connectWithMinimalEdge(component, weight)
      //connect with some other component using a minimal edge
    tree.add(e) //add the edge into the spanning tree
  return tree
```

# Example

Image	components
	$\{A\}$ $\{B\}$ $\{C\}$ $\{D\}$ $\{E\}$ $\{F\}$ $\{G\}$

Source: wiki

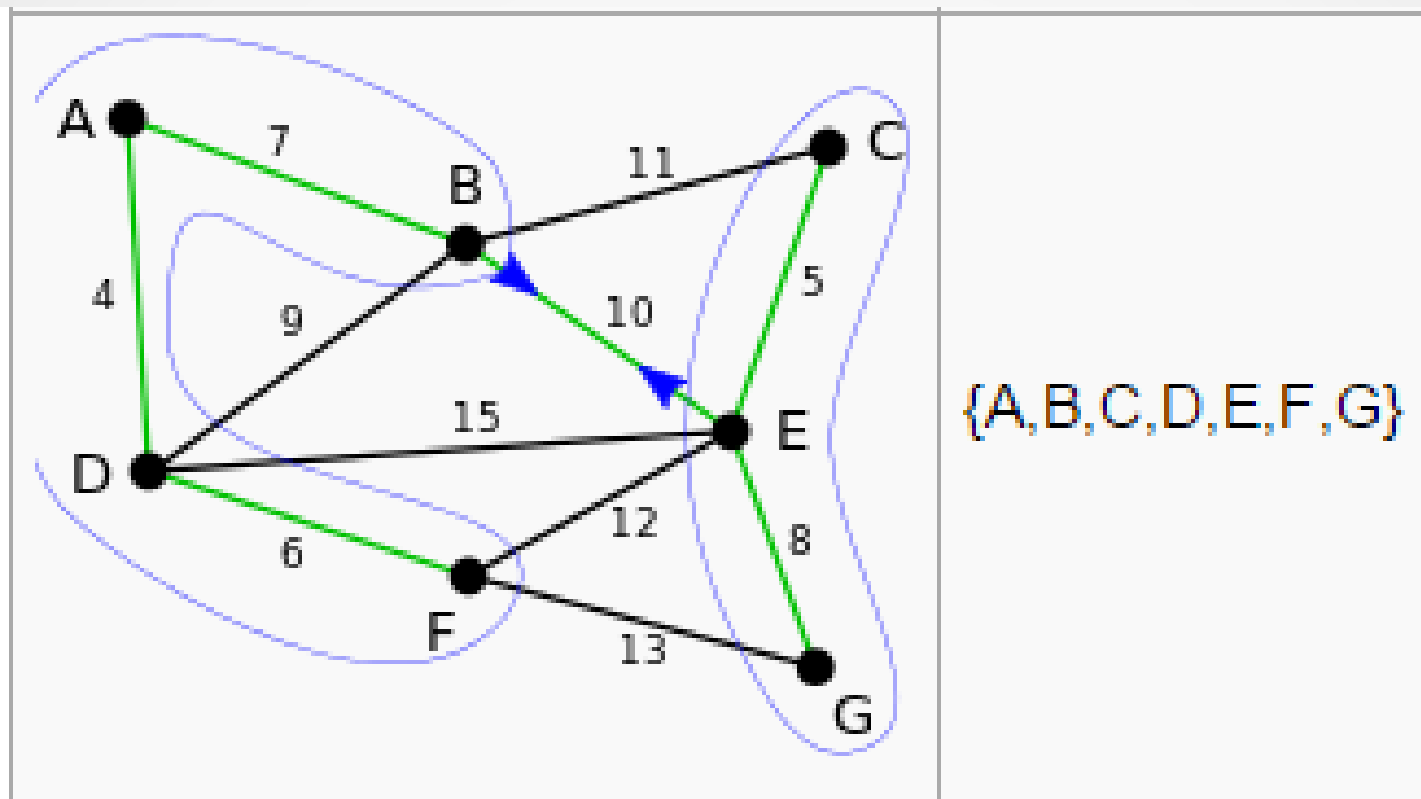
# Example cont.



Source:



# Example cont.



Source: wiki

# Shortest Path

The problem:

Finding a path between two vertices (or nodes) in a graph  $G(V,E)$  such that the sum of the weights of its constituent edges is minimized

# Shortest Path

Types:

## 1. Single - Source Shortest Paths

1. Dijkstra's algorithm –  $O(V^2, O((E+V)\log V))$
2. Bellman–Ford algorithm –  $O(VE)$

## 2. All – Pairs Shortest Path

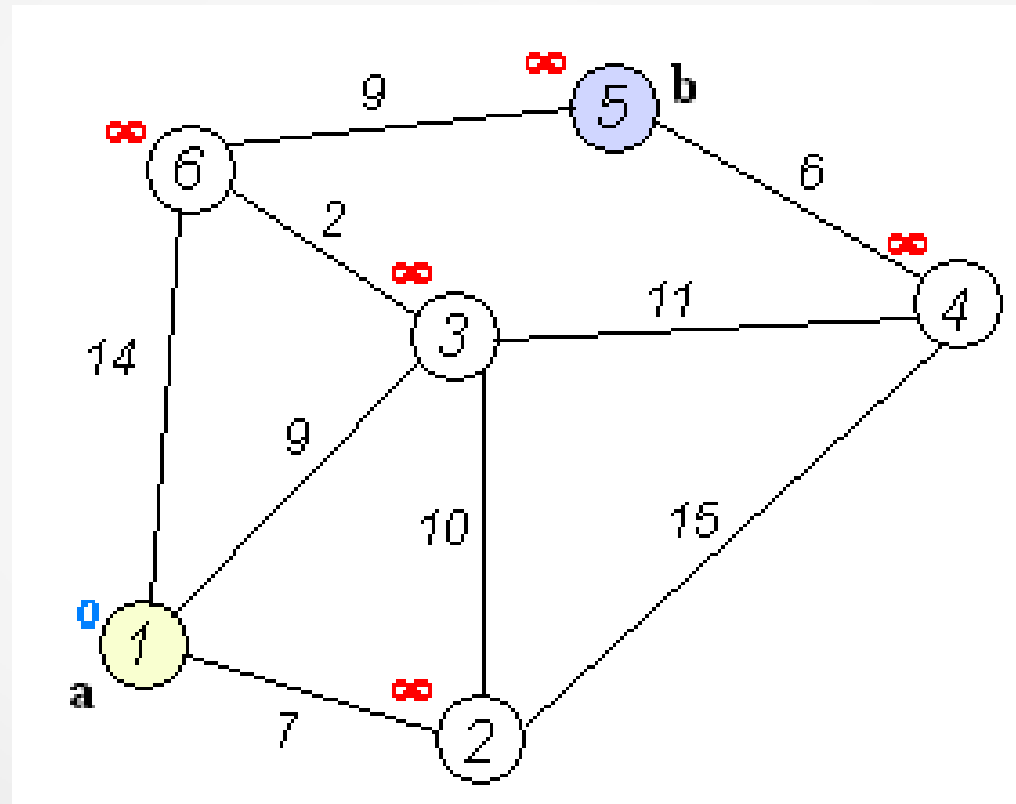
1. Floyd-Warshall algorithm -  $O(V^3)$

# Dijkstra's Algorithm

```
For each vertex v in Graph: // Initialization
    if v ≠ source           // Where v has not yet been removed from Q (unvisited nodes)
        dist[v] ← infinity    // Unknown distance function from source to v
        prev[v] ← undefined   // Previous node in optimal path from source
    end if
    add v to Q               // All nodes initially in Q (unvisited nodes)
End For
While Q is not empty:
    u ← vertex in Q with min dist[u] // Source node in first case
    remove u from Q
    for each neighbor v of u:         // where v is still in Q.
        alt ← dist[u] + length(u, v)
        if alt < dist[v]:             // A shorter path to v has been found
            dist[v] ← alt
            prev[v] ← u
        end if
    end for
End While
```

# Dijkstra's Algorithm

## Animation



# Bellman – Ford Algorithm

Pseudocode:

```
function BellmanFord(list vertices, list edges, vertex source)
    ::distance[],predecessor[]
// Step 1: initialize graph
// Step 2: relax edges repeatedly
// Step 3: check for negative-weight cycles
```

# Bellman – Ford Algorithm

// Step 1: initialize graph

for each vertex  $v$  in vertices:

if  $v$  is source then  $\text{distance}[v] := 0$

else  $\text{distance}[v] := \text{inf}$

$\text{predecessor}[v] := \text{null}$

# Bellman – Ford Algorithm

```
// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u
```

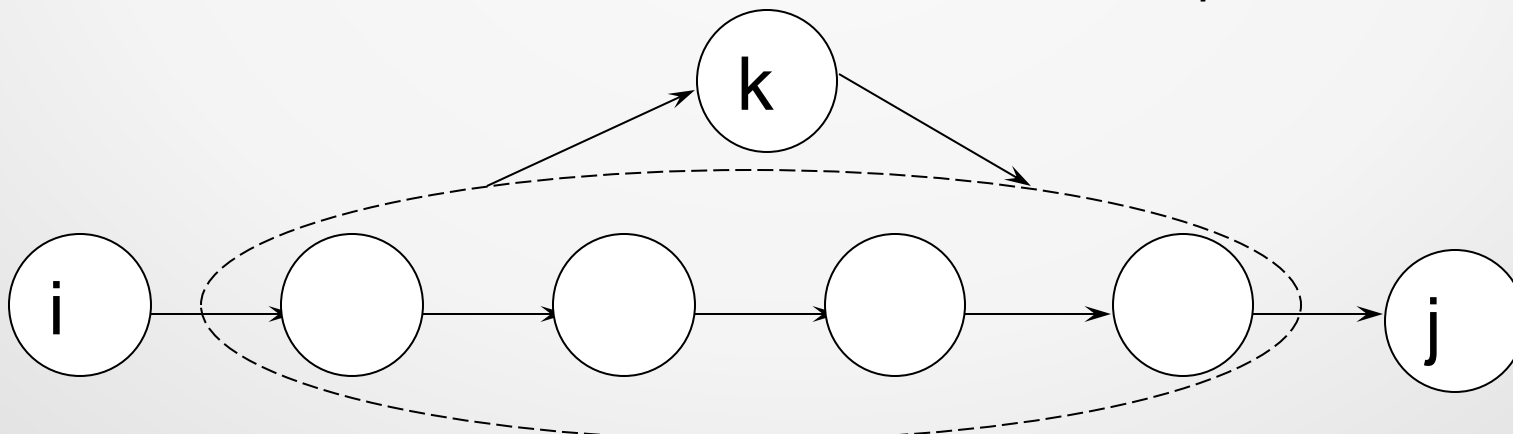


# Bellman – Ford Algorithm

```
// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```

# Floyd -Warshall

- Description
  - Let the vertices in a graph be numbered from 1..n. Consider the subset  $\{1,2,\dots, k\}$  of these n vertices.
  - Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set  $\{1,2,\dots,k\}$  only.
  - There are two situations:
    - k is an intermediate vertex on the shortest path.
    - k is not an intermediate vertex on the shortest path.



# Floyd - Warshall

## Pseudocode:

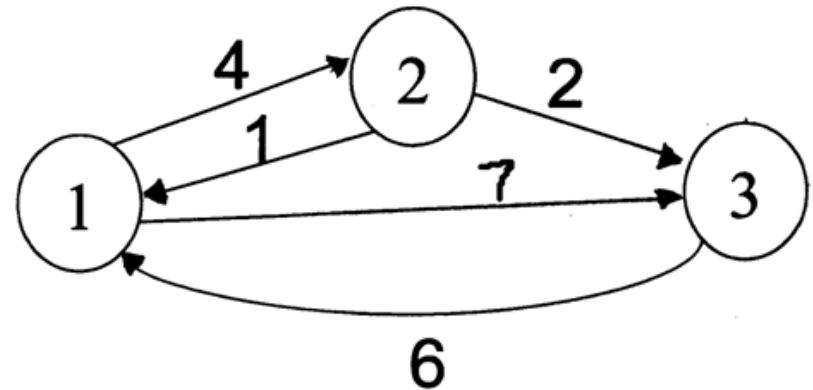
```
for i = 1 to N
  for j = 1 to N
    if there is an edge from i to j
       $\text{dist}[0][i][j] = \text{the length of the edge from } i \text{ to } j$ 
    else
       $\text{dist}[0][i][j] = \text{INFINITY}$ 

for k = 1 to N
  for i = 1 to N
    for j = 1 to N
       $\text{dist}[k][i][j] = \min(\text{dist}[k-1][i][j], \text{dist}[k-1][i][k] + \text{dist}[k-1][k][j])$ 
```

# Floyd - Warshall Example

$$S \quad D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

**Consider Vertex 1:**

$$D(3,2) = D(3,1) + D(1,2)$$

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

**Consider Vertex 2:**

$$D(1,3) = D(1,2) + D(2,3)$$

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

**Consider Vertex 3:**

Nothing changes.

Source:

<http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/>

# Floyd – Warshall Algorithm

An excellent animation:

<https://www.cs.usfca.edu/~galles/visualization/Floyd.html>

# END