Graphs

Hours 4
Marks 5

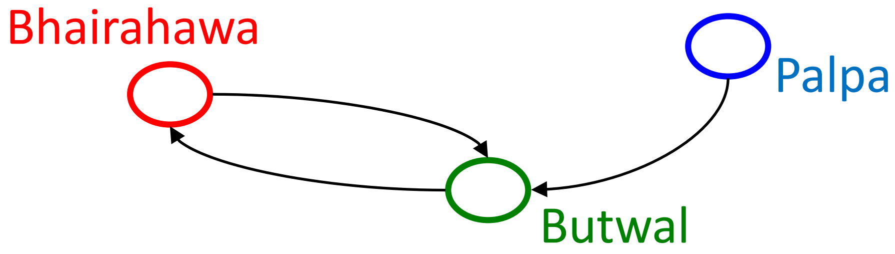
Graph

- Graph
- Introduction
- Representation of Graph
  - Array
  - Linked List
- Traversal
  - Depth First Search
  - Breadth First Search
- Minimum Spanning Tree
  - Kruskal’s algorithm

Introduction

- Graphs are a formalism for representing relationships between objects.
  - a graph **G** is represented as **G = (V, E)**,

    where,
    - **V** is a set of vertices
    - **E** is a set of edges



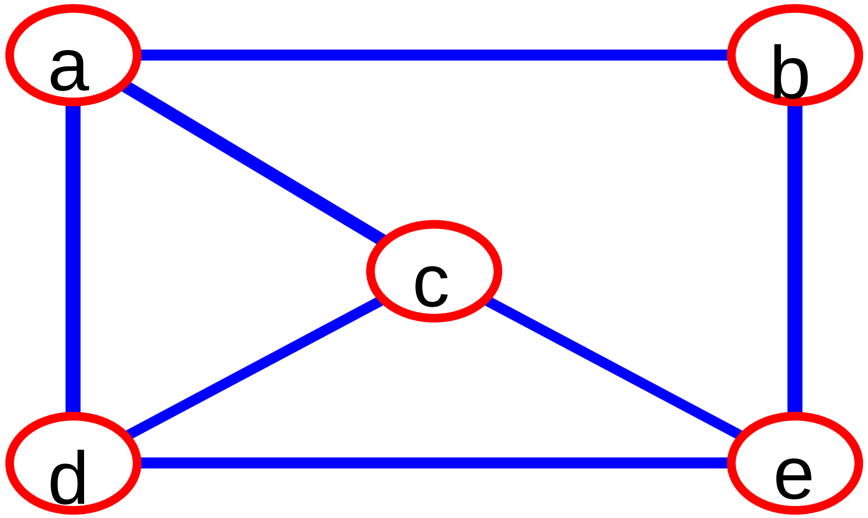
V = {Bhairahawa, Butwal, Palpa}
E = {(Palpa, Butwal),
(Bhairahawa, Butwal),
(Butwal, Bhairahawa)}

Graph

- A graph G = (V,E) is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- **Example:**



V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

Graph ADT

structure *Graph* is
objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.
functions:
for all $graph \in Graph$, v, v_1, and $v_2 \in Vertices$

Graph Create()	::=	**return** an empty graph.
Graph InsertVertex(*graph*, v)	::=	**return** a graph with v inserted. v has no incident edges.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with a new edge between v_1 and v_2.
Graph DeleteVertex(*graph*, v)	::=	**return** a graph in which v and all edges incident to it are removed.
Graph DeleteEdge(*graph*, v_1, v_2)	::=	**return** a graph in which the edge (v_1, v_2) is removed. Leave the incident nodes in the graph.
Boolean IsEmpty(*graph*)	::=	**if** (*graph* == empty graph) **return** *TRUE* **else return** *FALSE*.
List Adjacent(*graph*, v)	::=	**return** a list of all vertices that are adjacent to v.

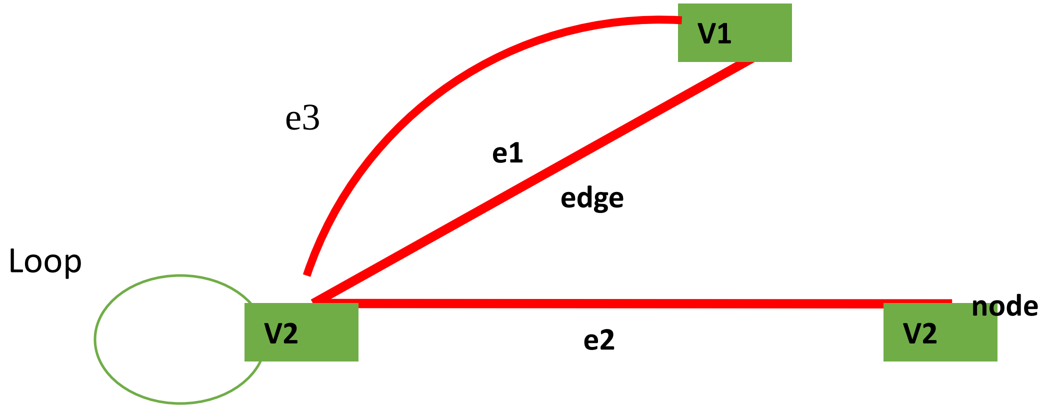
Graph Terminology:

- **Node**
  Each element of a graph is called node of a graph
- **Edge**
  Line joining two nodes is called an edge.
  It is denoted by e=[u,v] where u and v are adjacent vertices.

- **Loop**
  An edge of the form (u, u) is said to be a ***loop***. Here in figure [v2,v2]

- **Multiedge**

  two or more edges that are incident to the same two vertices, or in a directed graph, two or more edges with both the same tail vertex and the same head vertex. In figure, e1 and e3.



Adjacent and Incident

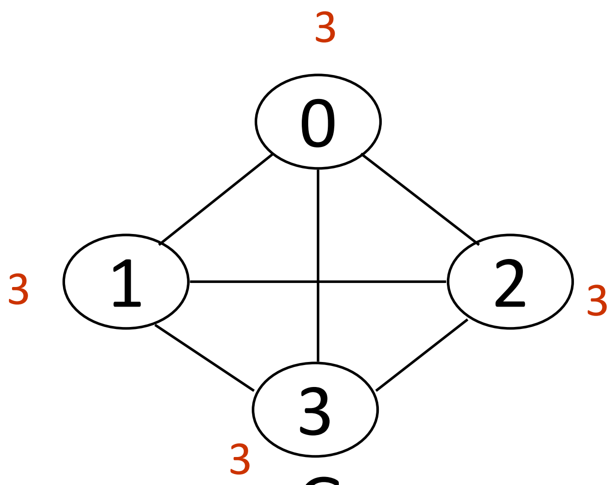
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

Degree of a Vertex

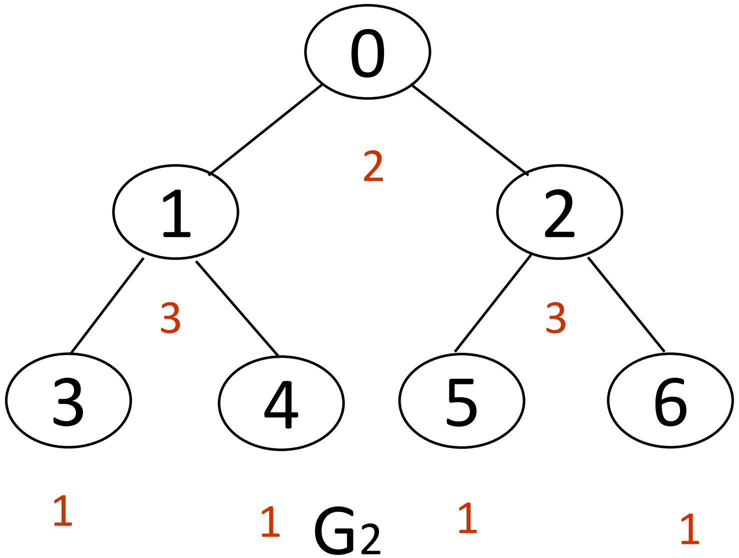
- The degree of a vertex is the number of edges incident to that vertex
- A node with degree 0 is known as isolated node.
- A node with degree 1 is known as pendant node.
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Examples



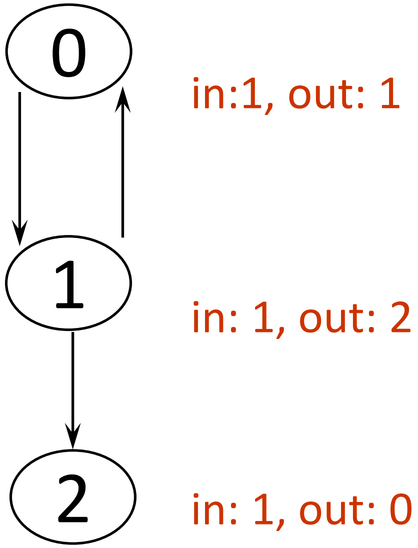
G1



G2

directed graph

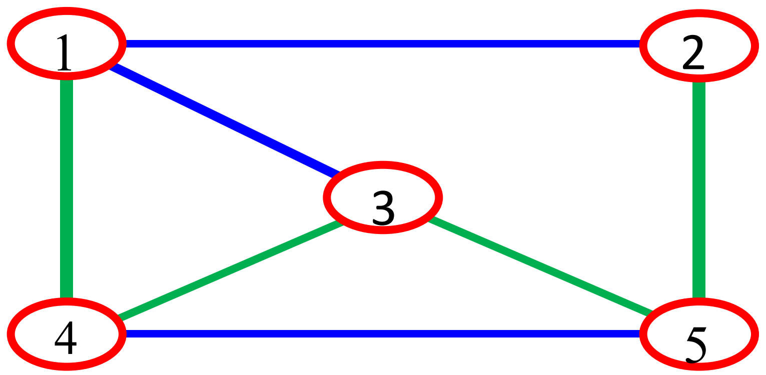
in-degree
out-degree



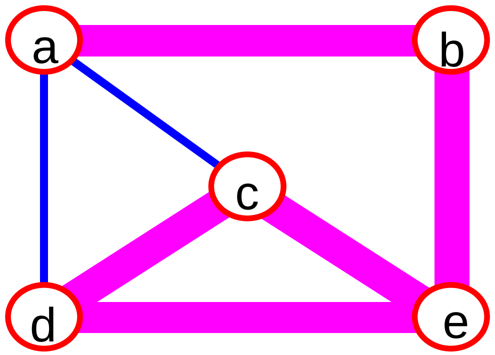
G3

Path

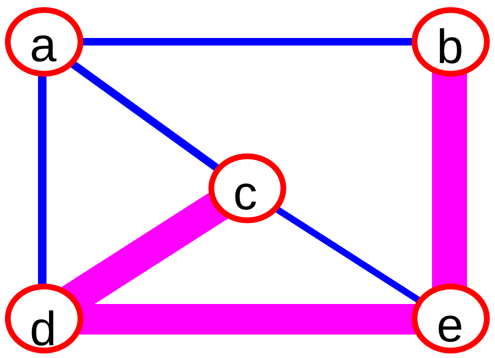
- Path: A sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.
- Example: {1, 4, 3, 5, 2}



- **Length of a path:** Number of edges on the path

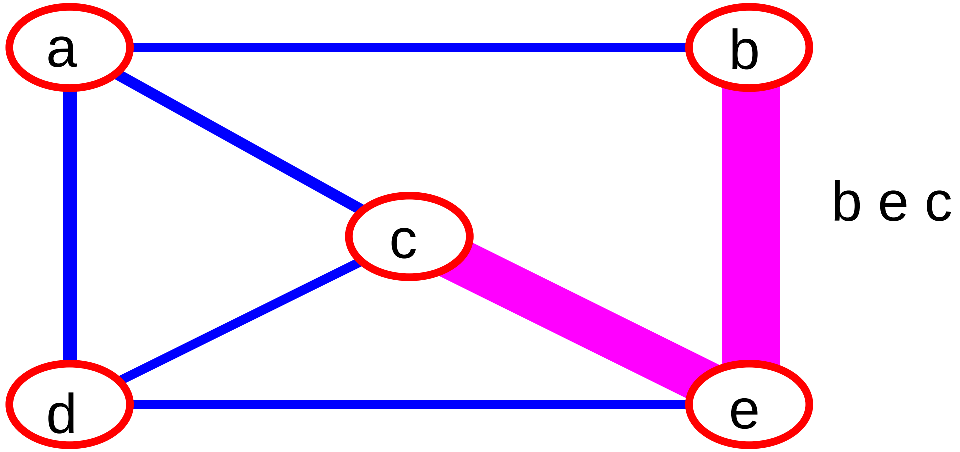


a b e d c

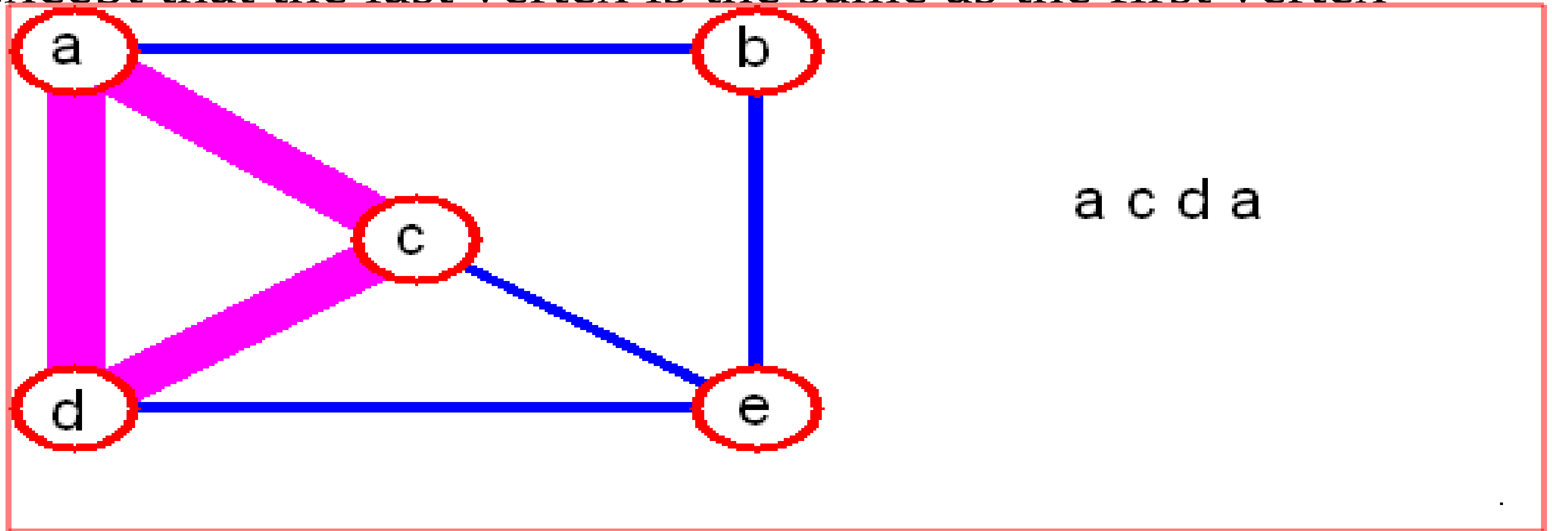


b e d c

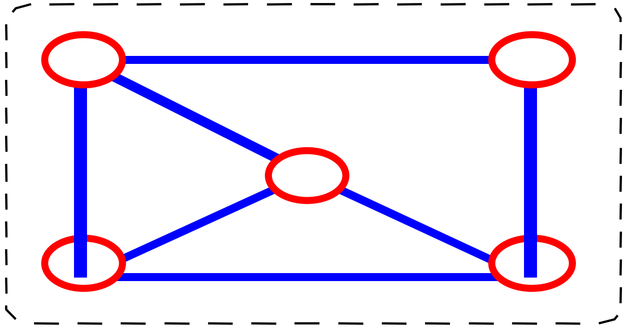
- simple path: The path with no repeated vertices



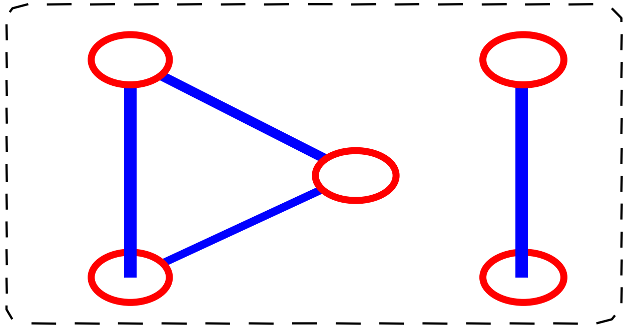
- cycle: simple path, except that the last vertex is the same as the first vertex



- connected graph: any two vertices are connected by some path

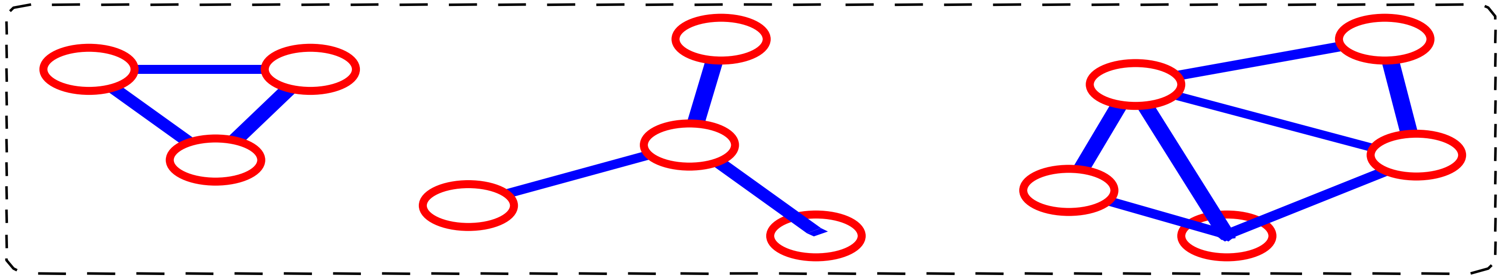


connected



not connected

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.

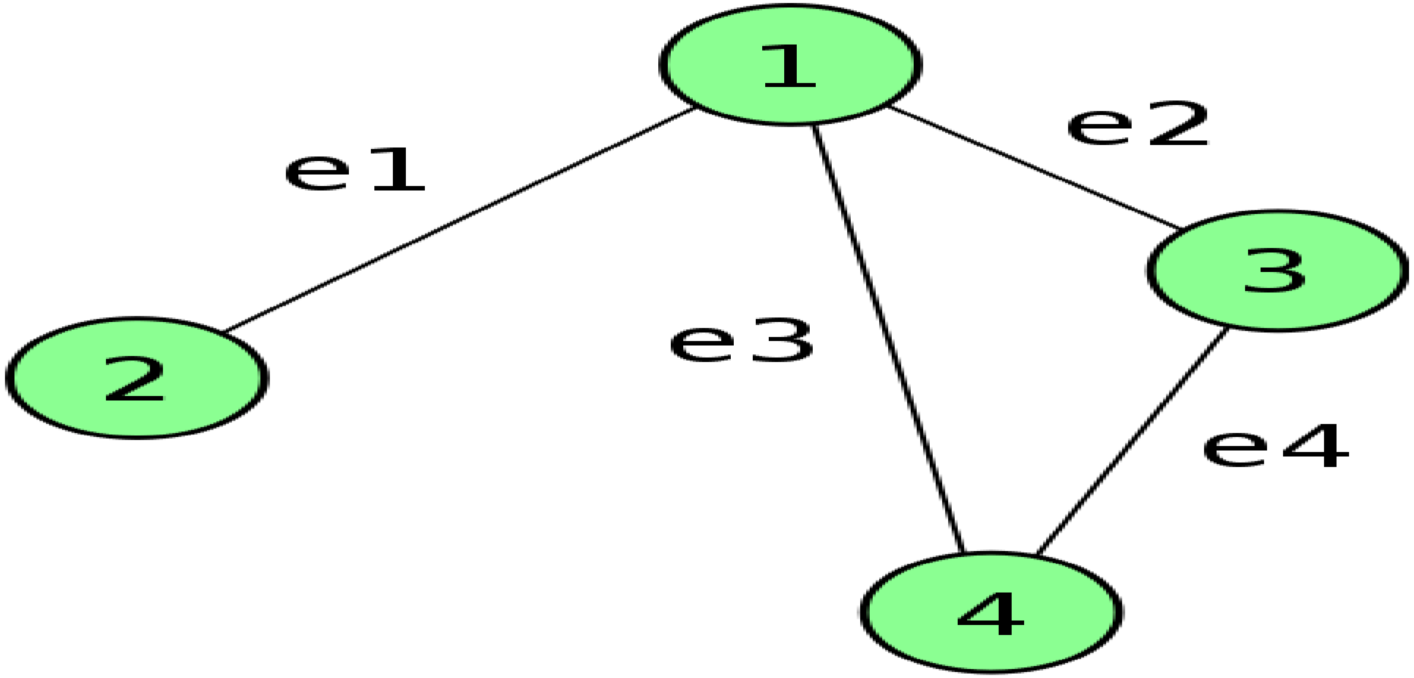


Types

- Graphs are generally classified as,
  - Directed graph
  - Undirected graph

UnDirected graph

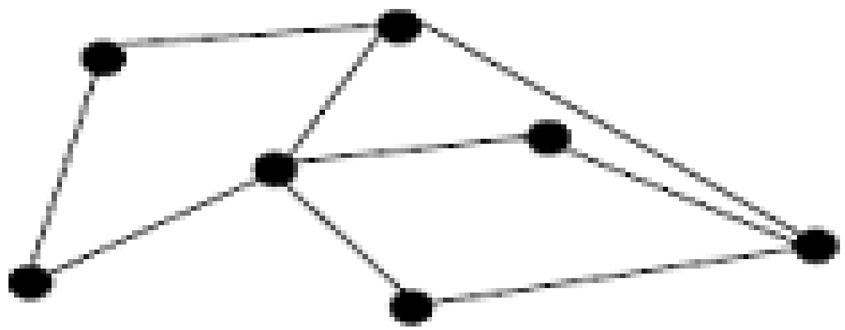
- A graphs G is called undirected graph if each edge has no direction.



Types of Graph

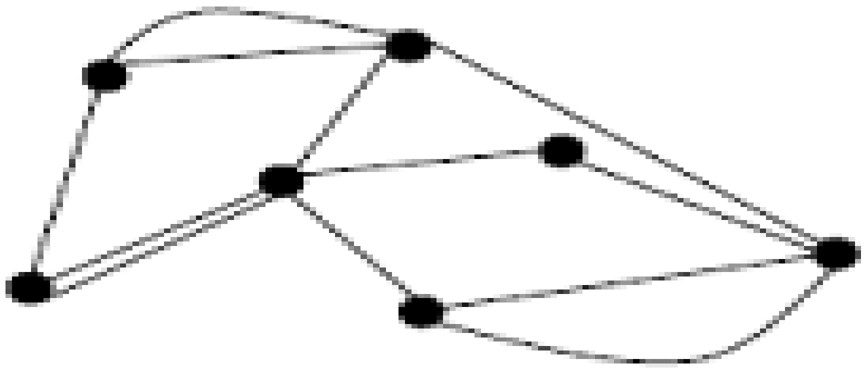
- Simple Graph

  A graph in which there is no loop and no multiple edges between two nodes.



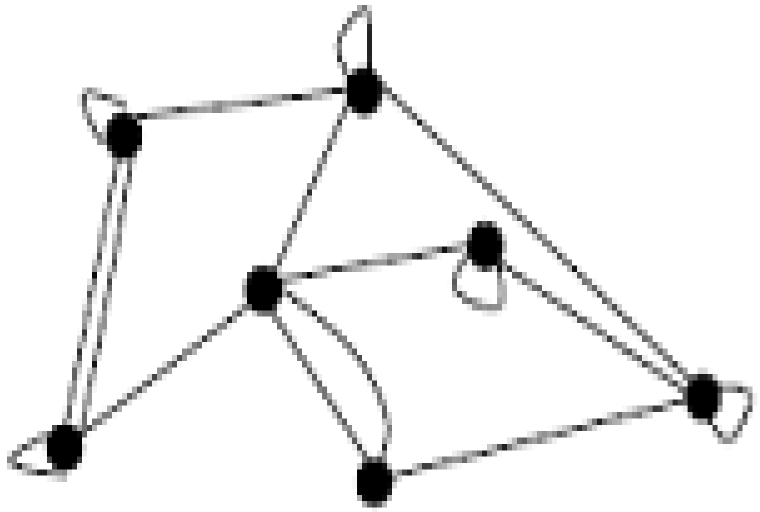
- Multigraph

  The graph which has multiple edges between any two nodes but no loops is called multigraph.



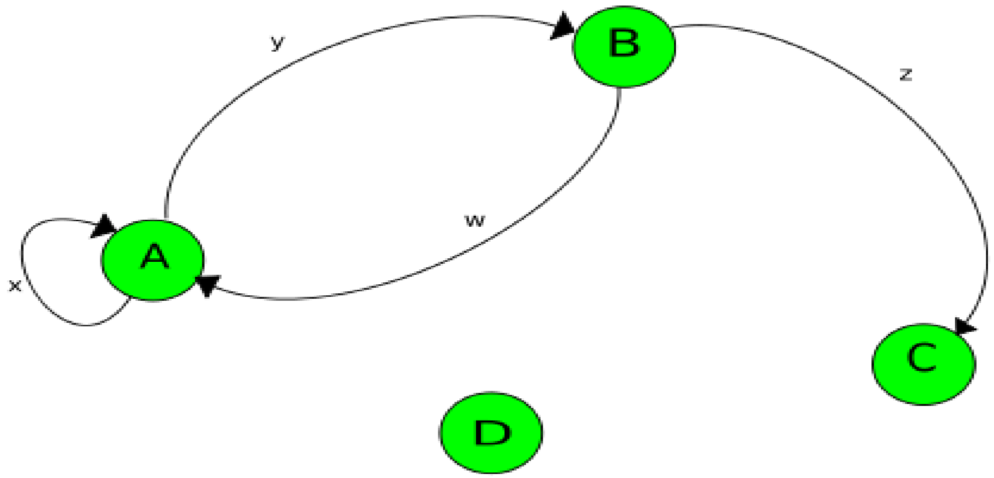
- Pseudograph

A graph which has loop is called pseudograph.



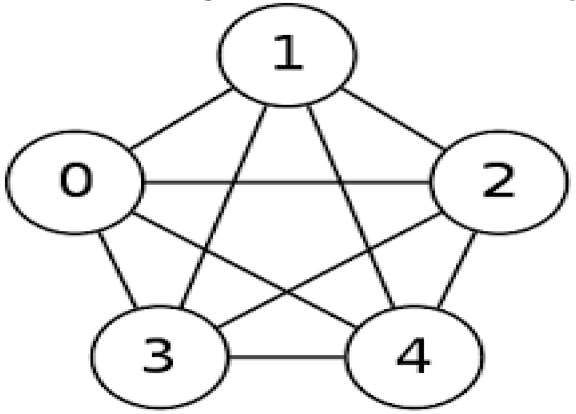
Directed graph

- A graphs G is called directed graph if each edge has a direction.



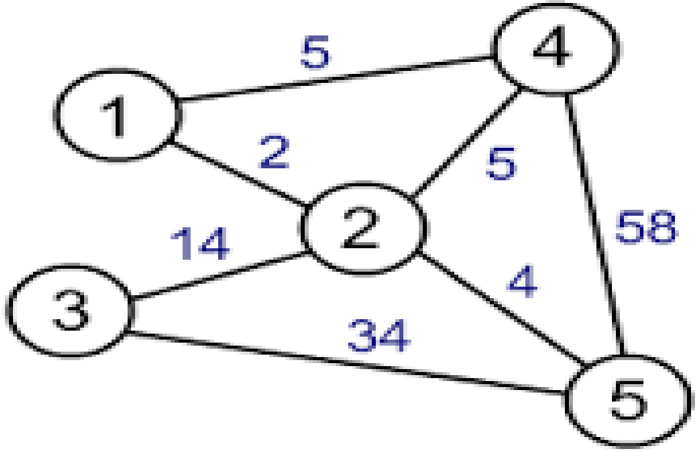
- **Complete graph**

  A graph G is called complete, if every nodes are adjacent with other node



- **Weighted graph**

  If each edge of graph is assigned a number or value, then it is weighted graph. The number is weight.



Why Use Graphs?

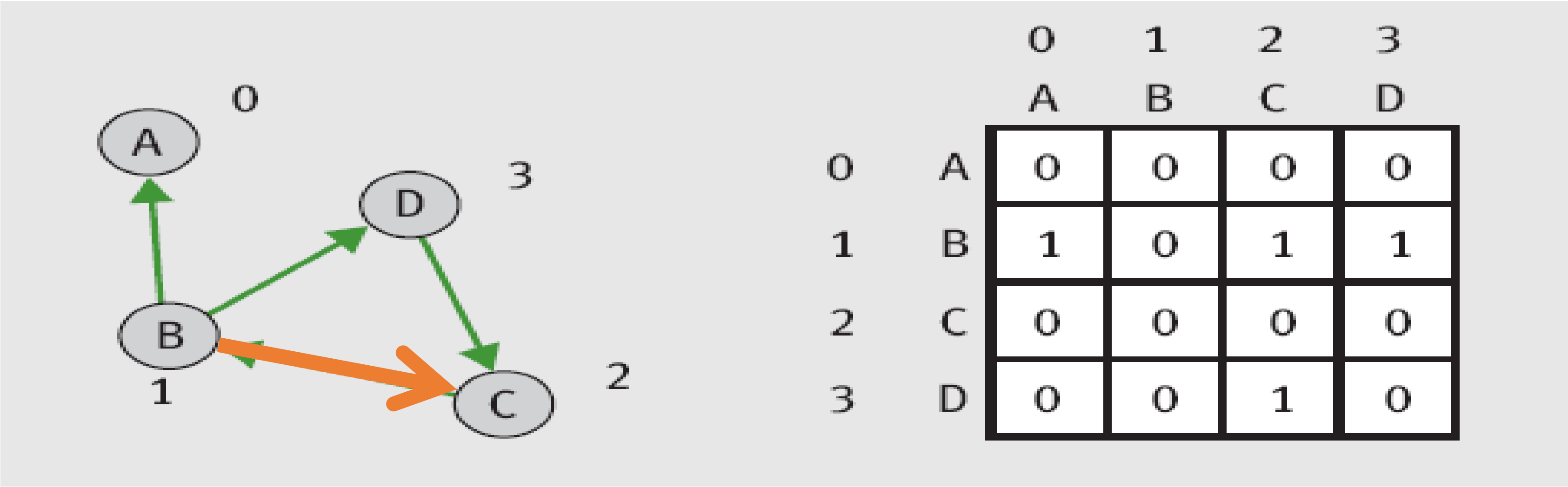
- Graphs serve as models of a wide range of objects:
  - A roadmap
  - A map of airline routes
  - A layout of an adventure game world
  - A schematic of the computers and connections that make up the Internet
  - The links between pages on the Web
  - The relationship between students and courses
  - A diagram of the flow capacities in a communications or transportation network

Representations of Graphs

- To represent graphs, you need a convenient way to store the vertices and the edges that connect them
- Two commonly used representations of graphs:
  - The **adjacency matrix**
  - The **adjacency list**

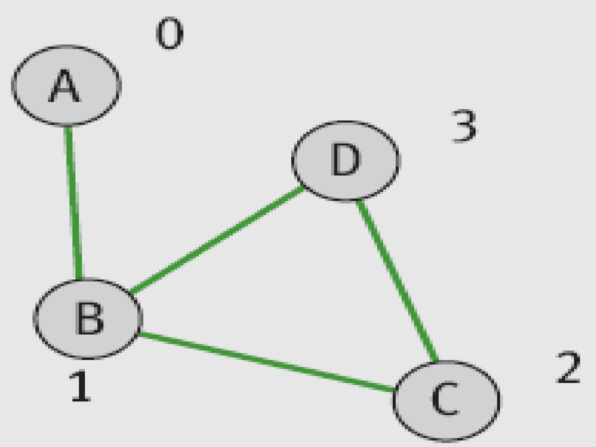
Adjacency Matrix

- If a graph has N vertices labeled 0, 1, . . . , $N-1$:
  - The adjacency matrix for the graph is a grid G with N rows and N columns
  - Cell $G[i][j] = 1$ if there's an edge from vertex i to j
    - Otherwise, there is no edge and that cell contains 0
- These are the simplest ways for representing graphs.
- Space requirement: $O(n^2)$
- Adding and deleting edge: O(1)
- Testing an edge : O(1)



Adjacency Matrix (continued)

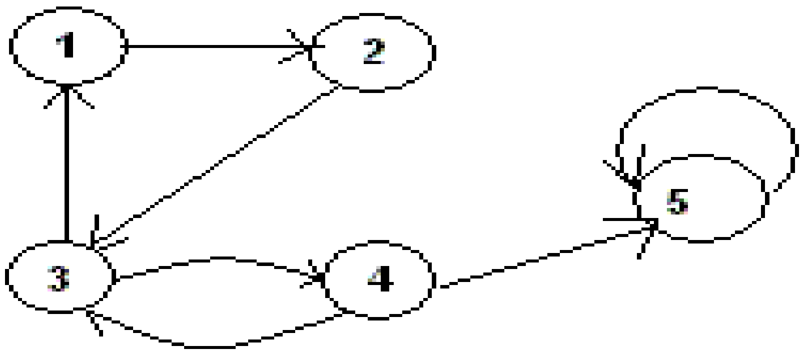
- If the graph is undirected, then four more cells are occupied by 1:
- If the vertices are labeled, then the labels can be stored in a separate one-dimensional array

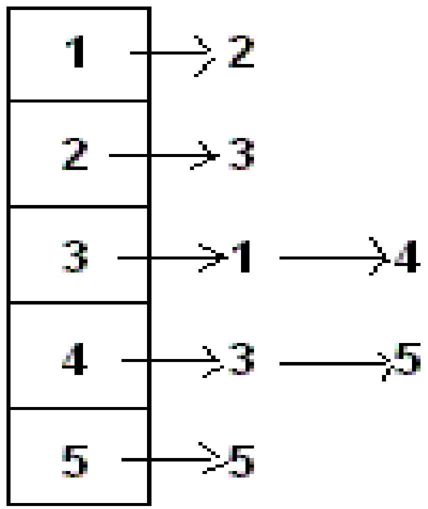


		0	1	2	3
		A	B	C	D
0	A	0	1	0	0
1	B	1	0	1	1
2	C	0	0	0	1
3	D	0	1	1	0

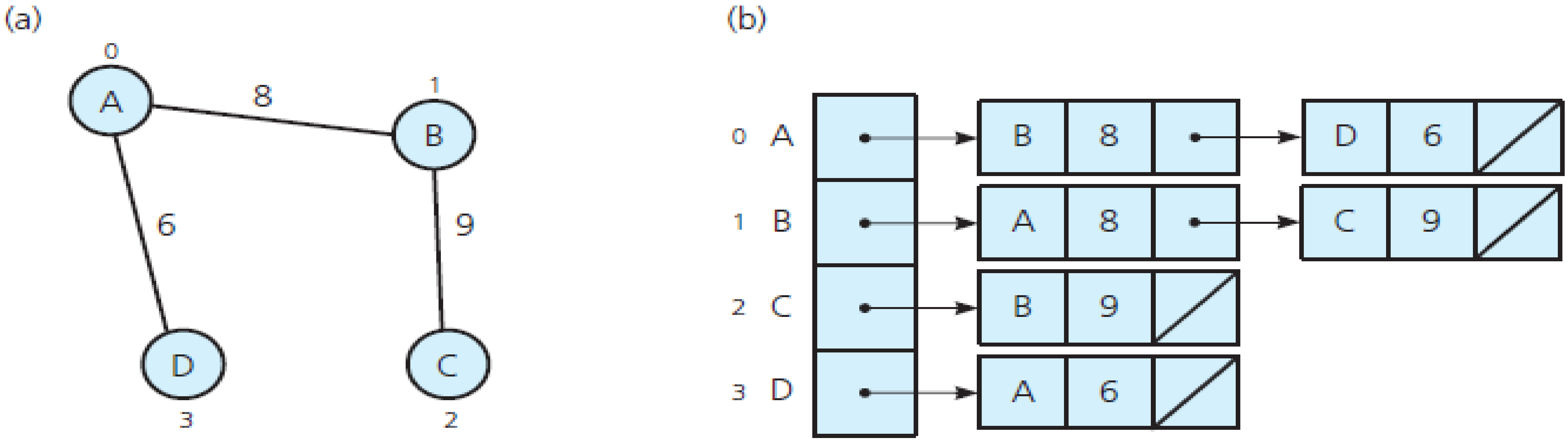
Adjacency List

- If a graph has N vertices labeled 0, 1, . . . , N – 1,
  - The adjacency list for the graph is an array of N linked lists
  - The i^{th} linked list contains a node for vertex j if and only if there is an edge from vertex i to vertex j
  - It is suitable for sparse graphs i.e. graphs with the few edges
  - Space required: O(V+E)
  - Time for
    - Testing edge to u O(deg(u))
    - Finding adjacent vertices: O(deg(u))
    - Insertion and deletion : O(deg(u))





Adjacency List (continued)



Graph Traversals

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:
  - Printing out the contents of each edge and vertex.
  - Counting the number of edges.
  - Identifying connected components of a graph.
- Graph traversal algorithms visit the vertices of a graph, according to some Strategy.
- Given $G=(V,E)$ and vertex v, find all $w \in V$, such that w connects v
  - ✓ Depth First Search (DFS): preorder traversal
  - ✓ Breadth First Search (BFS): level order traversal

DFS

The basic idea is:

- Start from the given vertex and go as far as possible
  i.e. search continues until the end of the path if not visited already,
- Otherwise, backtrack and try another path.
- DFS uses stack to process the nodes.

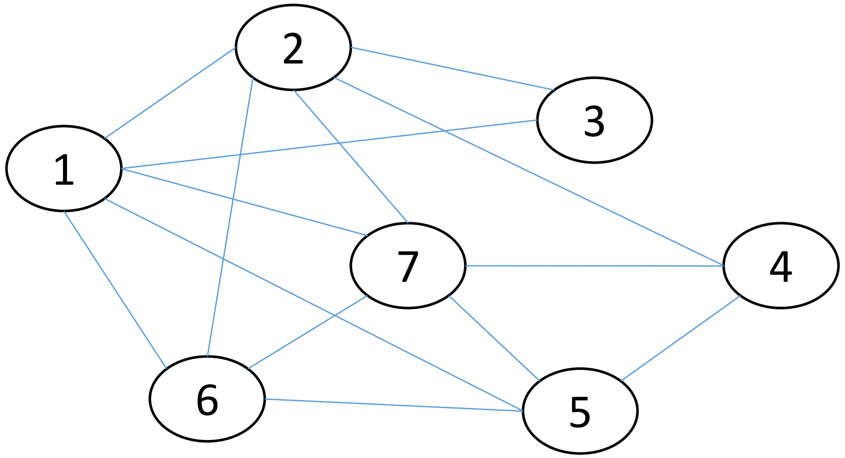
Algorithm:

```
DFS(G,S)
{
   T = {  S  };
   Traverse(S);
}
```

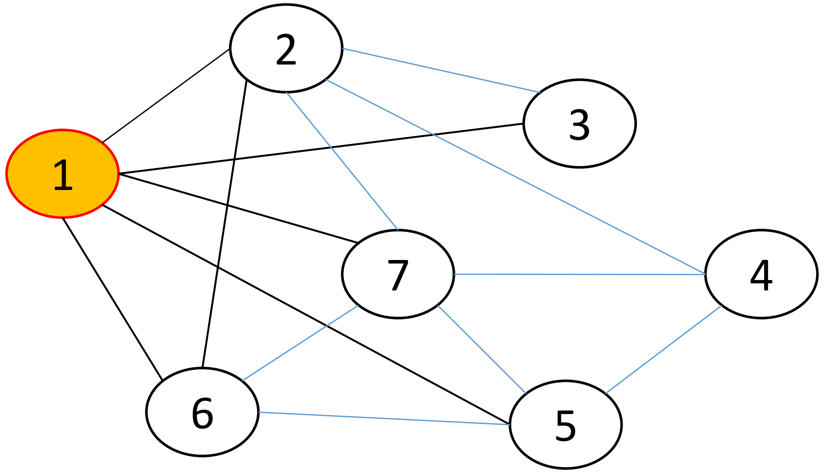
```
Traverse(v)
{
     for each w adjacent to v not yet in T
      {
            T = T U {w};   // add edge {v,w} in T
            Traverse(w);
      }
}
```

Example: DFS Tracing

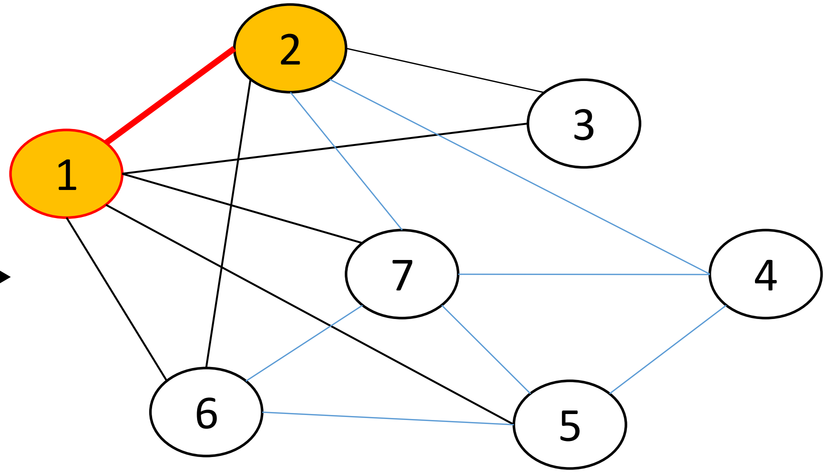
Starting Vertex : 1



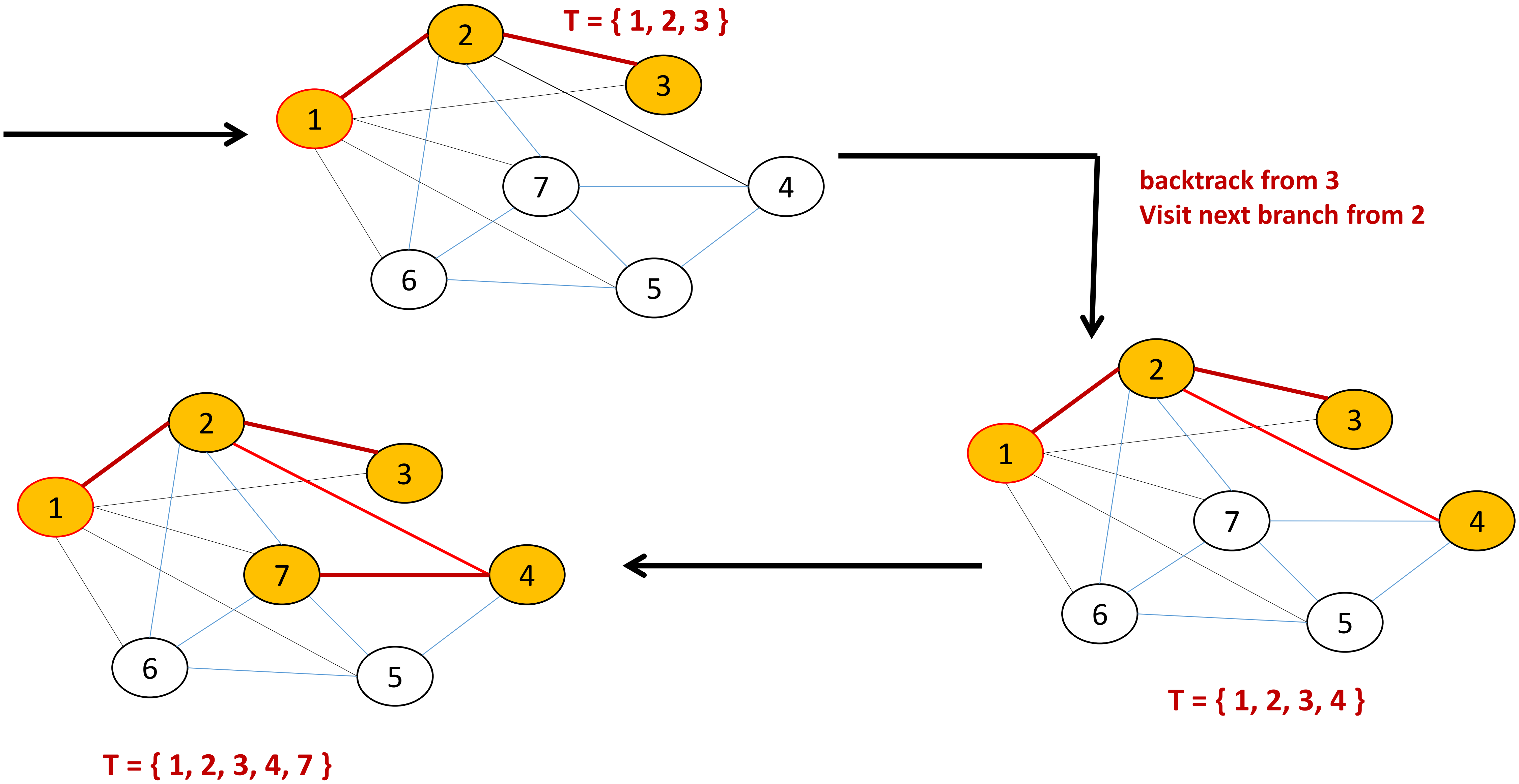
Visited Nodes: Implement Stack and list of visited nodes T = { }



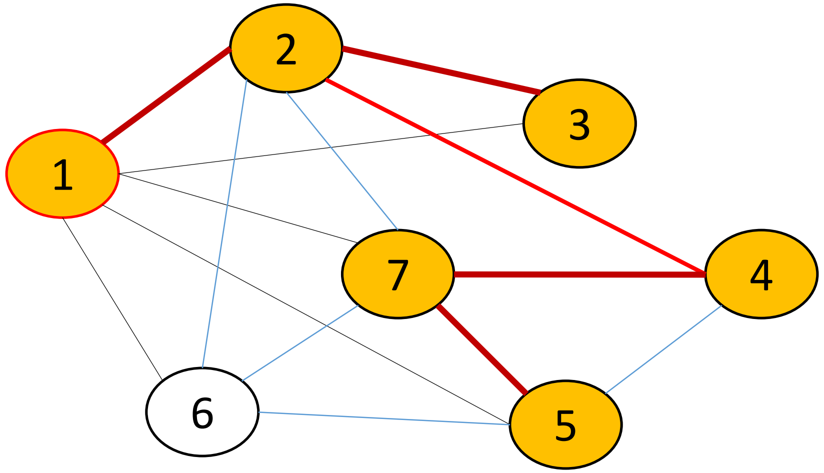
T = { 1 }



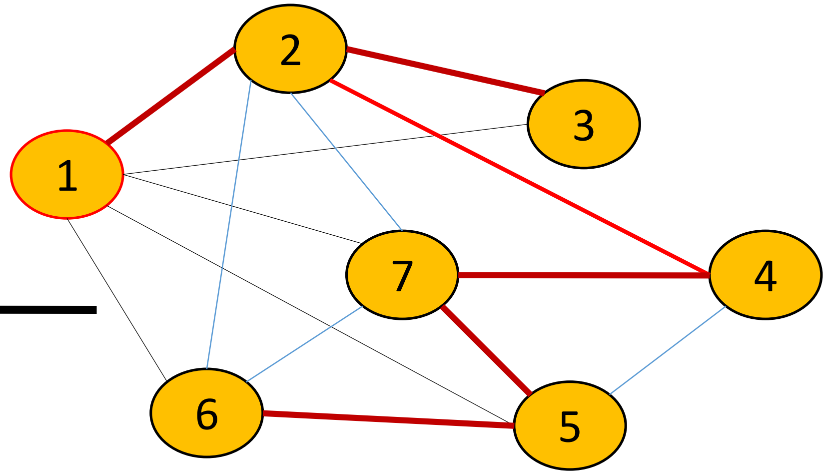
T = { 1, 2 }



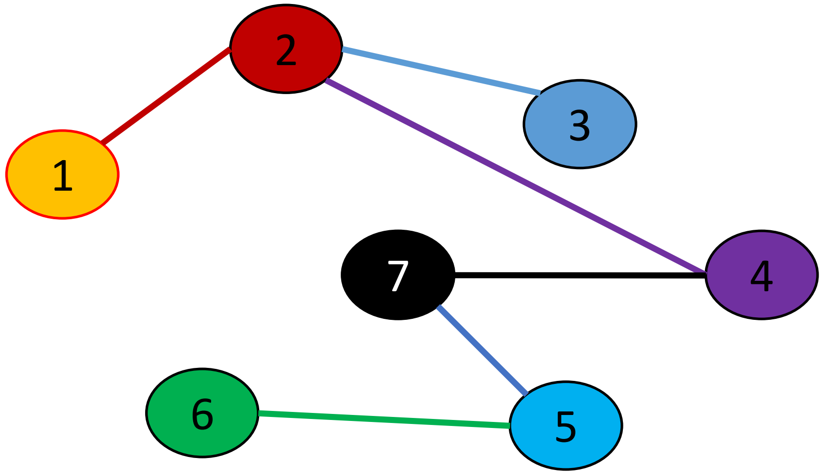
T = { 1, 2, 3 }
backtrack from 3
Visit next branch from 2
T = { 1, 2, 3, 4 }
T = { 1, 2, 3, 4, 7 }
1
2
3
4
5
6
7



T = { 1, 2, 3, 4, 7, 5 }



T = { 1, 2, 3, 4, 7, 5, 6 }



Final DFS tree.

Analysis:

- The complexity of the algorithm is greatly affected by Traverse function we can write its running time in terms of the relation,

$$T(n) = T(n-1) + O(n),$$

- At each recursive call a vertex is decreased and for each vertex atmost n adjacent vertices can be there. So, $O(n)$.
- Solving this we get, $T(n) = O(n^2)$. This is the case when we use adjacency matrix.
- If adjacency list is used, $T(n) = O(n + e)$, where e is number of edges.

BFS

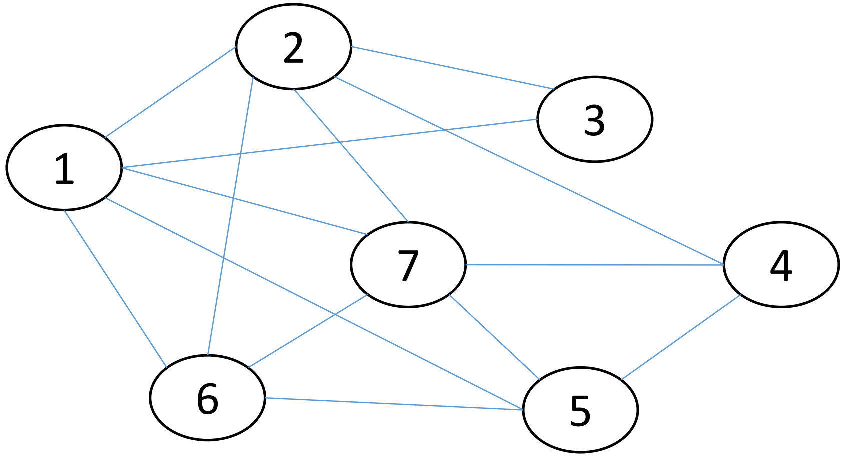
- This is one of the simplest methods of graph searching.
- Choose some vertex as a root or starting vertex.
- Add it to the queue. Mark it as visited and dequeue it from the queue.
- Add all the adjacent vertices of this node into queue.
- Remove one node from front and mark it as visited. Repeat this process until all the nodes are visited.

```
BFS (G, S)
{
    Initialize Queue, q = {  }
    mark S as visited;
    enqueue(q, S);
    while( q != Empty)
    {
        v = dequeue(q);
        for each w adjacent to v
          {
            if w is not marked as visited
             {
                    enqueue(q, w)
                    mark w as visited.
             }
          }
    }
}
```

Analysis:

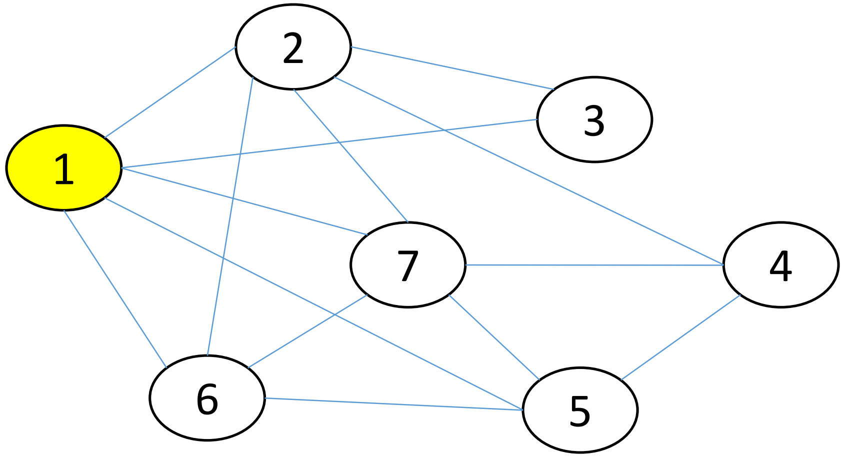
- This algorithms puts all the vertices in the queue and they are accessed one by one.
- for each accessed vertex from the queue their adjacent vertices are looked up for O(n) time (for worst case).
- Total time complexity , $T(n) = O(n^2)$, in case of adjacency matrix.
- $T(n) = O(n + e)$, in case of adjacency list.

Example: BFS Algorithm Tracing



Solution:

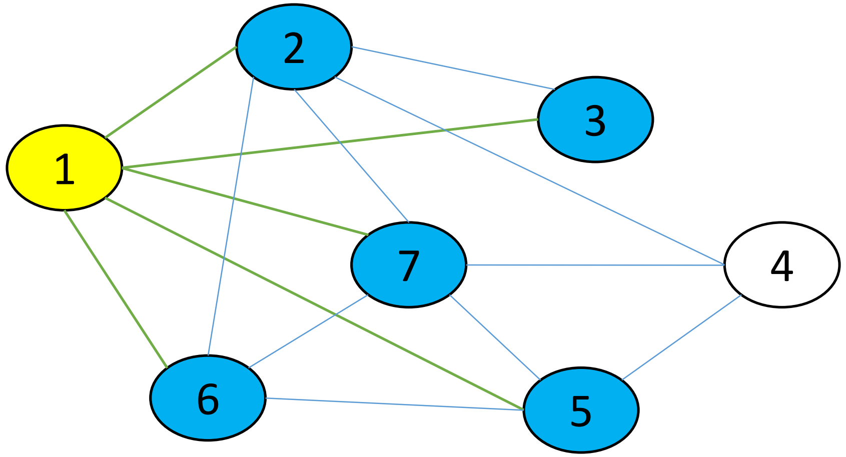
- ➢ **Starting vertex : 1**



Queue :

1						

Visited: { 1 }

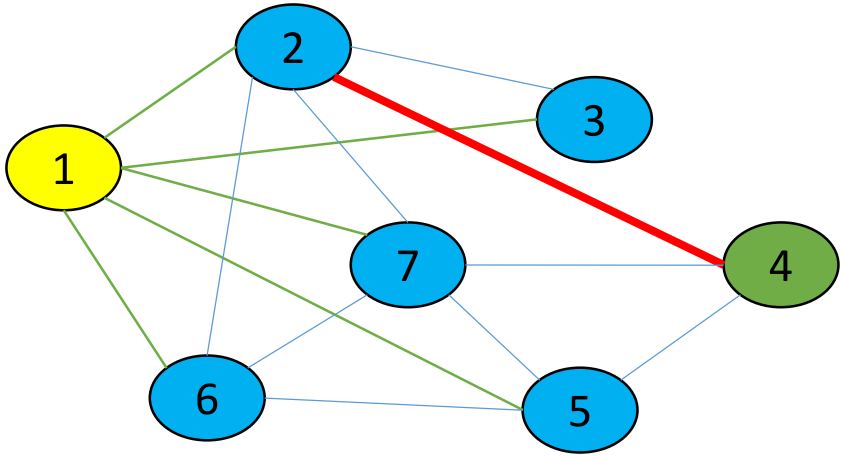


Queue :

2	3	7	5	6		

Visited: { 1,2, 3, 7, 5, 6 }

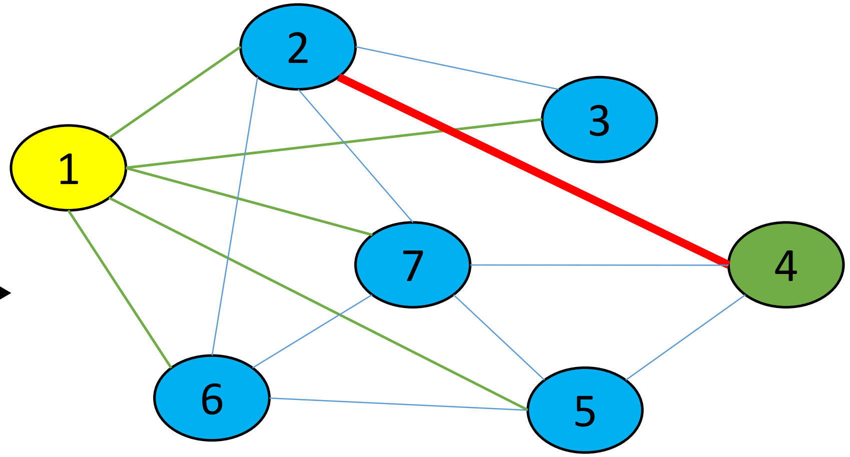
- Dequeue front of queue. Add its all unvisited adjacent nodes to the queue.



Queue

	3	7	5	6	4	

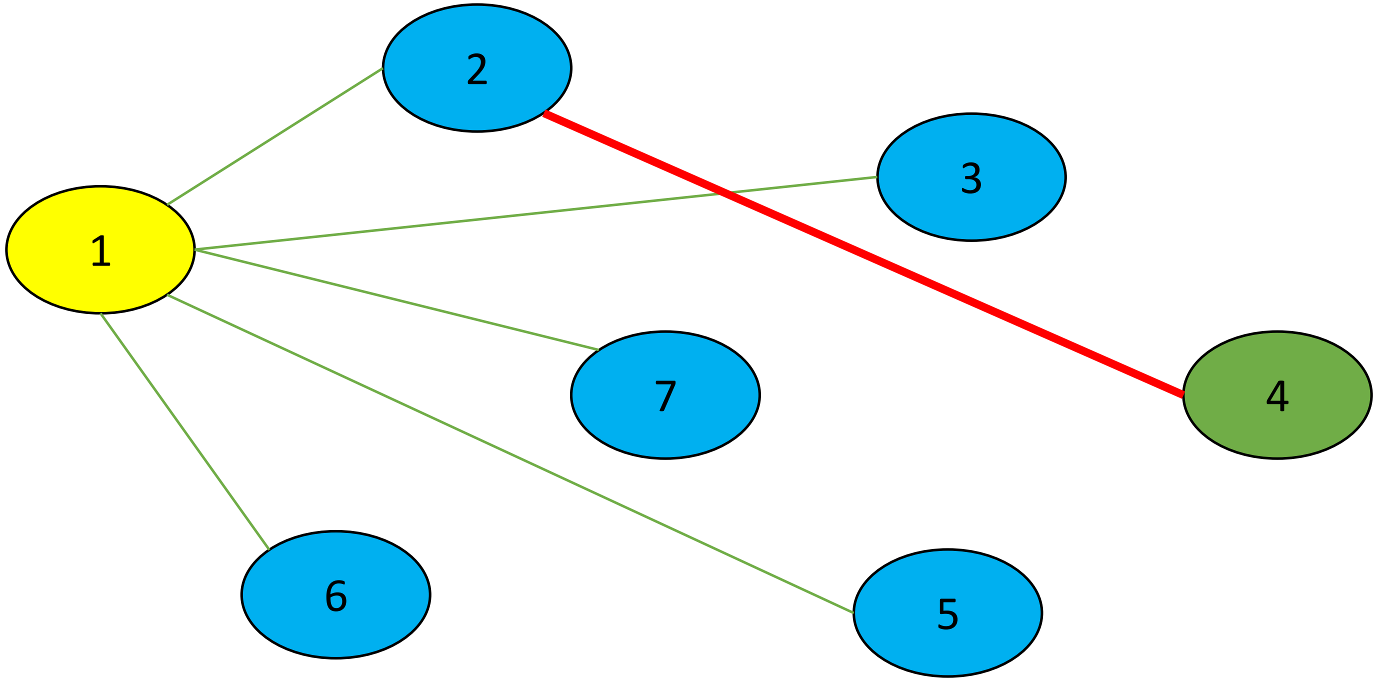
Visited : { 1, 2, 3, 7, 5, 6, 4 }



Queue

Visited : { 1, 2, 3, 7, 5, 6, 4 }

Final BFS Tree.



Spanning Tree

A spanning tree of a connected undirected graph G is a sub graph T of without cycle G that connects all the vertices of G.
i.e. A spanning tree for a connected graph G is a tree containing all the vertices of G

Minimum Spanning Trees

- A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.
- It represents the cheapest way of connecting all the nodes in G.
- It is not necessarily unique.
- Any time you want to visit all vertices in a graph at minimum cost (e.g., wire routing on printed circuit boards, sewer pipe layout, road planning…)

MST Generating Algorithms

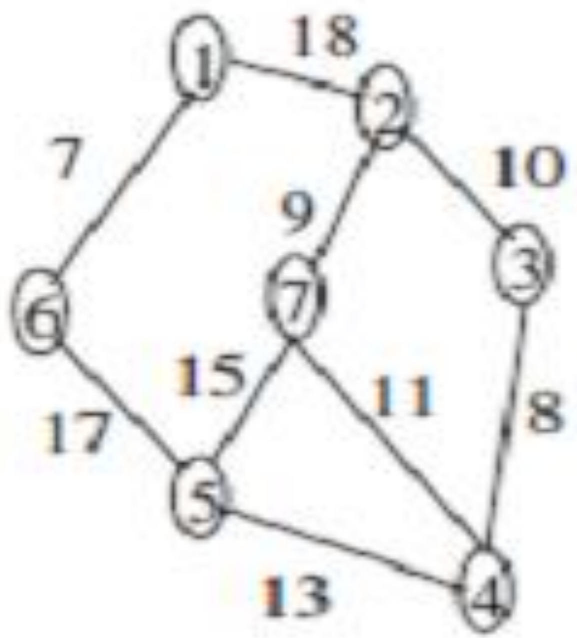
- Two algorithms that are used to construct the minimum spanning tree from the given connected weighted graph of given graph are:
  - Kruskal's Algorithm
  - Prim's Algorithm

Kruskal's Algorithm

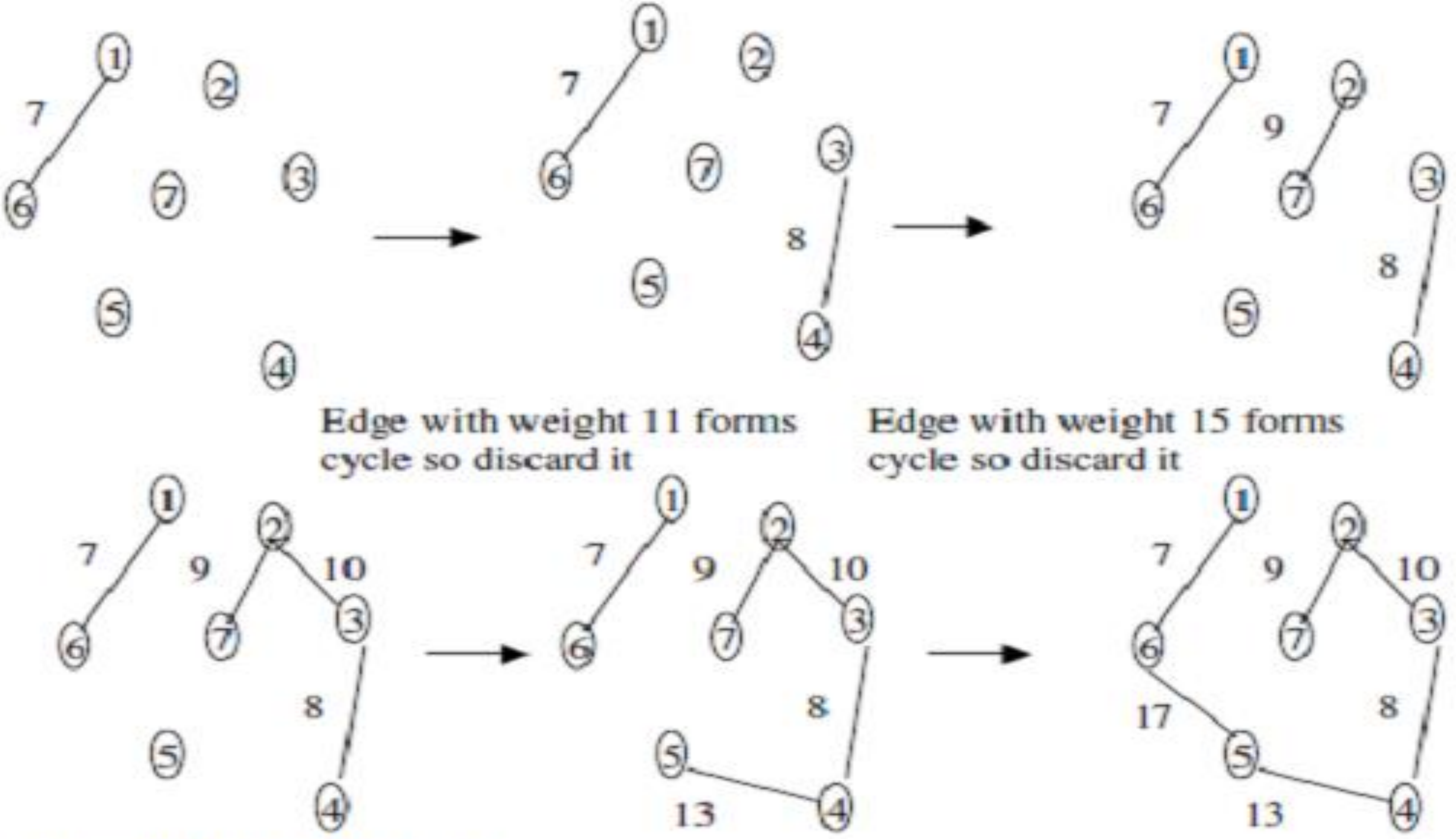
We have V as a set of n vertices and E as set of edges of graph G. The idea behind this algorithm is:

- The nodes of the graph are considered as n distinct partial trees with one node each.
- At each step of the algorithm, two partial trees are connected into single partial tree by an edge of the graph.
- While connecting two nodes of partial trees, minimum weighted arc is selected.
- After n-1 steps MST is obtained.

Example:
Find the MST and its weight of the graph.



Solution:



The total weight of MST is 64.

Algorithm:

```
Kruskal_MSt( G )
{
        T = { V }   // forest of n nodes
         S = Set of edges sorted in non-decreasing order of weight
        while( |T| < n-1  AND  S != Empty)
        {
                select edge (u,v) from S in order
                S = S – (u,v)
                if (u,v) does not form cycle in T
                        T = T U {(u,v)}
        }
}
```

Complexity Analysis

To form the forest of n trees takes O(n) time, the creation of S takes O(E.log E) time and while loop executes O(n) time and the steps inside loop take almost linear time.
So, total time – O(n) + O (E logE) +O(n log n)

Exercise: Trace Kruskals algorithm.