

## 9 Graphs

- A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

- **Why are Graphs Useful?**

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- Family trees in which the member nodes have an edge from parent to each of their children.
- Transportation networks in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

- Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

- **Definition**

A graph  $G$  is defined as an ordered set  $(V, E)$ , where  $V(G)$  represents the set of vertices and  $E(G)$  represents the edges that connect these vertices.

- Figure 9.1 shows a graph with  $V(G) = \{A, B, C, D \text{ and } E\}$  and  $E(G) = \{(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)\}$ . Note that there are five vertices or nodes and six edges in the graph.

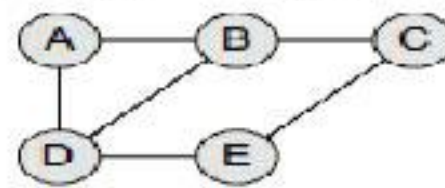


Figure 9-1: Undirected Graph

- A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 9.1 shows an undirected graph because it does not give any information about the direction of the edges.

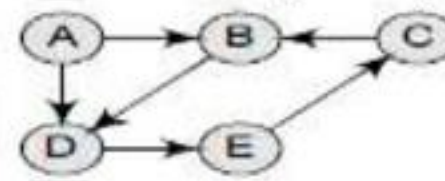


Figure 9-2: Directed Graph

- Look at Figure 9.2 which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge  $(A, B)$  is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

- **Graph Terminology**

- **Adjacent nodes** or neighbours For every edge,  $e = (u, v)$  that connects nodes  $u$  and  $v$ , the nodes  $u$  and  $v$  are the end-points and are said to be the adjacent nodes or neighbours.
- **Degree of a node** Degree of a node  $u$ ,  $\deg(u)$ , is the total number of edges containing the node  $u$ . If  $\deg(u) = 0$ , it means that  $u$  does not belong to any edge and such a node is known as an isolated node.
- **Regular graph** It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree  $k$  is called a  $k$ -regular graph or a regular graph of degree  $k$ . Figure 9.3 shows regular graphs.

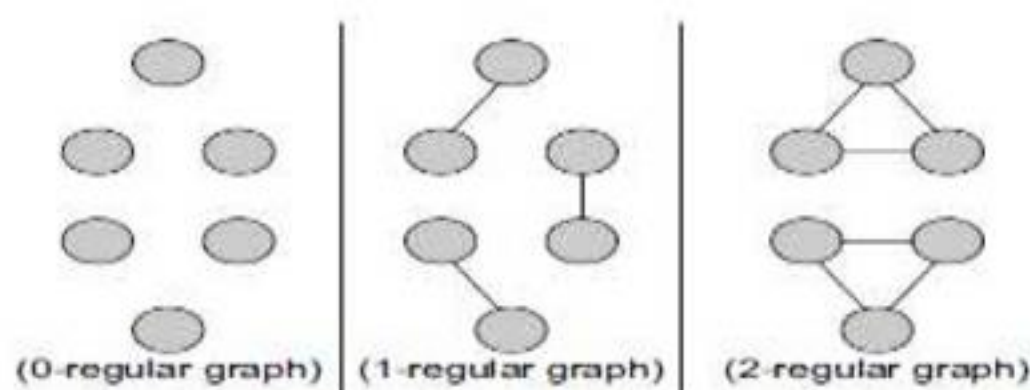


Figure 9-3: Regular graphs



- **Path** A path  $P$  written as  $P = \{v_0, v_1, v_2, \dots, v_n\}$ , of length  $n$  from a node  $u$  to  $v$  is defined as a sequence of  $(n+1)$  nodes. Here,  $u = v_0$ ,  $v = v_n$  and  $v_{i-1}$  is adjacent to  $v_i$  for  $i = 1, 2, 3, \dots, n$ .
- **Closed path** A path  $P$  is known as a closed path if the edge has the same end-points. That is, if  $v_0 = v_n$ .
- **Simple path** A path  $P$  is known as a simple path if all the nodes in the path are distinct with an exception that  $v_0$  may be equal to  $v_n$ . If  $v_0 = v_n$ , then the path is called a closed simple path.
- **Cycle** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).
- **Connected graph** A graph is said to be connected if for any two vertices  $(u, v)$  in  $V$  there is a path from  $u$  to  $v$ . That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a **tree**. Therefore, a tree is treated as a special graph (Refer Fig. 9.4(b)).

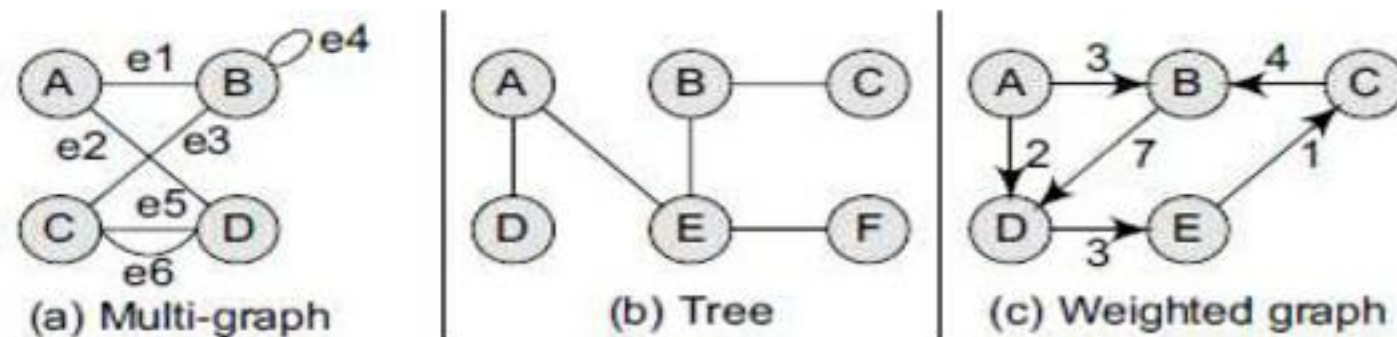


Figure 9-4: Multi-graph, tree, and weighted graph

- **Complete graph** A graph  $G$  is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has  $n(n-1)/2$  edges, where  $n$  is the number of nodes in  $G$ .
- **Clique** In an undirected graph  $G = (V, E)$ , clique is a subset of the vertex set  $C \subseteq V$ , such that for every two vertices in  $C$ , there is an edge that connects two vertices.
- **Labelled graph or weighted graph** A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by  $w(e)$  is a positive value which indicates the cost of traversing the edge. Figure 9.4(c) shows a weighted graph.
- **Multiple edges** Distinct edges which connect the same end-points are called multiple edges. That is,  $e = (u, v)$  and  $e' = (u, v)$  are known as multiple edges of  $G$ .
- **Loop** An edge that has identical end-points is called a loop. That is,  $e = (u, u)$ .
- **Multi-graph** A graph with multiple edges and/or loops is called a multi-graph. Figure 9.4(a) shows a multi-graph.
- **Size of a graph** The size of a graph is the total number of edges in it.

## 9.1 Representations of Graph

Graph is a mathematical structure and finds its application in many areas of interest in which problem need to be solved using computer. Thus, this mathematical structure must be represented in some kind of data structure. There are three common ways of storing graphs in the computer's memory. They are:

- Sequential representation by using an **adjacency matrix**.
- Linked representation by using an **adjacency list** that stores the neighbours of a node using a linked list.
- **Adjacency multi-list** which is an extension of linked representation.

### 9.1.1 Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph  $G$ , if node  $v$  is adjacent to node  $u$ , then there is definitely an edge from  $u$  to  $v$ . That is, if  $v$  is adjacent to  $u$ , we can get from  $u$  to  $v$  by traversing one edge. For any graph  $G$  having  $n$  nodes, the adjacency matrix will have the dimension of  $n \times n$ .

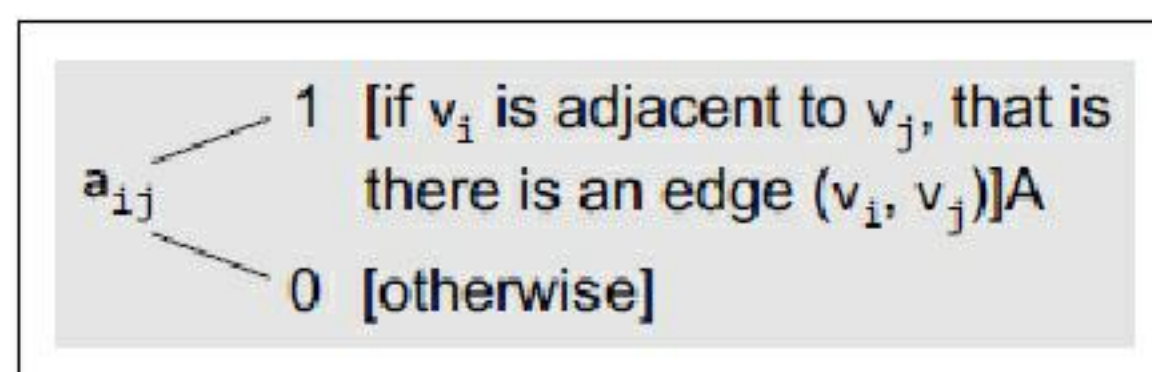


Figure 9-5: Adjacency matrix entry



In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry  $a_{ij}$  in the adjacency matrix will contain 1, if vertices  $v_i$  and  $v_j$  are adjacent to each other. However, if the nodes are not adjacent,  $a_{ij}$  will be set to zero. It is summarized in Figure 9.5 Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in  $G$ . Therefore, a change in the order of nodes will result in a different adjacency matrix. Figure 9.6 shows some graphs and their corresponding adjacency matrices.

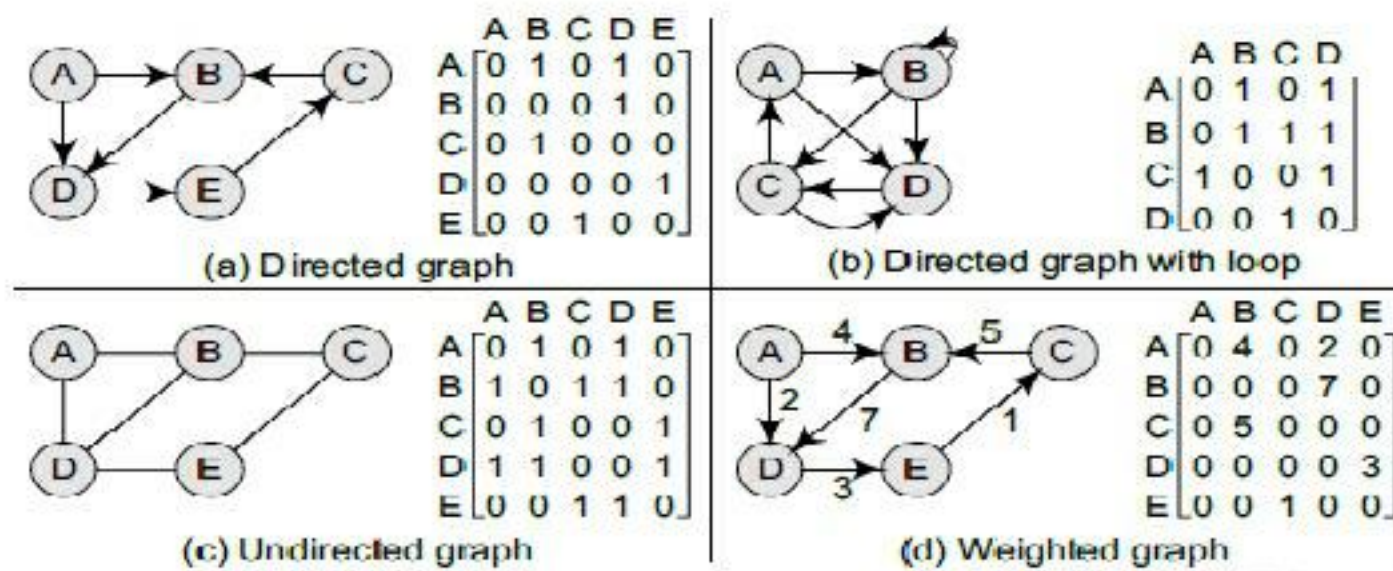


Figure 9-6: Graphs and their corresponding adjacency matrices

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is  $O(n^2)$ , where  $n$  is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix. From adjacency matrix  $A^1$ , we can conclude that an entry 1 in the  $i$ th row and  $j$ th column means that there exists a path of length 1 from  $v_i$  to  $v_j$ . Now consider,  $A^2$ ,  $A^3$ , and  $A^4$ .

$$(a_{ij})^2 = \sum a_{ik} a_{kj}$$

Any entry  $a_{ij} = 1$  if  $a_{ik} = a_{kj} = 1$ . That is, if there is an edge  $(v_i, v_k)$  and  $(v_k, v_j)$ , then there is a path from  $v_i$  to  $v_j$  of length 2.

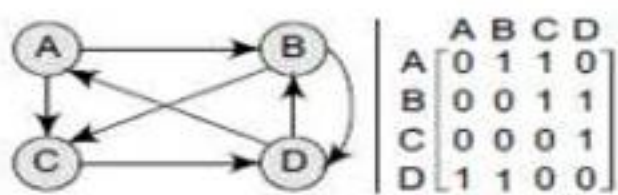


Figure 9.7 Directed graph with its adjacency

Similarly, every entry in the  $i$ th row and  $j$ th column of  $A^3$  gives the number of paths of length 3 from node  $v_i$  to  $v_j$ .

In general terms, we can conclude that every entry in the  $i$ th row and  $j$ th column of  $A^n$  (where  $n$  is the number of nodes in the graph) gives the number of paths of length  $n$  from node  $v_i$  to  $v_j$ . Consider a directed graph given in Figure 9.7 Given its adjacency matrix  $A$ , let us calculate  $A^2$ ,  $A^3$ , and  $A^4$ .

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$



Now, based on the above calculations, we define matrix  $B$  as:

$$B^r = A^1 + A^2 + A^3 + \dots + A^r$$

An entry in the  $i$ th row and  $j$ th column of matrix  $B^r$  gives the number of paths of length  $r$  or less than  $r$  from vertex  $v_i$  to  $v_j$ . The main goal to define matrix  $B$  is to obtain the path matrix  $P$ . The path matrix  $P$  can be calculated from  $B$  by setting an entry  $P_{ij} = 1$ , if  $B_{ij}$  is non-zero and  $P_{ij} = 0$ ,

$P_{ij} \rightarrow 1$  [if there is a path from  $v_i$  to  $v_j$ ]  
 $P_{ij} \rightarrow 0$  [otherwise]

if otherwise. The path matrix is used to show whether there exists a simple path from node  $v_i$  to  $v_j$  or not. This is shown in Figure 9.8

Let us now calculate matrix  $B$  and matrix  $P$  using the above discussion.

Figure 9.8: Path matrix entry

$$B = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 6 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 3 & 3 & 5 \\ 6 & 8 & 7 & 8 \end{bmatrix}$$

Now the path matrix  $P$  can be given as:

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In this representation,  $n^2$  memory location is required to represent a graph with  $n$  vertices. The adjacency matrix is a simple way to represent a graph, but it has two disadvantages.

- It takes  $n^2$  space to represent a graph with  $n$  vertices, even for a sparse graph and
- It takes  $O(n^2)$  time to solve the graph problem.

### 9.1.2 Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in  $G$ . Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in  $G$  is easy and straightforward when  $G$  is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Consider the graph given in Figure 9.9 and see how its adjacency list is stored in the memory. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in  $G$ . However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in  $G$  because an edge  $(u, v)$  means an edge from node  $u$  to  $v$  as well as an edge from  $v$  to  $u$ . Adjacency lists can also be modified to store weighted graphs. Let us now see an adjacency list for an undirected graph as well as a weighted graph. This is shown in Figure 9.10.

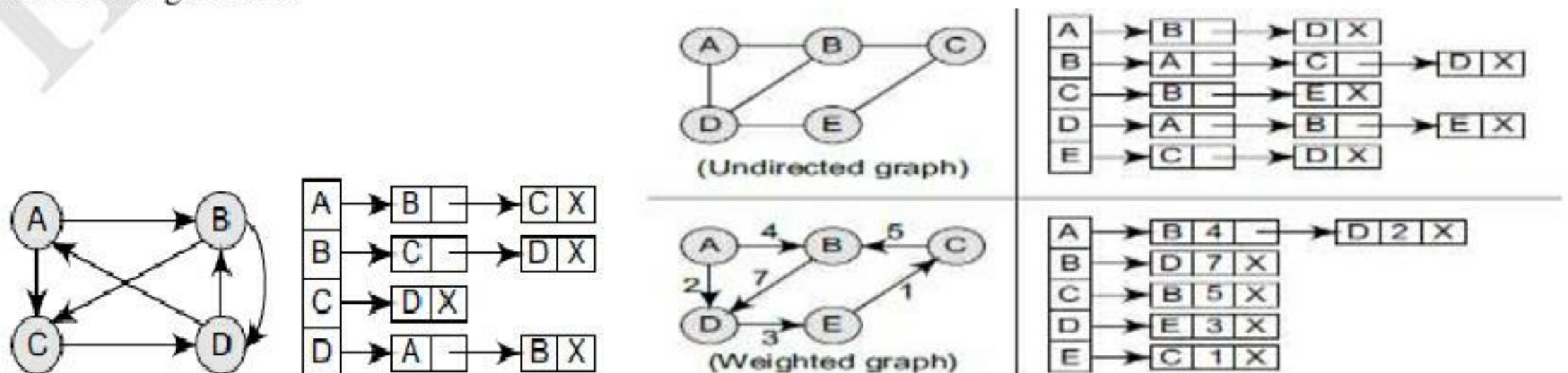


Figure 9-9: Graph  $G$  and its adjacency list      Figure 9-10: Adjacency list for an undirected graph and a weighted graph



### 9.1.3 Adjacency Multi-list Representation

Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs. A multi-list representation basically consists of two parts—a **directory of nodes' information** and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node  $i$  points to the adjacency list for node  $i$ . This means that the nodes are shared among several lists.

In a multi-list representation, the information about an edge  $(v_i, v_j)$  of an undirected graph can be stored using the following attributes:

$m$ : A single bit field to indicate whether the edge has been examined or not.

$v_i$ : A vertex in the graph that is connected to vertex  $v_j$  by an edge.

$v_j$ : A vertex in the graph that is connected to vertex  $v_i$  by an edge.

Link  $i$  for  $v_i$ : A link that points to another node that has an edge incident on  $v_i$ .

Link  $j$  for  $v_j$ : A link that points to another node that has an edge incident on  $v_j$ .

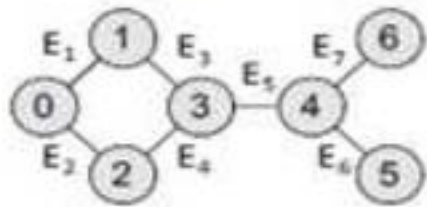


Figure 9.11: Undirected graph

Consider the undirected graph given in Figure 9.11.

The adjacency multi-list for the graph can be given as:

Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL

Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7



## 9.2 Implementation of graph in C

// Write a program to create a graph of **n** vertices using an **adjacency list**. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct node
{
    char vertex;
    struct node *next;
};
struct node *gnode;
void createEmptyGraph(struct node *Adj[], int no_of_nodes);
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void inputGraph(struct node *adj[], int no_of_nodes);
int main()
{
    struct node *Adj[10];
    int i, no_of_nodes;
    printf("\n Enter the number of nodes in G: ");
    scanf("%d", &no_of_nodes);
    createEmptyGraph(Adj, no_of_nodes);
    inputGraph(Adj, no_of_nodes);
    printf("\n The graph is: ");
    displayGraph(Adj, no_of_nodes);
    deleteGraph(Adj, no_of_nodes);
    getch();
    return 0;
}
void createEmptyGraph(struct node *Adj[], int no_of_nodes)
{
    int i;
    for(i = 0; i < no_of_nodes; i++)
        Adj[i] = NULL;
}
void inputGraph(struct node *Adj[], int no_of_nodes)
{
    struct node *new_node, *last;
    int i, j, n, val;
    for(i = 0; i < no_of_nodes; i++)
    {
        last = NULL;
        printf("\n Enter the number of neighbours of %d: ", i);
        scanf("%d", &n);
        for(j = 0; j < n; j++)
        {
            printf("\n Enter the neighbour %d of %d: ", j+1, i);
            scanf("%d", &val);
            new_node = (struct node *)malloc(sizeof(struct node));
            new_node->vertex = val;
            new_node->next = NULL;
            if (Adj[i] == NULL)
```

```

        Adj[i] = last=new_node;
    else
    {
        last -> next = new_node;
        last = new_node;
    }
}
}
}
void displayGraph (struct node *Adj[], int no_of_nodes)
{
    struct node *ptr;
    int i;
    for(i = 0; i < no_of_nodes; i++)
    {
        ptr = Adj[i];
        printf("\n The neighbours of node %d are:", i);
        while(ptr != NULL)
        {
            printf("\t%d", ptr -> vertex);
            ptr = ptr -> next;
        }
    }
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{
    int i;
    struct node *temp, *ptr;
    for(i = 0; i < no_of_nodes; i++)
    {
        ptr = Adj[i];
        while(ptr != NULL)
        {
            temp = ptr;
            ptr = ptr -> next;
            free(temp);
        }
        Adj[i] = NULL;
    }
}

```

```

Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 1
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0
Enter the neighbour 2 of 1: 2
Enter the number of neighbours of 2: 1
Enter the neighbour 1 of 2: 1

The graph is:
The neighbours of node 0 are: 1
The neighbours of node 1 are: 0      2
The neighbours of node 2 are: 1

```



### 9.3 Graph Traversals

Many application of graph requires a structured system to examine the vertices and edges of a graph G. That is a graph traversal, which means visiting all the nodes of the graph. There are two graph traversal methods.

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

#### 9.3.1 Breadth First Search

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

```

Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
  
```

Figure: Algorithm for breadth-first search

#### Analysis

From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in  $O(n)$  time (for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e.  $n$ , produce complexity of an algorithm as  $O(n^2)$ .

**Example:** Consider the graph G given in Figure below. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

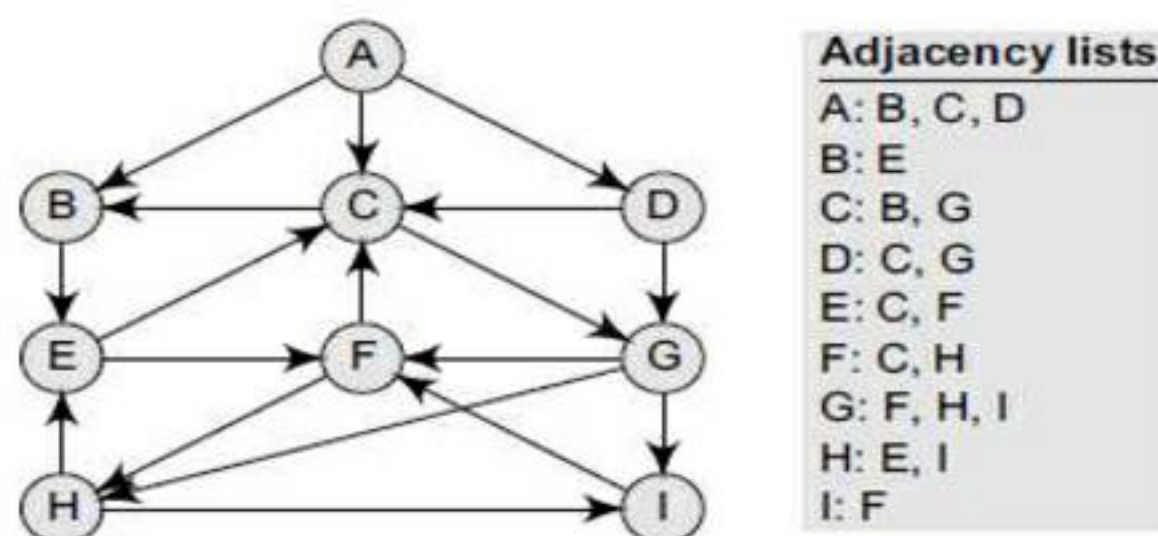


Figure: Graph G and its adjacency list



**Solution**

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = -1. The algorithm for this is as follows:

- (a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

- (b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

- (c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

- (d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

- (e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

- (f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

- (g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A → C → G → I.



### ✚ **Features of Breadth-First Search Algorithm**

**Space complexity:** In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor  $b$  (number of children at each node) and depth  $d$ , the asymptotic space complexity is the number of nodes at the deepest level  $O(b^d)$ . If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as  $O(|E| + |V|)$ , where  $|E|$  is the total number of edges in  $G$  and  $|V|$  is the number of nodes or vertices.

**Time complexity** In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches  $O(b^d)$ . However, the time complexity can also be expressed as  $O(|E| + |V|)$ , since every vertex and every edge will be explored in the worst case.

**Completeness** Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

**Optimality** Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

### ✚ **Applications of Breadth-First Search Algorithm**

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph  $G$ .
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes,  $u$  and  $v$ , of an unweighted graph.
- Finding the shortest path between two nodes,  $u$  and  $v$ , of a weighted graph.

### 9.3.2 Depth First Search

- The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of  $G$  and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node  $A$  which becomes the current node. Then, it examines each node  $N$  along a path  $P$  which begins at  $A$ . That is, we process a neighbor of  $A$ , then a neighbour of neighbour of  $A$ , and so on. During the execution of the algorithm, if we reach a path that has a node  $N$  that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- The algorithm proceeds like this until we reach a dead-end (end of path  $P$ ). On reaching the deadend, we backtrack to find another path  $P'$ . The algorithm terminates when backtracking leads back to the starting node  $A$ . In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges.
- Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

```

Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:  Pop the top node N. Process it and set its
        STATUS = 3 (processed state)
Step 5:  Push on the stack all the neighbours of N that
        are in the ready state (whose STATUS = 1) and
        set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
  
```

Figure: Algorithm for depth-first search



### Analysis:

The complexity of the algorithm is greatly affected by Traverse function we can write its running time in terms of the relation  $T(n) = T(n-1) + O(n)$ , here  $O(n)$  is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is  $O(n^2)$ .

**Example:** Consider the graph G given in Figure The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

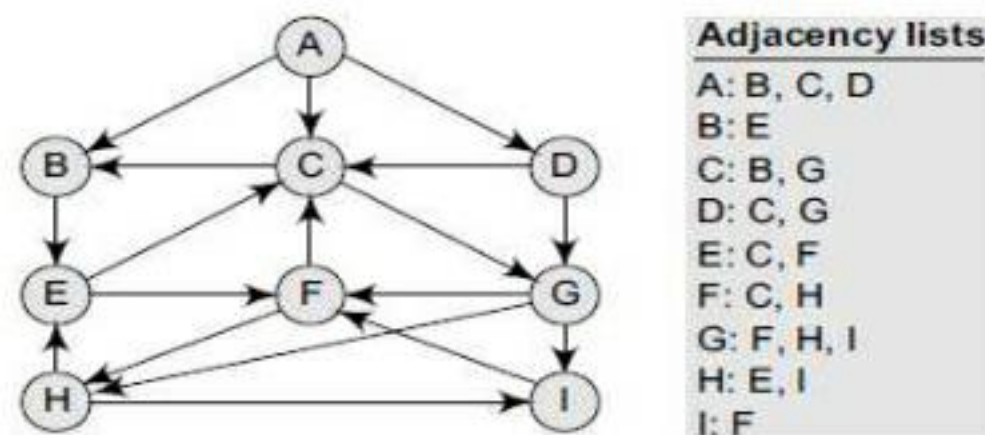


Figure: Graph G and its adjacency list

### Solution

- (a) Push H onto the stack.

STACK: H

- (b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H

STACK: E, I

- (c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I

STACK: E, F

- (d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F

STACK: E, C

- (e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C

STACK: E, B, G

- (f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G

STACK: E, B

- (g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B

STACK: E

- (h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E

STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

**These are the nodes which are reachable from the node H.**



### ✚ Features of Depth-First Search Algorithm

**Space complexity** The space complexity of a depth-first search is lower than that of a breadth first search.

**Time complexity** The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as  $O(|V| + |E|)$ .

**Completeness** Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

### ✚ Applications of Depth-First Search Algorithm

Depth-first search is useful for:

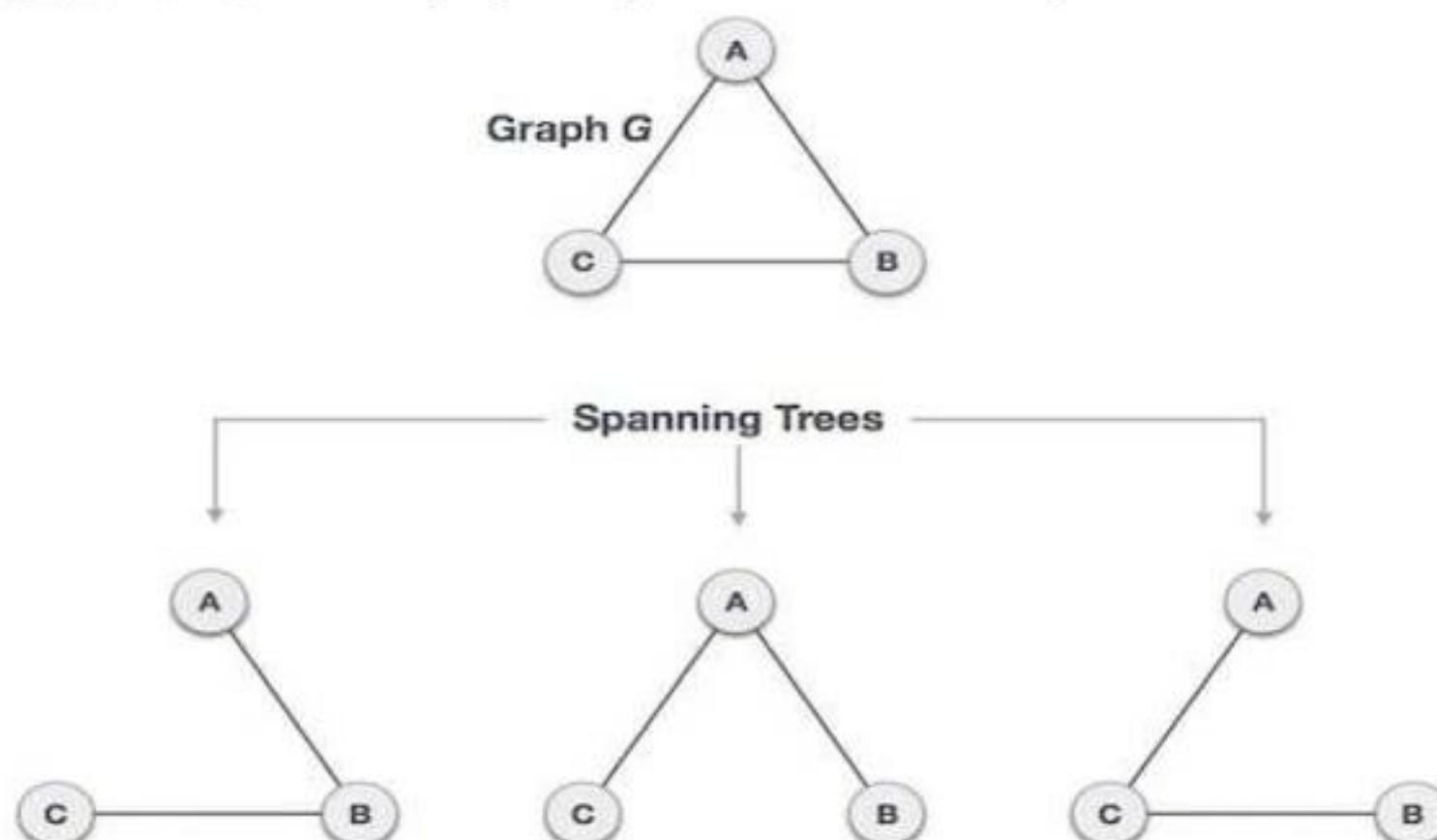
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

### ❖ Application of Graph Structures

- Graph is an important data structure whose extensive applications are known in almost all application areas. Nowadays, many applications related with computation can be managed efficiently with graph structures.
- It is not possible to discuss all the problems where graph structures are involved because this domain is very large; however, we limit discussions with a few important of them which play significant roles in various applications.
- In order to maintain the generality, we will discuss them as graph theoretic problems. The major graph theoretic problems are:
  - Shortest path problem
  - Topological sorting of a graph
  - Spanning trees
  - Connectivity of a graph
  - Euler's path and Hamiltonian path
  - Binary decision diagram

## 9.4 Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.
- By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



- We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n-2$  number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence  $3-2 = 3$  spanning trees are possible.



**General Properties of Spanning Tree**

- We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G.
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

**Mathematical Properties of Spanning Tree**

- Spanning tree has  $n-1$  edges, where  $n$  is the number of nodes (vertices).
- From a complete graph, by removing maximum  $e - n + 1$  edges, we can construct a spanning tree.
- A complete graph can have maximum  $n^{n-2}$  number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

**Application of Spanning Tree**

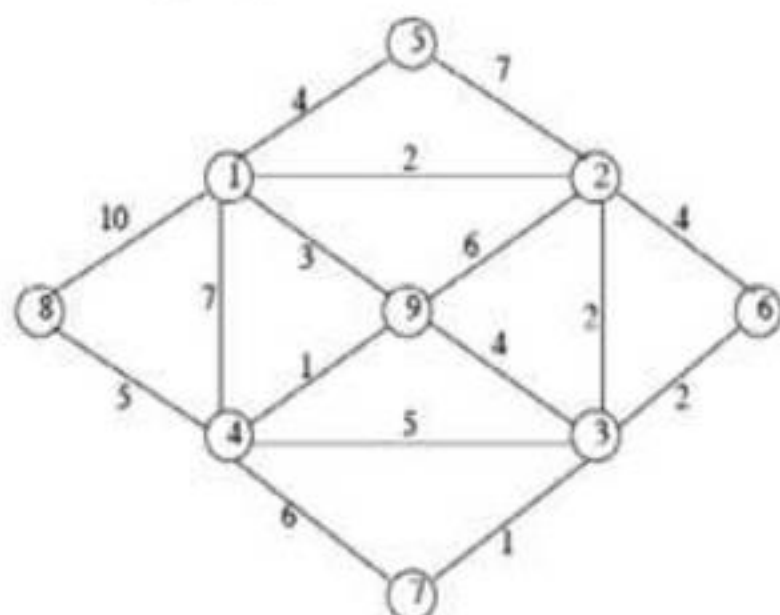
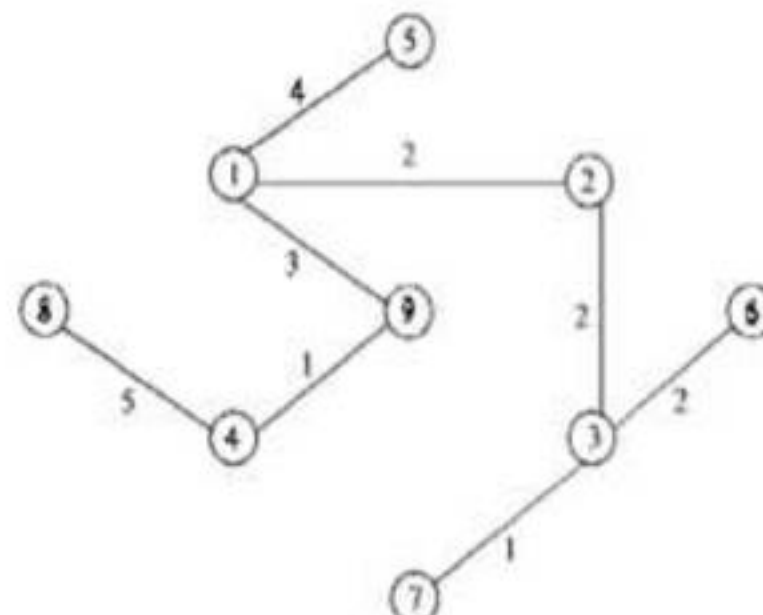
- Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are:
  - Civil Network Planning
  - Computer Network Routing Protocol
  - Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

**9.5 Minimum Spanning Tree**

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.
- A minimum spanning tree (MST) for a graph  $G = (V, E)$  is a sub graph  $G_1 = (V_1, E_1)$  of  $G$  contains all the vertices of  $G$ .
  - The vertex set  $V_1$  is same as that at graph  $G$ .
  - The edge set  $E_1$  is a subset of  $G$ .
  - And there is no cycle.

If a graph  $G$  is not a connected graph, then it cannot have any spanning tree. In this case, it will have a spanning forest. Suppose a graph  $G$  with  $n$  vertices then the MST will have  $(n - 1)$  edges, assuming that the graph is connected. A minimum spanning tree (MST) for a weighted graph is a spanning tree with minimum weight. That is all the vertices in the weighted graph will be connected with minimum edge with minimum weights. Figure B shows the minimum spanning tree of the weighted graph in Figure A.

**Figure A****Figure B**



### 9.5.1 Kruskal's Algorithm

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a minimum spanning forest. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- This is a one of the popular algorithm and was developed by Joseph Kruskal. To create a minimum cost spanning trees, using Kruskal's, we begin by choosing the edge with the minimum cost (if there are several edges with the same minimum cost, select any one of them) and add it to the spanning tree. In the next step, select the edge with next lowest cost, and so on, until we have selected  $(n - 1)$  edges to form the complete spanning tree. The only thing of which beware is that we don't form any cycles as we add edges to the spanning tree. Consider the graph of fig18.

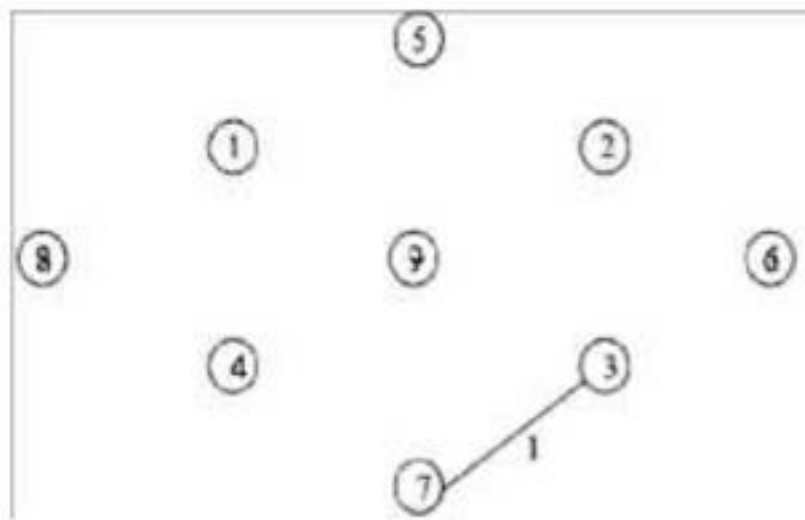


Figure A

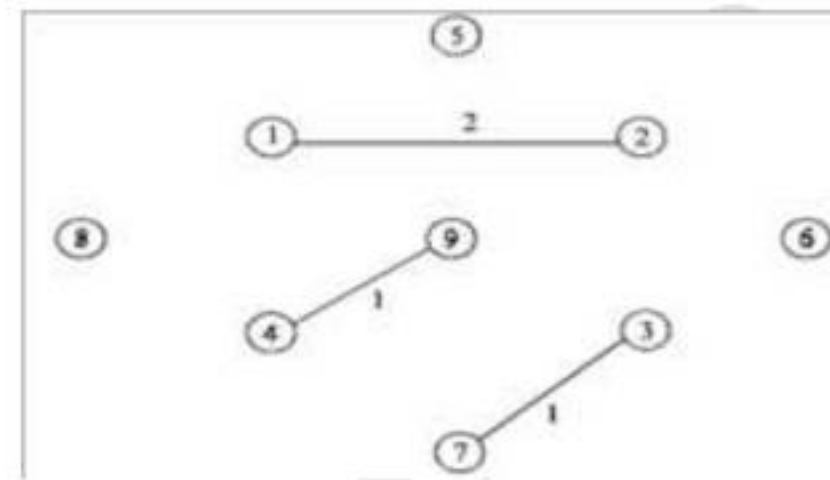


Figure B

The minimum cost edge in the graph G in Fig18 is 1. If we further analyze closely there are two edges (i.e., (7, 3), (4, 9)) with the minimum cost 1. As the algorithm says select any one of them. Here we select the edge (7, 3) as shown in Figure A. Again we select minimum cost edge (i.e., 1), which is (4, 9). Next we select minimum cost edge (i.e., 2). If we analyze closely there are two edges (i.e., (1, 2), (2, 3), (3, 6)) with the minimum cost 2. As the algorithm says select any one of them. Here we select the edge (1, 2) as shown in the Figure B.

Again we select minimum cost edge (i.e., 2), which is (2, 3). Similarly we select minimum cost edge (i.e., 2), which is (3, 6) as shown in Figure C

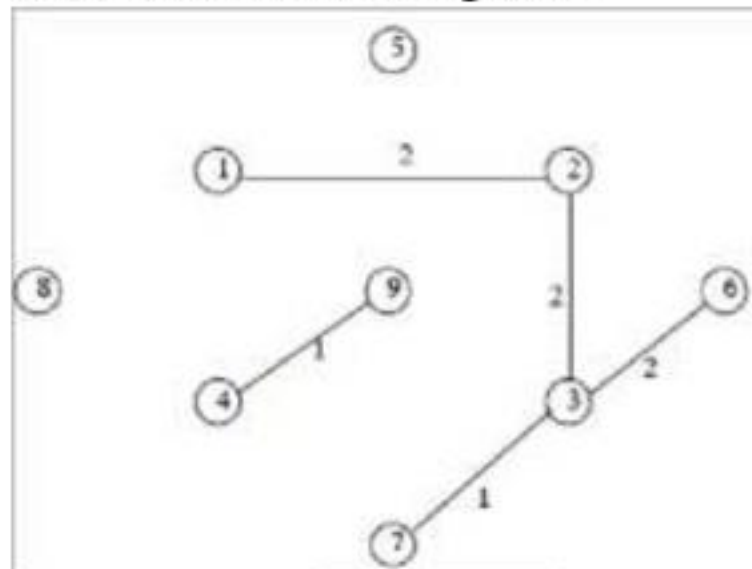


Figure C

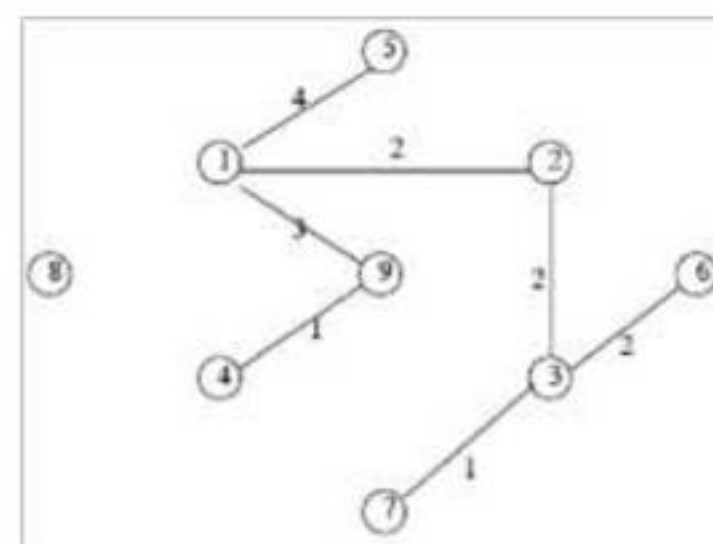


Figure D

Next minimum cost edge is (1, 9) with cost 3. Add the minimum cost edge to the minimumspanning tree. Again, the next minimum cost edge is (1, 5) with cost 4. Add the minimum cost edge to the minimum-spanning tree as shown in Figure D. Next minimum cost edge is (4, 8) with cost 5. Add the minimum cost edge to the minimum-spanning tree as shown in Figure E which is the minimum spanning tree with minimum cost of edges.

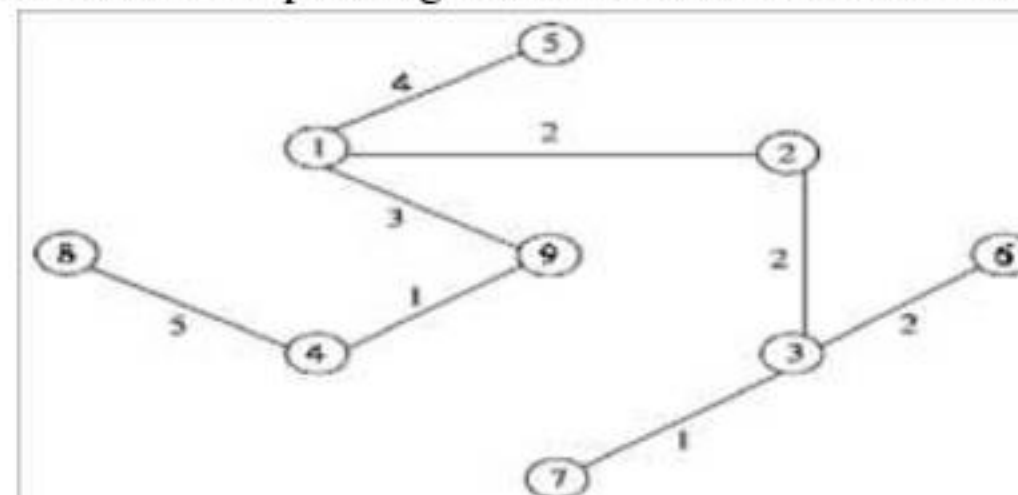


Figure E



### Algorithm

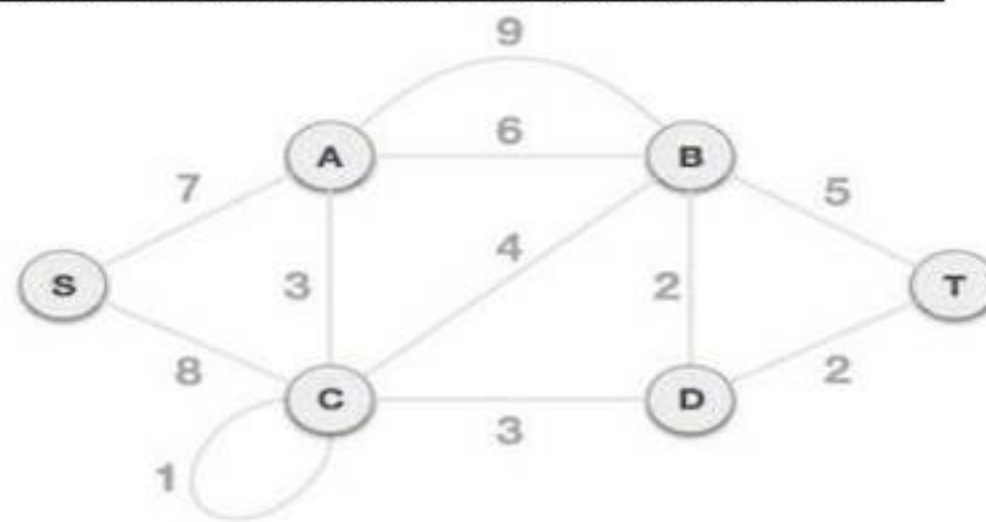
Suppose  $G = (V, E)$  is a graph, and  $T$  is a minimum spanning tree of graph  $G$ .

1. Initialize the spanning tree  $T$  to contain all the vertices in the graph  $G$  but no edges.
2. Choose the edge  $e$  with lowest weight from graph  $G$ .
3. Check if both vertices from  $e$  are within the same set in the tree  $T$ , for all such sets of  $T$ . If it is not present, add the edge  $e$  to the tree  $T$ , and replace the two sets that this edge connects.
4. Delete the edge  $e$  from the graph  $G$  and repeat the step 2 and 3 until there is no more edge to add or until the spanning tree  $T$  contains  $(n-1)$  edges.
5. Exit

### Analysis:

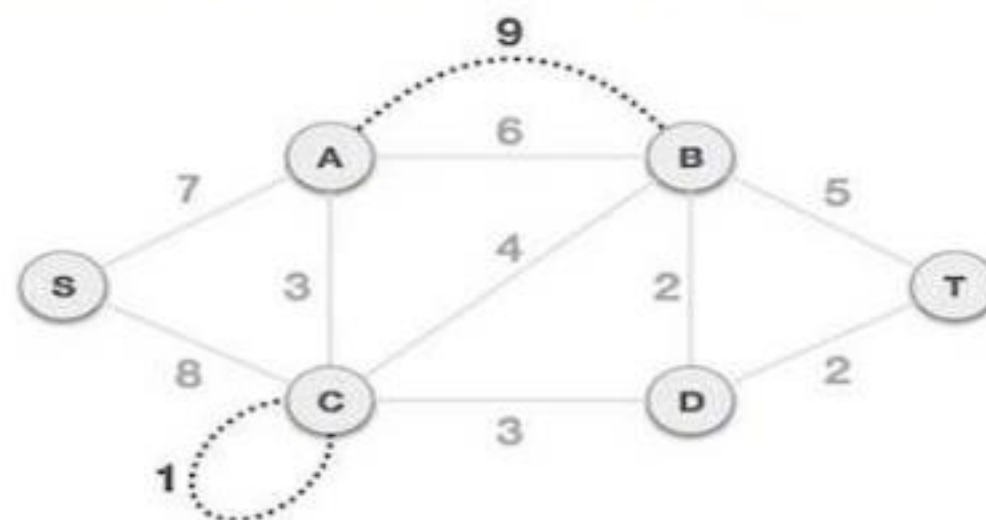
In the above algorithm the  $n$  tree forest at the beginning takes  $(V)$  time, the creation of set  $S$  takes  $O(E \log E)$  time and while loop execute  $O(n)$  times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is  $O(E \log E)$

### To understand Kruskal's algorithm let us consider the following example

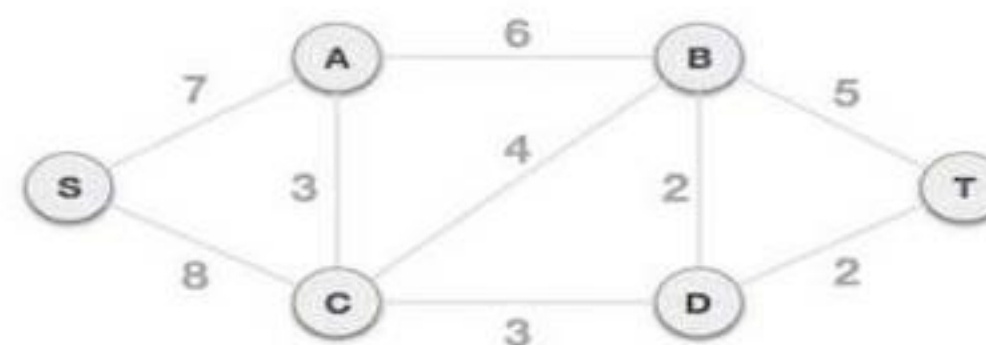


### Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



### Step 2 - Arrange all edges in their increasing order of weight

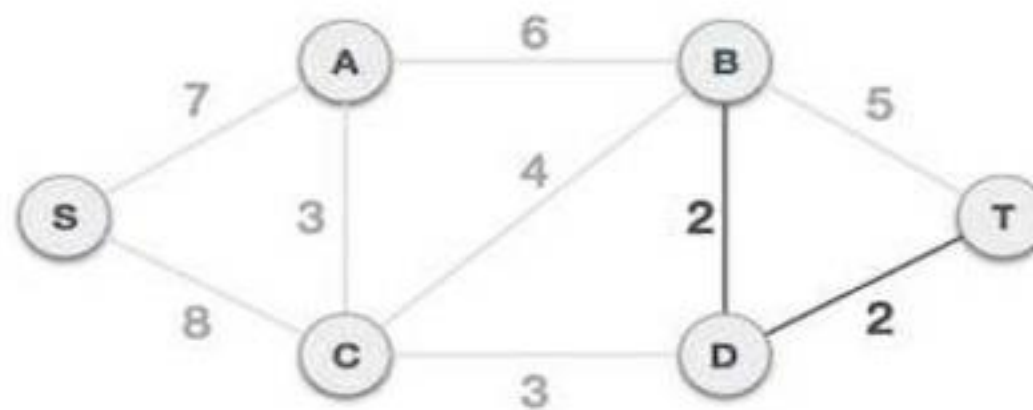
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



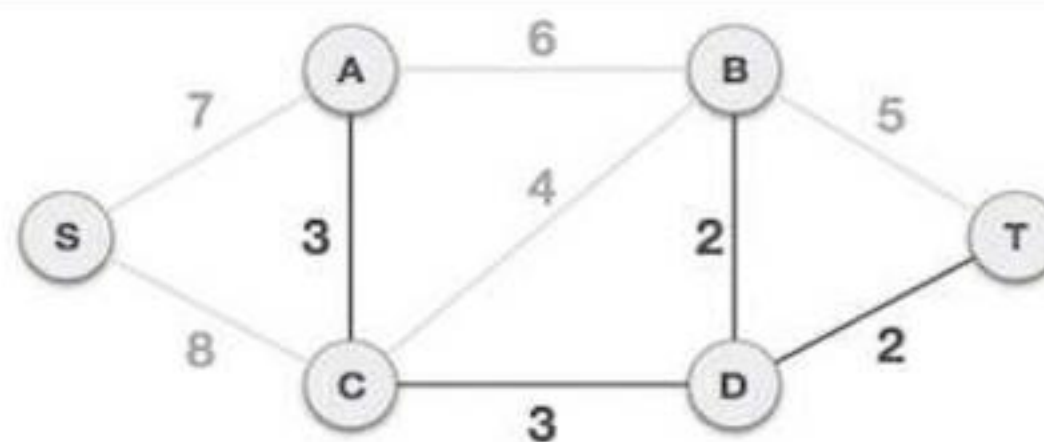
### Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning tree properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

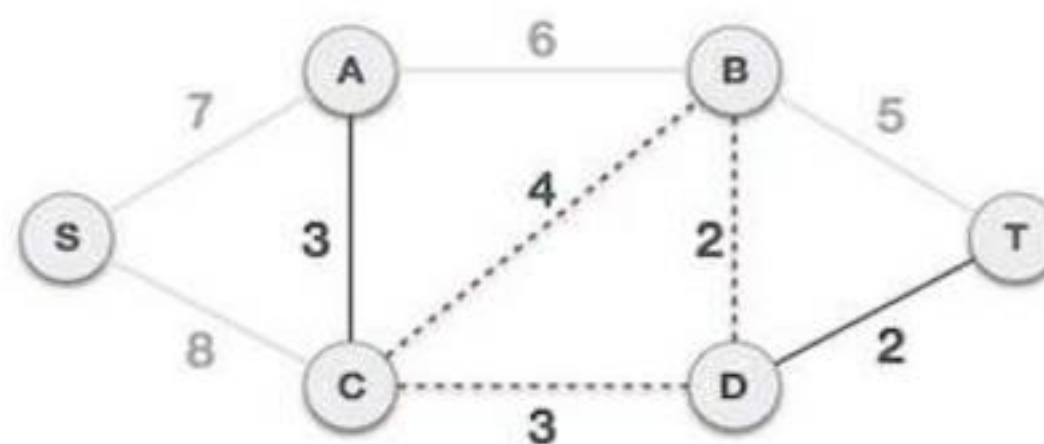


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

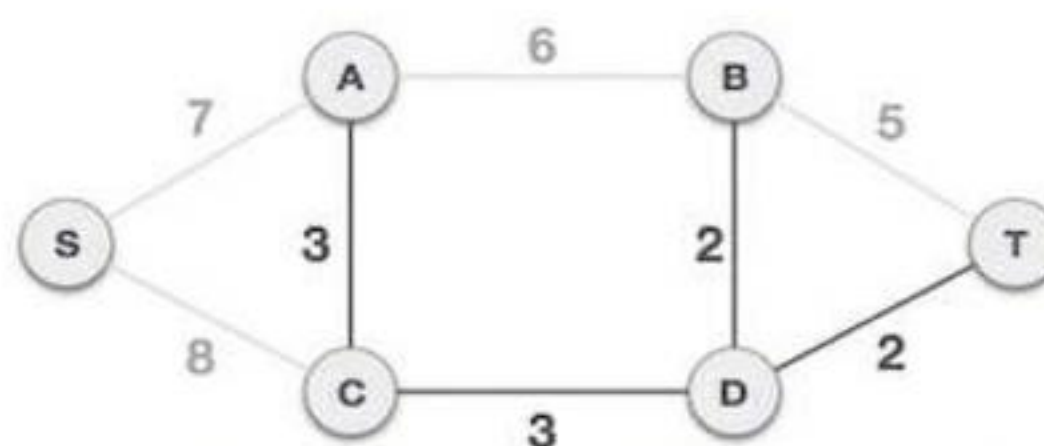
Next cost is 3, and associated edges are A,C and C,D. We add them again –



Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –

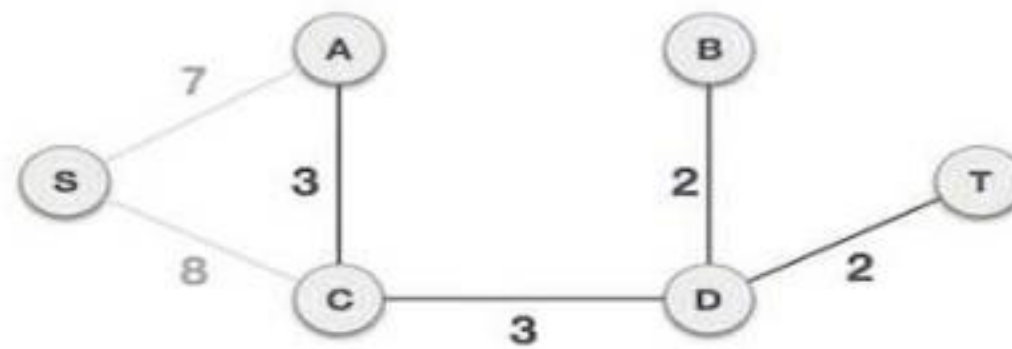


We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

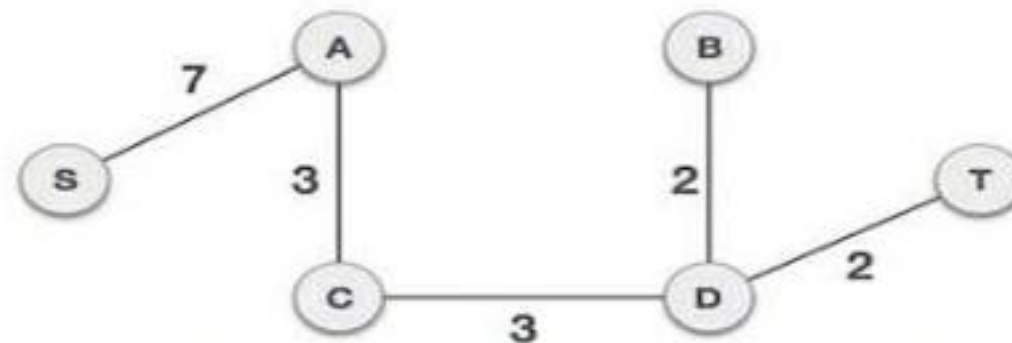


We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.





Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

### 9.5.2 Round Robin Algorithm

Round Robin algorithm, which provides even better performance when the number of edges is low. The algorithm is similar to Kruskal's except that there is a priority queue of arcs associated with each partial tree, rather than one global priority queue of all unexamined arcs.

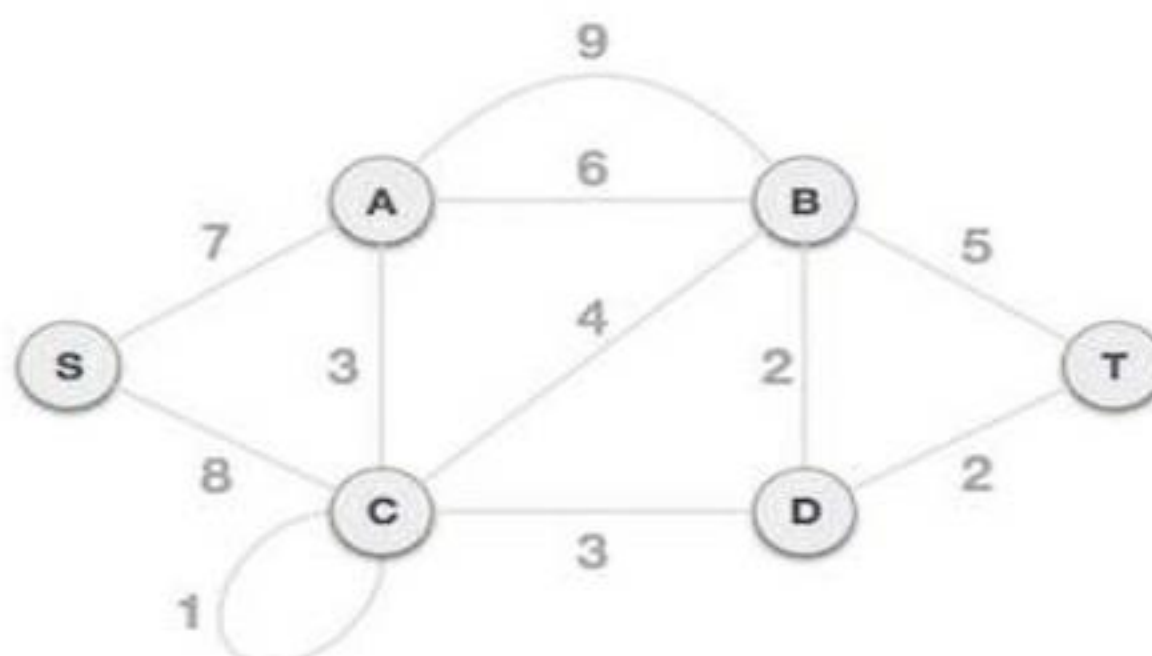
All partial trees are maintained in a queue,  $Q$ . Associated with each partial tree,  $T$ , is a priority queue,  $P(T)$ , of all arc with exactly one incident node in a tree, ordered by the weights of the arcs. Initially, as in Kruskal's algorithm, each node is a particular tree. A priority queue of the entire arcs incident to  $nd$  is created for each node  $nd$ , and the single node trees are inserted into  $Q$  in arbitrary order. The algorithm proceeds by removing a partial tree,  $T_1$ , from the front of  $Q$ , finding the minimum weight arc  $a$  in  $P(T_1)$ , deleting from  $Q$  the tree,  $T_2$ , at the other end of arc  $a$ ; combining  $T_1$  and  $T_2$  into a single new tree  $T_3$  and at the same time combining  $P(T_1)$  and  $P(T_2)$ , with  $a$  deleted, into  $P(T_3)$ ; and adding  $T_3$  to the rear of  $Q$ . This continues until  $Q$  contains a single tree; the minimum spanning tree.

### 9.5.3 Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

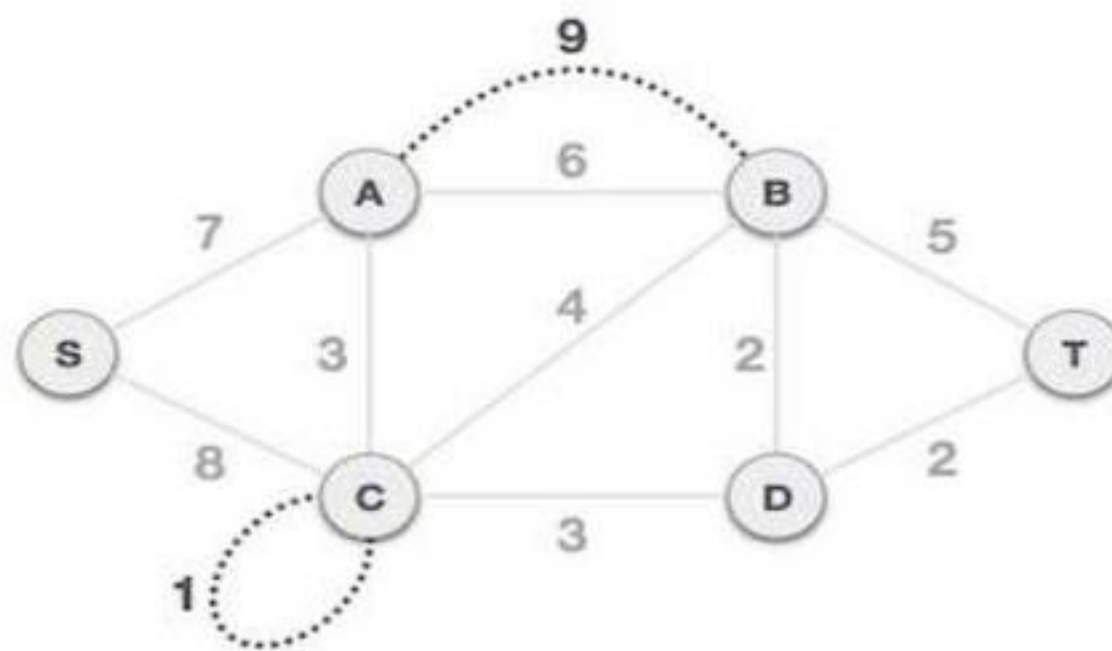
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –

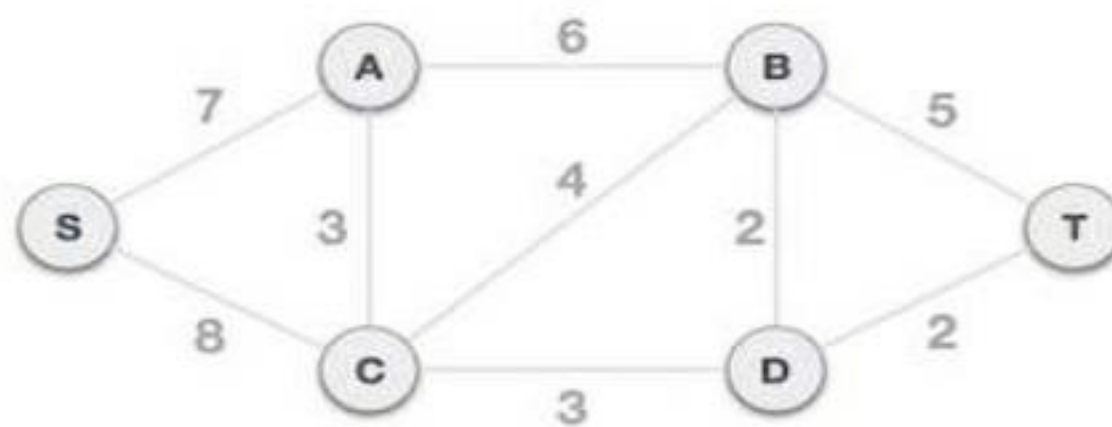




## Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

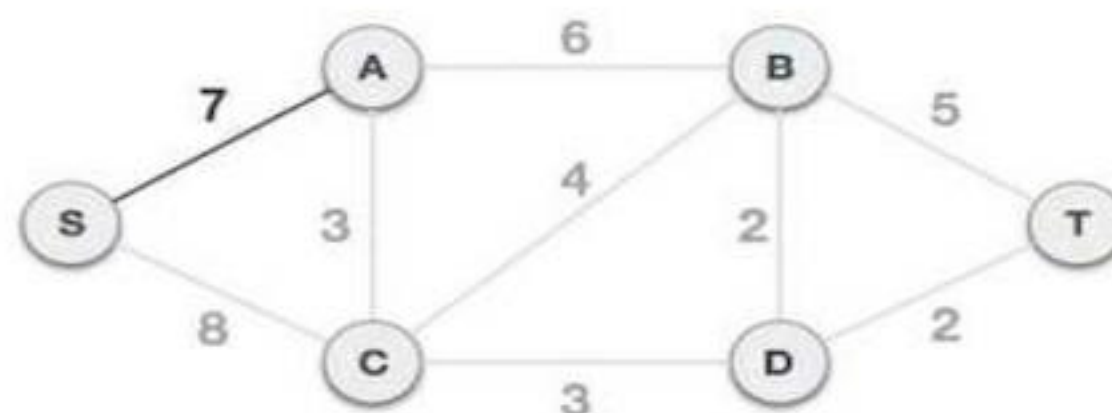


## Step 2 - Choose any arbitrary node as root node

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

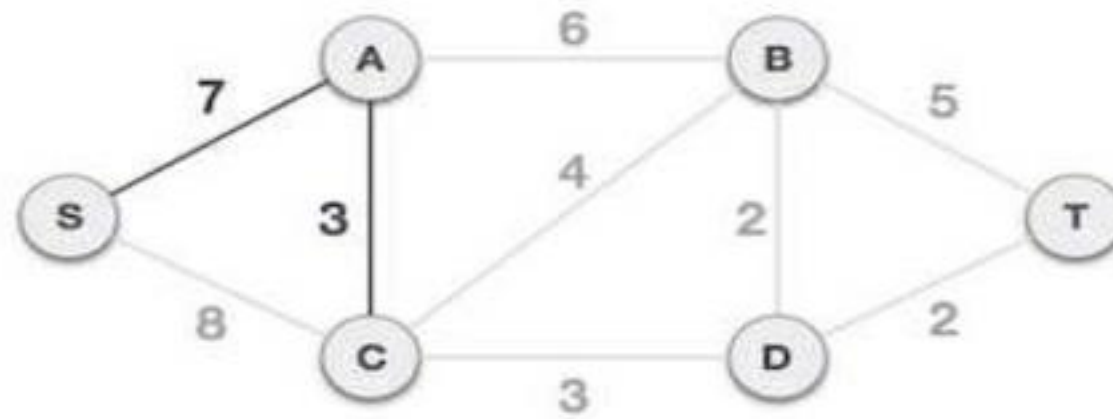
## Step 3 - Check outgoing edges and select the one with less cost

After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

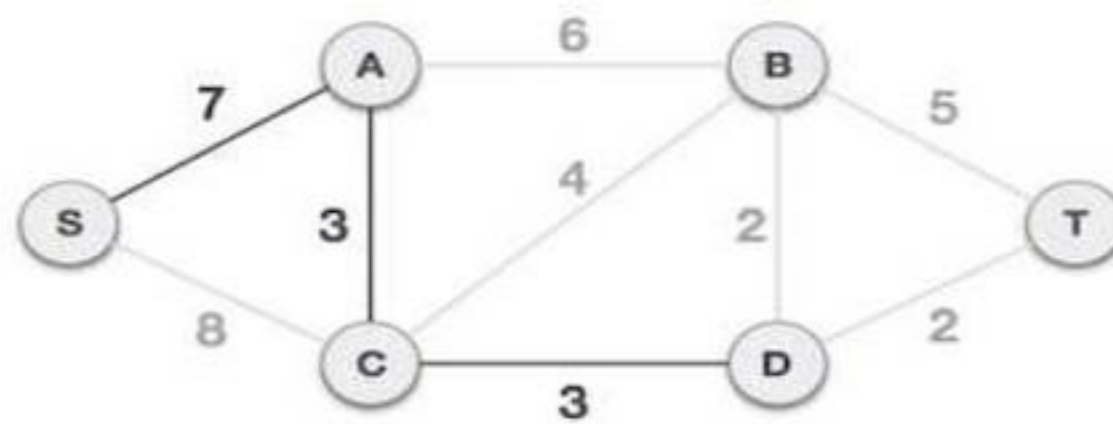


Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

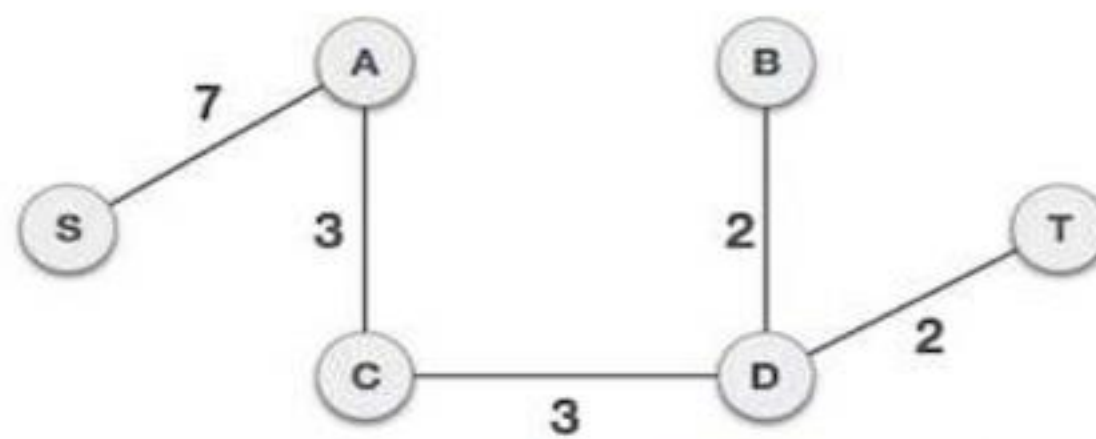




After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

## 9.6 Shortest Path

A path from a source vertex *a* to *b* is said to be shortest path if there is no other path from *a* to *b* with lower weights. There are many instances, to find the shortest path for traveling from one place to another. That is to find which route can reach as quick as possible or a route for which the traveling cost is minimum. Dijkstra's Algorithm is used to find shortest path.

### Greedy Algorithm

An optimization problem is one in which we want to find, not just a solution, but the best solution. A greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases. At each phase: we take the best we can get right now, without regard for future consequences and we hope that by choosing a local optimum at each step, we will end up at a global optimum.



## 9.7 Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First). Given a graph  $G$  and a source node  $A$ , the algorithm is used to find the shortest path (one having the lowest cost) between  $A$  (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node. For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

### Algorithm

Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can mean anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node  $Y$  can be given as the distance from the initial node to that node.

1. Select the source node also called the initial node
2. Define an empty set  $N$  that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0, and insert it into  $N$ .
4. Repeat Steps 5 to 7 until the destination node is in  $N$  or there are no more labelled nodes in  $N$ .
5. Consider each node that is not in  $N$  and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in  $N$  has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.  
(b) Else if the node that is not in  $N$  was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in  $N$  that has the smallest label assigned to it and add it to  $N$ .

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: temporary and permanent. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

- If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
- If the destination node is not labelled, then there is no path from the source to the destination node.

**Example:** Consider the graph  $G$  given in Figure Taking  $D$  as the initial node, execute the Dijkstra's algorithm on it.

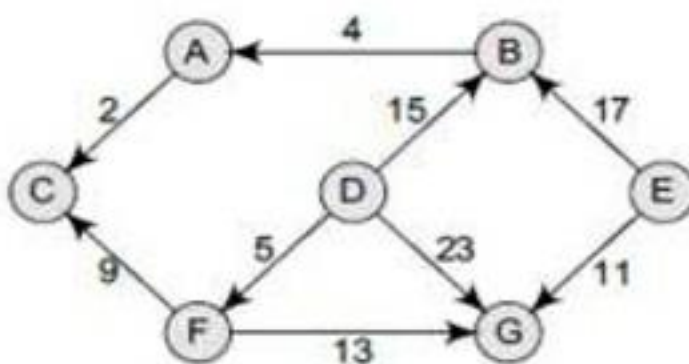


Figure 1 Graph  $G$

*Step 1:* Set the label of  $D = 0$  and  $N = \{D\}$ .

*Step 2:* Label of  $D = 0$ ,  $B = 15$ ,  $G = 23$ , and  $F = 5$ . Therefore,  $N = \{D, F\}$ .

*Step 3:* Label of  $D = 0$ ,  $B = 15$ ,  $G$  has been re-labelled 18 because minimum  $(5 + 13, 23) = 18$ ,  $C$  has been re-labelled 14  $(5 + 9)$ . Therefore,  $N = \{D, F, C\}$ .

*Step 4:* Label of  $D = 0$ ,  $B = 15$ ,  $G = 18$ . Therefore,  $N = \{D, F, C, B\}$ .

*Step 5:* Label of  $D = 0$ ,  $B = 15$ ,  $G = 18$  and  $A = 19$   $(15 + 4)$ . Therefore,  $N = \{D, F, C, B, G\}$ .

*Step 6:* Label of  $D = 0$  and  $A = 19$ . Therefore,  $N = \{D, F, C, B, G, A\}$ .

Note that we have no labels for node  $E$ ; this means that  $E$  is not reachable from  $D$ . Only the nodes that are in  $N$  are reachable from  $D$ .

The running time of Dijkstra's algorithm can be given as  $O(|V|^2 + |E|) = O(|V|^2)$  where  $V$  is the set of vertices and  $E$  in the graph.



## 9.8 Warshall's Algorithm

If a graph  $G$  is given as  $G=(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, the path matrix of  $G$  can be found as,  $P = A + A^2 + A^3 + \dots + A^n$ . This is a lengthy process, so Warshall has given

a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices  $P_0, P_1, P_2, \dots, P_n$  as given in Figure A

This means that if  $P_0[i][j] = 1$ , then there exists an edge from node  $v_i$  to  $v_j$ .

If  $P_1[i][j] = 1$ , then there exists an edge from  $v_i$  to  $v_j$  that does not use any other vertex except  $v_1$ .

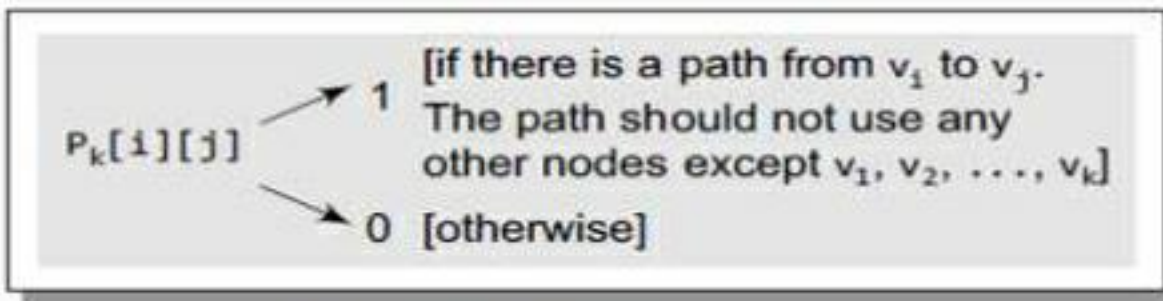


Figure A Path matrix entry

If  $P_2[i][j] = 1$ , then there exists an edge from  $v_i$  to  $v_j$  that does not use any other vertex except  $v_1$  and  $v_2$ .

Note that  $P_0$  is equal to the adjacency matrix of  $G$ . If  $G$  contains  $n$  nodes, then  $P_n = P$  which is the path matrix of the graph  $G$ .

From the above discussion, we can conclude that  $P_k[i][j]$  is equal to 1 only when either of the two following cases occur:

- There is a path from  $v_i$  to  $v_j$  that does not use any other node except  $v_1, v_2, \dots, v_{k-1}$ . Therefore,  $P_{k-1}[i][j] = 1$ .
- There is a path from  $v_i$  to  $v_k$  and a path from  $v_k$  to  $v_j$  where all the nodes use  $v_1, v_2, \dots, v_{k-1}$ . Therefore,

$$P_{k-1}[i][k] = 1 \text{ AND } P_{k-1}[k][j] = 1$$

Hence, the path matrix  $P_n$  can be calculated with the formula given as:

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][k] \wedge P_{k-1}[k][j])$$

where  $\vee$  indicates logical OR operation and  $\wedge$  indicates logical AND operation.

Figure 13.38 shows the Warshall's algorithm to find the path matrix  $P$  using the adjacency matrix  $A$ .

```

Step 1: [INITIALIZE the Path Matrix] Repeat Step 2 for I = 0 to n-1,
        where n is the number of nodes in the graph
Step 2:   Repeat Step 3 for J = 0 to n-1
Step 3:   IF A[I][J] = 0, then SET P[I][J] = 0
          ELSE P[I][J] = 1
          [END OF LOOP]
        [END OF LOOP]
Step 4: [Calculate the path matrix P] Repeat Step 5 for K = 0 to n-1
Step 5:   Repeat Step 6 for I = 0 to n-1
Step 6:   Repeat Step 7 for J = 0 to n-1
Step 7:   SET P_K[I][J] = P_{K-1}[I][J] \vee (P_{K-1}[I][K]
          \wedge P_{K-1}[K][J])
Step 8: EXIT

```

**Example:** Consider the graph in Figure B and its adjacency matrix  $A$ . We can straightaway calculate the path matrix  $P$  using the Warshall's algorithm.

The path matrix  $P$  can be given in a single step as:

$$P = \begin{matrix} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{matrix}$$

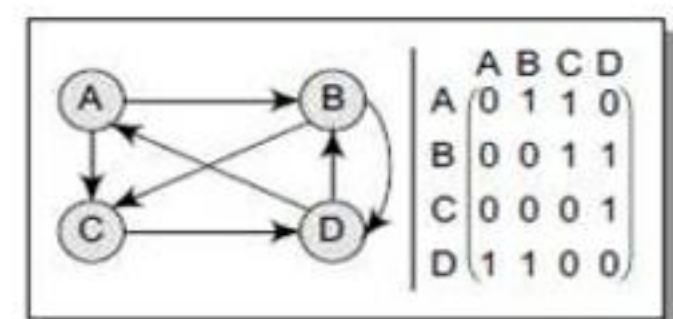


Figure B Graph  $G$  and its path matrix  $P$

Thus, we see that calculating  $A, A^2, A^3, A^4, \dots, A^5$  to calculate  $P$  is a very slow and inefficient technique as compared to the Warshall's technique.



### 9.9 Modified Warshall's Algorithm

Warshall's algorithm can be modified to obtain a matrix that gives the shortest paths between the nodes in a graph  $G$ . As an input to the algorithm, we take the adjacency matrix  $A$  of  $G$  and replace all the values of  $A$  which are zero by infinity ( $\infty$ ). Infinity ( $\infty$ ) denotes a very large number and indicates that there is no path between the vertices. In Warshall's modified algorithm, we obtain a set of matrices  $Q_0, Q_1, Q_2, \dots, Q_n$  using the formula given below.

$$Q_k[i][j] = \text{Minimum}(M_{k-1}[i][j], M_{k-1}[i][k] + M_{k-1}[k][j])$$

$Q_0$  is exactly the same as  $A$  with a little difference that every element having a zero value in  $A$  is replaced by ( $\infty$ ) in  $Q_0$ . Using the given formula, the matrix  $Q_n$  will give the path matrix that has the shortest path between the vertices of the graph.

```

Step 1: [Initialize the Shortest Path Matrix, Q] Repeat Step 2 for I = 0
        to n-1, where n is the number of nodes in the graph
Step 2:   Repeat Step 3 for J = 0 to n-1
Step 3:   IF A[I][J] = 0, then SET Q[I][J] = Infinity (or 9999)
          ELSE Q[I][J] = A[I][J]
          [END OF LOOP]
        [END OF LOOP]
Step 4: [Calculate the shortest path matrix Q] Repeat Step 5 for K = 0
        to n-1
Step 5:   Repeat Step 6 for I = 0 to n-1
Step 6:   Repeat Step 7 for J=0 to n-1
Step 7:   IF Q[I][J] <= Q[I][K] + Q[K][J]
          SET Q[I][J] = Q[I][J]
          ELSE SET Q[I][J] = Q[I][K] + Q[K][J]
          [END OF IF]
        [END OF LOOP]
      [END OF LOOP]
    [END OF LOOP]
Step 8: EXIT

```

### 9.10 Applications of Graphs

Graphs are constructed for various types of applications such as:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.