Chapter

12 Graphs

12.1 INTRODUCTION

A graph is another important non-linear data structure. The only non-linear data structure that we have seen so far is tree. In a tree data structure, there is a hierarchical relationship between parent and children, *i.e.*, one parent and many children. A tree in fact is a special type of graph. In a graph, the relationship is less restricted, *i.e.*, the relationship is from many parents to many children.

Graphs are data structures which have wide-ranging applications in real-life like. Airlines, analysis of electrical circuits, source, destination networks, finding shortest routes, flow chart of a program, statistical analysis etc.

Figure 12.1. shows the non-linear data structure called a graph.

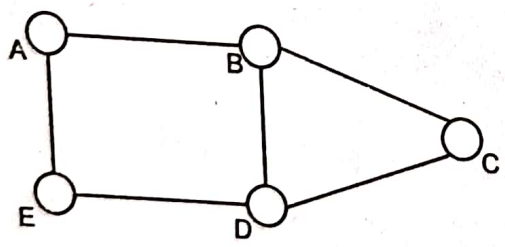


Fig. 12.1.

Chapter Outline

- Introduction
- Definition
- Terminologies Used
- Representation of Graph
- Implementation of a Graph
- Operations on Graph
- Traversing a Graph
- Breadth First Search
- Depth First Search
- Topological Sort
- Strongly Connected Components
- Spanning Tree
- Minimum Spanning Trees
- Kruskal's Algorithm
- Prim's Algorithm
- Shortest Paths
- Dijkstra's Algorithm
- Warshall's Algorithm
- Transitive Closure

12.2 DEFINITION

A graph G consists of a set V, where members are called the **vertices** of G and the set E, whose members are called the **edges**. The set of vertices is non-empty and the set of edges contains pair of vertices. If $e = (u, v)$ is an edge with vertices u and v, then u and v are said to lie on e and e is said to be incident with u and v.

Thus, a graph G is a collection of two sets V and E, where V is the vertices $v_0, v_1, \ldots v_{n-1}$ and E is the collection of edges $e_1, e_2, \ldots e_n$. This can be represented as $G = (V, E)$ where,

$$V(G) = (v_0, v_1, \ldots v_n) \text{ or set of vertices}$$
$$E(G) = (e_1, e_2, \ldots e_n) \text{ or set of edges}$$

A graph can be of two types :

1. Undirected Graph
2. Directed Graph.

Undirected Graph : If the pair of vertices are unordered then Graph G is called an **undirected graph**. That means if there is an edge between v_1 and v_2 then it can be represented as (v_1, v_2) or (v_2, v_1) also.

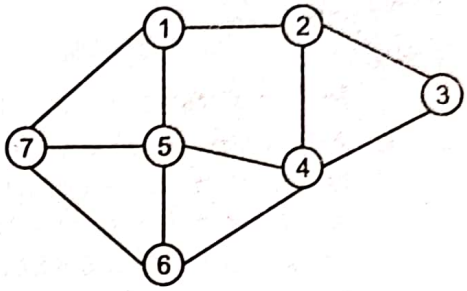


Fig. 12.2.

If the Fig. 12.2.

$$V(G) = \{1, 2, 3, 4, 5, 6, 7\}$$
$$E(G) = \{(1, 2), (2, 3), (3, 4), (2, 4), (4, 5), (5, 6), (7, 5), (1, 5)\}$$

That is, this graph 7 nodes and 8 edges.

Directed Graph : If the pair of vertices are ordered then Graph G is called **directed graph**. That is, a directed graph or digraph is a graph which has ordered pair of vertices (v_1, v_2) where v_1 is the tail and v_2 is the head of the edge. If the graph is directed then the line segments or arcs have arrow heads indicating the direction.

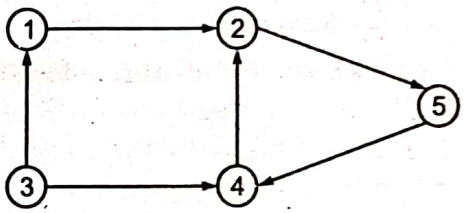


Fig. 12.3.

Directed graph is often shortend to diagraph.

$$V(G) = \{1, 2, 3, 4, 5\}$$
$$E(G) = \{(1, 2), (2, 5), (5, 4), (4, 2), (3, 4), (3, 1)\}$$

This graph has 5 nodes and 6 nodes.

12.3 TERMINOLOGIES USED

1. **Weighted Graph :** A graph is said to be a weighted graph if all the edges in it are labelled with some numbers. It is shown in the Fig. 12.4 below.

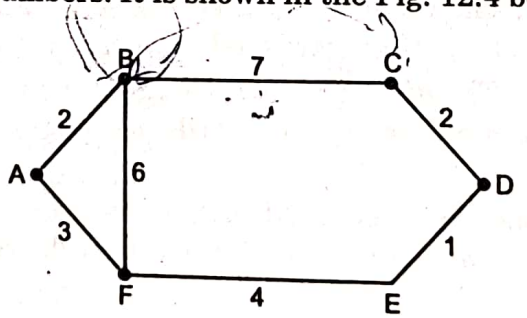


Fig. 12.4.

2. **Self Loop :** If there is an edge whose starting and end vertices are same that is (v_2, v_2) is an edge then it is called a self loop or simply a loop. It is shown in Fig. 12.5.
3. **Parallel Edges :** If there are more than one edge between the same pair of vertices then they are known as parallel edges.

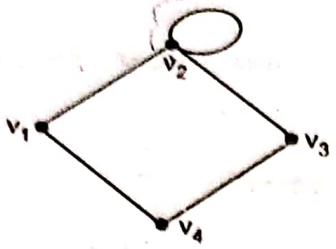


Fig. 12.5.

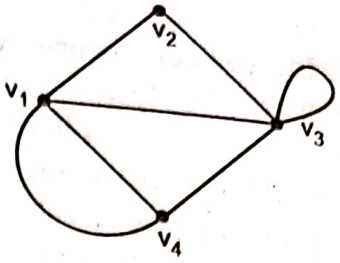


Fig. 12.6.

In Fig. 12.6, there are two edges between the same pair of vertices v_1 and v_4.

4. **Adjacent vertices :** A vertex u is adjacent to (or the neighbour of) other vertex v if there is an edge from u to v. In undirected graph if (v_1, v_2) is an edge then v_1 is adjacent to v_2 and v_2 is adjacent to v_1. In a directed graph if (v_1, v_2) is an edge then v_1 is adjacent to v_2 and v_2 is adjacent from v_1.
5. **Incidence :** In an undirected graph the edge (u, v) is incident on vertices u and v. In a digraph the edge (u, v) is incident from node u and is incident to node v.
6. **Degree of vertex :** The degree of a vertex is the number of edges incident to that vertex. In an undirected graph, the number of edges connected to a node is called the degree of that node.

Consider the graph, as shown in Fig. 12.7.

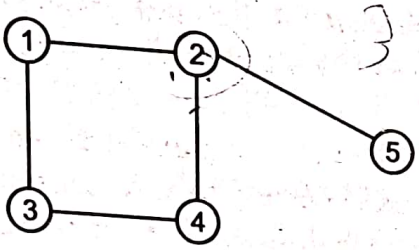


Fig. 12.7.

Here, for the vertex 2, the degree is 3 because 3 edges are incident to the vertex 2.

In a digraph, there are two degrees for every node.

1. In degree.
2. Out degree.

In degree of a vertex : The indegree of a vertex is the number of edges coming to that vertex or in other words, edges incident to it.

Outdegree of a vertex : The outdegree of a vertex is the number of edges going outside from that node or in other words, the edges incident from it.

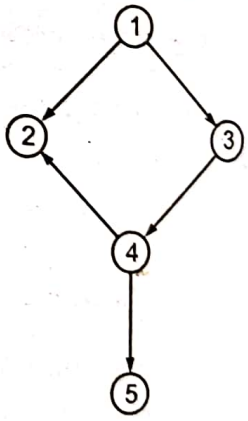


Fig. 12.8.

Here, for the vertex 2, the indegree is 2 because two edges reach 2 and for the vertex 4, three are two outgoing edges so the outdegree of the vertex 4 is 2.

7. **Simple graph :** A graph or directed graph which does not have only self-loop or parallel edges is called a simple graph.
8. **Multi-graph :** A graph which has either a self-loop or parallel edges or both is called a multi-graph.
9. **Complete graph :** A graph is complete graph if each vertex is adjacent to every other vertex in graph or we can say that there is an edge between any pair of nodes in the graph. An undirected complete graph will contain $n(n-1)/2$ edges.

10. **Regular graph :** A graph is regular if every node is adjacent to the same number of nodes.
    Here every node is adjacent to 3 nodes.

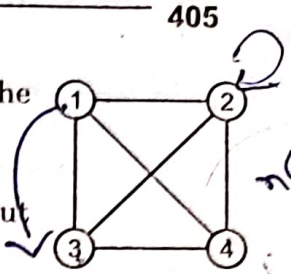
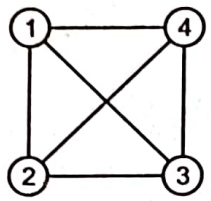
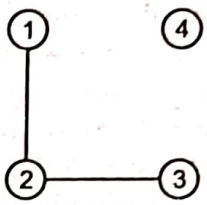


Fig. 12.9. Regular graph

11. **Planer graph :** A graph is planar if it can be drawn in a plane without any two edges intersecting.
12. **Connected graph :** In a graph G, two vertices v_1 and v_2 are said to be connected if there is a path in G from v_1 to v_2 or v_2 to v_1. A graph is said to be connected if there is a path from any node of graph to any other node, *i.e.*, for every pair of distinct vertices in G, there is a path.



(a) Connected graph



(b) Not connected graph

Fig. 12.10.

(*a*) ***Strongly Connected Graph*** **:** A directed graph is said to be strongly connected graph if for every pair of distinct vertices in G, there is a path.

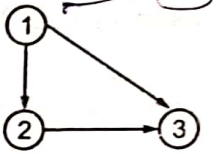


Fig. 12.11.

(*b*) ***Weakly Connected Graph*** **:** A diagram is called weakly connected or unilaterally connected if for any pair of nodes u and v, there is a path from u to v or a path from v to u. If from the diagraph we remove the directions and the resulting undirected graph is connected then that diagraph is weakly connected. Figure 12.11 shows a weakly connected graph.

13. **Cycle :** If there is a path containing one or more edges which starts from a vertex and terminates into the same vertex then the path is called as a cycle.
14. **Acyclic graph :** If a graph (digraph) does not have any cycle then it is called as acyclic graph.
15. **Cyclic graph :** A graph that has cycles is called a cyclic graph.
16. **Maximum edges in graph :** In an undirected graph there can be $\frac{n(n-1)}{2}$ maximum edges and in a directed graph there can be $n(n-1)$ maximum edges. Where n is the total number of vertices in the graph.
17. **Articulation point :** If on removing a node from the graph, the graph becomes disconnected then that node is called the articulation point.
18. **Bridge :** If on removing an edge from the graph, the graph becomes disconnected then that edge is called the bridge.
19. **Biconnected graph :** A graph with no articulation points is called a biconnected graph.

12.4 REPRESENTATION OF GRAPH

Usually, a graph can be represented in many ways. Some of these representations are:

1. Set representation.

2. Sequential Representation
   (*i*) Adjacency
   (*ii*) Incidence
3. Linked representation.

Set Representation

In this representation, two sets are maintained. They are :

1. Set of vertices v, and
2. Set of edges E, which is the subset of $V \times V$.

For the Fig.

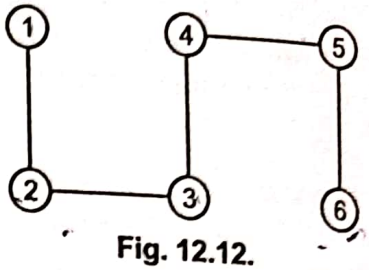


Fig. 12.12.

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$
$$E(G) = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)\}$$

Advantage

1. From the memory point of view it is most efficient method.

Disadvantages

1. This representation does not allow the storing of parallel edges in a multigraph.
2. Multiplication of the graph is difficult.

Sequential Representation

The graphs can be represented a matrices in sequential representation. There are two most common matrices.

1. Adjacency matrix.
2. Incidence matrix.

Adjacency Matrix Representation

Adjacency matrix is the matrix, which keeps the information of adjacent nodes. In other words, we can say that this matrix keeps the information that whether this vertex is adjacent to any other vertex or not.

The general representation of the adjacency matrix shown in the Fig. 12.13.

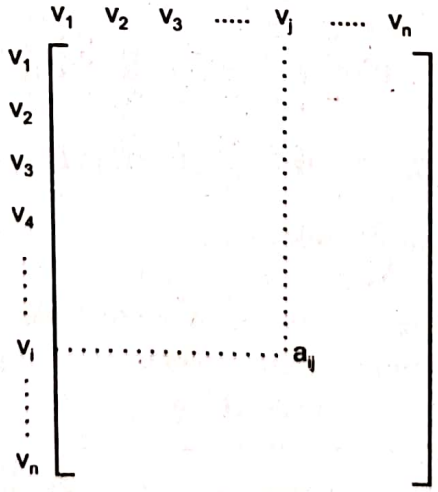


Fig. 12.13.

The entries in this matrix are placed according to the following rule :

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

This adjacency matrix is also called a **bit matrix** or **Boolean matrix** because the entries are either 0 or 1.

Let us take a graph.

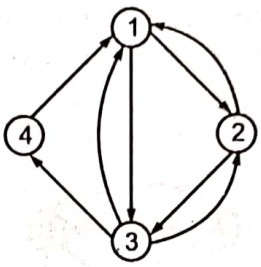


Fig. 12.14.

The corresponding adjacency matrix for this graph will be:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Now let us take an undirected graph

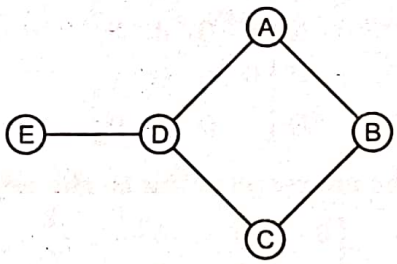


Fig. 12.15.

The corresponding adjacency matrix for this graph will be:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Note that, the adjacency matrix for an undirected graph is symmetric as the edge (AB) in $E(G)$ iff the edge (BA) is also in $E(G)$.

The adjacency matrix for a directed graph need not be symmetric.

Note : The space needed to represent a graph using its adjacency matrix is n^2 bits.

- From the adjacency matrix of an undirected graph, the degree of any vertex i is its row sum.
- From the adjacency matrix of a directed graph the row sum is the out-degree while the column sum is the in-degree.

- If A is an adjacency matrix of the graph G and if $A = A^T$ where A^T is the transpose of A then graph G is a simple undirected graph.
- An adjacency matrix is also useful to store multi-graphs as well as weighted graphs. In case of a multi-graph, instead of the entry 1, the entry will be the number of edges between two vertices and in the case of a weighted graph, the entries in the adjacency matrix are weights of the edges between the vertices instead of 0 or 1.

Let us take the graph,

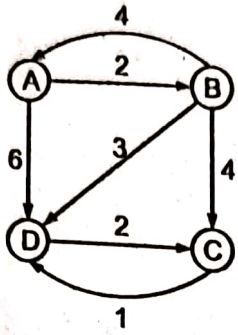


Fig. 12.16.

The corresponding adjacency matrix is:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 6 \\ 4 & 0 & 4 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix} \end{array}$$

EXAMPLE 12.1. Draw the graphs corresponding to the adjacency matrices.

$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \qquad A(G') = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Solution: (*a*) Since $A(G)$ is a 5×5 matrix, hence G will have 5 vertices say v_1, v_2, v_3, v_4 and v_5.

Draw an edge from v_i to v_j where $a_{ij} = 1$.

The required graph is shown in figure 12.17.

(*b*) Since $A(G')$ is a 4×4 square matrix, the graph G' has 4 vertices v_1, v_2, v_3, v_4 and the required graph is shown in figure 12.18.

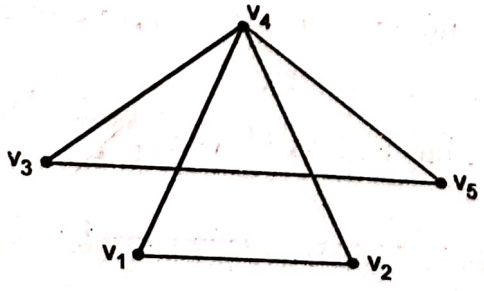


Fig. 12.17.

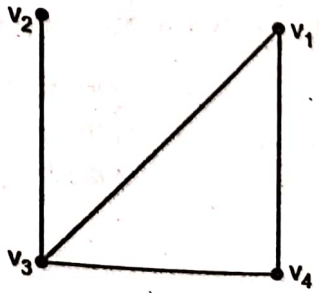


Fig. 12.18.

Powers of Adjacency Matrix

If A be an adjacency matrix of a diagraph then $A^n = [a_{ij}^n]$ gives the number of paths of length n from v_i to v_j where A^n is the n^{th} power matrix of A.

Consider the graph

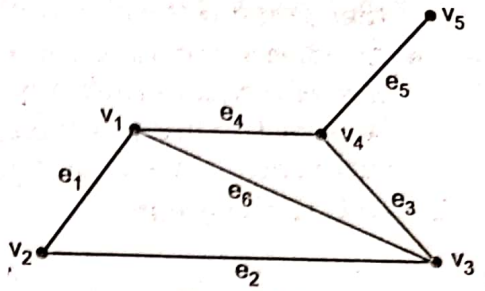


Fig. 12.19.

Whose adjacency matrix is

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

The matrices A^2, A^3 and A^4 for the graph as follows:

$$A^2 = \begin{bmatrix} 3 & 1 & 2 & 1 & 1 \\ 1 & 2 & 1 & 2 & 0 \\ 2 & 1 & 3 & 1 & 1 \\ 1 & 2 & 1 & 3 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad A^3 = \begin{bmatrix} 4 & 5 & 4 & 5 & 1 \\ 5 & 2 & 5 & 2 & 2 \\ 5 & 5 & 4 & 6 & 1 \\ 6 & 2 & 6 & 2 & 3 \\ 1 & 2 & 1 & 2 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 14 & 8 & 14 & 9 & 5 \\ 9 & 10 & 9 & 12 & 2 \\ 15 & 9 & 16 & 10 & 6 \\ 12 & 12 & 10 & 15 & 2 \\ 5 & 2 & 5 & 2 & 2 \end{bmatrix}$$

Note that $a_{2,4}^2 = 2$. So there are two paths of length 2 from v_2 to v_4 *i.e.*, $\{e_2, e_3\}$ and $\{e_1, e_4\}$.

Also $a_{4,5}^3 = 3$, So there are 3 paths of length 3 from v_4 to v_5.

Incidence Matrix

Let G be a graph with n vertices and e edges then the incidence matrix is a matrix of order $n \times e$, whose n rows correspond to the n vertices and e columns correspond to the e edges as follows :

$$a_{ij} = \begin{cases} 1 & \text{if the } j^{th} \text{ edge } e_j \text{ is incident on } i^{th} \text{ vertex } v_i \\ 0 & \text{otherwise} \end{cases}$$

For example, suppose the graph G is as follows

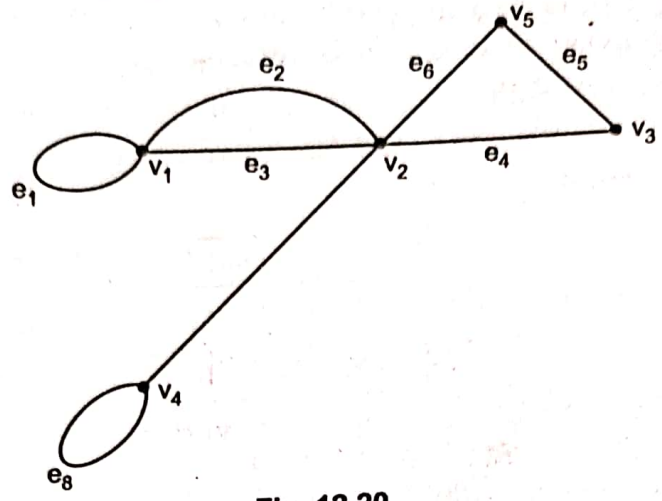


Fig. 12.20

The incidence matrix for this graph is

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Path Matrix

Let G be a simple directed graph with n vertices $v_1, v_2, \ldots, v_n$. An $n \times n$ matrix $P = [P_{ij}]$ defined as follows

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Suppose there is a path from v_i to v_j then there must be simple path from v_i to v_j when $v_i \neq v_j$ or there must be a cycle from v_i to v_j when $v_i = v_j$.

Matrix P is called the path matrix or **reachability matrix** of the graph G.

The path matrix P of a given graph G can be obtained from its adjacency matrix by following steps :

1. From the adjacency matrix of A, we can determine whether there exists an edge from one vertex to another.
2. Find A^n for some possible integer n.
3. Add the matrices $A, A^2, A^3, \ldots, A^n$.
   i.e., $B^n = A + A^2 + \ldots + A^n$
4. Now path matrix P can be obtained from B^n as follows :
   $P_{ij} = 1$ if and only if there is a non-zero element in the i, j entry of the matrix B^n.
   otherwise $P_{ij} = 0$.

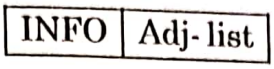
Linked Representation

Let G be a directed graph with n vertices. The sequential representation of G in memory has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes

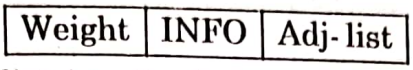
in G. This is because the size of matrix may need to be changed and the nodes may need to be reordered, so there may be many changes in the matrix. Furthermore, if the number of edges is $O(n)$ or $O(n \log_2 n)$ then the matrix will be sparse (will contain many zeros). So, G is represented in memory by a linked representation.

In linked representation, two node structures are used.

1. For non-weighted graph



2. For weighted graph



where Adj-list is the adjacency list *i.e.*, the list of vertices which are adjacent for the corresponding node.

In the linked representation of graphs, the number of list depends on the number of vertices in the graph. The header node in each list maintains a list of all adjacent vertices of that node for which the header node is meant. Consider a graph adjacency list representation of this undirected graph is:

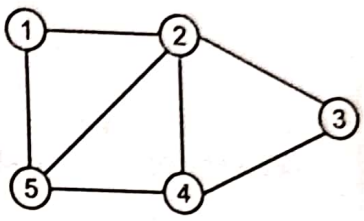
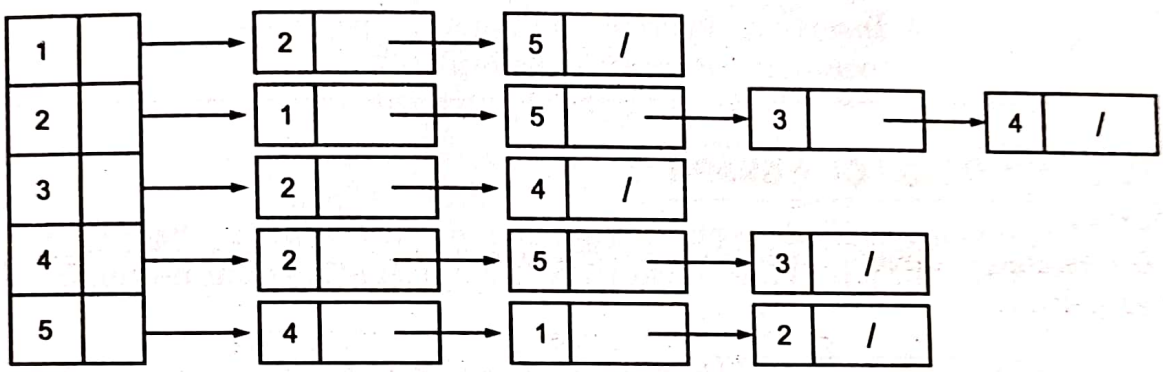


Fig. 12.21.



Suppose a directed graph.

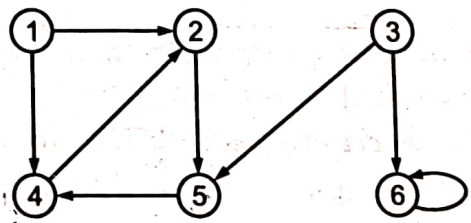
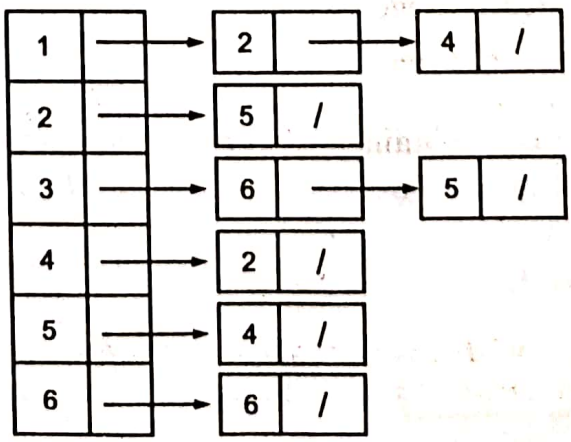


Fig. 12.22.

the adjacency list representation of this graph.



Comparison of various representations :

Representation	*Advantages*	*Disadvantages*
1. Set representation	1. It is the most efficient method from the memory point of view.	1. It does not allow storing of parallel edges. 2. Manipulation has a lot of difficulities.
2. Matrix representation	1. Manipulation of the graph with this representation is very easy. 2. It allows storing of parallel edges.	1. Space is wasted. 2. Insertion and deletion operations are not that much easy when compared to the linked representation
3. Linked representation	1. It is a space-saving method. 2. It is dynamic in nature. 3. It allows storing of parallel edges. 4. Insertion, deletion and other operations arts easy to perform.	1. Pointer fields are used as extra fields in addition to the data fields. 2. There is no faster way than to search the entire list for the presence of the edge.

12.5 IMPLEMENTATION OF A GRAPH

Let us first consider non-weighted graph with number of vertices MAX, where MAX is defined equal to 20, then the basic node type which store the information of **non-weighted graph** is defined as:

```
#define MAX 30
typedef struct node
    {
        int vertex;
        struct node *next;
    }node1;
```

Now, we can declare the adjacency list on the basis of number of vertices of graph as:

```
node1 *adj [MAX];
```

and for **weighted graph,** it becomes

```
typedef struct node
    {
        int vertex;
            int weight ;
        struct node *next;
    }node 2;
node2 *adj [MAX];
```

12.6 OPERATIONS ON GRAPH

Most of the algorithms developed assumed that the input graph is represented in adjacency-list representation.

(a) Creating an empty Graph :

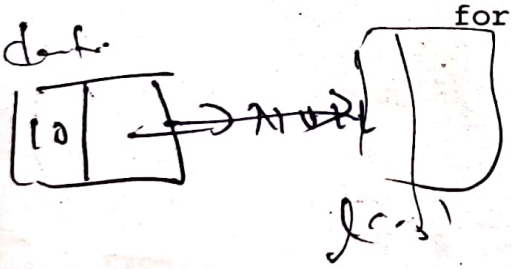
```
void createGraph (node1 *adj[], int num)
    {
        int i;
        for (i=1, i<=num; i++)
            adj[i]=NULL;
    }
```

To create an empty graph, the entire adjacency list contains the NULL value.

(b) Inserting values in a Graph

(i) Non-weighted graph

```
void input (node1 *adj[], int num)
    {
        node1 *ptr, *last;
        int i, j, k, value;
        for (i=1; i<=num; i++)
            {
                last = NULL;
                printf("\n Number of Nodes in the adjacency list of
                    node %d:", i);
                scanf ("%d", &k);
                for (j=1; j <=k; j++)
                {
                    printf("Enter the node %d:", j);
                    scanf ("%d", &value);
                    ptr = (node1*) malloc (sizeof (node1));
                    ptr → vertex = value ;
                    ptr → next = NULL;
                    If (adj [i] == NULL)
                        adj [i] = last = ptr;
                    else
                        {
                            last → next = ptr;
                            last = ptr;
                        }
                }
            }
    }
```



(ii) Weighted Graph

```
void inputweighted (node2 * adj[], int n)
    {
        node2 *ptr, *last;
        int i, j, k, value, w;
        for (i = 1; i <=n; i++)
            {
                last=NULL;
                printf("\n No. of nodes in the adjacency list of node
                    %d:", i);
                scanf ("%d",&k);
                for (j=1; j <=k; j++)
                    {
                        printf ("Enter node %d:", j);
                        scanf ("%d", &value);
                        printf ("Enter weight for edge (%d %d)",i,value");
                        scanf ("%d", &w);
                        ptr=(node2*) malloc (sizeof (node2));
                        ptr → vertex = value;
                        ptr → weight = w;
                        ptr → next = NULL;
                        if (adj[i] == NULL)
                            adj [i] = last = ptr;
                        else
                            {
                                last → next = ptr;
                                last = ptr;
                            }
                    }
            }
    }
```

(c) Printing a Graph

This shows the way graph information can be outputted

```
void printgraph (node1*adj[],int n)
    {
        node1 * ptr;
        int i;
        for (i=1, i<=n; i++)
            {
                ptr = adj[i];
```

```
                printf("%d",i);
                while(ptr!=NULL)
                    {
                        printf("→ %d", ptr → vertex);
                        ptr = ptr → next;
                    }
                printf("\n");
            }
    }
```

(d) Deleting a Graph :

This shows a graph removed from memory

```
void delegraph (node1 *adj[], int n)
    {
        int i;
        node1 *temp, *ptr;
        for (i=1; i<=n; i++)
            {
                ptr=adj[i];
                while(ptr!=NULL)
                    {
                        temp=ptr;
                        ptr=ptr → next;
                        free (temp);
                    }
                adj[i] = NULL;
            }
    }
```

12.7 TRAVERSING A GRAPH

As we know that traversing is nothing but visiting each node in some systematic approach. Graph is represented by its nodes and edges, so traversal of each node is the traversing in graph. There are two graph traversal methods :

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

In BFS, we use **queue** for keeping nodes, which will be used for next processing and in DFS, we use **stack**, which keeps the node for next processing.

12.7.1 Difference Between Traversal in Graph and Tree

Traversal in graph is different from traversal in tree or list because of the following reasons :

(*a*) There is no first node or root node in a graph, hence the traversal can start from any node.

(*b*) In tree or list when we start traversing from the first node, all the nodes are traversed but in graph only those nodes will be traversed which are reachable from the starting

node. So if we want to traverse all the reachable nodes we again have to select another starting node for traversing the remaining nodes.

(c) In tree or list while traversing we never encounter a node more than once but while traversing graph, there may be a possibility that we reach a node more than once. So to ensure that each node is visited only once we have to keep the status of each node whether it has been visited or not.

(d) In tree or list we have unique traversals. For example, if we are traversing the tree in inorder there can be only one sequence in which nodes are visited. But in graph, for the same technique of traversal there can be different sequences in which nodes can be visited.

12.8. BREADTH FIRST SEARCH

This graph traversal technique uses *queue* for traversing all the nodes of the graph. In this, first we take any node as a starting node then we take all the nodes adjacent to that starting node. Similar approach we take for all other adjacent nodes, which are adjacent to the starting node and so on. We maintain the status of visited node in one array so that no node can be traversed again.

Suppose V_0 is our starting node and V_1, V_2, V_3 are nodes adjacent to it. V_{11}, V_{12}, V_{13} are nodes adjacent to V_1, V_{21}, V_{22} are nodes adjacent to V_2 and V_{31} is adjacent to V_3. So we will traverse V_0 first and then all nodes adjacent to V_0, *i.e.*, V_1, V_2, V_3. Then we will traverse all nodes adjacent to V_1, *i.e.*, V_{11}, V_{12}, V_{13} and then we will traverse nodes adjacent to V_2, *i.e.*, V_{21}, V_{22} and then nodes adjacent to V_3, *i.e.*, V_{31}. Traversal will be in following order ;

$V_0\ V_1\ V_2\ V_3\ V_{11}\ V_{12}\ V_{13}\ V_{21}\ V_{22}\ V_{31}$

Let us take a graph and apply BFS to it

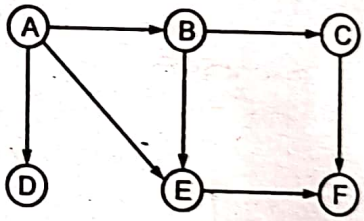


Fig. 12.23.

Take the node *A* as the starting node and start the traversal of the given graph.

First we traverse node *A*, then we traverse all adjacent nodes to node *A*, *i.e.*, *B*, *D* and *E*. We can traverse these nodes in any order. Suppose we traverse in *B*, *D*, *E* order, so now traversal is:

A, *B*, *D*, *E*

Now we traverse all nodes adjacent to *B*, then all the nodes adjacent to *D* then all the nodes adjacent to *E*, *i.e.*, *C*, *F* we can see that node adjacent to node *B* is *E* and *C*, but *E* is already traversed so we will ignore it and now the traversal is:

A, *B*, *D*, *E*, *C*, F

This was the traversal when we take node *A* as the starting node.

BFS through Queue

Take an array queue which will be used to keep the unvisited neighbours of the node. Take a boolean array *visited* which will have value *true* if the node has been *visited* and will have value *false* if the node has *not been visited*.

Initially queue is empty and front = – 1 and rear = – 1.

Initially visited [*i*] = false where *i* = 1 to *n*, *n* is total number of nodes.

Procedure

1. Insert starting node into the queue.

2. Delete front element from the queue and insert all its unvisited neighbours into the queue at the end, and traverse them. Also make the value of visited array true for these nodes.
3. Repeat step 2 until the queue is empty.

Let us take node 1 as the starting node for traversal.

Consider the graph G in Fig. 12.24. below:

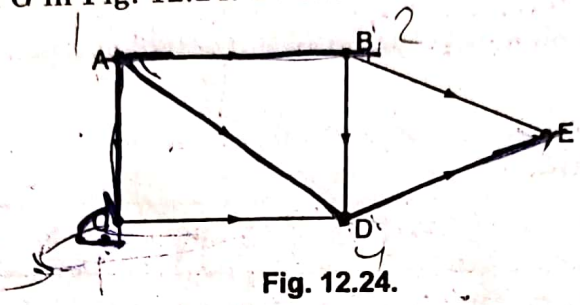


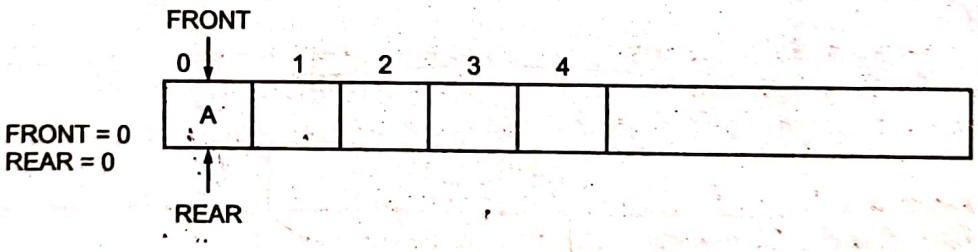
Fig. 12.24.

The linked list or adjacency list representation of the graph is follows :

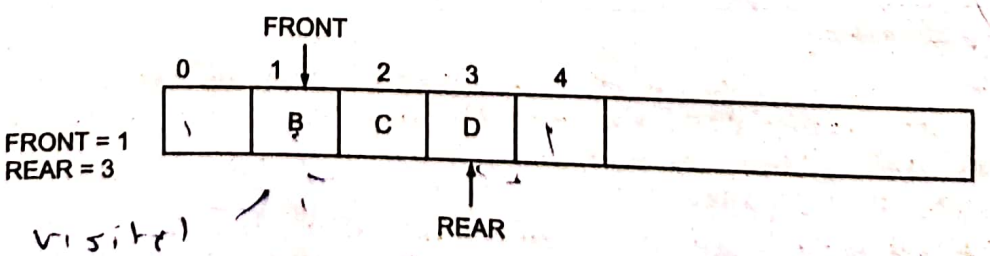
Vertex	*Adjacency list*
A	*B, C, D*
B	*D, E*
C	*D*
D	*E*
E	

Suppose the source vertex is A. Then following steps will illustrate the BFS.

Step 1 : Initially push A (the source vertex) to the queue.

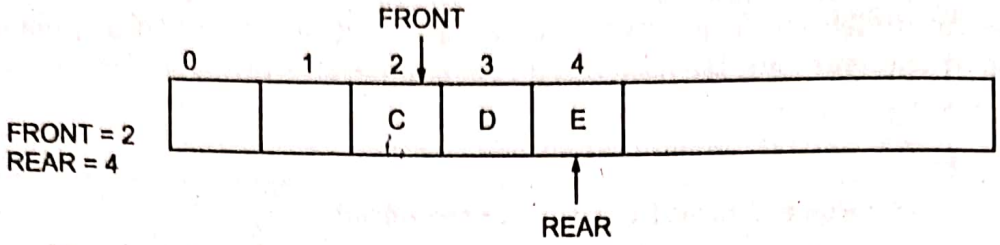


Step 2 : Remove the front element A from the queue (by incrementing FRONT = FRONT + 1) and display it. Then push all the neighbouring vertices of A to the queue (by incrementing REAR = REAR + 1) if it is not in queue.



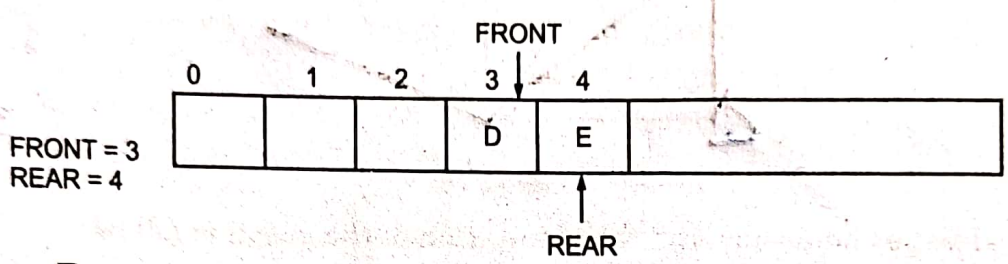
Traversed nodes = A.

Step 3 : Remove the front element B from the queue and display it then add the neighbouring vertices of B to the queue, if it is not in queue.



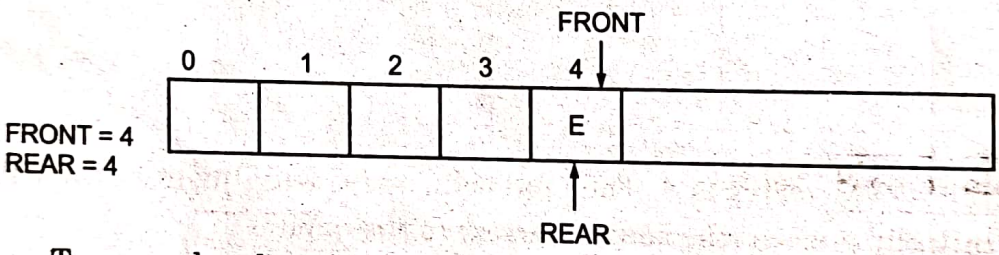
Traversed nodes = *A*, *B*.

Step 4 : Remove the front element *C* and add the neighbouring vertex if it is not present in the queue, *i.e.*,



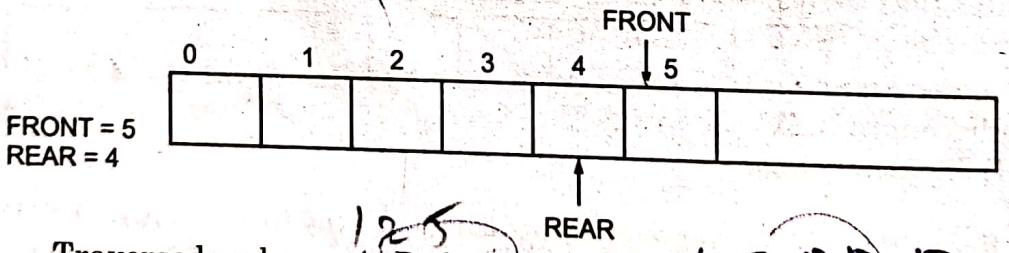
Traversed nodes = *A*, *B*, *C*.

Step 5 : Again the process is repeated (until FRONT > REAR), *i.e.*, remove the front element *D* of the queue and add the neighbouring vertex if it is not present in the queue.



Traversed nodes = *A*, *B*, *C*, *D*.

then,



Traversed nodes = *A*, *B*, *C*, *D*, *E*.

So, *A*, *B*, *C*, *D*, *E* is the BFS traversal of the graph.

C Implementation

```
BFS (int v)
    {
        int i, front, rear;
        int que[20];
        front=rear=-1;
        printf("%d",V);
        visited[v]=true;
        rear++;
```

```
            front++;
            que[rear]=v;
            while(front<=rear)
            {
                v=que[front]; /* delete fron queue */
                front++;
                for(i=1;i<=n;i++)
                {
                    /* Check or adjacent unvisited nodes */
                    if(adj[v][i]==1 && visited[i]==false)
                    {
                        printf("%d",i);
                        visited[i]=true;
                        rear++;
                        que[rear]=i;
                    }
                }
            }
        }
```

BFS by Colouring Scheme

To keep track of the progress, BFS colours each vertex WHITE, GRAY or BLACK, where vertex WHITE colour indicates that the vertex is undiscovered, vertex with GRAY colour indicates that the vertex is discovered but not yet processed and the vertex with BLACK colour indicates that the vertex has been processed. So all the vertices are initially WHITE and later on become GRAY and then BLACK.

Thus if $(u, v) \in E$ and vertex u is BLACK, then vertex v is either GRAY or BLACK, *i.e.*, all vertices adjacent to BLACK vertices have been discovered. Gray vertices may have some adjacent WHITE vertices.

```
BFS (G, s)
    1.  for each vertex u ∈ V[G] - {s}
    2.      do color[u] ← WHITE
    3.          d[u] ← ∞/* distance from vertex u */
    4.          π[u] ← NIL
    5.  color[s] ← GRAY
    6.  d[s] ← 0
    7.  π[s] ← NIL
    8.  Q ← ϕ
    9.  ENQUEUE (Q, s)
    10. while Q ≠ ϕ
    11.     do u ← DEQUEUE (Q)
```

```
12.          for each v ∈ Adj[u]
13.              do if color[v] = WHITE
14.                  then color[v] ← Gray
15.                       d[v] ← d[u] + 1
16.                       π[v ] ← u
17.                       ENQUEUE (Q, v)
18.          color[u] ← Black
```

EXAMPLE 12.2. Consider the graph *G* in Fig. 12.25. Describe the whole process of breadth first search. Using vertex 3 as the source.

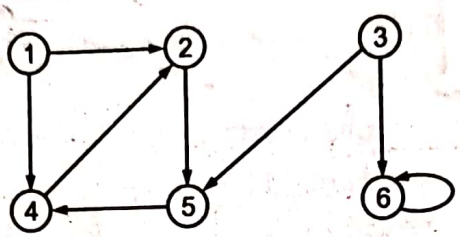
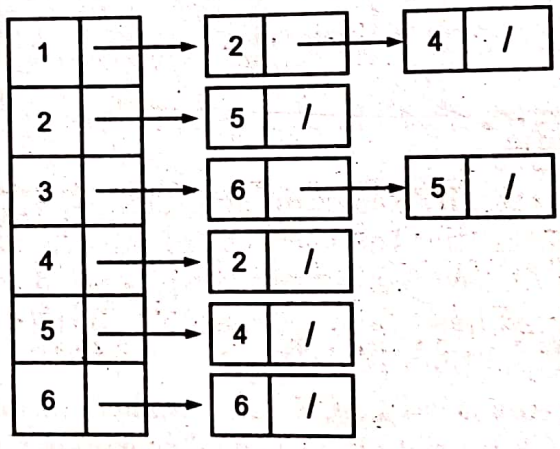


Fig. 12.25.

Solution: First, we create adjacency-list representation.



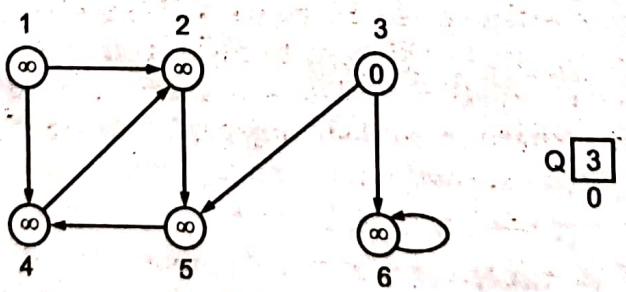


Fig. 12.26.

So, adj[3] = [6, 5] so color [6] = GRAY
color [5] = GRAY

and $d[6] = 1$ $d[5] = 1$

Now, $\pi[6] \leftarrow 3$ $\pi[5] \leftarrow 3$

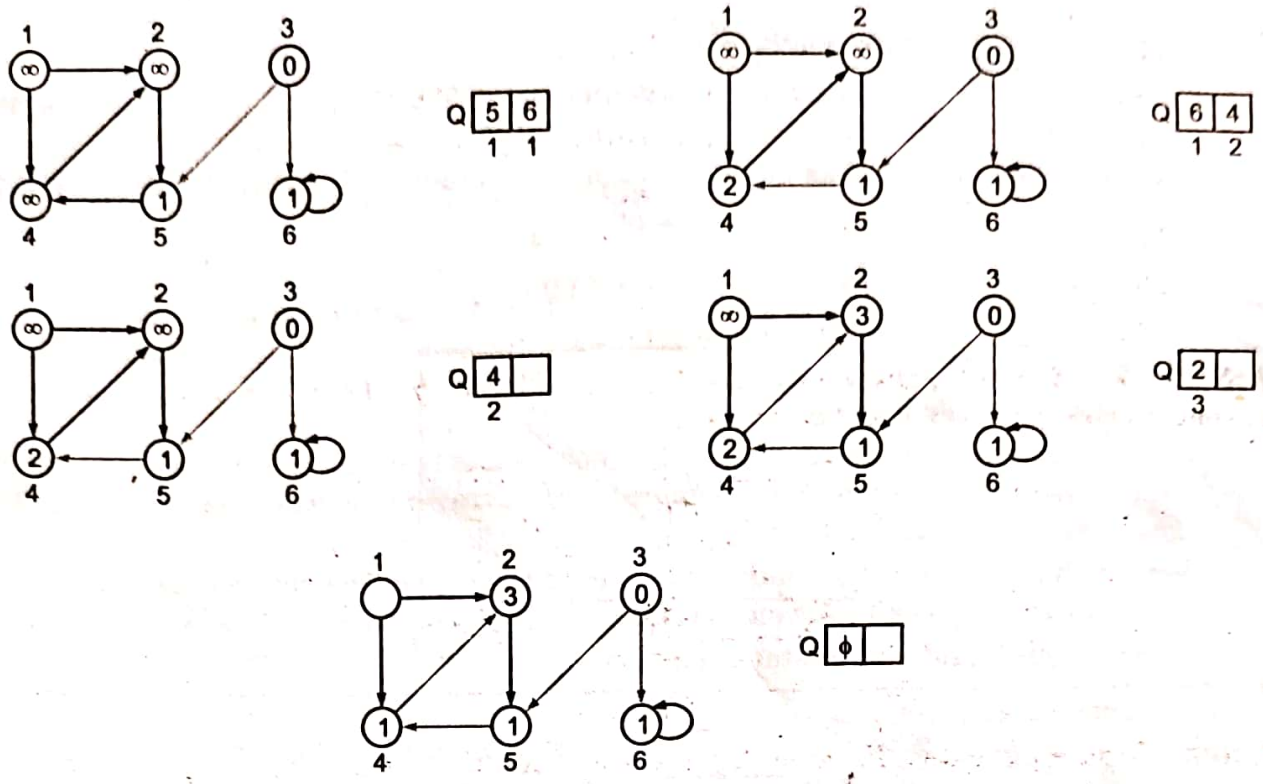


Fig. 12.27.

Thus vertex 1 cannot be reachable from source.

12.9 DEPTH FIRST SEARCH

The Depth First Search (DFS), as its name suggest, is to search deeper in the graph, whenever possible. Given as input graph $G = (V, E)$ and a source vertex S, from where the searching starts. First we visit the starting node, then we travel through each node along a path, which begins at S. That is, we visit a neighbour vertex of S and again a neighbour of a neighbour of S and so on. DFS also works on both directed as well as on undirected graphs.

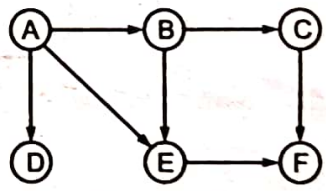


Fig. 12.28.

DFS uses **stack**.

Let us take the graph (see Fig. 12.28).

Suppose node A as the starting node. First, we will traverse node A. Then, we will traverse any node adjacent to node A. Suppose we traverse node B, then traverse node E, which is adjacent to node B, then traverse adjacent node of E, *i.e.*, node F. Now there is no node adjacent to node F, *i.e.*, dead end. So we will move backward. Till now the traversal is:

$$A, B, E, F$$

Now, we reach node E, see if there is any node adjacent to it. There is no such node. So we will move backward. Now, we reach node B, see if there is any node adjacent to it and not traversed yet. Here node is C. So we traverse it. Now there is no untraversed node adjacent to node C. So, we will move backward. On reaching node A we see that node D is adjacent to it and has not been traversed. So we traverse it. Now there is no untraversed node adjacent to node D. We can't move forward and we can't move backward also, so we will stop. So, the traversal is:

A, B, E, F, C, D.

Depth First Search Through Stack

Depth Search technique uses stack. Take an array **STACK**, which will be used to keep the unvisited neighbours of the node. Take a another Boolean array **VISITED**, which will have value **TRUE** if the node has been *visited* and will have value **FALSE** if the node has *not been visited*.

Initially stack is empty and TOP = – 1.

Initially VISITED [i] = FALSE where $i = 1$ to n, n is total number of nodes.

Procedure

1. Push starting node into the stack.
2. Pop an element from the stack, if it has not been traversed then traverse it, if it has already been traversed then just ignore it. After traversing make the value of visited array true for this node.
3. Now push all the unvisited adjacent nodes of the popped element on stack. Push the element even if it is already on the stack.
4. Repeat steps 3 and 4 until stack is empty.

Let us consider a graph.

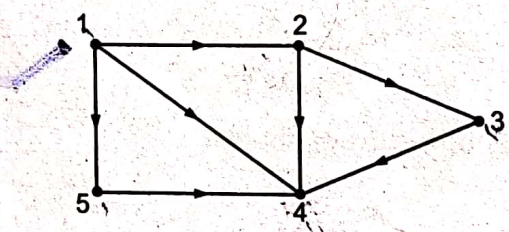


Fig. 12.29.

Suppose the starting node for traversal is 1.

Step 1 : Push node 1 into stack.

Now, TOP = 0 and STACK = 1.

Step 2 : POP node 1 from stack and traverse it

So, Traversed node = 1 and VISITED [1] = TRUE

Now push all the unvisited adjacent nodes 2, 5, 4 of the popped element on the stack.

Now TOP = 2 and STACK = 2, 5, 4

Traversal = 1.

Step 3 : POP the element node 4 from the stack, traversed it and push all its unvisited adjacent nodes. There is no adjacent nodes. So,

Traversed node = 4

VISITED [4] = TRUE

TOP = 1 STACK = 2, 5

Traversal = 1, 4

Step 4 : POP the element 5 from the stack, traverse it and push all its unvisited adjacent nodes. Here node 4 is adjacent node of 5 but it is visited node. So,

Traversed node = 5

VISITED [5] = TRUE

TOP = 0 STACK = 2

Traversal = 1, 4, 5

Step 5 : Pop the element 2 from the stack, traverse it and push all its unvisited adjacent nodes, *i.e.*, node 3.

Now, Traversed node = 2
VISITED[2] = TRUE
TOP = 0 STACK = 3
Traversal = 1, 4, 5, 2

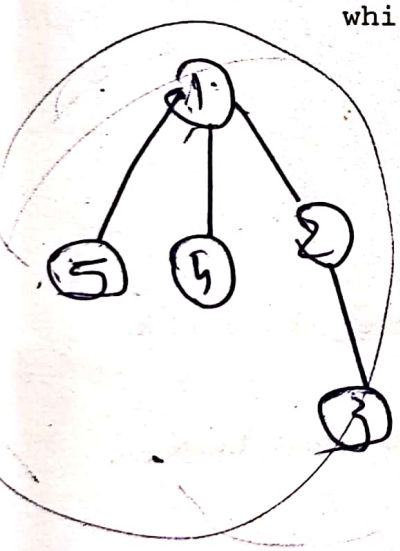
Step 6 : POP the element 3 from the stack, traverse it and push all its unvisited adjacent nodes. No node is here.

Now, Traversed node = 3
VISITED[3] = TRUE
TOP = – 1 STACK = NULL
Traversal = 1, 4, 5, 2, 3

Since the stack is empty, so we will stop our process. The function for DFS is as follows :

C Implementation

```
DFS (int v)
    {
        int i, stack[MAX], TOP = - 1, POP, j, k;
        int ch;
        TOP ++;
        stack[TOP] = v;
        while(TOP>=0)
            {
                POP = stack[TOP];
                TOP--;
                if(visited[POP] == false)
                    {
                        printf("%d",pop);
                        visited[pop] = true;
                    }
                else
                    continue
                for(i=n; i>=1; i--)
                    {
                        if(adj[pop][i] == 1 && visited[i] == false)
                            {
                                top++;
                                stack[top]=i;
                            }
                    }
            }
    }
```



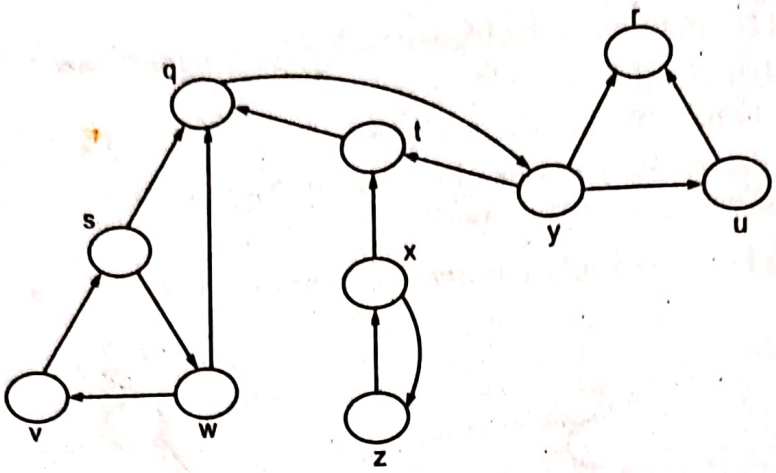


Fig. 12.45.

Call DFS (G^T).

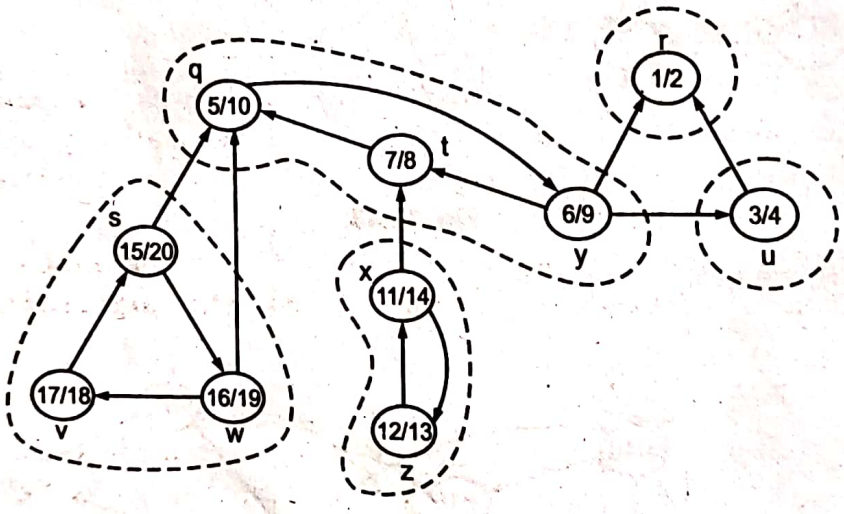


Fig. 12.46.

Thus, strongly connected components are

$$\{r\}, \{u\}, \{q, t, y\}, \{x, z\}, \{s, v, w\}$$

12.12 SPANNING TREE

A spanning tree of a graph is just a sub-graph that contains all the vertices and is a tree. That is, a spanning tree of a connected graph G contains all the vertices and has the edges which connects all the vertices. So, the number of edges will be 1 less than the number of nodes.

Let us take a connected graph.

Here all these trees are spanning tree and the number of edges is one less than the number of nodes.

If graph is not connected, *i.e.*, a graph with n vertices has edges less than $n - 1$ then no spanning tree is possible. A graph may have many spanning trees. **DFS and BFS spanning trees** are the spanning trees which are obtained by DFS and BFS traversal respectively.

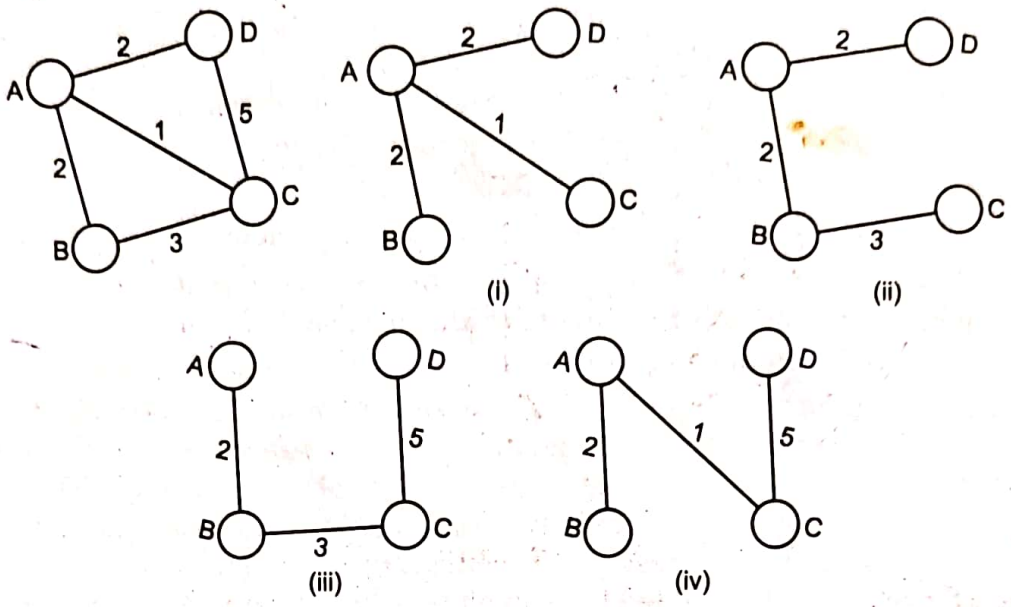


Fig. 12.47.

12.13 MINIMUM SPANNING TREES

Given a connected weighted graph G, it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible. Such a tree is called a **minimum spanning tree** and represents the "cheapest" way of connecting all the nodes in G.

There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are **Prim's** and **Kruskal algorithm.**

12.14 KRUSKAL'S ALGORITHM

This algorithm creates a *forest of trees.* Initially, the forest consists of n single node trees and no edges. That is, in the method initially we take n distinct trees for all n nodes of the graph. At each step, we add one (the cheapest one) edge, so that it joins two trees together. If it forms a cycle, it does simply links two nodes that were already part of a single connected tree, so that this edge does *not included.*

The steps are as follows :

1. Initially construct a separate tree for each node in a graph.
2. Edges are placed in a *priority queue, i.e.,* we take edges in ascending order. We can use a heap for the priority queue.
3. Until we have added $n - 1$ edges.
   (*a*) Extract the cheapest edge from the queue.
   (*b*) If it forms a cycle, reject it
   else
   add it to the forest.
4. Whenever we insert an edge, two trees will be joined, *i.e.,* every step will have joined two trees in the forest together so that at the end, there will be only one tree.

In this method, first we examine all the edges one-by-one starting from the smallest edge. To decide whether the selected edge should be included in the spanning tree or not, we will examine the two nodes connecting the edge. If the two nodes belong to the same tree

then we will not include the edge in the spanning tree, since the two nodes are in the same tree, they are already connected and adding this edges would result in a cycle. So we will insert an edge in the spanning tree only if it's nodes are in different trees.

Now, we will see how to decide whether two nodes are in the same tree or not. For this, we need a **UNION-FIND** structure.

To understand the UNION-FIND structure, we need to look at a **partition** of a set.

(*a*) Every element of the set belongs to one of the sets in the partition.

(*b*) No element of the set belongs to more than one of the sub-sets.

(*c*) Every element of a set belongs to one and only one of the sets of a partition.

A partition of a set may be thought of as a set of equivalence classes. Each sub-set of the partition contains a set of equivalent elements. For each subset, we denote one element as the representative of that subset. Each element in the sub-set is, some how, equivalent and represented by the representative. When we add elements to the sub-set, we arrange that all the elements point to their representative. Initially, each node is its own representative. As the initial pairs of nodes are joined to form a tree, the representative pointer of one of the nodes is made to point to the other, which becomes the representative of the tree. As trees are joined, the representative pointer of the representative of one of them is set to point to any element of the other. Let x denote an object, we wish to support the following operations.

1. **MAKE -SET (x)** creates a new set whose only member (and thus representative) is pointed to by x.
2. **FIND-SET (x)** returns a pointer to the representative of the set containing x.
3. **UNION (x, y)** unites the dynamic sets that contain x and y, say S_x and S_y into a new set that is the union of these two sets. The representative of the resulting set is some member of $S_x \cup S_y$. Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y removing them from the collection.

MST-KRUSKAL (G, W)

```
1.  A ← ϕ.
2.  for each vertex v ∈ V[G]
        do MAKE-SET (v)
3.  sort the edges of E into increasing order by weight w.
4.  for each edge (u, v) ∈ E taken in increasing order.
        do if FIND-SET (u) ≠ FIND-SET (v)
            then A ← A U {u, v}
                UNION (u, v)
5.  return A.
```

EXAMPLE 12.6. Find the minimum spanning tree of the following graph using Kruskal algorithm.

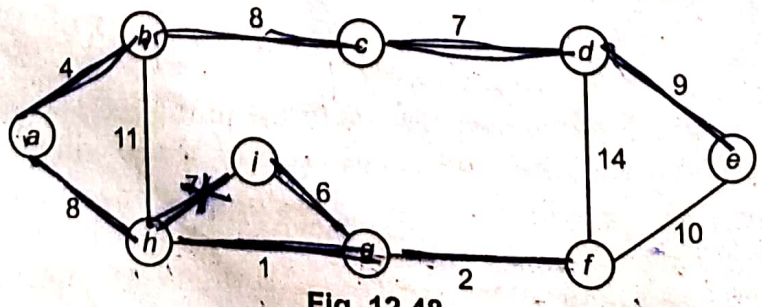


Fig. 12.48.

Solution: First we initialise the set A to the empty set and create $|v|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, *i.e.*,

Edge	*Weight*
(h, g)	1
(g, f)	2
(a, b)	4
(i, g)	6
(h, i)	7
(c, d)	7
(b, c)	8
(a, h)	8
(d, e)	9
(e, f)	10
(b, h)	11
(d, f)	14

Now, check for each edge (u, v), whether the end points u and v belong to the same tree. If they do, then the edge (u, v) cannot be added. Otherwise, the two vertices belong to different trees and the edge (u, v) is added to A and the vertices in the two trees are merged in by UNION procedure.

So, first take (h, g) edge

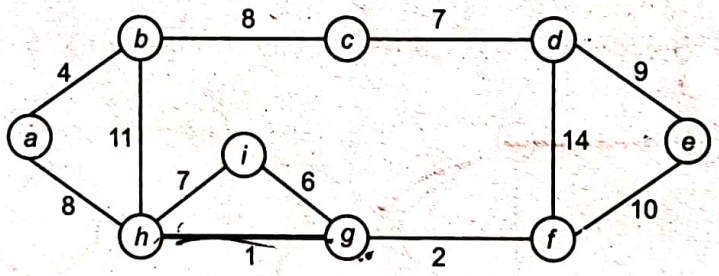


Fig. 12.49.

the (g, f) edge.

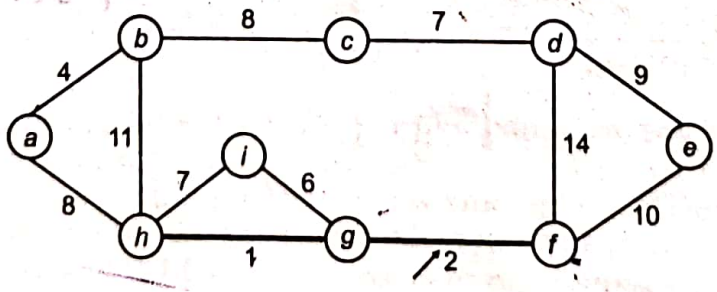


Fig. 12.50.

then (a, b) and (i, g) edges are considered and forest becomes.

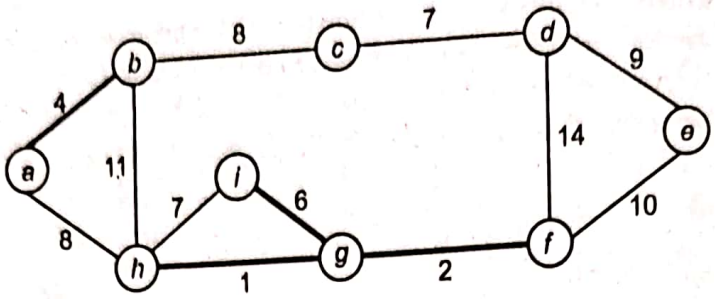


Fig. 12.51.

Now, edge (h, i). Both h and i vertices are in same set, thus it creates a cycle. So this edge is *discarded*.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are connected and forest becomes.

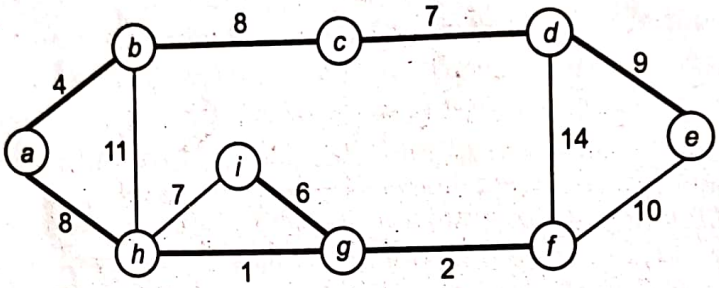


Fig. 12.52.

In (e, f) edge both end point e and f exist in same tree so *discarded* this edge. Then (b, h) edge, it also creates a cycle.

After that edge (d, f) and the final spanning tree is shown as in dark lines.

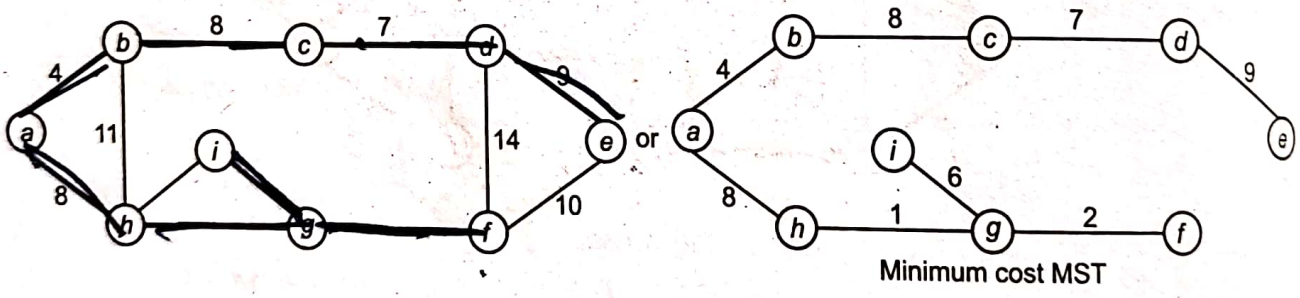


Fig. 12.53.

12.15 PRIM'S ALGORITHM

In this, we start with any node and add the other node in spanning tree on the basis of weight of edge connecting to that node. Suppose we start with the node 'u' then we have to need to see all the connecting edges and which edge has minimum weight. Then we will add that edge and node to the spanning tree. Suppose if two nodes u_1 and u_2 are in spanning tree and both have edge connecting to v_3 then edge. Which has minimum weight will be added in spanning tree.

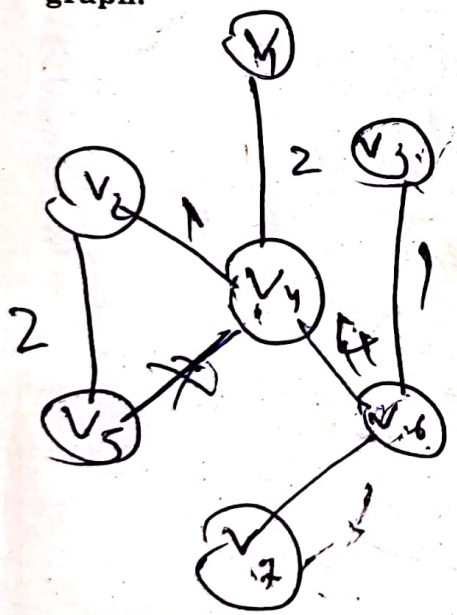
The main idea of Prim's algorithm is similar to that of **Dijkstra's algorithm** (discussed later) for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A *always from a single tree*. We begin with some vertex v in a given graph $G = (V, E)$, defining the initial set of vertices A. Then, in each iteration, we choose a minimum

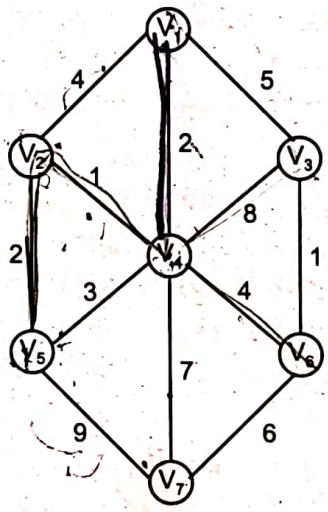
weight edge (u, v) connecting a vertex v in the set A to the vertex u outside to set A. Then vertex u is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edge in set A form a MST.

```
MST-Prim (G, w, r)
1.  for each u ∈ V[G]
2.       do key[u] ← ∞
3.            π[u] ← NIL
4.  key[r] ← 0
5.  Q ← V[G]
6.  while Q ≠ ϕ
7.       do u ← EXTRACT-MIN (Q)
8.            for each v ∈ Adj[u]
9.                 do if v ∈ Q and w (u, v) < key [v]
10.                     then π[u] ← u
11.                          key[v] ← w(u, v)
```

EXAMPLE 12.7. Find the minimum spanning tree vising Prim's algorithm for the given graph.

(GBTU, B.Tech., 2011-12)





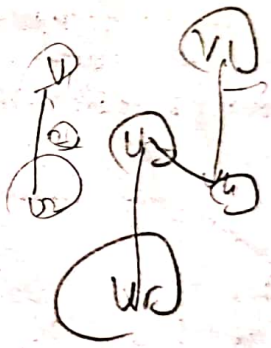


Fig. 12.54.

Solution: First we intialize the priority queue Q and key of each vertex to ∞ except for the root whose key is o. Suppose V_1 vertex is the root.

$$\text{Adj } [V_1] = \{V_2, V_3, V_4\}$$

Removing V_1 from the set Q and adds it in the tree.

Now, update the key and π fields of every vertex adjacent to V_1 but not in the tree.

$$\text{key } [V_2] = \infty$$

$$\omega (V_1, V_2) = 4 \quad i.e., \quad \omega (V_1, V_2) < \text{key } [V_2]$$

So, $\quad \pi [V_2] = V_1 \quad \text{and} \quad \text{key } [V_2] = 4$

Similarly, $\quad \text{key } [V_2] = \infty$

$$\omega(V_1, V_3) = 5 \quad i.e., \quad \omega(V_2, V_3) < \text{key}[V_3]$$

So, $\pi[V_3] = V_1$ and key $[V_3] = 5$

and $\pi[V_4] = V_1$ and key $[V_4] = 2$

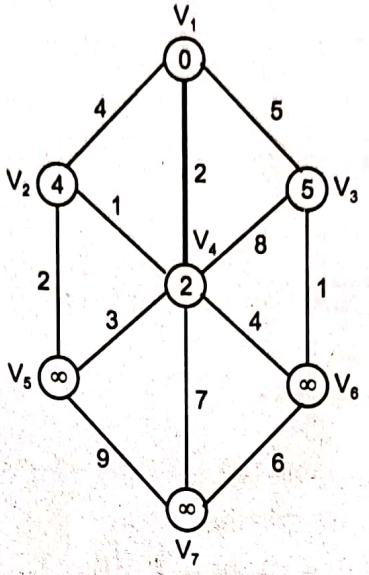


Fig. 12.55.

Now, by EXTRACT_MIN (Q). Remove V_4 node becuase key $[V_4] = 2$, which is minimum.

Adj $[V_4] = \{V_1, V_2, V_3, V_5, V_6, V_7\}$

Removing V_4 from the set Q and adds it to the set $V - Q$ of vertices in the tree where V_1 is already there.

key $[V_2] = 4$

So, $\omega(V_2, V_4) = 1 \quad \omega(V_2, V_4) < \text{key}[V_2]$

$\pi[V_2] = V_4$ and key $[V_2] = 1$

Similarly, $\pi[V_5] = V_4$ key $[V_5] = 3$

$\pi[V_7] = V_4$ key $[V_7] = 7$

$\pi[V_6] = V_4$ key $[V_6] = 4$

But $\omega(V_3, V_4) = 8$ and key $[V_3] = 5$

i.e., $\omega(V_3, V_4) > \text{key}[V_3]$

So, no change and V_1 is not in the Q.

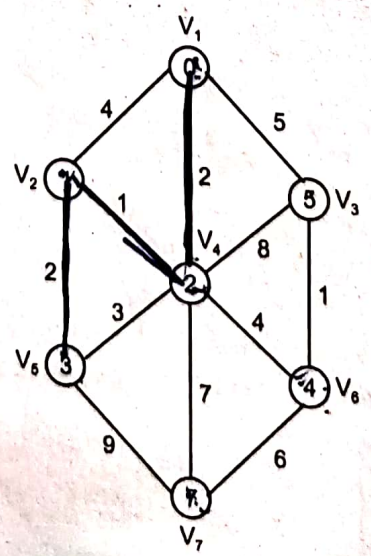


Fig. 12.56.

Now, By EXTRACT_MIN(Q). Remove V_2 because key $[V_2] = 1$ which is minimum and adds it to the set V-Q of vertices in the tree. The elements in tree now $\{V_1, V_4, V_2\}$

$$\text{Adj } [V_2] = \{V_1, V_4, V_5\}$$

V_1, V_4 are not in the Q so

$$\omega (V_2, V_5) = 2$$
$$\text{key } [V_5] = 3$$
$$\omega (V_2, V_5) < \text{key } [V_5]$$

So, $\pi (V_5) = V_2$ and key $[V_5] = 2$

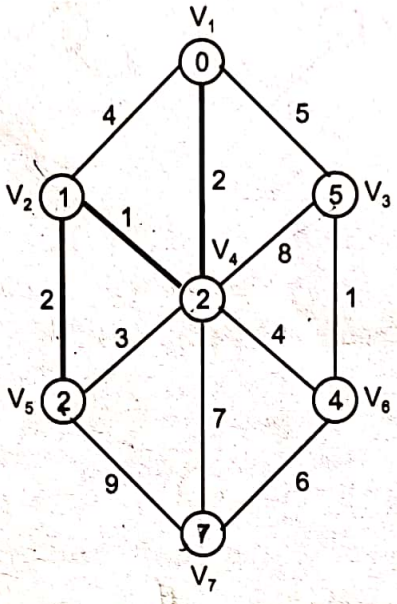


Fig. 12.57.

Now, by EXTRACT_MIN (Q). Remove V_5 because key $[V_5] = 2$ which is minimum. Now,

$$\text{Adj } [V_5] = \{V_2, V_4, V_7\}$$

where V_2, V_4 not in Q. So,

$$\text{key } [V_7] = 7 \text{ and } \omega (V_5, V_7) = 9$$
$$\omega (V_5, V_7) > \text{key } [V_7]. \text{ No change.}$$

Now, by EXTRACT_MIN (Q). Remove V_6 because key $[V_6] = 4$

$$\text{Adj } [V_6] = \{V_3, V_4, V_7\}$$
$$\omega (V_3, V_6) = 1, \quad \text{key } [V_3] = 5$$
$$\omega (V_3, V_6) < \text{key } [V_3]$$

So, key $[V_3] = 1$ and $\pi [V_3] = V_6$

Similarly, $\omega (V_6, V_7) = 6$ key $[V_7] = 7$

$$\omega (V_6, V_7) < \text{key } [V_7]$$

So, key $[V_7] = 6$ and $\pi [V_7] = V_6$

Now, Remove V_3 because key $[V_3] = 1$

$$\text{Adj } [V_3] = \{V_1, V_4, V_6\}$$

No node in Q.

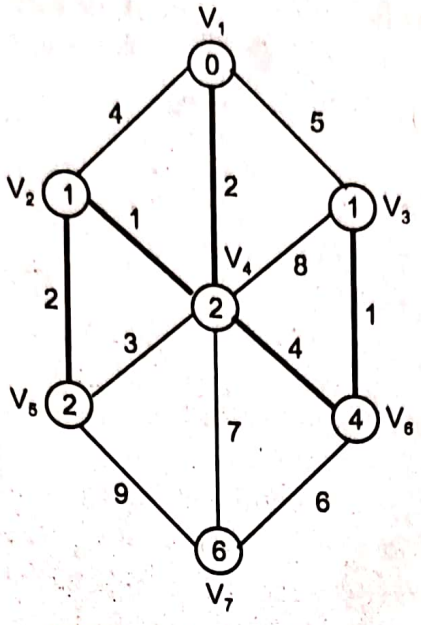


Fig. 12.58.

Now, by EXTRACT_MIN (Q) remove V_7 which is the last node in Q. So, final spanning tree is

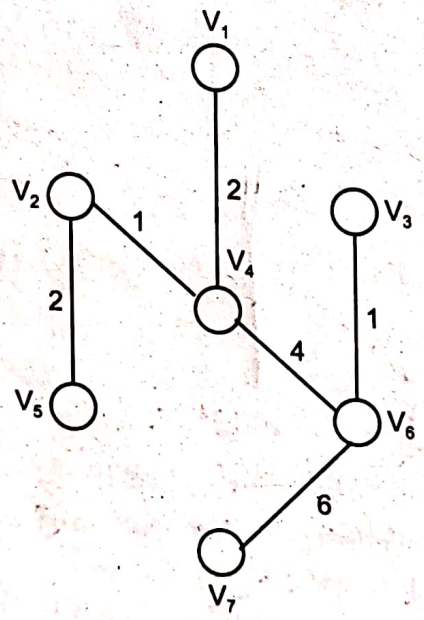


Fig. 12.59.

The edges that belong to MST are

(V_1, V_4), (V_4, V_2), (V_2, V_5), (V_4, V_6), (V_6, V_3), (V_6, V_7)

Weight of MST will be

$$2 + 1 + 2 + 4 + 1 + 6 = 16.$$

EXAMPLE 12.8. Generate minimum cost spanning tree for the following graph using Prim's algorithm.

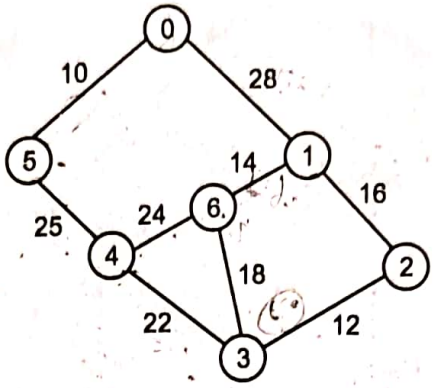


Fig. 12.60.

Solution: In Prim's algorithm, first we initialise the priority queue Q to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, *i.e.*, r. By EXTRACT-MIN (Q) produre, now $u = r$ and Adj[u] = [5, 1].

Removing u from the set Q and adds it to the set $V - Q$ of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree.

$$key[5] = \infty$$

$$w(0, 5) = 10 \quad i.e., \quad w(u, v) < key[5]$$

so, $\pi[5] = 0$ and $key[5] = 10$.

and $key[1] = \infty$

$$w(0, 1) = 28 \quad i.e., \quad w(u, v) < key[5]$$

so $\pi[1] = 0$ and $key[1] = 28$.

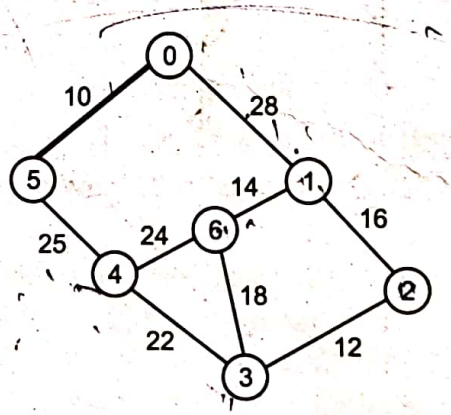


Fig. 12.61.

Now, by EXTRACT_MIN (Q) Remove 5 because key [5] = 10 which is minimum so $u = 5$

Adj[5] = {4}

$w(u, v) < key\ [v]$ then $key\ [4] = 25$

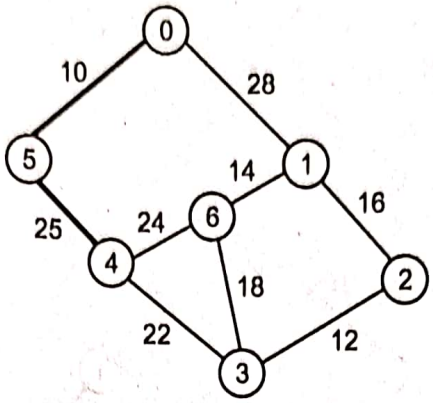


Fig. 12.62.

Now remove 4 because $key[4] = 25$ which is minimum so $u = 4$

Adj[4] = {6, 3}

$w(u, v) < key\ [v]$ then $key\ [6] = 24$

$key\ [3] = 22$

Now remove 3 because $key\ [3] = 22$ is minimum so $u = 3$

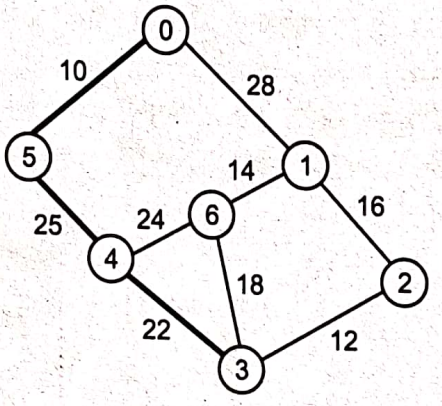


Fig. 12.63.

Adj[3] = {4, 6, 2}

$4 \notin Q$ $key\ [6] = 24$ now becomes $key\ [6] = 18$

$key\ [2] = 12$

Now, in Q $key\ [2] = 12$, key [6] = 18, key [1] = 28

By EXTRACT_MIN (Q) Remove 2, because key [2] = 12 is minimum

Adj[2] = {3, 1}

$3 \notin Q$

Now, $key\ [1] = 16$.

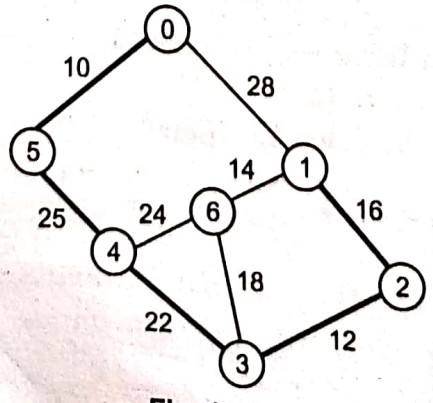


Fig. 12.64.

By EXTRACT_MIN (Q) Remove 1 because key [1] = 16 is minimum

Adj[1] = {0, 6, 2}

{0, 2} $\notin$ Q, *key* [6] = 14.

Now because Q contains only one vertex 6. By EXTRACT_MIN remove it

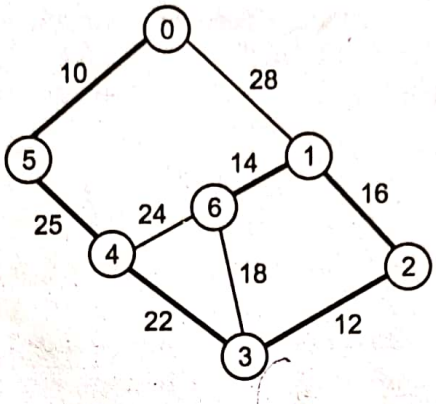


Fig. 12.65.

Thus, the final spanning tree is

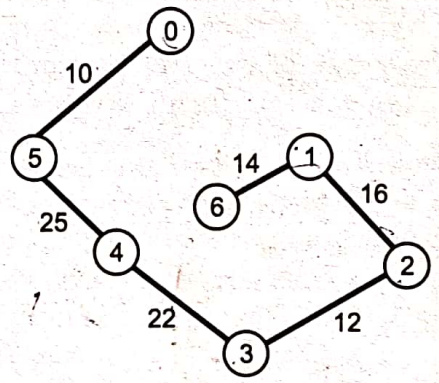


Fig. 12.66.

12.16 SHORTEST PATHS

A path from source vertex s to t is shortest path from s to t if there is no path from s to t with lower weights.

In a **shortest-paths problem,** we are given a weighted directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$ mapping edges to real-valued weights. The weight of path $p = < v_0, v_1, \ldots\ldots v_k >$ is the sum of the weights of its constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

we define the shortest-path weight from u to v by

$$\delta(u, v) = \min (w(p) : u \rightarrow v),$$

If there is a path from u to v. The shortest paths are not necessarily unique.

(a) Shortest Path for Given Source and Destination :

In order to find a shortest path from a given source to given destination, we discuss **Dijkstra's shortest-path algorithm.**

Relaxation

The single-source shortest-paths algorithms are based on a technique known as *relaxation*, a method that repeatedly decreases an upper bound on the actual shortest-path weight of each vertex until the upper bound equals the shortest-path weight. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a **shortest-path estimate**. We initialize the shortest-path estimates and predecessors by the following procedure.

```
INITIALIZE-SINGLE-SOURCE (G, s)
1.   for each vertex v ∈ V[G]
2.        do d[v] ← ∞
3.        π[v] ← NIL
4.   d[s] ← 0
```

d[v] : represents the shortest distance of v from the source and

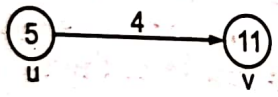
π[v] : represents the predecessor of node v, *i.e.*, the node which precedes the given node in the shortest-path from source.

After initialisation, $\pi[v] = \text{NIL}$ for all $v \in V$, $d[v] = 0$ for $v = s$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The **process of relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$. A relaxation step may decrease the value of the shortest-path-estimate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

```
RELAX (u, v, w)
1.   if d[v] > d[u] + w(u, v)
2.        then d[v] ← d[u] + w(u, v)
3.             π[v] ← u
```

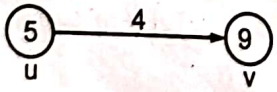
For example,



Here, $d[u] = 5$
$d[v] = 11$
$w(u, v) = 4$

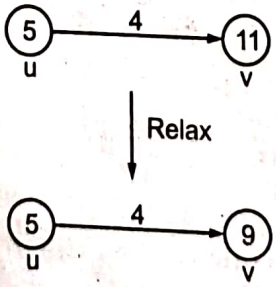
i.e.,
$d[v] > d[u] + w(u, v)$
$11 > 5 + 4$
$11 > 9$

By apply RELAX, we get



i.e., $d[v] = 9$
$d[u] = 5$ and $w(u, v) = 4$

i.e.,



12.17 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist **Edsger Dijkstra**, is a greedy algorithm that solves the single-source shortest path problem for a directed

graph $G = (V, E)$ with non-negative edge weights, *i.e.*, we assume that $w(u, v) \geq 0$ each edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set S *of* vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. We maintain a priority queue Q that contains all the vertices in $v - s$, keyed by their d values. Graph G is represented by adjacency lists.

```
DIJKSTRA (G, w, s)
1.   INITIALIZE-SINGLE-SOURCE (G, s)
2.   S ← ϕ
3.   Q ← V[G]
4.   while Q ≠ ϕ
5.        do u ← EXTRACT-MIN (Q)
6.             S ← S ∪ {u}
7.             for each vertex v ∈ Adj[u]
8.             do RELAX (u, v, w)
```

Because Dijkstra's always chooses the "lightest" or "closest" vertex in $V - S$ to insert into set S, we say that it uses a **greedy strategy.**

Dijkstra's algorithm bears some similarity to both **breadth-first search** and **Prim's algorithm** for computing minimum spanning trees. It is like breadth-first search in that set S correspond to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest- path weights, so black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), insert this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

EXAMPLE : Consider *A* as a source vertex.

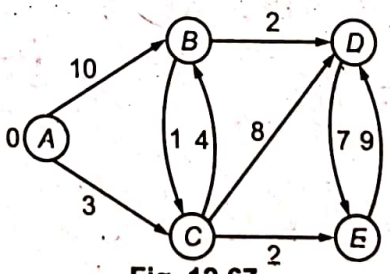
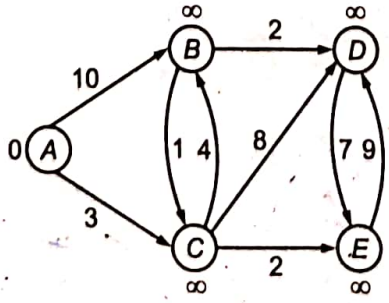


Fig. 12.67.

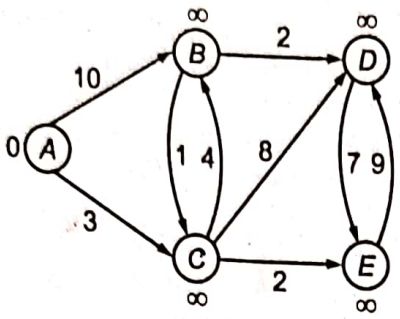
Initialise :



S : {}

Q:	A	B	C	D	E
A	0	∞	∞	∞	∞
B					
C					
D					

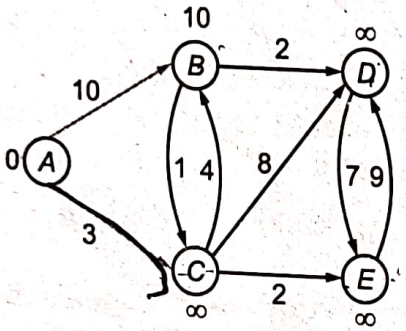
"A" ← EXTRACT-MIN (Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S : $\{A\}$

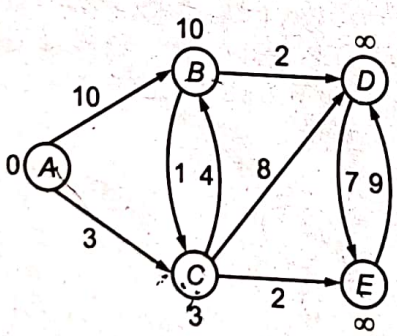
Relax all edges leaving A :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S : $\{A\}$

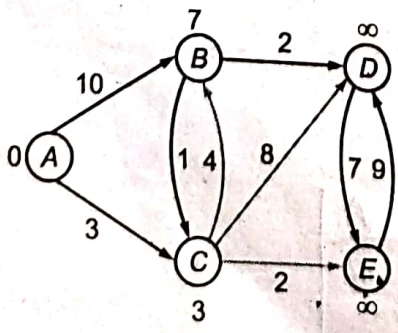
"C" ← EXTRACT-MIN (Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		–	–	–	–

S : $\{A, C\}$

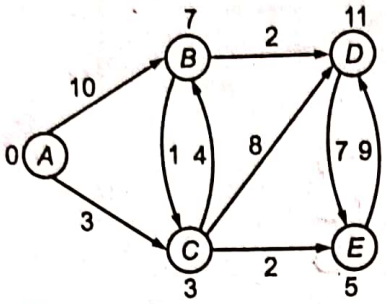
Relax all edges leaving C :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7	–	11	5

S : $\{A, C\}$

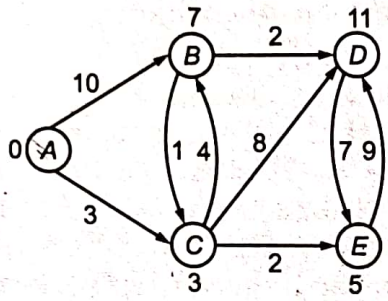
"E" ← EXTRACT-MIN (Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S : $\{A, C, E\}$

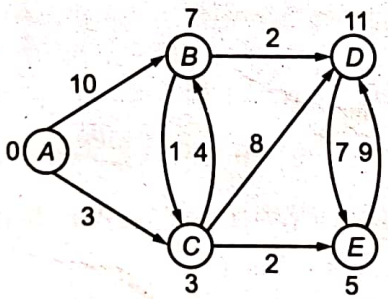
Relax all edges leaving E :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S : $\{A, C, E\}$

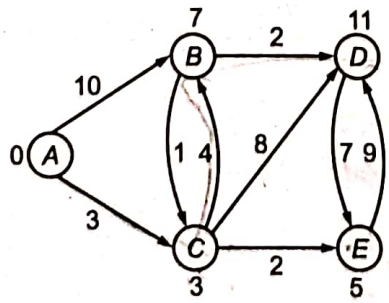
"B" ← EXTRACT-MIN (Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S : $\{A, C, E, B\}$

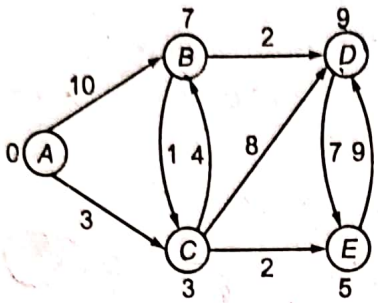
Relax all edges leaving B :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S : $\{A, C, E, B\}$

"*D*" ← EXTRACT-MIN (*Q*) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
A		10	3	∞	∞
B		7		11	5
C		7		11	
D				9	

S : {*A*, *C*, *E*, *B*, *D*}

12.18 WARSHALL'S ALGORITHM

The algorithm considers the "*intermediate*" vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2, \ldots v_m \rangle$ is any vertex of p other than v_1 or v_m, that is, any vertex in the set $\{v_2, v_3, \ldots v_{m-1}\}$.

The Floyd-Warshall algorithm is based on the following observation. Let the vertices of G be $V = \{1, 2, \ldots, n\}$, and consider a sub-set $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple, since we assume that G contains no negative-weight cycles). The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $(1, 2, \ldots, k)$.

If k is intermediate vertex of path p, then we break p down into $i \xrightarrow{p_1} k \xrightarrow{p_2} j$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

and

$$w_{ij} = \begin{cases} 0 & \text{if } i = k \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

Thus,

$$d_{ij}^{0} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Let us define the $V \times V$ matrix

$$D^{(m)} = (d_{ij}^{(m)})$$

$d_{ij}^{(m)}$ = the length of the shortest path from i to j with $\leq$ m edges.

When $m = 0$, there is a shortest path from i to j with no edges if and only if $i = j$.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } i \neq j, (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

```
FLOYD-WARSHELL (W)
1.  n ← rows [W]
2.  D(0) ← W
3.  for k ← 1 to n
4.      do for i ← 1 to n
5.          do for j ← 1 to n
6.              do d_ij^(k) ← min (d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7.  return D(n)
```

Constructing a Shortest Path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshell algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. This method can be implemented to run in $O(n^3)$ time. Given predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure can be used to print the vertices on a given shortest path.

We can compute the predecessor matrix Π just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $(1, 2, \ldots, k)$.

The recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i = j \text{ and } w_{ij} < \infty \end{cases}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$ then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $(1, 2, \ldots, k-1)$. Otherwise , we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $(1, 2, \ldots, k-1)$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

EXAMPLE 12.9. Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix $D^{(k)}$ that results each iteration.

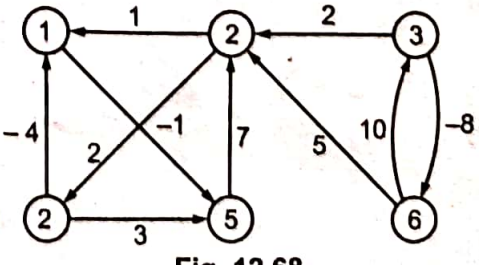


Fig. 12.68.

Solution:

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

We know,

$$d_{ij}^{(k)} = \min [d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

k	i	j	$d_{ij}^{(k-1)}$	$d_{ik}^{(k-1)}$	$d_{kj}^{(k-1)}$	d_{ij}^{k}
1	1	1	0	0	0	0
1	1	2	∞	0	∞	∞
1	1	3	∞	0	∞	∞
1	1	4	∞	0	∞	∞
1	1	5	−1	0	−1	−1
1	1	6	∞	0	∞	∞
1	2	1	1	1	0	1
1	2	2	0	1	∞	0
1	2	3	∞	1	∞	∞
1	2	4	2	1	∞	2
1	2	5	∞	1	−1	0
1	2	6	∞	1	∞	∞
1	3	1	∞	∞	0	∞
1	3	2	2	∞	∞	2
1	3	3	0	∞	∞	0
1	3	4	∞	∞	∞	∞
1	3	5	∞	∞	−1	∞
1	3	6	−8	∞	∞	−8
1	4	1	−4	−4	0	−4
1	4	2	∞	−4	∞	∞
1	4	3	∞	−4	∞	∞
1	4	4	0	−4	∞	0
1	4	5	3	−4	−1	−5
1	4	6	∞	−4	∞	∞
1	5	1	∞	∞	0	∞
1	5	2	7	∞	∞	7
1	5	3	∞	∞	∞	∞
1	5	4	∞	∞	∞	∞
1	5	5	0	∞	−1	0
1	5	6	∞	∞	∞	∞
1	6	1	∞	∞	0	∞
1	6	2	5	∞	∞	5
1	6	3	10	∞	∞	10
1	6	4	∞	∞	∞	∞
1	6	5	∞	∞	−1	∞
1	6	6	0	∞	∞	0

So, the $D^{(1)}$ matrix is

$D^{(1)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	−8
4	−4	∞	∞	0	−5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Similarly, we get

$D^{(2)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(3)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(4)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	∞	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

$D^{(5)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	2	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

and

$D^{(6)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	(−5)	−3	0	−1	−6	−8
4	−4	2	∞	0	(−5)	∞
5	5	7	∞	9	0	∞
6	3	(5)	10	7	2	0

The shortest path from 3 to 1 is

$3 \rightarrow 6 \rightarrow 2 \rightarrow 4 \rightarrow 1$ and is -5 units.

Similarly, if we want to find the shortest path from 6 node to 2 node then it is 5 units $[6 \rightarrow 2]$ and the shortest path from 4 to 5 is -5, *i.e.*, $4 \rightarrow 1 \rightarrow 5$.