Z/8 100/-

Chapter 6

Graph

A graph G is a collection of two sets V & E where V is the collection of vertices $v_0, v_1, \ldots\ldots v_{n-1}$ also called nodes and E is the collection of edges $e_1, e_2, \ldots.. e_n$ where an edge is an arc which connects two nodes. This can be represented as-

$G = (V,E)$

$V(G) = (v_0, v_1, \ldots\ldots v_n)$ or set of vertices

$E(G) = (e_1, e_2, \ldots\ldots e_n)$ or set of edges

A graph can be of two types-

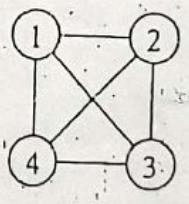
1. Undirected graph
2. Directed graph

Undirected Graph-

A graph, which has unordered pair of vertices, is called undirected graph. Suppose there is an edge between v_0 & v_1 then it can be represented as (v_0, v_1) or (v_1, v_0) also

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

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



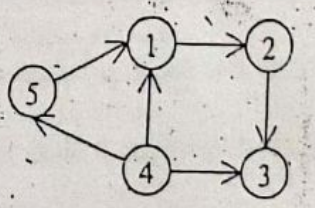
Here we can see this graph has 4 nodes and 6 edges.

Directed Graph-

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. In this type of graph each edge has direction, means <v1,v2> and <v2,v1> will represent different edges. Here the edge means that a direction will be associated with that edge. Directed graph is also known as digraph.

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

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



This graph has 5 nodes and 6 edges.

Weighted graph A graph is said to be weighted if it's edges have been assigned some non negative value as weight. A weighted graph is also known as network. Graph G9 is a weighted graph.

Adjacent nodes A node u is adjacent to another node or is a neighbour of another node

v if there is an edge from node u to node v. In undirected graph if (v_0,v_1) is an edge then v_0 is adjacent to v_1 and v_1 is adjacent to v_0. In a digraph if $<v_0,v_1>$ is an edge then v_0 is adjacent to v_1 and v_1 is adjacent from v_0..

Incidence In an undirected graph the edge (v_0,v_1) is incident on nodes v_0 and v_1. In a digraph the edge $<v_0,v_1>$ is incident from node v_0 and is incident to node v_1.

Path A path from node u_0 to node u_n is a sequence of nodes u_0,u_1,u_2,u_3------- u_{n-1}, u_n such that u_0 is adjacent to u_1, u_1 is adjacent to u_2 ---------,u_{n-1} is adjacent to u_n. In other words we can say that (u_0,u_1) ,(u_1,u_2),(u_2,u_3)----------(u_{n-1},u_n) are edges.

Length of path Length of a path is the total number of edges included in the path.

Closed path A path is said to be closed if first and last nodes of the path are same.

Simple path. Simple path is a path in which all the nodes are distinct with an exception that the first and last nodes of the path can be same.

Cycle Cycle is a simple path in which first and last nodes are the same or we can say that a closed simple path is a cycle.

In a digraph a path is called a cycle if it has one or more nodes and the start node is connected to the last node. In graph G7, path ACBA is a cycle and in graph G9 path ABA is a cycle.

In an undirected graph a path is called a cycle if it has at least three nodes and the start node is connected to the last node. In undirected graph if (u,v) is an edge then u-v-u should not be considered a path since (u,v) and (v,u) are the same edges. So for a path to be a cycle in an undirected graph there should be at least three nodes.

Cyclic graph A graph that has cycles is called a cyclic graph.

Acyclic graph A graph that has no cycles is called an acyclic graph.

Dag A directed acyclic graph is named as dag after its acronym. Graph G5 is an example of a dag.

Degree In an undirected graph, the number of edges connected to a node is called the degree of that node, or we can say that degree of a node is the number of edges incident on it. In graph G2 degree of node A is 1,degree of node B is zero. In graph G3 degree of node A is 3,of node B is 2.

In a digraph, there are two degrees for every node known as indegree and outdegree.

Indegree The indegree of a node is the number of edges coming to that node or in other words edges incident to it. In graph G8, the indegree of nodes A, B, D and G are 0, 2, 6 and 1 respectively.

Outdegree The outdegree of node is the number of edges going outside from that node, or in other words the edges incident from it. In graph G8, outdegrees of nodes A, B, D, F, and G are 3, 1, 6, 3, and 2 respectively.

Source A node, which has no incoming edges, but has outgoing edges, is called a source. The indegree of source is zero. In graph G8, nodes A and F are sources.

Sink A node, which has no outgoing edges but has incoming edges, is called a sink. The outdegree of a sink is zero. In graph G8, node D is a sink.

Pendant node A node is said to be pendant if its indegree is equal to 1 and outdegree is equal to 0.

Reachable If there is a path from a node to any other node then it will be called as reachable from that node.

Isolated node If a node has no edges connected with any other node then its degree will be 0 and it will be called isolated node, In graph G2, node B is an isolated node.

Successor and predecessor In digraph if a node v_0 is adjacent to node v_1 ,then v_0 is the predecessor of v_1 , and v_1 is the successor of v_0. In graph G, node A is predecessor of node B ,node B is predecessor of node C, node B is successor of node A and node C is successor of node B.

Connected graph An undirected graph is connected if there is a path from any node of graph to any other node, or any node is reachable from any other node. Graph G1 and G3 are connected graphs while graph G2 is not a connected graph.

Strongly connected A digraph is strongly connected if there is a directed path from any node of graph to any other node. We can also say that a digraph is strongly connected if for any pair of nodes u and v, there is a path from u to v and also a path from v to u. Graph G7 is a strongly connected graph.

Weakly connected A digraph 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 digraph we remove the directions and the resulting undirected graph is connected then that digraph is weakly connected. Graph G6 is a weakly connected graph.

Maximum edges in graph In an undirected graph there can be n(n-1)/2 maximum edges and in a digraph there can be n(n-1) maximum edges, where n is the total number of nodes in the graph.

Complete graph A graph is complete if any node in the graph is adjacent to all the nodes of the 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.

Multiple edge If between a pair of nodes there are more than one edges then they are known as multiple edges or parallel edges. In graph G3, there are multiple edges between nodes A and C.

Loop An edge will be called loop or self edge if it starts and ends on the same node. Graph G4 has a loop at node B.

Multigraph A graph which has loop or multiple edges can be described as multigraph. Graphs G3 and G4 are multigraphs.

Regular graph A graph is regular if every node is adjacent to the same number of nodes. Graph G1 is regular since every node is adjacent to 3 nodes.

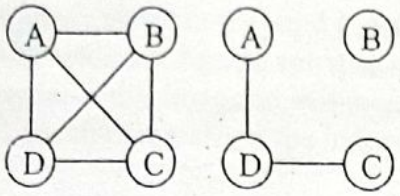
Planar graph A graph is called planar if it can be drawn in a plane without any two edges intersecting. Graph G1 is not a planar graph, while graphs G2, G3, G4 are planar graphs.

Articulation point If on removing a node from the graph the graph becomes disconnected then that node is called the articulation point.

Bridge If on removing an edge from the graph the graph becomes disconnected then that edge is called the bridge.

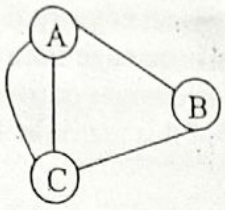
Biconnected graph A graph with no articulation points is called a biconnected graph.

Tree An undirected connected graph will be called tree if there is no cycle in it. So we can see that the tree structure, which we studied in earlier chapter, is a special form of graph structure. In tree structure, any node can have many children but it will have only one parent, while in graph structure any node can have many children and many parents. Graph G10 is a tree.

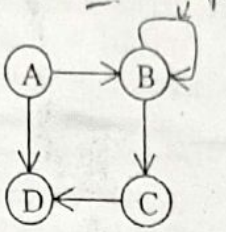


G1

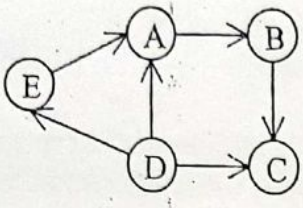
G2



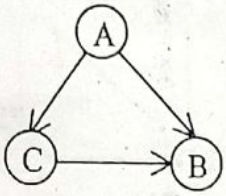
G3



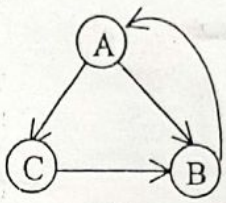
G4



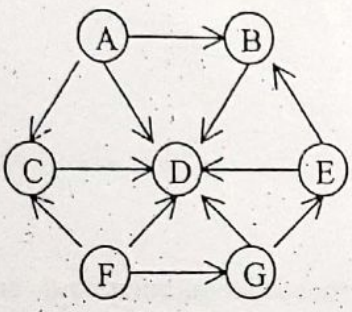
G5



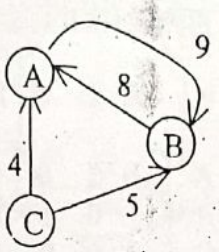
G6



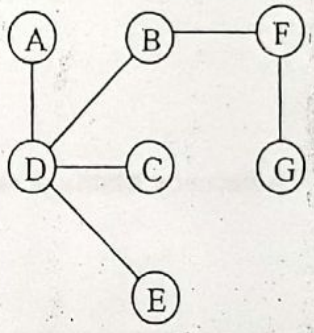
G7



G8



G9



G10

Representation of Graph-

We have mainly two components in graph, nodes & edges. Now we have to design the data structure to keep these components in mind. There are two ways for representing the graph in computer memory. First one is the sequential representation and second one is linked list representation.

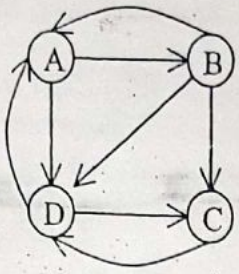
1. Adjacency Matrix-

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 node is adjacent to any other node or not. We know very well that we can represent a matrix in two dimensional array of n x n or array[n][n], where first subscript will be row and second subscript will be column of that matrix. Suppose there are 4 nodes in graph then row1 represents the node1, row2 represents the node2 and so on. Similarly column1 represents node1, column2 represents node2 and so on. The entry of this matrix will be as-

Arr[i][j] = 1 If there is an edge from node i to node j
= 0 If there is no edge from node i to node j

Hence, all the entries of this matrix will be either 1 or 0.

Let us take a graph-



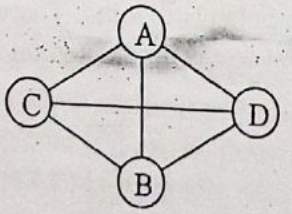
The corresponding adjacency matrix for this graph will be-

Adjacency Matrix A =

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

This adjacency matrix is maintained in the array arr[4][4]. Here the entry of matrix arr[0][1] = 1, which represents there is an edge in the graph from node A to node B. Similarly arr[2][0] = 0, which represents there is no edge from node C to node A.

Let us take an undirected graph-



The corresponding adjacency matrix for this graph will be-

Adjacency Matrix A =

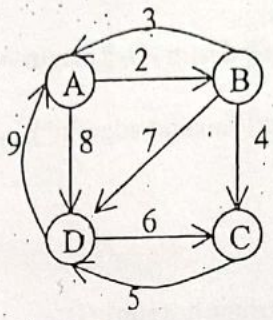
	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

Note that the adjacent matrix for an undirected graph will be a symmetric matrix. This implies that for every i and j, A[i][j]=A[j][i] in an undirected graph.
In an undirected graph rowsum and columnsum for a node is same and represents the degree of that node and in directed graph rowsum represents the outdegree and columnsum represents the indegree of that node.

Suppose a graph has some weight on its edge then the elements of adjacency matrix will be defined as-

Arr[i][j] = Weight on edge If there is an edge from node i to node j.
= 0 Otherwise

Let us take the same graph-



The corresponding adjacency matrix will be as-

Weighted Adjacency Matrix W =

	A	B	C	D
A	0	2	0	8
B	3	0	4	7
C	0	0	0	5
D	9	0	6	0

Here all the elements of matrix represent the weight on that edge.

```
/* Program for creation of adjacency matrix */
#include<stdio.h>
#define max 20

int adj[max][max]; /*Adjacency matrix */
int n;  /* Denotes number of nodes in the graph */

main( )
{
        int max_edges,i,j,origin,destin;
        char graph_type;
        printf("Enter number of nodes : ");
        scanf("%d",&n);
```

```
        printf("Enter type of graph, directed or undirected (d/u) : ");
        fflush(stdin);
        scanf("%c",&graph_type);

        if(graph_type=='u')
                max_edges=n*(n-1)/2;
        else
                max_edges=n*(n-1);

        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d( 0 0 to quit ) : ",i);
                scanf("%d %d",&origin,&destin);
                if( (origin==0) && (destin==0) )
                        break;
                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                {
                        adj[origin][destin]=1;
                        if( graph_type=='u')
                                adj[destin][origin]=1;
                }
        }/*End of for*/

        printf("The adjacency matrix is :\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        printf("%4d",adj[i][j]);
                printf("\n");
        }
}/*End of main( )*/
```

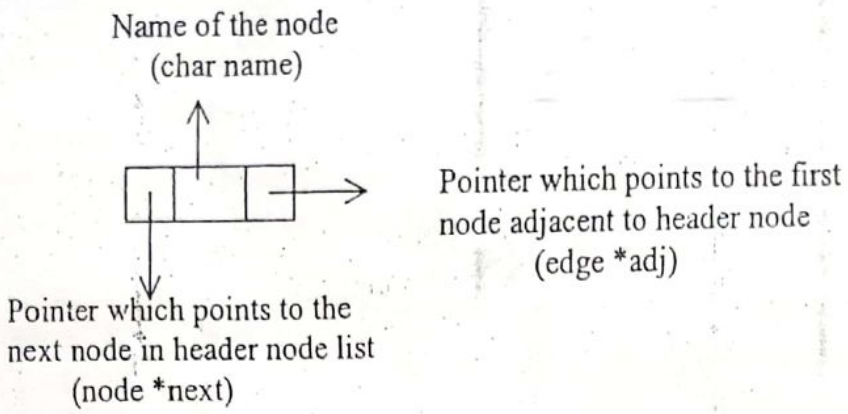
2. Adjacency List-

If the adjacency matrix of the graph is sparse then it is more efficient to represent the graph through adjacency list.

In adjacency list representation of graph, we will maintain two lists. First list will keep track of all the nodes in the graph and second list will maintain a list of adjacent nodes for each node. Suppose there are n nodes then we will create one list which will keep information of all nodes in the graph and after that we will create n lists., where each list will keep information of all adjacent nodes of that particular node.

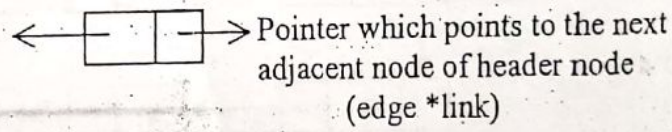
Each list has a header node, which will be the corresponding node in the first list.

Structure of header node



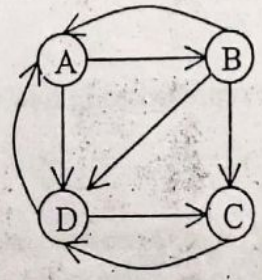

```
struct node{
        struct node *next;
        char name;
        struct edge *adj;
      };
```

Structure of edge

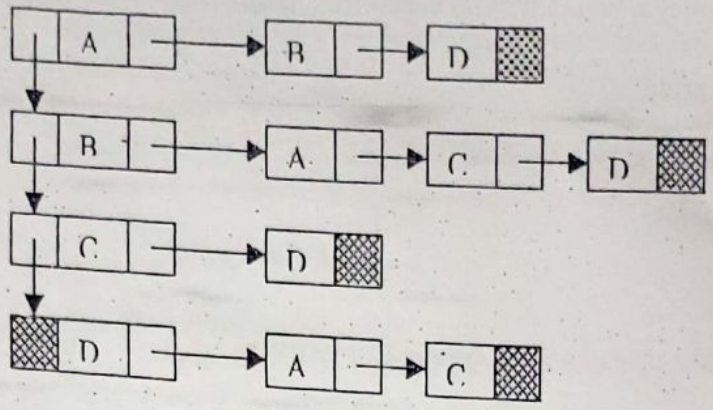



```
struct edge{
        char dest;
        struct edge *link;
      };
```

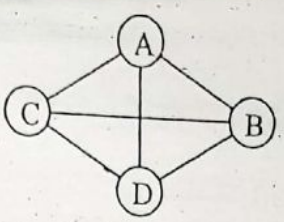
Let us take the same graph-



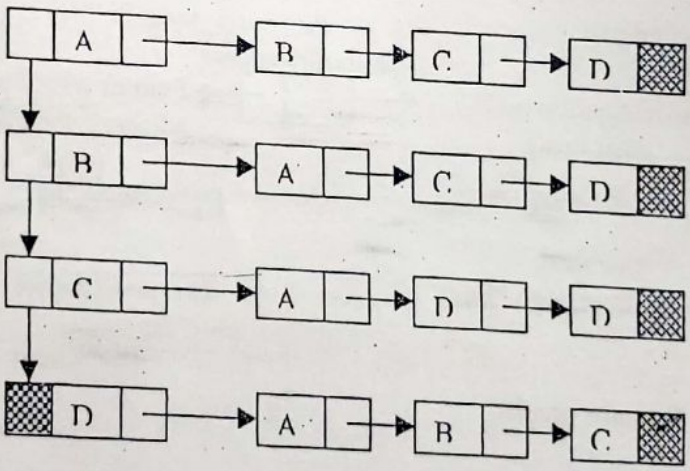
The adjacency list for this graph will be as-



Let us take an undirected graph-



The adjacency list for this graph will be as -



Operations on Graph-

As we have seen, a graph can be represented in two ways-
1. Adjacency Matrix
2. Adjacency list

The two main operations on graph will be-
1. Insertion
2. Deletion

Traversing of graph will be described later.

Here insertion and deletion will be also on two things-
1. On node
2. On edge
Now we will describe insertion and deletion operation on adjacency matrix and adjacency list.

Insertion in Adjacency Matrix-

Node insertion-
Insertion of node requires only addition of one row and and one column with zero entries in that row and column. Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Suppose we want to add one node E in the graph then we have a need to add one row and one column with all zero entries for node E.

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	0	0	1	0
D	1	0	1	0	0
E	0	0	0	0	0

Edge insertion-
We know that entry of adjacency matrix 1 represents the edge between two nodes, and 0 represents no edge between those two nodes. Therefore, insertion of edge requires changing the value 0 into 1 for those particular nodes.
Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Here we can see that there is no edge between D to B. So adjacency matrix has 0 entry at 4th row 2nd column. Suppose we want to insert an edge between D to B, then we have a need to change the 0 entry into 1 at the position 4th row 2nd column. Now the adjacency matrix will be-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	1	1	0

Deletion in adjacency matrix-

Node deletion-

Deletion of node requires deletion of that particular row and column in adjacency matrix for node to be deleted, because node deletion requires deletion of all the edges which are connected to that particular node.

Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Suppose we want to delete the node D, then 4th row and 4th column of adjacency matrix will be deleted. Now the adjacency matrix will be-

	A	B	C
A	0	1	0
B	1	0	1
C	0	0	0

Edge deletion-

Deletion of an edge requires changing the value 1 to 0 for those particular nodes.

Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Here we can see that an edge exists between node B to node C. So adjacency matrix has entry 1 at 2nd row 3rd column. Suppose we want to delete the edge which is in between B and C, then we have a need to change the entry 1 to 0 at the position 2nd row, 3rd column. Now the adjacency matrix will be-

	A	B	C	D
A	0	1	0	1
B	1	0	0	1
C	0	0	0	1
D	1	0	1	0

```
/*Program for addition and deletion of nodes and edges of graph using adjacency matrix */
#include<stdio.h>
#define max 20
int adj[max][max];
int n;
main( )
{
        int choice;
        int node,origin,destin;

        create_graph( );
        while(1)
        {
                printf("1.Insert a node\n");
                printf("2.Insert an edge\n");
                printf("3.Delete a node\n");
                printf("4.Delete an edge\n");
                printf("5.Dispaly\n");
                printf("6.Exit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1:
                        insert_node( );
                        break;
                case 2:
                        printf("Enter an edge to be inserted : ");
                        scanf("%d %d",&origin,&destin);
                        insert_edge(origin,destin);
                        break;
                case 3:
                        printf("Enter a node to be deleted : ");
                        scanf("%d",&node);
                        delete_node(node);
                        break;
                case 4:
                        printf("Enter an edge to be deleted : ");
                        scanf("%d %d",&origin,&destin);
```

```
                    del_edge(origin,destin);
                    break;
                case 5:
                    display( );
                    break;
                case 6:
                    exit( );
                default:
                    printf("Wrong choice\n");
                    break;
                }/*End of switch*/
        }/*End of while*/
}/*End of main( )*/

create_graph( )
{
        int i,max_edges,origin,destin;

        printf("Enter number of nodes : ");
        scanf("%d",&n);
        max_edges=n*(n-1); /* Taking directed graph */

        for(i=1;i<=max_edges;i++)
        {
            printf("Enter edge %d( 0 0 ) to quit : ",i);
            scanf("%d %d",&origin,&destin);
            if((origin==0) && (destin==0))
                break;
            if( origin > n || destin > n || origin<=0 || destin<=0)
            {
                printf("Invalid edge!\n");
                i--;
            }
            else
                adj[origin][destin]=1;
        }/*End of for*/
}/*End of create_graph( )*/

display( )
{
        int i,j;
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
                printf("%4d",adj[i][j]);
            printf("\n");
        }
}/*End of display( )*/
```

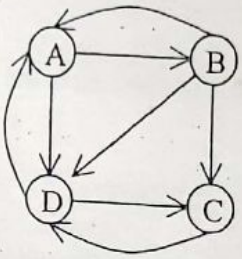
```
insert_node( )
{
        int i;
        n++;   /*Increase number of nodes in the graph*/
        printf("The inserted node is %d \n",n);
        for(i=1;i<=n;i++)
        {
                adj[i][n]=0;
                adj[n][i]=0;
        }
}/*End of insert_node( )*/

delete_node(char u)
{
        int i,j;
        if(n==0)
        {
                printf("Graph is empty\n");
                return;
        }
        if( u>n )
        {
                printf("This node is not present in the graph\n");
                return;
        }
        for(i=u;i<=n-1;i++)
          for(j=1;j<=n;j++)
          {
                adj[j][i]=adj[j][i+1];  /* Shift columns left */
                adj[i][j]=adj[i+1][j];  /* Shift rows up */
          }
        n--; /*Decrease the number of nodes in the graph */
}/*End of delete_node( )*/

insert_edge(char u,char v)
{
        if(u > n)
        {
                printf("Source node does not exist\n");
                return;
        }
        if(v > n)
        {
                printf("Destination node does not exist\n");
                return;
        }
        adj[u][v]=1;
}/*End of insert_edge( )*/
```

```
del_edge(char u,char v)
{
   if(u>n || v>n || adj[u][v]==0)
   {
        printf("This edge does not exist\n");
        return;
   }
   adj[u][v]=0;
}/*End of del_edge( )*/
```

Insertion in adjacency list-

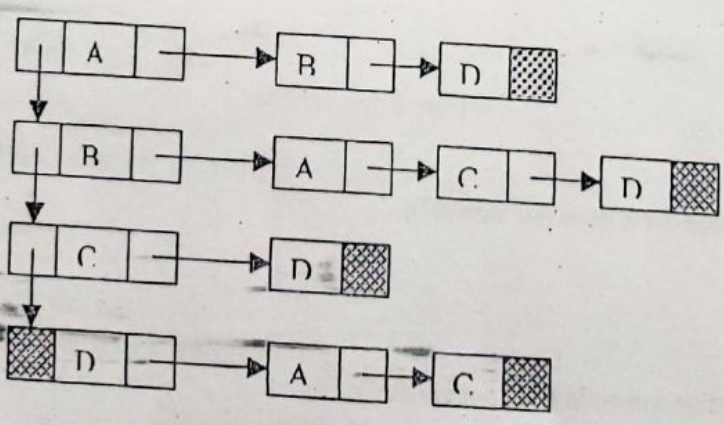
Node insertion-

Insertion of node in adjacency list requires only insertion of that node in header nodes of adjacency list. Let us take a graph G-



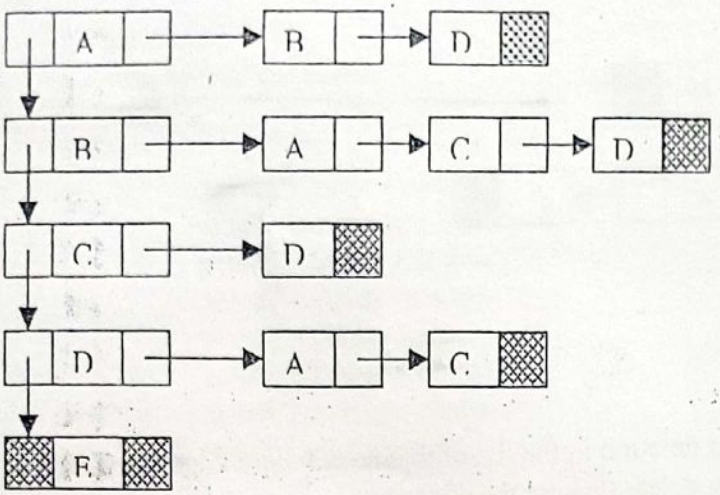
The adjacency list for this graph will be as-

Header Nodes



Suppose we want to insert one node E then it requires addition of node E in header node only. Now the adjacency list will be-

Header Nodes

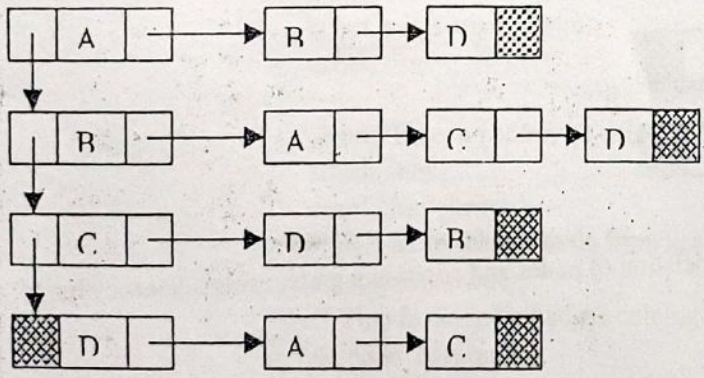


Edge insertion-

Insertion of an edge requires add operation in the list of the starting node of edge. Remember here graph is directed graph. In undirected graph, it will be added in the list of both nodes.

Suppose we want to add the edge which starts from node C and ends at node B, then add operation is needed in the list of C node. Now the adjacency list of graph will be-

Header Nodes



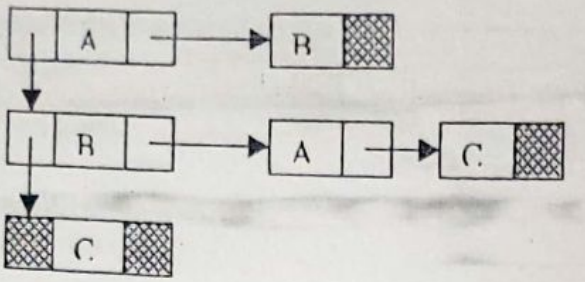
Deletion in adjacency list-

Node deletion-

Deletion of node requires deletion of that particular node from header node and from the entire list wherever it is coming. Deletion of node from header will automatically free the list attached to that node.

Suppose we want to delete the node D from graph then we have a need to delete node D from header node and from the list of A, B and C nodes. Now the adjacency list of graph will be-

Header Nodes

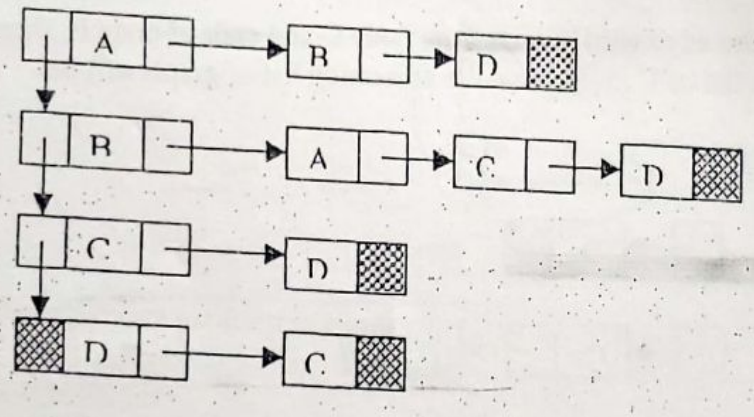


Edge deletion-

Deletion of edge requires deletion in the list of that node where edge starts, and that element of the list will be deleted where the edge ends.

Suppose we want to delete the edge which starts from D and ends at A then we have a need to delete in the list of node D and element in the list deleted will be A. Now the adjacency list of the graph will be-

Header Nodes




```
/* Program for insertion and deletion of nodes and edges in a graph using adjacency list */
#include<stdio.h>
struct edge;
struct node
{
        struct node *next;
        char name;
        struct edge *adj;
}*start=NULL;

struct edge
{
        char dest;
        struct edge *link;
};
struct node *find(char item);
```

```
main( )
{
        int choice;
        char node,origin,destin;
        while(1)
        {
                printf("1.Insert a node\n");
                printf("2.Insert an edge\n");
                printf("3.Delete a node\n");
                printf("4.Delete an edge\n");
                printf("5.Display\n");
                printf("6.Exit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1:
                        printf("Enter a node to be inserted : ");
                        fflush(stdin);
                        scanf("%c",&node);
                        insert_node(node);
                        break;
                case 2:
                        printf("Enter an edge to be inserted : ");
                        fflush(stdin);
                        scanf("%c %c",&origin,&destin);
                        insert_edge(origin,destin);
                        break;
                case 3:
                        printf("Enter a node to be deleted : ");
                        fflush(stdin);
                        scanf("%c",&node);
                        /*This fn deletes the node from header node list*/
                        delete_node(node);
                        /* This fn deletes all edges coming to this node */
                        delnode_edge(node);
                        break;
                case 4:
                        printf("Enter an edge to be deleted : ");
                        fflush(stdin);
                        scanf("%c %c",&origin,&destin);
                        del_edge(origin,destin);
                        break;
                case 5:
                        display( );
                        break;
                case 6:
                        exit( );
```

```
                default:
                        printf("Wrong choice\n");
                        break;
                }/*End of switch*/
        }/*End of while*/
}/*End of main( )*/

insert_node(char node_name)
{
        struct node *tmp,*ptr;
        tmp=malloc(sizeof(struct node));
        tmp->name=node_name;
        tmp->next=NULL;
        tmp->adj=NULL;

        if(start==NULL)
        {
                start=tmp;
                return;
        }
        ptr=start;
        while( ptr->next!=NULL)
                ptr=ptr->next;
        ptr->next=tmp;
}/*End of insert_node( )*/

delete_node(char u)
{
      struct node *tmp,*q;
      if(start->name == u)
      {
              tmp=start;
              start=start->next;  /* first element deleted */
              free(tmp);
              return;
      }
      q=start;
      while(q->next->next != NULL)
      {
              if(q->next->name==u)   /* element deleted in between */
              {
                      tmp=q->next;
                      q->next=tmp->next;
                      free(tmp);
                      return;
              }
              q=q->next;
      }/*End of while*/
```

```
        if(q->next->name==u)   /* last element deleted */
        {
                tmp=q->next;
                free(tmp);
                q->next=NULL;
        }
}/*End of delete_node( )*/

delnode_edge(char u)
{
        struct node *ptr;
        struct edge *q,*start_edge,*tmp;
        ptr=start;
        while(ptr!=NULL)
        {
                /* ptr->adj points to first node of edge linked list */
                if(ptr->adj->dest == u)
                {
                        tmp=ptr->adj;
                        ptr->adj=ptr->adj->link;  /* first element deleted */
                        free(tmp);
                        continue; /* continue searching in another edge lists */
                }
                q=ptr->adj;
                while(q->link->link != NULL)
                {
                        if(q->link->dest==u)  /* element deleted in between */
                        {
                                tmp=q->link;
                                q->link=tmp->link;
                                free(tmp);
                                continue;
                        }
                        q=q->link;
                }/*End of while*/
                if(q->link->dest==u)   /* last element deleted */
                {
                        tmp=q->link;
                        free(tmp);
                        q->link=NULL;
                }
                ptr=ptr->next;
        }/*End of while*/
}/*End of delnode_edge( )*/

insert_edge(char u,char v)
{
        struct node *locu,*locv;
        struct edge *ptr,*tmp;
```

```
        locu=find(u);
        locv=find(v);

        if(locu==NULL )
        {
                printf("Source node not present ,first insert node %c\n",u);
                return;
        }
        if(locv==NULL )
        {
                printf("Destination node not present ,first insert node %c\n",v);
                return;
        }
        tmp=malloc(sizeof(struct edge));
        tmp->dest=v;
        tmp->link=NULL;

        if(locu->adj==NULL)  /* item added at the begining */
        {
                locu->adj=tmp;
                return;
        }
        ptr=locu->adj;
        while(ptr->link!=NULL)
                ptr=ptr->link;
        ptr->link=tmp;

}/*End of insert_edge( )*/

struct node *find(char item)
{
        struct node *ptr,*loc;
        ptr=start;
        while(ptr!=NULL)
        {
                if(item==ptr->name)
                {
                        loc=ptr;
                        return loc;
                }
                else
                        ptr=ptr->next;
        }
        loc=NULL;
        return loc;
}/*End of find( )*/
```

```
del_edge(char u,char v)
{
        struct node *locu,*locv;
        struct edge *ptr,*tmp,*q;
        locu=find(u);

        if(locu==NULL )
        {
                printf("Source node not present\n");
                return;
        }
        if(locu->adj->dest == v)
        {
                tmp=locu->adj;
                locu->adj=locu->adj->link;  /* first element deleted */
                free(tmp);
                return;
        }
        q=locu->adj;
        while(q->link->link != NULL)
        {
                if(q->link->dest==v)  /* element deleted in between */
                {
                        tmp=q->link;
                        q->link=tmp->link;
                        free(tmp);
                        return;
                }
                q=q->link;
        }/*End of while*/
        if(q->link->dest==v)   /* last element deleted */
        {
                tmp=q->link;
                free(tmp);
                q->link=NULL;
                return;
        }
        printf("This edge not present in the graph\n");
}/*End of del_edge( )*/

display( )
{
        struct node *ptr;
        struct edge *q;

        ptr=start;
        while(ptr!=NULL)
        {
                printf("%c ->",ptr->name);
```

```
            q=ptr->adj;
            while(q!=NULL)
            {
                printf(" %c",q->dest);
                q=q->link;
            }
            printf("\n");
            ptr=ptr->next;
        }
}/*End of display( )*/
```

Path Matrix-

Let us take a graph G with n nodes $v_1, v_2, \ldots\ldots v_n$. The path matrix or reachable matrix of G can be defined as-

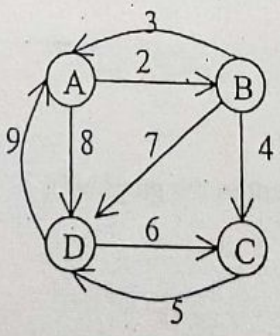
$$P[i][j] = \begin{cases} 1 & \text{if there is a path in between } v_i \text{ to } v_j \\ 0 & \text{Otherwise.} \end{cases}$$

If there is a path from v_i to v_j then it can be a simple path from v_i to v_j of length n-1 or less or there can be a cycle of length n or less.

A graph G will be strongly connected if there are no zero entries in path matrix means a path exists between all v_i to v_j and v_j to v_i also

Computing Path matrix from powers of adjacency matrix-

Let us take a graph and compute path matrix for it from its adjacency matrix.



The corresponding weighted adjacency matrix will be as-

Weighted Adjacency Matrix W =

	A	B	C	D
A	0	2	0	8
B	3	0	4	7
C	0	0	0	5
D	9	0	6	0

The adjacency matrix will be

$$\text{Adjacency Matrix A} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Path length denotes the number of edges in the path. Adjacency matrix is the path matrix of path length 1. Now if we multiply the adjacency matrix with itself then we get the path matrix of length 2.

$$AM_2 = A^2 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix} \end{array}$$

In this matrix, value of $AM_2[i][j]$ will represent the number of paths of path length 2 from node v_i to v_j. For example here node A has 2 paths of path length 2 to node C, and node D has 2 paths of path length 2 to itself, node C has 1 path of path length 2 to node A.

To obtain the path matrix of length 3 we will multiply the path matrix of length 2 with adjacency matrix.

$$AM_3 = AM_2 * A = A^3 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 2 & 1 & 4 \\ 3 & 1 & 3 & 3 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 3 & 1 \end{bmatrix} \end{array}$$

Here $AM_3[i][j]$ will represent the number of paths of path length 3 from node v_i to node v_j. For example, here node A has 4 paths of path length 3 to node D, and node D has no path of path length 3 to node B.

Similarly, we can find out the path matrix for path length 4.

$$AM_4 = AM_3 * A = A^4 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 6 & 1 & 6 & 4 \\ 4 & 3 & 4 & 7 \\ 3 & 0 & 3 & 1 \\ 1 & 3 & 1 & 6 \end{bmatrix} \end{array}$$

Let us take a matrix X where

$Xn=AM_1+AM_2+----------AMn$

Here X[i][j] has the value of number of paths ,less than or equal to length n ,from node v_i to v_j. Here n is the total number of nodes in the graph.

For the above graph the value of X_4 wil be as-

$$X_4 = \begin{bmatrix} 9 & 4 & 9 & 10 \\ 9 & 5 & 9 & 13 \\ 4 & 1 & 4 & 4 \\ 5 & 4 & 5 & 9 \end{bmatrix}$$

From definition of path matrix we know that P[i][j]=1 if there is a path from v_i to v_j ,and this path can have length n or less than n.

Now in this matrix, if we replace all nonzero entries by 1 then we will get the path matrix or reachability matrix.

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

This graph is strongly connected since all the entries are equal to 1.

```
/* Program to find out the path matrix by powers of adjacency matrix */
#include<stdio.h>
#define MAX 20

int n;
main( )
{

        int w_adj[MAX][MAX],adj[MAX][MAX],adjp[MAX][MAX];
        int x[MAX][MAX],path[20][20],i,j,p;

        printf("Input number of vertices : ");
        scanf("%d",&n);
        printf("Enter the weighted adjacent matrix :\n");
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
                scanf("%d",&w_adj[i][j]);
        printf("The weighted adjacency matrix is :\n");

        display(w_adj);
```

```
        /* Converts weighted adjacency matrix to boolean matrix */
        to_boolean(w_adj,adj);

        printf("The boolean adjacency matrix is :\n");
        display(adj);
        while(1)
        {
                printf("Enter the path length to be searched (0 to quit) : ");
                scanf("%d",&p);
                if(p==0)
                        break;

                /* Matrix adjp is equal to adj raised to power p */
                pow_matrix(adj,p,adjp);
                printf("The path matrix for lengths equal to %d is : \n",p);
                display(adjp);
        }/*End of while */

        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                x[i][j]=0;

        /*All the powers of adj will be added to matrix x */
        for(p=1;p<=n;p++)
        {
                pow_matrix(adj,p,adjp);
                for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        x[i][j]=x[i][j]+adjp[i][j];
        }
        printf("The matrix x is :\n");
        display(x);

        to_boolean(x,path);

        printf("The path matrix is :\n");
        display(path);
}/*End of main( ) */

/*This function computes the pth power of matrix adj and stores result in adjp */
pow_matrix(int adj[MAX][MAX],int p,int adjp[MAX][MAX] )
{
        int i,j,k,tmp[MAX][MAX];

        /*Initially adjp is equal to adj */
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
           adjp[i][j]=adj[i][j];
```

```
        for(k=1;k<p;k++)
        {
                /*Multiply adjp with adj and store result in tmp */
                multiply(adjp,adj,tmp);
                for(i=0;i<n;i++)
                 for(j=0;j<n;j++)
                   adjp[i][j]=tmp[i][j]; /* New adjp is equal to tmp */
        }
}/*End of pow_matrix( )*/

/*This function multiplies mat1 and mat2 and stores the result in mat3 */
multiply(int mat1[MAX][MAX],int mat2[MAX][MAX],int mat3[MAX][MAX])
{
        int i,j,k;

        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
          {
                mat3[i][j]=0;
                for(k=0;k<n;k++)
                  mat3[i][j] = mat3[i][j]+ mat1[i][k] * mat2[k][j];
          }
}/*End of multiply( )*/

/*This fn converts matrix mat into boolean matrix and stores result in boolmat */
to_boolean( int mat[MAX][MAX], int boolmat[MAX][MAX] )
{
        int i,j;
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
          if (mat[i][j] == 0 )
                boolmat[i][j]=0;
          else
                boolmat[i][j]=1;
}/*End of to_boolean( )*/

display(int matrix[MAX][MAX])
{
        int i,j;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                        printf("%4d",matrix[i][j]);

                printf("\n");
        }
}/*End of display( )*/
```

Warshall's Algorithm-

As we have seen earlier, we can find the path matrix P of a given graph G with the use of powers of adjacency matrix. But this method is not efficient one. Warshall has given one efficient technique for finding path matrix of a graph which is called Warshall's algorithm.

Let us take a graph G of n vertices $v_1,v_2,v_3.....v_n$. First we will take Boolean matrices $P_0,P_1,.....P_n$ where $P_k[i,j]$ is defined as-

$$P_k[i][j] = \begin{cases} 1 & \text{If there is a simple path from vertices } v_i \text{ to } v_j \text{ which does not use any other node except possibly } v_1,v_2.....v_k. \text{ or this path does not use any node numbered higher than k} \\ 0 & \text{Otherwise} \end{cases}$$

Or we can say,

$P_0[i][j] = 1$ If there is a simple path from vi to vj.which does not use any node.

$P_1[i][j] = 1$ If there is a simple path from vi to vj which does not use any other nodes except possibly v_1.

$P_2[i][j] = 1$ If there is a simple path from vi to vj which does not use any other nodes except v_1,v_2.

.....

.....

$P_k[i][j] = 1$ If there is a simple path from vi to vj which does not use any other nodes except v_1,v_2------v_k..

.....

.....

$P_n[i][j] = 1$ If there is a simple path from vi to vj which does not use any other nodes except v_1,v_2------v_n..

Here P_0 represents the adjacency matrix and P_n represents the path matrix.

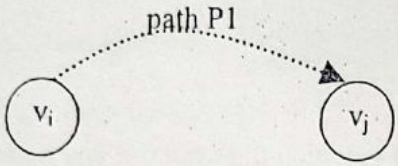
$P_0[i][j]=1$, If there is a simple path from v_i to v_j which does not use any node. The only way to go directly from v_i to v_j without using any node is to go directly from v_i to v_j. Hence $P_0[i][j]=1$ if there is any edge from v_i to v_j So P_0 will be the adjacency matrix.

$P_n[i][j]=1$, If there is a simple path from v_i to v_j which does not use any nodes except $v_1,v_2,$-----v_n. There are total n nodes means this path can use all n nodes, hence from the definition of path matrix we observe that P_n is the path matrix.

Now we'll see how we can find out the elements of matrix P_k. We know that P_0 is equal to the adjacency matrix .We have to find matrices $P_1,P_2,$----P_n. First, we have a need to know how to find $P_k[i][j]$ from $P_{k-1}[i][j]$ then we can find all these matrices. So first we will find $P_k[i][j]$ from $P_{k-1}[i][j]$.

There can be two cases for $P_{k-1}[i][j]$
$P_{k-1}[i][j]=1$ or $P_{k-1}[i][j]=0$

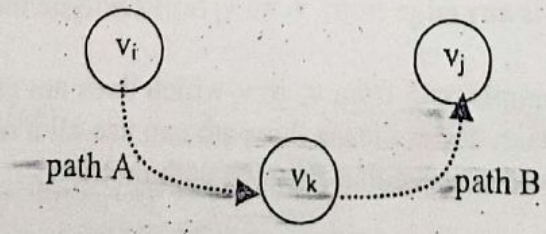
(i) If $P_{k-1}[i][j]=1$
This implies that there is a simple path P1 from v_i to v_j which does not use any nodes except possibly $v_1,v_2,\text{-------}v_{k-1}$ or this path does not use any nodes numbered higher than k-1. So it is obvious that this path does not use any nodes numbered higher than k also or we can say that this path does not use any other nodes except $v_1,v_2\text{-----}v_k$. So $P_k[i][j]$ will be equal to 1.
Hence if $P_{k-1}[i][j]=1$ then definitely $P_k[i][j]=1$.



(ii) If $P_{k-1}[i][j]=0$
This means path P1 defined above does not exist.
Now $P_k[i][j]$ can be 0 or 1.
The only case for $P_k[i][j]$ will be 1 for $P_{k-1}[i][j]=0$ is when there is a path from v_i to v_j passing through nodes $v_1,v_2,\text{-----}v_k$,but there is no path from v_i to v_j using only nodes $v_1,v_2,\text{-----}v_{k-1}$. Hence we see that if we use only nodes $v_1,v_2,\text{-----}v_{k-1}$ then there is no path from v_i to v_j but if we use node v_k also then we get a path from v_i to v_j .This means that there is a path from v_i to v_j which definitely passes through v_k and all other nodes in the path can be from nodes $v_1,v_2,\text{-----}v_{k-1}$.

Now we can break this path, into two paths.
1. Path A from v_i to v_k using nodes $v_1,v_2,\text{-----}v_{k-1}$.
2. Path B from v_k to v_j using nodes $v_1,v_2,\text{-----}v_{k-1}$.

From path A we can write that $P_{k-1}[i][k]=1$
From path B we can write that $P_{k-1}[k][j]=1$



So we can conclude that if $P_{k-1}[i][j]=0$ then $P_k[i][j]$can be equal to 1 only if $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$.

So we have two cases where $P_k[i,j] = 1$

1. There is a simple path from v_i to v_j which does not use any other nodes except possibly $v_1, v_2 \ldots\ldots v_{k-1}$, hence

$P_{k-1}[i][j] = 1$ $\quad v_i$-------->---------->v_j

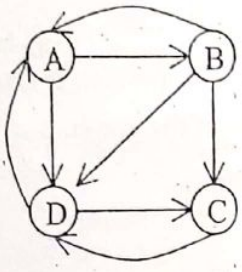
2. There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except $v_1, v_2 \ldots\ldots v_{k-1}$, hence

$P_{k-1}[i][k] = 1$ and $P_{k-1}[k][j] = 1$ $\quad v_i$--->---v_k---->----v_j

So the element of path matrix P_k can be defined as-

$$P_k[i][j] = \begin{matrix} P_{k-1}[i][j] \\ \text{or} \\ P_{k-1}[i][k] \text{ And } P_{k-1}[k][j] \end{matrix}$$

Let us take a graph and find out the values of P_0, P_1, P_2, P_3, P_4.



The first matrix P_0 is the adjacecy matrix.

$$P_0 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now we have to find P_1

Now wherever $P_0[i][j]=1$ $\quad P_1[i][j]=1$

If $P_0[i][j]=0$ then see $P_0[i][k]$ and $P_0[k][j]$, if both are 1 then $P_1[i][j]=1$

$$P_1 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Similarly, P_2 =

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	1
D	1	1	1	1

Similarly, P_3 =

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	1
D	1	1	1	1

And P_4 =

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	1	1	1	1
D	1	1	1	1

Here P_0 is the adjacency matrix and P_4 is the path matrix of the graph.

```
/* Program to find path matrix by Warshall's algorithm */
#include<stdio.h>
#define MAX 20
main( )
{
        int i,j,k,n;
        int w_adj[MAX][MAX],adj[MAX][MAX],path[MAX][MAX];

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        printf("Enter weighted adjacency matrix :\n");
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
                scanf("%d",&w_adj[i][j]);

        printf("The weighted adjacency matrix is :\n");
        display(w_adj,n);

        /* Change weighted adjacency matrix into boolean adjacency matrix*/
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
                if(w_adj[i][j]==0)
                        adj[i][j]=0;
                else
                        adj[i][j]=1;
```

```
        printf("The adjacency matrix is :\n");
        display(adj,n);

        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
                path[i][j]=adj[i][j];

        for(k=0;k<n;k++)
        {
                printf("P%d is :\n",k);
                display(path,n);
                for(i=0;i<n;i++)
                  for(j=0;j<n;j++)
                     path[i][j]=( path[i][j] || ( path[i][k] && path[k][j] ) );
        }
        printf("Path matrix P%d of the given graph is :\n",k);
        display(path,n);
}/*End of main( ) */

display(int matrix[MAX][MAX],int n)
{
        int i,j;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                        printf("%3d",matrix[i][j]);
                printf("\n");
        }
}/*End of display( )*/
```

Modified Warshall's Algorithm-

We have seen that Warshall's algorithm gives the path matrix of graph. Now by modifying this algorithm, we will find out the shortest path matrix Q. Here Q[i][j] will represent the length of shortest path from v_i to v_j.

There can be many paths from any node v_i to v_j. Our purpose is to find the length of the shortest path in between these two nodes.

We have seen that weighted matrix can be defined as-

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge from node i to node j.} \\ 0 & \text{otherwise} \end{cases}$$

Now we will take weight on each edge as the length of that edge.

As in Warshall's algorithm matrices were P_0,P_1,P_2....., here we will take matrices as Q_0,Q_1,Q_2.....

$$Q_k[i][j] = \begin{cases} \text{length of shortest path from } v_i \text{ to } v_j \text{ using nodes } v_1,v_2,v_3\text{-------}v_k. \\ \infty \quad \text{(if there is no path from } v_i \text{ to } v_j \text{ using nodes } v_1,v_2,v_3\text{-------}v_k.) \end{cases}$$

Hence we can say that
$Q_0[i][j]$=length of an edge from v_i to v_j
$Q_1[i][j]$=length of shortest path from v_i to v_j using v_1.
$Q_2[i][j]$=length of shortest path from v_i to v_j using v_1,v_2.
$Q_3[i][j]$=length of shortest path from v_i to v_j using v_1,v_2,v_3

We can find out Q_0 from the weighted adjacency matrix by replacing all zero entry by ∞. If there are n nodes in the graph then matrix Q_n will represent the shortest path matrix.So now our purpose is to find out matrices Q_1,Q_2,Q_3----------Qn. We have already found out matrix Q_0 by weighted adjacency matrix .Now if we know how to find out the matrix Q_k from matrix Q_{k-1} then we can easily find out matrices Q_1,Q_2,Q_3----------Qn also. Now we'll see how to find out matrix Q_k from matrix Q_{k-1}.

We have seen in Warshall's algorithm that $P_k[i][j]=1$ if any of these two conditions is true.
1. $P_{k-1}[i][j]=1$ or
2. $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$

This means there is a path from v_i to v_j using v_1,v_2,v_3 --------v_k. in two conditions

1. There is a path from v_i to v_j using $v_1,v_2,$------v_{k-1} (path P1)
   or
2. There is a path from v_i to v_k using v_1,v_2,v_3---------v_{k-1} and there is a path from v_k to v_j using v_1,v_2,v_3---------v_{k-1} (path P2)

Here we are dealing with path lengths so lengths of paths will be as -
Length of first path will be $Q_{k-1}[i][j]$
Length of second path will be $Q_{k-1}[i][k]+Q_{k-1}[k][j]$

Now we'll select the smaller one from these two path lengths.
So value of $Q_k[i][j]$ = Minimum($Q_{k-1}[i][j]$, $Q_{k-1}[i][k]+Q_{k-1}[k][j]$)

Case1-
$Q_{k-1}[i][j]=\infty$ and $Q_{k-1}[i][k]+Q_{k-1}[k][j]=\infty$
Path P1 and path P2 do not exist.
So $Q_{k-1}[i][j]$=Minimum(∞,∞)=∞
means there will be no path from node i to node j,using v_1,v_2,v_3----------v_k.

Case 2-

$Q_{k-1}[i][j]= \infty$ and $Q_{k-1}[i][k]+Q_{k-1}[k][j]= b$

There is no path from v_i to v_j using nodes v_1,v_2,v_3------v_{k-1} but when we include node v_k then there is a path (path P2) of length b.

So $Q_{k-1}[i][j]$=Minimum(∞,b)=b

Case 3-

$Q_{k-1}[i][j]= a$ and $Q_{k-1}[i][k]+Q_{k-1}[k][j]= \infty$

There is a path(path P1) of length a from v_i to v_j when we use nodes v_1,v_2,v_3------v_{k-1} but when we include node v_k then there is no path. This is possible when $Q_{k-1}[i][k]$ or $Q_{k-1}[k][j]$ or both are infinity.

So $Q_{k-1}[i][j]$=Minimum(a,∞)=a

Case 4-

$Q_{k-1}[i][j]= a$ and $Q_{k-1}[i][k]+Q_{k-1}[k][j]= b$

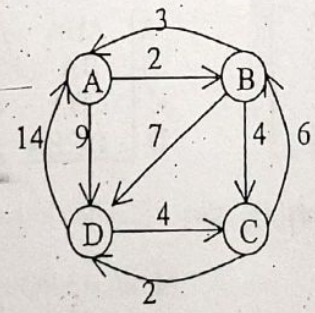
Both path P1 and path P2 exist. Length of the shorter of the two will be the value of $Q_k[i][j]$.

So $Q_{k-1}[i][j]$=Minimum(a,b)

We replaced every zero in the weighted matrix by ∞ to obtain Q_0 because if we take zero then whenever we use minimum function we will get zero as the result and we won't be able to compute the shortest path.

In program, we take ∞ to be a very large number.

Now let us take a graph and find out the shortest path matrix for it.



Weighted adjacency matrix for this graph is

$$W = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 9 \\ 3 & 0 & 4 & 7 \\ 0 & 6 & 0 & 2 \\ 14 & 0 & 4 & 0 \end{bmatrix} \end{array}$$

Case 2-

$Q_{k-1}[i][j] = \infty$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = b$

There is no path from v_i to v_j using nodes $v_1, v_2, v_3 \text{------} v_{k-1}$ but when we include node v_k then there is a path (path P2) of length b.

So $Q_{k-1}[i][j] = \text{Minimum}(\infty, b) = b$

Case 3-

$Q_{k-1}[i][j] = a$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = \infty$

There is a path(path P1) of length a from v_i to v_j when we use nodes $v_1, v_2, v_3 \text{------} v_{k-1}$ but when we include node v_k then there is no path. This is possible when $Q_{k-1}[i][k]$ or $Q_{k-1}[k][j]$ or both are infinity.

So $Q_{k-1}[i][j] = \text{Minimum}(a, \infty) = a$

Case 4-

$Q_{k-1}[i][j] = a$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = b$

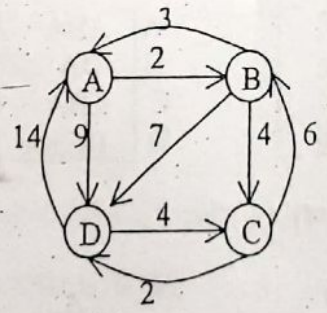
Both path P1 and path P2 exist. Length of the shorter of the two will be the value of $Q_k[i][j]$.

So $Q_{k-1}[i][j] = \text{Minimum}(a, b)$

We replaced every zero in the weighted matrix by ∞ to obtain Q_0 because if we take zero then whenever we use minimum function we will get zero as the result and we won't be able to compute the shortest path.

In program, we take ∞ to be a very large number.

Now let us take a graph and find out the shortest path matrix for it.



$K_0 =$ [

Weighted adjacency matrix for this graph is

$W =$

	A	B	C	D
A	0	2	0	9
B	3	0	4	7
C	0	6	0	2
D	14	0	4	0

$Q_0 =$

	A	B	C	D
A	∞	2	0	9
B	3	∞	4	7
C	∞	6	∞	2
D	14	∞	4	∞

	A	B	C	D
A	--	AB	--	AD
B	BA	--	BC	BD
C	--	CB	--	CD
D	DA	--	DC	--

After including node A (k=1)

$Q_1 =$

	A	B	C	D
A	∞	2	0	9
B	3	**5**	4	7
C	∞	6	∞	2
D	14	**16**	4	**23**

	A	B	C	D
A	--	AB	--	AD
B	BA	**BAB**	BC	BD
C	--	CB	--	CD
D	DA	**DAB**	DC	**DAD**

After including node B (k=2)

$Q_2 =$

	A	B	C	D
A	**5**	2	**6**	9
B	3	5	4	7
C	**9**	6	**10**	2
D	14	16	4	23

	A	B	C	D
A	**ABA**	AB	**ABC**	AD
B	BA	BAB	BC	BD
C	**CBA**	CB	**CBC**	CD
D	DA	DAB	DC	DAD

After including node C (k=3)

$Q_3 =$

	A	B	C	D
A	5	2	6	**8**
B	3	5	4	**6**
C	9	6	10	2
D	**13**	**10**	4	**6**

	A	B	C	D
A	ABA	AB	ABC	**ABCD**
B	BA	BAB	BC	**BCD**
C	CBA	CB	CBC	CD
D	**DCBA**	**DCB**	DC	**DCD**

After including node D (k=4)

$Q_4 =$

	A	B	C	D
A	5	2	6	8
B	3	5	4	6
C	9	6	**6**	2
D	13	10	4	6

	A	B	C	D
A	ABA	AB	ABC	ABCD
B	BA	BAB	BC	BCD
C	CBA	CB	**CDC**	CD
D	DCBA	DCB	DC	DCD

$Q_1(1,3)$=Minimum [$Q_0(1,3)$, $Q_0(1,1)+Q_0(1,3)$] =Minimum (∞ , ∞)= ∞

$Q_1(2,2)$=Minimum [$Q_0(2,2)$, $Q_0(2,1)+Q_0(1,2)$] =Minimum (∞ ,3+2)= 5

$Q_2(3,1)$=Minimum [$Q_1(3,1)$, $Q_1(3,2)+Q_1(2,1)$] =Minimum (∞ ,6+3)= 9

$Q_3(1,4)$=Minimum [$Q_2(1,4)$, $Q_2(1,3)+Q_2(3,4)$] =Minimum (9, 6+2)= 8

$Q_4(3,3)$=Minimum [$Q_3(3,3)$, $Q_3(3,4)+Q_3(4,3)$] =Minimum (10,2+4)= 6

```
/*Program for Modified Warshall's algorithm to find shoretst path matrix */
#include<stdio.h>
#define infinity 9999
#define MAX 20

main( )
{
        int i,j,k,n;
        int adj[MAX][MAX],path[MAX][MAX];

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        printf("Enter weighted matrix :\n");
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
                scanf("%d",&adj[i][j]);

        printf("Weighted matrix is :\n");
        display(adj,n);

        /*Replace all zero entries of adjacency matrix with infinity*/
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
          if(adj[i][j]==0)
                path[i][j]=infinity;
          else
                path[i][j]=adj[i][j];

        for(k=0;k<n;k++)
        {
                printf("Q%d is :\n",k);
                display(path,n);
                for(i=0;i<n;i++)
                  for(j=0;j<n;j++)
                    path[i][j]=minimum( path[i][j] , path[i][k]+path[k][j] );
        }
        printf("Shortest path matrix Q%d is :\n",k);
        display(path,n);
}/*End of main( ) */

minimum(int a,int b)
{
        if(a<=b)
          return a;
        else
          return b;
}/*End of minimum( )*/
```

```
display(int matrix[MAX][MAX],int n )
{
        int i,j;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                        printf("%7d",matrix[i][j]);
                printf("\n");
        }
}/*End of display( )*/
```

Traversal In Graph-

Previously we have seen that traversal is nothing but visiting each node in some systematic approach. As we traverse a binary tree in preorder, postorder and inorder manner. Graph is represented by it's nodes and edges. So traversal of each node is the traversing in graph. There are two efficient techniques for traversing the graph. First is depth first search and second is breath first search. We maintain graph in an adjacency matrix or in adjacency list for nodes of graph and edge from one node to other. In breadth first search, we use queue for keeping nodes, which will be used for next processing, and in depth first search we use stack, which keeps the node for next processing.

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 .

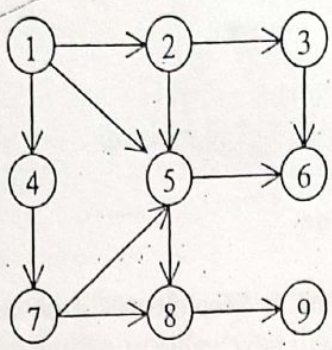
Breadth First Search-

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 breadth first traversal to it.



Take the node 1 as starting node and start the traversal of graph.
First we will traverse the node 1. Then we will traverse all nodes adjacent to node1 i.e 2,5,4. Here we can traverse these three nodes in any order. So we can see that traversal is not unique. Suppose we traverse these nodes in order 2,4,5.
So now the traversal is-

1 2 4 5

Now first we traverse all the nodes adjacent to 2,then all the nodes adjacent to 4 then all the nodes adjacent to 5. So first we will traverse 3 ,then 7 and then 6,8.
So now the traversal is -

1 2 4 5 3 7 6 8

Now we will traverse one by one all the nodes adjacent to nodes 3,7,6,8. We can see that node adjacent to node 3 is node 6 , but it has already been traversed so we will just ignore it and proceed further. Now node adjacent to node 7 is node 8 which has already been traversed so ignore it also .Node 6 has no adjacent nodes. Node 8 has node 9 adjacent to it which has not been traversed, so traverse node 9.
So now the traversal is-

1 2 4 5 3 7 6 8 9

This was the traversal when we take node 1 as the starting node. Suppose we take node 2 as the starting node. Then applying above technique, we will get the following traversal-

2 3 5 6 8 9

We can see that all the nodes are not traversed when starting node is 2. The nodes which are in the traversal are those nodes which are reachable from 2. See the different traversals when we take different starting nodes.

Start Node	Traversal
1	2 4 5 3 7 6 8 9
2	2 3 5 6 8 9
3	3 6
4	4 7 5 8 6 9
5	5 6 8 9
6	6
7	7 5 8 6 9
8	8 9
9	9

Breadth First Search 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.

Step 1-
Insert starting node 1 into queue.
Traversed nodes = 1
visited[1]=true
front =0 rear=0 queue=1
Traversal = 1

Step 2-
Delete front element node 1 from queue and insert all its unvisited neighbours 2, 4, 5 into the queue .
Traversed nodes - 2, 4, 5
visited[2]=true, visited[4]=true, visited[5]=true
front =1 rear=3 queue = 2, 4, 5
Traversal = 1, 2, 4, 5

Step 3-
Delete front element node 2 from queue, traverse it's unvisited neighbour node 3 ,and insert it into the queue.
Traversed nodes - 3
visited[3] = true
front =2 rear = 4 queue = 4,5,3
Traversal = 1, 2, 4, 5, 3

Step 4-
Delete front element node 4 from queue, traverse it's unvisited neighbour node 7 and insert it into the queue.
Traversed nodes - 7
visited[7] = true
front = 3 rear = 5 queue = 5, 3, 7
Traversal = 1, 2, 4, 5, 3, 7

Step 5-
Delete front element node 5 from the queue, traverse it's unvisited neighbours nodes 6, 8 and insert them into the queue.
Traversed nodes – 6, 8
visited[6] = true, visited[8] = true
front = 4 rear = 7 queue = 3, 7, 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 6-
Delete front element node 3 from queue. It has no unvisited neighbours.
front = 5 rear = 7 queue = 7, 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 7-
Delete front elemnt node 7 from queue. It has no unvisited neighbours .
front = 6 rear = 7 queue = 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 8-
Delete front element node 6 from queue. It has no unvisited neighbours.
front = 7 rear = 7 queue = 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 9-
Delete front element node 8 from queue, traverse it's unvisited neighbour node 9 and insert it into the queue.
Traversed nodes - 9
visited[9] = true
front = 8 rear = 8 queue = 9
Traversal = 1, 2, 4, 5, 3, 7, 6, 8, 9

Step 10-
Delete front element node 9 from queue. It has no unvisited neighbours
front = 8 rear = 7 queue = EMPTY
Traversal=1,2,4,5,3,7,6,8,9

Since front > rear, hence now queue is empty so we will stop our process
As we have seen this process can traverse only those nodes which are reachable from the starting node. If we want to traverse all the nodes then we will take the unvisited node as starting node and again start this process from starting node. This process will continue until all the nodes are traversed.

The function for Breadth first search is as-

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

Depth First Search-

In Depth first search technique also we take one node as a starting node. Then go to the path which is from starting node and visit all the nodes which are in that path. When we reach at the last node then we traverse another path starting from that node. If there is no path in the graph from the last node then it returns to the previous node in the path and

traverse another path and so on. This technique uses stack.

Let us take the same graph and starting node as node 1.
First, we will traverse node 1. Then we will traverse any node adjacent to node 1. Here we traverse node 2, then traverse node 3, which is adjacent to node 2, then traverse adjacent node 6. Now there is no node adjacent to node 6, means we have reached the end of the path or a dead end from where we can't go forward. So we will move backward. Till now the traversal is -
1 2 3 6
Now we reach node 3, see if there is any node adjacent to it, and not traversed yet. There is no such node so we will reach node 2 and we see that node 5 is adjacent to it and not traversed yet . So we will traverse node 5 and move along the path which starts at node 5. So we will traverse 8 and then 9 .Now there is no node adjacent to node 9 so we have reached the end of the path and now we will move backward. Till now the traversal is-
1 2 3 6 5 8 9
Now we reach node 8 and we see that there is no node adjacent to it and not traversed yet, so we reach node 5, here also we observe the same thing, so we reach node 6 here also there is no untraversed adjacent node, then we reach 3 and then 2. On reaching node 1 we see that node 4 is adjacent to it and has not been traversed. So we traverse it and move along a path which starts at node 4. So we traverse node 7. Now there is no untraversed node adjacent to node 7 so we have reached a dead end. We can't move forward but now we can't move backward also so we will stop our process. The traversal is as-
1 2 3 6 5 8 9 4 7

See the different traversals when we take different starting nodes.

Start Node	Traversal
1	1 2 3 6 5 8 9 4 7
2	2 3 6 5 8 9
3	3 6
4	4 7 5 6 8 9
5	5 6 8 9
6	6
7	7 5 6 8 9
8	8 9
9	9

Depth First Search through stack-

Take an array stack 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 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.

Suppose the starting node for traversal is 1.

Step1-
Push node 1 into stack
top = 0 stack = 1

Step 2-
Pop node 1 from stack ,traverse it.
Traversed node - 1
visited[1] = true
Now push all the unvisited adjacent nodes 5,4,2 of the popped element on stack .
top = 2 stack = 5, 4, 2
Traversal=1

Step 3-
Pop the element node 2 from the stack, traverse it and push all its unvisited adjacent nodes 5,3 on the stack
Traversed node - 2
visited[2] = true
top = 3 stack = 5, 4, 5, 3
Traversal = 1, 2

Step 4-
Pop the element node 3 from the stack, traverse it and push its unvisited adjacent node 6 on the stack
Traversed node - 3
visited[3] = true
top = 3 stack = 5, 4, 5, 6
Traversal = 1, 2, 3

Step 5-
Pop the element node 6 from stack, traverse it. Node 6 has no adjacent nodes,so nothing is pushed
Traversed node – 6
visited [6] = true
top = 2 stack = 5, 4, 5
Traversal = 1, 2, 3, 6

Step 6-
Pop the element node 5 from stack, traverse it and push its unvisited adjacent node 8 on the stack,node 6 is its adjacent node but it has been visited so it is not pushed.
Traversed node - 5
visited[5] = true
top = 2 stack = 5, 4, 8
Traversal = 1, 2, 3, 6, 5

Step 7-
Pop the element node 8 from stack, traverse it and push its unvisited adjacent node 9 on the stack
Traversed node - 8
visited[8] = true
top = 2 stack = 5, 4, 9
Traversal = 1, 2, 3, 6, 5, 8

Step 8-
Pop the element 9 from the stack and traverse it ,9 has no adjacent nodes
visited[9]=true
top=1 stack=5, 4
Traversal=1,2,3,6,5,8,9

Step 9 -
Pop the element node 4 from stack, traverse it and push its unvisited adjacent node 7 on the stack
Traversed node - 4
visited[4] = true
top = 1 stack = 5, 7
Traversal = 1, 2, 3, 6, 5, 8, 9, 4

Step 10-
Pop the element node 7 from stack, traverse it ,it has no unvisited adjacent nodes.
Traversed node - 7
visited[7]=true
top = 0 stack = 5
Traversal = 1, 2, 3, 6, 5, 8, 9, 4, 7

Step 11-
Pop the element node 5 from stack, since visited[5] = true, so just ignore it.
top = -1 Stack = EMPTY

Since the stack is empty so we will stop our process.

The function for Depth First Search is as-

```
dfs(int v)
{
        int i,stack[MAX],top=-1,pop_v,j,t;
        int  ch;

        top++;
        stack[top]=v;s
        while (top>=0)
        {
                pop_v=stack[top];
                top--;                  /*pop from stack*/
                if( visited[pop_v]==false)
                {
                        printf("%d ",pop_v);
                        visited[pop_v]=true;
                }
                else
                        continue;

                for(i=n;i>=1;i--)
                {
                        if( adj[pop_v][i]==1 && visited[i]==false)
                        {
                                top++;  /* push all unvisited neighbours of pop_v */
                                stack[top]=i;
                        }/*End of if*/
                }/*End of for*/
        }/*End of while*/
}/*End of dfs( )*/

/* Program for traversing a graph through BFS and DFS */
#include<stdio.h>
#define MAX 20

typedef enum boolean{false,true} bool;
int adj[MAX][MAX];
bool visited[MAX];
int n;   /* Denotes number of nodes in the graph */
```

```
main( )
{
        int i,v,choice;

        create_graph( );
        while(1)
        {
                printf("\n");
                printf("1. Adjacency matrix\n");
                printf("2. Depth First Search using stack\n");
                printf("3. Depth First Search through recursion\n");
                printf("4. Breadth First Search\n");
                printf("5. Adjacent vertices\n");
                printf("6. Components\n");
                printf("7. Exit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1:
                        printf("Adjacency Matrix\n");
                        display( );
                        break;
                case 2:
                        printf("Enter starting node for Depth First Search : ");
                        scanf("%d",&v);
                        for(i=1;i<=n;i++)
                                visited[i]=false;
                        dfs(v);
                        break;
                case 3:
                        printf("Enter starting node for Depth First Search : ");
                        scanf("%d",&v);
                        for(i=1;i<=n;i++)
                                visited[i]=false;
                        dfs_rec(v);
                        break;
                case 4:
                        printf("Enter starting node for Breadth First Search : ");
                        scanf("%d", &v);
                        for(i=1;i<=n;i++)
                                visited[i]=false;
                        bfs(v);
                        break;
                case 5:
                        printf("Enter node to find adjacent vertices : ");
                        scanf("%d", &v);
                        printf("Adjacent Vertices are : ");
```

```
                        adj_nodes(v);
                        break;
                case 6:
                        components( );
                        break;
                case 7:
                        exit(1);
                default:
                        printf("Wrong choice\n");
                        break;
                }/*End of switch*/
        }/*End of while*/
}/*End of main( )*/

create_graph( )
{
        int i,max_edges,origin,destin;

        printf("Enter number of nodes : ");
        scanf("%d",&n);
        max_edges=n*(n-1);

        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d( 0 0 to quit ) : ",i);
                scanf("%d %d",&origin,&destin);

                if((origin==0) && (destin==0))
                        break;

                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                {
                        adj[origin][destin]=1;
                }
        }/*End of for*/
}/*End of create_graph( )*/

display( )
{
        int i,j;
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        printf("%4d",adj[i][j]);
```

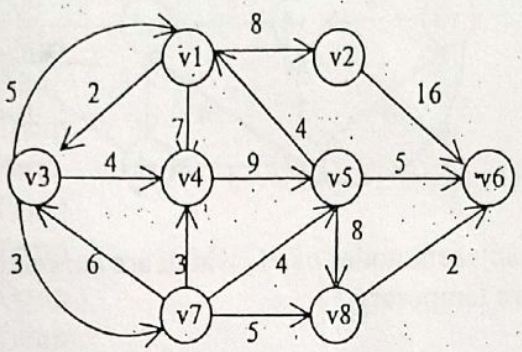
```
                printf("\n");
        }
}/*End of display( )*/

dfs_rec(int v)
{
        int i;
        visited[v]=true;
        printf("%d ",v);
        for(i=1;i<=n;i++)
                if(adj[v][i]==1 && visited[i]==false)
                        dfs_rec(i);
}/*End of dfs_rec( )*/

dfs(int v)
{
        int i,stack[MAX],top=-1,pop_v,j,t;
        int  ch;

        top++;
        stack[top]=v;
        while (top>=0)
        {
                pop_v=stack[top];
                top--;                  /*pop from stack*/
                if( visited[pop_v]==false)
                {
                        printf("%d ",pop_v);
                        visited[pop_v]=true;
                }
                else
                        continue;

                for(i=n;i>=1;i--)
                {
                        if( adj[pop_v][i]==1 && visited[i]==false)
                        {
                                top++;   /* push all unvisited neighbours of pop_v */
                                stack[top]=i;
                        }/*End of if*/
                }/*End of for*/
        }/*End of while*/
}/*End of dfs( )*/

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

adj_nodes(int v)
{
        int i;
        for(i=1;i<=n;i++)
        if(adj[v][i]==1)
                printf("%d ",i);
        printf("\n");
}/*End of adj_nodes( )*/

components( )
{
        int i;
        for(i=1;i<=n;i++)
                visited[i]=false;
        for(i=1;i<=n;i++)
        {
                if(visited[i]==false)
                        dfs_rec(i);
        }
        printf("\n");
}/*End of components( )*/
```

Shortest Path Algorithm (Dijkstra)-

There are several cases in graph where we have a need to know the shortest path from one node to another node. General electric supply system and water distribution system also follow this approach. The best example we can take is of a railway track system. Suppose one person wants to go from one station to another then he needs to know the shortest path from one station to another. Here station represents the node and tracks represent edges. In computers, it is very useful in network for routing concepts.

There can be several paths for going from one node to another node, but the shortest path is that path in which the sum of weights of the included edges is the minimum.
There are several algorithms to find out the shortest path. Here we will describe the Dijkstra's algorithm. Let us take a graph and find out the shortest path from the source node to destination node.



We label each node with dist, predecessor and status. Dist of node represents the shortest distance of that node from the source node, and predecessor of a node represents the node which precedes the given node in the shortest path from source. Status of a node can be permanent or temporary. In the figure, shaded circles represent permanent nodes. Making a node permanent means that it has been included in the shortest path. Temporary nodes can be relabeled if required but once a node is made permanent, it can't be relabeled.

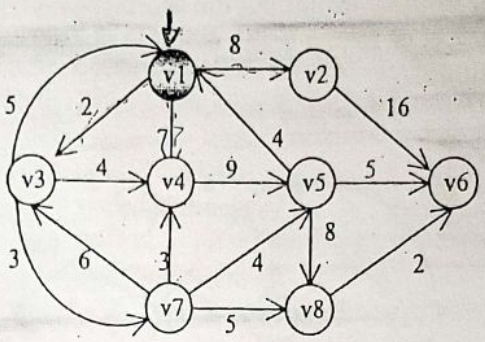
The procedure is as –

1. Initially make source node permanent and make it the current working node. All other nodes are made temporary
2. Examine all the temporary neighbours of the current working node and after checking the condition for minimum weight , relabel the required nodes.
3. From all the temporary nodes, find out the node which has minimum value of dist , make this node permanent and now this is our current working node.
4. Repeat steps 2 and 3 until destination node is made permanent.

Suppose the source node is v1

v1 is the current working node

Node	dist	pred	status
v1	0	0	Perm
v2	∞	0	Temp
v3	∞	0	Temp
v4	∞	0	Temp
v5	∞	0	Temp
v6	∞	0	Temp
v7	∞	0	Temp
v8	∞	0	Temp



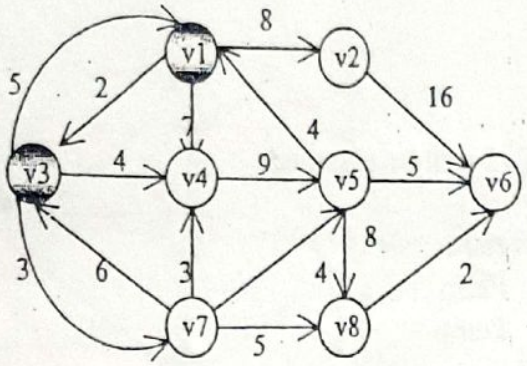
Now we will check adjacent nodes of v1, which are temporary also. Here v2,v3,v4 are adjacent to v1 and are temporary.

v2.dist > v1.dist + distance(v1,v2) ∞ > 0 + 8 relabel v2
v3.dist > v1.dist + distance(v1,v3) ∞ > 0 + 2 relabel v3
v4.dist > v1.dist + distance(v1,v4) ∞ > 0 + 7 relabel v4

Now we will change the pred and dist of v2, v3 and v4.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Temp
v3	2	v1	Temp
v4	7	v1	Temp
v5	∞	0	Temp
v6	∞	0	Temp
v7	∞	0	Temp
v8	∞	0	Temp

Now from all the temporary nodes find out the node that has smallest distance from source i.e. smallest value of dist. Here v3 has the smallest value of dist in all temporary nodes so make it permanent and now v3 is our current working node.



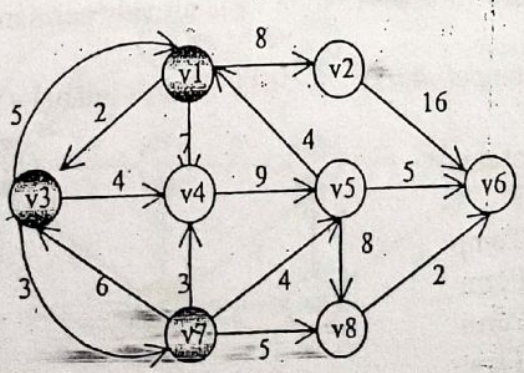
Adjacent nodes of v3 are v4 and v7 and both are temporary.

v4.dist > v3.dist + distance(v3,v4) 7 > 2+4 relabel v4
v7.dist > v3.dist + distance(v3,v7) ∞ > 2+3 relabel v7

Now we will change the pred and dist of v4 and v7.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Temp
v3	2	v1	Perm
v4	6	v3	Temp
v5	∞	0	Temp
v6	∞	0	Temp
v7	5	v3	Temp
v8	∞	0	Temp

Now from all the temporary nodes ,v7 has smallest value of dist so make it permanent and now v7 is our current working node



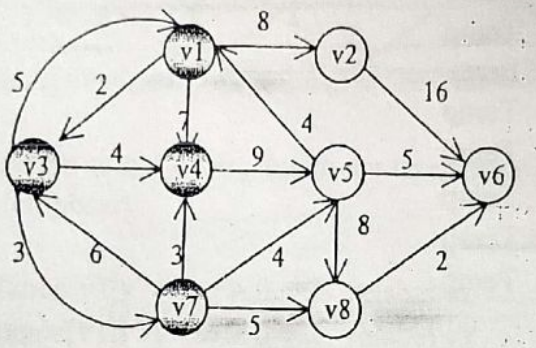
Adjacent nodes of v7 are v3,v4,v5 Since v3 is permanent we will not relabel it ,

v4.dist < v7.dist + distance(v7,v4) 6 < 5+3 Don't relabel v4
v5.dist > v7.dist + distance(v7,v5) ∞ > 5+4 relabel v5

Now pred and dist of v5 will be changed.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Temp
v3	2	v1	Perm
v4	6	v3	Temp
v5	9	v7	Temp
v6	∞	0	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v4 has smallest value of dist so make it permanent. Now v4 is the current working node.

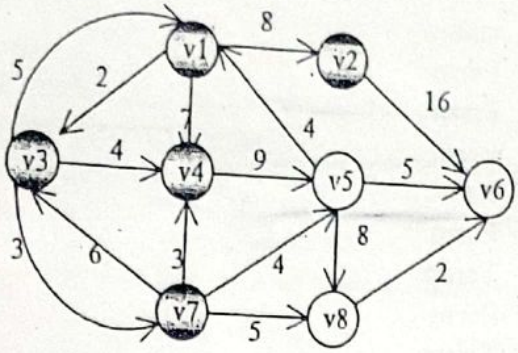


Adjacent nodes of v4 are v5 and v1.Node v1 is already permanent. So we will check for v5 only.

v5.dist < v4.dist + distance(v4,v5) 9 < 6+9 Don't relabel v5

Node	dist	pred	status
v1	0	0	P
v2	8	v1	Temp
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Temp
v6	∞	0	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v2 has smallest value of dist so make it permanent. Now v2 is our current working node.



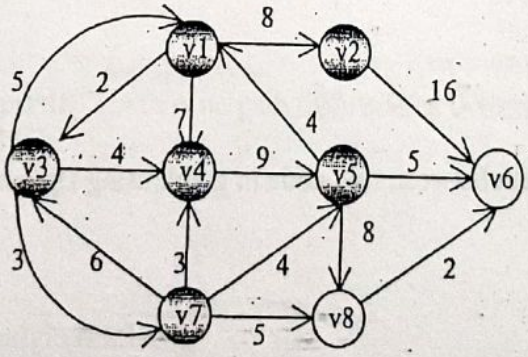
Node adjacent to v2 is v6.

v6.dist < v2.dist + distance(v2,v6) $\infty > 8+16$ relabel v6

Now pred and dist of v6 will be changed,

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Perm
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Temp
v6	24	v2	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v5 has smallest value of dist, so make it permanent .Now v5 is our current working node.



Adjacent nodes of v5 are v6 and v8.

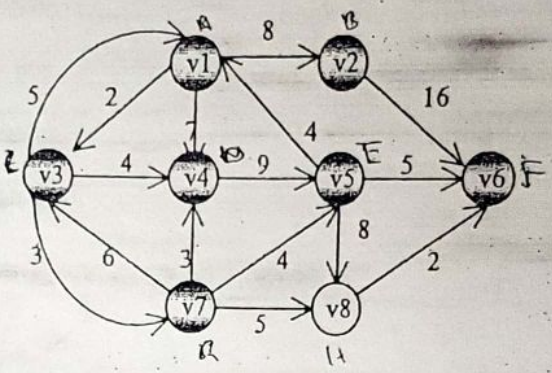
v6.dist < v5.dist + distance(v5,v6) $24 > 9+5$ relabel v6

v8.dist < v5.dist + distance(v5,v8) $\infty > 9+8$ relabel v8

Now pred and dist of v6 and v8 will be changed.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Perm
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Perm
v6	14	v5	Temp
v7	5	v3	Perm
v8	17	v5	Temp

Now from all temporary nodes v6 has smallest value of dist so make it permanent. Since v6 is our destination node and it has been made permanent, so we will stop our process.



We can find out the shortest path from the last table. Start from the destination node and keep on seeing the predecessors until we get source node as a predecessor.

pred of v6 is v5
pred of v5 is v7
pred of v7 is v3
pred of v3 is v1

So the path is v1 -- v3 -- v7 -- v5 -- v6

```
/* Program of shortest path between two node in graph using Djikstra algorithm */
#include<stdio.h>
#define MAX 10
#define TEMP 0
#define PERM 1
#define infinity 9999

struct node
{
        int predecessor;
        int dist; /*minimum distance of node from source*/
        int status;
};
```

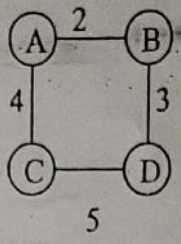
```
int adj[MAX][MAX];
int n;
void main( )
{
        int i,j;
        int source,dest;
        int path[MAX];
        int shortdist,count;

        create_graph( );
        printf("The adjacency matrix is :\n");
        display( );

        while(1)
        {
                printf("Enter source node(0 to quit) : ");
                scanf("%d",&source);
                printf("Enter destination node(0 to quit) : ");
                scanf("%d",&dest);

                if(source==0 || dest==0)
                        exit(1);

                count = findpath(source,dest,path,&shortdist);
                if(shortdist!=0)
                {
                        printf("Shortest distance is : %d\n", shortdist);
                        printf("Shortest Path is : ");
                        for(i=count;i>1;i--)
                                printf("%d->",path[i]);
                        printf("%d",path[i]);
                        printf("\n");
                }
                else
                        printf("There is no path from source to destination node\n");
        }/*End of while*/
}/*End of main( )*/

create_graph( )
{
        int i,max_edges,origin,destin,wt;

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges=n*(n-1);
        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d(0 0 to quit) : ",i);
                scanf("%d %d",&origin,&destin);
```

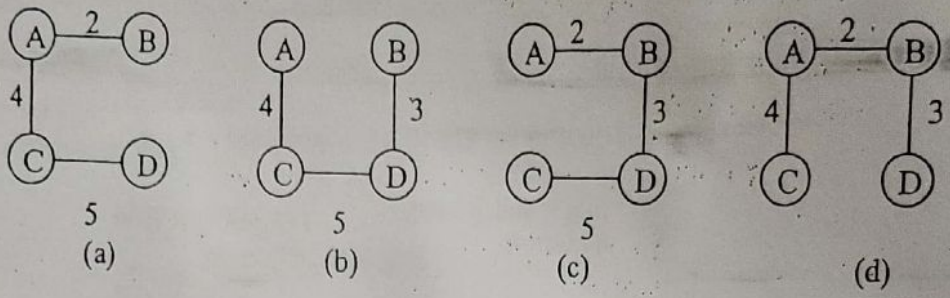
```
                if((origin==0) && (destin==0))
                        break;
                printf("Enter weight for this edge : ");
                scanf("%d",&wt);
                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin]=wt;
        }/*End of for*/
}/*End of create_graph( )*/

display( )
{
        int i,j;
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        printf("%3d",adj[i][j]);
                printf("\n");
        }
}/*End of display( )*/

int findpath(int s,int d,int path[MAX],int *sdist)
{
        struct node state[MAX];
        int i,min,count=0,current,newdist,u,v;
        *sdist=0;
        /* Make all nodes temporary */
        for(i=1;i<=n;i++)
        {
                state[i].predecessor=0;
                state[i].dist = infinity;
                state[i].status = TEMP;
        }

        /*Source node should be permanent*/
        state[s].predecessor=0;
        state[s].dist = 0;
        state[s].status = PERM;

        /*Starting from source node until destination is found*/
        current=s;
        while(current!=d)
        {
                for(i=1;i<=n;i++)
                {
```

```
                /*Checks for adjacent temporary nodes */
                if ( adj[current][i] > 0 && state[i].status == TEMP )
                {
                        newdist=state[current].dist + adj[current][i];
                        /*Checks for Relabeling*/
                        if( newdist < state[i].dist )
                        {
                                state[i].predecessor = current;
                                state[i].dist = newdist;
                        }
                }
        }/*End of for*/

        /*Search for temporary node with minimum distand make it current node*/
        min=infinity;
        current=0;
        for(i=1;i<=n;i++)
        {
                if(state[i].status == TEMP && state[i].dist < min)
                {
                        min = state[i].dist;
                        current=i;
                }
        }/*End of for*/

        if(current==0) /*If Source or Sink node is isolated*/
                return 0;
        state[current].status=PERM;
}/*End of while*/

/* Getting full path in array from destination to source     */
while( current!=0 )
{
        count++;
        path[count]=current;
        current=state[current].predecessor;
}

/*Getting distance from source to destination*/
for(i=count;i>1;i--)
{
        u=path[i];
        v=path[i-1];
        *sdist+= adj[u][v];
}
return (count);
}/*End of findpath( )*/
```

Spanning Tree-

A spanning tree of a connected graph G contains all the nodes and has the edges, which connects all the nodes. So number of edges will be 1 less than the number of nodes. Let us take a connected graph G-





Here all these trees are spanning tree. The number of edges is 3, which is 1 less than the number of nodes.

If a graph is not connected , i.e. a graph with n nodes has edges less than n-1 then no spanning tree is possible.

DFS and BFS spanning trees are the spanning trees which are obtained by DFS and BFS traversals.

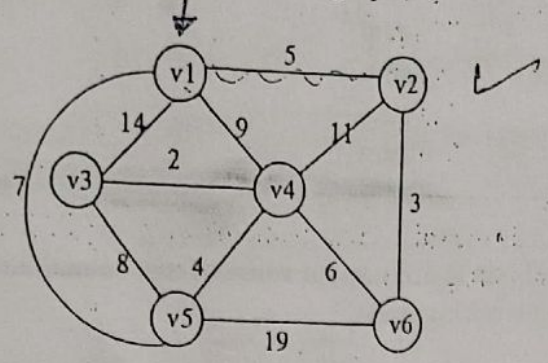
Minimum Spanning Tree-

Minimum spanning tree is the spanning tree in which the sum of weights on edges is minimum. In the above example (d) is the minimum spanning tree with weight 9. There are many ways for creating minimum spanning tree but the most famous methods are Prim's and Kruskal algorithm.

Prim's Algorithm-

In this algorithm, 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 v_1 then we have a 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. So here we will be in need to know the nodes in spanning tree and weights on the edge connecting to other nodes because if two nodes v_1 & v_2 are in spanning tree and both have edge connecting to v3 then edge which has minimum weight will be added in spanning tree. So we can say it creates growing tree and each step adds one node in spanning tree.

The method of making minimum spanning tree from Prim's algorithm is like Dijkstra's algorithm with some modifications. Here also we will label each node with dist, predecessor and status of each node. Dist of node will represent shortest distance of that node from its predecessor while in Dijkstra's algorithm dist represented distance of node from the source. Status of a node can be permanent or temporary. Whenever a node is made permanent, we include it in the spanning tree and make it the current working node. Once a node is made permanent, it is not relabeled. Only temporary nodes will be relabeled if required. Let us take a graph G-

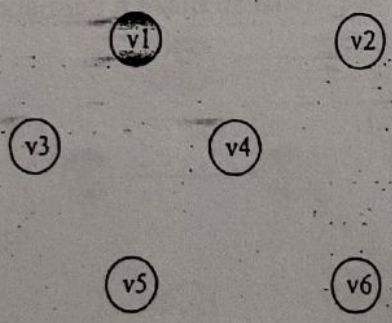


The procedure is same as in Dijkstra's algorithm. In Dijkstra's algorithm we stopped our process when destination node was made permanent, while here we will stop our process when all the nodes will be made permanent.

Initially make all the nodes temporary.

Nodes	Dist	Pred	Status
v1	∞	0	Temp
v2	∞	0	Temp
v3	∞	0	Temp
v4	∞	0	Temp
v5	∞	0	Temp
v6	∞	0	Temp

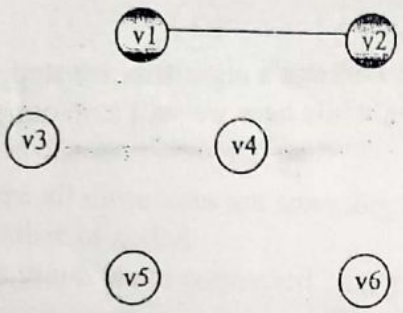
Make first node v1 permanent.



v2.dist >Distance(v1,v2)	∞ > 5	Relabel v2
v3.dist >Distance(v1,v3)	∞ > 14	Relabel v3
v4.dist >Distance(v1,v4)	∞ > 9	Relabel v4
v5.dist >Distance(v1,v5)	∞ > 7	Relabel v5

Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Temp
v3	14	v1	Temp
v4	9	v1	Temp
v5	7	v1	Temp
v6	∞	0	Temp

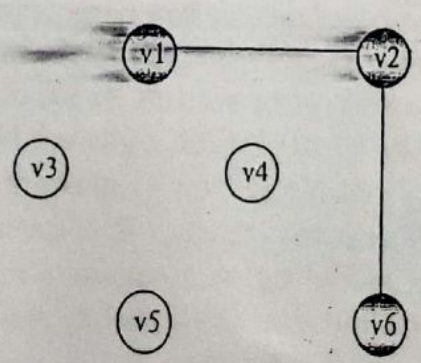
Now from all the temporary nodes v2 has the minimum value of dist so make it permanent and now v2 is our current working node.



v6.dist >Distance(v2,v6)	∞ > 3		Relabel v6

Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Perm
v3	14	v1	Temp
v4	9	v1	Temp
v5	7	v1	Temp
v6	3	v2	Temp

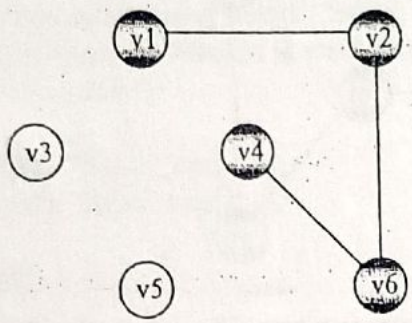
From all temporary nodes v6 has the minimum value of dist so make it permanent and now v6 is the current working node.



v4.dist >Distance(v6,v4) 9 > 6 Relabel v4

Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Perm
v3	14	v1	Temp
v4	6	v6	Temp
v5	7	v1	Temp
v6	3	v2	Perm

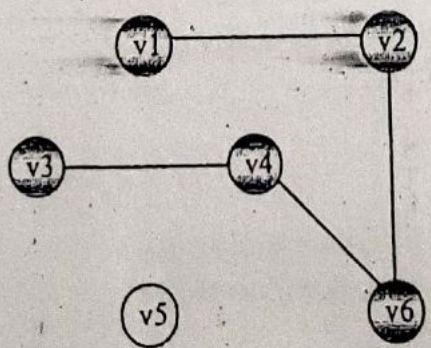
From all temporary nodes v4 has the minimum value of dist so make it permanent and now v4 is the current working node.



v3.dist >Distance(v4,v3) 14 > 2 Relabel v3
v5.dist >Distance(v4,v5) 7 > 4 Relabel v5

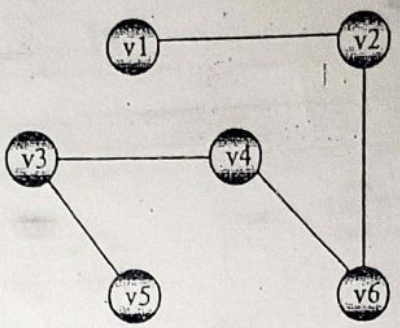
Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Perm
v3	2	v4	Temp
v4	6	v6	Perm
v5	4	v4	Temp
v6	3	v2	Perm

From all temporary nodes v3 has the minimum value of dist so make it permanent and now v3 is the current working node.



Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Perm
v3	2	v4	Perm
v4	6	v6	Perm
v5	4	v4	Temp
v6	3	v2	Perm

Make v5 permanent.



Now we have a complete minimum spanning tree. The edges that belong to minimum spanning tree are
(v1,v2), (v3,v4), (v3,v5), (v2,v6), (v4,v6)
Weight of minimum spanning tree will be-
5 + 2 + 8 + 6 + 3 = 24

```
/* Program for creating minimum spanning tree from Prim's algorithm */
#include<stdio.h>
#define MAX 10
#define TEMP 0
#define PERM 1
#define FALSE 0
#define TRUE 1
#define infinity 9999

struct node
{
        int predecessor;
        int dist; /*Distance from predecessor */
        int status;
};

struct edge
{
        int u;
        int v;
};
```

```
int adj[MAX][MAX];
int n;

main( )
{
        int i,j;
        int path[MAX];
        int wt_tree,count;
        struct edge tree[MAX];

        create_graph( );
        printf("Adjacency matrix is :\n");
        display( );
        count = maketree(tree,&wt_tree);
        printf("Weight of spanning tree is : %d\n", wt_tree);
        printf("Edges to be included in spanning tree are : \n");
        for(i=1;i<=count;i++)
        {
                printf("%d->",tree[i].u);
                printf("%d\n",tree[i].v);
        }
}/*End of main( )*/

create_graph( )
{
        int i,max_edges,origin,destin,wt;

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges=n*(n-1)/2;
        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d(0 0 to quit) : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==0) && (destin==0))
                        break;
                printf("Enter weight for this edge : ");
                scanf("%d",&wt);
                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                {
                        adj[origin][destin]=wt;
                        adj[destin][origin]=wt;
                }
        }/*End of for*/
```

```
        if(i<n-1)
        {
                printf("Spanning tree is not possible\n");
                exit(1);
        }
}/*End of create_graph( )*/

display( )
{
        int i,j;
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        printf("%3d",adj[i][j]);
                printf("\n");
        }
}/*End of display( )*/

int maketree(struct edge tree[MAX],int *weight)
{
        struct node state[MAX];
        int i,k,min,count,current,newdist;
        int m;
        int u1,v1;
        *weight=0;
        /*Make all nodes temporary*/
        for(i=1;i<=n;i++)
        {
                state[i].predecessor=0;
                state[i].dist = infinity;


                state[i].status = TEMP;
        }
        /*Make first node permanent*/
        state[1].predecessor=0;
        state[1].dist = 0;
        state[1].status = PERM;

        /*Start from first node*/
        current=1;
        count=0; /*count represents number of nodes in tree */
        while( all_perm(state) != TRUE ) /*Loop till all the nodes become PERM*/
        {
                for(i=1;i<=n;i++)
                {
                        if ( adj[current][i] > 0 && state[i].status == TEMP )
                        {
```

```
                        if( adj[current][i] < state[i].dist )
                        {
                                state[i].predecessor = current;
                                state[i].dist = adj[current][i];
                        }
                }
        }/*End of for*/

        /*Search for temporary node with minimum distance
        and  make it current node*/
        min=infinity;
        for(i=1;i<=n;i++)
        {
                if(state[i].status == TEMP && state[i].dist < min)
                {
                        min = state[i].dist;
                        current=i;
                }
        }/*End of for*/

        state[current].status=PERM;

        /*Insert this edge(u1,v1) into the tree */
        u1=state[current].predecessor;
        v1=current;
        count++;
        tree[count].u=u1;
        tree[count].v=v1;
        /*Add wt on this edge to weight of tree  */
        *weight=*weight+adj[u1][v1];
    }/*End of while*/
    return (count);
}/*End of maketree( )*/

/*This function returns TRUE if all nodes are permanent*/
int all_perm(struct node state[MAX] )
{
    int i;
    for(i=1;i<=n;i++)
        if( state[i].status == TEMP )
            return FALSE;
    return TRUE;
}/*End of all_perm( )*/
```

Kruskal's Algorithm

In this method initially we take n distinct trees for all n nodes of the graph. Then we take edges in ascending order, whenever we insert an edge two trees will be joined.

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'll 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 edge would result in a cycle. Since we are making a tree, so there should not be any cycles. We will insert an edge in the spanning tree only if it's nodes are in different trees.

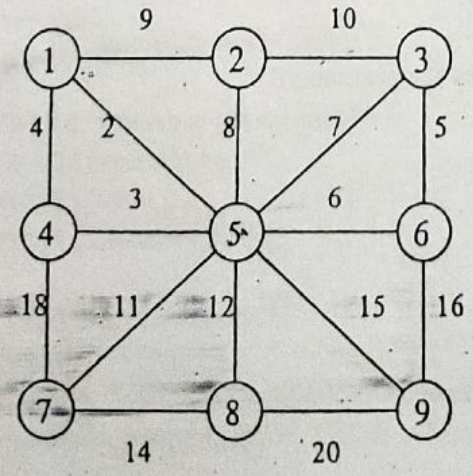
Now we'll see how to decide whether two nodes are in the same tree or not. We will keep record of the father of every node. Since this is a tree, every node will have only one distinct father. Initially father of all nodes will be zero. We will recognize each tree by a root node and a node will be a root if its father is zero. For finding out, to which tree a node belongs, we will find out the root of that tree. So we will traverse all the ancestors of the node till we reach a node whose father is zero. This will be the root of the tree to which the node belongs.

Now we know the root of both nodes of edge, if roots are same means both nodes are in the same tree and are already connected. If the roots are different, then we'll insert this edge into the spanning tree and we'll join the two trees, which are having these two nodes. For joining the two trees, we will make root of one tree as the father of root of another tree

After joining two trees all the nodes of both trees will be connected and have the same root.

To obtain the edges in ascending order we can insert them in a priority queue.

Let us take a graph and make a minimum spanning tree by Kruskal's algorithm.



Initially father of every node is zero , and hence every node is a tree ,and the root of that tree is the node itself.

node	1	2	3	4	5	6	7	8	9
father	0	0	0	0	0	0	0	0	0

Step 1 -

Edge selected is 1-5 wt=2
n1=1 root_n1=1 n2=5 root_n2=5
Roots are different so edge is inserted in the spanning tree.
Join the two trees father[5]=1
node 1 2 3 4 5 6 7 8 9
father 0 0 0 0 1 0 0 0 0

Step 2 -

Edge selected is 4-5 wt=3
n1=4 root_n1=4 n2=5 root_n2=1
Edge Inserted , father[1]=4
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 0 0 0 0

Step 3 -

Edge selected is 1-4 wt=4
n1=1 root_n1=4 n2=4 root_n2=4
Roots are same so edge is not inserted in the spanning tree.
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 0 0 0 0

Step 4 -

Edge selected is 3-6 wt=5
n1=3 root_n1=3 n2=6 root_n2=6
Edge inserted father[6]=3
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 3 0 0 0

Step 5 -

Edge selected is 5-6 wt=6
n1=5 root_n1=4 n2=6 root_n2=3
Edge inserted father[3]=4
node 1 2 3 4 5 6 7 8 9
father 4 0 4 0 1 3 0 0 0

Step 6 -

Edge selected is 3-5 wt=7
n1=3 root_n1=4 n2=5 root_n2=4
Edge not inserted
node 1 2 3 4 5 6 7 8 9
father 4 0 4 0 1 3 0 0 0

Step 7 -

Edge selected is 2-5 wt=8
n1=2 root_n1=2 n2=5 root_n2=4
Edge inserted, father[4]=2
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 0 0 0

Step 8 -

Edge selected is 1-2 wt=9
n1=1 root_n1=2 n2=2 root_n2=2
Not inserted
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 0 0 0

Step 9 -

Edge selected is 2-3 wt=10
n1=2 root_n1=2 n2=3 root_n2=2
Edge not inserted
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 0 0 0

Step 10 -

Edge selected is 5-7 wt=11
n1=5 root_n1=2 n2=7 root_n2=7
Edge inserted, father[7]=2
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 2 0 0

Step 11 -

Edge selected is 5-8 wt=12
n1=5 root_n1=2 n2=8 root_n2=8
Edge inserted, father[8]=2
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 2 2 0

Step 12 -

Edge selected is 7-8 wt=14
n1=7 root_n1=2 n2=8 root_n2=2
Edge not inserted
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 2 2 0

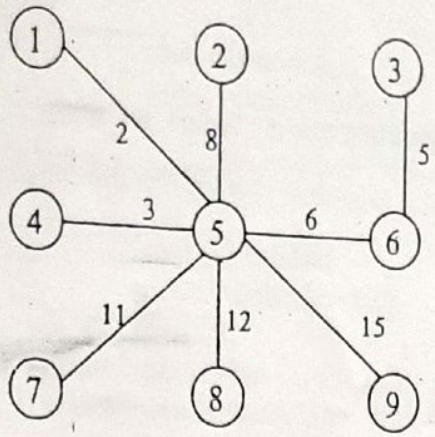
Step 13 -

Edge selected is 5-9 wt=15
n1=5 root_n1=2 n2=9 root_n2=9
Edge inserted, father[9]=2
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 2 2 2

The minimum spanning tree should contain n-1 edges where n is the number of nodes in the graph. This graph contains 9 nodes so after including 8 edges in the spanning tree, we will not examine other edges of the graph and stop our process.

We can also see that now there is only one tree which is rooted at node 2. We can take any node of the tree and it's root will come out to be node 2.

The resulting minimum spanning tree will be-



Edges included in this spanning tree are (1,5), (4,5), (3,6), (5,6), (2,5), (5,7), (5,8), (5,9)
Weight of this spanning tree is 2 + 3 + 5 + 6 + 8 + 11 + 12 +15 = 62

```
/* Program for creating a minimum spanning tree from Kruskal's algorithm */
#include<stdio.h>
#define MAX 20

struct edge
{
        int u;
        int v;
        int weight;
        struct edge *link;
}*front = NULL;

int father[MAX]; /*Holds father of each node */
struct edge tree[MAX]; /* Will contain the edges of spanning tree */
int n;  /*Denotes total number of nodes in the graph */
int wt_tree=0; /*Weight of the spanning tree */
int count=0;  /* Denotes number of edges included in the tree */

/* Functions */
void make_tree( );
void insert_tree(int i,int j,int wt);
void insert_pque(int i,int j,int wt);
struct edge *del_pque( );

main( )
{
        int i;
        create_graph( );
        make_tree( );
        printf("Edges to be included in spanning tree are :\n");
        for(i=1;i<=count;i++)
        {
                printf("%d->",tree[i].u);
```

```
                printf("%d\n",tree[i].v);
        }
        printf("Weight of this minimum spanning tree is : %d\n", wt_tree);
}/*End of main( )*/

create_graph( )
{
        int i,wt,max_edges,origin,destin;

        printf("Enter number of nodes : ");
        scanf("%d",&n);
        max_edges=n*(n-1)/2;
        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d(0 0 to quit): ",i);
                scanf("%d %d",&origin,&destin);
                if( (origin==0) && (destin==0) )
                        break;
                printf("Enter weight for this edge : ");
                scanf("%d",&wt);
                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        insert_pque(origin,destin,wt);
        }/*End of for*/
        if(i<n-1)
        {
                printf("Spanning tree is not possible\n");
                exit(1);
        }
}/*End of create_graph( )*/

void make_tree( )
{
        struct edge *tmp;
        int node1,node2,root_n1,root_n2;

        while( count < n-1) /*Loop till n-1 edges included in the tree*/
        {
                tmp=del_pque( );
                node1=tmp->u;
                node2=tmp->v;
                printf("n1=%d  ",node1);
                printf("n2=%d  ",node2);
```

```
            while( node1 > 0)
            {
                    root_n1=node1;
                    node1=father[node1];
            }
            while( node2 >0 )
            {
                    root_n2=node2;
                    node2=father[node2];
            }
            printf("rootn1=%d  ",root_n1);
            printf("rootn2=%d\n",root_n2);

            if(root_n1!=root_n2)
            {
                insert_tree(tmp->u,tmp->v,tmp->weight);
                wt_tree=wt_tree+tmp->weight;
                father[root_n2]=root_n1;
            }
        }/*End of while*/
}/*End of make_tree( )*/

/*Inserting an edge in the tree */
void insert_tree(int i,int j,int wt)
{
        printf("This edge inserted in the spanning tree\n");
        count++;
        tree[count].u=i;
        tree[count].v=j;
        tree[count].weight=wt;
}/*End of insert_tree( )*/

/*Inserting edges in the priority queue */
void insert_pque(int i,int j,int wt)
{
        struct edge *tmp,*q;

        tmp = (struct edge *)malloc(sizeof(struct edge));
        tmp->u=i;
        tmp->v=j;
        tmp->weight = wt;

        /*Queue is empty or edge to be added has weight less than first edge*/
        if( front == NULL || tmp->weight < front->weight )
        {
                tmp->link = front;
                front = tmp;
        }
```

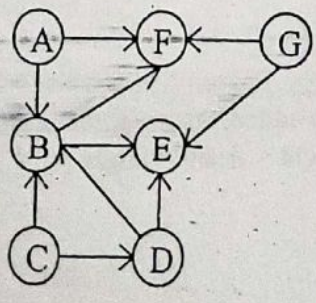
```
        else
        {
                q = front;
                while( q->link != NULL && q->link->weight <= tmp->weight )
                        q=q->link;
                tmp->link = q->link;
                q->link = tmp;
                if(q->link == NULL)    /*Edge to be added at the end*/
                        tmp->link = NULL;
        }/*End of else*/
}/*End of insert_pque( )*/

/*Deleting an edge from the priority queue*/
struct edge *del_pque( )
{
        struct edge *tmp;
        tmp = front;
        printf("Edge processed is %d->%d  %d\n",tmp->u,tmp->v,tmp->weight);
        front = front->link;
        return tmp;
}/*End of del_pque( )*/
```

Topological Sorting-

There are so many applications where execution of one task is necessary before starting another task. Like understanding of 'C' language, programming is necessary before starting 'Data Structure Through C'. Similarly in booting process of computer or printing process from printer. Same in this book also after understanding of tree, we are going to heap sort or binary tree sort.

In any directed graph which has no cycle, topological sort gives the sequential order of all the nodes x,y $\in$G and x comes before y in sequential order if a path exists from x to y. So this sequential order will indicate the dependency of one task on another.

Let us take a directed graph G.



Adjacency list of G will be-

A→ B, F
B→ E, F
C → B,D
D→ B,E
E→
F→
G→ E,F

Now we have a need to find the way to get topological sorting. We should first take those nodes which have no predecessors. It can be chosen very easily. The nodes which have indegree (number of edges coming to that node) zero will be those nodes. So topological sorting can be defined as-

1. Take all the nodes which have zero indegree.
2. Delete those nodes and edges going from those nodes .
3. Do the same process again until all the nodes are deleted.

This operation will need a queue where all the nodes which have indegree zero will be first added then deleted. Every time we need to get the information of indegree of every node. One array is also required where all the nodes will be in sequence of deletion from queue. Let us take a graph and find the topological sorting-
Here we are taking a queue for maintaining the zero indegree elements and array topSort for sequence of deleted elements from queue.

Step1-
Indegree of nodes are-
A=0, B=3, C=0, D=1, E=3, F=3, G=0
Step2-
(i) Taking all the nodes, which have zero indegree.
A, C, G
(ii) Add all zero indegree nodes to queue
Queue: A,C,G Front=1, Rear=3 topSort:
(iii) Delete the node A and edges going from A.
Queue:C,G Front=2,Rear=3 topSort:A
(iv) Now the indegree of nodes will be-
B=2, D=1, E=3, F=2
Step3-
(i) Delete the node C and edges going from C
Queue:G Front=3,Rear=3 topSort:A,C
(ii) Now the indegree of nodes will be-
B=1, D=0, E=3, F=2

Step4-
(i) Add D to queue
Queue:G,D Front=3,Rear=4 topSort:A,C
(ii) Delete the node G and edges going from G.
Queue:D Front=4,Rear=4 topSort:A,C,G
(iii) Now the indegree of nodes will be-
B=1, E=2, F=2

Step5-
(i) Delete the node D and edges going from D.
Queue: Front=5,Rear=4 topSort:A,C,G,D
(ii) Now the indegree of nodes will be-
B=0, E=1, F=2

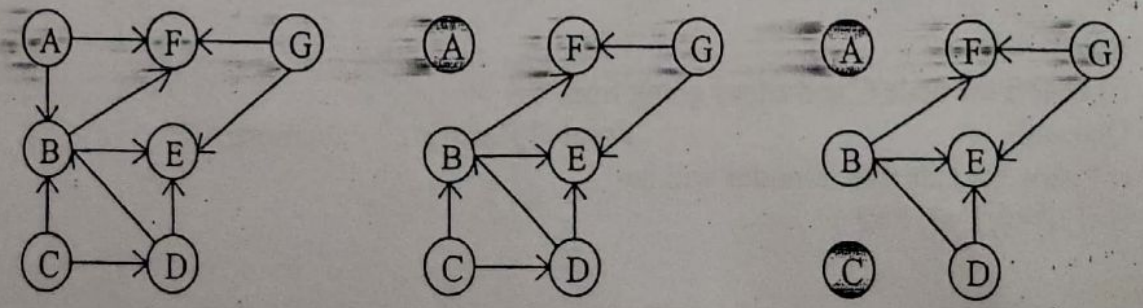
Step6-
(i) Add B to queue
Queue: B Front = 5, Rear = 5 topSort: A,C,G,D
(ii) Delete the node B and edges going from B.
Queue: Front = 6, Rear = 5 topSort: A,C,G,D,B
(iii) Now the indegree of nodes will be-
E = 0, F = 1

Step 7-
(i) Add E to queue
Queue: E Front = 6, Rear = 5 topSort: A,C,G,D,B
(ii) Delete the node E and edges going from E
Queue: Front = 7, Rear = 5 topSort: A,C,G,D,B,E
(iii) Now the indegree of nodes will be-
F = 0

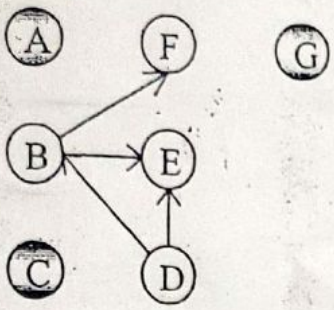
Step 8-
(i) Add F to queue
Queue: F Front = 7, Rear = 7 topSort: A,C,G,D,B,E
(ii) Delete the node F and edges going from F
Queue: Front = 8, Rear = 7 topSort: A,C,G,D,B,E,F

Now we have no more nodes in the graph. So the topological sorting of graph will be-
A, C, G, D, B, E, F

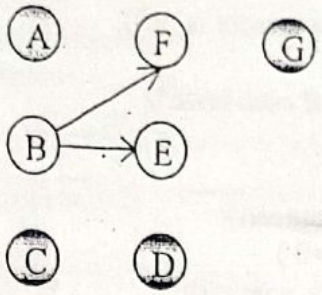
Step 1 **Step 2** **Step 3**



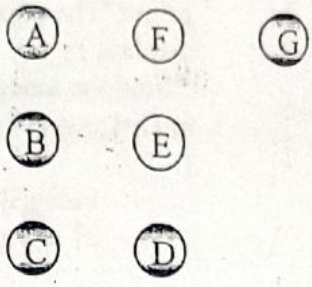
Step 4



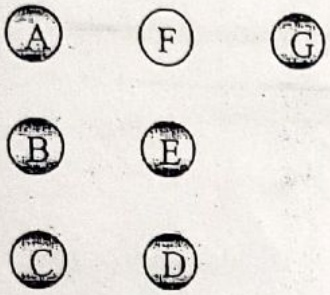
Step 5



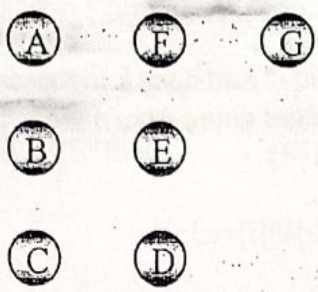
Step 6

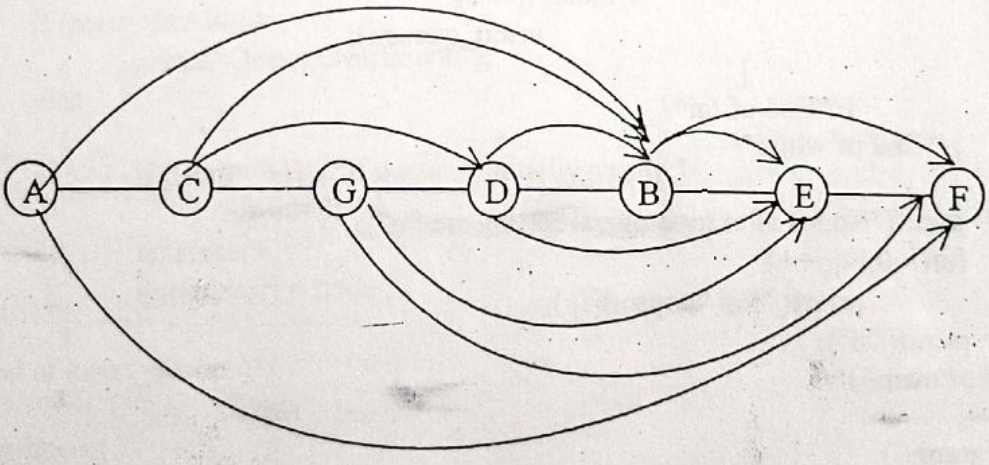


Step 7



Step 8






```
/* Program for topological sorting */
#include<stdio.h>
#define MAX 20

int n,adj[MAX][MAX];
int front=-1,rear=-1,queue[MAX];

main( )
{
        int i,j=0,k;
        int topsort[MAX],indeg[MAX];

        create_graph( );
```

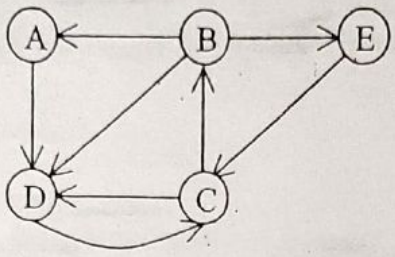
```
        printf("The adjacency matrix is :\n");
        display( );
        /*Find the indegree of each node*/
        for(i=1;i<=n;i++)
        {
                indeg[i]=indegree(i);
                if( indeg[i]==0 )
                        insert_queue(i);
        }

        while(front<=rear) /*Loop till queue is not empty */
        {
                k=delete_queue( );
                topsort[j++]=k; /*Add node k to topsort array*/
                /*Delete all edges going from node k */
                for(i=1;i<=n;i++)
                {
                        if( adj[k][i]==1 )
                        {
                                adj[k][i]=0;
                                indeg[i]=indeg[i]-1;
                                if(indeg[i]==0)
                                        insert_queue(i);
                        }
                }/*End of for*/
        }/*End of while*/

        printf("Nodes after topological sorting are :\n");
        for(i=0;i<j;i++)
                printf( "%d ",topsort[i] );
        printf("\n");
}/*End of main( )*/

create_graph( )
{
        int i,max_edges,origin,destin;

        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edges=n*(n-1);

        for(i=1;i<=max_edges;i++)
        {
                printf("Enter edge %d(0 0 to quit): ",i);
                scanf("%d %d",&origin,&destin);

                if((origin==0) && (destin==0))
                        break;
```

```
                if( origin > n || destin > n || origin<=0 || destin<=0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin]=1;
        }/*End of for*/
}/*End of create_graph( )*/

display( )
{
        int i,j;
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        printf("%3d",adj[i][j]);
                printf("\n");
        }
}/*End of display( )*/

insert_queue(int node)
{
        if (rear==MAX-1)
                printf("Queue Overflow\n");
        else
          {
                if (front==-1)  /*If queue is initially empty */
                        front=0;
                rear=rear+1;
                queue[rear] = node ;
          }
}/*End of insert_queue( )*/

delete_queue( )
{
        int del_item;
        if (front == -1 || front > rear)
        {
                printf("Queue Underflow\n");
                return ;
        }
        else
        {
                del_item=queue[front];
                front=front+1;
                return del_item;
        }
}/*End of delete_queue( ) */
```

```
int indegree(int node)
{
        int i,in_deg=0;
        for(i=1;i<=n;i++)
                if( adj[i][node] == 1 )
                        in_deg++;
        return in_deg;
}/*End of indegree( ) */
```

Exercise-

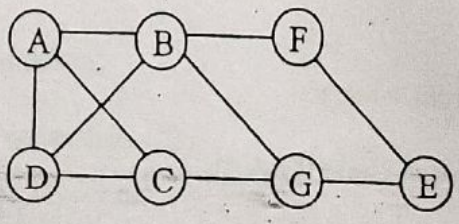
1. Find out the adjacency matrix and adjacency list for the graph given below-



Modify the adjaceny matrix and adjacency list after each of the operations given below.

(i) Delete the edge D→C.
(ii) Insert an edge A→E.
(iii) Delete the node E from the graph.
(iv) Insert a new node F in the graph.
(v) Insert an edge C→F.

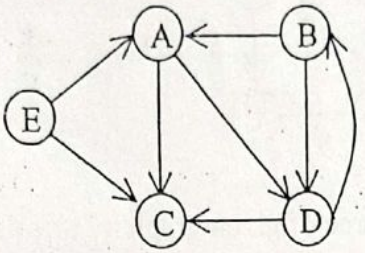
2.Consider the graph given below and anwer the following questions



(a) Find all the simple paths from node A to node E. Compute the lengths of each of these paths and find out the shortest path.
(b)Find out the degree of each node in the graph. Is there any isolated node in the graph?
(c) Is the above graph a connected graph?
(d) Are there any cycles in the above graph, if yes then how many.
(e) Is the above graph a tree?

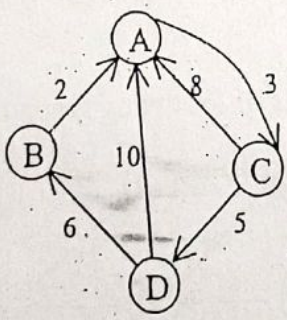
(f) How many maximum edges can be there in this graph?
(g) Find the diameter of the above graph.(the diameter of a graph is defined as the length of the longest path in the graph)

3.Consider the directed graph given below and answer the following questions.



(i) Write down the indegree and outdegree of each node in the graph.
(ii) Which nodes are sources and sinks in this graph?
(iii) Is the graph weakly connected or strongly connected?
(iv) Is the above graph a complete graph?
(v) Are there any cycles in the above graph?

4. Consider the graph given below.

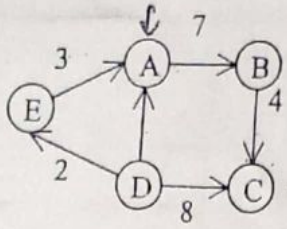


(i) Find out the adjacency matrix for this graph.
(ii) Find out the path matrix by powers of adjacency matrix.
(iii) Find out the path matrix by Warshall's algorithm.

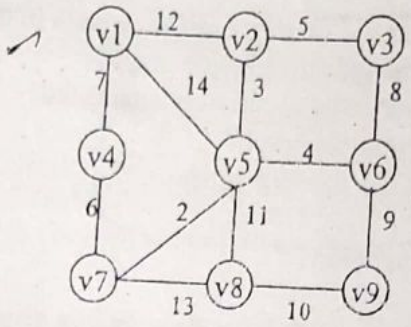
5. Find out the shortest path matrix by modified warshalls algorithm for graph whose weighted adjacency matrix is given below

$$W = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 0 & 3 & 6 \\ 0 & 5 & 0 & 0 \\ 8 & 0 & 7 & 5 \\ 4 & 2 & 0 & 9 \end{bmatrix} \end{array}$$

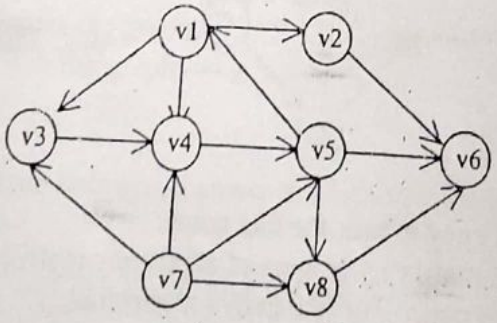
6. Find out the shortest path between all pair of nodes in the given graph by djikstra algorithm.



7. Find out the minimum spanning tree of given graph by prim's and kruskal's algorithm.



8. Find out the DFS and BFS traversals of following graph taking v1 as the starting node.



10. Write a program which takes input as an adjacency matrix and find out whether there are any loops in the graph.

11. Write a program which finds out whether a graph is regular or not.

12. Write a program to find the indegree and outdegree of a node in a directed graph.

13. Write a program to find out the number of source nodes and sink nodes in a graph.

14. Write a program which finds out all the successors and predecessors of a given node.