9 Graphs

This chapter discusses another nonlinear data structures, graphs. Graphs representations have found application in almost all subjects like geography, engineering and solving games and puzzles.

A graph G consist of

1. Set of vertices V (called nodes), (V = {v_1, v_2, v_3, v_4.....}) and

2. Set of edges E (*i.e.*, E {e_1, e_2, e_3.....cm}

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge '*e*' in E is identified with a unique pair (*a*, *b*) of nodes in V, denoted by *e* = [*a*, *b*].

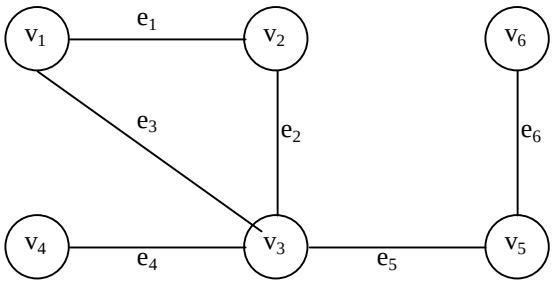


Fig. 9.1

Consider a graph, G in Fig. 9.1. Then the vertex V and edge E can be represented as: V = {v_1, v_2, v_3, v_4, v_5, v_6} and E = {e_1, e_2, e_3, e_4, e_5, e_6} E = {(v_1, v_2) (v_2, v_3) (v_1, v_3) (v_3, v_4), (v_3, v_5) (v_5, v_6)}. There are six edges and vertex in the graph

9.1. BASIC TERMINOLOGIES

A *directed graph* G is defined as an ordered pair (V, E) where, V is a set of vertices and the ordered pairs in E are called edges on V. A directed graph can be represented geometrically as a set of marked points (called vertices) V with a set of arrows (called edges) E between pairs of points (or vertex or nodes) so that there is at most one arrow from one vertex to another vertex. For example, Fig 9.2 shows a directed graph, where G = {*a*, *b*, *c*, *d* }, {(*a*, *b*), (*a*, *d*), (*d*, *b*), (*d*, *d*), (*c*, *c*)}

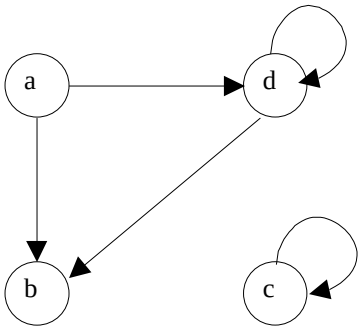


Fig. 9.2

An edge (*a*, *b*), in said to the incident with the vertices it joints, *i.e.*, *a*, *b*. We can also say that the edge (*a*, *b*) is incident from *a* to *b*. The vertex a is called the *initial vertex* and the vertex *b* is called the *terminal vertex* of the edge (*a*, *b*). If an edge that is incident from and into the same vertex, say (*d*, *d*) of (*c*, *c*) in Fig. 9.2, is called a *loop*.

Two vertices are said to be adjacent if they are joined by an edge. Consider edge (*a*, *b*), the vertex *a* is said to be adjacent to the vertex *b*, and the vertex *b* is said to be adjacent from the vertex *a*. A vertex is said to be an *isolated vertex* if there is no edge incident with it. In Fig. 9.2 vertex C is an isolated vertex.

An *undirected graph* G is defined abstractly as an ordered pair (V, E), where V is a set of vertices and the E is a set at edges. An undirected graph can be represented geometrically as a set of marked points (called vertices) V with a set at lines (called edges) E between the points. An undirected graph G is shown in Fig. 9.3.

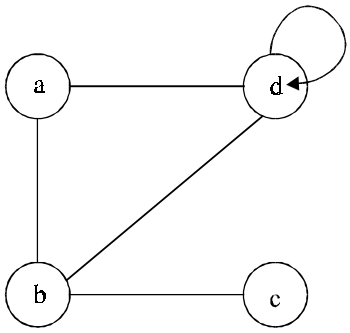


Fig. 9.3

Two graphs are said to be *isomorphic* if there is a one-to-one correspondence between their vertices and between their edges such that incidence are prevented. Fig. 9.4 show an isomorphic undirected graph.

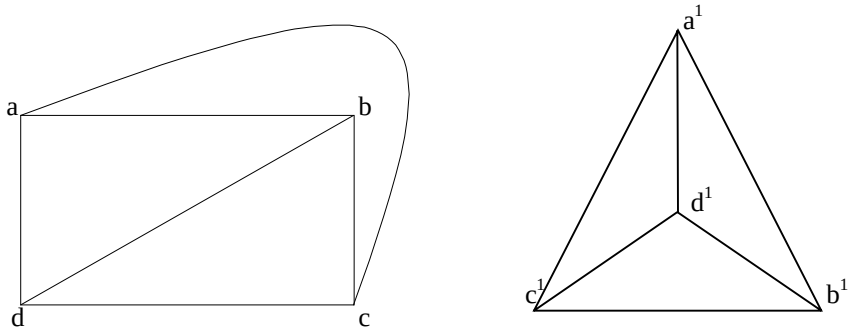


Fig. 9.4

Let G = (V, E) be a graph. A graph $G^1 = (V^1, E^1)$ is said to be a *sub-graph* of G if E^1 is a subset at E and V^1 is a subset at V such that the edges in E^1 are incident only with the vertices in V^1. For example Fig 9.5 (*b*) is a sub-graph at Fig. 9.5(*a*). A sub-graph of G is said to be a *spanning sub-graph* if it contains all the vertices of G. For example Fig. 9.5(*c*) shows a spanning sub-graph at Fig. 9.5(*a*).

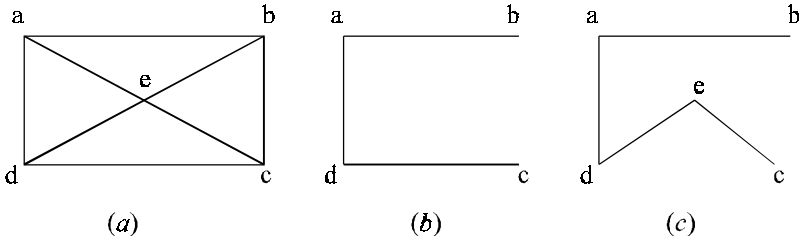


Fig. 9.5

The number of edges incident on a vertex is its *degree*. The degree of vertex *a*, is written as degree (*a*). If the degree of vertex a is zero, then vertex *a* is called isolated vertex. For example the degree of the vertex *a* in Fig. 9.5 is 3.

A graph G is said to be *weighted graph* if every edge and/or vertices in the graph is assigned with some weight or value. A weighted graph can be defined as $G = (V, E, W_e, W_v)$ where V is the set of vertices, E is the set at edges and W_e is a weights of the edges whose domain is E and W_v is a weight to the vertices whose domain is V. Consider a graph.

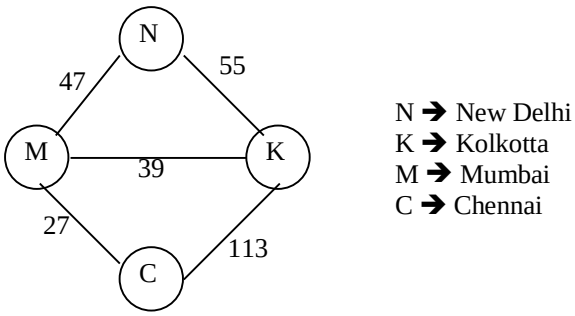


Fig. 9:6

In Fig 9:6 which shows the distance in km between four metropolitan cities in India. Here V = {N, K, M, C,} E = {(N, K), (N,M,), (M,K), (M,C), (K,C)} W_e = {55,47, 39, 27, 113} and W_v = {N, K, M, C} The weight at the vertices is not necessary to maintain have become the set W_v and V are same.

An undirected graph is said to be *connected* if there exist a path from any vertex to any other vertex. Otherwise it is said to be *disconnected*.

Fig. 9.7 shows the disconnected graph, where the vertex *c* is not connected to the graph. Fig. 9.8 shows the connected graph, where all the vertexes are connected.

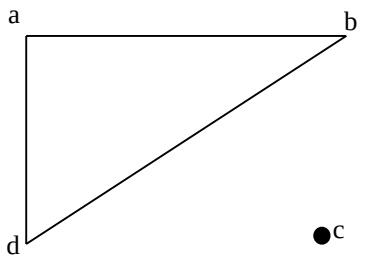


Fig. 9.7

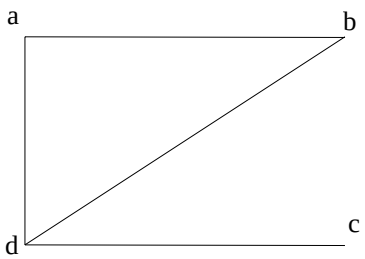


Fig. 9.8

A graph G is said to complete (or fully connected or strongly connected) if there is a path from every vertex to every other vertex. Let *a* and *b* are two vertices in the directed graph, then it is a complete graph if there is a path from *a* to *b* as well as a path from *b* to *a*. A complete graph with *n* vertices will have $n(n-1)/2$ edges. Fig 9.9 illustrates the complete undirected graph and Fig 9.10 shows the complete directed graph.

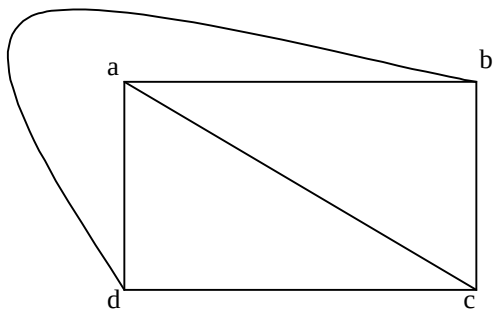


Fig. 9.9

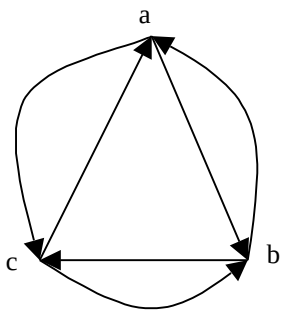


Fig 9:10

In a directed graph, a *path* is a sequence of edges (e_1, e_2, e_3, ... e_n) such that the edges are connected with each other (*i.e.*, terminal vertex e_n coincides with the initial vertex e_1). A path is said to be *elementary* if it does not meet the same vertex twice. A path is said to be *simple* if it does not meet the same edges twice. Consider a graph in Fig. 9.11

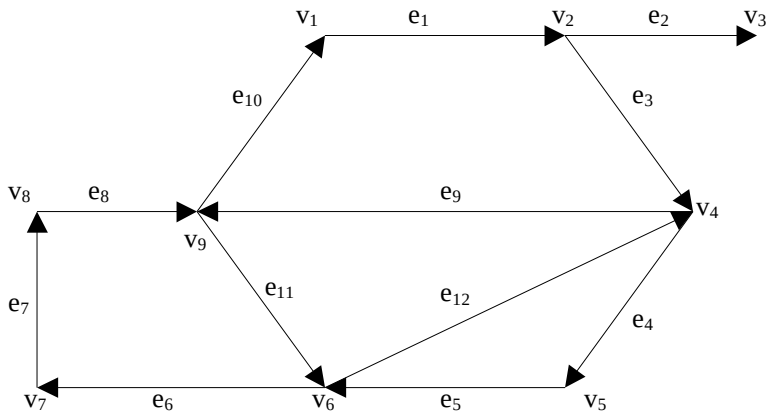


Fig. 9.11

Where (e_1, e_2, d_3, e_4, e_5) is a path; (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{11}, e_6, e_7, e_8, e_{11}) is a path but not a simple one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{11}, e_{12}) is a simple path but not elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8) is an elementary path.

A circuit is a path (e_1, e_2, ... e_n) in which terminal vertex of en coincides with initial vertex of e1. A circuit is said to be simple if it does not include (or visit) the same edge twice. A circuit is said to be elementary if it does not visit the same vertex twice. In Fig. 9:11 (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{10}) is a simple circuit but not a elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{10}) is an elementary circuit.

9.2. REPRESENTATION OF GRAPH

Graph is a mathematical structure and finds its application is many areas, where the problem is to be solved by computers. The problems related to graph G must be repre-

sented in computer memory using any suitable data structure to solve the same. There are two standard ways of maintaining a graph G in the memory of a computer.

1. Sequential representation of a graph using adjacent
2. Linked representation of a graph using linked list

9.2.1. ADJACENCY MATRIX REPRESENTATION

The s A of a graph G = (V, E) with *n* vertices, is an $n \times n$ matrix. In this section let us see how a directed graph can be represented using adjacency matrix. Considered a directed graph in Fig. 9.12 where all the vertices are numbered, (1, 2, 3, 4..... etc.)

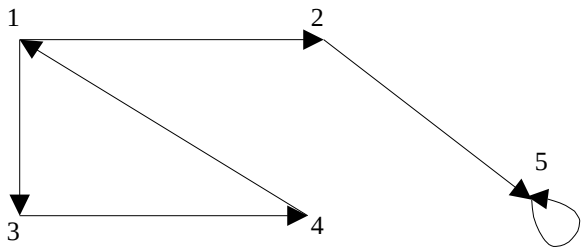


Fig. 9.12

The adjacency matrix A of a directed graph G = (V, E) can be represented (in Fig 9.13) with the following conditions

A_{ij} = 1 {if there is an edge from V_i to V_j or if the edge (*i*, *j*) is member of E.}

A_{ij} = 0 {if there is no edge from V_i to V_j}

i \ *j*	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	1

Fig. 9.13

We have seen how a directed graph can be represented in adjacency matrix. Now let us discuss how an undirected graph can be represented using adjacency matrix. Considered an undirected graph in Fig. 9.14

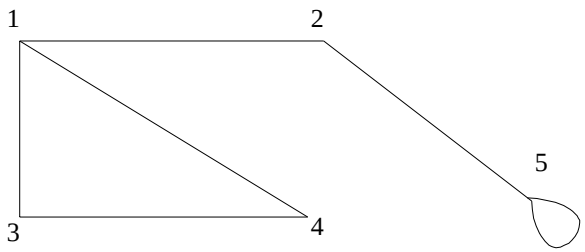


Fig. 9.14

The adjacency matrix A of an undirected graph G = (V, E) can be represented (in Fig 9.15) with the following conditions

A_{ij} = 1 {if there is an edge from V_i to V_j or if the edge (*i*, *j*) is member of E}

A_{ij} = 0 {if there is no edge from V_i to V_j or the edge *i*, *j*, is not a member of E}

i \ *j*	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	0
5	0	1	0	0	1

Fig. 9.15

To represent a weighted graph using adjacency matrix, weight of the edge (*i*, *j*) is simply stored as the entry in *i* th row and *j* th column of the adjacency matrix. There are some cases where zero can also be the possible weight of the edge, then we have to store some sentinel value for non-existent edge, which can be a negative value; since the weight of the edge is always a positive number. Consider a weighted graph, Fig. 9.16

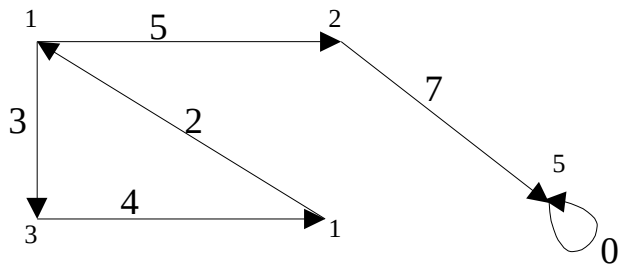


Fig. 9.16

The adjacency matrix A for a directed weighted graph G = (V, E, W_e) can be represented (in Fig. 9.17) as

$A_{ij} = W_{ij}$ { if there is an edge from V_i to V_j then represent its weight W_{ij}.}

$A_{ij} = -1$ { if there is no edge from V_i to V_j}

i \ j	1	2	3	4	5
1	– 1	5	3	– 1	– 1
2	– 1	– 1	– 1	– 1	7
3	– 1	– 1	– 1	4	– 1
4	2	– 1	– 1	– 1	– 1
5	– 1	– 1	– 1	– 1	0

Fig. 9.17

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

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

9.2.2. LINKED LIST REPRESENTATION

In this representation (also called adjacency list representation), we store a graph as a linked structure. First we store all the vertices of the graph in a list and then each adjacent vertices will be represented using linked list node. Here terminal vertex of an edge is stored in a structure node and linked to a corresponding initial vertex in the list. Consider a directed graph in Fig. 9.12, it can be represented using linked list as Fig. 9.18.

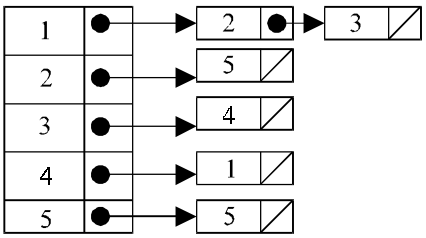


Fig. 9.18

Weighted graph can be represented using linked list by storing the corresponding weight along with the terminal vertex of the edge. Consider a weighted graph in Fig. 9.16, it can be represented using linked list as in Fig. 9.19.

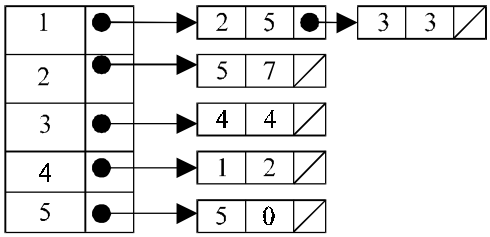


Fig. 9.19

Although the linked list representation requires very less memory as compared to the adjacency matrix, the simplicity of adjacency matrix makes it preferable when graph are reasonably small.

9.3. OPERATIONS ON GRAPH

Suppose a graph G is maintained in memory by the linked list representation. This section discuss the different operations such as creating a graph, searching, deleting a vertices or edges.

9.3.1. CREATING A GRAPH

To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

Step 1: Input the total number of vertices in the graph, say *n*.

Step 2: Allocate the memory dynamically for the vertices to store in list array.

Step 3: Input the first vertex and the vertices through which it has edge(s) by linking the node from list array through nodes.

Step 4: Repeat the process by incrementing the list array to add other vertices and edges.

Step 5: Exit.

9.3.2. SEARCHING AND DELETING FROM A GRAPH

Suppose an edge (1, 2) is to be deleted from the graph G. First we will search through the list array whether the initial vertex of the edge is in list array or not by incrementing the list array index. Once the initial vertex is found in the list array, the corresponding link list will be search for the terminal vertex.

Step 1: Input an edge to be searched

Step 2: Search for an initial vertex of edge in list arrays by incrementing the array index.

Step 3: Once it is found, search through the link list for the terminal vertex of the edge.

Step 4: If found display “the edge is present in the graph”.

Step 5: Then delete the node where the terminal vertex is found and rearrange the link list.

Step 6: Exit

PROGRAM 9.1

```
//PROGRAM TO IMPLEMENT ADDITION AND DELETION OF NODES
//AND EDGES IN A GRAPH USING ADJACENCY MATRIX
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
#include<process.h>

#define max 20

int adj[max][max];
int n;

void create_graph()
{
      int i,max_edges,origin,destin;
      clrscr();
      printf ("\nEnter number of nodes:");
      scanf("%d",&n);
      max_edges=n*(n–1); /* Taking directed graph */

      for(i=1;i<=max_edges;i++)
      {
            printf ("\nEnter 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 ("\nInvalid edge!\n");
                  i--;
            }
            else
                  adj[origin][destin]=1;
      }/*End of for*/
```

```
}/*End of create_graph()*/

void 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()*/

void insert_node()
{
	int i;
	n++;	/*Increase number of nodes in the graph*/
	printf ("\nThe inserted node is %d \n",n);
	for(i=1;i<=n;i++)
	{
		adj[i][n]=0;
		adj[n][i]=0;
	}
}/*End of insert_node()*/
void delete_node(char u)
{
	int i,j;
	if(n==0)
	{
		printf ("\nGraph is empty\n");
		return;
	}
	if ( u>n )
	{
		printf ("\nThis 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*/
void insert_edge(char u,char v)
{
    if (u > n)
    {
        printf ("\nSource node does not exist\n");
        return;
    }
    if(v > n)
    {
        printf("\nDestination node does not exist\n");
        return;
    }
    adj[u][v]=1;
}/*End of insert_edge()*/

void del_edge(char u,char v)
{
   if (u>n || v>n || adj[u][v]==0)
   {
      printf("\nThis edge does not exist\n");
      return;
   }
   adj[u][v]=0;
}/*End of del_edge()*/
void main()
{
    int choice;
    int node,origin,destin;

    create_graph();
    while(1)
    {
        clrscr();
        printf ("\n1.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 ({\nEnter your choice:");
            scanf ("%d",&choice);

            switch(choice)
            {
             case 1:
                   insert_node();
                   break;
             case 2:
                   printf("\nEnter an edge to be inserted:");
                   fflush(stdin);
                   scanf("%d %d",&origin,&destin);
                   insert_edge(origin,destin);
                   break;
             case 3:
                   printf ("\nEnter a node to be deleted:");
                   fflush(stdin);
                   scanf ("%d",&node);
                   delete_node(node);
                   break;
             case 4:
                   printf ("\nEnter an edge to be deleted:");
                   fflush(stdin);
                   scanf ("%d %d",&origin,&destin);
                   del_edge(origin,destin);
                   break;
             case 5:
                   display();
                   break;
             case 6:
                   exit(0);
             default:
                   printf("\nWrong choice\n");
                   break;
            }/*End of switch*/
      }/*End of while*/
}/*End of main()*/
```

9.3.3. TRAVERSING A GRAPH

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

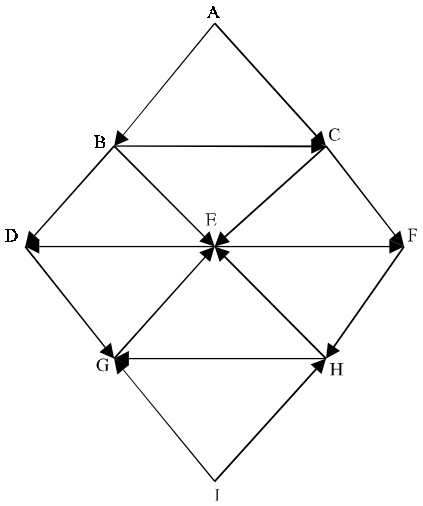
(*a*) Breadth First Search (BFS)

(*b*) Depth First Search (DFS)

9.4. BREADTH FIRST SEARCH

Given an input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverse the edges of G to explore every vertex that is reachable from S. Then we examine all the vertices neighbor to source vertex S. Then we traverse all the neighbors of the neighbors of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbor nodes.

BFS can be further discussed with an example. Considering the graph G in Fig. 9.20

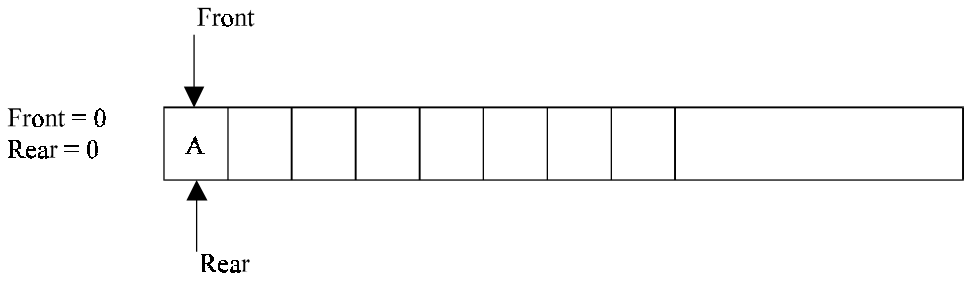


Vertex	*Adjacency list*
A	B, C
B	C, D, E
C	E, F
D	G
E	D, F
F	H
G	E
H	E, G
I	G, H

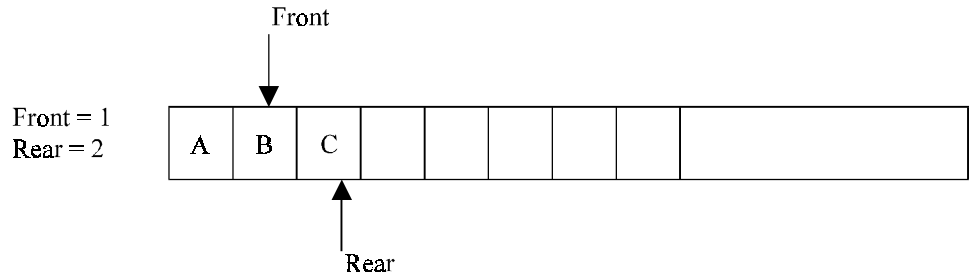
Fig. 9.20

The linked list (or adjacency list) representation of the graph Fig. 9.20 is also shown. Suppose the source vertex is A. Then following steps will illustrate the BFS.

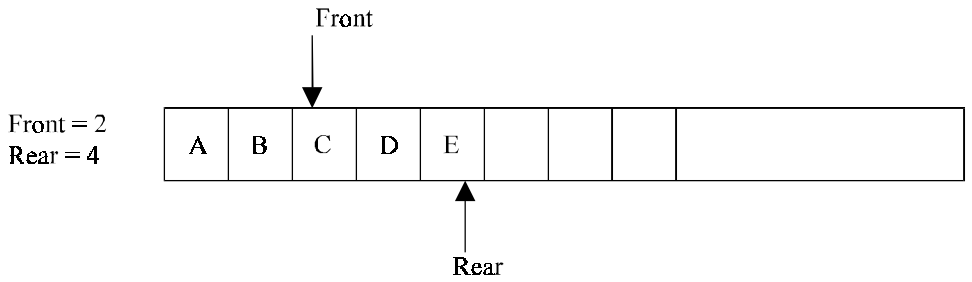
Step 1: Initially push A (the source vertex) to the queue.



Step 2: Pop (or remove) the front element A from the queue (by incrementing front = front +1) and display it. Then push (or add) the neighboring vertices of A to the queue, (by incrementing Rear = Rear +1) if it is not in queue.

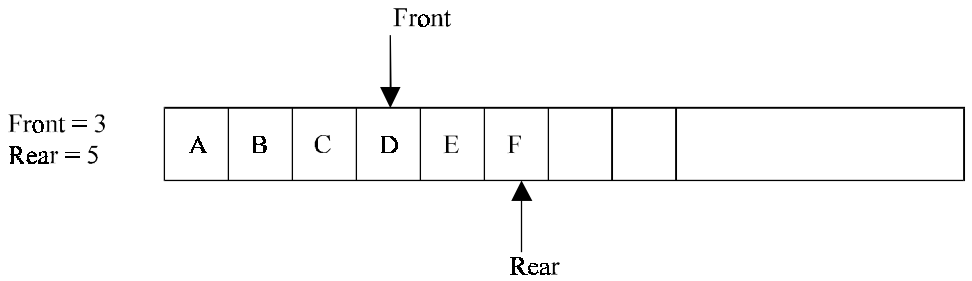


Step 3: Pop the front element B from the queue and display it. Then add the neighboring vertices of B to the queue, if it is not in queue.



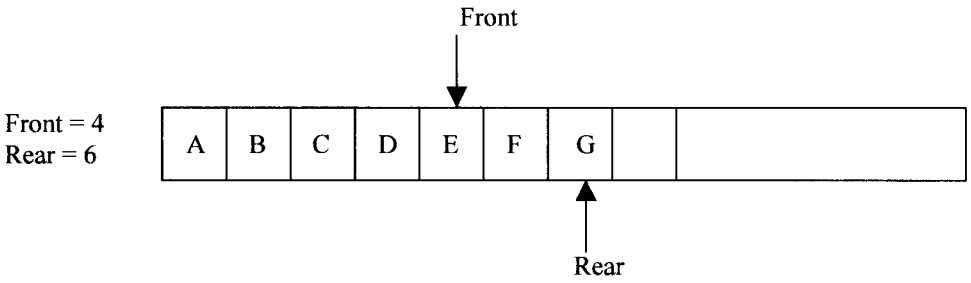
One of the neighboring element C of B is preset in the queue, So C is not added to queue.

Step 4: Remove the front element C and display it. Add the neighboring vertices of C, if it is not present in queue.

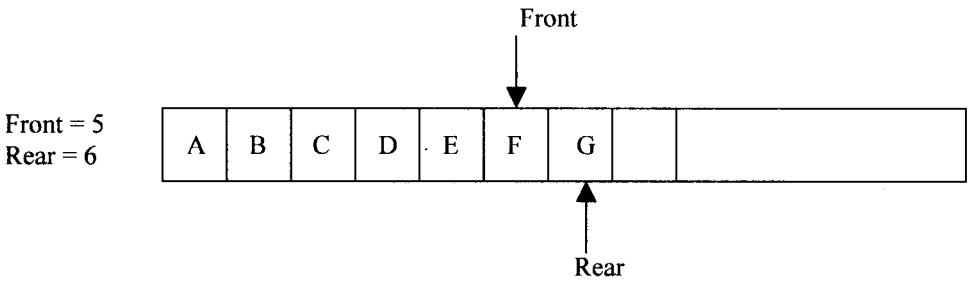


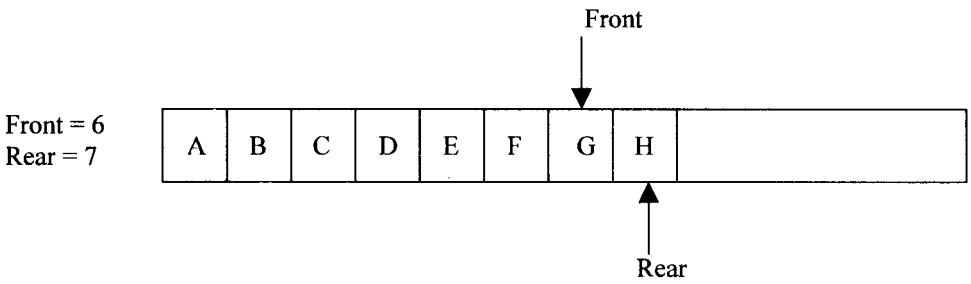
One of the neighboring elements E of C is present in the queue. So E is not added.

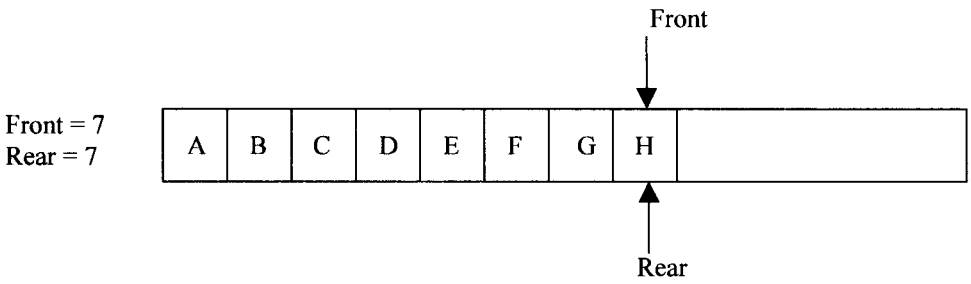
Step 5: Remove the front element D, and add the neighboring vertex if it is not present in queue.

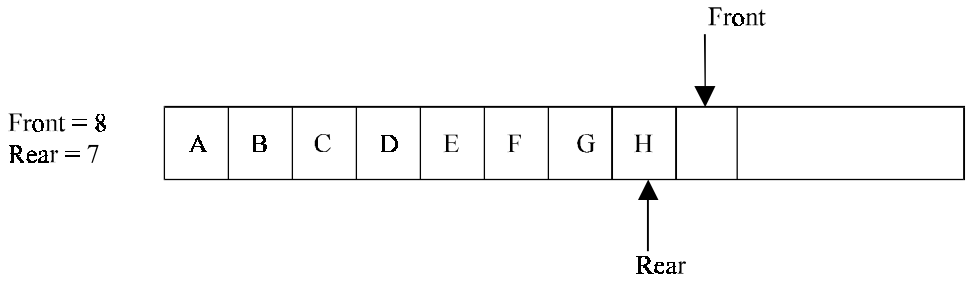


Step 6: Again the process is repeated (Until front > Rear). That is remove the front element E of the queue and add the neighboring vertex if it is not present in queue.









So A, B, C, D, E, F, G, H is the BFS traversal of the graph in Fig. 9.20

ALGORITHM

1. Input the vertices of the graph and its edges G = (V, E)
2. Input the source vertex and assign it to the variable S.
3. Add or push the source vertex to the queue.
4. Repeat the steps 5 and 6 until the queue is empty (*i.e.*, front > rear)
5. Pop the front element of the queue and display it as visited.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the queue and displayed (*i.e.*, not visited).
7. Exit.

PROGRAM 9.2

```
//PROGRAM TO IMPLEMENT BFS USING LINKED LIST
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>

struct node
{
int data;
struct node *next;
};
typedef struct node *node;
node bpush(node,int);

node create(int n)
{
node b,t;
```

```
int i, j;
char c;
b=(node)malloc(n*sizeof (struct node));
printf ("\nEnter The %d Vertices",n);
for(i=0;i<n;i++)
{
        scanf ("%d",&b[i].data);
        b[i].next=NULL;
}
for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
        do
        {
printf ("\nThe vertice %d have any edge to %d (y/n)....:",b[i].data,b[j].data);
                        c=getche();
                }while(c!='n'&&c!='N'&&c!='y'&&c!='N');
                if(c=='y'||c=='Y')
                {
                        t=&b[i];
                        while(t->next!=NULL)
                        t=t->next;

                        t->next=(node)malloc(sizeof(struct node));
                        t=t->next;
                        t->next=NULL;
                        t->data=b[j].data;
                }
        }
return b;
}

void bfs(node b,int n)
{
        node h=NULL,f,t,m=NULL;
        int s,j=-1,i;
        printf ("\nEnter the Starting Vetice:");
        scanf ("%d",&s);
        do
        {
                j++;
}while(b[j].data!=s && j<n);
```

```
printf ("\n\tBFS Traversal:");
if(j >= n)
{
        printf ("\n%d is Not Present in the Graph",s);
        bfs(b,n);
        return;
}
h=bpush(h,b[j].data);
m=h;
while(h!=NULL)
{
    t=&b[j];
    while(t!=NULL)
    {
            t=t->next;
            if(t->data!=0)
            bpush(m,t->data);
}
    i=bpop(h);
    h=h->next;
    j=-1;
    do
    {
            j++;
    }while(b[j].data!=i && j<n );
}
getch();
return;
}

node bpush(node s,int i)
{
    int fla=1;
    node f=s,n=s;
    if(s==NULL)
{
        s=(node)malloc(sizeof(struct node));
        f=s;
        s->next=NULL;
        s->data=i;
}
else
```

```
{
        while(n!=NULL)
        {
            if(n->data==i)
            fla=0;
            n=n->next;
        }
        if(fla==1)
        {
            while(s->next!=NULL)
                s=s->next;
            s->next=(node)malloc(sizeof(struct node));
            s=s->next;
            s->next=NULL;
            s->data=i;
        }
}
return f;
}

int bpop(node f)
{
    int i=f->data;
    f=f->next;
    printf("\t%d",i);
    return i;
    }

void main()
{

    node b;
    int n;
    clrscr();
    printf("\nEnter The Number Of Vertices      :");
    scanf("%d",&n);
    b=create(n);
    bfs(b,n);
    return;
}
```

9.5. DEPTH FIRST SEARCH

The depth first search (DFS), as its name suggest, is to search deeper in the graph, when ever possible. Given an input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we visit a neighbor vertex of S and again a neighbor of a neighbor of S, and so on. The implementation of BFS is almost same except a stack is used instead of the queue. DFS can be further discussed with an example. Consider the graph in Fig. 9.20 and its linked list representation. Suppose the source vertex is I.

The following steps will illustrate the DFS

Step 1: Initially push I on to the stack.

STACK: I

DISPLAY:

Step 2: Pop and display the top element, and then push all the neighbors of popped element (*i.e.*, I) onto the stack, if it is not visited (or displayed or not in the stack.

STACK: G, H

DISPLAY: I

Step 3: Pop and display the top element and then push all the neighbors of popped the element (*i.e.*, H) onto top of the stack, if it is not visited.

STACK: G, E

DISPLAY: I, H

The popped element H has two neighbors E and G. G is already visited, means G is either in the stack or displayed. Here G is in the stack. So only E is pushed onto the top of the stack.

Step 4: Pop and display the top element of the stack. Push all the neighbors of the popped element on to the stack, if it is not visited.

STACK: G, D, F

DISPLAY: I, H, E

Step 5: Pop and display the top element of the stack. Push all the neighbors of the popped element onto the stack, if it is not visited.

STACK: G, D

DISPLAY: I, H, E, F

The popped element (or vertex) F has neighbor(s) H, which is already visited. Then H is displayed, and will not be pushed again on to the stack.

Step 6: The process is repeated as follows.

STACK: G

DISPLAY: I, H, E, F, D

STACK: //now the stack is empty

DISPLAY: I, H, E, F, D, G

So I, H, E, F, D, G is the DFS traversal of graph Fig 9:20 from the source vertex I.

Algorithm

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty.
5. Pop the top element of the stack and display it.
6. Push the vertices which is neighbor to just popped element, if it is not in the queue and displayed (ie; not visited).
7. Exit.

PROGRAM 9.3

```
//PROGRAM TO IMPLEMENT DFS USING ADJACENCY MATRIX
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
#define max 10
/* a function to build adjacency matrix of a graph */
void buildadjm(int adj[][max], int n)
{
    int i,j;
    for(i=0;i<n;i++)
       for(j=0;j<n;j++)
       {
           printf("\nEnter 1 if there is an edge from %d to %d, otherwise enter 0 \n",i,j);
           scanf("%d",&adj[i][j]);
       }
}

/* a function to visit the nodes in a depth first order */
void dfs(int x,int visited[],int adj[][max],int n)
{
   int j;
   visited[x] = 1;
   printf("\nThe node visited id %d\n",x);
   for(j=0;j<n;j++)
      if (adj[x][j] ==1 && visited[j] ==0)
           dfs(j,visited,adj,n);
}
void main()
{
    int adj[max][max],node,n;
```

```
    int i, visited[max];
    printf("\nEnter the number of nodes in graph maximum = %d\n",max);
    scanf("%d",&n);
    buildadjm(adj,n);
    for(i=0; i<n; i++)
    visited[i] =0;
    for(i=0; i<n; i++)
       if(visited[i] ==0)
              dfs(i,visited,adj,n);
}
```

9.6. MINIMUM SPANNING TREE

A minimum spanning tree (MST) for a graph G = (V, E) is a sub graph $G^1 = (V^1, E^1)$ of G contains all the vertices of G.

1. The vertex set V^1 is same as that at graph G.
2. The edge set E^1 is a subset of G.
3. And there is no cycle.

If a graph G is not a connected graph, then it cannot have any spanning tree. In this case, it will have a spanning forest. Suppose a graph G with *n* vertices then the MST will have $(n - 1)$ edges, assuming that the graph is connected.

A minimum spanning tree (MST) for a weighted graph is a spanning tree with minimum weight. That is all the vertices in the weighted graph will be connected with minimum edge with minimum weights. Fig. 9.22 shows the minimum spanning tree of the weighted graph in Fig. 9.21.

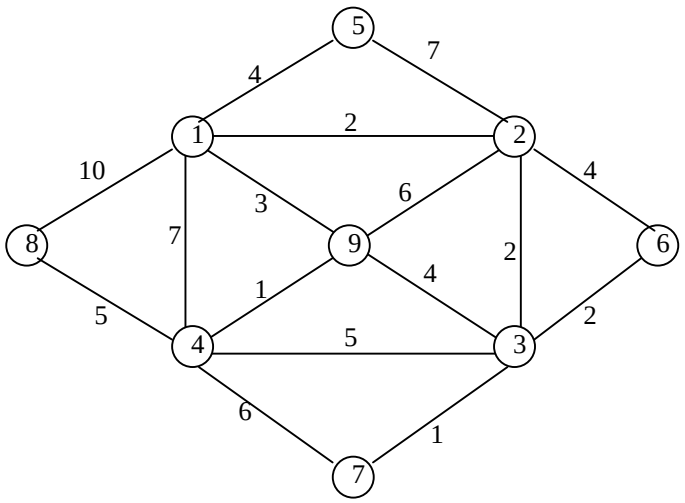


Fig. 9.21

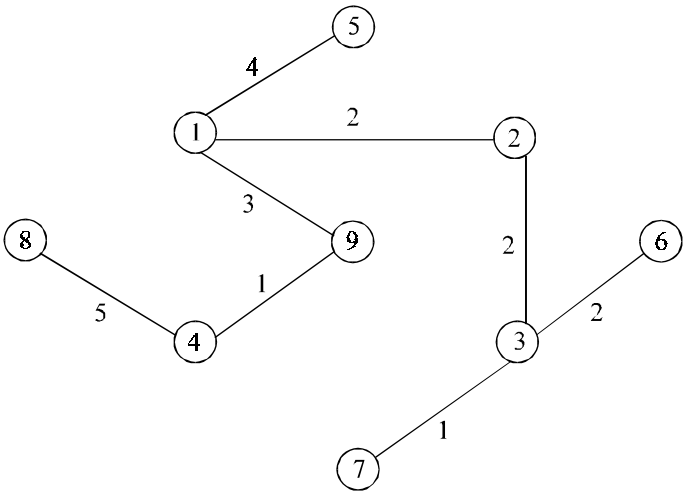


Fig. 9.22

Three different famous algorithms can be used to obtain a minimum spanning tree of a connected weighted and undirected graph.

1. Kruskal's Algorithm
2. Prim's Algorithm
3. Sollin's Algorithm

All three algorithms are using a design strategy called the greedy methods that is one which seeks maximum gain at each step.

9.6.1. KRUSKAL'S ALGORITHM

This is a one of the popular algorithm and was developed by Joseph Kruskal. To create a minimum cost spanning trees, using Kruskalls, we begin by choosing the edge with the minimum cost (if there are several edges with the same minimum cost, select any one of them) and add it to the spanning tree. In the next step, select the edge with next lowest cost, and so on, until we have selected (n – 1) edges to form the complete spanning tee. The only thing of which beware is that we don't form any cycles as we add edges to the spanning tree. Let us discuss this with an example. Consider a graph G in Fig. 9.21 to generate the minimum spanning tree.

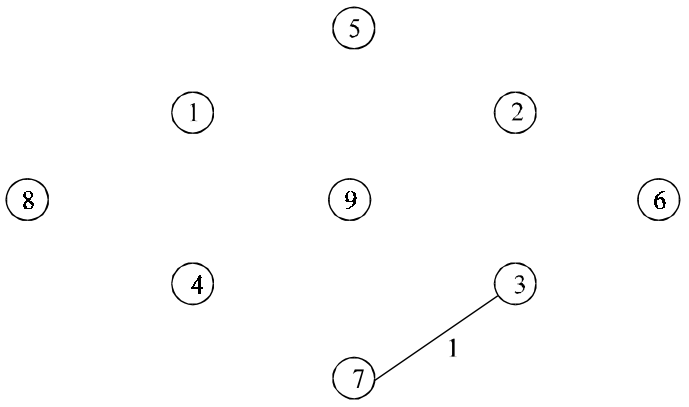


Fig. 9.23

The minimum cost edge in the graph G in Fig. 9.21 is 1. If you analyze closely there are two edges (*i.e.*, (7, 3), (4, 9)) with the minimum cost 1. As the algorithm says select any one of them. Here we select the edge (7, 3) as shown in Fig. 9.23. Again we select minimum cost edge (*i.e.*, 1), which is (4, 9) as shown in Fig. 9.24.

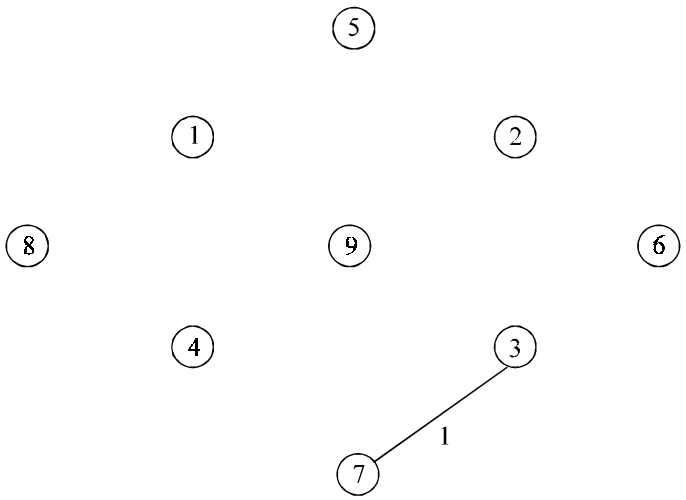


Fig 9:24

Next we select minimum cost edge (*i.e.*, 2). If you analyze closely there are two edges (*i.e.*, (1, 2), (2, 3), (3, 6)) with the minimum cost 2. As the algorithm says select any one of them. Here we select the edge (1, 2) as shown in the above Fig. 9.25. Again we select minimum cost edge (*i.e.*, 2), which is (2, 3) as shown in Fig. 9.26. Next we select minimum cost edge (*i.e.*, 2), which is (3, 6) as shown in Fig. 9.27.

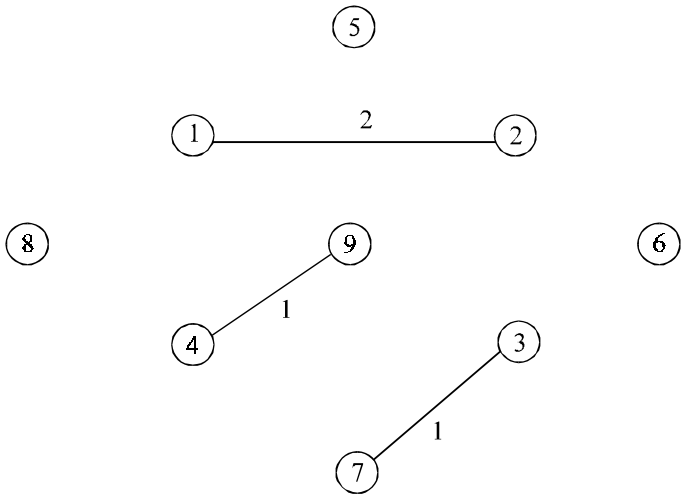


Fig. 9.25

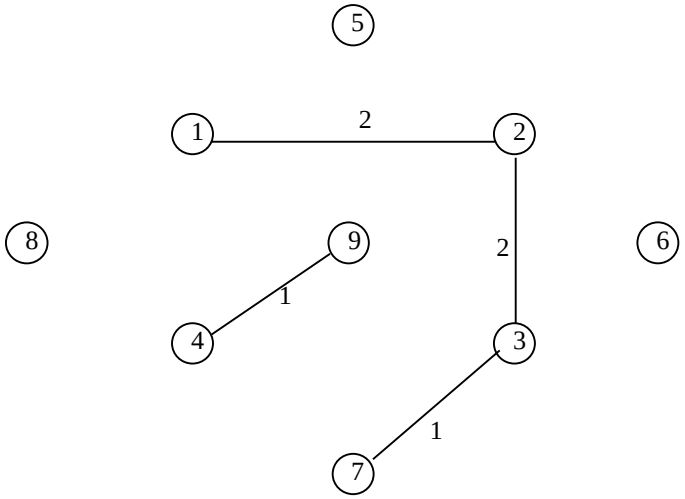


Fig. 9.26

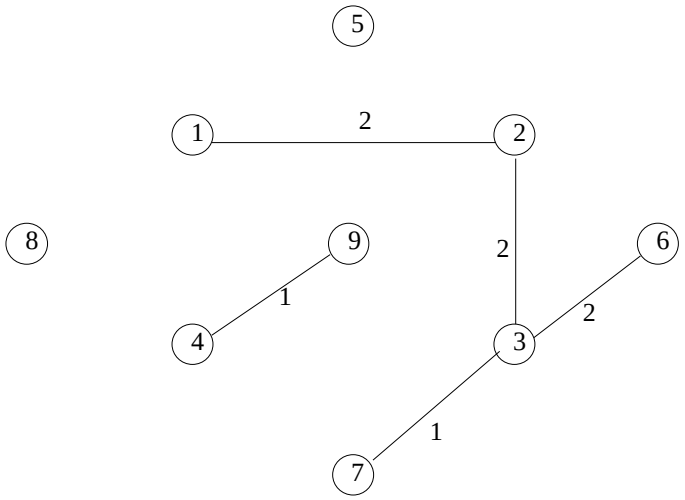


Fig. 9.27

Next minimum cost edge is (1, 9) with cost 3. Add the minimum cost edge to the minimum spanning tree as shown in Fig. 9.28. If we analyze, next minimum cost edge is (1, 5) with cost 4. Add the minimum cost edge to the minimum spanning tree as shown in Fig. 9.29.

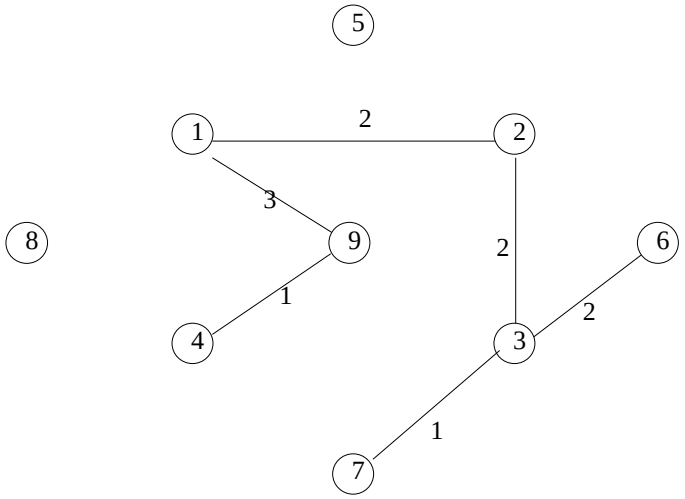


Fig. 9.28

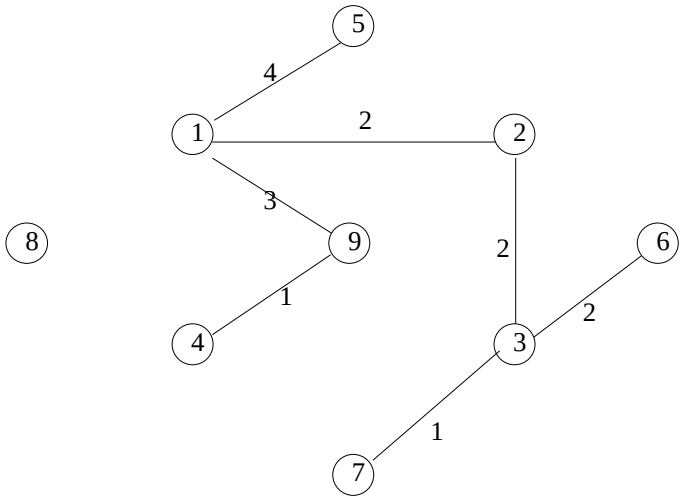


Fig. 9.29

Next minimum cost edge is (4, 8) with cost 5. Add the minimum cost edge to the minimum spanning tree as shown in Fig 9.30.

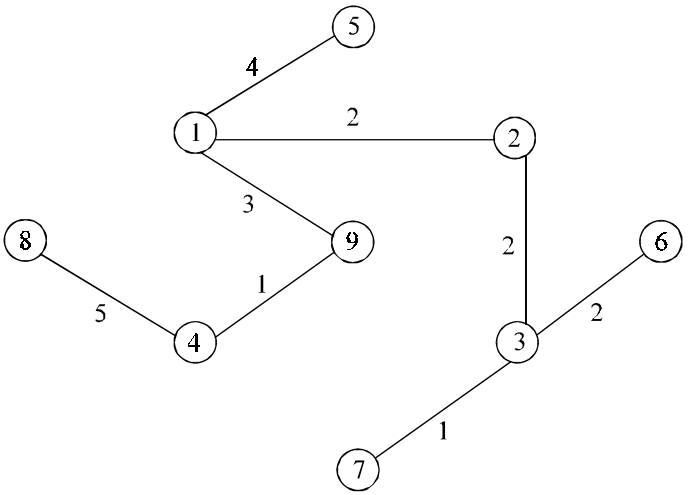


Fig. 9.30

Above figures shows different stages of Kruskasl's Algorithm.

ALGORITHM

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

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

PROGRAM 9.4

```
//PROGRAM TO CREATING A MINIMUM SPANNING TREE
//USING KRUSKAL'S ALGORITHM
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.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();

void 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 main()
{
        int i;
        clrscr();
        create_graph();
        make_tree();
        printf("\nEdges 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("\nWeight of this minimum spanning tree is : %d\n", wt_tree);
}/*End of main()*/

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()*/
```

9.6.2. JARNIK-PRIM'S ALGORITHM

This algorithm was discovered by Vojtech Jarnik in 1936 and later rediscovered by Robert Prim. Prim's algorithm also constructs the minimum-cost spanning tree, edge by edge. Prim's algorithm begin with a tree T that contain a single vertex (This vertex can be of any vertices in the original graph), generally it is selected as lower most cost edge in the tree. Then we add a least cost edge (u, v) to T such that T U { (u, v) } is also a tree. Repeat this edge-addition step until T contains $n - 1$ edges.

Construction of the minimum-cost spanning tree using Prim's algorithm can be explained with an example. Consider a graph G in Fig. 9.21.

The minimum cost edge in the graph G in Fig. 9:21 is 1. If you analyze closely there are two edges (*i.e.*, (4, 9), (7, 3)) with the minimum cost 1. As the Prim's algorithm says select any one of them. Here we select the edge (4, 9) as shown in Fig. 9.31.

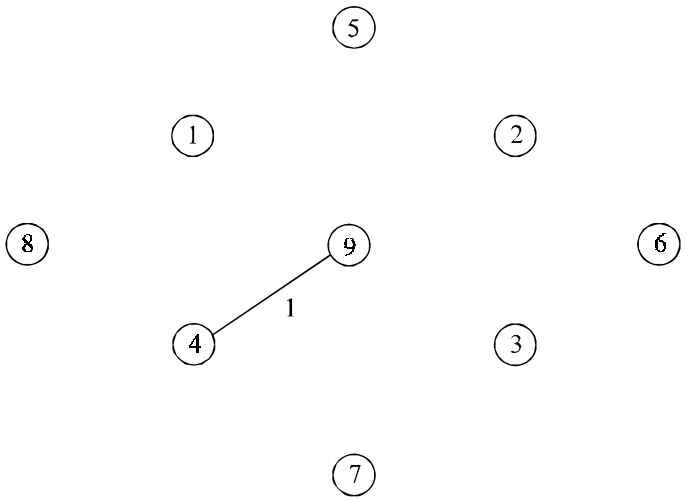


Fig. 9.31

Consider all the edges adjacent to the vertices of the recently selected edge (here recently selected edge is (4, 9)). And find the minimum cost edge that connects from recently selected edge. Here it is (9, 1) as shown in Fig. 9.32.

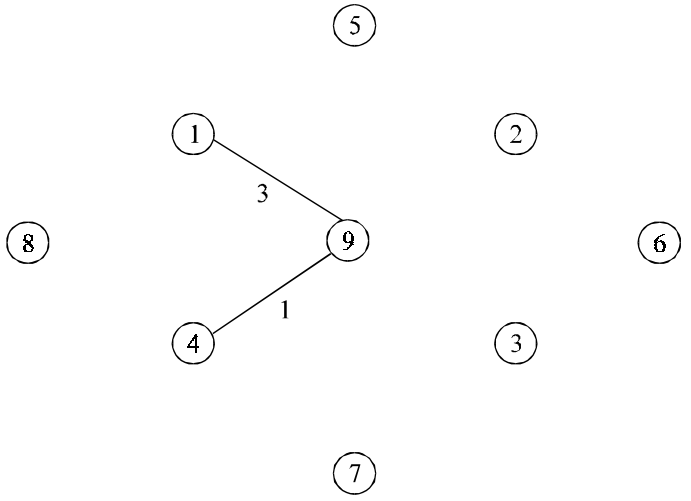


Fig. 9.32

Again consider all the edges adjacent to the vertices of the recently selected edge (here recently selected edge is (9, 1)). And find the minimum cost edge that connects from recently selected edge. Here it is (1, 2) as shown in Fig. 9.33.

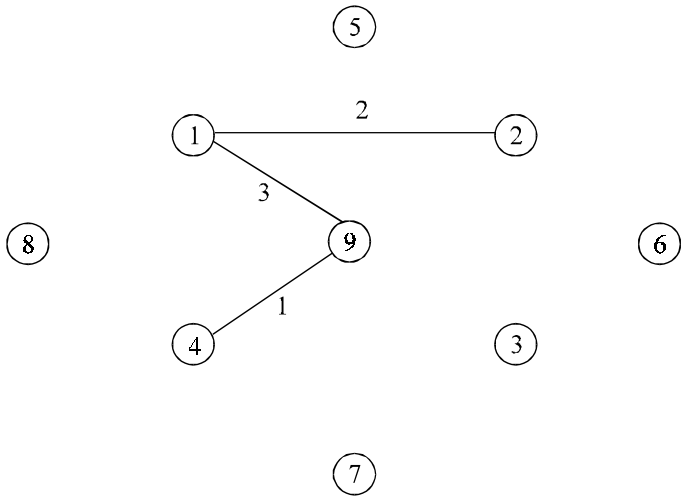


Fig. 9.33

Consider all the edges adjacent to the vertices of the recently selected edge (here recently selected edge is (1, 2)). And find the minimum cost edge that connects from recently selected edge. Here it is (2, 3) as shown in Fig 9:34. And repeat the process of finding the minimum cost edge that connects from recently selected edge.

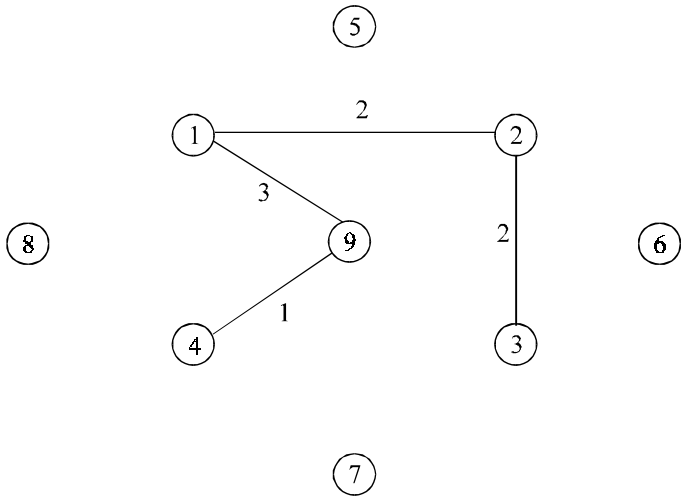


Fig. 9.34

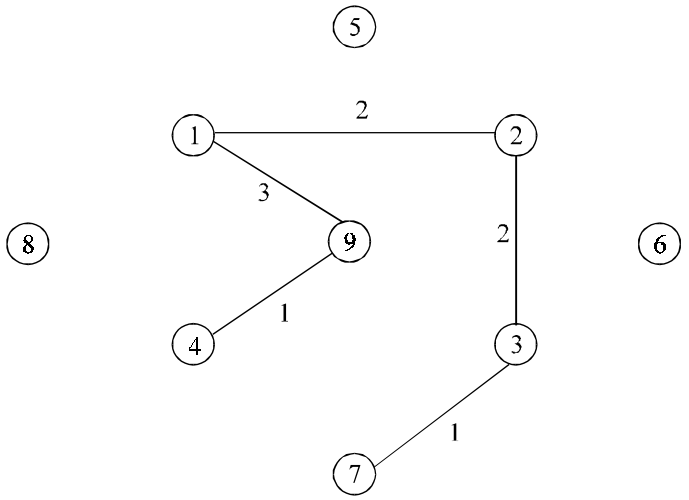


Fig. 9.35

Since all the vertices in the adjacent edges that can be reached from the recently selected edge (*i.e.*, (2, 3)) are visited, backtrack (or go back) to the path so as any other minimum cost adjacent edges is there to connect the vertices which are not connected yet. Thus we find the adjacent edge (3, 6) as shown in the Fig. 9.36.

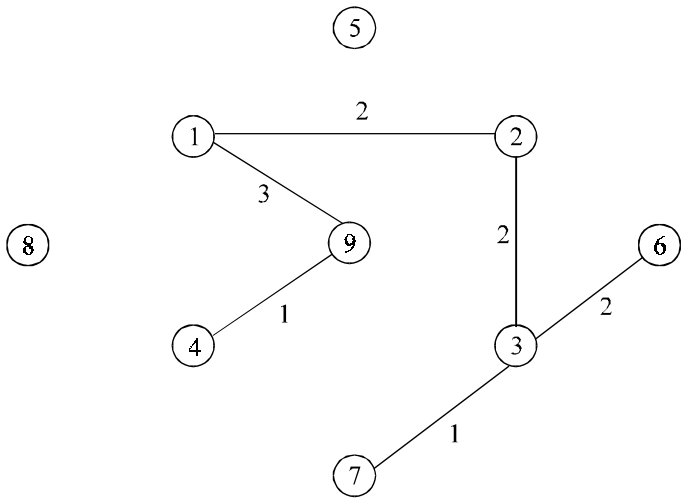


Fig. 9.36

Again since all the vertices in the adjacent edges that can be reached from the recently selected edge (*i.e.*, (3, 6)) are visited, backtrack (or go back) to the path so as any other minimum cost adjacent edges is there to connect the vertices which are not connected yet. Thus we find the adjacent edge (1, 5) as shown in the Fig. 9.37.

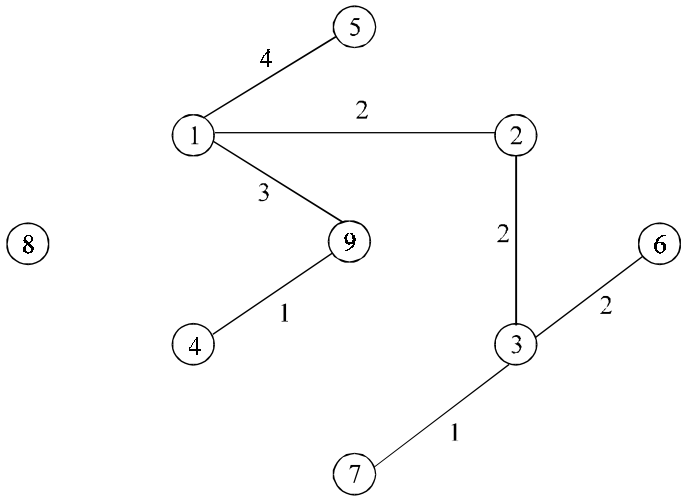


Fig. 9.37

Next we find the minimum cost adjacent edges (4, 8) as shown in the Fig. 9.38.

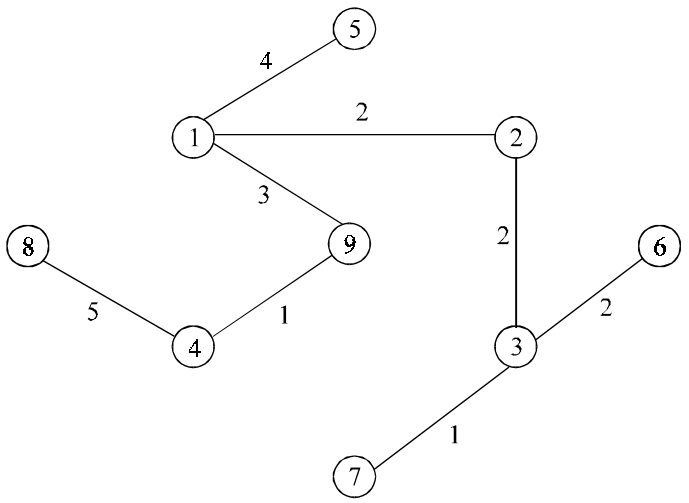


Fig. 9.38

ALGORITHM

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

1. Initialize the spanning tree T to contain a vertex v_1.
2. Choose an edge $e = (v_1, v_2)$ of G such that v_2 not equal to v_1 and e has smallest weight among the edges of G incident with v_1.
3. Select an edge $e = (v_2, v_3)$ of G such that v_2 is not equal to v_3 and e has smallest weight among the edge of G incident with v_2.

4. Suppose the edge e_1, e_2, e_3, ... e_i Then select an edge $e_i + 1 = (V_j, V_k)$ such that
   (*a*) $V_j \in \{v_1, v_2, v_3, ... v_i, v_i + 1\}$ and
   (*b*) $V_k \notin \{v_1, v_2, v_3, ... v_i, v_i + 1\}$ such that ei+1 has smallest weight among the edge of G
5. Repeat the step 4 until $(n - 1)$ edges have been chosen
6. Exit

PROGRAM 9.5

```
//PROGRAM TO CREATE MINIMUM SPANNING TREE
//USING PRIM'S ALGORITHM
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
#include<process.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;

void 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()*/

void 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()*/
```

```
/*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()*/

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()*/
void main()
{
    int i, j;
    int path[MAX];
    int wt_tree,count;
    struct edge tree[MAX];
    clrscr();
    create_graph();
    printf("\nAdjacency matrix is:\n");
    display();

    count = maketree(tree,&wt_tree);
```

```
        printf("\nWeight of spanning tree is:%d\n", wt_tree);
        printf("\nEdges 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);
        }
        getch();
}/*End of main()*/
```

9.6.3. SOLLIN'S ALGORITHM

Sollin's Algorithm construct the minimum cost spanning tree by selecting several edge at each stage. Select few edges of lowest weight, to form a spanning forest. If more than one edge exists with minimum cost, select all the edges. Next stage, we select one edge of minimum weight for each tree in this forest. It is possible for two trees in the forest to select the same edge. So multiple copies of the same edge is eliminated. Also, when the graph has several edges with the same cost, it is possible to select two different edges that connect them together. Repeat the steps until there is only one three or when no edges remain to be selected.

Construction of the minimum-cost spanning tree using Prim's algorithm can be explained with an example. Consider a graph G in Fig. 9.21. Following figures shows different stages in Sollin's Algorithm.

The minimum cost edge in the graph G in Fig. 9.21 is 1. If you analyze closely there are two edges (*i.e.*, (7, 3), (4, 9)) with the minimum cost 1. As the algorithm says select all minimum cost edges as the edges in the minimum-cost spanning tree as shown in Fig. 9.39.

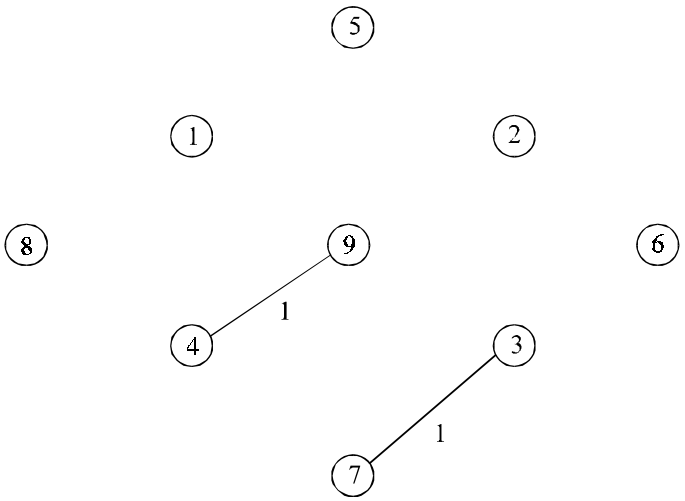


Fig. 9.39

Next select minimum weight edge incident to recently selected edge(*s*). Here (4,9), (7, 3) are the recently selected edges. Minimum cost edges incident to this edges are (9, 1) and (3, 6) respectively. As the algorithm says select these minimum cost edges as the edges in the minimum-cost spanning tree as shown in Fig. 9.40. And repeat the same process.

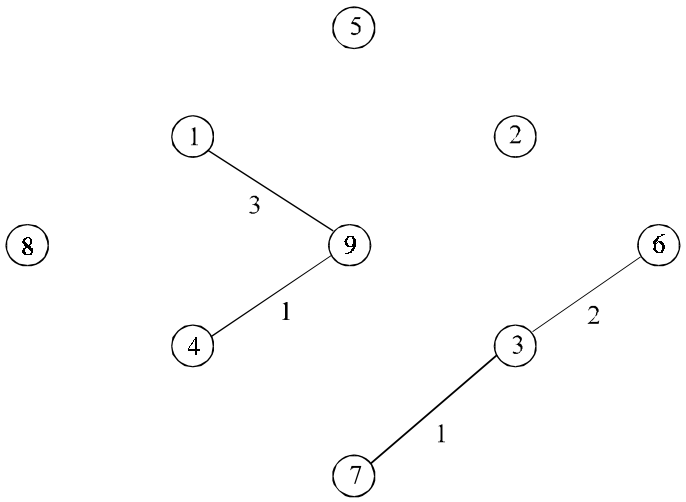


Fig. 9.40

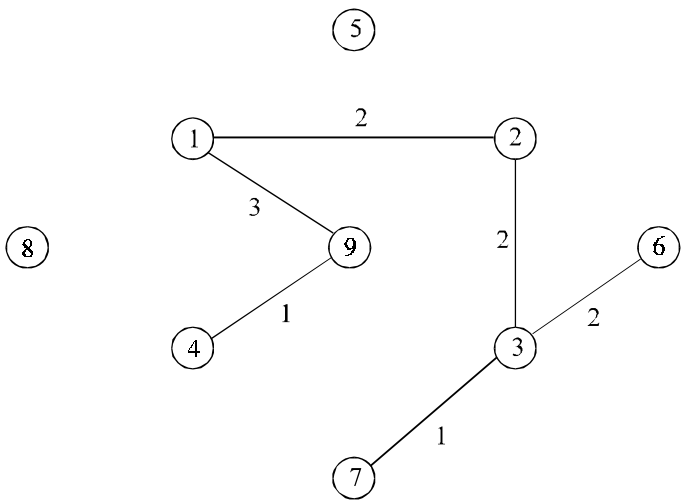


Fig. 9.41

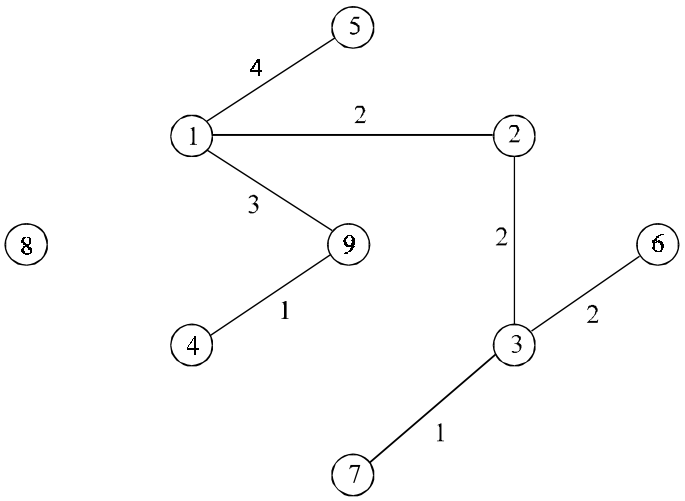


Fig. 9.42

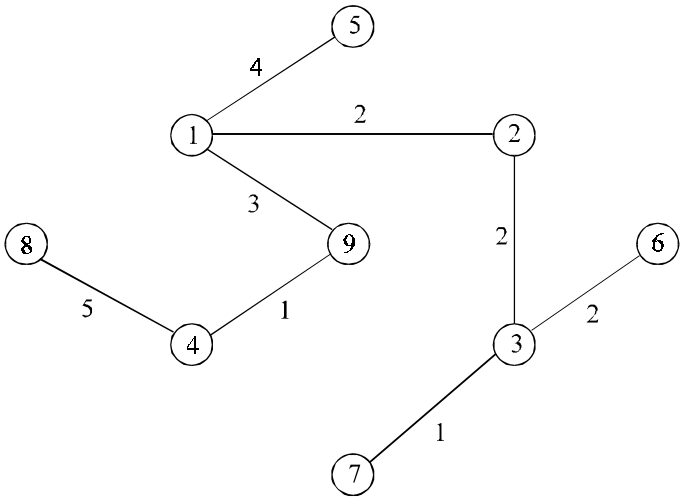


Fig. 9.43

9.7. SHORTEST PATH

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

9.7.1. DIJKSTRA'S ALGORITHM

Let G be a directed graph with n vertices $V_1, V_2, V_3 \ldots\ldots V_n$. Suppose $G = (V, E, W_e)$ is weighted graph. *i.e.*, each edge e in G is assigned a non- negative number, we called the weight or length of the edge e. Consider a starting vertices. Dijstra's algorithm will find the weight or length to each vertex from the source vertex.

ALGORITHM

Set $V = \{V_1, V_2, V_3 \ldots\ldots V_n\}$ contains the vertices and the edges $E = \{e_1, e_2, \ldots\ldots e_m\}$ of the graph G. W(e) is the weight of an edge e, which contains the vertices V_1 and V_2. Q is a set of vertices, which are not visited. m is the vertex in Q for which weight W(m) is minimum, *i.e.*, minimum cost edge. S is a source vertex.

1. Input the source vertices and assign it to S
   (*a*) Set W(s) = 0 and
   (*b*) Set W (v) = ___ for all vertices V is not equal to S
2. Set Q = V which is a set of vertices in the graph
3. Suppose m be a vertices in Q for which W(m) is minimum
4. Make the vertices m as visited and delete it from the set Q
5. Find the vertices I which are incident with m and member of Q (That is the vertices which are not visited)
6. Update the weight of vertices $I = \{i_1, i_2 \ldots\ldots i_k\}$ by
   (*a*) $W(i_1) = \min [W(i_1), W(m) + W(m, i_1)]$
7. If any changes is made in W(v), store the vertices to corresponding vertices i, using the array, for tracing the shortest path
8. Repeat the process from step 3 to 7 until the set Q is empty
9. Exit

The above algorithm is illustrated with a graph in Fig. 9.44

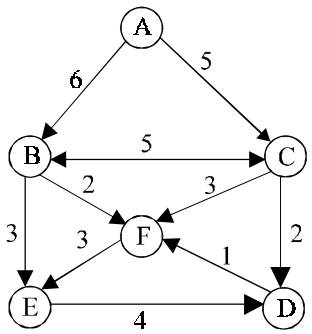


Fig. 9.44

Source vertices is = A W (A) = 0

V = {A, B, C, D, E, F) = Q

V	A	B	C	D	E	F
W(V)	0					
Q	A	B	C	D	E	F

A	
B	
C	
D	
E	
F	

ITERATION 1:

m = A

W(A, A) = 0 (Distance from A to A)

Now the Q = { B, C, D, E, F}

Two edges are incident with *m*

i.e., I = { B, C}

W(B) = min (W(B), W (A) + W (A, B)]

= Min (_, 0 + 6)

= 6

W(C) = min (w(c), W (A) + W (A,C)]

= min (–, 0 + 5) = 5

V	A	B	C	D	E	F
W(V)	0	6	5			
Q		B	C	D	E	F

A	
B	A
C	A
D	
E	
F	

ITERATION 2:

m = C (Because W(*v*) in minimum vertex and is also a member of Q)

Now the Q become (B, D, E, F)

Two edge are incident with C = {D, F} = I

W(D) = min (W(D), [W(C) + W(C, D)])

= min (_, [5+2]) =7

W(F) = min (W(F), [W(C) + W (C, F)])

= min (_, [5+3]) =8

V	A	B	C	D	E	F
W(V)	0	6	5	7		8
Q		B		D	E	F

A	
B	A
C	A
D	C
E	
F	C

ITERATION 3:

m = B (Because *w* (V) in minimum in vertices B and is also a member of Q)

Now the Q become (D, E, F)

Three edge are incident with B = { C, E, F}

Since C is not a member of Q so I = {E, F}

W(E) = min (_ , 6 + 3) =9

W(F) = min (8, 6 + 2) =8

V	A	B	C	D	E	F
W(V)	0	6	5	7	9	8
Q				D	E	F

A	
B	A
C	A
D	C
E	B
F	C

ITERATION 4:

m = D

Q = {E, F)

Incident vertices of D = { F } = I

W(F) = min (W(F) , [W(D) + W(D,F))

W(F) = min (8 , 7 + 1) =8

V	A	B	C	D	E	F
W(V)	0	6	5	7	9	8
Q					E	F

A	
B	A
C	A
D	C
E	B
F	C

ITERATION 5:

s = F

Q = { F)

Incident vertices of F = { E }

W(E) = min (W(F) , [W(E) + W(F,E))

W(E) = min (9 , 9 + 3) = 9

V	A	B	C	D	E	F
W(V)	0	6	5	7	9	8
Q					E	

A	
B	A
C	A
D	C
E	B
F	C

now E is the only chain, hence we stop the iteration and the final table is

V	A	B	C	D	E	F
W(V)	0	6	5	7	9	8

A	
B	A
C	A
D	C
E	B
F	C

If the source vertex is A and the destination vertex is D then the weight is 7 and the shortest path can be traced from table at the right side as follows.

Start finding the shortest path from destination vertex to its shortest vertex. For example we want to find the shortest path from A to D. Then find the shortest vertex from D, which is C. Check the shortest vertex, is equal to source vertex. Otherwise assign the shortest vertex as new destination vertex to find its shortest vertex as new destination vertex to find its shortest vertex. This process continued until we reach to source vertex from destination vertex.

D $\rightarrow$ C

C $\rightarrow$ A

A, C, D is the shortest path

The efficiency of the Dijskras's algorithm is analyzed by the iteration of the loop structures. The while loop iteration $n - 1$ times to visit the minimum weighted edge. Potentially loop must be repeated n times to examine every vertices in the graph. So the time complexity is $O(n^2)$.

PROGRAM 9.6

```
//PROGRAM OF SHORTEST PATH BETWEEN TWO NODE IN
//GRAPH USING DJIKSTRA ALGORITHM
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
#include<process.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 create_graph()
{
        int i,max_edges,origin,destin,wt;

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

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

void 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()*/

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

```
        printf("\nThe adjacency matrix is:\n");
        display();
        getch();

        while(1)
        {
                clrscr();
                printf("\nEnter source node(0 to quit):");
                scanf("%d",&source);
                printf("\nEnter 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("\nShortest distance is:%d\n", shortdist);
                        printf("\nShortest Path is:");
                        for(i=count;i>1;i—)
                                printf("%d–>",path[i]);
                        printf("%d",path[i]);
                        printf("\n");
                }
                else
                printf("\nThere is no path from source to destination node\n");
        }/*End of while*/
}/*End of main()*/
```

SELF REVIEW QUESTIONS

1. Define a graph. Explain depth first search of traversing ? [*MG - MAY 2004* (*BTech*)]
2. Write an algorithm for the depth first search of a graph? State its advantages and disadvantages? [*MG - MAY 2004* (*BTech*), *MG - NOV 2004* (*BTech*), *MG - MAY 2003* (*BTech*)]
3. Distinguish between adjacency matrix and adjacency list? [*MG - NOV 2004* (*BTech*)]
4. Explain the method of representing graphs by using matrices? [*MG - NOV 2002* (*BTech*)]
5. Explain the use of graph in data structures? [*MG - MAY 2000* (*BTech*)]
6. Explain the two methods of graph traversing? [*MG - MAY 2000* (*BTech*)]

7. Write an algorithm to find the shortest path between any two nodes of a given graph. Illustrate with an example. [*Calicut - APR 1995* (*BTech*), *CUSAT - JUL 2002* (*MCA*)]
8. Give the various representations of a graph.
   [*ANNA - DEC 2004* (*BE*), *Calicut - APR 1997* (*BTech*)
   ANNA - MAY 2004 (*MCA*)]
9. What is a spanning tree? Present algorithms to obtain the spanning trees for a graph. Illustrate them with examples. [*ANNA - MAY 2004* (*BE*), *Calicut - APR 1997* (*BTech*)]
10. Discuss the implementation of dfs and bfs graph traversals with suitable example.
    [*ANNA - DEC 2004* (*BE*), *CUSAT - NOV 2002* (*BTech*)
    KERALA - MAY 2002 (*BTech*), *ANNA - DEC 2004* (*BE*)]
11. Explain the Dijkstra's algorithm for shortest path in a graph with suitable example.
    [*CUSAT - NOV 2002* (*BTech*)]
12. Explain the following:
    (*a*) Graph (*b*) Multigraph (*c*) Digraph (*d*) Spanning tree.
    (*e*) Give anyone representations for a graph structure.
    (*f*) Explain what is a minimum spanning tree.
    (*g*) Explain Kruskal's algorithm. [*CUSAT - JUL 2002* (*MCA*)]
13. Explain the Prim's algorithm to find minimal spanning tree for a graph.
    [*ANNA - MAY 2004* (*BE*)]
14. Explain various application of the graph
    [*KERALA - JUN 2004* (*BTech*), *KERALA - DEC 2004* (*BTech*) ;]
    KERALA - DEC 2003 (*BTech*)]
15. Distinguish between DFS and BFS
    [*KERALA - DEC 2002* (*BTech*), *KERALA - DEC 2004* (*BTech*)
16. Explain complete Graph. *KERALA - DEC 2004* (*BTech*)]
17. What are graphs? Give various representation of graphs?
    [*KERALA - MAY 2003* (*BTech*)]
18. Define Directed graph and Undirected graph. [*KERALA - DEC 2002* (*BTech*)]
19. What is the time required to determine the total number of edges in G ?
    [*KERALA - DEC 2002* (*BTech*)]
20. Explain the various procedure for finding the shortest path in a network.
    [*KERALA - DEC 2002* (*BTech*)]
21. What are graphs? Explain the applications of graphs. [*KERALA - MAY 2001* (*BTech*)]