## 7.14 Minimum spanning tree

The sum of weights of edges of different spanning trees of a graph may be different. A spanning tree of graph G whose sum of weights is minimum amongst all spanning trees of G, is called the Minimum Spanning Tree of G. Let us take a graph and draw its different spanning trees.

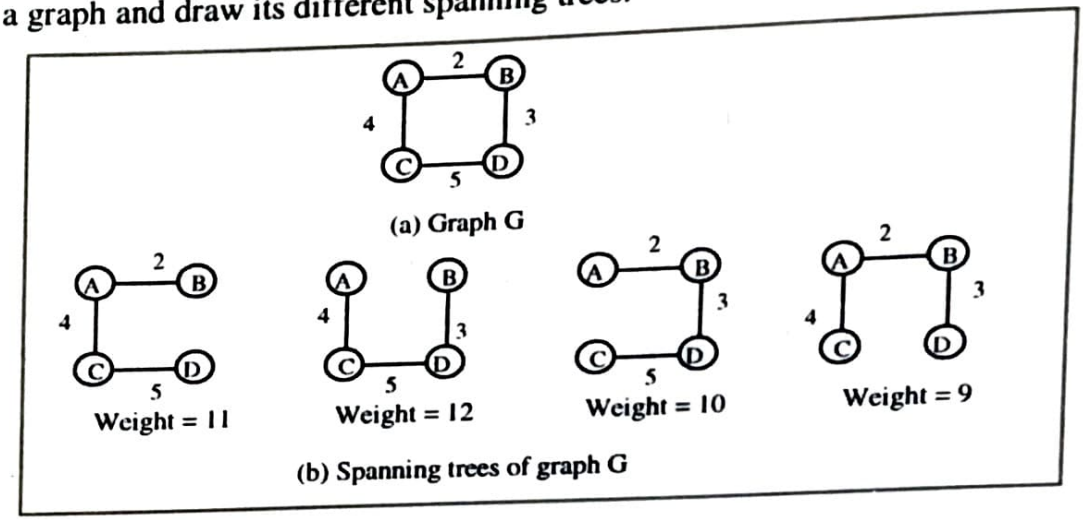



**Figure 7.61**

Here the tree with weight 9 is the minimum spanning tree. It is not necessary that a graph has unique minimum spanning tree. If there are duplicate weights in the graph then more than one spanning trees are possible, but if the all weights are unique then there will be only one minimum spanning tree.

Minimum spanning tree gives us the most economical way of connecting all the vertices in a graph. For example, in a network of computers we can connect all the computers with the least cost if we construct a minimum spanning tree for the graph where the vertices are computers. Similarly in a telephone communication network we can connect all the cities in the network with the least possible cost.

There are many ways for creating minimum spanning tree but the most famous methods are Prim's and Kruskal's algorithm. Both these methods use the greedy approach.

### 7.14.1 Prim's Algorithm

In this algorithm we start with an arbitrary vertex as the root and at each step a vertex is added to the tree till all the vertices are in the tree.

The method of making minimum spanning tree from Prim's algorithm is like Dijkstra's algorithm for shortest paths. Each vertex is given a status, which can be permanent or temporary. Initially all the vertices are temporary and at each step of the algorithm, a temporary vertex is made permanent. The process stops when all the vertices are made permanent. Making a vertex permanent means that it has been included in the tree. The temporary vertices are those vertices which have not been added to the tree.

We label each vertex with length and predecessor. The label length represents the weight of the shortest edge connecting the vertex to a permanent vertex and predecessor represents that permanent vertex. Once a vertex is made permanent, it is not relabeled. Only temporary vertices will be relabeled if required.

Applying the greedy approach, the temporary vertex that has the minimum value of length is made permanent. In other words we can say that the temporary vertex which is adjacent to a permanent vertex by an edge of least weight is added to the tree.

The steps for making a minimum spanning tree by Prim's algorithm are as-

(A) Initialize the length of all vertices to infinity and predecessors of all vertices to NIL. Make the status of all vertices temporary.

(B) Select any arbitrary vertex as the root vertex and make its length label equal to 0.

(C) From all the temporary vertices in the graph, find out the vertex that has smallest value of length, make it permanent and now this is our current vertex. (If there are many with the same value of length then anyone can be selected)

(D) Examine all the temporary vertices adjacent to the current vertex. Suppose `current` is the current vertex and v is a temporary vertex adjacent to `current`.

(i) If weight (`current`,v) < length(v)
Relabel the vertex v
Now length(v) = weight(`current`,v)
predecessor(v) = `current`

(ii) If weight (`current`,v) >= length(v)
Vertex v is not relabelled

(E) Repeat steps (C) and (D) till there are no temporary vertices left, or all the temporary vertices left have length equal to infinity. If the graph is connected, then the procedure will stop when all n vertices have been made permanent and n-1 edges are added to the spanning tree. If the graph is not connected, then those vertices that are not reachable from the root vertex will remain temporary with length infinity. In this case no spanning tree is possible.

Let us take an undirected connected graph and construct the minimum spanning tree

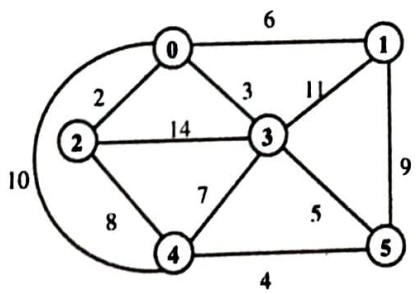
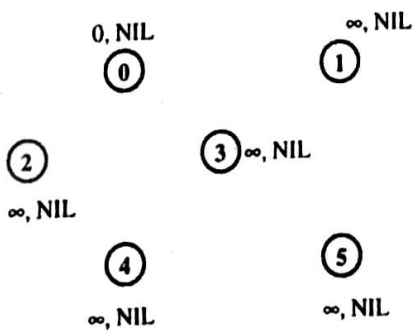



**Figure 7.62**

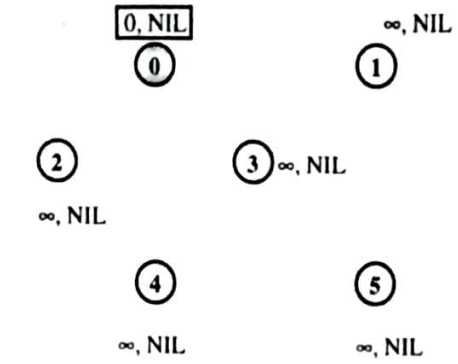
Initially length values for all the vertices are set to a very large number(larger than weight of any edge). Suppose ∞ is such a number. We have taken the predecessor of all vertices NIL(-1) in the beginning.

Initially all the vertices are temporary. We select the vertex 0 as the root vertex and make its length label equal to zero.



| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Temp |
| 1 | ∞ | NIL | Temp |
| 2 | ∞ | NIL | Temp |
| 3 | ∞ | NIL | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |

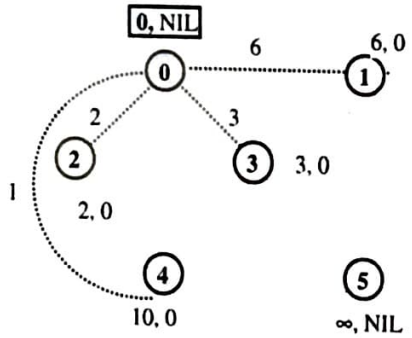
From all the temporary vertices, vertex 0 has the smallest length, so it will be made permanent. This is the first vertex to be included in the tree. Its predecessor will remain NIL. Now vertex 0 is the current vertex.




| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | ∞ | NIL | Temp |
| 2 | ∞ | NIL | Temp |
| 3 | ∞ | NIL | Temp |
| 4 | ∞ | NIL | Temp |
| 5 | ∞ | NIL | Temp |

Vertices 1, 2, 3, 4 are temporary vertices adjacent to 0

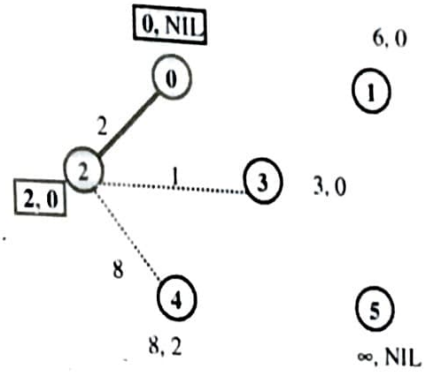
weight(0,1) < length(1) 6 < ∞ Relabel 1
predecessor[1]=0 , length[1]=6
weight(0,2) < length(2) 2 < ∞ Relabel 2
predecessor[2]=0 , length[2]=2
weight(0,3) < length(3) 3 < ∞ Relabel 3
predecessor[3]=0 , length[3]=3
weight(0,4) < length(4) 10 < ∞ Relabel 4
predecessor[4]= 0, length[4]=10




| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Temp |
| 2 | 2 | 0 | Temp |
| 3 | 3 | 0 | Temp |
| 4 | 10 | 0 | Temp |
| 5 | ∞ | NIL | Temp |

From all the temporary vertices, vertex 2 has the smallest length so make it permanent i.e. include it in the tree. Its predecessor is 0, so the edge that is added to the tree is (0,2). Now vertex 2 is the current vertex. Vertices 3, 4 are temporary vertices adjacent to vertex 2.

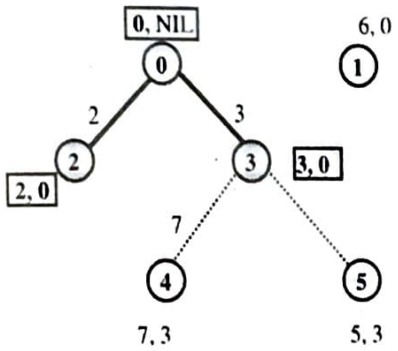
weight(2,3) > length(3) 14 > 3 Don't Relabel 3
weight(2,4) < length(4) 8 < 10 Relabel 4
predecessor[4]=2 , length[4]=8




| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 3 | 0 | Temp |
| 4 | 8 | 2 | Temp |
| 5 | ∞ | NIL | Temp |

From all temporary vertices, vertex 3 has the smallest value of length so make it permanent. Its predecessor is 0, so the edge (0,3) is included in the tree. Now vertex 3 is the current working vertex. Vertices 1, 4, 5 are temporary vertices adjacent to vertex 3

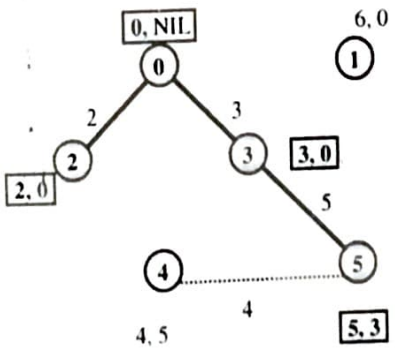
weight(3,1) > length(1) 11>6 Don't relabel 1
weight(3,4) < length(4) 7 < 8 Relabel 4
predecessor[4]=3 , length[4]=7
weight(3,5) < length(5) 5 < ∞ Relabel 5
predecessor[5]=3 , length[5]=5




| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 3 | 0 | Perm |
| 4 | 7 | 3 | Temp |
| 5 | 5 | 3 | Temp |

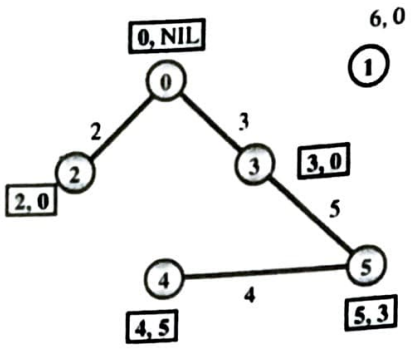
From all temporary vertices, vertex 5 has the smallest length so make it permanent. Its predecessor is 3 so include edge (3,5) in the tree. Now vertex 5 is the current vertex. Vertices 1, 4 are temporary vertices adjacent to vertex 5

weight(5,1) > length(1) 9 > 6 Don't relabel 1
weight(5,4) < length(4) 4 < 7 Relabel 4
predecessor[4]=5 , length[4]=4



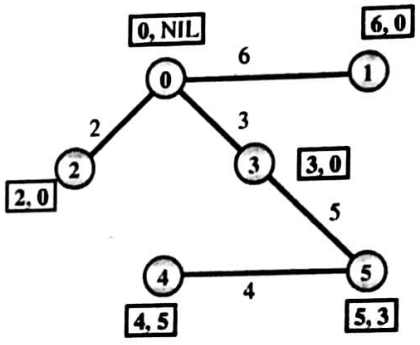
| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 3 | 0 | Perm |
| 4 | 4 | 5 | Temp |
| 5 | 5 | 3 | Perm |

From all temporary vertices, vertex 4 has the smallest length so make it permanent. Its predecessor is 5, so include edge (5,4) in the tree. Now vertex 4 is the current vertex. There are no temporary vertices adjacent to 4




| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Temp |
| 2 | 2 | 0 | Perm |
| 3 | 3 | 0 | Perm |
| 4 | 4 | 5 | Perm |
| 5 | 5 | 3 | Perm |

Vertex 1 is the only temporary vertex left and its length is 6, so make it permanent. Its predecessor is 0, so edge (0,1) is included in the tree.




Now all the vertices are permanent so we stop our procedure.

| Vertex | length | Prede-cessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Perm |
| 2 | 2 | 0 | Perm |
| 3 | 3 | 0 | Perm |
| 4 | 4 | 5 | Perm |
| 5 | 5 | 3 | Perm |

Now we have a complete minimum spanning tree. The edges that belong to minimum spanning tree are - (0,1), (0,2), (0,3), (5,4), (3,5)

Weight of minimum spanning tree will be-

$6 + 2 + 3 + 4 + 5 = 20$

Now let us take a graph that is not connected.

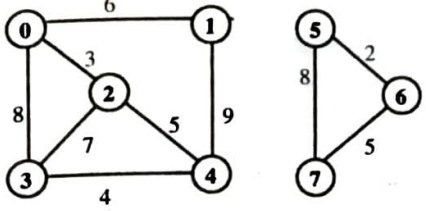



**Figure 7.63**

After making vertices 0,2,4,3,1 permanent, the situation would be-

| Vertex | length | Predecessor | Status |
|---|---|---|---|
| 0 | 0 | NIL | Perm |
| 1 | 6 | 0 | Perm |
| 2 | 3 | 0 | Perm |
| 3 | 4 | 4 | Perm |
| 4 | 5 | 2 | Perm |
| 5 | ∞ | NIL | Temp |
| 6 | ∞ | NIL | Temp |
| 7 | ∞ | NIL | Temp |

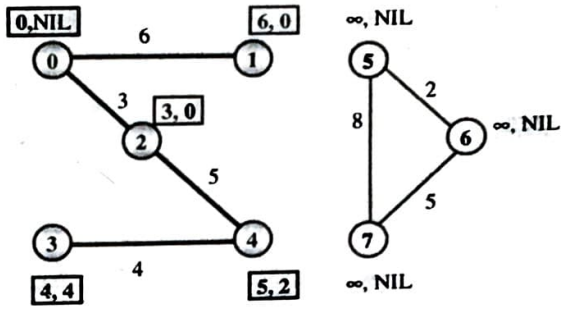



**Figure 7.64**

Vertices 5, 6, 7 are temporary with length infinity so we stop the procedure and state that the graph is not connected and hence no spanning tree is possible.

```
/*P7.12 Program for creating minimum spanning tree using Prim's algorithm*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
#define TEMP 0
#define PERM 1
#define infinity 9999
#define NIL -1
struct edge
{
      int u;
      int v;
};
int n;
int adj[MAX][MAX];
int predecessor[MAX];
int status[MAX];
int length[MAX];
void create_graph();
void maketree(int r, struct edge tree[MAX]);
int min_temp();
main()
{
      int wt_tree = 0;
      int i, root;
      struct edge tree[MAX];
      create_graph();
      printf("Enter root vertex : ");
      scanf("%d",&root);
      maketree(root, tree);
      printf("Edges to be included in spanning tree are : \n");
      for(i=1; i<=n-1; i++)
      {
            printf("%d->",tree[i].u);
            printf("%d\n",tree[i].v);
            wt_tree += adj[tree[i].u][tree[i].v];
```

```
        }
        printf("Weight of spanning tree is : %d\n", wt_tree);
}/*End of main()*/
void maketree(int r, struct edge tree[MAX])
{
        int current,i;
        int count = 0;/*number of vertices in the tree*/
        for(i=0; i<n; i++)      /*Initialize all vertices*/
        {
                predecessor[i] = NIL;
                length[i] = infinity;
                status[i] = TEMP;
        }
        length[r] = 0; /*Make length of root vertex 0*/
        while(1)
        {
                /*Search for temporary vertex with minimum length
                and  make it current vertex*/
                current = min_temp();
                if(current==NIL)
                {
                  if(count==n-1) /*No temporary vertex left*/
                      return;
                  else /*Temporary vertices left with length infinity*/
                  {
                    printf("Graph is not connected, No spanning tree possible\n");
                    exit(1);
                  }
                }
                status[current] = PERM; /*Make the current vertex permanent*/
                /*Insert the edge (predecessor[current], current) into the tree,
                except when the current vertex is root*/
                if(current!=r)
                {
                   count++;
                   tree[count].u = predecessor[current];
                   tree[count].v = current;
                }
                for(i=0; i<n; i++)
                        if(adj[current][i]>0 && status[i]==TEMP)
                                if(adj[current][i] < length[i])
                                {
                                        predecessor[i] = current;
                                        length[i] = adj[current][i];
                                }
        }
}/*End of make_tree()*/
/*Returns the temporary vertex with minimum value of length, Returns NIL if no temporary
vertex left or all temporary vertices left have pathLength infinity*/
int min_temp()
{
        int i;
        int min = infinity;
        int k = -1;
        for(i=0; i<n; i++)
        {
                if(status[i]==TEMP && length[i]<min)
                {
                        min = length[i];
                        k = i;
                }
        }
        return k;
}/*End of min_temp()*/
```

```
void create_graph()
{
    int i,max_edges,origin,destin,wt;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edges = n*(n-1)/2; /*Undirected graph*/
    for(i=1; i<=max_edges; i++)
    {
        printf("Enter edge %d(-1 -1 to quit) : ",i);
        scanf("%d %d",&origin,&destin);
        if((origin==-1) && (destin==-1))
            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 create_graph()*/
```

## 7.14.2 Kruskal's Algorithm

This approach of constructing a minimum spanning tree was formulated by J.B.Kruskal and is named after him as Kruskal's algorithm.

In this algorithm, initially we take a forest of n distinct trees for all n vertices of the graph. So at the start of algorithm, each tree is a single vertex tree and no edges are there. In each step of the algorithm an edge is examined and it is included in the spanning tree if its inclusion does not form a cycle. If this edge forms a cycle then it is rejected.

Now the question arises as to which edge should be considered for examining. Here we apply the greedy approach and use the edge with the minimum weight. For this we can maintain a list of edges sorted in ascending order of their weights. At each step we take an edge from this list and include it if it does not form a cycle. If the edge forms a cycle, we reject it. The algorithm completes when n-1 edges have been included. If the graph contains less than n-1 edges, then it means that graph is not connected and no spanning tree is possible.

Initially we have a forest of n trees, whenever an edge is included two distinct trees are joined into a single tree. So at any stage of this algorithm we don't have a single tree as in Prim's, but we have a forest of trees. Whenever an edge is inserted, the number of trees in the forest decreases by one, and at the end when n-1 edges have been included we are left with only one tree which is our minimum spanning tree.Let us take a graph and create a minimum spanning tree by Kruskal's algorithm.

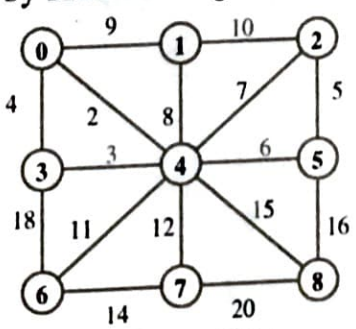



**Figure 7.65**

Initially we take a forest of 9 trees, with each tree consisting of a single vertex. All the edges are examined in increasing order of their weight.

| | |
|---|---|
| Edge 0-4, wt = 2 | Inserted, see figure 7.66(b) |
| Edge 3-4, wt = 3 | Inserted, see figure 7.66(c) |
| Edge 0-3, wt = 4 | Not inserted, forms cycle 0-3-4-0 in figure 7.66 (c) |
| Edge 2-5, wt = 5 | Inserted, see figure 7.66(d) |
| Edge 4-5, wt = 6 | Inserted, see figure 7.66(e) |
| Edge 2-4, wt = 7 | Not inserted, forms cycle 2-4-5-2 in figure 7.66(e) |
| Edge 1-4, wt = 8 | Inserted, see figure 7.66(f) |
| Edge 0-1, wt = 9 | Not inserted, forms cycle 0-1-4-0 in figure 7.66(f) |
| Edge 1-2, wt = 10 | Not inserted, forms cycle 1-2-5-4-1 in figure 7.66(f) |
| Edge 4-6, wt = 11 | Inserted, see figure 7.66(g) |
| Edge 4-7, wt = 12 | Inserted, see figure 7.66(h) |
| Edge 6-7, wt = 14 | Not inserted, forms cycle 4-6-7-4 in figure 7.66(h) |
| Edge 4-8, wt = 15 | Inserted, see figure 7.66(i) |

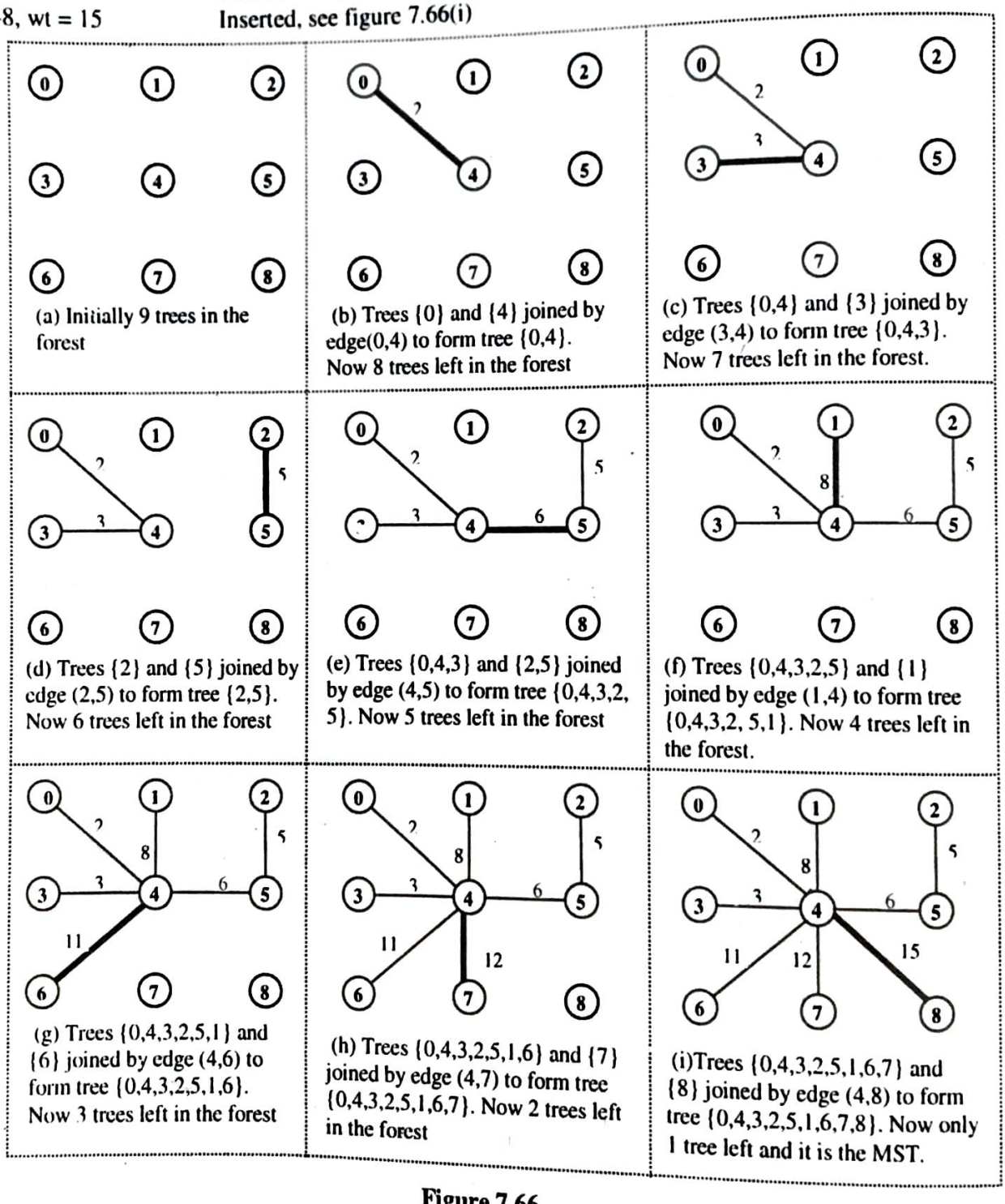



**Figure 7.66**

The resulting minimum spanning tree is-

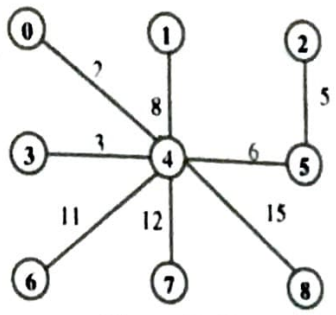



**Figure 7.67**

Let us see how we can implement this algorithm. We examine all the edges one by one starting from the smallest edge. To decide whether the selected edge should be included in the spanning tree or not, we will examine the two vertices connecting the edge. If the two vertices belong to the same tree, it means that they are already connected and adding this edge would result in a cycle. So we will insert an edge in the spanning tree only if its vertices are in different trees.

Now the question is how to decide whether two vertices are in the same tree or not. We will keep record of the father of every vertex. Since this is a tree, every vertex will have only one distinct father. We will recognize each tree by a root vertex and a vertex will be a root if its father is NIL(-1). Initially we have only single vertex trees; each vertex is a root vertex so we will take father of all vertices as NIL. For finding out, to which tree a vertex belongs, we will find out the root of that tree. So we will traverse all the ancestors of the vertex till we reach a vertex whose father is NIL. This will be the root of the tree to which the vertex belongs.

Now we know the root of both vertices of an edge, if roots are same means both vertices are in the same tree and are already connected so this edge is rejected. If the roots are different, then we will insert this edge into the spanning tree and we will join the two trees, which are having these two vertices. 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 vertices of both trees will be connected and have the same root.Initially father of every vertex is NIL(-1), and hence every vertex is a root vertex.

vertex 0 1 2 3 4 5 6 7 8
father N N N N N N N N N

| Edge | Wt | v1 | v2 | root of v1 | root of v2 | Result | |
|---|---|---|---|---|---|---|---|
| 0-4 | 2 | 0 | 4 | 0 | 4 | Inserted father[4]=0 | vertex 0 1 2 3 **4** 5 6 7 8<br>father N N N N **0** N N N N |
| 3-4 | 3 | 3 | 4 | 3 | 0 | Inserted father[0]=3 | vertex **0** 1 2 3 4 5 6 7 8<br>father **3** N N N 0 N N N N |
| 0-3 | 4 | 0 | 3 | 3 | 3 | Not inserted | vertex 0 1 2 3 4 5 6 7 8<br>father 3 N N N 0 N N N N |
| 2-5 | 5 | 2 | 5 | 2 | 5 | Inserted father[5]=2 | vertex 0 1 2 3 4 **5** 6 7 8<br>father 3 N N N 0 **2** N N N |
| 4-5 | 6 | 4 | 5 | 3 | 2 | Inserted father[2]=3 | vertex 0 1 **2** 3 4 5 6 7 8<br>father 3 N **3** N 0 2 N N N |
| 2-4 | 7 | 2 | 4 | 3 | 3 | Not inserted | vertex 0 1 2 3 4 5 6 7 8<br>father 3 N 3 N 0 2 N N N |
| 1-4 | 8 | 1 | 4 | 1 | 3 | Inserted father[3]=1 | vertex 0 1 2 **3** 4 5 6 7 8<br>father 3 N 3 **1** 0 2 N N N |
| 0-1 | 9 | 0 | 1 | 1 | 1 | Not inserted | vertex 0 1 2 3 4 5 6 7 8<br>father 3 N 3 1 0 2 N N N |
| 1-2 | 10 | 1 | 2 | 1 | 1 | Not inserted | vertex 0 1 2 3 4 5 6 7 8<br>father 3 N 3 1 0 2 N N N |
| 4-6 | 11 | 4 | 6 | 1 | 6 | Inserted father[6]=1 | vertex 0 1 2 3 4 5 **6** 7 8<br>father 3 N 3 1 0 2 **1** N N |
| 4-7 | 12 | 4 | 7 | 1 | 7 | Inserted father[7]=1 | vertex 0 1 2 3 4 5 6 **7** 8<br>father 3 N 3 1 0 2 1 **1** N |
| 6-7 | 14 | 6 | 7 | 1 | 1 | Not inserted | vertex 0 1 2 3 4 5 6 7 8<br>father 3 N 3 1 0 2 1 1 N |
| 4-8 | 15 | 4 | 8 | 1 | 8 | Inserted father[8]=1 | vertex 0 1 2 3 4 5 6 7 **8**<br>father 3 N 3 1 0 2 1 1 **1** |