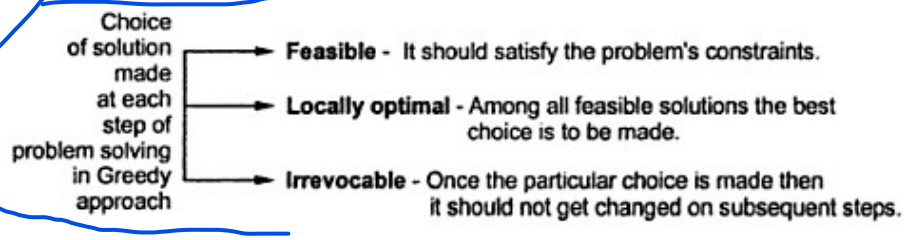
3

Greedy Method

3.1 Introduction

In an algorithmic strategy like Greedy, the decision of solution is taken based on the information available. The Greedy method is a straightforward method. This method is popular for obtaining the optimized solutions. In Greedy technique, the solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. At each step the choice made should be,



In short, while making a choice there should be a Greed for the optimum solution.

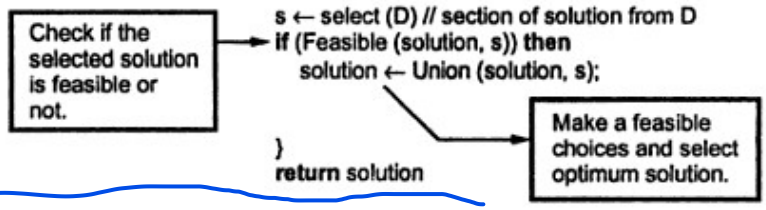
In this chapter we will first understand the concept of Greedy method. Then we will discuss various examples for which Greedy method is applied.

3.2 General Method

In this section we will understand *"What is Greedy method?"*.

Algorithm

```
Greedy (D, n)
  //In Greedy approach D is a domain
  //from which solution is to be obtained of size n
  //Initially assume
        Solution ← 0
        for i ← 1 to n do
                    {
```



In Greedy method following activities are performed.

1. First we select some solution from input domain.
2. Then we check whether the solution is feasible or not.
3. From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function. Such a solution is called optimal solution.
4. As Greedy method works in stages. At each stage only one input is considered at each time. Based on this input it is decided whether particular input gives the optimal solution or not.

3.2.1 Applications of Greedy method

In this section we will discuss various problems that can be solved Greedy approach -

i) Knapsack problem
ii) Prim's algorithm for minimum spanning tree.
iii) Kruskal's algorith for minimum spanning tree.
iv) Finding shortest path.
v) Job sequencing with deadlines.
vi) Optimal storage on tapes.

For solving all above problems a set of feasible solutions is obtained. From these solution optimum solution is selected. This optimum solution then becomes the final solution for given problem.

With these fundamental issues of Greedy method, let us now discuss various applications of Greedy algorithm.

3.3 Knapsack Problem

The Knapsack problem can be stated as follows.

Suppose there are n objects from i = 1, 2,...n. Each object i has some positive weight w_i and some profit value is associated with each object which is denoted as p_i. This Knapsack carry at the most weight W.

While solving above mentioned Knapsack problem we have the capacity constraint. When we try to solve this problem using Greedy approach our goal is,

1. Choose only those objects that give maximum profit.
2. The total weight of selected objects should be ≤ W.

And then we can obtain the set of feasible solutions. In other words,

$$\text{maximized} \sum_{n}^{1} p_i x_i \text{ subject to } \sum_{n}^{1} w_i x_i \leq W$$

Where the Knapsack can carry the fraction x_i of an object i such that $0 \leq x_i \leq 1$ and $1 \leq i \leq n$.

Example 3.1 : *Consider that there are three items. Weight and profit value of each item is as given below,*

i	W_I	P_I
1	18	30
2	15	21
3	10	18

Also W=20. Obtain the solution for the above given Knapsack problem.

Solution : The feasible solutions are as given below.

X_1	X_2	X_3
1/2	1/3	1/4
1	2/15	0
0	2/3	1
0	1	1/2

Let us compute $\sum W_i X_i$

1. 1/2 *18 + 1/3 *15 + ¼ *10

   = 16.5

2. 1*18 + 2/15 *15 + 0*8
   = 20
3. 0*18 +2/3*15 +10
   = 20
4. 0*18 +1*15 +1/2*10
   = 20

Let us compute $\sum P_iX_i$

1. 1/2 *30 + 1/3 *21 + ¼ *18
   = 26.5
2. 1*30 + 2/15 *21 + 0*18
   = 32.8
3. 0*30 +2/3*21 +18
   = 32
4. 0*30 +1*21 +1/2*18
   = 30

To summarize this

$\sum W_iX_i$	$\sum P_iX_i$
16.5	26.5
20	32.8
20	32
20	30

The solution 2 gives the maximum profit and hence it turns out to be optimum solution.

3.3.1 Algorithm

The algorithm for solving Knapsack problem with Greedy approach is as given below.

```
Algorithm Knapsack_Greedy(W,n)
{
//p[i] contains the profits of i items such that 1 ≤ i ≤ n
//w[i] contains weights of I items.
//x[i] is the solution vector.
//W is the total size of Knapsack.
```

```
    for i : =1 to n do
{
if[w[i]<W) then//capacity of knapsack is a constraint
{
 x[i] : =1.0;
 W=W – w[i];
}
}
if(i<=n) then x[i] : =W/w[i];
}
```

This is the basic operation of selecting x[i] lies within for loop.
∴ Time complexity = Θ(n).

3.4 Minimum Cost Spanning Tree

Spanning Tree

A spanning tree of a graph G is a subgraph which is basically a tree and it contains **all the vertices** of G containing **no circuit.**

Minimum Spanning Tree

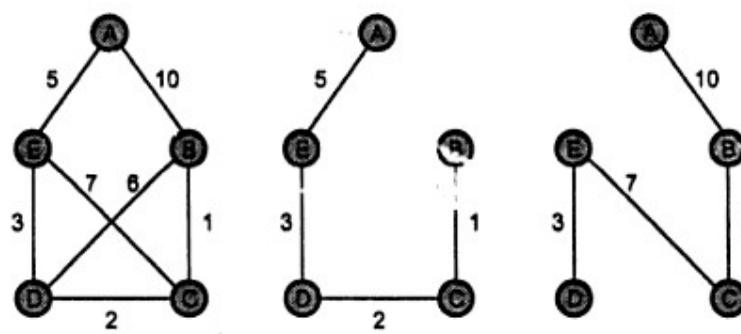
A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.

Weight of the Tree

A weight of the tree is defined as the sum of weights of all its edges.

For example :

Consider a graph G as given below. This graph is called weighted connected graph because some weights are given along every edge and the graph is a connected graph.



Graph G (a) (b) (c)

Fig. 3.1 Graph and two spanning trees out of which (b) is a minimum spanning tree

Applications of spanning trees :

1. Spanning trees are very important in designing efficient routing algorithms.
2. Spanning trees have wide applications in many areas such as network design.

3.4.1. Prim's Algorithm

Let us understand the prim's algorithm with the help of example.

Example : Consider the graph given below :

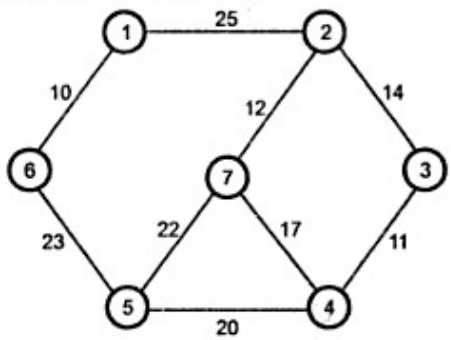
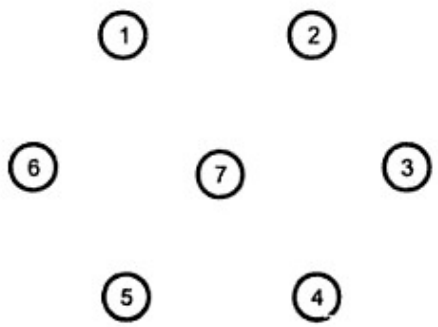


Fig. 3.2 Graph

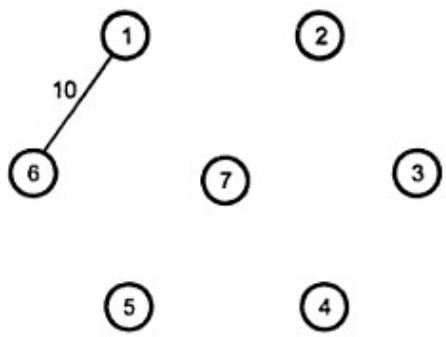
Now, we will consider all the vertices first. Then we will select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight. Care should be taken for not forming circuit.

Step 1 :



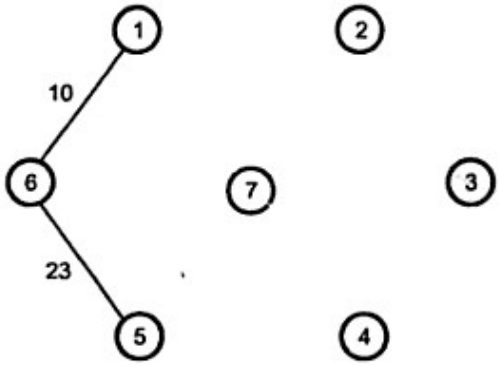
Total weight = 0

Step 2 :



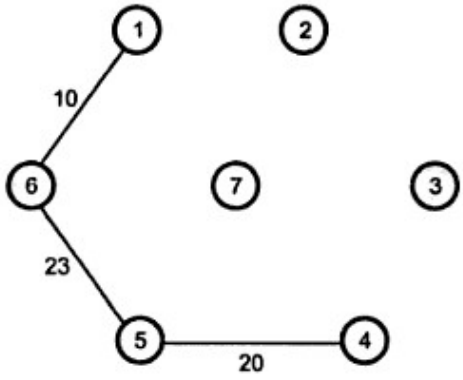
Total weight = 10

Step 3 :



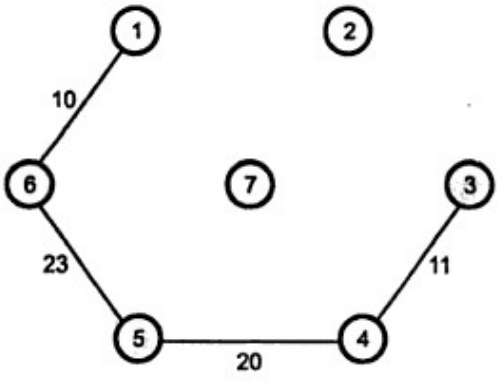
Total weight = 33

Step 4 :



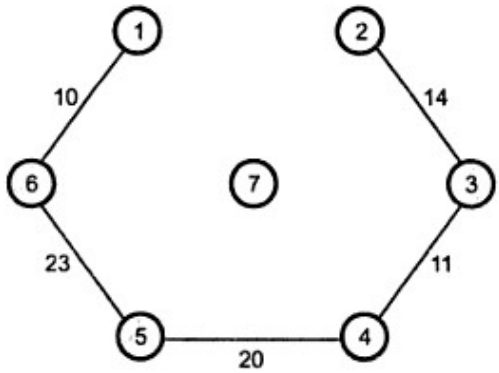
Total weight = 53

Step 5 :



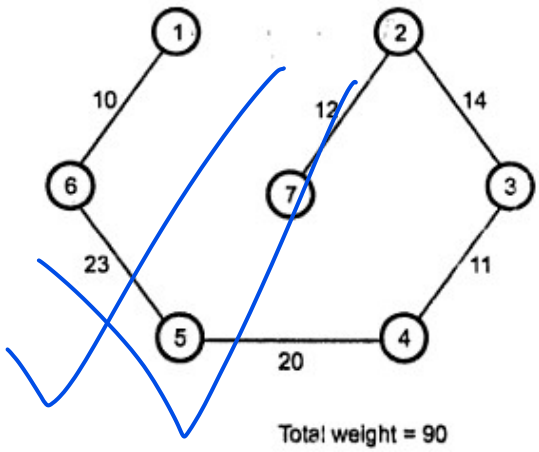
Total weight = 64

Step 6 :



Total weight = 78

Step 7 :



Total weight = 90

Prim's Algorithm

Algorithm

```
Prim(G[0...Size-1,0...Size-1],nodes)
//Problem Description : This algorithm is for implementing
//Prim's algorithm for finding spanning tree
//Input : Weighted Graph G and total number of nodes
//Output : Spanning tree gets printed with total path length
    total=0;
   //Initialize the selected vertices list
      for i←0 to nodes–1 do
        tree[i] ← 0
        tree[0] = 1;//take initial vertex
```

```
for k←1 to nodes do
{
   min_dist ← ∞
//initially assign minimum dist as infinity
for i←0 to nodes-1 do
{
      for j←0 to nodes-1 do
   {
      if (G [i,j]AND ((tree[i]AND !tree[j])OR
       (!tree[i] AND tree[j])))then
       {
         if(G[i,j] < min_dist)then
            {
            min_dist←G[ij]
            v1←i
            v2←j
          }
       }
   }
}
```

Select an edge such that one vertex is selected and other is not and the edge has the least weight.

Obtained edge with minimum wt.

Picking up those vertices yielding minimum edge.

```
Write(v1,v2,min_dist);
    tree[v1] ← tree[v2] ← 1
  total ← total+min_dist
  }
  Write("Total Path Length Is",total)
```

C function

```
void Prim(int G[][SIZE], int nodes)
{
      int tree[SIZE], i, j, k;
      int min_dist, v1, v2,total=0;
      //Initialize the selected vertices list
      for (i=0 ; i<nodes ; i++)
            tree[i] = 0;
      printf("\n\n The Minimal Spanning Tree Is : \n");
      tree[0] = 1;
      for (k=1 ; k<=nodes-1 ; k++)
      {
          min_dist = INFINITY;
```

```
        //initially assign minimum dist as infinity
          for (i=0 ; i<=nodes-1 ; i++)
                {
                   for (j=0 ; j<=nodes-1 ; j++)
                {
                   if (G[i][j] && ((tree[i] && !tree[j])||
                      (!tree[i] && tree[j])))
                   {
                      if (G[i][j] < min_dist)
                       {
                        min_dist = G[i][j];
                        v1 = i;
                        v2 = j;
                       }
                }
             }
          }

          printf("\n Edge (%d %d )and weight = %d",v1,v2,min_dist);
             tree[v1] = tree[v2] = 1;
          total =total+min_dist;
          }
          printf("\n\n\t Total Path Length Is = %d",total);
    }
```

Analysis

The algorithm spends most of the time in selecting the edge with minimum length. Hence the **basic operation** of this algorithm is to find the edge with **minimum path length.** This can be given by following formula.

$$T(n) = \sum_{k=1}^{nodes-1} \left(\sum_{i=0}^{nodes-1} 1 + \sum_{j=0}^{nodes-1} 1 \right)$$

Time taken by for k=1 to nodes–1 loop

Time taken by for i=0 to nodes–1 loop

Time taken by for j=0 to nodes–1 loop

We take variable n for 'nodes' for the sake of simplicity of solving the equation then

$$T(n) = \sum_{k=1}^{n-1} \left(\sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1 \right)$$

$$= \sum_{k=1}^{n-1} [((n-1) + 0 + 1) + ((n-1) + 0 + 1)]$$

$$= \sum_{k=1}^{n-1} (2n)$$

$$= 2n \sum_{k=1}^{n-1} 1$$

$$= 2n[(n-1) - 1 + 1] = 2n(n-1)$$

$$= 2n^2 - 2n$$

$$\therefore \quad T(n) \approx n^2$$

$$\therefore \quad T(n) = \Theta(n^2)$$

But n stands for total number of nodes or vertices in the tree. Hence we can also state

Time complexity of Prim's Algorithm is $\Theta(\|V^2\|)$.

But if th Prim's algorithm is implemented using binary heap with the creation of graph using adjacency list then its time complexity becomes $\Theta(E \log_2 V)$ where E stands for total number of edges and V stands for total number of vertices.

C Program

```
/*************************************************************************
This program is to implement Prim's Algorithm using Greedy Method
*************************************************************************/
  #include<stdio.h>
  #include<conio.h>
  #define SIZE 20
  #define INFINITY 32767

  /*This function finds the minimal spanning tree by Prim's Algorithm */
```

```
void Prim(int G[][SIZE],int nodes)
{
        int tree[SIZE],i,j,k;
        int min_dist,v1,v2,total=0;
    //Initialize the selected vertices list
    for (i=0; i<nodes; i++)
      tree[i] = 0;
    printf("\n\n The Minimal Spanning Tree Is :\n");
    tree[0] = 1;
    for (k=1; k<=nodes-1; k++)
    {


        min_dist = INFINITY;
   //initially assign minimum dist as infinity
          for (i=0; i<=nodes-1; i++)
          {
                    for (j=0; j<=nodes-1; j++)
             {
                 if (G[i][j]&&((tree[i]&&!tree[j])||
                              (!tree[i]&&tree[j])))
                {
                  if (G[i][j]<min_dist)
                    {
                    min_dist=G[i][j];
                    v1 = i;
                    v2 = j;
                  }
                }
          }
    }

       printf("\n Edge (%d %d )and weight = %d",v1,v2,min_dist);
           tree[v1] = tree[v2] = 1;
       total =total+min_dist;
        }
       printf("\n\n\t Total Path Length Is = %d",total);
}
```

```
void main()
{
        int G[SIZE][SIZE],nodes;
        int v1,v2,length,i,j,n;

        clrscr();
        printf("\n\t Prim'S Algorithm\n");

        printf("\n Enter Number of Nodes in The Graph");

        scanf("%d",&nodes);
        printf("\n Enter Number of Edges in The Graph");
        scanf("%d",&n);

        for (i=0 ; i<nodes ; i++)// Initialize the graph
            for (j=0 ; j<nodes ; j++)
                G[i][j] = 0;
        //entering weighted graph
        printf("\n Enter edges and weights \n");
        for (i=0 ; i<n; i++)
        {
            printf("\n Enter Edge by V1 and V2:");
            printf("[Read the graph from starting node 0]");
            scanf("%d %d",&v1,&v2);
            printf("\n Enter corresponding weight:");
            scanf("%d",&length);
            G[v1][v2] = G[v2][v1] = length;
        }
        getch();
        printf("\n\t");
        clrscr();
        Prim(G,nodes);
        getch();
}
```

Output

Graph will be

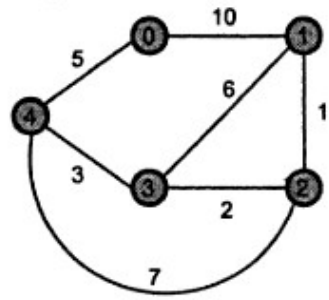


Fig. 3.3

Prim'S Algorithm

Enter Number of Nodes in The Graph 5

Enter Number of Edges in The Graph 7

Enter edges and weights

Enter Edge by V1 and V2 : [Read the graph from starting node 0]
0 1

Enter corresponding weight : 10

Enter Edge by V1 and V2 : [Read the graph from starting node 0]
1 2

Enter corresponding weight : 1

Enter Edge by V1 and V2 :[Read the graph from starting node 0]
2 3

Enter corresponding weight :2

Enter Edge by V1 and V2 : [Read the graph from starting node 0]
3 4

Enter corresponding weight : 3

Enter Edge by V1 and V2 : [Read the graph from starting node 0]
4 0

Enter corresponding weight :5

Enter Edge by V1 and V2 : [Read the graph from starting node 0]
1 3

Enter corresponding weight : 6

Enter Edge by V1 and V2 : [Read the graph from starting node 0]

4 2

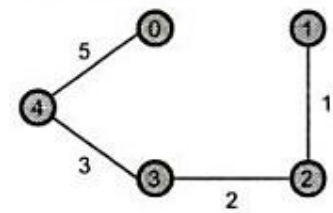
Enter corresponding weight : 7

The Minimal Spanning Tree Is :

Edge (0 4)and weight = 5
Edge (3 4)and weight = 3
Edge (2 3)and weight = 2
Edge (1 2)and weight = 1

Total Path Length Is = 11

The MST will be



3.4.2. Kruskal's Algorithm

Kruskal's algorithm is another algorithm of obtaining minimum spanning tree. Th algorithm is discovered by a second year graduate student Joseph Kruskal. In this algorithm always the minimum cost edge has to be selected. But it is not necessary that selected optimum edge is adjacent.

Difference between Prim's and Kruskal's Algorithm

Prim's Algorithm	Kruskal's Algorithm
This algorithm is for obtaining minimum spanning tree by selecting the adjacent vertices of already selected vertices.	This algorithm is for obtaining minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertics.

Let us understand this algorithm with the help of some example.

Example : Consider the graph given below :

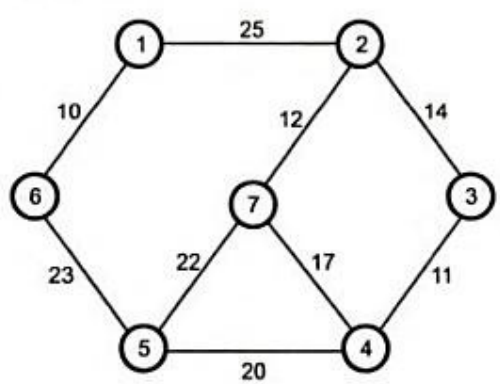
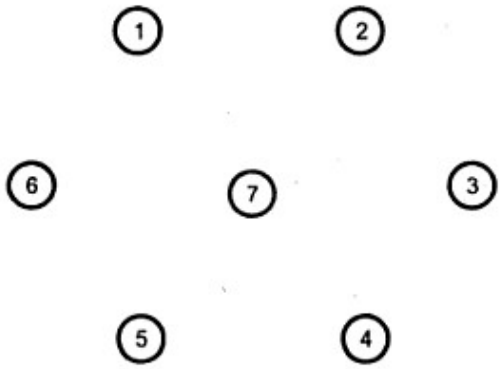


Fig. 3.4 Graph

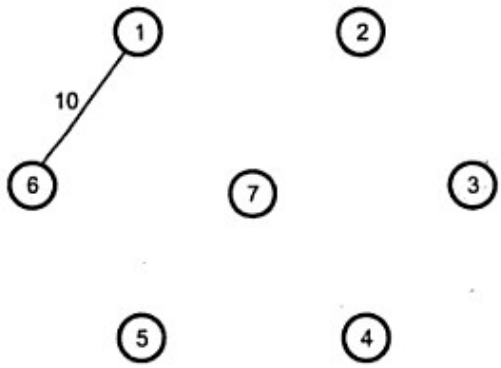
First we will select all the vertices. Then an edge with optimum weight is selected from heap, even though it is not adjacent to previously selected edge. Care should be taken for not forming circuit.

Step 1 :



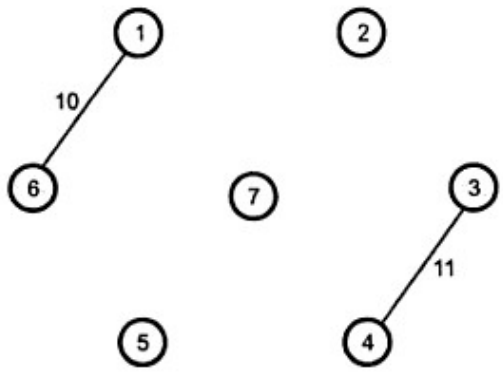
Total weight = 0

Step 2 :



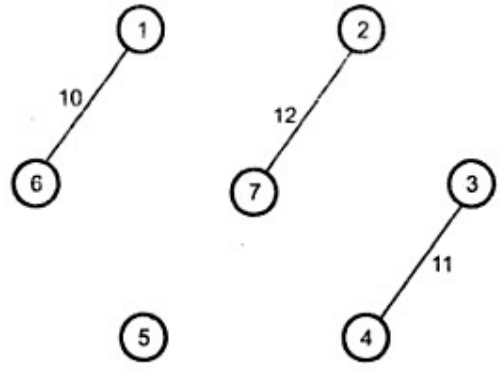
Total weight = 10

Step 3 :



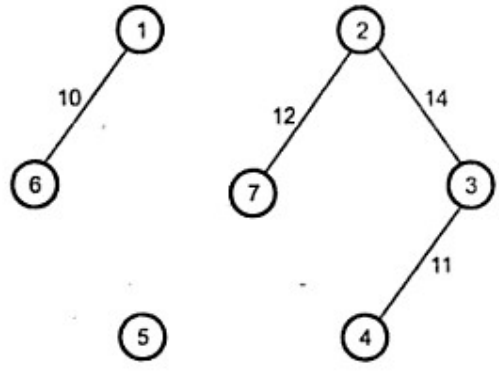
Total weight = 21

Step 4 :



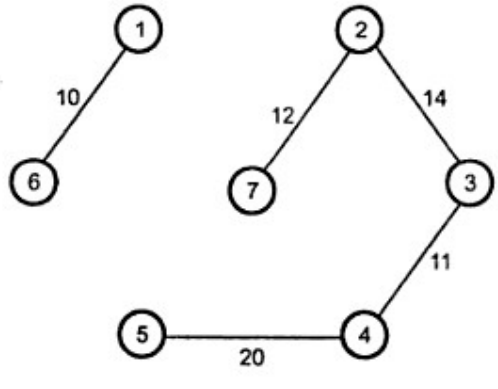
Total weight = 33

Step 5 :



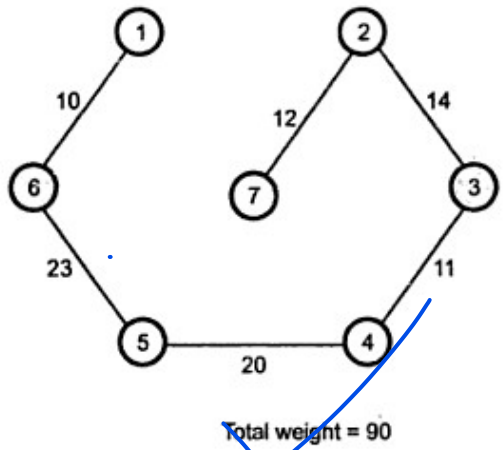
Total weight = 47

Step 6 :



Total weight = 67

Step 7 :



3.4.3 Algorithm

```
Algorithm  spanning_tree()
//Problem Description : This algorithm finds the minimum
//spanning tree using Kruskal's algorithm
//Input : The adjacency matrix graph G containing cost
//Output : prints the spanning tree with the total cost of
//spanning tree
 count←0
 k←0
 sum←0
   for i←0 to tot_nodes do
      parent[i]←i
while(count!=tot_nodes-1)do
 {
   pos←Minimum(tot_edges);//finding the minimum cost edge
      if(pos=-1)then//Perhaps no node in the graph
         break
       v1←G[pos].v1
       v2←G[pos].v2
       i←Find(v1,parent)
       j←Find(v2,parent)
       if(i!=j)then
       {
        tree[k][0] ←v1
//storing the minimum edge in array tree[]
```

Tree [ ] [ ] is an array in which the spanning tree edges are stored.

```
        tree[k][1] ←v2
        k++
        count++;
        sum+←G[pos].cost
  //accumulating the total cost of MST
        Union(i,j,parent);
        }
        G[pos].cost INFINITY
       }
       if(count=tot_nodes-1)then
       {
          for i←0 to tot_nodes-1
        {
          write(tree[i][0],tree[i][1])
        }
         write("Cost of Spanning Tree is ",sum)
       }
```

Computing total cost of all the minimum distances.

For each node of i, the minimum distance edges are collected in array **tree [ ] [ ]**. The spanning tree is printed here.

C function

```
void spanning_tree()
{
 int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
 int sum;
 int Find(int v2,int parent[]);
 void Union(int i,int j,int parent[]);
 count=0;
 k=0;
 sum=0;
 for(i=0;i<tot_nodes;i++)
       parent[i]=i;
 while(count!=tot_nodes-1)
 {
        pos=Minimum(tot_edges);//finding the minimum cost edge
        if(pos==-1)//Perhaps no node in the graph
            break;
        v1=G[pos].v1;
        v2=G[pos].v2;
        i=Find(v1,parent);
```

```
        j=Find(v2,parent);
        if(i!=j)
        {
            tree[k][0]=v1;//storing the minimum edge in array
                          tree[]
            tree[k][1]=v2;
            k++;
            count++;
            sum+=G[pos].cost;//accumulating the total cost of MST
            Union(i,j,parent);
        }
        G[pos].cost=INFINITY;
       }
if(count==tot_nodes-1)
       {
        printf("\n Spanning tree is...");
        printf("\n--------------------------\n");
        for(i=0;i<tot_nodes-1;i++)
        {
             printf("[%d",tree[i][0]);
             printf(" - ");
             printf("%d",tree[i][1]);
             printf("]");
        }
        printf("\n--------------------------");
        printf("\nCost of Spanning Tree is = %d",sum);
       }
       else
        {
            printf("There is no Spanning Tree");
        }
}
```

Analysis

The efficiency of Kruskal's algorithm is $\Theta(|E| \log |E|)$ where E is total number of edges in the graph.

C Program

```
/*****************************************************************
    Implementation of Kruskal's Algorithm
*****************************************************************/
#include<stdio.h>
#define INFINITY 999
typedef struct Graph
{
 int v1;
 int v2;
 int cost;
}GR;
GR G[20];
int tot_edges,tot_nodes;
void create();
void spanning_tree();
int Minimum(int);

void main()
{
 printf("\n\t Graph Creation by adjacency matrix ");
 create();
 spanning_tree();
}
void create()
{

 int k;
 printf("\n Enter Total number of nodes: ");
 scanf("%d",&tot_nodes);
 printf("\n Enter Total number of edges: ");
 scanf("%d",&tot_edges);
 for(k=0;k<tot_edges;k++)
 {
    printf("\n Enter Edge in (V1 V2)form ");
    scanf("%d%d",&G[k].v1,&G[k].v2);
    printf("\n Enter Corresponding Cost ");
    scanf("%d",&G[k].cost);
```

```
 }
}
void spanning_tree()
{
 int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
 int sum;
 int Find(int v2,int parent[]);
 void Union(int i,int j,int parent[]);
 count=0;
 k=0;
 sum=0;
 for(i=0;i<tot_nodes;i++)
        parent[i]=i;
 while(count!=tot_nodes-1)
 {
       pos=Minimum(tot_edges);//finding the minimum cost edge
       if(pos==-1)//Perhaps no node in the graph
           break;
       v1=G[pos].v1;
       v2=G[pos].v2;
       i=Find(v1,parent);
       j=Find(v2,parent);
       if(i!=j)
       {
           tree[k][0]=v1;//storing the minimum edge in array
                          tree[]
           tree[k][1]=v2;
           k++;
           count++;
           sum+=G[pos].cost;//accumulating the total cost of MST
           Union(i,j,parent);
       }
       G[pos].cost=INFINITY;
    }
       if(count==tot_nodes-1)
       {
        printf("\n Spanning tree is...");
        printf("\n-------------------------\n");
```

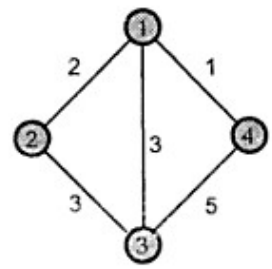
```
        for(i=0;i<tot_nodes-1;i++)
        {
            printf("[%d",tree[i][0]);
            printf(" - ");
            printf("%d",tree[i][1]);
            printf("]");
        }
        printf("\n-------------------------");
        printf("\nCost of Spanning Tree is = %d",sum);
       }
       else
        {
            printf("There is no Spanning Tree");
        }
}
int Minimum(int n)
{
 int i,small,pos;
 small=INFINITY;
 pos=-1;
 for(i=0;i<n;i++)
 {
       if(G[i].cost<small)
       {
           small=G[i].cost;
           pos=i;
       }
 }
 return pos;
}
int Find(int v2,int parent[])
{
  while(parent[v2]!=v2)
{
       v2=parent[v2];
    }
    return v2;
}
```

```
void Union(int i,int j,int parent[])
{
 if(i<j)
        parent[j]=i;
 else
        parent[i]=j;
}
```

Output

```
        Graph Creation by adjacency matrix
Enter Total number of nodes: 4

Enter Total number of edges: 5

Enter Edge in (V1 V2)form 1 2

Enter Corresponding Cost 2

Enter Edge in (V1 V2)form 1 4

Enter Corresponding Cost 1

Enter Edge in (V1 V2)form 1 3

Enter Corresponding Cost 3

Enter Edge in (V1 V2)form 2 3

Enter Corresponding Cost 3

Enter Edge in (V1 V2)form 4 3

Enter Corresponding Cost 5

Spanning tree is...
---------------------------
  |1 - 4||1 - 2||1 - 3|
---------------------------
  Cost of Spanning Tree is = 6
```

Graph to be taken as input



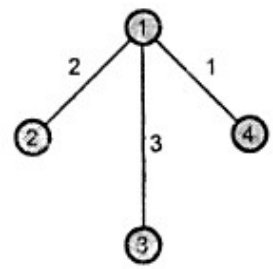


Fig. 3.5 Spanning tree

3.5 Single Source Shortest Path

Many times, Graph is used to represent the distances between two cities. Everybody is often interested in moving from one city to other as quickly as possible. The single source shortest path is based on this interest.

In single source shortest path problem the shortest distance from a single vertex called source is obtained.

Let G(V, E) be a graph, then in single source shortest path the shortest paths from vertex v_0 to all remaining vertex is determined. The vertex v_0 is then called as source and the last vertex is called destination.

It is assumed that all the distances are positive.

Example : Consider a graph G as given below.

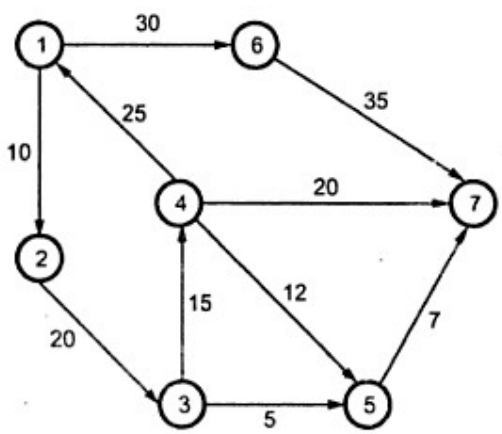


Fig. 3.6 Directed graph for single source shortest path

We will start from source vertex 1. Hence set S[1] = 1.

Now shortest distance from vertex 1 is 10. i.e. $1 \rightarrow 2 = 10$. Hence {1, 2} and min = 10.

From vertex 2 the next vertex chosen is 3.

{1, 2} = 10

{1, 3} = ∞

{1, 5} = ∞

{1, 6} = ∞

{1, 7} = ∞

Now

$$\{1, 2, 3\} = 30$$
$$\{1, 2, 4\} = \infty$$
$$\{1, 2, 7\} = \infty$$
$$\{1, 2, 5\} = \infty$$
$$\{1, 2, 6\} = \infty$$

Hence select 3.

$$\therefore \quad S[3] = 1$$

Now

$$\{1, 2, 3, 4\} = 45$$
$$\{1, 2, 3, 5\} = 35$$
$$\{1, 2, 3, 6\} = \infty$$
$$\{1, 2, 3, 7\} = \infty$$

Hence select next vertex as 5.

$$\therefore \quad S[5] = 1$$

Now

$$\{1, 2, 3, 5, 6\} = \infty$$
$$\{1, 2, 3, 5, 7\} = 42$$

Hence vertex 7 will be selected. In single source shortest path if destination vertex is 7 then we have achieved shortest path 1 - 2 - 3 - 5 - 7 with path length 42. The single source shortest path from each vertex is summerised below -

1, 2	10
1, 2, 3	30
1, 2, 3, 4	45
1, 2, 3, 5	35
1, 2, 3, 4, 7	65
1, 2, 3, 5, 7	42
1, 6	30
1, 6, 7	65

3.5.1 Algorithm

The algorithm for single source shortest path is given as

```
Algorithm Single_Short_Path( p,cost,Dist,n)
{
// cost is an adjacency matrix storing cost of each edge i.e. cost[1 : n,1 : n]. Given
// graph can be represented by cost.
// Dist is a set that stores the shortest path from the source vertex 'p' to any other
// vertex in the graph.
// S stores all the visited vertices of graph. It is of Boolean type array.
// initially
 for i ←1 to n do
{
 S[i] ← 0;
  Dist ← cost[p,i];
}
S[p] ←1 // set pth vertex to true in array S and i.e. put p in S
Dist[p] ← 0.0;
for val ← 2 to n–2 do
{
   // obtain n–1 paths from p
  Choose q from the vertices that are not visited (not in S) and with minimum
distance.
Dist[q] = min{Dist[i]};
 S[q] ← 1// put q in S
 /*update the Distance values of the other nodes*/
  for (all node r adjacent to q with S[r] = 0) do
    if( Dist[r]>(Dist[q]+cost[p,q])) then
        Dist[r] ← Dist[q]+Dist[p,q];
 }
}
```

3.5.2 Analysis

Time Complexity of the algorithm:

- The 1st for-loop clearly takes O(n) time
- Choosing q takes O(n) time, because it involves finding a minimum in an array.

- The innermost for-loop for updating Dist iterates at most n times, thus takes O(n) time.
- Therefore, the for-loop iterating over val takes O(n*n) time

Thus, single source shortest path takes $O(n^2)$ time.

3.5.3 C Program

```
/*****************************************************************
This program is for implementing Dijkstra's single source shortest
path Algorithm
*****************************************************************/
  #include<stdio.h>
  #include<conio.h>
  #define infinity 999
  int path[10];
  void main()
  {
   int tot_nodes,i,j,cost[10][10],dist[10],s[10];
   void create(int tot_nodes,int cost[][10]);
   void Dijkstra(int tot_nodes,int cost[][10],int i,int dist[10]);
   void display(int i,int j,int dist[10]);
   clrscr();
   printf("\n\t\t Creation of graph ");
   printf("\n Enter total number of nodes ");
   scanf("%d",&tot_nodes);
   create(tot_nodes,cost);
   for(i=0;i<tot_nodes;i++)
   {
        printf("\n\t\t\t Press any key to continue...");
        printf("\n\t\t When Source =%d\n",i);
        for(j=0;j<tot_nodes;j++)
        {
         Dijkstra(tot_nodes,cost,i,dist);
         if(dist[j]==infinity)
            printf("\n There is no path to %d\n",j);
         else
         {

            display(i,j,dist);
```

```
        }
       }
    }
}
void create(int tot_nodes,int cost[][10])
{
 int i,j,val,tot_edges,count=0;
 for(i=0;i<tot_nodes;i++)
 {
       for(j=0;j<tot_nodes;j++)
       {
        if(i==j)
           cost[i][j]=0;//diagonal elements are 0
        else
           cost[i][j]=infinity;
       }
 }
 printf("\n Total number of edges ");
 scanf("%d",&tot_edges);
 while(count<tot_edges)
       {
        printf("\n Enter Vi and Vj");
        scanf("%d%d",&i,&j);
        printf("\n Enter the cost along this edge ");
        scanf("%d",&val);
        cost[j][i]=val;
        cost[i][j]=val;
        count++;
       }
}
void Dijkstra(int tot_nodes,int cost[10][10],int source,int
dist[])
{
 int i,j,v1,v2,min_dist;
 int s[10];
 for(i=0;i<tot_nodes;i++)
 {
       dist[i]=cost[source][i];//initially put the
       s[i]=0; //distance from source vertex to i
```

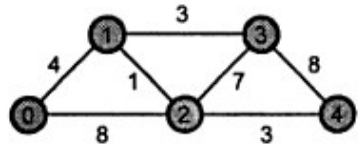
```
        //i is varied for each vertex
        path[i]=source;//all the sources are put in path
    }
    s[source]=1;
    for(i=1;i<tot_nodes;i++)
    {
        min_dist=infinity;
        v1=-1;//reset previous value of v1
        for(j=0;j<tot_nodes;j++)
        {
         if(s[j]==0)
         {
            if(dist[j]<min_dist)
            {
             min_dist=dist[j];//finding minimum disatnce
             v1=j;
            }
         }
        }
        s[v1]=1;
        for(v2=0;v2<tot_nodes;v2++)
        {
         if(s[v2]==0)
         {
            if(dist[v1]+cost[v1][v2]<dist[v2])
            {
                dist[v2]=dist[v1]+cost[v1][v2];
                path[v2]=v1;
            }
         }
        }
    }
}
void display(int source,int destination,int dist[])
{
 int i;
 getch();
 printf("\n Step by Step shortest path is...\n");
 for(i=destination;i!=source;i=path[i])
```

```
{
        printf("%d<-",i);
}
printf("%d",i);
printf("  The length=%d",dist[destination]);
}
```

Output

```
  Creation of graph
Enter total number of nodes 5

Total number of edges 7

Enter Vi and Vj 0 1

Enter the cost along this edge 4

Enter Vi and Vj 0 2

Enter the cost along this edge 8

Enter Vi and Vj 1 2

Enter the cost along this edge 1

Enter Vi and Vj 1 3

Enter the cost along this edge 3

Enter Vi and Vj 2 3

Enter the cost along this edge 7

Enter Vi and Vj 2 4

Enter the cost along this edge 3

Enter Vi and Vj 3 4
```

The input graph is



```
Enter the cost along this edge 8

                Press any key to continue...
            When Source =0
 Step by Step shortest path is...
0  The length=0
 Step by Step shortest path is...
1<- 0  The length=4
 Step by Step shortest path is...
2<-1<-0  The length=5
 Step by Step shortest path is...
3<-1<-0  The length=7
 Step by Step shortest path is...
4<-2<-1<-0  The length=8
                Press any key to continue...
            When Source =1

 Step by Step shortest path is...
0<-1  The length=4
 Step by Step shortest path is...
1  The length=0
 Step by Step shortest path is...
2<-1  The length=1
 Step by Step shortest path is...
3<-1  The length=3
 Step by Step shortest path is...
4<-2<-1  The length=4
                Press any key to continue...
            When Source =2

 Step by Step shortest path is...
0<-1<-2  The length=5
 Step by Step shortest path is...
1<-2  The length=1
 Step by Step shortest path is...
2  The length=0
 Step by Step shortest path is...
```

```
3<-1<-2  The length=4
 Step by Step shortest path is...
4<-2  The length=3
                    Press any key to continue...
                When Source =3

 Step by Step shortest path is...
0<-1<-3  The length=7
 Step by Step shortest path is...
1<-3  The length=3
 Step by Step shortest path is...
2<-1<-3  The length=4
 Step by Step shortest path is...
3  The length=0
 Step by Step shortest path is...
4<-2<-1<-3  The length=7
                    Press any key to continue...
                When Source =4

 Step by Step shortest path is...
0<-1<-2<- 4  The length=8
 Step by Step shortest path is...
1<-2<- 4  The length=4
 Step by Step shortest path is...
2<- 4  The length=3
 Step by Step shortest path is...
3<-1<-2<- 4  The length=7
 Step by Step shortest path is...
4  The length=0
```

3.6 Job Sequencing with Deadlines

Consider that there are n jobs that are to be executed. At any time t=1,2,3,.. only exactly one job is to be executed. The profits pi are given. These profits are gained by corresponding jobs. For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

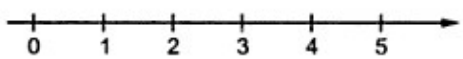
Let n = 4

n	pi	di
1	70	2
2	12	1
3	18	2
4	35	1

We will follow following rules to obtain the feasible solution

- Each job takes one unit of time.
- If job starts **before or at its deadline**, profit is obtained, otherwise no profit.
- Goal is to schedule jobs to maximize the total profit.
- Consider all possible schedules and compute the minimum total time in the system.

Consider the Time line as



The feasible solutions are obtained by various permutations and combinations of jobs.

n	pi
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105
4,3	53

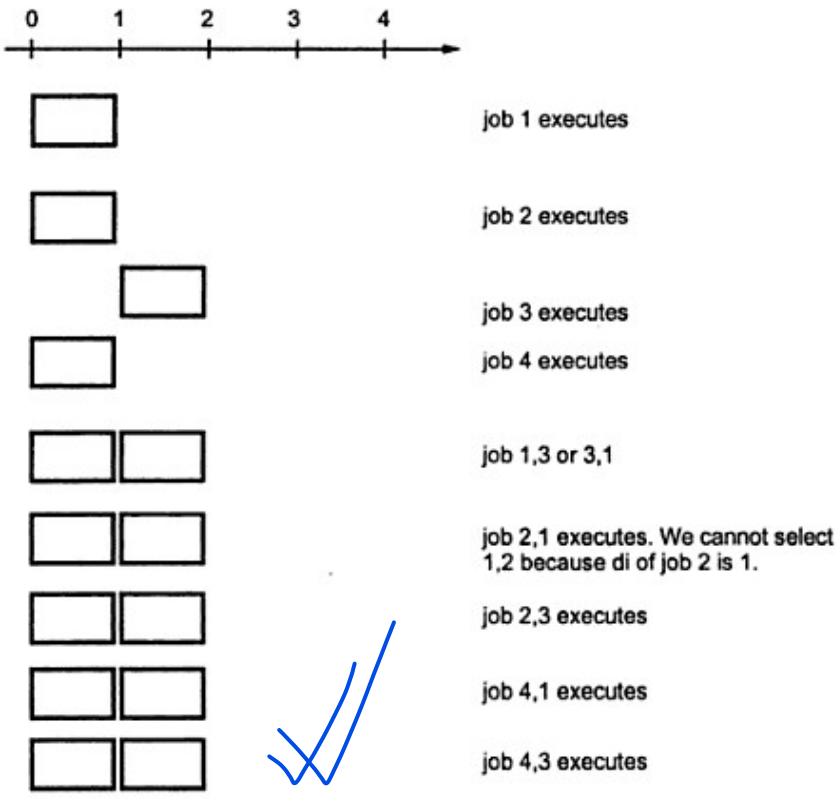


Fig. 3.7 Job sequencing with deadline

Each job takes only one unit of time. Deadline of job means a time **on which or before which the job has to be executed.** The sequence {2,4} is not allowed because both have deadline 1. If job 2 is started at 0 it will be completed on 1 but we cannot start job 4 on 1 since deadline of job 4 is 1. The feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines and highest profit can be gained.

- The optimal solution is a feasible solution with maximum profit.
- In above example sequence 3,2 is not considered as $d_3 > d_2$ but we have considered the sequence 2,3 as feasible solution because $d_2 < d_3$.
- We have chosen job 1 first then we have chosen job 4. The solution 4,1 is feasible as the order of execution is 4 then 1. Also $d_4 < d_1$. If we try {1,3,4} then it is not a feasible solution, hence reject 3 from the set. Similarly if we add job 2 in the sequence then the sequence becomes {1,2,4}. This is also not a feasible solution hence reject it. Finally the feasible sequence is 4,1. This sequence is optimum solution as well.

3.6.1 Algorithm

The algorithm for job sequencing is as given below

```
Algorithm Job_seq(D,J,n)
{
//Problem description : This algorithm is for job sequencing using Greedy method.
//D[i] denotes ith deadline where 1 ≤ i ≤ n
//J[i] denotes ith job
// D[J[i]] ≤ D[J[i+1]]
//Initially
D[0]←0;
J[0]←0;
J[1]←1;
count←1;
for ←2 to n do
{
 t←count;
 while(D[J[t]] > D[i]) AND D[J[t]]!=t)) do t←t-1;
 if((D[J[t]] ≤ D[i])AND(D[i] > t)) then
 {
   //insertion of ith feasible sequence into J array
   for s←count to (t+1) step -1 do
   J[s+1] ← J[s];
   J[t+1]←I;
   count←count+1;
 }//end of if
}//end of while
return count;
}
```

The sequence of J will be inserted if and only if D[J[t]]!=t. This also means that the job J will be processed if it is in within the deadline.

The computing time taken by above Job_seq algorithm is $O(n^2)$, because the basic operation of computing sequence in array J is within two nested for loops.

3.7 Optimal Storage on Tapes

The computer programs are stored on magnetic tape. When user wants any program then that program can be retrieved from the tape. These programs can be of

any arbitrary lengths. Of course, the sum of lengths of all these programs must be less than or equal to the length of the storage tape.

Let, n be the number of programs that can be stored on the tape.

Let, L be the length of the magnetic tape on which n programs can be stored. And L_i be the length of each program, such that $1 \le i \le n$. When we want to retrieve any program then it is assumed that tape head is positioned at front. Let t_j be the time required to retrieve the program. Suppose, if we retrieve the programs equally then **Mean Retrieval Time (MRT)** is given by following formula.

$$MRT = \frac{1}{n}\sum_{j=1}^{n} t_j$$

According to the ordering of the programs mean retrieval time varies. There is another constant d(I) using which optimal ordering can be decided.

Here $I = i_1, i_2, i_3 \ldots i_n$. That means I represents ordering of programs.

For example : Consider that there are 3 programs that are stored on the tape. That means n = 3. The lengths of these programs $(L_1, L_2, L_3) = (7, 12, 4)$. Then there are n! = 3! = 6 possible orderings. Based on these ordering the d value can be computed as follows.

Ordering I	d(I)	Result
1, 2, 3	7+7+12+7+12+4	49
1, 3, 2	7+7+4+7+4+12	41 ← Minimum value
2, 1, 3	12+12+7+12+7+4	54
2, 3, 1	12+12+4+7+4+12	51
3, 1, 2	4+4+7+12+7+12	46
3, 2, 1	4+4+12+4+12+7	43

From above computations, clearly 1, 3, 2 yields the minimum d(I) value.

Using greedy approach the next program for the required permutation is used. If some sorting algorithm is used for determining the ordering of the programs then it will require O (n log n) time.

Following pseudo code can be used for optimal ordering on storage tape.

```
Algorithm Assign_program (n, k)
//Problem description: This algorithm assigns programs on the tape
//Input: n is number of programs and k is number of tapes
    j ← 0      //tape number
```

```
for (i ← 1 to n) do
{
    Write ("Program", i, "for tape" j)
    j ← (j+1) mod k
}
```

Analysis

The basic operation is done in the for loop. Hence time complexity of above algorithm is $\theta(n)$.

Solved Exercise

Q.1 *Write procedure for GREEDY KNAPSACK (P,W,M,X,N) where P and W contains profits and weights, M is Knapsack size and X is the solution vector.*

Ans. : Algorithm for Greedy Knapsack problem is as given below -

```
Algorithm GREEDY_KNAPSACK (P,W,M,X,N)
{
  //P[i] contains the profits of i items.
  //W[i] contains weights of i items.
  //where 1 ≤ i ≤ N.
  // X[i] is the solution vector.
  //M is Knapsack size.
    for (i ← 1 to n) do
    {
      if (W[i]<M) then  //with the capacity
      {
        X[i] ← 1.0;
        M ← M – W[i];
      }
    }
    if (i<= N) then
    X[i] ← M/W[i];      // X[i] gives the output.
}
```

Q.2 *Write a general procedure GREEDY. Apply it to the optimal storage on tapes if n = 3 and $(I_1, I_2, I_3) = (5, 10, 3)$.*

Ans. : The general procedure for GREEDY is as given below -

```
Algorithm GREEDY (a,n)
//Array a[1 : n] is the objective domain.
//From which solution can be picked up.
//Initially solution is empty.
{
   solution ← ϕ;
   for i ← 1 to n do
   {
      S ← selection(a);
      if (feasible(solution,S)) then
      {
         solution ← Add (solution,S);
         // adds the S to solution set.
      }
      else
         reject();//if solution is not feasible then reject it.
         }
         return solution;
   }
```

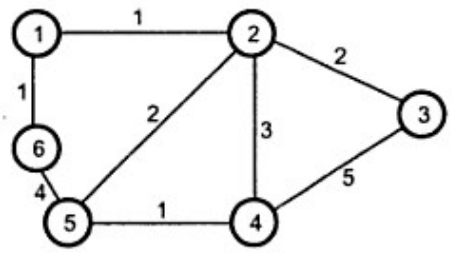
Let, n = 3 and $(I_1, I_2, I_3) = (5, 10, 3)$ for optimal storage on tape. Here I_1, I_2 and I_3 represent the length of the program on the input tape. If we retrieve a program from the tape, the tape is initially positioned at front. Hence if programs are stored in the order i_1, i_2, i_3..... then some time t is required to retrieve any program i_j. If all programs can be retrieved equally then the retrieval time which is required is called mean retrieval time.

We want such a sequence that gives minimum mean retrieval time. For n = 3, n! = 6 orderings are possible. The computation of mean retrieval time with corresponding ordering is as given below -

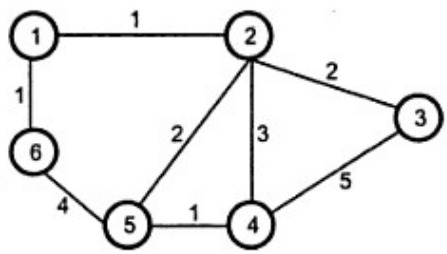
Solution	Ordering	Mean retrieval time
1	I_1 I_2 I_3	5+5+10+5+10+3=38
2	I_1 I_3 I_2	5+5+3+5+3+10=31
3	I_2 I_1 I_3	10+10+5+10+5+3=43
4	I_2 I_3 I_1	10+10+3+10+3+5=41
5	I_3 I_1 I_2	3+3+5+3+5+10=29
6	I_3 I_2 I_1	3+3+10+3+10+5=34

In above computations solution 5 is optimum solution. Hence optimal ordering is 3, 1, 2.

Q.3 *Write algorithm for Prim's minimum spanning tree. Apply it on following graph step by step.*

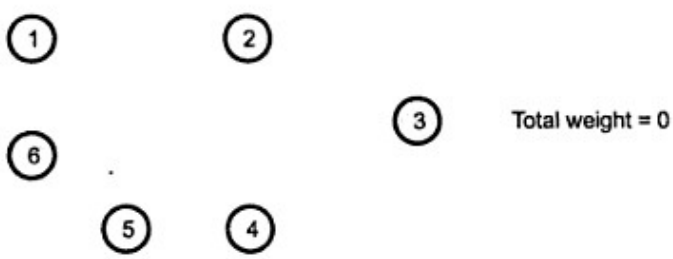


Ans. : For Prim's Algorithms refer section 3.4.1. Consider the given graph,

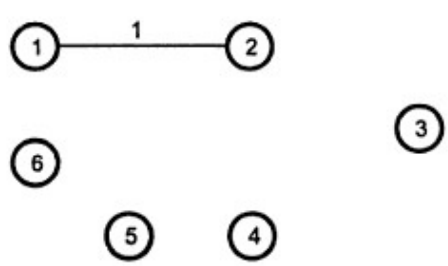


Now, we will consider all the vertices first. Then we will select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight. Care should be taken for not forming a circuit. And all the vertices should be visited.

Step 1 :

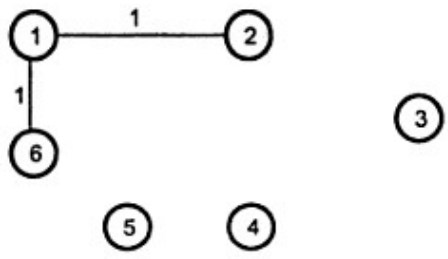


Step 2 :



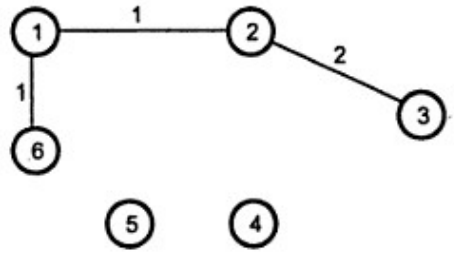
Total weight = 1

Step 3 :



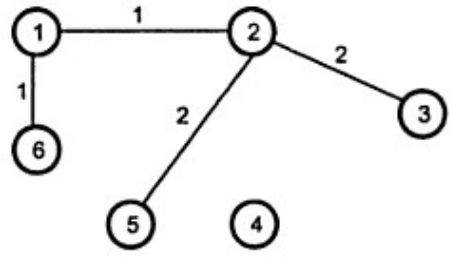
Total weight = 2

Step 4 :



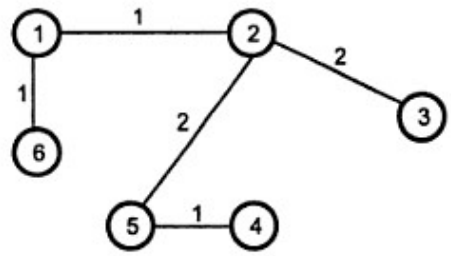
Total weight = 4

Step 5 :



Total weight = 6

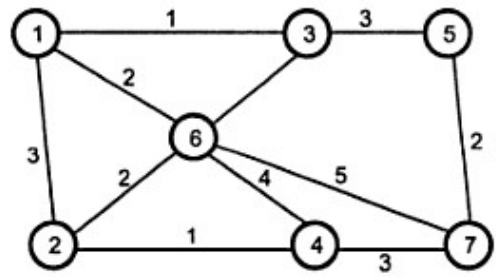
Step 6 :



Total weight = 7

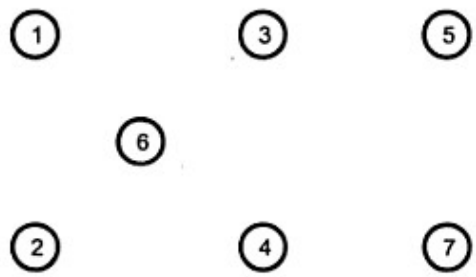
This is the minimum spanning tree with total weight 7. There is no circuit formed and we have got all the vertices in a minimum spanning tree.

Q.4 *Write algorithm for Kruskal's minimum spanning tree. Apply it on following graph step by step.*



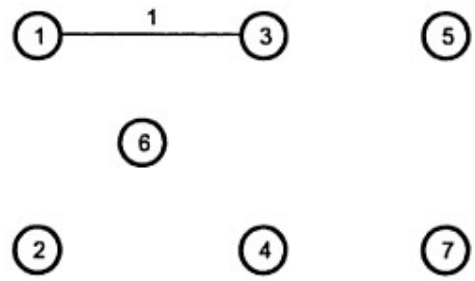
Ans. : Consider the given graph. First we will select all the vertices. Then an edge with optimum weight is selected form heap, even though it is not adjacent to previously selected edge. Care should be taken for not forming a circuit and all the vertices should be visited.

Step 1 :



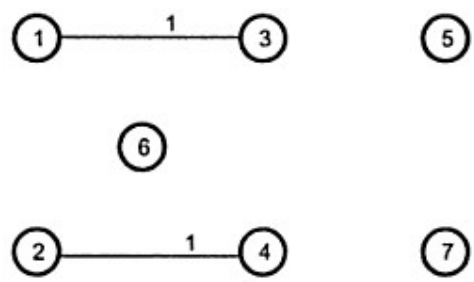
Total weight = 0

Step 2 :



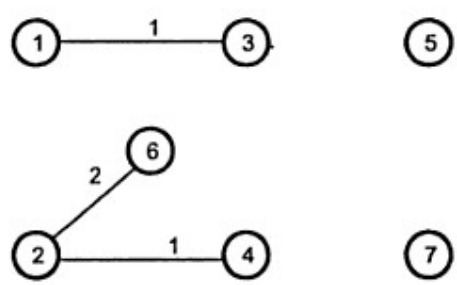
Total weight = 1

Step 3 :



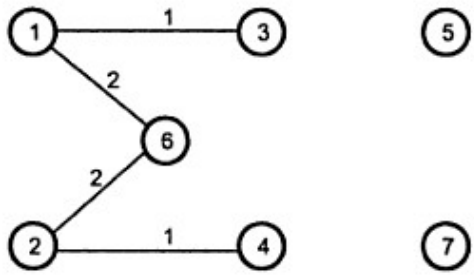
Total weight =2

Step 4 :



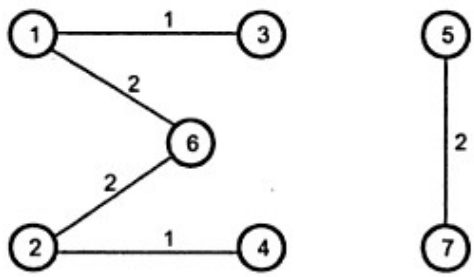
Total weight = 4

Step 5 :



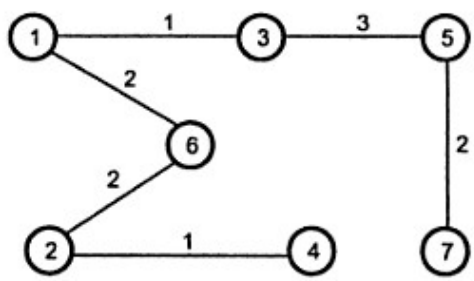
Total weight = 6

Step 6 :



Total weight = 8

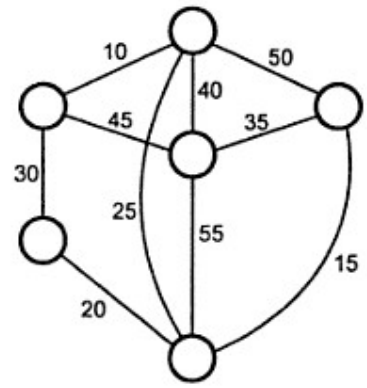
Step 7 :



Total weight = 11

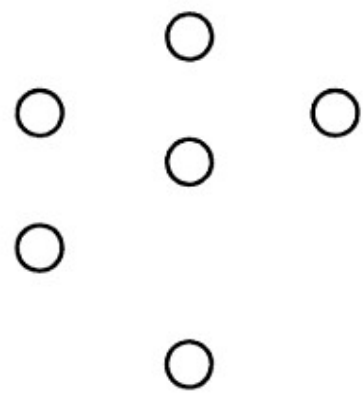
Thus minimum spanning tree is obtained using Kruskal's algorithm.

Q.5 *Obtain minimum spanning tree for the graph given below using Prims algorithm.*



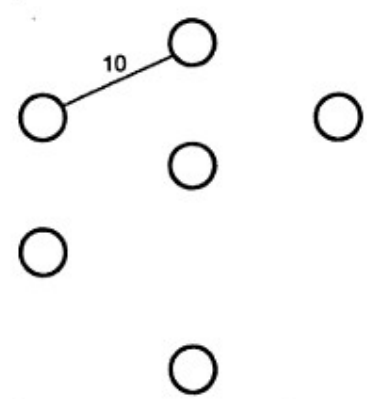
Ans. : Consider all vertices only. Then proceed by selecting an adjacent edge with optimum weight.

Step 1 :



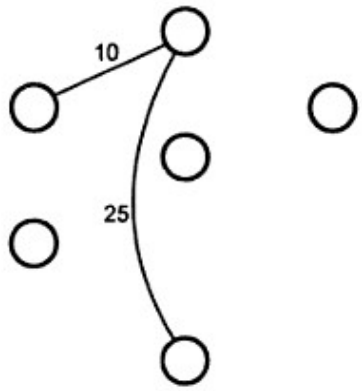
Total weight = 0

Step 2 :



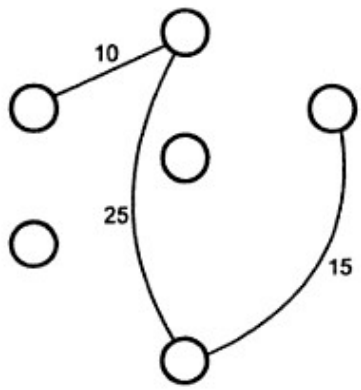
Total weight = 10

Step 3 :



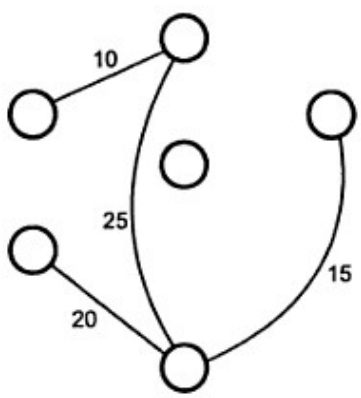
Total weight = 35

Step 4 :



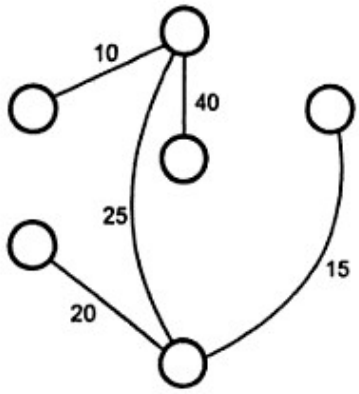
Total weight = 50

Step 5 :



Total weight = 70

Step 6 :



Thus minimum spanning tree is obtained using Prims algorithm.

Total weight = 110

Q.6 *Find an optimal solution to the Knapsack instance n = 7, M = 15 (P_1, P_2, P_3, ... P_7) = (10, 5, 15, 7, 6, 18, 3) and (W_1, W_2,.....W_7) = (2, 3, 5, 7, 1, 4, 1).*

Ans. : Let,

i	P_i	W_i	P_i / W_i
1	10	2	5
2	5	3	1.66
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	9/2 = 4.5
7	3	1	3

We will arrange $\frac{P_i}{W_i}$ in a decreasing order such that $\frac{P_i}{W_i} > \frac{P_{i+1}}{W_{i+1}}$.

$$\frac{P_5}{W_5} \geq \frac{P_1}{W_1} \geq \frac{P_6}{W_6} \geq \frac{P_3}{W_3} \geq \frac{P_7}{W_7} \geq \frac{P_2}{W_2} \geq \frac{P_4}{W_4}$$

Now we will select the objects in this order. We will sum up the profits and decreament the total capacity by the weight of selected object. Total capacity M = 15.

Action	Remaining Wt	Profit gained
Select i = 5	15 – 1 = 14	6 = 0 + 6
i = 1	14 – 2 = 12	16 = 10 + 6
i = 6	12 – 4 = 8	34 = 16 + 18
i = 7	8 – 1 = 7	34 + 3 = 37
i = 3	7 – 5 = 2	37 + 15 = 52
i = 2 We have selected fraction of this object	2 – 2 = 0	$52+\left(5*\frac{2}{3}\right)$ = 52 + 3.32 = 55.32

Thus we have selected fractional part of object 2. To obtain the fractional part of remaining object the formula used is

$$\frac{\text{remaining space in Knapsack}}{\text{actual weight of selected object}} \times (\text{Profit of selected object})$$

$$= \frac{2}{3} \times 5 = 3.32$$

Hence the solution to Knapsack is $\left(1,\ \frac{2}{3},\ 1,\ 0,\ 1,\ 1,\ 1\right)$

Here 1 indicates corresponding object is selected and 0 indicates corresponding object is not selected.

Q.7 *Apply Prim's algorithm to the following graph and obtain minimum spanning tree.*

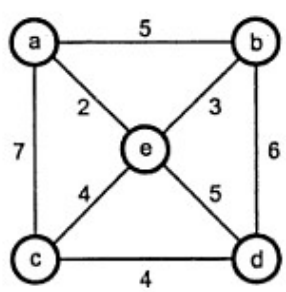
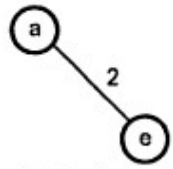
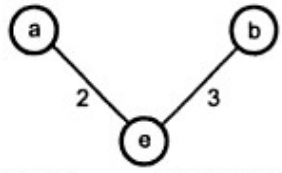
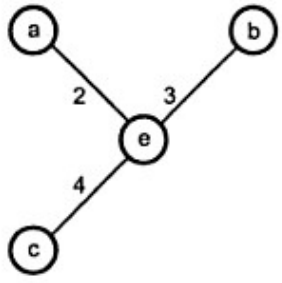
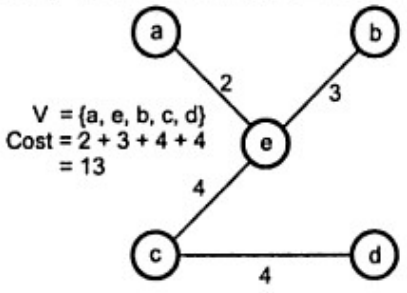
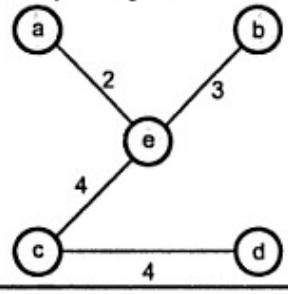


Fig. 3.8

Ans. :

We will first select a minimum distance edge from given graph.	**Step 1 :**  V = {a,e} Cost = 2
The next minimum distance edge is b-e. This edge is adjacent to previously selected vertex e.	**Step 2 :**  V = {a,e,b} Cost = 2 + 3 = 5
The next minimum distance edge is c-e which is adjacent to already selected edge b-e.	**Step 3 :**  V = {a,e,b,c} Cost = 2 + 3 + 4 = 9
Then next select an edge c-d which is with minimum distance and it is adjacent to already selected edge c-e.	**Step 4 :** 
Since all the vertices are visited and we get a connected tree as minimum spanning tree.	**Step 5 :** V{a, e ,b, c, d} and Cost = 2 + 3 + 4 + 4 = 13. The final minimum spanning tree will be - 

Q.8 *Apply Prim's algorithm to the following graph.*

Obtain MST.

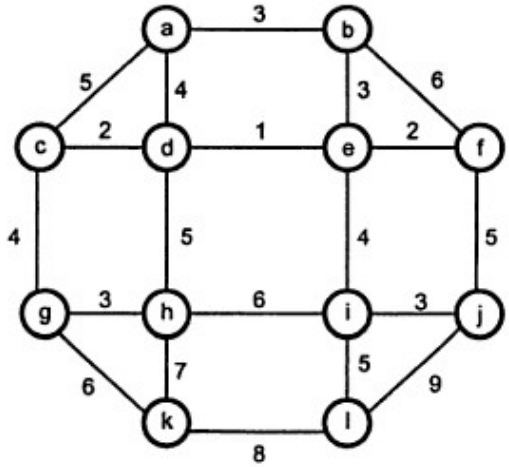
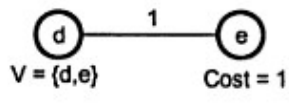
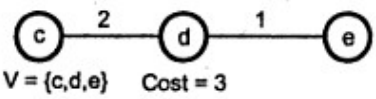
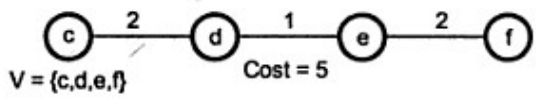
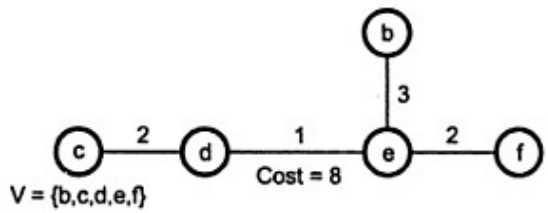
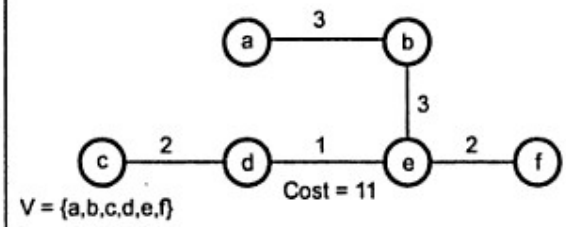
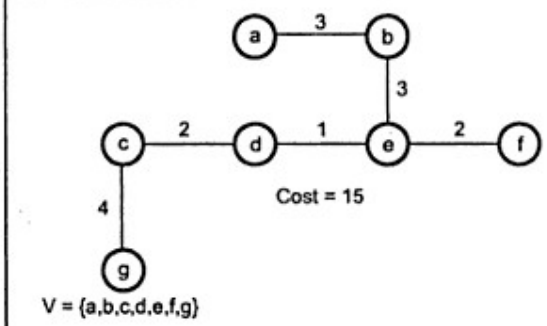
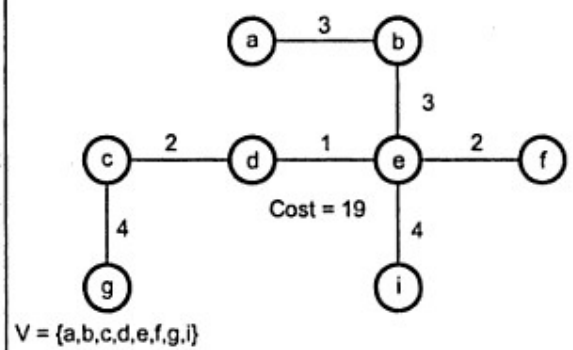
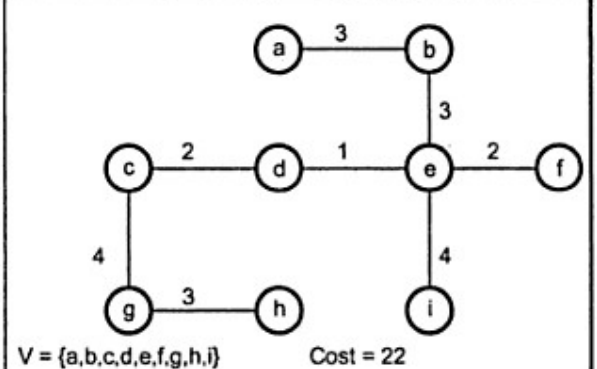
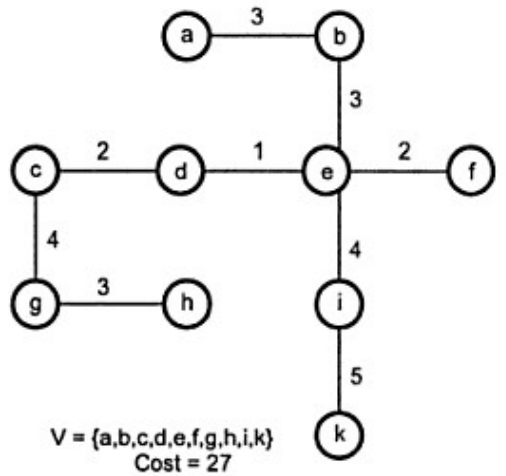
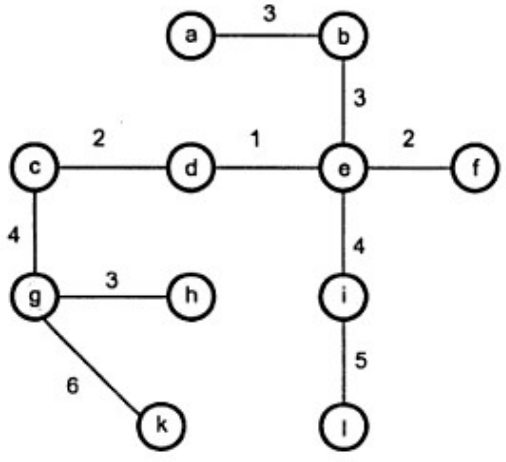


Fig. 3.9

Ans. :

We first select a minimum distance edge from given graph.	
Then select next minimum distance edge with previously selected edge.	
Then select next minimum distance edge with previously selected edge.	
Then select next minimum distance edge with previously selected edge.	

Then select next minimum distance edge with previously selected edge.	
Then select next minimum distance edge with previously selected edge.	
Then select next minimum vertex which is adjacent to already selected vertex.	
Then select next minimum vertex which is adjacent to already selected vertex.	

<table>
<tr>
<td>Then select next minimum vertex which is adjacent to already selected vertex.</td>
<td> 
</td>
</tr>
<tr>
<td>Then select next minimum vertex which is adjacent to already selected vertex.</td>
<td> 
</td>
</tr>
<tr>
<td colspan="2">Thus we get the spanning tree by visiting all the vertices and cost = 33.</td>
</tr>
</table>

Q.9 *Apply Kruskal's algorithm to find a minimum spanning tree of a given graph.*

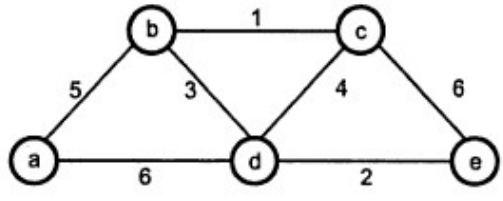
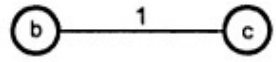
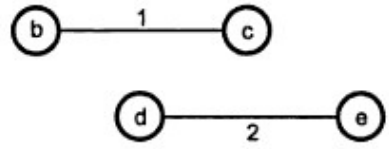
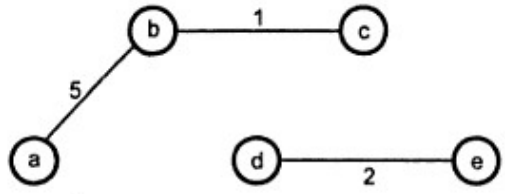
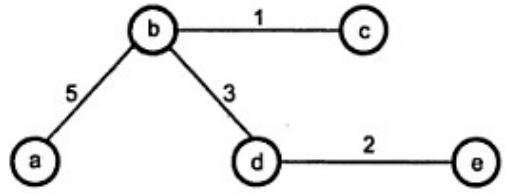


Fig. 3.10

Ans. :

We first select an edge with minimum weight.	 Total cost = 1
Then we select the next minimum weighted edge. It is not necessary that selected edge is adjacent.	 Total cost = 3
Then we select next minimum weight for an unvisited vertex.	 Total cost = 8
All the vertices are visited but since the spanning tree should be connected one. Hence we select an edge with minimum weight. Thus we get a minimum spanning tree with Kruskal's algorithm.	 Total cost = 11

Q.10 *Apply Kruskal's algorithm to find minimum spanning tree of following graph.*

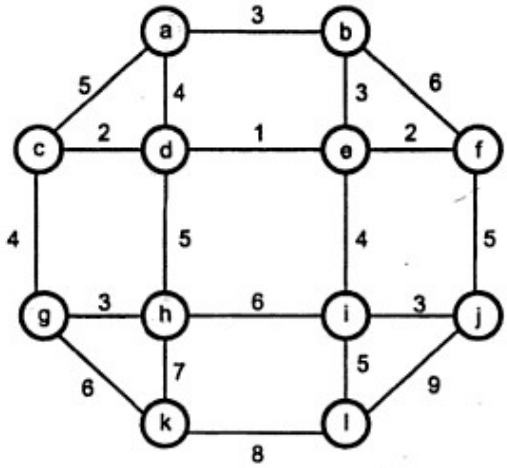
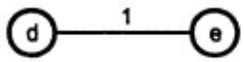
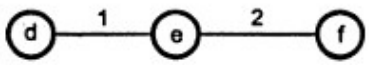
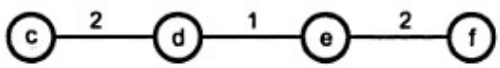
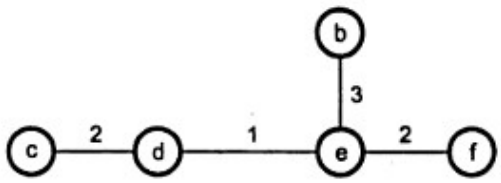
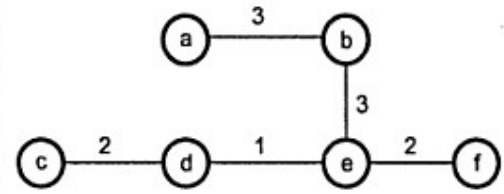
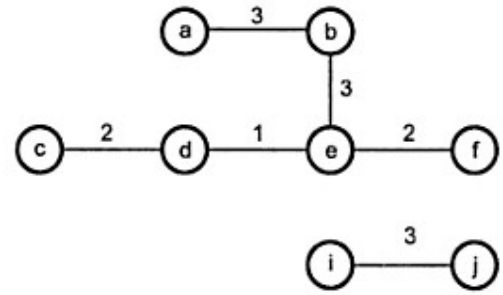
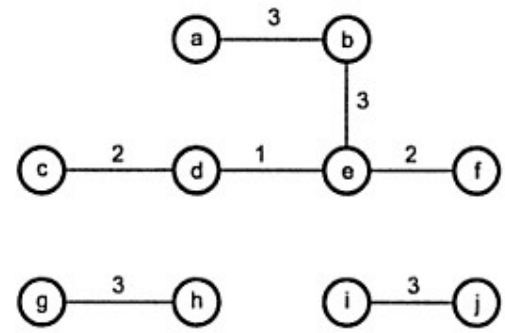
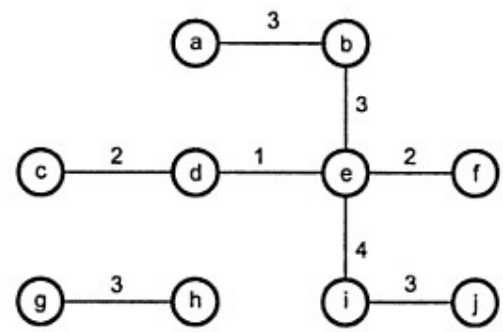


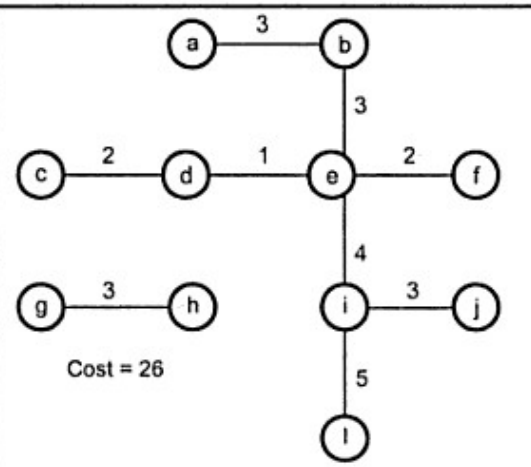
Fig. 3.11

Ans. :

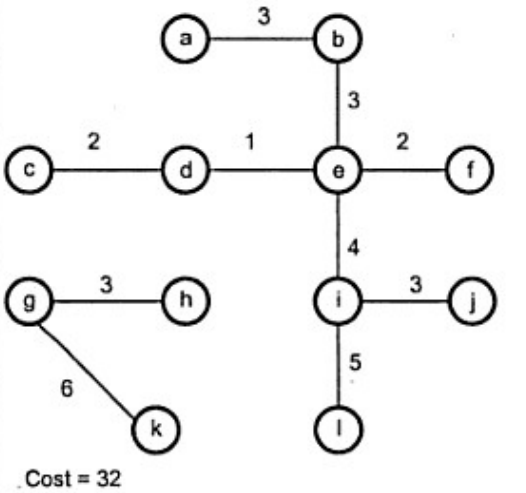
We first select an edge with minimum weight.	
Then we select the next minimum weighted edge. It is not necessary that selected edge is adjacent.	
Then select next minimum weighted edge.	 Cost = 5
Then select next minimum weighted edge.	 Cost = 8

Then select next minimum weighted edge.	 Cost = 11
Then select next minimum weighted edge.	 Cost = 14
Then select next minimum weighted edge.	 Cost = 17
Then select next minimum weighted edge in order to make the graph connected.	 Cost = 21

Then select next minimum weighted edge.



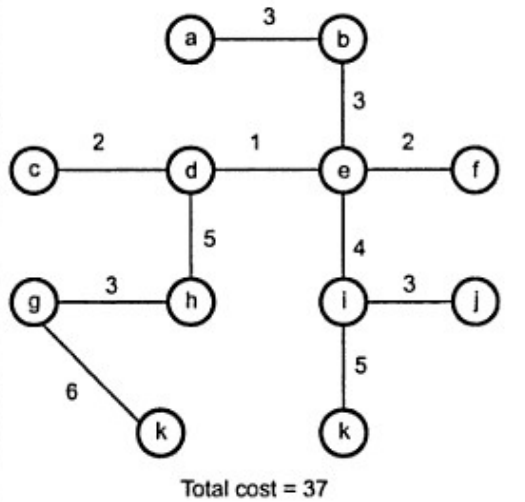
Then select next minimum weighted edge.



In order to make graph connected, select the minimum weighted edge.

Thus we get a spanning tree.

Total cost = 37



Q.11 *Solve the following instances of single-source shortest path problem with vertex a as source.*

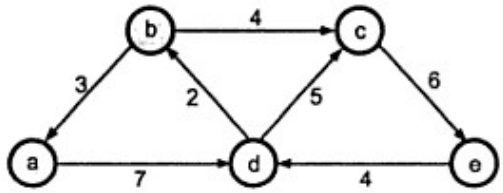
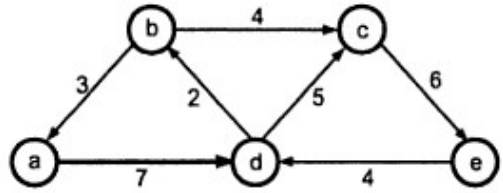
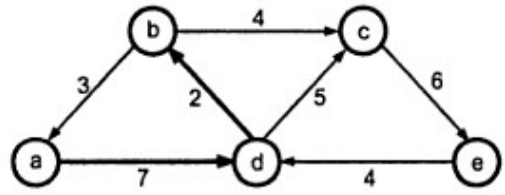
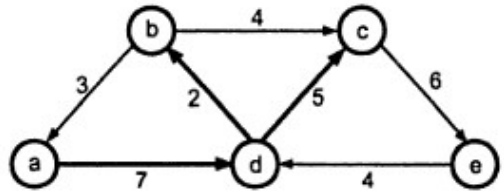
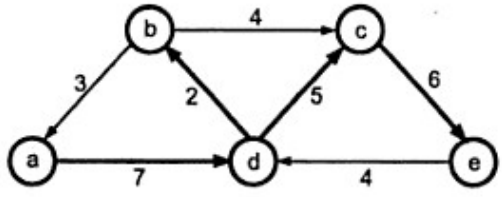


Fig. 3.12

Ans. : We will find the distances from source a to every other vertices. We write the distances in **destination (source minimum distance)** form.

a (–, 0)	b(a, ∞), c(a, ∞) **d(a, 7)**, e(a, ∞)	
d(a, 7)	**b(d, 9)**, c(d, 12) e(d, ∞)	
b(d, 9)	min[c(b, 13), c(d, 12)] = **c(d, 12)** e(b, ∞)	
c(d, 12)	e(d, 18)	

Thus from source a to all other remaining vertices the shortest paths are-

a to b	(a-d-b)	Path = 9
a to c	(a-d-c)	Path = 12
a to d	(a-d)	Path = 7
a to e	(a-d-c-e)	Path = 18

Review Questions

1. *Give the general method of Greedy algorithm.*
2. *What do you mean by feasible solution?*
3. *Explain how job sequencing with deadline can be solved using Greedy approach.*
4. *Consider n=7, m=15 (p1,p2,...p7) = (10,5,15,7,6,18,3) and (w1,w2,...w7)=(2,3,5,7,1,4,1). Obtain the optimal solution for this Knapsack instance.*
5. *Give an algorithm for Greedy Knapsack. Analyse your algorithm.*
6. *What do you mean by minimum spanning tree?*
7. *Give an algorithm for computing minimum spanning tree.*
8. *Explain Prim's algorithm with some suitable example.*
9. *Explain Kruskal's algorithm with some suitable example. Also analyze your algorithm.*
10. *Give single source shortest path algorithm. Give the time complexity.*

❑❑❑

4 Dynamic Programming

4.1 Introduction

Dynamic programming is typically applied to optimization problem. This technique is invented by a U.S. Mathematician Richard Bellman in 1950. In the word dynamic programming the word **programming** stands for **planning** and it does not mean by computer programming.

- Dynamic programming is technique for solving problems with overlapping subproblems.
- In this method each subproblem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem.

4.1.1 General Method

Dynamic programming is typically applied to optimization problems.

For each given problem, we may get any number of solutions we seek for optimum solution (i.e. minimum value or maximum value solution). And such an optimal solution becomes the solution to the given problem.

4.2 Difference between Divide and Conquer and Dynamic Programming

Sr. No.	Divide and conquer	Dynamic programming
1.	The problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of subproblems are collected together to get the solution to the given problem.	In dynamic programming many decision sequences are generated and all the overlapping subinstances are considered.

2.	In this method duplications in subsolutions are neglected. i.e., duplicate subsolutions may be obtained.	In dynamic computing duplications in solutions is avoided totally.
3.	Divide and conquer is less efficient because of rework on solutions.	Dynamic programming is efficient than divide and conquer strategy.
4.	The divide and conquer uses top down approach of problem solving (recursive methods).	Dynamic programming uses bottom up approach of problem solving (iterative method).
5.	Divide and conquer splits its input at specific deterministic points usually in the middle.	Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points it determines which split point is optimal.

4.2.1 Comparison between Greedy Algorithm and Dynamic Programming

In this section we will discuss *"What are the differences and similarities between Greedy algorithm and dynamic programming?"*

Sr. No.	Greedy method	Dynamic programming
1.	Greedy method is used for obtaining **optimum solution.**	Dynamic programming is also for obtaining **optimum solution.**
2.	In Greedy method a set of **feasible solutions** and the picks up the optimum solution.	There is **no** special set of **feasible solutions** in this method.
3.	In Greedy method the optimum selection is **without revising previously** generated solutions.	Dynamic programming considers **all possible sequences** in order to obtain the optimum solution.
4.	In Greedy method there is **no** as such **guarantee** of getting **optimum solution.**	It is guaranteed that the dynamic programming will generate optimal solution using **principle of optimality.**

4.2.2 Steps of Dynamic Programming

Dynamic programming design involves 4 major steps :

1. Characterize the structure of optimal solution. That means develop a mathematical notation that can express any solution and subsolution for the given problem.
2. Recursively define the value of an optimal solution.
3. By using bottom up technique compute value of optimal solution. For that you have to develop a recurrence relation that relates a solution to its subsolutions, using the mathematical notation of step 1.
4. Compute an optimal solution from computed information.

4.2.3 Principle of Optimality

The dynamic programming algorithm obtains the solution using principle of optimality.

The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal."

When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.

For example : Finding of shortest path in a given graph uses the principle of optimality.

4.3 Multistage Graphs

A multistage graph G = (V, E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k ≥ 2. In multistage graph problem we have to find the shortest path from source to sink. The cost of each path is calculated by using the weight given along that edge. The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path. In multistage graph problem we have to find the path from S to T. There is set of vertices in each stage. The multistage graph can be solved using forward and backward approach. Let us solve multistage problem for both the approaches with the help of some example.

Consider the graph G as shown in the Fig. 4.1.

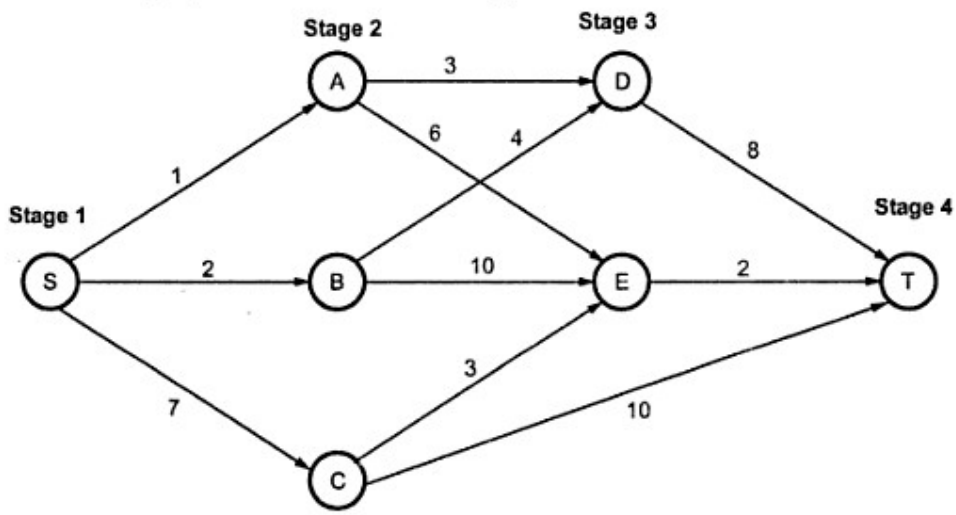


Fig. 4.1 Multistage graph

There is single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

$$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\} \quad \ldots (1)$$

We will now compute d(A, T), d(B, T) and d(C,T).

$$d(A, T) = \min \{3 + d(D, T), 6 + d(E, T)\} \quad \ldots (2)$$

$$d(B, T) = \min \{4 + d(D, T), 10 + d(E, T)\} \quad \ldots (3)$$

$$d(C, T) = \min \{3 + d(E, T), d(C, T)\} \quad \ldots (4)$$

Now let us compute d(D, T) and d(E, T).

$$\mathbf{d(D, T) = 8}$$

$$\mathbf{d(E, T) = 2} \qquad \textbf{backward vertex = E}$$

Let us put these values in equations (2), (3) and (4).

$$d(A, T) = \min \{3 + 8, 6 + 2\}$$

$$\mathbf{d(A, T) = 8} \qquad \textbf{A - E - T}$$

$$d(B, T) = \min \{4 + 8, 10 + 2\}$$

$$\mathbf{d(B, T) = 12} \qquad \textbf{A - D - T}$$

$$d(C, T) = \min \{3 + 2, 10\}$$

$$\mathbf{d(C, T) = 5} \qquad \textbf{C - E - T}$$

$$\therefore \quad d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\}$$

$$= \min \{1 + 8, 2 + 12, 7 + 5\}$$

$$= \min \{9, 14, 12\}$$

$$d(S, T) = 9 \qquad \textbf{S - A - E - T}$$

The path with minimum cost is S - A - E - T with the cost 9.

This method is called **backward reasoning**. The algorithm for this method is as follows.

```
Algorithm Backward_Gr (G, stages, n, p)
  // Problem description : This algorithm is for backward
  // approach of multistage graph
  // Input : The multistage graph G = (V, E),
  // 'stages' is for total number of stages
  // n is total number of vertices of G
  // p is an array for restoring path
  // Output : The path with minimum cost.
      back_cost[i] ← 0
```

```
for (i ← 0 to n – 2) do
{
   r ← Get_Min (i,n)  // r is edge with min cost
   back_cost[i] ← back_cost[r] + c[r][i]
   d[i] ← r
}
// finding minimum cost path
   p[0] ← 0
   p[stages – 1] ← n – 1
   for (i ← stages – 1 to 1) do
      p[i] ← d[p[i + 1]];
```

Assigning minimum distance in 'd array'

Computing backward cost of each vertex

Analysis Clearly the above algorithm has a time complexity $\Theta(|V| + |E|)$.

Forward approach

$d(S, A) = 1$

$d(S, B) = 2$

$d(S, C) = 7$

$d(S, D) = \min \{1 + d(A, D), 2 + d(B, D)\}$

$= \min \{1 + 3, 2 + 4\}$

$\mathbf{d(S, D) = 4}$

$d(S, E) = \min \{1 + d(A, E), 2 + d(B, E), 7 + d(C, E)\}$

$= \min \{1 + 6, 2 + 10, 7 + 3\}$

$= \min \{7, 12, 10\}$

$\mathbf{d(S, E) = 7}$ **i.e. Path S - A - E is chosen.**

$d(S, T) = \min \{d(S, D) + d(D, T), d(S, E) + d(E, T), d(S, C) + d(C, T)\}$

$= \min \{4 + 8, 7 + 2, 7 + 10\}$

$d(S, T) = 9$ **i.e. Path S - E, E - T is chosen.**

$\therefore$ The minimum cost = 9 with the path **S - A - E - T.**

This method is called **forward reasoning.**

The algorithm for forward approach is –

```
Algorithm Forward_Gr (G, stages, n, p)
// Problem description : This algorithm is for
// forward  approach of multistage graph
// Input : The multistage graph G = (V, E),
```

```
// 'Stages' is the variable representing number of stages
// n is total number of vertices of G
// p is an array for restoring path
// Output : The path with minimum cost
        cost[i] ← 0
    for (i ← n – 2 downto 0)
    {
        r ← Get_min (i,n)  // r is an edge with min cost
        cost[i] ← c[i][r] + cost[r]
            d[i] ← r
    }
    // finding minimum cost path
        p[0] ← 0
        p[stages – 1] ← n – 1
    for (i ← 1 to stages–1) do
        p[i] ← d[p[i+1]] ;
```

Analysis : The algorithm has $\Theta(|V|+|E|)$ time complexity.

Using dynamic programming strategy, the multistage graph problem is solved. This is because in multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage. Thus the sequence of decisions are taken by considering overlapped solution. In dynamic programming, we may get any number of solutions for given problem. From all these solutions we seek for optimal solution, finally optimal solution becomes the solution to given problem. Multistage graph problem is solved using this same approach.

4.4 All Pairs Shortest Paths

Problem Description

When a weighted graph, represented by its weight matrix W then objective is to find the distance between every pair of nodes.

We will apply dynamic programming to solve the all pairs shortest path.

Step 1 : We will decompose the given problem into subproblems.

Let,

$A^k_{(i,j)}$ be the length of shortest path from node i to node j such that the label for every intermediate node will be <= k.

We will compute A^k for k = 1 ..... n for n nodes.

Step 2 : For solving all pair shortest path, the principle of optimality is used. That means any subpath of shortest path is a shortest path between the end nodes. Divide the paths from i node to j node for every intermediate node, say 'k'. Then there arises two cases.

i) Path going from i to j via k.

ii) Path which is not going via k. Select only shortest path from two cases.

Step 3 : The shortest path can be computed using bottom up computation method. Following is recursion method.

Initially : $A^0 = W[i, j]$

Next computations :

$$A^k_{(i,j)} = \min\left\{ A^{k-1}_{(i,j)}, A^{k-1}_{(i,k)} + A^{k-1}_{(k,j)} \right\} \quad \text{where } 1 \le k \le n$$

Example 4.1 : *Compute all pair shortest path for the following graph.*

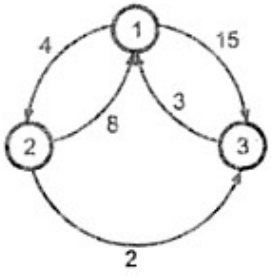


Fig. 4.2 Graph G

Solution :

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & (7) & 0 \end{bmatrix} \leftarrow \min(\infty, (3+4))$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & (6) \\ 8 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \leftarrow \min(15, (4+2))$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ (5) & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

A^3 gives shortest distances between any pair of vertices.

Example 4.2 : *Obtain all pair shortest paths for following graph.*

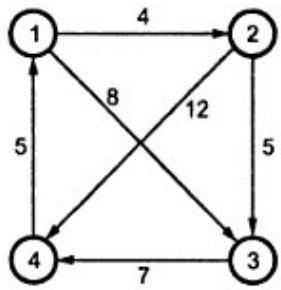


Fig. 4.3

Solution :

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ \infty & \infty & 0 & 7 \\ 5 & \infty & \infty & 0 \end{bmatrix}$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ 12 & \infty & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 16 \\ 17 & 0 & 5 & 12 \\ 12 & \infty & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 16 \\ 17 & 0 & 5 & 12 \\ 12 & 16 & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

Thus shortest distances between all pairs is obtained.

4.4.1 Algorithm

```
Algorithm All_Pair(W,A)
{
  for i=1 to n do
      for j=1 to n do
        A[i,j] := W[i,j]; //copy the weights as it is in matrix A
  for k=1 to n do
  {
   for i=1 to n do
       {
        for j=1 to n do
        {
         A[i,j]=min ( A[i,j],A[i,k] + A[k,j]);
        }
       }
   }
 }
```

Analysis

From above algorithm

The first double for-loop takes $O(n^2)$ time.

The nested three for loops take $O(n^3)$ time.

Thus, the whole algorithm takes $O(n^3)$ time.

4.5 Single Source Shortest Paths

If we want to travel from Mumbai to Bangalore then on giving the road map of India, we can find out the routes between these two cities. If we want to find the shortest path between these two cities then we find out all possible routes along with their distances of Mumbai to Bangalore. And we will select the route with minimum distance. But if we choose a distance from Mumbai to Jaipur and then from Jaipur to Bangalore then of course it will be a poor choice. Because Jaipur is far away from Bangalore. These types of problems in computer science are solved by graph theory. We can define shortest path. weight from a to b by :

$$\delta(a, b) = \begin{cases} \min \{path\ (a,b)\} & \text{where path exist} \\ \infty & \text{between a and b otherwise} \end{cases}$$

Basically the shortest path has two variants.

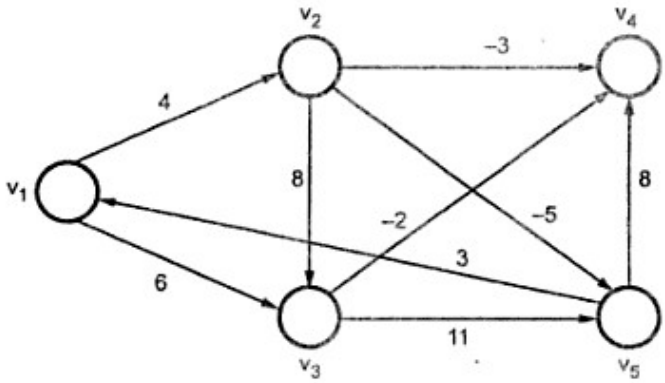
Single-pair shortest path - In these type of problems we have to find a shortest path from a to b for given vertices a and b.

All-pair shortest path - In these type of problems we have to find a shortest path from every pair of a and b.

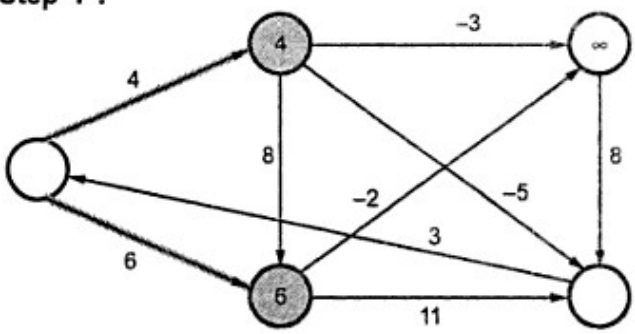
Let us now discuss single-pair shortest path algorithms.

1. The Bellman-Ford Algorithm

The Bellman-Ford algorithm works for finding single source shortest path. It works for **negative** edge weights. Let us understand this algorithm with the help of some example. Consider following for which the shortest path can be obtained from source vertex v_1.

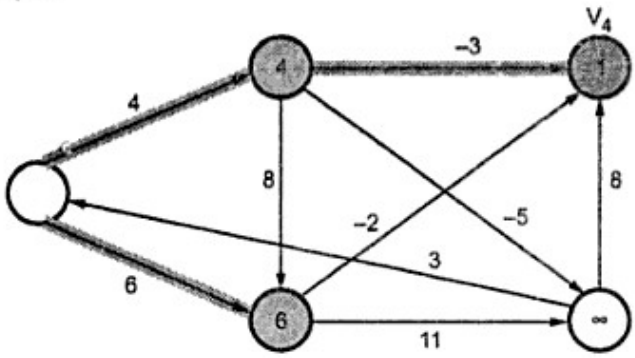


Step 1 :



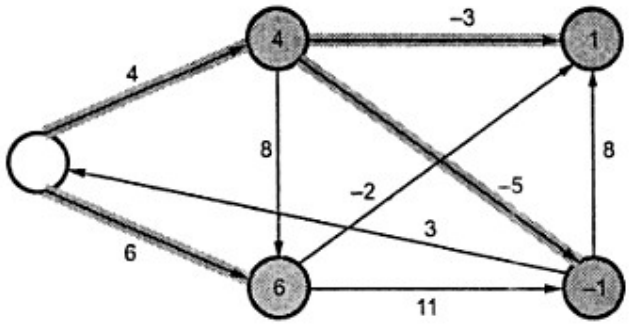
From vertex v_1 we find the distances to v_2 and v_3. The direct edges corresponds to the weights of destination vertices.

Step 2



Modify vertex v_4 by its minimum weight i.e. 1. The path is shown by shaded area.

Step 3 :



Process vertex v_5 by its minimum weight –1. The path is shown by shaded area.

Thus minimum distance of each vertex is obtained.

Algorithm

```
Algorithm Bellman Ford (vertices, edges, source)
 {
// Problem Description : This algorithm finds
// the shortest path using Bellman Ford method
for (each vertex v)
{
   if (v is source) then
   v distance ← 0
   else
   v. distance ← infinity
   v. prede ← Null
}
for (i←1 to toal_ vertices – 1)
{
   for (each edge uv)
   {
   U ← uv. source
   V ← uv. desination
   if (v. distance > u. distance + Uv. weight )   then
   {
      v. distance ← u. distance + Uv. weight
      v. prede ← u
   }
}
for (each edge uv)
{
   u ← uv. source
```

Graph initialization

Newly obtained minimum distance

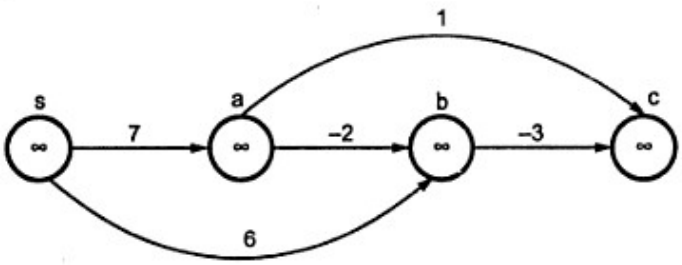
Relaxing edges

```
        v ← uv. destination
        if (v. distance > u.distance + uv. weight) then
        {
            Write ("Graph has negative edges")
            return False
        }
    }
    } // end of for return True
    } // end of algorithm
```

In above algorithm, we have used a term "**relaxing edges**". The process of relaxing edge (u, v) means testing whether we can get more shorter distance to vertex v from u. The **relaxation** step **decrease** the value of the shortest path estimated earlier and modify the predecessors of vertex v. The running time of Bellman-Ford algorithm is **O(nm)**.

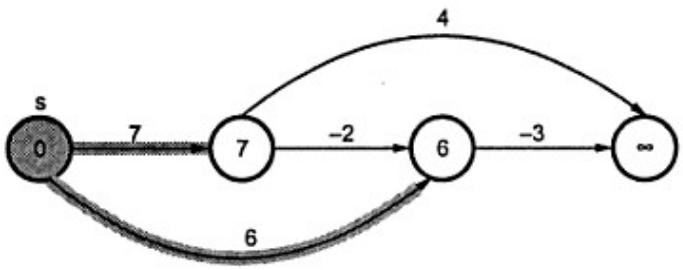
2. Single Source Shortest Paths in Directed a Cyclic Graphs (DAG)

The edges in weighted DAG are **relaxed** according to **topological sort,** of its vertices. And thereby the minimum distance at each vertex from single source is obtained. In DAG, even if negative weighted edges are present, we can compute the single source shortest path. But there should not be any negative weight cycle. Let us find out shortest path in DAG.

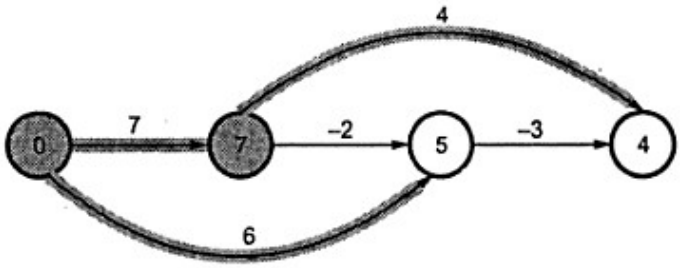


This is a DAG for finding out single source shortest path from source vertex s to a, b and c. Initially all distances are marked as ∞. Let compute shortest path from vertex s to a and b.

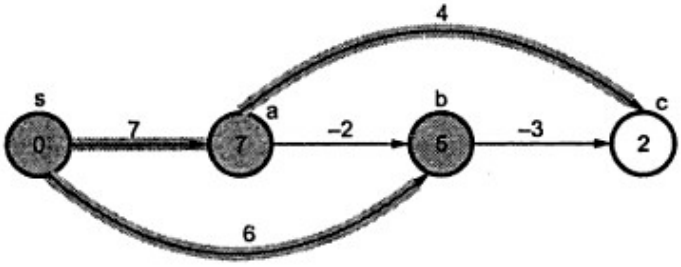
Step 1 :



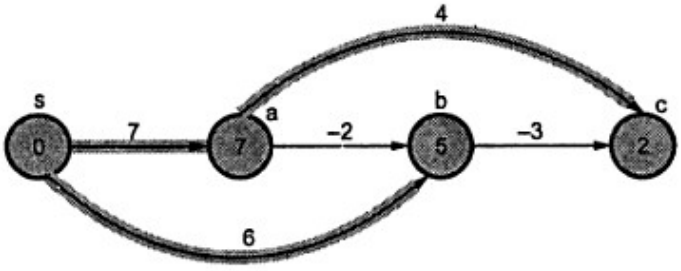
Step 2 :



Step 3 :



Step 4 :



Thus minimum distance at each vertex from source s is obtained. Let us now discuss the algorithm for the above given method.

```
Algorithm DAG_SSP (vertices, edges, s)
{
// Problem Description : This algorithm finds the
// shortest path for DAG
for (each vertex v)
topologically sort the vertices of graph.
for (each vertex v)
{
if (v is s) then
v. distance ← 0          Initialize single source
else
```

```
v. distance ← infinity
v. prede ← NULL
}
for (each vertex u, taken in topologically sorted order)
{
for (each vertex v ∈ Adjacent[u])
{
if (v. distance > u. distance + uv. weight) then
v. distance ← u. distance + uv. weight
v. prede ← u
} // end of for loop
} // end of algorithm.
```

Relaxation

The total running time of above algorithm is $\Theta(v+E)$.

Usefulness of this algorithm

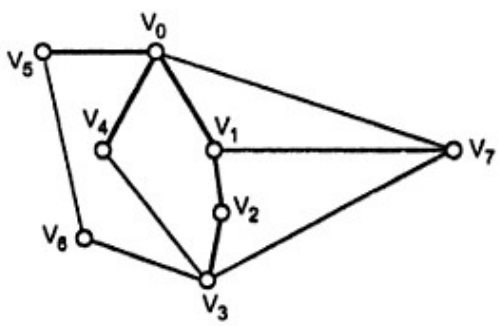
This algorithm is useful in determining critical paths in **PERT chart**. The PERT chart is a chart made for program evaluation and review technique.

3. Dijkstra's Algorithm

Dijkstra's algorithm is a popular algorithm for finding shortest path. This algorithm is called **single source shortest path** algorithm. In this algorithm, for a given vertex called source the shortest path to all other vertices is obtained. In this algorithm the main focus is not to find only one single path but to find the shortest paths from any vertex to all other remaining vertices.

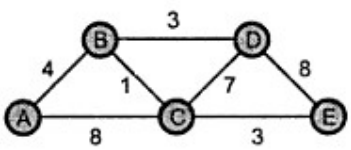
This algorithm applicable to graphs with non-negative weights only.

Dijkstra's algorithm finds shortest paths to graph's vertices in order of their distance from a given source. In this process of finding shortest path, first it finds the shortest path from the source to a vertex nearest to it, then second nearest and so on.

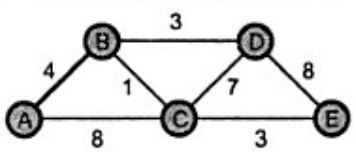
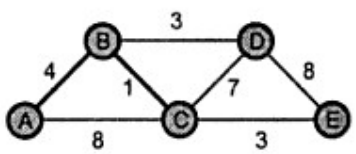
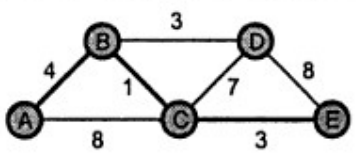


The shortest path from V_0 is obtained. First we find shortest path from V_0 - V_1 then V_1 - V_2 then from V_2 - V_3 the shortest distance is obtained. Let us understand this algorithm with some example.

Consider a weighted connected graph as given below.



Now we will consider each vertex as a source and will find the shortest distance from this vertex to every other remaining vertex. Let us start with vertex A.

Source vertex	Distance with other vertices	Path shown in graph
A	**A-B, path = 4** A-C, path = 8 A-D, path = ∞ A-E, path = ∞	
B	**B-C, path = 4 + 1** B-D, path = 4 + 3 B-E, path = ∞	
C	C-D, path = 5 + 7 = 12 **C-E, path = 5 + 3 = 8**	
D	D-E, path = 7 + 8 = 15	

But we have one shortest distance obtained from A to E and that is A - B - C - E with path length = 4 + 1 + 3 = 8. Similarly other shortest paths can be obtained by choosing appropriate source and destination.

Algorithm Dijkstra(int cost[1...n,1...n],int source,int dist[])

```
for i ← 0 to tot_nodes
{
    dist[i] ← cost[source,i]//initially put the
    s[i] ← 0 //distance from source vertex to i
```

```
        //i is varied for each vertex
        path[i] ← source//all the sources are put in path
    }
      s[source] ← 1
    for(i ← 1 to tot_nodes)
    {
        min_dist ← infinity;
        v1 ← -1//reset previous value of v1
        for(j ← 0 to tot_nodes-1)
        {
            if(s[j]=0)then
            {
               if(dist[j]<min_dist)then
               {
                min_dist ← dist[j]
                v1 ← j
               }
            }
        }
        s[v1] ← 1
        for (v2 ← 0 to tot_nodes-1)
        {
         if(s[v2]=0) then
         {
            if(dist[v1]+cost[v1][v2]<dist[v2]) then
            {
               dist[v2] ← dist[v1]+cost[v1][v2]
               path[v2] ← v1
            }
         }
        }
      }
}
```

Start from each source node

Finding minimum distance from selected source node.That is : source-j represents min. dist. edge

v_1 is next selected destination vertex with shortest distance. All such vertices are accumulated in array path[ ]

C function

```
void Dijkstra(int tot_nodes,int cost[10][10],int source,int
dist[])
{
 int i,j,v1,v2,min_dist;
 int s[10];
 for(i=0;i<tot_nodes;i++)
 {
       dist[i]=cost[source][i];//initially put the
       s[i]=0; //distance from source vertex to i
       //i is varied for each vertex
       path[i]=source;//all the sources are put in path
 }
 s[source]=1;
 for(i=1;i<tot_nodes;i++)
 {
       min_dist=infinity;
       v1=-1;//reset previous value of v1
       for(j=0;j<tot_nodes;j++)
       {
        if(s[j]==0)
        {
          if(dist[j]<min_dist)
          {
          min_dist=dist[j];//finding minimum distance
           v1=j;
          }
        }
       }
       s[v1]=1;
       for(v2=0;v2<tot_nodes;v2++)
       {
        if(s[v2]==0)
        {
          if (dist [v1]+cost [v1] [v2] <dist [v2])
          {
             dist [v2] =dist [v1] + cost [v1][v2] ;
             path [v2] = v1 ;
          }
        }
      }
  }
}
```

C Program

```
/*************************************************************
This program is for implementing Dijkstra's single source shortest
path algorithm.
*************************************************************/
  #include<stdio.h>
  #include<conio.h>
  #define infinity 999
  int path[10];
  void main()
  {
   int tot_nodes,i,j,cost[10][10],dist[10],s[10];
   void create(int tot_nodes,int cost[][10]);
   void Dijkstra(int tot_nodes,int cost[][10],int i,int dist[10]);
   void display(int i,int j,int dist[10]);
   clrscr();
   printf("\n\t\t Creation of graph ");
   printf("\n Enter total number of nodes ");
   scanf("%d",&tot_nodes);
   create(tot_nodes,cost);
   for(i=0;i<tot_nodes;i++)
   {
        printf("\n\t\t\t Press any key to continue...");
        printf("\n\t\t When Source =%d\n",i);
        for(j=0;j<tot_nodes;j++)
        {
         Dijkstra(tot_nodes,cost,i,dist);
         if(dist[j]==infinity)
            printf("\n There is no path to %d\n",j);
         else
         {

            display(i,j,dist);
         }
        }
    }
  }
  void create(int tot_nodes,int cost[][10])
  {
   int i,j,val,tot_edges,count=0;
```

```
for(i=0;i<tot_nodes;i++)
{
      for(j=0;j<tot_nodes;j++)
      {
       if(i==j)
          cost[i][j]=0;//diagonal elements are 0
       else
          cost[i][j]=infinity;
      }
}
printf("\n Total number of edges");
scanf("%d",&tot_edges);
while(count<tot_edges)
      {
       printf("\n Enter Vi and Vj");
       scanf("%d%d",&i,&j);
       printf("\n Enter the cost along this edge");
       scanf("%d",&val);
       cost[j][i]=val;
       cost[i][j]=val;
       count++;
      }
}
void Dijkstra(int tot_nodes,int cost[10][10],int source,int
dist[])
{
 int i,j,v1,v2,min_dist;
 int s[10];
 for(i=0;i<tot_nodes;i++)
 {
      dist[i]=cost[source][i];//initially put the
      s[i]=0; //distance from source vertex to i
      //i is varied for each vertex
      path[i]=source;//all the sources are put in path
 }
 s[source]=1;
 for(i=1;i<tot_nodes;i++)
 {
```

```
        min_dist=infinity;
        v1=-1;//reset previous value of v1
        for(j=0;j<tot_nodes;j++)
        {
         if(s[j]==0)
         {
            if(dist[j]<min_dist)
            {
             min_dist=dist[j];//finding minimum distance
             v1=j;
            }
         }
        }
        s[v1]=1;
        for(v2=0;v2<tot_nodes;v2++)
        {
         if(s[v2]==0)
         {
            if(dist[v1]+cost[v1][v2]<dist[v2])
            {
               dist[v2]=dist[v1]+cost[v1][v2];
               path[v2]=v1;
            }
         }
        }
    }
}
void display(int source,int destination,int dist[])
{
 int i;
 getch();
 printf("\n Step by Step shortest path is...\n");
 for(i=destination;i!=source;i=path[i])
 {
       printf("%d<-",i);
 }
 printf("%d",i);
 printf("  The length=%d",dist[destination]);
}
```

Output

Creation of graph

Enter total number of nodes 5

Total number of edges 7

Enter Vi and Vj 0 1

Enter the cost along this edge 4

Enter Vi and Vj 0 2

Enter the cost along this edge 8

Enter Vi and Vj 1 2

Enter the cost along this edge 1

Enter Vi and Vj 1 3

Enter the cost along this edge 3

Enter Vi and Vj 2 3

Enter the cost along this edge 7

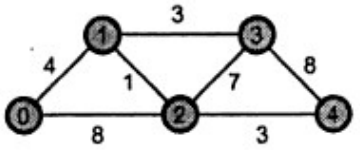
Enter Vi and Vj 2 4

Enter the cost along this edge 3

Enter Vi and Vj 3 4

Enter the cost along this edge 8

The input graph is



Press any key to continue...

When Source =0

Step by Step shortest path is...

```
0 The length=0
 Step by Step shortest path is...
1<-0 The length=4
 Step by Step shortest path is...
2<-1<-0 The length=5
 Step by Step shortest path is...
3<-1<-0 The length=7
 Step by Step shortest path is...
4<-2<-1<-0 The length=8
            Press any key to continue...
          When Source =1

 Step by Step shortest path is...
0<-1 The length=4
 Step by Step shortest path is...
1 The length=0
 Step by Step shortest path is...
2<-1 The length=1
 Step by Step shortest path is...
3<-1 The length=3
 Step by Step shortest path is...
4<-2<-1 The length=4
            Press any key to continue...
        When Source =2

 Step by Step shortest path is...
0<-1<-2 The length=5
 Step by Step shortest path is...
1<-2 The length=1
 Step by Step shortest path is...
2 The length=0
 Step by Step shortest path is...
3<-1<-2 The length=4
 Step by Step shortest path is...
4<-2 The length=3
            Press any key to continue...
        When Source =3
```

```
 Step by Step shortest path is...
0<-1<-3  The length=7
 Step by Step shortest path is...
1<-3  The length=3
 Step by Step shortest path is...
2<-1<-3  The length=4
 Step by Step shortest path is...
3  The length=0
 Step by Step shortest path is...
4<-2<-1<-3  The length=7
          Press any key to continue...
        When Source =4

 Step by Step shortest path is...
0<-1<-2<-4  The length=8
 Step by Step shortest path is...
1<-2<-4  The length=4
 Step by Step shortest path is...
2<-4  The length=3
 Step by Step shortest path is...
3<-1<-2<-4  The length=7
 Step by Step shortest path is...
4  The length=0
```

4.6 Optimal Binary Search Tree (OBST)

Suppose we are searching a word from a dictionary. And for every required word, we are looking up in the dictionary then it becomes time consuming process. To perform this lookup more efficiently we can build the binary search tree of common words as key elements. Again we can make this binary search tree efficient by arranging frequently used words nearer to the root and less frequently words away from the root. Such a binary search tree makes our task more simplified as well as efficient. This type of binary search tree is also called optimal binary search tree (OBST).

Problem Description

Let $\{a_1, a_2, \ldots a_n\}$ be a set of identifiers such that $a_1 < a_2 < a_3$. Let p(i) be the probability with which we can search for a_i and q(i) be the probability of searching element x such that $a_i < x < a_{i+1}$ and $0 \le i \le n$. In other words p(i) is the probability of successful search and q(i) be the probability of unsuccessful search. Also $\sum_{1 \le i \le n} p(i) + \sum_{1 \le i \le n} q(i)$ then obtain a tree with minimum cost. Such a tree with optimum cost is called optimal binary search tree.

To solve this problem using dynamic programming method we will perform following steps.

Step 1 : Notations used

Let,

$T_{ij} = OBST(a_{i+1}, \ldots, a_j)$

C_{ij} denotes the cost(T_{ij}).

W_{ij} is the weight of each T_{ij}.

T_{0n} is the final tree obtained.

T_{00} is empty.

$T_{i,i+1}$ is a single-node tree that has element a_{i+1}.

During the computations the root values are computed and r_{ij} stores the root value of T_{ij}.

Step 2 : The OBST can be build using the principle of optimality. Consider the process of creating OBST.

- Let T_{0n} be an OBST for the elements $a_1 < a_2 < \ldots < a_n$, and let L and R be its left subtree and right subtree. Suppose that the root of T_{0n} is a_k, for some k.
- Then the elements in the left subtree L are $a_1, a_2, \ldots, a_{k-1}$ and the elements in the right subtree R are $a_{k+1}, a_{k+2}, \ldots, a_n$.
- The cost of computing the T_{0n} can be given as

$$C(T_{0n}) = C(L) + C(R) + p_1 + p_2 + \ldots + p_n + q_0 + q_1 + q_2 + \ldots + q_n$$

i.e. $$C(T_{0n}) = C(L) + C(R) + W$$

where

$$W = p_1 + p_2 + \ldots + p_n + q_0 + q_1 + q_2 + \ldots + q_n$$

- If L is not an optimal BST for its elements, then we can find another tree L' for the same elements, with the property C(L') < C(L) (i.e. optimal cost).

  Let T' be the tree with root a_k, left subtree L' and right subtree R. Then

$$C(T') = C(L') + C(R) + W$$

  i.e. $$C(T') < C(L) + C(R) + W$$

  i.e. $$< C(T_{0n})$$

- That means ; T' is optimal than T_{0n}. This contradicts the fact that T_{0n} is an optimal BST. Therefore, L must be an optimal for its elements.
- In the same manner we can obtain optimal tree for R.
- Thus we can obtain the optimal binary search tree by building the optimal subtrees. This ultimately shows that optimal binary search tree follows the principle of optimality.

Step 3 : We will apply following formula for computing each sequence.

$$C(i, j) = \min_{i<k\le j} \{C(i, k-1) + C(k, j)\} + W(i, j)\}$$

$$W(i, j) = W[i, j-1] + p[j] + q[j];$$

$$r[i, j] = k$$

Example : Consider n = 4 and (q_1, q_2, q_3, q_4) = (do, if, int, while). The values for p's and q's are given as p(1:4) = (3, 3, 1, 1) and q(0:4) = (2, 3, 1, 1, 1). Construct the optimal binary search tree. We will apply following formulae for the computation of W, C and r.

$$W_{i,i} = q_i,\ r_{i,i} = 0,\ C_{i,i} = 0$$

$$W_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$r_{i,i+1} = i + 1$$

$$C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$W_{i,j} = W_{i,j-1} + p_j + q_j$$

$$r_{i,j} = k$$

$$C_{i,j} = \min_{i< k\le j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

We will construct tables for values of W, C and r.

Let $i = 0$

$W_{00} = q_0 = 2$

When i = 1

$W_{11} = 3$

When i = 2

$W_{22} = 1$

When i = 3

$W_{33} = 1$

When i = 4

$W_{44} = q_4 = 1$

When i = 0 and j – i = 1 then

$W_{01} = q_0 + q_1 + p_1$

$= 2 + 3 + 3$

$W_{01} = 8$

When i =1 and j – i = 2

$W_{12} = q_1 + q_2 + p_2$

$= 3 + 1 + 3$

$W_{12} = 7$

When i = 2 and j – i = 3

$W_{23} = q_2 + q_3 + p_3$

$= 1 + 1 + 1$

$W_{23} = 3$

When i = 3 and j – i = 4

$W_{34} = q_3 + q_4 + p_4$

$= 1 + 1 + 1$

$W_{34} = 3$

Now, when i = 0 and j – i = 2

$$W_{ij} = W_{i,j-1} + p_j + q_j$$

$$W_{02} = W_{01} + p_2 + q_2$$

$$= 8 + 3 + 1$$

$$W_{02} = 12$$

When i =1 and j – i = 2

$$W_{13} = W_{12} + p_3 + q_3$$

$$= 7 + 1 + 1$$

$$W_{13} = 9$$

When i = 2 and j – i = 2 then

$$W_{24} = W_{23} + p_4 + q_4$$

$$= 3 + 1 + 1$$

$$W_{24} = 5$$

Now, when i = 0 and j – i = 3 then

$$W_{03} = W_{02} + p_3 + q_3$$

$$= 12 + 1 + 1$$

$$W_{03} = 14$$

When i = 1 and j – i = 3 then

$$W_{14} = W_{13} + q_4 + p_4$$

$$= 9 + 1 + 1$$

$$W_{14} = 11$$

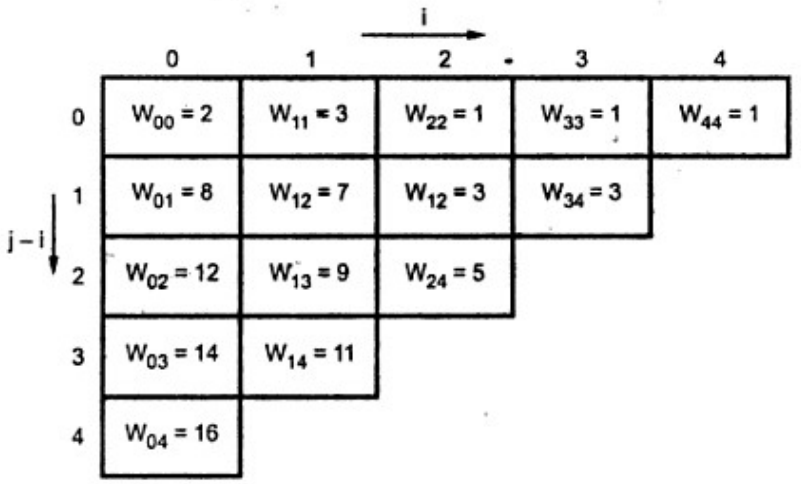
When i = 0 and j – i = 4 then

$$W_{04} = W_{03} + q_4 + p_4$$

$$= 14 + 1 + 1$$

$$W_{04} = 16$$

The table for W can be represented as



We will now compute for C and r.

As $C_{i,i} = 0$ and $r_{i,i} = 0$

$C_{00} = 0$ $C_{11} = 0$ $C_{22} = 0$ $C_{33} = 0$ $C_{44} = 0$

$r_{00} = 0$ $r_{11} = 0$ $r_{22} = 0$ $r_{33} = 0$ $r_{44} = 0$

Similarly, $C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$ and $r_{i,i+1} = i+1$

∴When i = 0

$$C_{01} = q_0 + q_1 + p_1$$
$$= 2 + 3 + 3$$
$$C_{01} = 8$$
$$r_{01} = 1$$

When $i = 1$

$$C_{12} = q_1 + q_2 + p_2$$
$$= 3 + 1 + 3$$
$$C_{12} = 7 \quad \text{and} \quad r_{12} = 2$$

When $i = 2$

$$C_{23} = q_2 + q_3 + p_3$$
$$= 1 + 1 + 1$$
$$C_{23} = 3 \quad \text{and} \quad r_{23} = 3$$

When $i = 1$

$$C_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$
$$C_{34} = 3 \quad \text{and} \quad r_{34} = 4$$

Now we will compute C_{ij} and r_{ij} for $j - 1 \geq 2$.

As $$C_{i,j} = \min_{i < k \leq j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

Hence we will find k.

For C_{02} we have $i = 0$ and $j = 2$. For $r_{i,j-1}$ to $r_{i+1,j}$ i.e. for r_{01} to $r_{1,2}$. We will compute minimum value of C_{ij}.

Let $r_{01} = 1$ and $r_{12} = 2$. Then we will assume value of $k = 1$ and will compute C_{ij}. Similarly with $k = 2$ we will compute C_{ij} and will pick up minimum value of C_{ij} only.

Let us compute C_{ij} with following formula,

$$C_{ij} = C_{i,k-1} + C_{kj}$$

For $k = 1$, $i = 0$, $j = 2$.

$$C_{02} = C_{00} + C_{12}$$
$$C_{02} = 0 + 7$$
$$C_{02} = 7 \quad \ldots(1)$$

For $k = 2$, $i = 0$, $j = 2$

$$C_{02} = C_{01} + C_{22}$$
$$= 8 + 0$$
$$C_{02} = 8 \quad \ldots(2)$$

From equations (1) and (2) we can select minimum value of C_{02} is 7. That means $k = 1$ gives us minimum value of $C_{i,j}$.

Hence $$r_{ij} = r_{02} = k = 1$$

Now

$$C_{02} = \min\{C_{(i,k-1)} + C_{kj}\} + W_{ij}$$
$$= 7 + W_{02}$$
$$= 7 + 12$$
$$C_{02} = 19$$

Continuing in this fashion we can compute C_{ij} and r_{ij}. It is as given below.

j−i \ i	0	1	2	3	4
0	$C_{00}=0$, $r_{00}=0$	$C_{11}=0$, $r_{11}=0$	$C_{22}=0$, $r_{22}=0$	$C_{33}=0$, $r_{33}=0$	$C_{44}=0$, $r_{44}=0$
1	$C_{01}=8$, $r_{01}=1$	$C_{12}=7$, $r_{12}=2$	$C_{23}=3$, $r_{23}=3$	$C_{34}=3$, $r_{34}=4$	
2	$C_{02}=19$, $r_{02}=1$	$C_{13}=12$, $r_{13}=2$	$C_{24}=8$, $r_{24}=3$		
3	$C_{03}=25$, $r_{03}=2$	$C_{14}=19$, $r_{14}=2$			
4	$C_{04}=32$, $r_{04}=2$				

Therefore T_{04} has a root r_{04}. The value of r_{04} is 2. From (a_1, a_2, a_3, a_4) = (do, if, int, while) a_2 becomes the root node.

$$r_{ij} = k$$

$$r_{04} = 2$$

Then r_{ik-1} becomes left child and r_{kj} becomes the right child. In other words r_{01} becomes the left child and r_{24} becomes right child of r_{ij}. Here $r_{01} = 1$ so a_1 becomes left child of a_2 and $r_{24} = 3$ so a_3 becomes right child of a_2.

For the node r_{24} i = 2, j = 4 and k =3. Hence left child of it is $r_{ik-1} = r_{22} = 0$. That means left child of $r_{24} = a_3$ is empty. The right child of r_{24} is $r_{34} = 4$. Hence a_4 becomes right child of a_3. Thus all n = 4 nodes are used to build a tree. The optimal binary search tree is

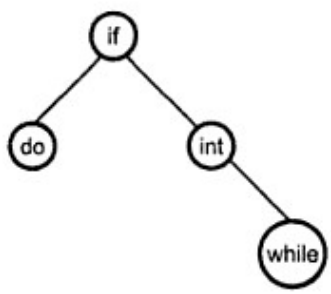


Fig. 4.4 OBST

4.6.1 Algorithm

```
Algorithm Wt(p,q,W,n)
//p is an input array [1..n]
//q is an input array [0..n]
// W[i,j] will be output for this function such that W[0..n,0..n]
{
  for i←1 to n do
    W[i,i] ← q[i]; // initialize
  for m←1 to n do
  {
   for I←0 to n-m do
    {
      k ← i+m;
      W[i,k]←W[i,k-1] + p[k] + q[k];
    }
  }
write(W[0:n]);//print the values of W
}
```

- For computing C_{ij}, r_{ij} following algorithm is used.

```
Algorithm OBST(p,q,W,C,r)
{
  //computation of first two rows
for i=0 to n do
{
      C[i,i] ← 0;
          r[i,i] ← 0;
          C[i,i+1] ← q[i]+q[i+1]+p[i+1];
          r[i,i+1]←i+1;
      }
      for m ← 2 to n do
      {
         For ← 0 to n-m do
{
   j←i+m;
   k←Min_Value(C,r,i,j); //call for algortihm Min_Value
// minimum value of Cij is obtained for deciding value of k
C[i,j]←W[i,j] + C[i,k-1] + C[k,j];
```

```
r[i,j]←k;
}
}
write(C[0:n],r[0:n]);//print values of C_ij and r_ij
}

Algorithm Min_Value(C,r,i,j)
{
   minimum← ∞;
// finding the range of k
   for (m←r[i,j-1] to r[i+1,j]) do
   {
      if(C[i,m-1]+C[m,j])< minimum then
      {
         minimum ← C[i,m-1]+C[m,j];
         k←m;
      }
      return k;//This k gives minimum value of C
   }
}
```

Following algorithm is used for creating the tree T_{ij}.

```
Algorithm build_tree(r,a,i,j)
{
  T ← new(node); //allocate memory for a new node
  k ← r[i,j];
  T -> data ← a[k];
  if (j==i+1)
     return;
  T->left=build_tree(r,a,i,k-1);
  T->right=build_tree(r,a,k,j);
}
```

In order to obtain the optimal binary search tree we will follow :

1. Compute the weight using Wt function.
2. Compute C_{ij} and r_{ij} using OBST.
3. Build the tree using build_tree function.

Analysis

The computation of each C and r can be done using three nested for loops. Hence the time complexity turns out to be $O(n^3)$.

4.7 0/1 Knapsack Problem

Problem Description : If we are given n objects and a Knapsack or a bag in which the object i that has weight w_i is to be placed. The Knapsack has a capacity W. Then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of Knapsack with maximum profit earned.

$$\text{maximized } \sum_{n}^{1} p_i x_i \text{ subject to constraint } \sum_{n}^{1} w_i x_i \leq W$$

where $1 \leq i \leq n$ and n is total number of objects. And $x_i = 0$ or 1

The greedy method does not work for this problem.

To solve this problem using dynamic programming method we will perform following steps.

Step 1 : The notations used are

Let,

$f_i(y_j)$ be the value of optimal solution.

Then S^i is a pair (p, w) where $p = f(y_j)$ and $w = y_j$

Initially $S^0 = \{(0, 0)\}$

We can compute S^{i+1} from S^i

These computations of S^i are basically the sequence of decisions made for obtaining the optimal solutions.

Step 2 : We can generate the sequence of decisions in order to obtain the optimum selection for solving the Knapsack problem.

Let x_n be the optimum sequence. Then there are two instances $\{x_n\}$ and $\{x_{n-1}, x_{n-2} \ldots x_1\}$. So from $\{x_{n-1}, x_{n-2} \ldots x_1\}$ we will choose the optimum sequence with respect to x_n. The selection of sequence from remaining set should be such that we should be able to fulfill the condition of filling Knapsack of capacity W with maximum profit. Otherwise $x_n, \ldots x_1$ is not optimal.

This proves that 0/1 Knapsack problem is solved using principle of optimality.

Step 3 : The formulae that are used while solving 0/1 Knapsack is

Let, $f_i(y_j)$ be the value of optimal solution. Then

$$f_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

Initially compute

$$S^0 = \{(0, 0)\}$$

$$S_1^i = \{(P, W) \mid (P - p_i, W - w_i) S^i\}$$

S_1^{i+1} can be computed by merging S^i and S_1^i

Purging rule

If S^{i+1} contains (P_j, W_j) and (P_k, W_k); these two pairs such that

$P_j \leq P_k$ and $W_j \geq W_k$, then (P_j, W_j) can be eliminated. This purging rule is also called as dominance rule. In purging rule basically the dominated tuples gets purged. In short, remove the pair with less profit and more weight.

Example 4.3 : *Solve Knapsack instance M = 8, and n = 4. Let P_i and W_i are as shown below.*

i	P_i	W_i
1	1	2
2	2	3
3	5	4
4	6	5

Solution : Let us build the sequence of decision S^0, S^1, S^2.

$$S^0 = \{(0, 0)\} \text{ initially}$$

$$S_1^0 = \{(1, 2)\}$$

That means while building S_1^0 we select the next i^{th} pair. For S_1^0 we have selected first (P, W) pair which is (1, 2).

Now

$$S^1 = \{\text{Merge } S^0 \text{ and } S_0^1\}$$

$$= \{(0, 0), (1, 2)\}$$

$$S_1^1 = \{\text{ Select next (P, W) pair and add it with } S^1\}$$

$= \{ (2, 3), (2+0, 3+0), (2+1, 3+2) \}$

$S_1^1 = \{(2, 3), (3, 5)\}$ $\because$ Repetition of (2, 3) is avoided.

$S^2 = \{$Merge candidates from S^1 and $S_1^1\}$

$= \{(0, 0), (1, 2), (2, 3),(3, 5)\}$

$\therefore$ $S_1^2 = \{$Select next (P, W) pair and add it with $S^2\}$

$= \{(5, 4), (6, 6), (7, 7), (8, 9)\}$

Now $S^3 = \{$Merge candidates from S^2 and $S_1^2\}$

$S^3 = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6), (7, 7), (8, 9)\}$

Note that the pair (3, 5) is purged from S^3. This is because, let us assume $(P_j, W_j) = (3, 5)$ and $(P_k, W_k) = (5, 4)$. Here $P_j \leq P_k$ and $W_j > W_k$ is true hence we will eliminate pair (P_j, W_j) i.e. (3, 5) from S^3.

$S_1^3 = \{(6, 5), (7, 7), (8, 8), (11, 9), (12, 11), (13, 12), (14, 14)\}$

$S^4 = \{(0,0), (1,2), (2,3), (5,4),(6,6),(7,7), (8,9)$
$(6,5), (8,8), (11,9), (12,11), (13,12),(14,14)\}$

Now we are interested in M = 8. We get pair (8, 8) in S^4. Hence we will set $x_4 = 1$. Now to select next object $(P - P_4)$ and $(W - W_4)$.

i.e $(8 - 6)$ and $(8 - 5)$.

i.e. (2, 3)

Pair $(2, 3) \in S^2$. Hence set $x_2 = 1$. So we get the final solution as (0, 1, 0, 1).

Example : Consider the Knapsack for the instance n = 4. $(W_1, W_2, W_3, W_4) = (10, 15, 6, 9)$ and $(P_1, P_2, P_3, P_4) = (2, 5, 8, 1)$ and m = 21 then we can generate sequence of decisions.

$S^i = \{P_i, W_i\}$

$S^0 = \{(0, 0)\}$, $S_1^0 = \{(2, 10)\}$

$S^1 = \{(0, 0), (2, 10)\}$; $S_1^1 = \{(5, 15), (7, 25)\}$

$S^2 = \{(0, 0), (2, 10), (5, 15), (7, 25)\}$; $S_1^2 = \{(8, 6), (10, 16), (13, 21), (15, 31)\}$

$S^3 = \{(0, 0), (2, 10), (5, 15), (8, 6), (10, 16), (13, 21) (15, 31) \}$;

$S_1^3 = \{(1, 9), (3, 19), (6, 24), (9, 15), (11, 25), (14, 30), (16, 40)\}$

$S^4 = \{(0, 0), (2, 10), (5, 15), (8, 6), (10, 16), (13, 21) (15, 31), (1, 9) (3, 19) (6, 24),$
$(9, 15), (11, 25), (14, 30) (16, 40)\}$

As pair (7, 25) is eliminated from S^3 because of purging rule.

Now for m = 21 we will search for a tuple in which value of W is 21. We obtain such a tuple (13, 21) in S^3. Hence set $x_3 = 1$.

$\therefore$ We will do (13, 21) – (P_3, W_3).

i.e. (13 – 8) and 21 – 6.

We will get (5, 15). As 5, 15 $\in S^2$ set $x_2 = 1$.

Now (5, 15) – (P_2, W_2)

i.e. (0, 0)

Hence we will terminate by setting remaining x_1 and x_4 to 0. Hence the final solution is (0110). In other words select item x_2 and x_4 for the Knapsack.

4.7.1 Algorithm

```
struct PW
{
 float p;
 float w;
};
Algorithm DKnap(int n,      //number objects for knapsack
        float p1 [ ],       // profits of objects
        float w1[ ],        // weights of objects
        float Total_wt,     // Knapsack capacity
        int x[ ])     // solution indicated by 0s and 1s
{
   struct PW pair[size]; // for storing the tuples (p,w)
   int buffer[MAXSIZE];  // keeps track of locations in S(i)
   int next;             //next free spot in array
   int start;            // start index of S(i-1)
   int end;              // end index of S(i-1)
   int i;                // index of object to be added
   int j; // index for adding next object to tuples in S(i-1)
   int k;                // index for traversing S(i-1)
   int up;       // largest index in  S(i-1)that fits obj
   float new_profit      // profit of new tuple in S(i)
   float new_wt;         // weight of new tuple in S(i)

   //Initially generate the first tuple S(0)= (0,0)
   pair[1].p=pair[1].w=0.0;
```

```
buffer[0]=1;

//Starting at end of S(0)
start=1; end=1;
buffer[1]=next=2;

    // Loop for adding successive objects to knapsack
for (i=1; i<=n-1; i++)
{
  k=start;        //k loops through S(i-1)
      // Return index of largest tuple in S(i-1) that does
      // not overflow     the knapsack up=Big
      // (pair,w1,start,end,i,Total_wt);
      // Add object to next tuple in S(i-1) to form
      // (new_profit,new_wt)
      // and merge S(i-1) tuples into S(i)

 for (j=start; j<=up; j++)
 {
   new_profit=pair[j].p+ p1[i];
   new_wt=pair[j].w+ w1[i];

//Copy the tuples in S(i-1) with less weight than new
//tuple.
    while ((k<=end) AND (pair[k].w<new_wt))
{
    pair[next].p=pair[k].p;
    pair[next].w=pair[k].w;
    next++;
    k++;
}

   //If next tuple has same weight (new_wt) take the one
   //with largest profit then update new_profit
   if((k <= end) && (pair[k].w == new_wt)) then
   {
     if(pair[k].p > new_profit) new_profit=pair[k].p;
     k++;
```

```
        }
        //if new_profit > profit of previous tuple, insert
        //new_profit and new_wt)
          if(new_profit > pair[next-1].p) then
          {
            pair[next].p=new_profit;
            pair[next].w=new_wt;
            next++;
          }
        // All remaining tuples in S(i-1) have weight > new_wt
        //Skip over (pruning rule) any tuples with less profit
            while ((k <= end)
            {
            if((pair[k].p <= pair[next-1].p)
            &&(pair[k].w >= pair[next-1].w))
                k++;
            }
       }//end of inner for loop
       //going through all tuples in S(i-1)that fit object in
      //knapsack. Then Merge in remaining tuples from S(i-1)
            while (k<=end)
            {
               pair[next].p=pair[k].p;
               pair[next].w=pair[k].w;
               next++; k++;
            }

            //Initialize S(i+1) for the next object
            start=end+1;        // set index for start of S(i+1)
            end=next-1;     // set index for end of S(i+1)
            buffer[i+1]=next;
        } //end of outer for loop

    //Trace back the solution.
    Trace_sol (p1,w1,pair,x,Total_wt,n,buffer);
}
```

Analysis

The worst case time complexity of dynamic Knapsack algorithm is $O(2^n)$.

4.8 Traveling Salesperson Problem

Problem Description

Let G be directed graph denoted by (V, E) where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij} . The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.

A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The traveling salesperson problem is to find the tour of minimum cost.

Dynamic programming is used to solve this problem.

Step 1 : Let the function C(1, V–{1}) is the total length of the tour terminating at 1. The objective of TSP problem is that the cost of this tour should be minimum.

Let d[i, j] be the shortest path between two vertices i and j.

Step 2: Let V_1, $V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour. Then (V_1, $V_2, \ldots V_n$) must be a shortest path from V_1 to V_n which passes through each vertex exactly once.

Here the principle of optimality is used. The path V_i, $V_{i+1}, \ldots V_j$ must be optimal for all paths beginning at V(i), ending at V(j), and passing through all the intermediate vertices $\{V_{(i+1)} \ldots V_{(j-1)}\}$ once.

Step 3: Following formula can be used to obtain the optimum cost tour.

$Cost(i, S) = \min \{d[i, j] + Cost(j, S–\{j\})\}$ where $j \in S$ and $i \notin S$.

Consider one example to understand solving of TSP using dynamic programming approach.

Example 4.4 : *For the given diagraph, obtain optimum cost Tour.*

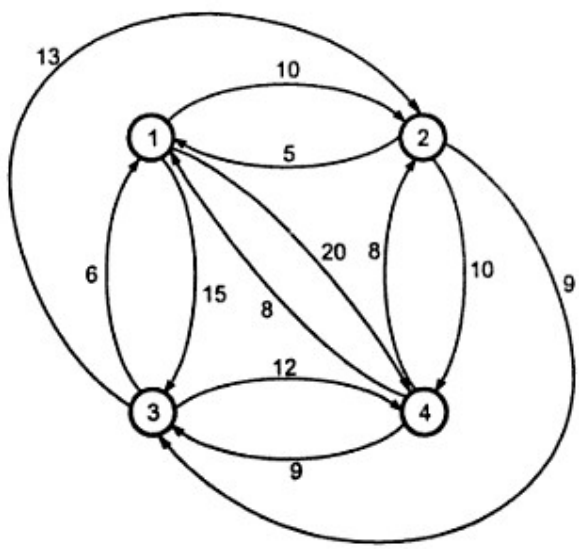


Fig. 4.5

Solution : The distance matrix can be given by,

from ↓ / to →	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

First we will select any arbitrary vertex say select 1.

Now process for intermediate sets with increasing size.

Step 1 : Let $S = \phi$ then,

$$\text{Cost } (2, \phi, 1) = d(2, 1) = 5$$

$$\text{Cost } (3, \phi, 1) = d(3, 1) = 6$$

$$\text{Cost } (4, \phi, 1) = d(4, 1) = 8$$

That means we have obtained dist (2, 1), dist (3, 1) and dist (4, 1).

Step 2 :

$$\text{Candidate (S)} = 1$$

Apply the formula,

$$\text{Cost } (i, S) = \min \{d[i, j] + \text{Cost } (j, S - \{j\})$$

Hence from vertex 2 to 1, vertex 3 to 1 and vertex 4 to 1 by considering intermediate path lengths we will calculate total optimum cost.

$$\text{Cost } (2, \{3\}, 1) = d(2, 3) + \text{Cost } (3, \phi, 1)$$

$$= 9 + 6$$

$$\text{Cost } (2, \{3\}, 1) = 15$$

$$\text{Cost } (2, \{4\}, 1) = d(2, 4) + \text{Cost } (4, \phi, 1)$$

$$= 10 + 8$$

$$\text{Cost } (2, \{4\}, 1) = 18$$

$$\text{Cost } (3, \{2\}, 1) = d(3, 2) + \text{Cost } (2, \phi, 1)$$

$$= 3 + 5$$

$$\text{Cost } (3, \{2\}, 1) = 18$$

$$\text{Cost}(3, \{4\}, 1) = d(3, 4) + \text{Cost}(4, \phi, 1)$$
$$= 12 + 8$$
$$\text{Cost}(3, \{4\}, 1) = 20$$
$$\text{Cost}(4, \{2\}, 1) = d(4, 2) + \text{Cost}(2, \phi, 1)$$
$$= 8 + 5$$
$$\text{Cost}(4, \{2\}, 1) = 13$$
$$\text{Cost}(4, \{3\}, 1) = d(4, 3) + \text{Cost}(3, \phi, 1)$$
$$= 9 + 6$$
$$\text{Cost}(4, \{3\}, 1) = 15$$

Step 3 : Consider candidate (S) = 2

$$\text{Cost}(2, \{3, 4\}, 1) = \min\{[d(2, 3) + \text{Cost}(3, \{4\}, 1)], [d(2, 4) + \text{Cost}(4, \{3\}, 1)]\}$$
$$= \min\{[9 + 20], [10+15]\}$$
$$\text{Cost}(2, \{3, 4\}, 1) = 25$$
$$\text{Cost}(3, \{2, 4\}, 1) = \min\{[d(3, 2) + \text{Cost}(2, \{4\}, 1)], [d(3, 4) + \text{Cost}(4, \{2\}, 1)]\}$$
$$= \min\{[13+18], [12+13]\}$$
$$\text{Cost}(3, \{2, 4\}, 1) = 25$$
$$\text{Cost}(4, \{2, 3\}, 1) = \min\{[d(4, 2) + \text{Cost}(2, \{3\}, 1)], [d(4, 3) + \text{Cost}(3, \{2\}, 1)]\}$$
$$= \min\{[8 + 15], [9 + 18]\}$$
$$\text{Cost}(4, \{2, 3\}, 1) = 23$$

Step 4 : Consider candidate (S) = 3. i.e. Cost (1, {2, 3, 4}) but as we have chosen vertex 1 initially the cycle should be completed i.e. starting and ending vertex should be 1.

∴ We will compute,

$$\text{Cost}(1, \{2, 3, 4\}, 1) = \min\left\{\begin{matrix}[d(1,2) + \text{Cost}(2, \{3,4\},1)], [d(1,3) + \text{Cost}(3, \{2,4\},1)] \\ [d(1,4) + \text{Cost}(4, \{2,3\},1)]\end{matrix}\right\}$$
$$= \min([10 + 25], [15 + 25], [20 + 23])$$
$$= 35$$

Thus the optimal tour is of path length 35.

Consider step 4 now, from vertex 1 we obtain the optimum path as d(1, 2). Hence select vertex 2. Now consider step 3, in which from vertex 2 we obtain optimum cost from d(2, 4). Hence select vertex 4. Now in step 2 we get remaining vertex 3 as d(4, 3) is optimum. Hence optimal tour is 1, 2, 4, 3, 1.

4.9 Flow - Shop Scheduling

There are distinct classes of scheduling problem and these are -

1. **Scheduling with only one machine :** When there exists only one machine for scheduling given jobs, then each job must be processed on single machine after completion of previous job.
2. **Flow shop scheduling :** If there are more than one machine and there are multiple jobs, then each job must be processed by corresponding machine or processor. That means with operation of each job must be processed on.
3. **Job-shop j^{th} machine :** If there are multiple jobs and more than each job must be processed on j^{th} machine but every job is associated with processing order for corresponding operation.
4. **Open shop :** There are multiple jobs and multiple machines, then any operation of any job can be processed by any machine.

Let us now discuss an example of flow shop scheduling.

Example 4.5 : *Schedule two jobs on 4 machines using flow shop scheduling technique. The time required by each operation of these jobs is given by following matrix.*

$$J = \begin{bmatrix} 3 & 0 \\ 0 & 3 \\ 4 & 2 \\ 5 & 2 \end{bmatrix}$$

Solution : Given that there are 4 machines, the flow shop scheduling for these operations is as shown below.

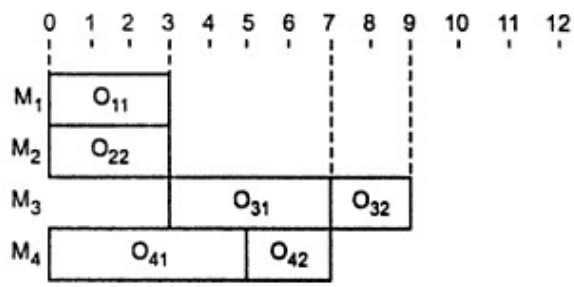


Fig. 4.6 (a)

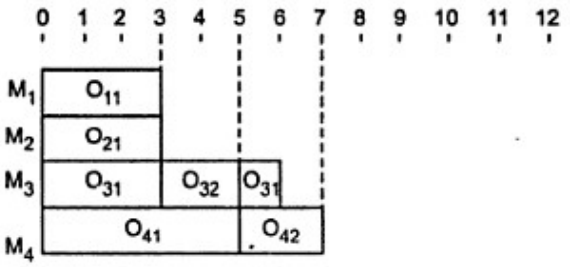


Fig. 4.6 (b)

Solved Exercise

Q.1 *Write algorithm for traveling salesperson problem using dynamic programming.*

Ans. : Algorithm for traveling salesperson problem

Problem Description : Let G be directed graph denoted by (V, E) where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij}. The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.

A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The traveling salesperson problem is to find the tour of minimum cost.

Dynamic programming is used to solve this problem.

Step 1 : Let the function C(1, V–{1}) is the total length of the tour terminating at 1. The objective of TSP problem is that the cost of this tour should be minimum.

Let d[i, j] be the shortest path between two vertices i and j.

Step 2 : Let $V_1, V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour. Then $(V_1, V_2, \ldots V_n)$ must be a shortest path from V_1 to V_n which passes through each vertex exactly once.

Here the principle of optimality is used. The path $V_i, V_{i+1}, \ldots V_j$ must be optimal for all paths beginning at v(i), ending at v(j), and passing through all the intermediate vertices $\{V_{(i+1)} \ldots V_{(j-1)}\}$ once.

Step 3 : Following formula can be used to obtain the optimum cost tour.

$$\text{Cost}(i, S) = \min \{d[i, j] + \text{Cost}(j, S - \{j\})\} \text{ where } j \in S \text{ and } i \in S$$

Q.2 *Consider 4 elements $a_1 < a_2 < a_3 < a_4$ with $q_0 = 0.25$, $q_1 = 3/16$, $q_2 = q_3 = q_4 = 1/16$. $p_1 = 1/4$, $p_2 = 1/8$, $p_3 = p_4 = 1/16$.*

a) *Construct the optimal binary search tree as a minimum cost tree.*

b) *Construct the table of values W_{ij}, C_{ij}, V_{ij} computed by the algorithm to compute the roots of optimal subtrees.*

Ans. : For convenience we will multiply the probabilities q_i and p_i by 16. Hence p_i and q_i values are $(p_1, p_2, p_3, p_4) = (4, 2, 1, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (4, 3, 1, 1, 1)$

Let us compute values of W first. For W_{ii} computation we will use,

$$W_{ii} = q_i$$

Hence

$$W_{00} = q_0$$

$$\therefore \quad W_{00} = 4$$

$$W_{11} = q_1$$

$$\therefore \quad W_{11} = 3$$

$$W_{22} = q_2$$

$$\therefore \quad W_{22} = 1$$

$$W_{33} = q_3$$

$$\therefore \quad W_{33} = 1$$

$$W_{44} = q_4$$

$$\therefore \quad W_{44} = 1$$

Compute W_{ii+1} using formula

$$W_{ii+1} = q_i + q_{i+1} + p_{i+1}$$

Hence,

$$W_{01} = q_0 + q_1 + p_1$$

$$= 4 + 3 + 4$$

$$\therefore \quad W_{01} = 11$$

$$W_{12} = q_1 + q_2 + p_2$$

$$= 3 + 1 + 2$$

$\therefore \quad W_{12} = 6$

$$W_{23} = q_2 + q_3 + p_3$$

$$= 1 + 1 + 1$$

$\therefore \quad W_{23} = 3$

$$W_{34} = q_3 + q_4 + p_4$$

$$= 1 + 1 + 1$$

$\therefore \quad W_{34} = 3$

For computing W_{ij} we will use following formula

$$W_{ij} = W_{ij-1} + p_j + q_j$$

$\therefore \quad W_{02} = W_{01} + p_2 + q_2$

$$= 11 + 2 + 1$$

$\therefore \quad W_{02} = 14$

$\therefore \quad W_{13} = W_{12} + p_3 + q_3$

$$= 6 + 1 + 1$$

$\therefore \quad W_{13} = 8$

$\therefore \quad W_{24} = W_{23} + p_4 + q_4$

$$= 3 + 1 + 1$$

$\therefore \quad W_{24} = 5$

Now,

$$W_{03} = W_{02} + p_3 + q_3$$

$$= 14 + 1 + 1$$

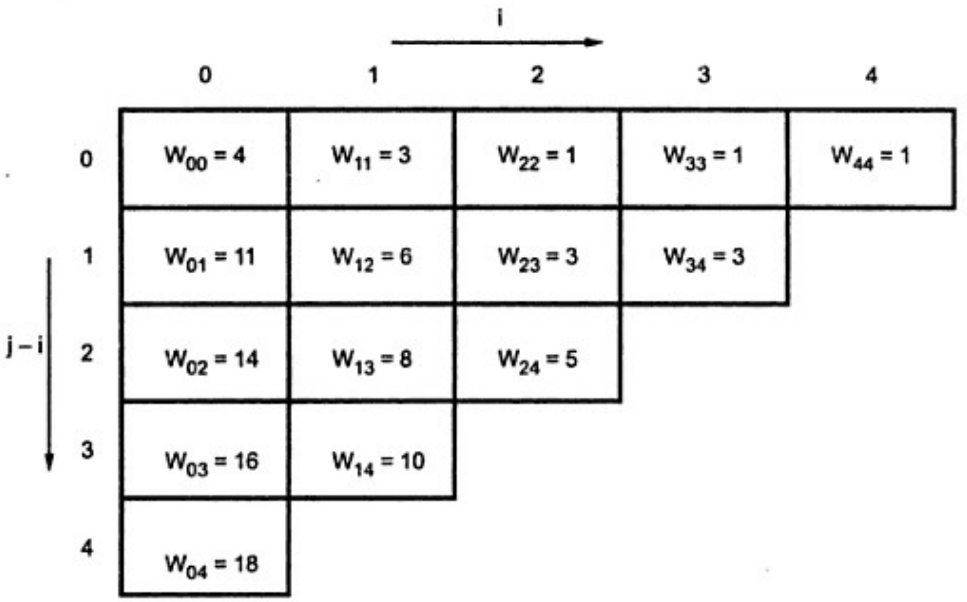
$\therefore \quad W_{03} = 16$

$$W_{04} = W_{03} + p_4 + q_4$$

$$= 16 + 1 + 1$$

$\therefore \quad W_{04} = 18$

To summarize W values



To compute C and r values we will use $C_{ii} = 0$ and $r_{ii} = 0$.

Hence

$C_{00} = 0$
$r_{00} = 0$
$C_{11} = 0$
$r_{11} = 0$
$C_{22} = 0$
$r_{22} = 0$
$C_{33} = 0$
$r_{33} = 0$
$C_{44} = 0$
$r_{44} = 0$

To compute C_{ii+1} and r_{ii+1} we will use following formulae

$$r_{i\,i+1} = i + 1$$

$$C_{i\,i+1} = q_i + q_{i+1} + p_{i+1}$$

Hence

$$r_{01} = 1$$

$$C_{01} = q_0 + q_1 + p_1$$

$$= 4 + 3 + 4$$

$$\therefore \quad C_{01} = 11$$

$$r_{12} = 2$$

$$C_{12} = q_1 + q_2 + p_2$$

$$= 3 + 1 + 2$$

$$\therefore \quad C_{12} = 6$$

$$r_{23} = 3$$

$$C_{23} = q_2 + q_3 + p_3$$

$$= 1 + 1 + 1$$

$$\therefore \quad C_{23} = 3$$

$$r_{34} = 4$$

$$C_{34} = q_3 + q_4 + p_4$$

$$= 1 + 1 + 1$$

$$\therefore \quad C_{34} = 3$$

To compute C_{ij} and r_{ij} we will compute the values of k first.

The value of k lies between values of $r_{i, j-1}$ to $r_{i+1, j}$. The min $\{C_{i\,k-1} + C_{kj}\}$ decides value of k.

Consider i = 0 and j = 2.

To decide value of k, the range for k is $r_{01} = 1$ to $r_{12} = 2$ when k = 1 and i = 0, j = 2. We will compute C_{ij} using formula

$$C_{ij} = C_{ik-1} + C_{kj}$$

$$\therefore \quad C_{02} = C_{00} + C_{02}$$

$$C_{02} = 6 \quad \rightarrow \text{Minimum value of } C_{02}.$$

When k = 2, and i = 0, j = 2.

$$C_{02} = C_{01} + C_{22}$$

$$C_{02} = 11 + 0$$

$$\therefore \quad C_{02} = 11$$

In above computations we get minimum value of C_{02} when k = 1. Hence the value of k becomes 1.

As $r_{ij} = k$

$$r_{02} = 1$$

For $C_{ij} = W_{ij} + \min \{C_{i, k-1} + C_{kj}\}$

$$\therefore \quad C_{02} = W_{02} + C_{02}$$

$$C_{02} = 14 + 6 = 20$$

Now for r_{13} and C_{13}

k is between $r_{12} = 2$ to $r_{23} = 3$ then when k = 2 and i = 1, j = 3.

$$C_{13} = C_{11} + C_{23}$$

$$= 0 + 3$$

$$\therefore \quad C_{13} = 3 \rightarrow \text{Minimum value of } C_{13}$$

When k = 3 and i = 1, j = 3

$$C_{13} = C_{12} + C_{33}$$

$$= 6 + 0$$

$$\therefore \quad C_{13} = 6$$

Hence k = 2 gives minimum value of C_{13}.

$$\therefore \quad r_{13} = 2$$

and $C_{13} = W_{13} + C_{13}$ (Minimum value)

$$= 8 + 3$$

$$\therefore \quad C_{13} = 11$$

Now for r_{24} and C_{24}

k is between $r_{23} = 3$ to $r_{34} = 4$.

$\therefore$ i = 2, j= 4 when k = 3.

$$C_{24} = C_{22} + C_{34}$$

$$= 0 + 3$$

$$\therefore \quad C_{24} = 3$$

When k = 4, i = 2, j = 4.

$$C_{24} = C_{23} + C_{44}$$

$$= 3 + 0$$

$$\therefore \quad C_{24} = 3$$

That means value of k can be 3. Let us consider k = 3. Then,

$$r_{24} = 3$$

and $$C_{34} = W_{34} + C_{34}$$

$$= 5 + 3$$

$$\therefore \quad C_{34} = 8$$

Now for r_{03} and C_{03}

Value of k lies between $r_{02} = 1$ to $r_{13} = 2$

When k = 1 and i = 0, j = 3.

$$C_{03} = C_{00} + C_{13}$$

$$= 0 + 11$$

$$\therefore \quad C_{03} = 11 \quad \rightarrow \text{Minimum value of } C_{03}.$$

When k = 2 and i = 0, j = 3.

$$C_{03} = C_{01} + C_{23}$$

$$= 11 + 3$$

$$\therefore \quad C_{03} = 14$$

Hence value of k = 1.

$$\therefore \quad r_{03} = 1$$

and $$C_{03} = W_{03} + C_{03}$$

$$= 16 + 11$$

$$\therefore \quad C_{03} = 27$$

5 Backtracking

5.1 Introduction

Backtracking is one of the most general technique. In this technique, we search for the set of solutions or optimal solution which satisfies some constraints. One way of solving a problem is by exhaustive search: we enumerate all possible solutions, and see which one produces the optimum result. For example, for the Knapsack problems, we could look at every possible subset objects, and find out one which has the greatest profit value and at the same time not greater than the weight bound. Backtracking is a variation of exhaustive search, where the search is refined by eliminating certain possibilities. Backtracking is usually faster method than an exhaustive search.

In this chapter we will discuss the concept of backtracking technique with the help of many applications such as n-queen's problem, sum of subsets, graph coloring and Hamiltonian cycle.

5.2 General Method

- In the backtracking method
  1. The desired solution is expressible as an n tuple $(x_1, x_2,...x_n)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function C $(x_1, x_2,...x_n)$.
- The problem can be categorized into three categories.

  For instance - For a problem P let C be the set of constraints for P. Let D be the set containing all solutions satisfying C then

  Finding whether there is any feasible solution?- is the decision problem.

  What is the best solution ? - is the optimization problem.

  Listing of all the feasible solution - is the enumeration problem.

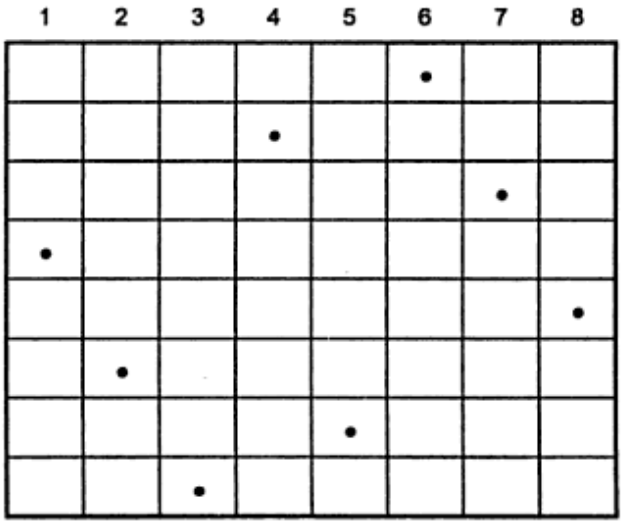
- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.
- Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.
- Explicit constraints are rules, which restrict each vector element to be chosen from the given set. Implicit constraints are rules, which determine which of the tuples in the solution space, actually satisfy the criterion function.

For example

Example 1 : 8-Queen's problem - The 8-queen's problem can be stated as follows. Consider a chessboard of order 8×8. The problem is to place 8 queens on this board such that no two queens can attack each other. That means no two queens can be placed on the same row, column or diagonal.

The solution to 8-queens problem can be obtained using backtracking method.

The solution can be given as below -



Example 2 : Sum of subsets -

There are n positive numbers given in a set. The desire is to find all possible subsets of this set, the contents of which add onto a predefined value M.

In other words,

Let there be n elements given by the set w = (w_1, w_2, w_3,..., w_n) then find out all the subsets from whose sum is M.

For example

Consider n = 6 and (w_1, w_2, w_3, w_4, w_5, w_6) = (25, 8, 16, 32, 26, 52) and M = 59 then we will get desired sum of subset as (25, 8, 26).

We can also represent the sum of subset as (1, 1, 0, 0, 1, 0). If solution subset is represented by an n-tuple (x_1, x_2,...x_n) such that x_i could be either 0 or 1. The x_i = 1 means the weight w_i is to be chosen and x_i = 0 means that weight w_i is not to be chosen.

5.2.1 Some Terminologies used in Backtracking

Backtracking algorithms determine problem solutions by systematically searching for the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can be placed on 4 × 4 chessboard. Then no two queens can attack each other".

Following figure shows tree organization for this problem.

See Fig. 5.1 on next page.

- Each node in the tree is called **a problem state.**
- All paths from the root to other nodes define the **state space of** the problem.
- The solution states are those problem states s for which the path from root to s defines a **tuple** in the solution space.

  In some trees the leaves define the **solution states.**
- **Answer states :** These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy the implicit constraints.

For example

See Fig. 5.2 on page no. 5 - 5.

- A node which is been generated and all whose children have not yet been generated is called **live node.**
- The live node whose children are currently being expanded is called **E-node.**
- **A dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

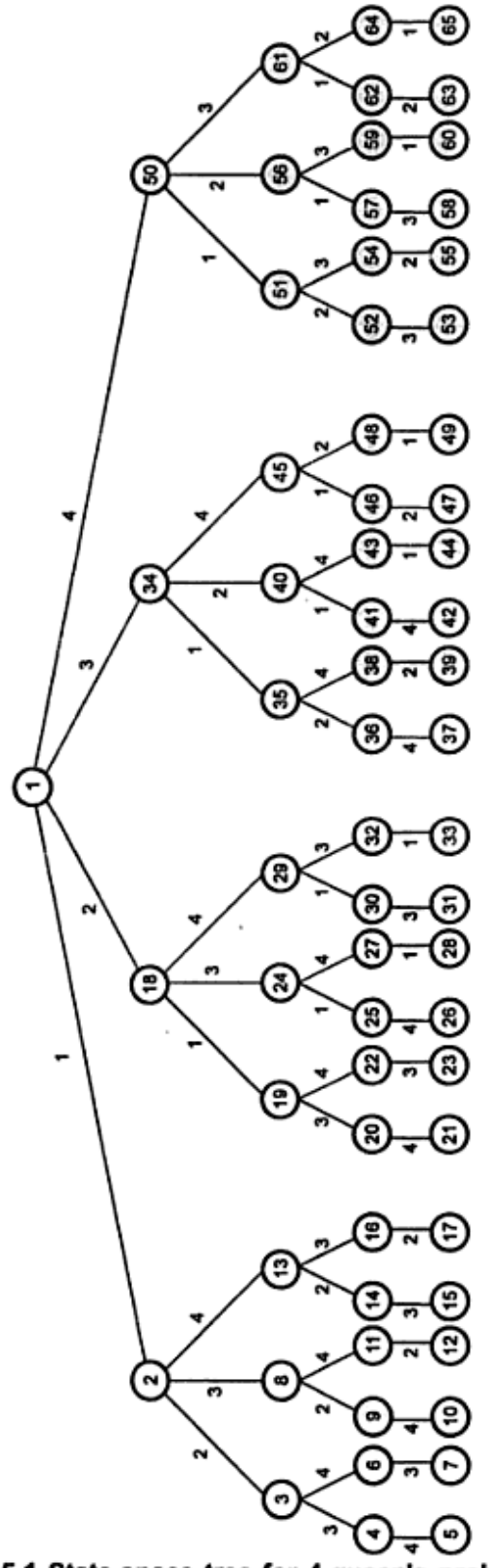


Fig. 5.1 State-space tree for 4-queen's problem

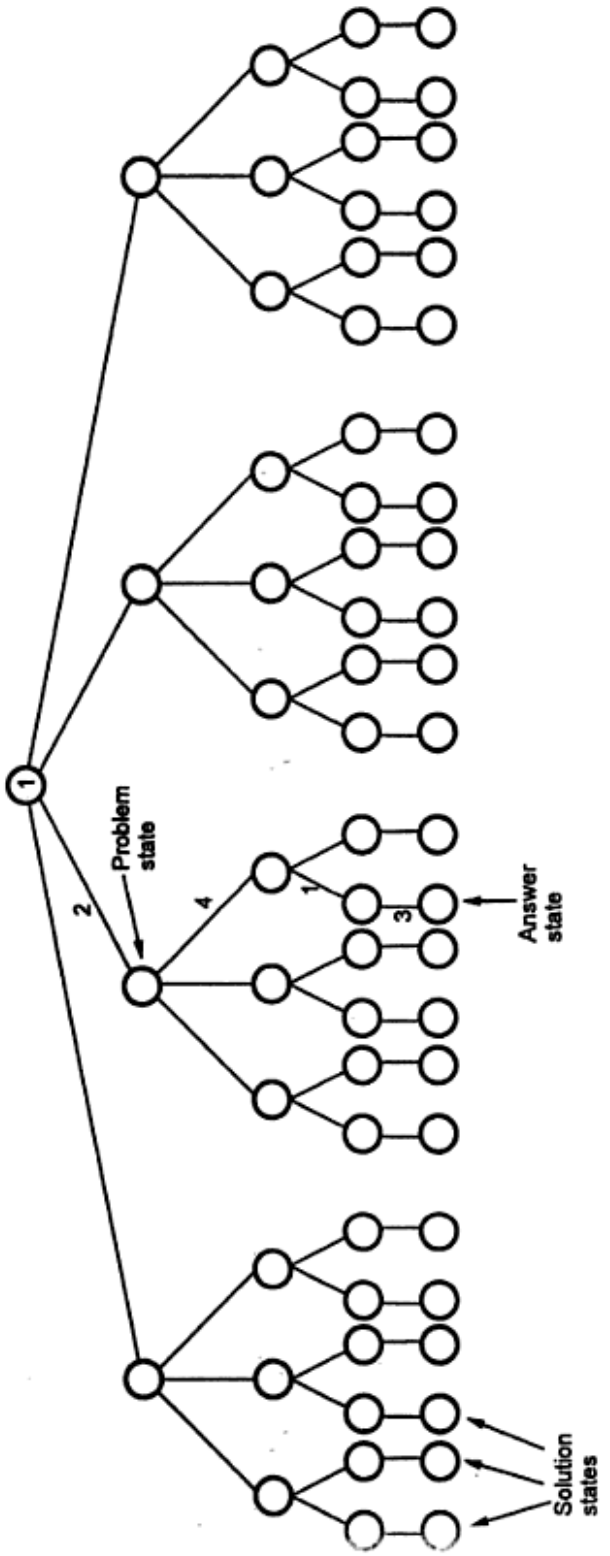


Fig. 5.2

- There are two methods of generating state search tree -
  i) Backtracking - In this method, as soon as new child C of the current E-node N is generated, this child will be the new E-node.
  The N will become the E-node again when the subtree C has been fully explored.
  ii) Branch and Bound - E-node will remain as E-node until it is dead.
- Both bracktracking and branch and bound methods use bounding functions. The bounding functions are used to kill live nodes without generating all their children. The care has to be taken while killing live nodes so that at least one answer node or all answer nodes are obtained.

Algorithm for backtracking -

```
Algorithm Backtrack()
//This is recursive backtracking algorithm
//a[k] is a solution vector. On entering (k-1) remaining next
//values can be computed.
//T(a1,a2,...ak) be the set of all values for a(i+1), such that
//(a1,a2,...ai+1) is a path to problem state.
// Bi+1 is a bounding function such that if Bi+1(a1,a2,...ai+1) is false
//for a path (a1,a2,...ai+1) from root node to a problem state then
//path can not be extended to reach an answer node
{
 for ( each ak that belongs to T((a1,a2,...ak-1) do
 {
  if(Bk(a1,a2,...ak) = true) then // feasible sequence
  {
     if ((a1,a2,...ak) is a path to answer node then
       print(a[1],a[2],...a[k]);
     if (k<n) then
       Backtrack(k+1); //find the next set.
  }
 }
}
```

The non recursive backtracking algorithm can be given as –

```
Algorithm Non_Rec_Back(n)
// This is a non recursive version of backtracking
//a[1,...,n] is a solution vector and each a1,a2,...ak will be used to print solution.
{
 k:=1;
```

```
while(k ≠ 0) do
{
  if(any a[k] that belongs to T(a[1],a[2],...a[k-1])remains untried)
  AND (B_k (a[1],a[2]...,a[k] is true) then
  {
   if((a[1],a[2]...a[k]) is a path to answer node) then
    write(a[1],a[2],..a[k]); // solution printed
   //consider next element of the set
   k:=k+1;
   else
     k : =k-1;  //backtrack to most recent value
  }
 }
}
```

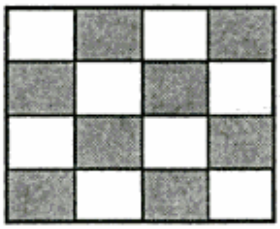
5.3 8-Queen's Problem

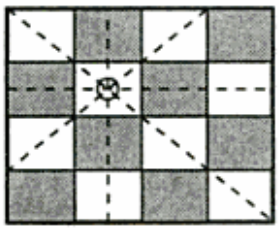
The n-queen's problem can be stated as follows.

Consider a n × n chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For example

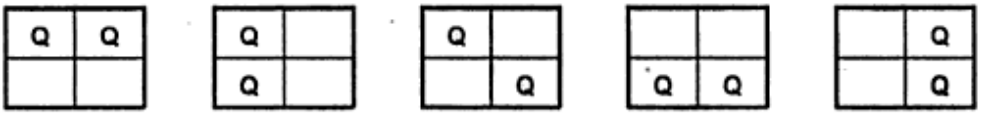
Consider 4 × 4 board



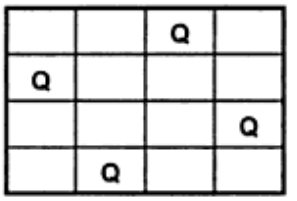


The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 2-Queen's problem is not solvable - Because 2-queens can be placed on 2 × 2 chessboard as



- But 4-queen's problem is solvable.

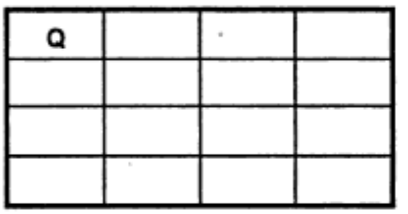


⇐ Note that no two queens can attack each other.

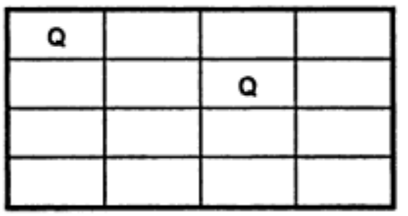
5.3.1 How to Solve n-Queen's Problem ?

Let us take 4-queens and 4 × 4 chessboard.

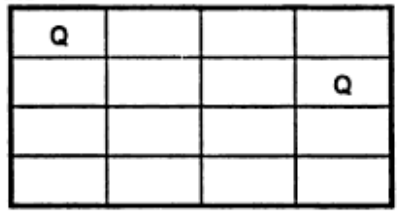
- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row. i.e. on 1st row and 1st column.



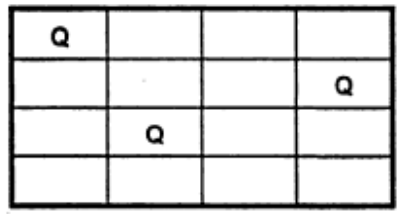
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2nd row and 3rd column.



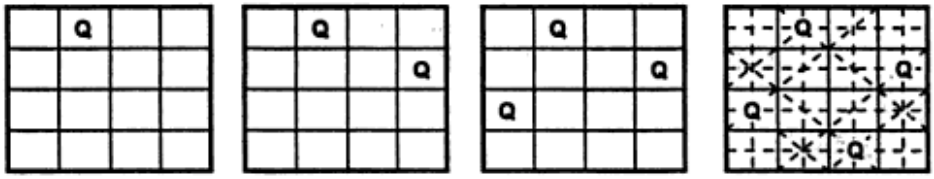
- This is the dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2nd queen at (2, 4) position.



- The place 3rd queen at (3, 2) but it is again another dead end as next queen (4th queen) cannot be placed at permissible position.



- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).



The state space tree of 4-queen's problem is shown in Fig. 5.3.

See Fig. 5.3 on next page.

5.3.2 Algorithm

```
Algorithm Queen(n)
//Problem description : This algorithm is for implementing n
//queen's problem
//Input : total number of queen's n.
    for column ←1 to n do
    {
      if(place(row,column))then
      {
       board[row]column//no conflict so place queen
      if(row=n)then//dead end
       print_board(n)
         //printing the board configuration
```

This function checks if two queens are on the same diagonal or not.

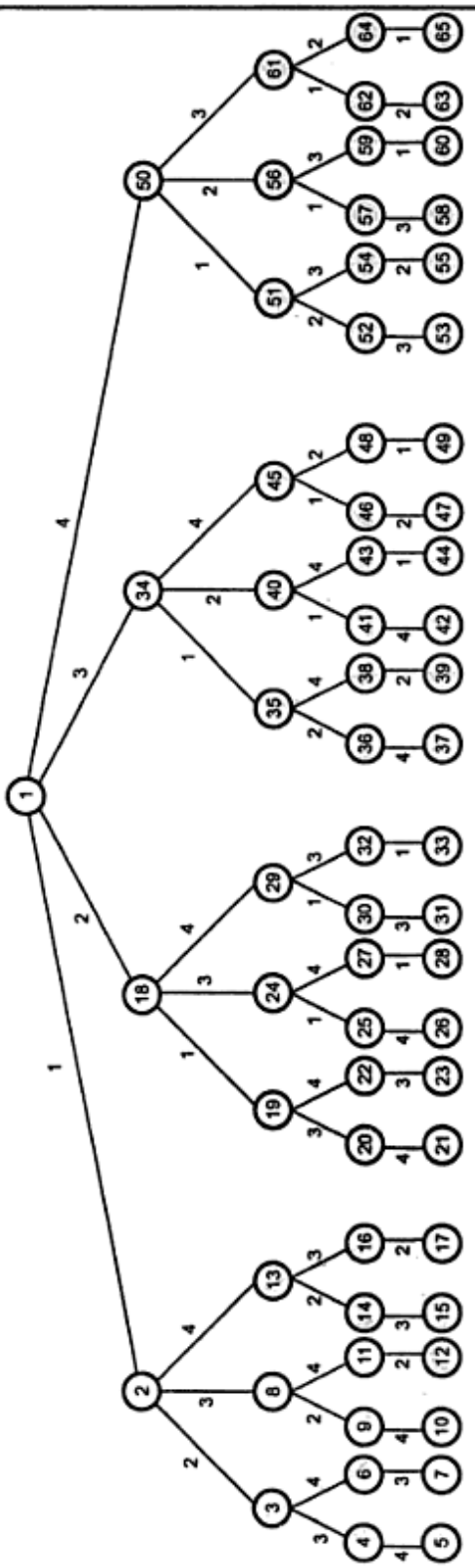


Fig. 5.3 State space tree for 4-queen's problem

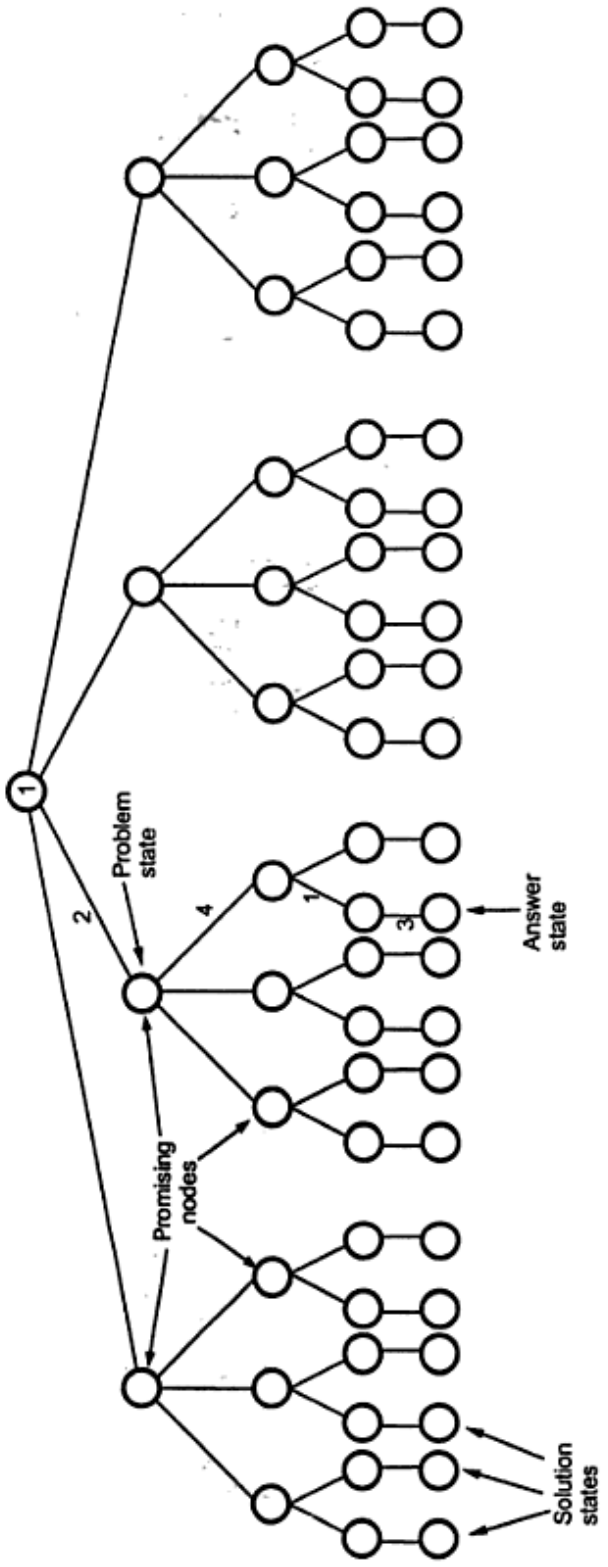


Fig. 5.4 Basic terms used in state space tree

```
        else//try next queen with next position
            Queen(row+1,n)
        }
    }
```

Row by row each queen is placed by satisfying constraints.

```
Algorithm place(row,column)
//Problem Description : This algorithm is for placing the
//queen at appropriate position
//Input : row and column of the chessboard
//output : returns 0 for the conflicting row and column
//position and 1 for no conflict.
    for i ← 1 to row-1 do
    { //checking for column and diagonal conflicts
      if(board[i] = column)then
        return 0
    else if(abs(board[i]- column) = abs(i - row))then
        return 0
    }
    //no conflicts hence Queen can be placed
  return 1
```

Same column by 2 queen's

This formula gives that 2 queens are on same diagonal

C functions

```
/* By this function we try the next free slot
and check for proper positioning of queen
*/
void Queen(int row,int n)
{
    int column;
    for(column=1;column<=n;column++)
    {
      if(place(row,column))

      {
        board[row] = column;//no conflict so place queen
        if(row==n)//dead end
          print_board(n);
              //printing the board configuration
        else  //try next queen with next position
          Queen(row+1,n);
      }
    }
}
```

```
int place(int row,int column)
{
        int i;
        for(i=1;i<=row – 1;i++)
        { //checking for column and diagonal conflicts
           if(board[i] == column)
              return 0;
           else
              if(abs(board[i] – column) == abs(i – row))
                 return 0;
        }
        //no conflicts hence Queen can be placed
        return 1;
}
```

C program

```
/*************************************************************************
This program is to implement n-queen's problem using backtracking
*************************************************************************/
  #include<stdio.h>
  #include<conio.h>
  #include<math.h>
  #include<process.h>

  int board[20];
  int count;
  void main()
  {
        int n,i,j;
        void Queen(int row,int n);
        clrscr();
        printf("\n\t Program for n-Queen's Using Backtracking");
        printf("\nEnter Number of Queen's");
        scanf("%d",&n);
        Queen(1,n);//trace using backtrack
        getch();
  }
  /* This function is for printing the solution
        to n-queen's problem
  */
  void print_board(int n)
  {
```

We start from 1st row and go row by row.

```
        int i,j;
        printf("\n\nSolution %d : \n\n",++count);
        //number of solution
        for(i=1;i<=n;i++)
        {
            printf("\t%d",i);
        }
        for(i=1;i<=n;i++)
        {
            printf("\n\n%d",i);
            for(j=1;j<=n;j++)// for n x n board
            {
                if(board[i]==j)
                    printf("\tQ");//Queen at i,j position
                else
                    printf("\t-");// empty slot
            }
        }
        printf("\n Press any key to continue...");
        getch();


}
/*
This function is for checking for the conflicts.
If there is no conflict for the desired position
it returns 1 otherwise it returns 0
*/
int place(int row,int column)
{
        int i;
        for(i=1;i<=row-1;i++)
        { //checking for column and diagonal conflicts
            if(board[i] == column)
                return 0;
            else
                if(abs(board[i] - column) == abs(i - row))
                    return 0;
        }
```

```
        //no conflicts hence Queen can be placed
        return 1;
}
/* By this function we try the next free slot
and check for proper positioning of queen
*/
void Queen(int row,int n)
{
        int column;
        for(column=1;column<=n;column++)
        {
            if(place(row,column))
            {
             board[row] = column;//no conflict so place queen
             if(row==n)//dead end
                print_board(n);
                        //printing the board configuration
                else  //try next queen with next position
                    Queen(row+1,n);
            }
        }
}
```

Output

```
 Program for n-Queen's Using Backtracking
Enter Number of Queen's 4

Solution 1 :

    1   2   3   4

1   -   Q   -   -

2   -   -   -   Q

3   Q   -   -   -

4   -   -   Q   -
```

Press any key to continue...

Solution 2 :

```
    1  2  3  4

1   -  -  Q  -

2   Q  -  -  -

3   -  -  -  Q

4   -  Q  -  -
```

Press any key to continue...

Example 5.1 : *Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).*

Solution : As the feasible sequence is given, we will place the queens accordingly and then try out the other remaining places.

	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5								
6								
7								
8								

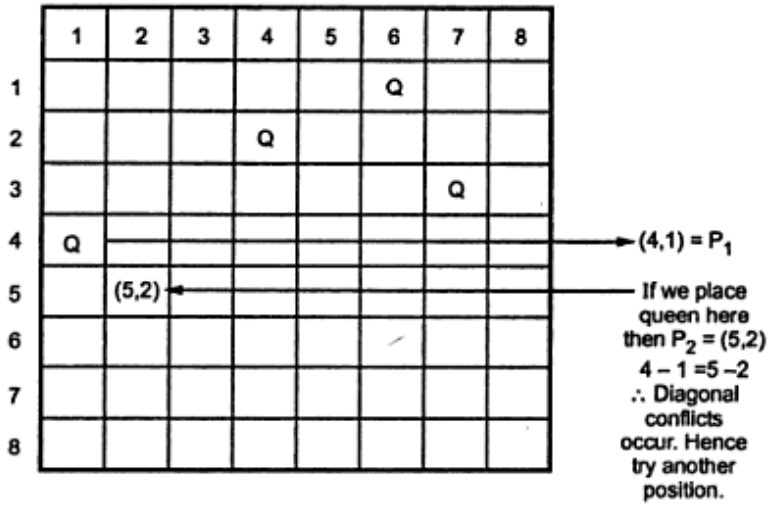
The diagonal conflicts can be checked by following formula-

Let, $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions. Then P_1 and P_2 are the positions that are on the same diagonal, if

$$i + j = k + l \quad \text{or}$$

$$i - j = k - l$$

Now if next queen is placed on (5, 2) then



It can be summarized below.

Queen Positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				As 4 – 1 = 5 – 2 conflict occurs.
6	4	7	1	3				5 – 3 ≠ 4 –1 or 5 + 3 ≠ 4 + 1 ∴ feasible
6	4	7	1	3	2			As 5 + 3 = 6 + 2. It is not feasible.
6	4	7	1	3	5			Feasible
6	4	7	1	3	5	2		Feasible
6	4	7	1	3	5	2	8	List ends and we have got feasible sequence.

The 8-queens on 8 × 8 board with this sequence is -

	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5			Q					
6					Q			
7		Q						
8								Q

8-queens with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

5.4 Sum of Subsets Problem

Problem Statement

Let, $S = \{S_1, ... S_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

It is always convenient to sort the set's elements in ascending order. That is,

$$S_1 \leq S_2 \leq ..... \leq S_n$$

Let us first write a general algorithm for sum of subset problem.

Algorithm

Let, S be a set of elements and d is the expected sum of subsets. Then -

Step 1 : Start with an empty set.

Step 2 : Add to the subset, the next element from the list.

Step 3 : If the subset is having sum d then stop with that subset as solution.

Step 4 : If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5 : If the subset is feasible then repeat step 2.

Step 6 : If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

This problem can be well understood with some example.

Example 5.2 : *Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Solve it for obtaining sum of subset.*

Solution :

Initially subset = { }	Sum = 0	–
5	5	Then add next element.
5, 10	15 ∵ 15 < 30	Add next element.
5, 10, 12	27 ∵ 27 < 30	Add next element.
5, 10, 12, 13	40	Sum exceeds d = 30 hence backtrack.
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack
5, 10, 12, 18	45	Sum exceeds d ∴ Not feasible. Hence backtrack.

5, 10		
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible ∴ Backtrack.
5, 10		
5, 10, 15	30	Solution obtained as sum = 30 = d

∴ The state space tree can be drawn as follows.

{5, 10, 12, 13, 15, 18}

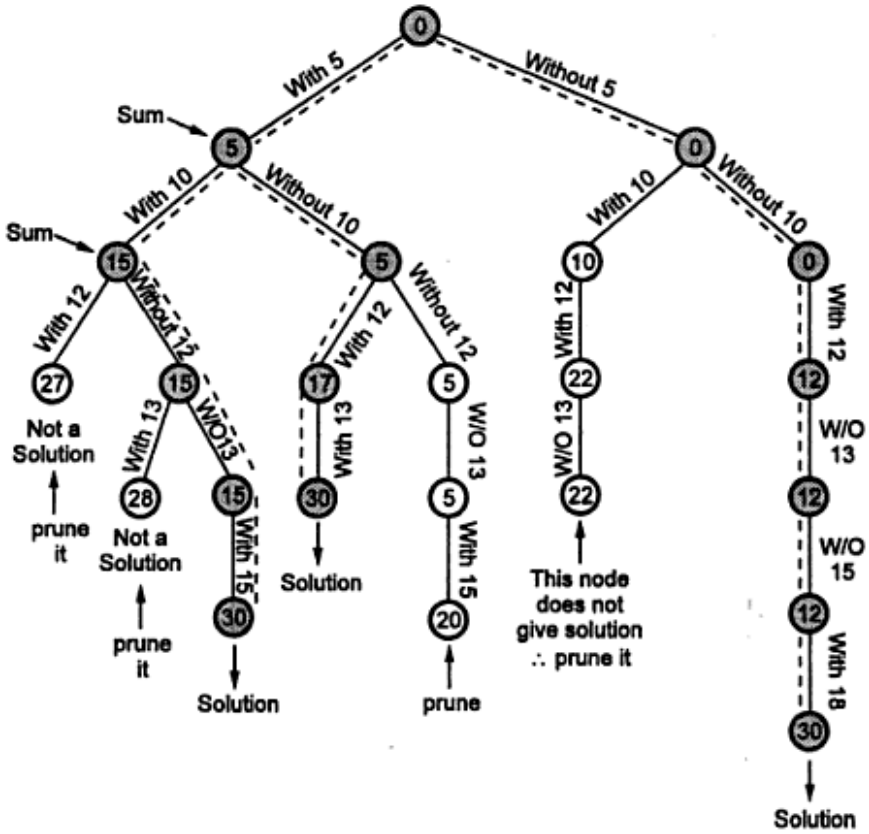


Fig. 5.5 State space tree for sum of subset

Example 5.3 : *Let m = 31 and W = {7, 11, 13, 24}. Draw a portion of state space tree for solving sum-of-subset problem for the above given algorithm.*

Solution : Initially we pass (0, 1, 55) to sum-of-subset function. The sum = 0, index = 1 and remaining_sum = 55 initially [∵ 7 + 11 + 13 + 24].

The recursive execution of the algorithm will be,

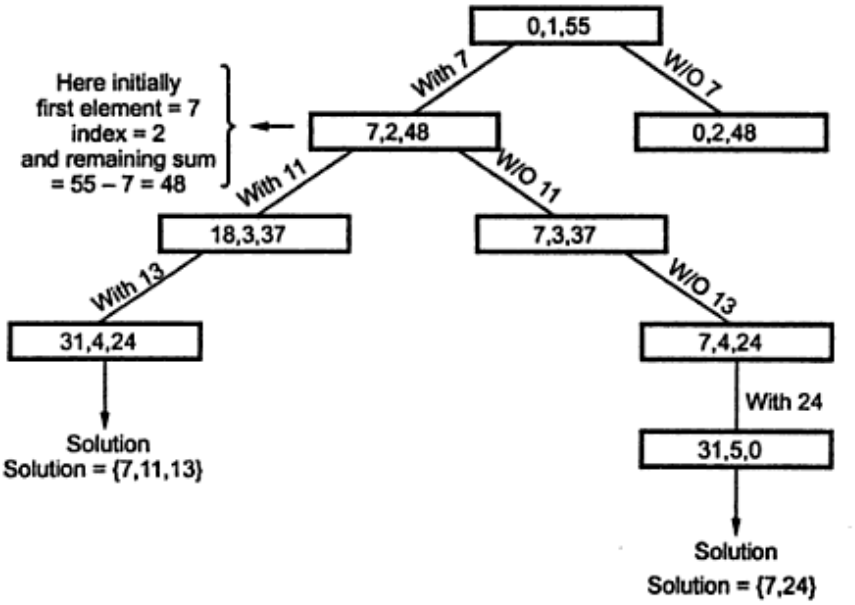


Fig. 5.6

5.4.1 Algorithm

Algorithm Sum_Subset(sum, index, Remaining_sum)

//sum is a variable that stores the sum of all the

//selected elements

//index denotes the index of chosen element from the given

//Remaining_sum is initially sum of all the elements.

//selection of some element from the set subtracts the

//chosen value

//from Remaining_sum each time.

//w[1...n] represents the set containing n elements

//a[j] represents the subset where $1 \le j \le$ index

//sum $= \sum_{j=1}^{index-1} w[j]*a[j]$

//Remaining_sum $= \sum_{j=index}^{n} = w[j].$

// w[j] is sorted in non-decreasing order

// For a feasible sequence assume that $w[1] \le d$ *and*

$\sum_{i=1}^{n} w[i] \ge d$

// Generate left child until sum + w[index] is $\le d$

```
        a[index]←1
  if(sum + w[index] = d) then
    write(a[1...index]) //subset is found          ← The subset is printed
      else if (sum + w[index] + w[index+1] d) then
       Sum_Subset((sum+w[index]), (index+1),(Remaining_sum – w[index]));
                  // Generate right child          ← Search the next element which can make sum ≤ d
       if(sum+Remaining_sum – w[index] ≥ d) AND
                  (sum+w[index+1]≤ d)) then
       {
              a[index] ← 0
        Sum_Subset(sum, (index+1), (Remaining_sum – w[index]));
     }
```

5.5 Graph Coloring

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used. This problem is also called m-coloring problem. If the degree of given graph is d then we can color it with d + 1 colors. The least number of colors needed to color the graph is called its chromatic number. For example : Consider a graph given in Fig. 5.7.

As given in Fig. 5.7 we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows -

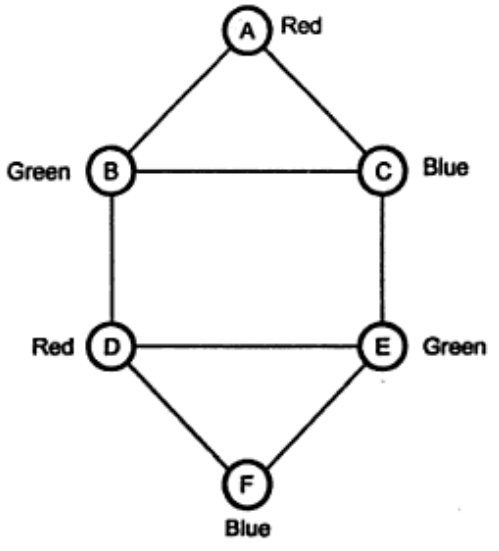
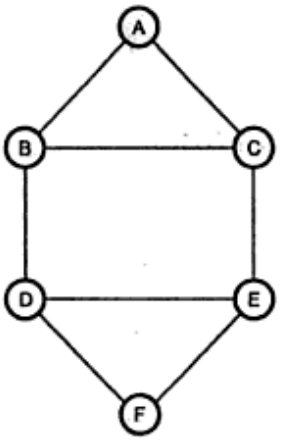


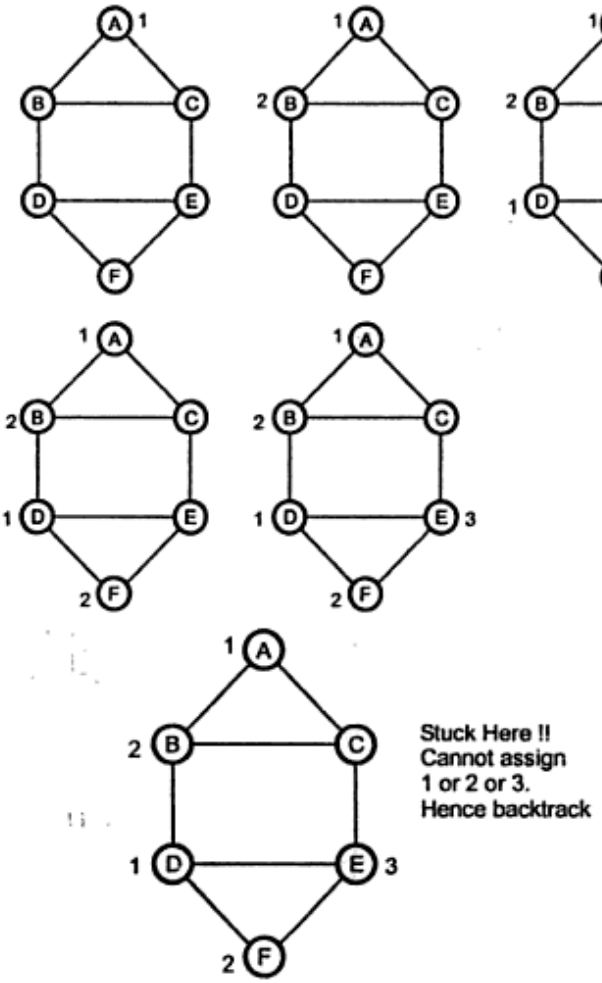
Fig. 5.7 Graph and its coloring

Step 1 :

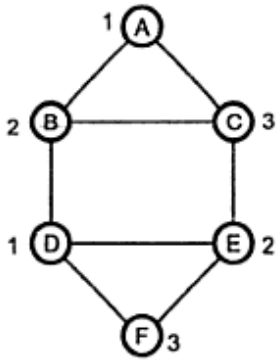


A Graph G consists of vertices from A to F. There are three colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.

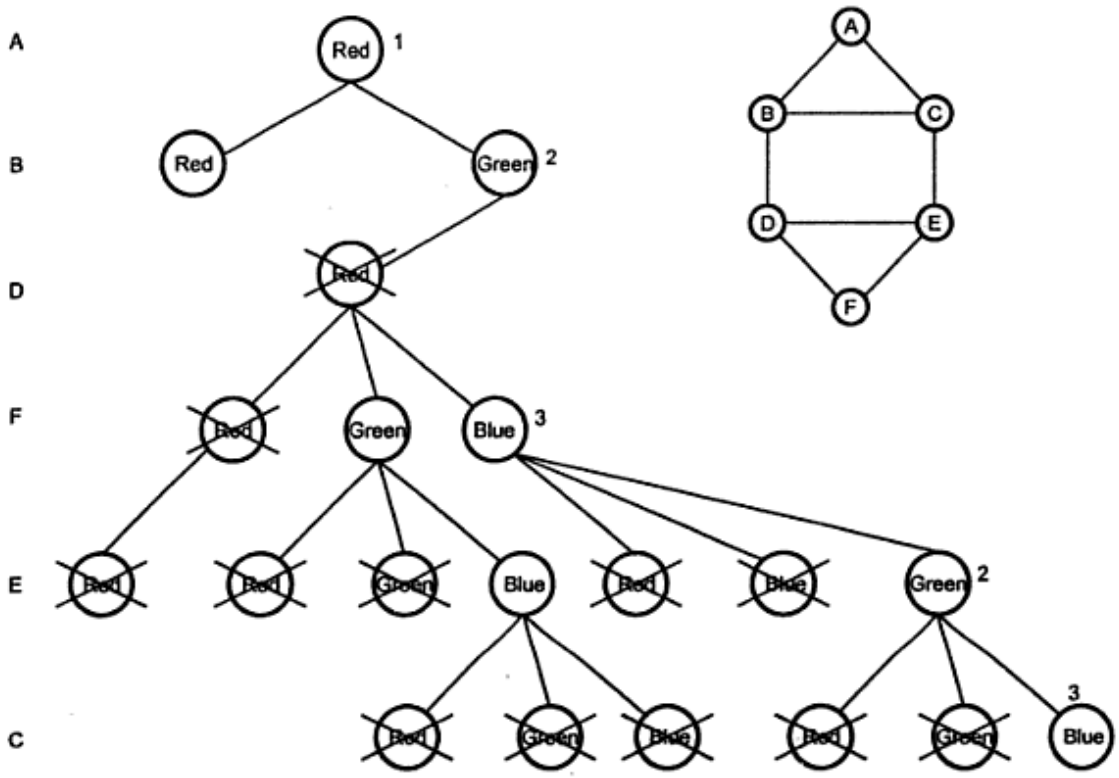
Step 2 :



Step 3 :



Thus the graph coloring problem is solved. The state-space tree can be drawn for better understanding of graph coloring technique using backtracking approach -



Here we have assumed, color index Red = 1, Green = 2 and Blue = 3.

Algorithm

The algorithm for graph coloring uses an important function Gen_Col_Value() for generating color index. The algorithm is -

```
Algorithm Gr_color(k)

//The graph G[1 : n,1 : n] is given by adjacency matrix.
//Each vertex is picked up one by one for coloring
//x[k] indicates the legal color assigned to the vertex
{
 repeat
 {
   // produces possible colors assigned
    Gen_Col_Value(k);

 if(x[k] = 0) then
    return; //Assignment of new color is not possible.
 if(k=n) then // all vertices of the graph are colored
   write(x[1:n]); //print color index
 else
  Gr_color(k+1) // choose next vertex
 }until(false);
}
```

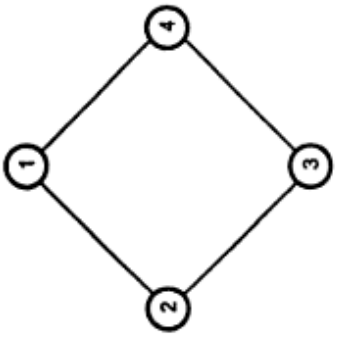
The algorithm used assigning the color is given as below

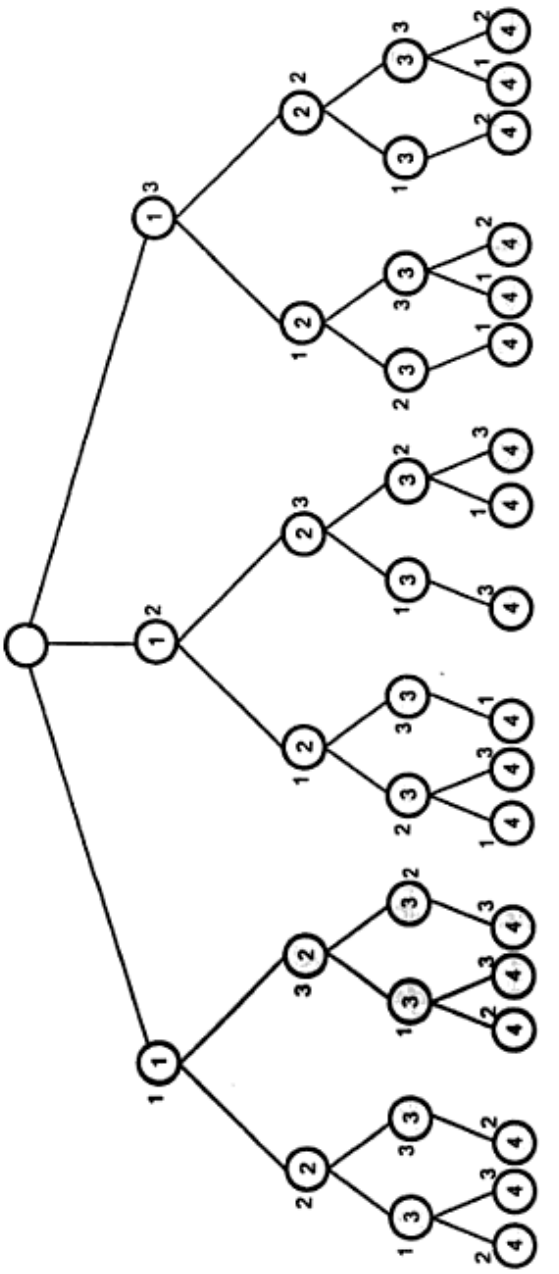
```
Algorithm Gen_Col_Value(k)
 //x[k] indicates the legal color assigned to the vertex
 // If no color exists then x[k] = 0
 {
// repeatedly find different colors
  repeat
   {
     x[k] ← (x[k]+1) mod (m+1) ; //next color index when it is
       //highest index
     if(x[k] = 0) then // no new color is remaining
       return;
    for (j ← 1 to n) do
    {
      // Taking care of having different colors for adjacent
   // vertices by using two conditions i) edge should be
   // present between two vertices
     // ii) adjacent vertices should not have same color
      if(G[k,j]!=0) AND (x[k]→x[j])) then
       break;
    }
   //if there is no new color remaining
    if ( j= n+1) then
       return;
```

```
    }until(false);
  }
```

The Gr_color takes computing time of $\sum_{i=0}^{n-1} m^i$. Hence total computing time for this algorithm is $O(nm^n)$.

If we trace the above algorithm then we can assign distinct colors to adjacent vertices. For example : Consider, a graph given below can be solved using three colors.





Consider,

Red = 1, Green = 2 and Blue = 3.

The number inside the node indicates vertex number and the number outside the node indicates color index.

5.6 Hamiltonian Cycle

Problem -

Given an undirected connected graph and two nodes x and y then find a path from x to y visiting each node in the graph exactly once.

For example : Consider the graph G -

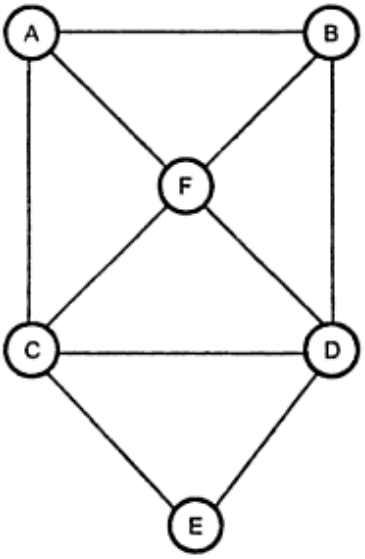


Fig. 5.8

Then the Hamiltonian cycle is A - B - D - E - C - F - A. This problem can be solved using backtracking approach. The state space tree is generated in order to find all the Hamiltonian cycles in the graph. Only distinct cycles are output of this algorithm. The Hamiltonian cycle can be identified as follows -

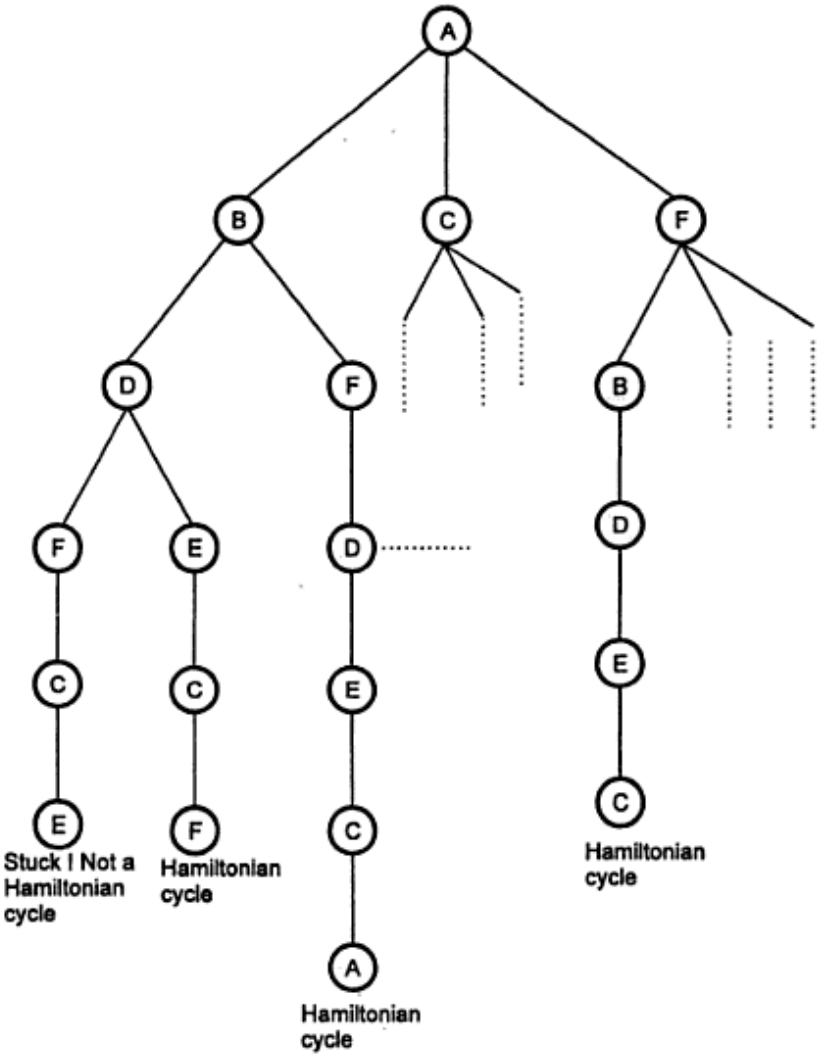


Fig. 5.9 Finding Hamiltonian cycle

In above Fig. 5.9 clearly the backtrack approach is adopted. For instance A - B - D - F - C - E; here we get stuck. For returning to A we have to revisit atleast one vertex. Hence we backtracked and from D node another path is chosen A - B - D - E - C - F - A which is Hamiltonian cycle.

```
Algorithm H_Cycle(k)
 // This is recursive backtracking algorithm for finding Hamiltonian cycle
//The graph G[1:n,1:n] is adjacency matrix used for graph G
//The cycle begins from the vertex 1
{
 repeat
```

```
  {
   Next_Vertex(k);//generates next legal vertex for determining cycle
   if(x[k]=0) then
     return;
   if(k=n) then
    write(x[1:n])  // print the Hamiltonian cycle
   else
     H_Cycle(k+1);
  } until(false);
}
```

The algorithm for generating next vertex which helps in finding Hamiltonian cycle is as given below -

```
Algorithm Next_Vertex(k)
// This algorithm finds the Hamiltonian path by choosing appropriate
// vertex each time. The x[1:k–1] is a Hamiltonian path.
//Each time the x[k] is assigned to a next highest numbered vertex
// which is not visited earlier. If x[k]=0 then no vertex is assigned
// to x[k].
//n denotes total number of vertices
{
 while(1)
 {
      //obtain next vertex
   x[k] : =x[k+1] mod (n+1);
   if(x[k]=0) then
      return;
  if(G[x[k–1],x[k]]=1) then // if edge between vertex k and
                            // k–1 is present
  {
   for (j=1 to k-1 ) do //for every adjacent vertex
     if(x[j]=x[k]) then
       break;// not a distinct vertex
    if(j=k) then // obtained a distinct vertex
    {
      if((k<n) OR ((k=n) AND G[x[n],x[1]]=1)) then
        return;//return a distinct vertex
     }
   }
  }
}
```

```
11 {
12    c ← c+w[i]
13    if (c < m) then     //m is capacity of knapsack
14      ub ← ub + p[i]
15    else
16      {
17          ub ← ub + (1–(c–m)/w[i])*p[i]
18          return ub
19       }
20    return ub
21 }
```

This function is invoked by a Knapsack function **Bk (k, cp, cw)**. The algorithm for the same is as given below.

```
    Algorithm Bk (k,cp,cw)
1   //Problem description : This algorithm is to obtain
2   //solution for knapsack problem. Using this algorithm
3   //a state space tree can be generated.
4   //Input : Initially k=1, cp=0 and cw=0. The
5   //k represents the index of currently referred
6   //item. The cp and cw represent the profit and
7   //weights of items, so far selected.
8   //Output : The set of selected items that are
9   //satisfying the objective of knapsack problem.
10  if (cw+w[i]<= m) then
11  {
12    temp[k] ← 1  //temp array stores currently selected object
13    if (k<n) then
14    Bk(k+1,cp+p[k],cw+w[k])   //recursive call
15    if ((cp+p[k] > final_profit) AND (k==n)) then
16    { //final_profit is initially –1, it represents final profit
17    final_profit ← cp + p[k]
18    final_wt ← cw + w[k]
19    for (j ← 1 to k) then
20      x[j] ← temp[j]
21  }
22}
23  if (Bound_calculation (cp,cw,k) ≥ final_profit) then
24  {
25    temp[k] ← 0
```

Generates left child

Finally get the solution

Generates right child

Invoke this function in order to obtain upper bound.

```
26    if (k < n) then
27        Bk(k+1, cp, cw)
28    if((cp > final_profit) AND (k=n)) then
29    {
30        final_profit ← cp
31        final_wt ← cw
32        for(j ← 1 to k)
33            x[j] ← temp [j]
34    }
35}
```

Finally get the solution

Example 5.4 : *Obtain the optimal solution to the Knapsack problem* $n = 3$, $m = 20$ $(p_1, p_2, p_3)=(25, 24, 15)$ *and* $(w_1, w_2, w_3) = (18, 15, 10)$.

Solution : Given that n = 3 and m = capacity of Knapsack = 20.

We have

i	p [i]	w [i]	p [i] / w [i]
1	25	18	1.3
2	24	15	1.6
3	15	10	1.5

As we want p[i] / w[i] ≥ p[i+1] / w [i+1], let us rearrange the items as

i	p [i]	w [i]	p [i] / w [i]
1	24	15	1.6
2	15	10	1.5
3	25	18	1.3

Step 1 : Here we will trace knapsack backtracking algorithm using above values.

Initially set final_profit = – 1

Bk (1, 0, 0)

$\therefore$ k = 1

cp = 0

cw = 0

check if (cw + w [k] < = m)

Step 4 :

$$k = 3$$

$$cp = 24 + p[2] = 24 + 15 = 39$$

$$cw = 15 + w[2] = 15 + 10 = 25$$

check if (cw + w [k] <= m)

i.e. if (25 + w [3] = 25 + 18 < 20) → no

Hence we will calculate upper bound.

Step 5 :

For calculating upper bound initially set.

$$ub = cp = 39$$
$$c = cw = 25$$

} Refer line 8 and 9 of algorithm **Bound_ calculation**

Set i = k + 1 i.e. 3 + 1 = 4 But i = 4 > n

Hence we will keep ub as it is

$$\therefore \quad ub = 39$$

As (39 > final_profit) i.e. 39 > – 1

Set temp [k] = temp [3] = 0

Now k = n = 3 and cp = 39 > final_profit (i.e. – 1)

$$final\_profit = cp = 39$$
$$final\_wt = cw = 25$$

} Refer line 30 and 31 of algorithm **Bk**

Now copy all the contents of temp array to an array x [ ]. Refer line 32, 33 of **Algorithm Bk.**

Hence **x = {1, 0, 0} i.e. item 1 with weight = 15 and profit = 24 is selected**

The state space tree can be drawn as

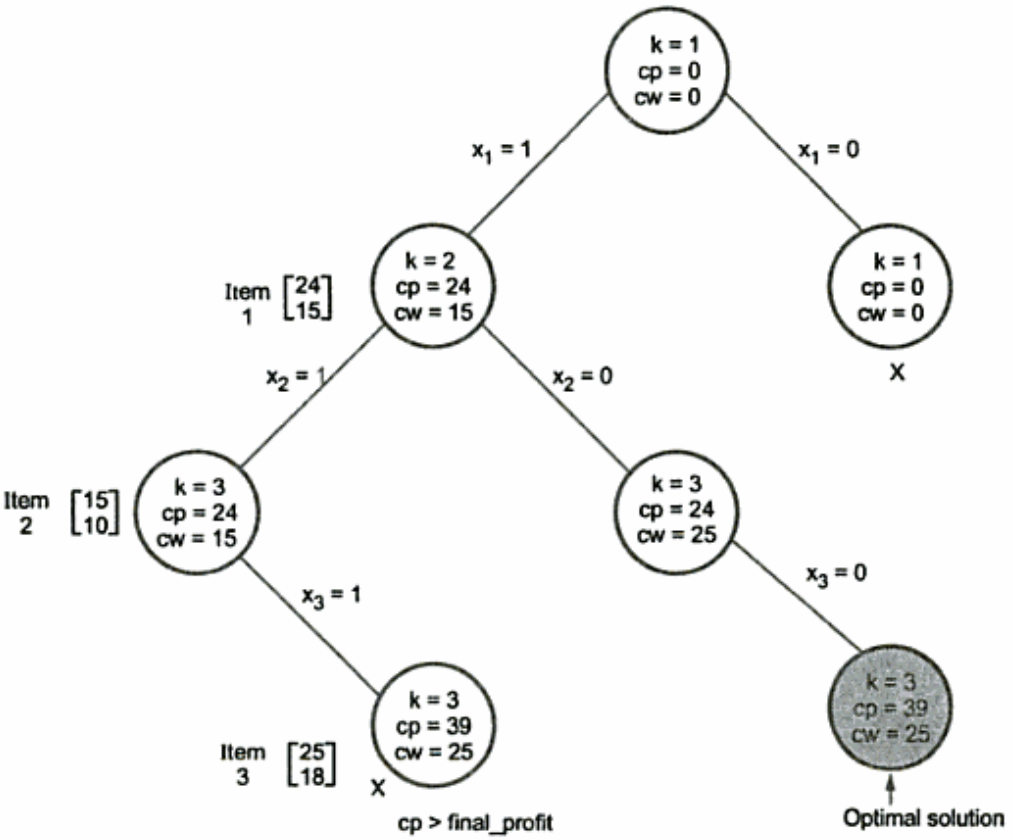


Fig. 5.10 State space tree for Knapsack

Here the left branch indicates inclusion of item and right branch indicates exclusion of items. Hence $x_1 = 1$ means 1^{st} item selected and $x_2 = 0$, $x_3 = 0$ means 2^{nd} and 3^{rd} items are not selected. The state space tree can be created in DFS manner.

Solved Exercise

Q.1 *Draw the tree organization of the 4-queen's solution space. Number the nodes using depth first search.*

Ans. : The state space tree for 4-queen's problem consists of 4! leaf nodes. That means there are 24 leaf nodes. The solution space is defined by a path from root to leaf node.

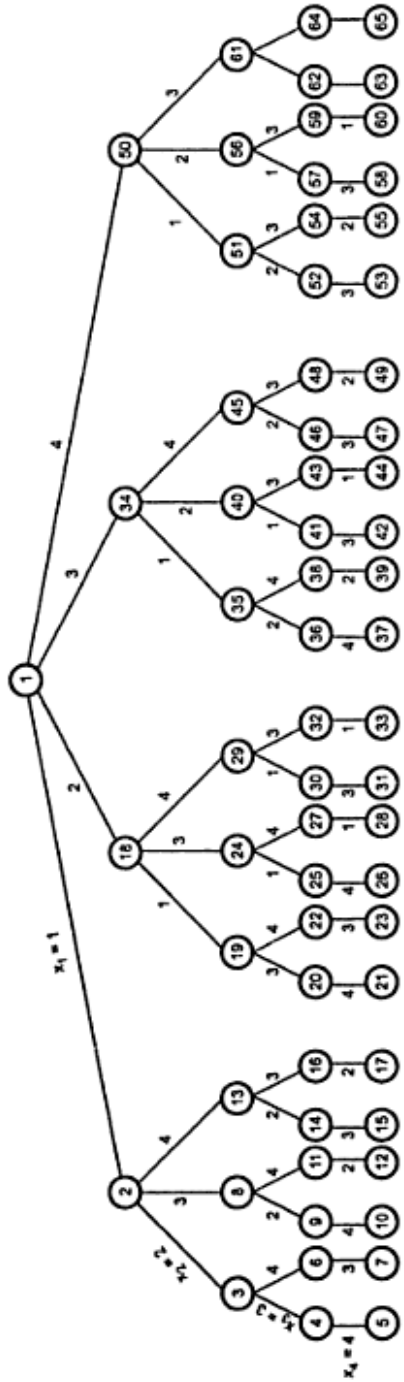


Fig. 5.11 State space tree for 4-queen's problem

The solution can be obtained for 4 queen's problem. For instance from 1 to leaf 31 represents one possible solution.

State space : All paths from root to other nodes define the state space of the problem.

Explicit constraints : Explicit constraints are rules, which restrict, each vector element to be chosen from given set.

Implicit constraints : Implicit constraints are rules which determine which of the tuples in the solution space satisfy the criterion function.

Problem states : Each node in the state space tree is called problem state.

Solution states : The solution states are those problem states s for which the path from root to s defines a tuple in the solution space. In some trees the leaves define solution states.

Answer states : These are the leaf nodes which correspond to an element in the set of solutions. These are the states which satisfy the implicit constraints.

Live node : A node which is generated and whose children have not yet been generated is called live node.

E-node : The live node whose children are currently being expanded is called E-node.

Dead node : A dead node is a generated node which is not to be expanded further or all of whose children have been generated.

Bounding functions : Bounding functions will be used to kill live nodes without generating all their children. A care should be taken while doing so, because atleast one answer node should be produced or even all the answer nodes be generated if problem needs to find all possible solutions.

Q.3 *For a feasible sequence (7, 5, 3, 1) solve 8 queen's problem using backtracking.*

Ans. : While placing the queen at next position we have to check whether the current position chosen is on the same diagonal of previous queen's position.

If $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions then P_1 and P_2 lie on the same diagonal if $i + j = k + l$ or $i - j = k - l$. Let us put the given feasible sequence and try out remaining positions.

Q.4 *Draw a pruned state space tree for a given sum of subset problem.*

$S = \{3, 4, 5, 6\}$ *and* $d = 13$

Ans. :

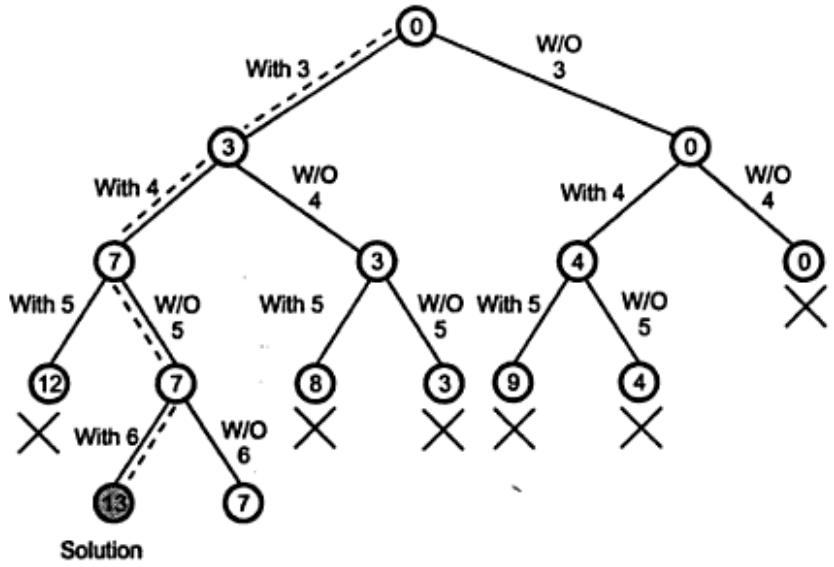


Fig. 5.13

Fig. 5.13 state space tree the total sum is given inside each node. Pruned nodes has cross at its bottom.

Hence the solution is {3, 4, 6}.

Q.5 *Obtain all possible solutions to 4-queen's problem. Establish the relationship between the two solutions.*

Ans. : The solutions to 4-queen's problem are as given below.

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

Solution 1
(2, 4, 1, 3)

	1	2	3	4
1			Q	
2	Q			
3				Q
4		Q		

Solution 2
(3, 1, 4, 2)

If these two solutions are observed then we can say that second solution can be simply obtained by reversing the first solution.

ii) {62, 77, 45, 96, 90, 63, 41, 92} *sum= 212*

8. *Write an algorithm for finding sum of subset using backtracking.*
9. *Is it possible to solve sum of subset using dynamic programming approach? Justify your answer.*
10. *State and explain Graph coloring problem? How backtracking approach is useful for assigning different colors to adjacent vertices?*
11. *Give an algorithm for graph coloring using backtracking. Analyze your algorithm.*
12. *Give an algorithm for finding Hamiltonian cycles using backtracking. Analyze your algorithm.*

❑❑❑