Design and Analysis of Algorithms
UNIT-IV::THE GREEDY METHOD

Greedy Method:General Method, Applications- Job sequencing with dead lines, 0/1 knapsack problem, minimum cost spanning trees, single source shortest path problem.

------------------------------*

Divide and conquer technique is applicable only for problems, which can be divisible. There exist some problems which cannot be divisible.

In divide and conquer approach, a problem is divided recursively into sub problems of same kind as the original problem, until they are small enough to be solved and finally the solutions of the sub problems are combined to get the solution of the original problem. In Greedy approach, a problem is solved by determining a subset to satisfy some constraints. If that subset satisfies the given constraints, then it is called as feasible solution, which maximizes or minimizes a given objective function. A feasible solution that either maximizes or minimizes an objective function is called as optimal solution.

Most of the problems have n inputs and require us to obtaining a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an ***optimal solution***.
The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure.
Example 1) Let us find the maximum value for the following problem. Given objective function is $Z = 3x + 4y$ subjected to $0 <= x <= 1$
$-1 <= y <= 1$
Assume input set as (x,y)
{(0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1), (2, 2), (3, 3), (-4, -4) }
After applying conditions to input set, {(2, 2), (3, 3), (-4, -4)} pairs are removed from input set. Remaining pairs {(0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1) }are called feasible solution.
Among the feasible solution at (1,1) the objective function is maximum.
$Z=3x1 + 4x1 = 7$ So (1, 1) is an optimal solution for the given objective function.

Control Abstraction of Greedy Method

```
Algorithm greedy(a,n)                    // a contains n inputs
{
    solution:=0;
      for i:= 1 to n do
       {
           x=select(a);
           if feasible (solution, x) then
            {
                solution := Union( solution, x);
            }
           else
             reject();
       }
    return solution;
}
```

APPLICATIONS

JOB Sequencing with Dead Lines, 0/1 KNAPSAK PROBLEM , MINIMUM COST SPANNING TREES, SINGLE SOURCE SHORTEST PATH PROBLEM

KNAPSACK PROBLEM

We are given n objects and a knapsack (Bag). Object i has a weight w_i and knapsack has a capacity of m. If a fraction x_i such that $0<=x_i<=1$ of object i is placed into a knapsack, then a profit of $p_i x_i$ is earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

Maximize $\sum_{1\le i\le n} p_i x_i$ ---------------------------(1)

Subject to $\sum_{1\le i\le n} w_i x_i \le m$ -----------------------(2)

And $0 \le x_i \le 1,\ 1 \le i \le n$ ---------------(3)

We obtain a feasible solution when equations (2) & (3) are satisfied & optimal solution is obtained when eq(1) is also satisfied.

```
Algorithm greedyknapsack(m,n)
{
    for i:= 1 to n do
       x[i]:=0;
     u:=m;
     for i:= 1 to n do
      {
           if (w[i] > u) then break;
           x[i]:=1.0;
           u:=u-w[i];
      }
    if (i<=n) then
         x[i]:=u/w[i];
}
```

Example 1)

Total number of objects	n=3,
Total capacity	m=20,
Profits of Knapsack	(p1,p2,p3)=(25,24,15),
Weights	(w1,w2,w3)=(18,15,10)

```
Algorithm greedy_knapsack(20,3)
{
    for i:= 1 to 3 do
       x[1]:=x[2]:=x[3]:=0;
     u:=20;
     for i:= 1
      {
           if (w[1] > 20) then break;        i.e.  18>20 false
           x[1]:=1.0;
           u:=u-w[i];          u=20-18=2
      }
    for i:= 2
         if (w[2] > 2) then break;        i.e.  15>2  true so break

    if (2<=3) then
         x[i]:=u/w[i];         x[2]:=2/15=0.13
}
```

Total profit = 25*1 +24*0.13 = 28.2

To solve the knapsack problem we consider 3 optimization measures:

1) Consider the objects with their profits in descending order.
2) Consider the objects with their weights in ascending order.
3) Consider the objects with their profit/weight ratio in descending order.

Case 1: Try to fill the knapsack by including the object with largest profit. If an object under consider does not fit, then a faction of it is included to fill the knapsack. Thus each time an object is included into the knapsack, we obtain largest possible increase in profit i.e. object1 (p1=25) is placed into the knapsack, and then x1=1 and a profit of 25 is earned. Then m=20-18=2. i.e. 2 units of space is left in the knapsack. Objet2 has the second largest profit (p2=24) but w2=15>2 and does not fit into the knapsack. Using x2=2/15 fills the knapsack exactly with the part of object2. Profit earned is 24*2/15=3.2
Total profit earned is 25+3.2=28.2 .

This method used to obtain the solution is termed as "Greedy method" because at each step, we choose to introduce that object which will increase the objective function value the most. However, this did not yield the optimal solution.

$$\sum_{i=1}^{n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

= 25*1+24*2/15+15*0
= 25+3.2+0
= 28.5

Case 2: Try to be greedy with the capacity & use it up as slowly as possible. This requires to consider the objects in the order of increasing weights. The object with lowest weight is object3 (w3=10) is placed into the knapsack first. So, x3=1 and the profit of 15*1=15 is earned. Object 2 has the next highest weight(w2=15). But it does not fit into the knapsack. Using x2=10/15 fits the knapsack exactly with part of object2 and the profit earned is 24*10/15=16.
Total profit earned is 15+16=31

$$\sum_{i=1}^{n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

= 25*0+24*10/15+15*1
= 0+16+15
= 31

Case 3: Consider the object that has max profit/weight ratio used, i.e consider the objects in the ratio of pi/wi in decreasing order. The first object i.e to be considered is object2 (p2/w2=1.6). So, x2=1 and a profit of 21*1=24 is earned. M= 20-15=5 units of space is left in the knapsack. The object to be considered next is object3 (p3/w3=1.5) but it does not fit into the knapsack. So, fraction of object of object3 i.e x3=5/10=05 is inserted into the knapsack & profit earned is 15*0.5=7.5.

Total profit earned is 31.5.

p1/w1=1.4 p2/w2=1.6 p3/w3=1.5

descending order of profit/weight ratio p2, p3, p1

$$\sum_{i=1}^{n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

= 25*0 + 24*1 + 15*1/2
= 0 + 24 + 7.5
= 31.5

X1	X2	X3	∑ wixi	∑ pixi	
1	2/15	0	20	28.2	
0	2/3	1	20	31	
0	1	½	20	31.5	Optimal Solution

The solution for Knapsack problem is obtained when the objects are considered according to their profits/weights ratio in descending order.

Example 2) Find the optimal solution for given instance of Kanpsack problem
N=7,
M=15,
(p1, p2, p3, p4, p5, p6, p7) = (10, 5, 15, 7, 6, 18, 3)
(w1,w2,w3,w4,w5,w6,w7) = (2, 3, 5, 7, 1, 4, 1)

Find the optimal solution for
1) Maximum profit
2) Minimum weight
3) Maximum profit per unit weight

Solution:

Case 1) Maximum profit ---- Decreasing order of profits (P6,P3,P1,P4,P5,P2,P7)
X6=1
X3=1
X1=1
X4=4/7
∑pixi = p1x1+p2x2+p3x3+p4x4
= 18*1+15*1+10*1+7*4/7=47

Case 2) Minimum Weight – Increasing order of weights (w5,w7,w1,w2,w6,,w3,w4)
X7=1
X5=1
X1=1
X2=1
X6=1
X3=4/5
∑pixi =p1x1+p2x2+p3x3+p5x5+p6x6+p7x7
=10*1+5*1+15*4/5+6*1+18*1+3*1=54

Case 3) Descending order of profit / weight ratio
p1/w1 =10/2=5
P2/w2=5/3=1.6
P3/w3=15/5=3
P4/w4=7/7=1
P5/w5=6/1=6
P6/w6=18/4=4.5
P7/w7=3/1=3
∑pixi = p5x5+p1x1+p6x6+p3x3+p7x7+p2x2
= 6*1+10*1+ 18*1 + 15*1 + 3*1+ 5*2/3 = 55.3

OPTIMAL STORAGE ON TAPES : There are 'n' programs that are to be stored on a computer tape of length 'L'. Associated with each program 'i' is a length L_i, 1<=i<=n. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L'.

We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence If the programs are stored in the order I = i1, i2, i3…,in, the time tj needed to retrieve program ij is proportional to $\sum_{1\le k\le j} l_{ik}$. If all programs are retrieved equally often, then the expected or mean retrieval time (MRT) is $\left(\frac{1}{n}\right)\sum_{1\le j\le n} t_j$. In the optimal storage on the tape problem, we are required to find a permutation for the n programs so that when they are required to find a permutation for the n programs do that when they are stored on the tape in this order the MRT is minimized. The problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I)=\sum_{1\le j\le n}\sum_{1\le k\le j} l_{ik}$.

Example) Let n=3 and (l1,l2,l3)=(5,10,3). There are n! = 6 possible orders. These orderings and their respective d values are:

Ordering	**d(I)**
1,2,3	5 + (5 + 10) + (5 + 10 + 3) = 38
1,3,2	5 + (5 + 3) + (5 + 3 + 10) = 31
2,1,3	10 + (10 + 5) + (10 + 5 + 3) = 43
2,3,1	10 + (10 + 3)+ (10 + 3 + 5) = 41
3,1,2	3 + (3 + 5) + (3 + 5 + 10) = 29
3,2,1	3 + (3 + 10) + (3 + 10 + 5) = 34

The optimal ordering is 3,1,2.

A greedy method approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the d value of the permutation constructed so far.

$$d(I)=\sum_{k=1}^{n}\sum_{j=1}^{k} l_{ij}=\sum_{k=1}^{n}(n-k+1)l_{ik} = (3-1+1)5+(3-2+1)10+(3-3+1)3=15+20+3=38$$

$$d(I)=\sum_{k=1}^{n}\sum_{j=1}^{k} l_{ij} = (5)+(5+10)+(5+10+3)=38$$

```
Algorithm storageontapes(n,m)
// n number of programs, m number of tapes
{
   j:=0;
   for i:= 1 to n do
    {
     write("append Program",i,"to permutation for tape",j);
        j := (j+1) mod m;
    }
}
```

JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline d_i>=0 and a profit p_i>0. for any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in j, or $\sum_{i \in J} p_i$. An optimal solution involves the identification of a subset, it fits the subset paradigm.

Example 1) Let n =4, (p1,p2,p3,p4)=(100,10,15,27) and (d1,d2,d3,d4)=(2,1,2,1). The feasible solutions and their values are

S.No	Feasible solution	Processing sequence	Value
1	(1,2)	2,1	100+10=110
2	(1,3)	1,3 or 3,1	100+15=115
3	(1,4)	4,1	27+100=127
4	(2,3)	2,3	10+15=25
5	(3,4)	4,3	27+15=42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing time of job 4 begins at time zero and that of job1 is completed at time 2.

Example 2) Solve the job sequencing problem given n=5, profits(1,5,20,15,10) and deadlines(1,2,4,1,3) using greedy method.

Since the maximum deadline is 4 units of time the feasible solution set must have <= 4 jobs.
Now arranging the jobs in the decreasing order of profits
(P1,P2,P3,P4,P5) = (1,5,20,15,10)
Decreasing Order of profits (P3,P4,P5,P2,P1) = (20,15,10,5,1)
Similarly deadlines (d3,d4,d5,d2,d1) = (4,1,3,2,1)

Feasible solutions and their profits

S.No	Feasible solution	Processing sequence	Value
1	{3}	3	20
2	{3,4}	4,3	20+15=35
3	{3,4,5}	4,5,3 or 4,3,5	20+15+10=45
4	{3,4,5,2}	4,2,5,3 or 4,2,3,5,	20+15+10+5=50
5	{3,5,2,1}	1,2,5,3	1+5+10+20=36

Solution (4) is an optimal solution. The jobs must be processed in the order 4,2,5,3 or 4,2,3,5 and the value of the optimal solution is 50.

```
Algorithm greedyjobseq(d,j,n)
// d-delay
// j-set of jobs that can be completed by their deadline
//n- number of jobs
{
    j:=1;
    for i:=2 to n do
        {
            if all jobs in jU{i} can be completed by their deadlines then
            j=jU{i}
        }
}
```

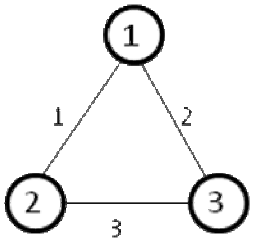
```
Algorithm job_seq(D,J,N)
{
    D[0]:=0;
     J[0]:=0;
     J[1]:=1;
     count := 1;

      for i:= 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
             {
                for s:= count to (t+1) step -1 do
                    J[s+1] := J[s];
                    J[t+1] := 1;
                    count := count +1;
             }
        }
      return count;
}
```

The computing time taken by above job sequencing algorithm is $O(n^2)$.

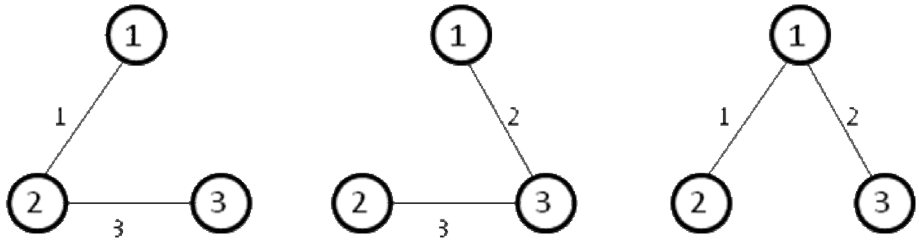
Spanning Trees :

A Spanning tree of a graph is any tree that includes every vertex in the graph.
A Spanning tree of a graph G is a sub graph of G that is a tree and contains all the vertices of G containing no circuit or cycle.
An edge of a spanning tree is called a branch
An edge in the graph that is not in the spanning tree is called a Chord.
It spans the graph, i.e. it includes every vertex of the graph.
It is a minimum cost spanning tree i.e. the total weights of all the edges is as low as possible.

Example 1) An Undirected graph and three of its spanning trees



Weighted Graph

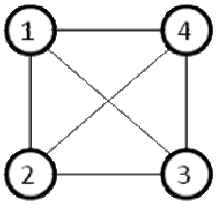


COST= 1-3=4 COST= 2-3=5 COST= 1-2=3

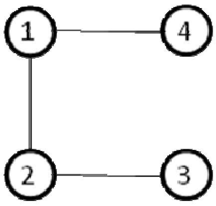
Spanning Trees

If a graph consist of n vertices then the possible spanning trees are n^{n-2}, for above example n=3, i.e 3^{3-2}=3 spanning trees.

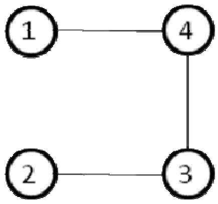
Example 2) Number of vertices =4
Number of spanning trees= $4^{(4-2)} = 4^2 = 16$



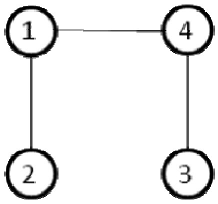
Graph



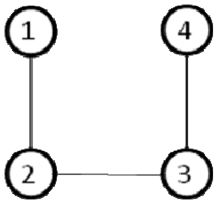
(a)



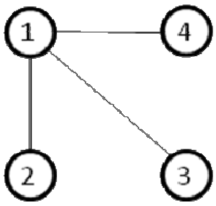
(b)



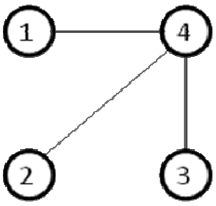
(c)



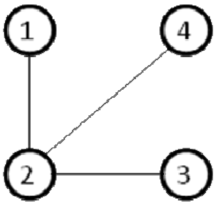
(d)



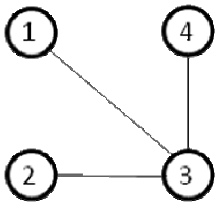
(e)



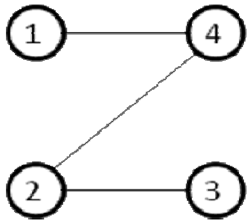
(f)



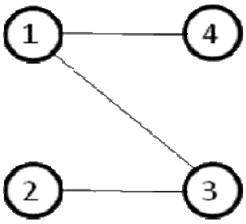
(g)



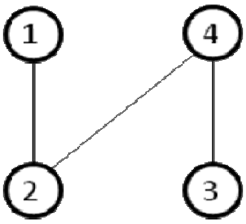
(h)



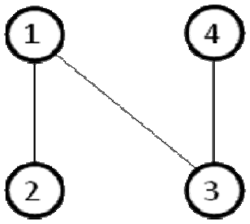
i



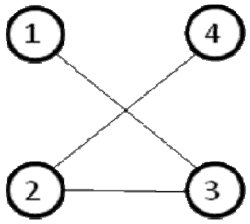
j



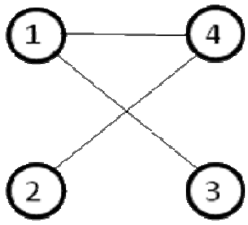
k



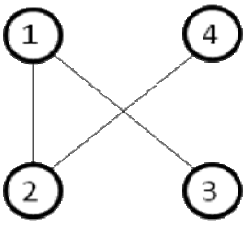
l



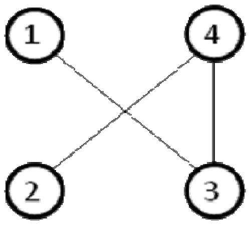
m



n



o



p

MINIMUM-COST SPANNING TREES

Let G=(V,E) be an undirected connected graph. A sub graph t=(V,E’) of G is a spanning tree of G iff t is a tree.

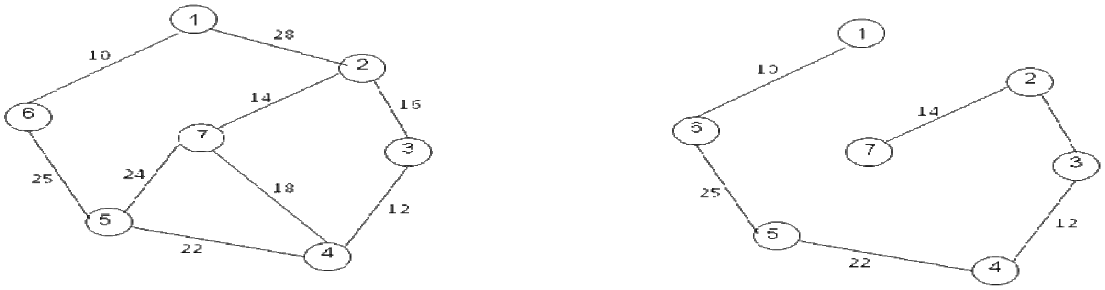


Figure 2) A graph and its minimum cost spanning tree

Applications of Spanning Trees:

1) They can be used to obtain an independent set of circuit equations for an electric network.

2) Using the property of spanning trees that a spanning tree is a minimum su graph G’ of G such that V(G’)=V(G) and G’ is connected. If the nodes of G represent cities, edges of G represent possible communication links connecting the 2 cities, then minimum no of links needed to connect ‘n’ cities is (n-1) .

Given a weighted graph in which edges have weights assigned to them where weights represent cost of construction, length of link,… One need to have min total cost or minimum total length. In either case the links selected have to form a tree. If this is not so, then the selection of links contain a cycle.

The identification of min cost spanning tree involves the selection of subset of edges.

The two algorithms used to obtain minimum cost spanning trees from a given graph are

1) Prim’s Algorithm
2) Kruskal’s Algorithm

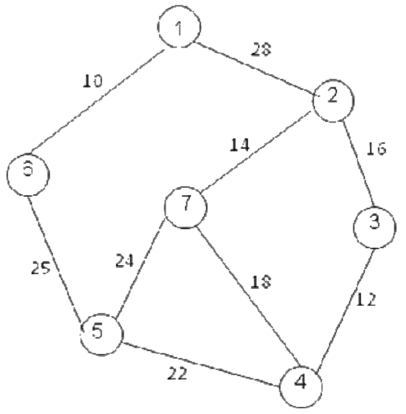
PRIM'S ALGORITHM

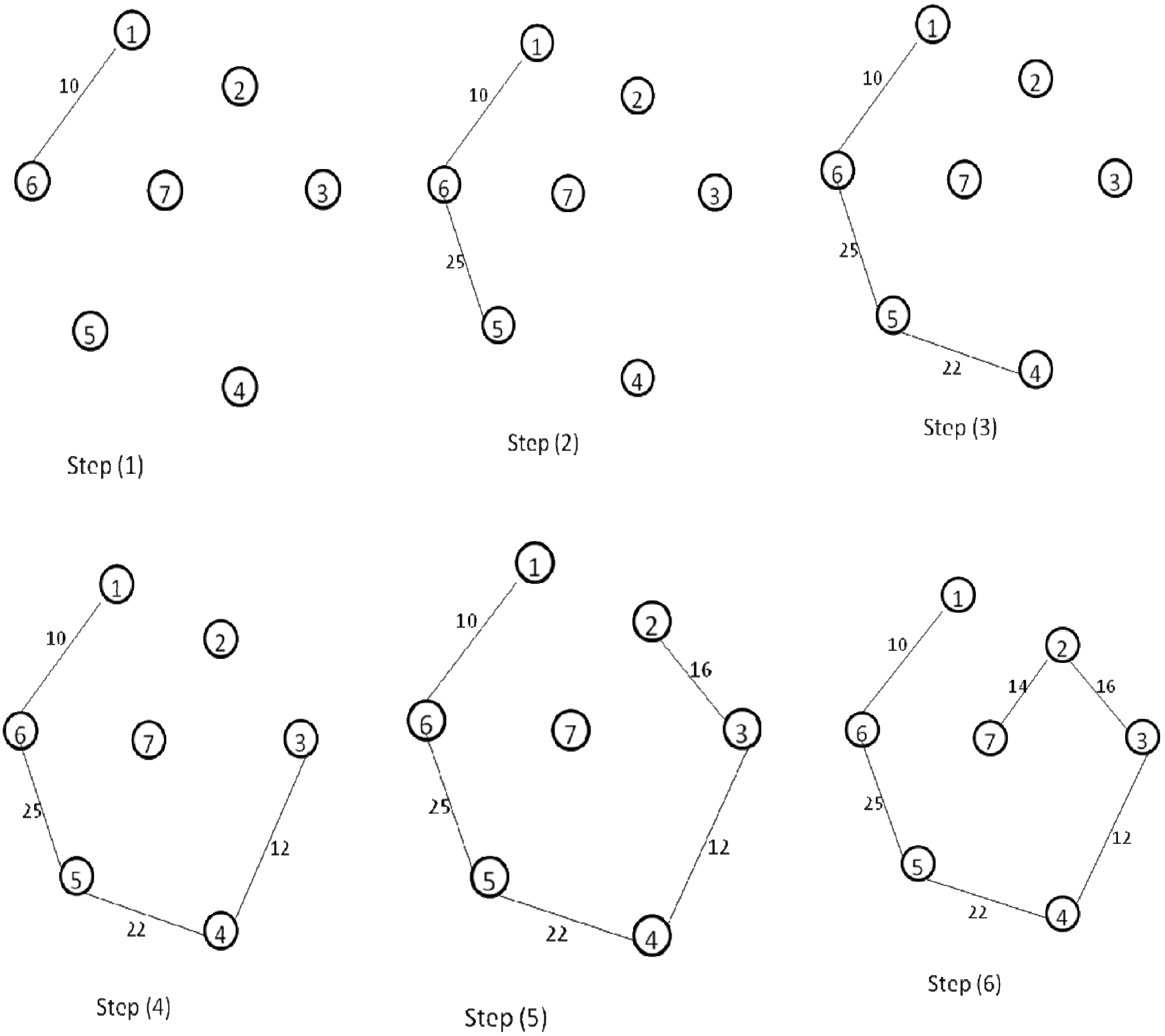
```
Algorithm prim(E, cost, n, t)
{
    let (k,l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1,1]:=k;
    t[1,2]:=l;
      for i:=1 to n do
          if (cost[i,l]<cost[i,k]) then near[i]:=l;
          else near[i]:=k;
       near[k]:=near[l]:=0;
       for i:= 2 to n-1 do
       {
           Let j be an index such that near[j] != 0 and cost[j, near[j]] is minimum
             t[I,1]:=j;
             t[I,2]:=near[j];
             mincost := mincost + cost[j,near[j]];
             near[j] := 0;
             for k:= 1 to n do
                if (near[k]!= 0) and (cost[k,near[k]]>cost[k,j]) then
                      near[k]:=j;
        }
     return mincost;
}
```

This is a greedy method to obtain a minimum cost spanning tree which builds the tree edge by edge. The next edge to be included is chosen according to a criteria i.e. choose an edge that results in minimum increase in sum of edges cost so far included.

The algorithm will start with a tree that includes only the min cost edge of 'G', then edges are added to this tree one by one. The next edge (i, j) to be added is such that 'i' a vertex already included in the tree & 'j' is a vertex not yet included, in the tree & cost (i, j) is minimum. Among all edges (i, j) efficiently,. We associate with each vertex j, a value near[j] which is a vertex in the tree such that cost [j, near[j]] is min. among all choices for next near[j]. We define near[j]=0 for all vertices j that are already in the tree. The next edge to be included is defined by vertex 'j' such that near[j]!=0 and cost[j, near[j]] is minimum.

The Time Complexity of Prim's algorithm is $O(n^2)$. The algorithm spends most of the time in finding the smallest edge. So time of the algorithm basically depends on how do we search this edge. Therefore Prim's algorithm runs in $O(n^2)$ time.




Step (1)

Step (2)

Step (3)

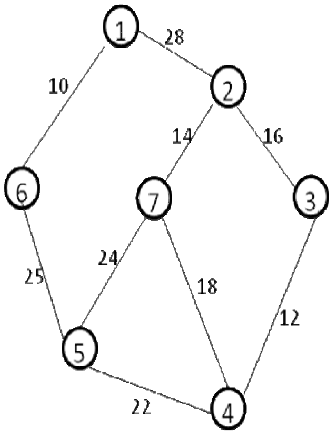
Step (4)

Step (5)

Step (6)

Figure 3) Stages in Prim's algorithm

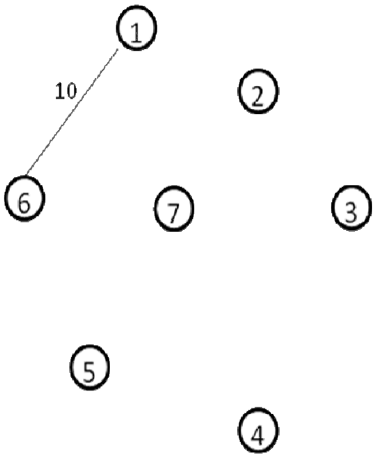
Tracing of the Prim's algorithm



Cost Matrix

	1	2	3	4	5	6	7
1	0	28	α	α	α	10	α
2	28	0	16	α	α	α	14
3	α	16	0	12	α	α	α
4	α	α	12	0	22	α	18
5	α	α	α	22	0	25	24
6	10	α	α	α	25	0	α
7	α	14	α	18	24	α	0

Minimum cost edge(k,l) = (1,6) i.e. mincost=10 select (1,6)
t[1,1] = k = 1
t[1,2] = l = 6



Step (1)

for i=1
    if cost[i,l]<cost[i,k] then near[i]=l else near[i]=k
    cost[1,6]<cost[1,1] ?
    10<0 ? no so near[i]=k i.e. near[1]=1

for i=2
    cost[2,6]<cost[2,1] ?
    α < 28 ? no so near[i]=k i.e. near[2]=1

for i=3
    cost[3,6]<cost[3,1] ?
    α < α ? no so near[i]=k i.e. near[3]=1

for i=4
    cost[4,6]<cost[4,1] ?
    α < α ? no so near[i]=k i.e. near[4]=1

for i=5
    cost[5,6]<cost[5,1] ?
    25 < α ? yes so near[i]=l i.e. near[5]=6

for i=6
    cost[6,6]<cost[6,1] ?
    0 < 10 ? yes so near[i]=l i.e. near[6]=6

for i=7
    cost[7,6]<cost[7,1] ?
    α < α ? no so near[i]=k i.e. near[7]=1

near[1]=0
near[6]=0 since edge(1,6) is included in the tree

i = 2

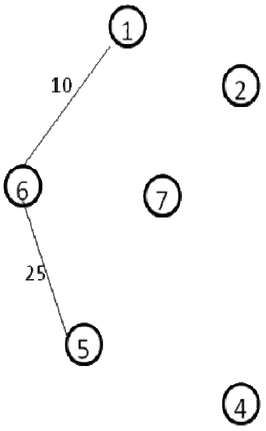
j	1	2	3	4	5	6	7
near[j]	0	1	1	1	6	0	1
cost[j, near[j]	-	28	α	α	25	--	α

we select j=5 since cost[j,near[j] i.e. cost [5,6]=25 is minimum
edge(5,6) is included

near[j]!=0
t[2,1]=5
t[2,2]=6
mincost = mincost + cost[j,near[j]]
= 10 + 25 = 35

near[5]=0

k	1	2	3	4	5	6	7
near[k]	0	1	1	1	0	0	1
cost[k, near[k]	--	28	α	α	--	--	α



Step (2)

for all k where near[k]!=0 && (cost[k,near[k]]>cost[k,j])
j=5
k=2 near[k]!=0 && cost[2,1]>cost[2,5] ?
28>α ? no

k=3 near[k]!=0 && cost[3,1]>cost[3,5] ?
α > α ? no

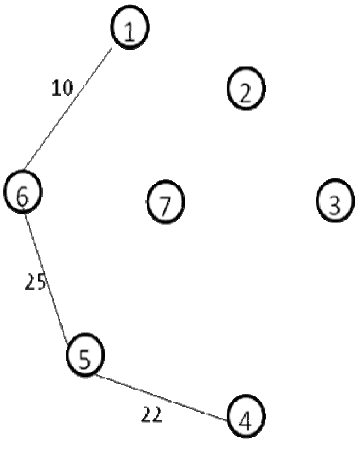
k=4 near[k]!=0 && cost[4,1]>cost[4,5] ?
α > 22 ? yes so near[k]=j i.e. near[4]=5
k=7 near[k]!=0 && cost[7,1]>cost[7,5] ?
α > 24 ? yes so near[7]=j i.e. near[7]=5

i = 3

J	1	2	3	4	5	6	7
near[j]	0	1	1	5	0	0	5
cost[j, near[j]	--	28	α	22	--	--	24

we select j=4 since cost[j,near[j]] i.e cost[4,5] = 22 is minimum edge(4,5) is included
j=4
t[3,1] = 4
t[3,2] = 5
mincost = mincost + cost[j,near[j]]
= 35+22 = 57
near[4]=0

K	1	2	3	4	5	6	7
near[k]	0	1	1	0	0	0	5
cost[k, near[k]	--	28	α	--	--	--	24



Step (3)

For all k where near[k]!=0 && (cost[k,near[k]>cost[k,j])
K=2
near[k]!=0 && cost[2,1] > cost[2,4] ?
28 > α ? no

K=3
near[k]!=0 && cost[3,1] > cost[3,4] ?
α >12 ? yes
near[k]=j i.e. near[3]=4

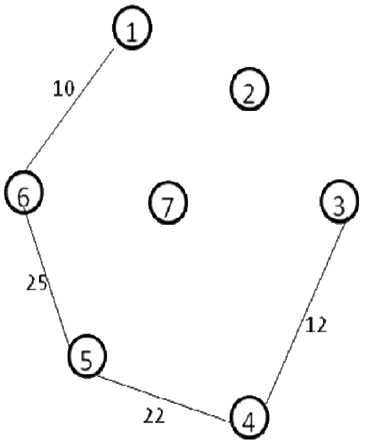
K=7
near[k]!=0 && cost[7,5] > cost[7,4] ?
24 > 18 ? yes
near[k]=j i.e. near[7]=4

i = 4

J	1	2	3	4	5	6	7
near[j]	0	1	4	0	0	0	4
cost[j, near[j]	--	28	12	--	--	--	18

we select j=3 since cost[j,near[j]]
i.e. cost[3,4]=12 is minimum
edge(3,4) is included
j=3

t[4,1] = 3
t[4,2] = 4
mincost=mincost+cost[j,near[j]]
=57+12= 69
near[3]=0



Step (4)

K	1	2	3	4	5	6	7
near[k]	0	1	0	0	0	0	4
cost[k, near[k]	--	28	--	--	--	--	18

For all k where near[k]!=0 && (cost[k,near[k]>cost[k,j])
j=3
K=2
near[k]!=0 && cost[2,1] > cost[2,3] ?
28 > 16 ? yes
Near[k]=j i.e near[2]=3

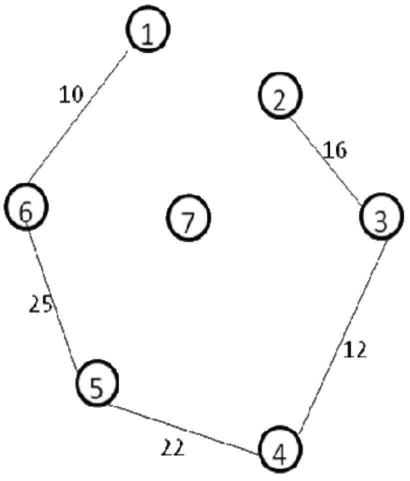
K=7
near[k]!=0 && cost[7,4] > cost[7,3] ?
18 > α ? no

i = 5

j	1	2	3	4	5	6	7
near[j]	0	3	0	0	0	0	4
cost[j, near[j]	--	16	--	--	--	--	18

we select j=2 since cost[j,near[j]]
i.e. cost[2,3]=16 is minimum
edge(2,3) is included
j=2

t[5,1] = 2
t[5,2] = 3
mincost=mincost+cost[j,near[j]]
=69+16= 85
near[2]=0



Step (5)

K	1	2	3	4	5	6	7
near[k]	0	0	0	0	0	0	4
cost[k, near[k]	--	--	--	--	--	--	18

For all k where near[k]!=0 && (cost[k,near[k]>cost[k,j])
j=2

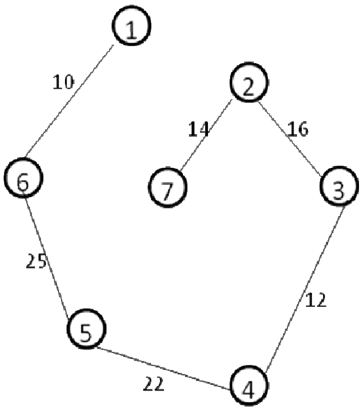
K=7
near[k]!=0 && cost[7,4] > cost[7,2] ?
18 > 14 ? yes
Near[k]=j i.e. near[7]=2

i = 6

j	1	2	3	4	5	6	7
near[j]	0	3	0	0	0	0	2
cost[j, near[j]	--	--	--	--	--	--	14

we select j=7 since cost[j,near[j]]
i.e. cost[7,2]=14 is minimum
edge(7,2) is included
j=7

t[6,1] = 7
t[6,2] = 4
mincost=mincost+cost[j,near[j]]
=85+14= 99
near[7] = 0

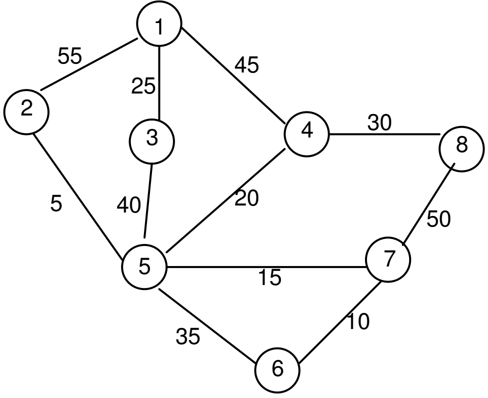


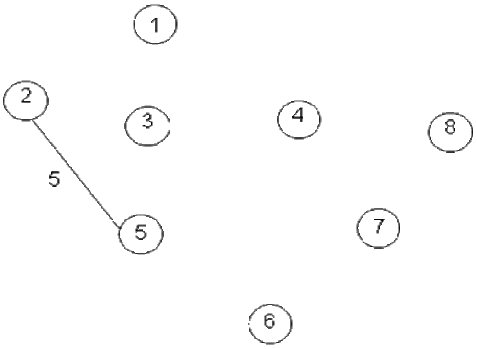
Step (6)

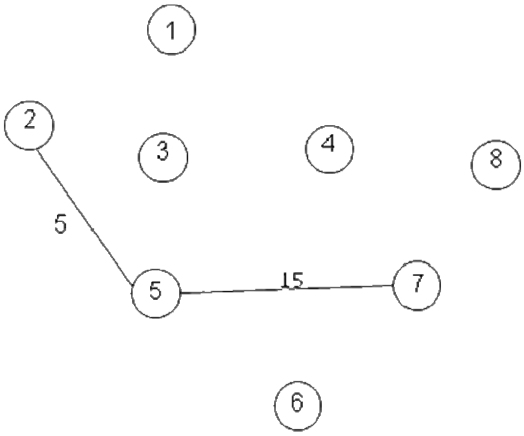
K	1	2	3	4	5	6	7
near[k]	0	0	0	0	0	0	0
cost[k, near[k]	--	--	--	--	--	--	--

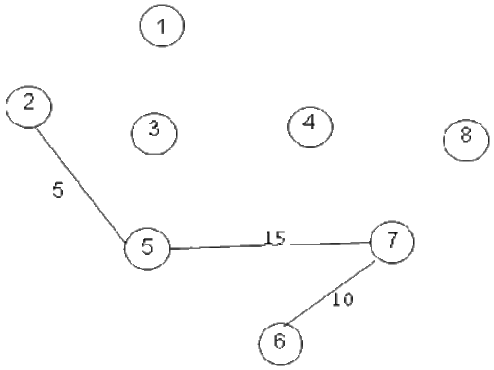
i reaches n-1 i.e. (7-1=6) the algorithm terminates and returns mincost as 99 and the edges of MST are stored in array t.

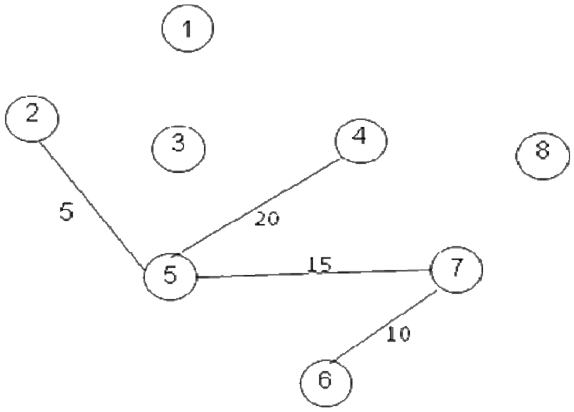
Ex 2) Minimal spanning tree using Prim's algorithm

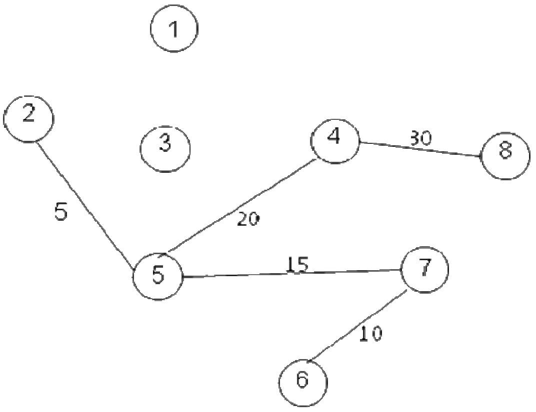


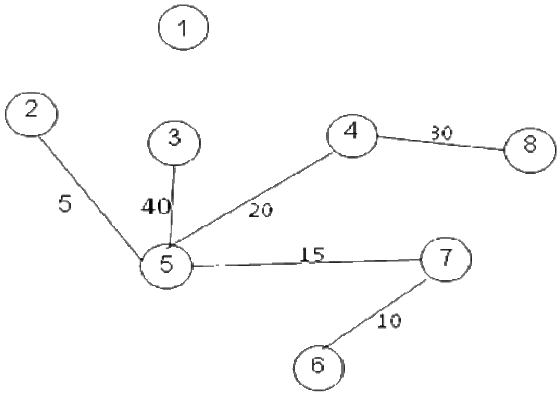


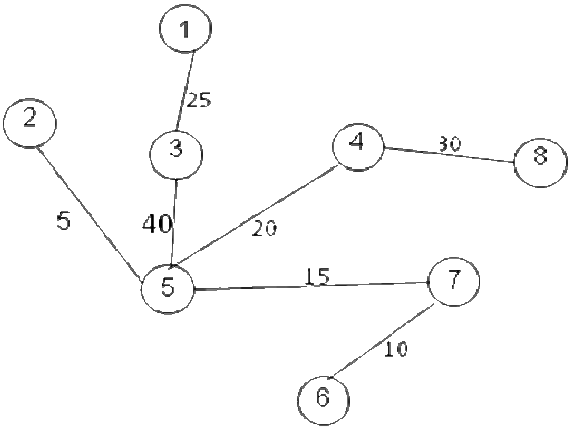












Total weight = 5+ 15 + 10 + 20 + 30 + 40 + 25 = 145

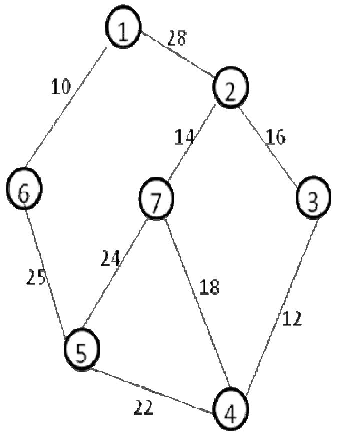
KRUSKAL’S ALGORITHM

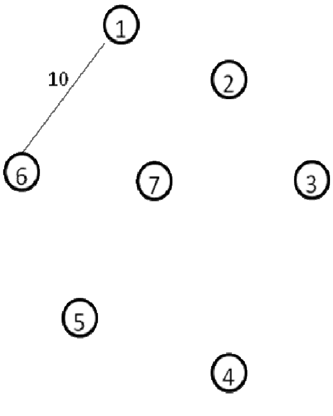
The set t(edges) is initially empty. As the algorithm progresses, edges are added to ‘t’. When ‘t’ is initially empty, each node of G forms a distinct trivial connected component. As long as no solution is found, partial graph formed by the nodes and edges in the ‘t’ consists of several connected components. The elements of t included in a given connected component form a minimum spanning tree for the nodes in this component. At the end of the algorithm only one connected component remains. So, t is then a minimum spanning tree for all nodes of G. To build bigger and bigger connected components, we examine the edges of G in the order of increasing length. If an edge joins 2 nodes in different connected components, we add it to t. Consequently, the 2 connected components now form a simple one. Otherwise the edge is rejected.

To construct a minimal spanning tree, we use the following procedure.

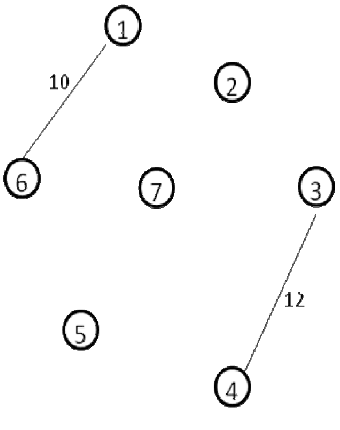
1) Arrange all edges in the increasing order of weight
2) Select an edge with minimum weight. This is the first edge of spanning tree T to be constructed.
3) Select the next edge with minimum weight that do not form a cycle with the edges already included in T.
4) Continue step 3 until T contains (n-1 edges, where n is the number of vertices of G. Arranging the edges in increasing order of their weights.

Edge	Cost
{1,6}	10
{3,4}	12
{2,7}	14
{2,3}	16
{7,4}	18
{5,4}	22
{7,5}	24
{6,5}	25
{1,2}	28

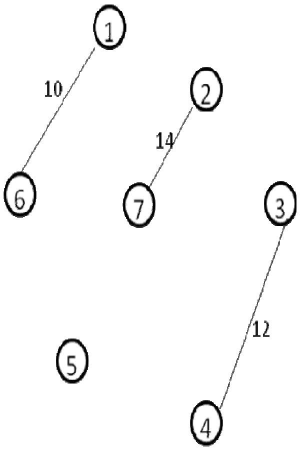




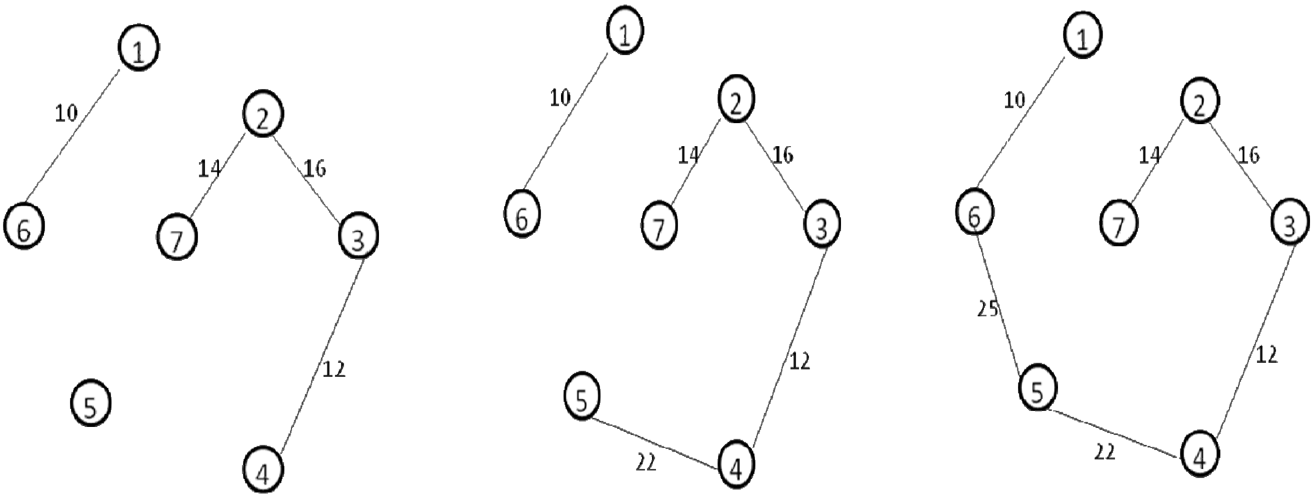
Step (1)



Step (2)



Step (3)



Step (4) Step (5) Step (6)

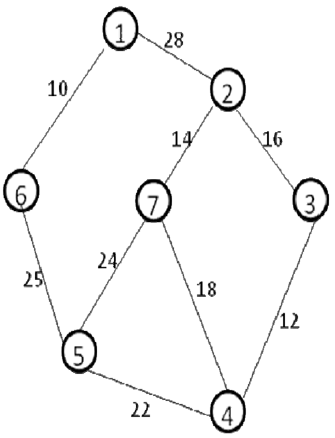
Figure 4) Stages in Kruskal's algorithm

```
Algorithm Kruskal(E,cost,n,t)
{
   Construct a heap out of the edge costs using Heapify;
   for i:= 1 to  n do
            parent[i]:= -1;
    i:=0;
    mincost := 0.0;
     while ((i<n-1) and heap not empty)) do
       {
           delete a minimumcost edge (u,v) from heap;
           and reheapify using Adjust;
           j:=find(u);
           k:=find(v);
           if (j!=k) then
             {
                i:=i+1;
               t[i,1]:=u;
               t[i,2]:=v;
               mincost := mincost+cost[u,v];
               union(j,k);
             }
        }
     if (i!= n-1) then
            write ("no spanning tree");
      else
           return mincost;
}
```

The computing time of Kruskal's algorithm is O(E log n).
Where E is the number of edges.

Tracing of the Kruskal's algorithm for MST

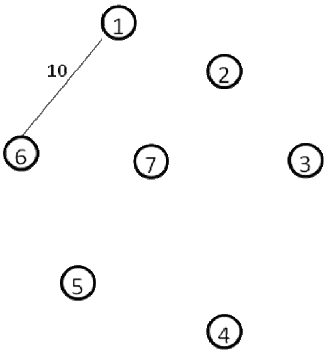
Edge	Cost
{1,6}	10
{3,4}	12
{2,7}	14
{2,3}	16
{7,4}	18
{5,4}	22
{7,5}	24
{6,5}	25
{1,2}	28



Initialization of all vertices as roots

parent	-1	-1	-1	-1	-1	-1	-1
vertex	1	2	3	4	5	6	7

i=0
mincost=0
(u,v)=(1,6) with a cost of 10
j=find(1) = 1
k=find(6) = 6
as j!=k include the edge in the spanning tree
i=1
t[1,1]=1
t[1,2]=6
mincost=0+10=10
union(1,6)



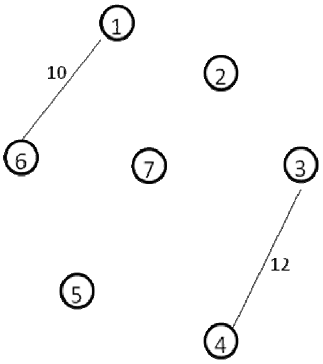
Step (1)

Tree matrix

t	1	6

parent	-1	-1	-1	-1	-1	1	-1
vertex	1	2	3	4	5	6	7

i=1
mincost=10
(u,v)=(3,4) with a cost of 12
j=find(3) = 3
k=find(4) = 4
as j!=k include the edge in the spanning tree
i=2
t[2,1]=3
t[2,2]=4
mincost=10+12=22
union(3,4)



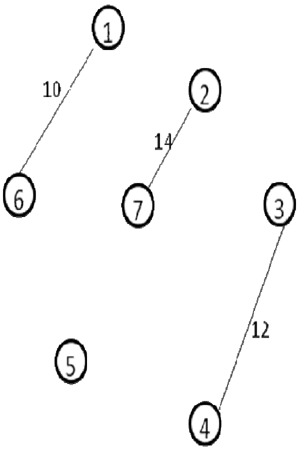
Step (2)

Tree matrix

t	1	6
	3	4

parent	-1	-1	-1	3	-1	1	-1
vertex	1	2	3	4	5	6	7

i=2
mincost=22
(u,v)=(2,7) with a cost of 14
j=find(2) = 2
k=find(7) = 7
as j!=k include the edge in the spanning tree
i=3
t[3,1]=2
t[3,2]=7
mincost=22+14=36
union(2,7)



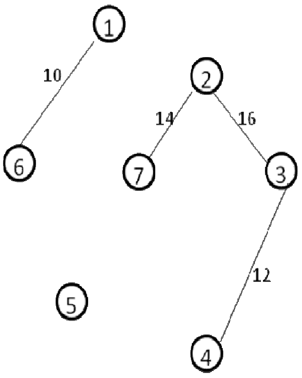
Step (3)

Tree matrix

t	1	6
	3	4
	2	7

parent	-1	-1	-1	3	-1	1	2
vertex	1	2	3	4	5	6	7

i=3
mincost=36
(u,v)=(2,3) with a cost of 16
j=find(2) = 2
k=find(3) = 3
as j!=k include the edge in the spanning tree
i=4
t[4,1]=2
t[4,2]=3
mincost=36+16=52
union(2,3)



Step (4)

Tree matrix

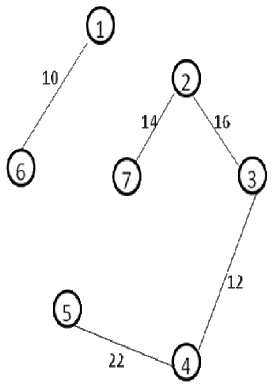
t	1	6
	3	4
	2	7
	2	3

parent	-1	-1	2	2	-1	1	2
vertex	1	2	3	4	5	6	7

i=4
mincost=54
(u,v)=(7,4) with a cost of 18
j=find(7) = 2
k=find(4) = 2
as j=k inclusion of this edge (7,4) forms a cycle in the MST so we discard this edge

parent	-1	-1	2	3	-1	1	2
vertex	1	2	3	4	5	6	7

i=5
mincost=54
(u,v)=(5,4) with a cost of 22
j=find(5) = 5
k=find(4) = 3
as j!=k include the edge in the spanning tree
i=5
t[5,1]=5
t[5,2]=4
mincost=52+22=74
union(5,4)



Step (5)

Tree matrix

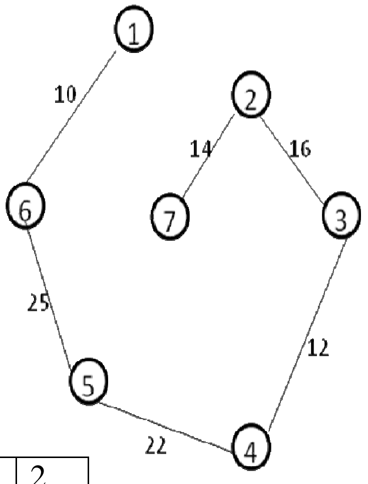
t	1	6
	3	4
	2	7
	2	3
	5	4

parent	-1	-1	2	3	4	1	2
vertex	1	2	3	4	5	6	7

i=5
mincost=74
(u,v)=(7,5) with a cost of 24
j=find(7) = 2
k=find(5) = 2
as j=k inclusion of this edge forms a cycle discard this edge

parent	-1	-1	2	3	4	1	2
vertex	1	2	3	4	5	6	7

i=5
mincost=74
(u,v)=(6,5) with a cost of 25
j=find(6) = 1
k=find(5) = 2
as j!=k include the edge in the spanning tree
i=6
t[6,1]=6
t[6,2]=5
mincost=74+25=99
union(6,5)



Step (6)

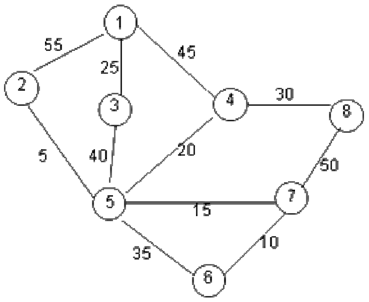
Tree matrix

t	1	6
	3	4
	2	7
	2	3
	5	4
	6	5

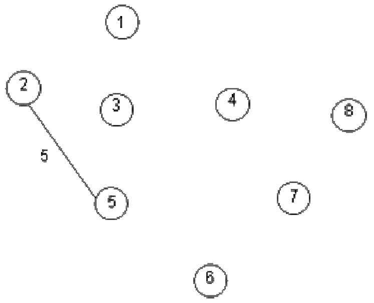
parent	6	-1	2	3	4	5	2
vertex	1	2	3	4	5	6	7

The minimum cost spanning tree is with 99

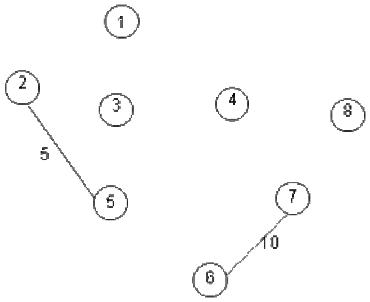
Kruskal’s Algorithm



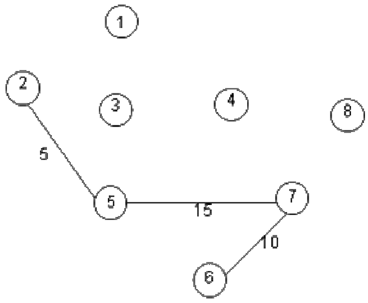
Graph



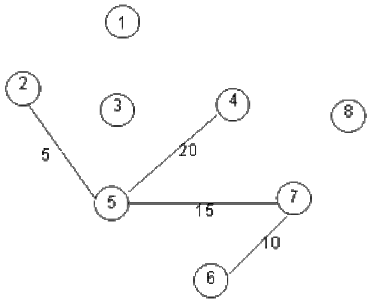
Step 1



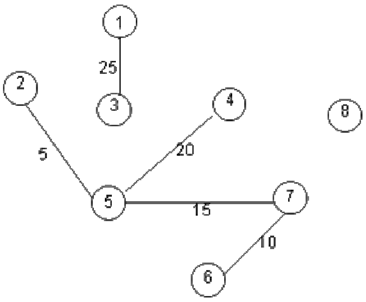
Step 2



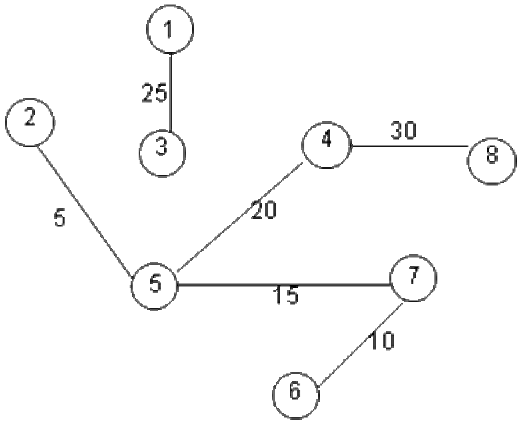
Step 3



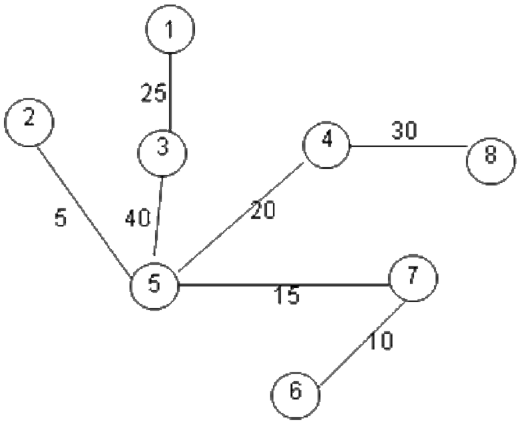
Step 4



Step 5



Step 6



Step 7

Total Weight = 5- 10-15-20-25-30-40 = 145

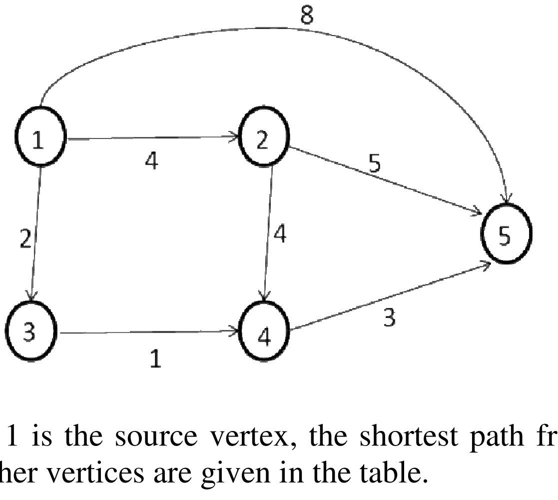
SINGLE SOURCE SHORTEST PATH PROBLEM

Let G=(V,E) be a directed graph with weighting function w for the edges of G. The starting vertex of the path is called the source and the last vertex is called the destination. Let v be any other vertex which belongs to set of vertices V. The problem to determine a shortest path to given destination vertex v from source is called single source shortest path problem.

Dijkstra's Algorithm

```
Algorithm shortestpath(v,cost,dist,n)
// dist[i], 1<=i<=n is the distance or short path starting from source passing through the
// vertices that are in S and ending at i
{
   for i:= 1 to n do
    {
       S[i]:=0;                                //1 for true ; 0 for false
        dist[i]:= cost[v,i];
     }

     S[v]:=1;                                  //1 for true 0 for false
     dist[v]:=0;

     for k:= 2 to n-1 do
      {
          choose u from among those vertices not in S such that dist[u] is minimum;
           S[u]:=1;   // put u in S.
           for (each w adjacent to u with s[w] =0) do
             {
                     if (dist[w] > dist[u]+ cost[u,w]) then
                            dist[w]:=dist[u]+cost[u,w];
             }
       }
}
```

Example 1) Find shortest path from node 1 to all other nodes



Path	length
1,2	4
1,3	2
1,3,4	3
1,3,4,5	6

If 1 is the source vertex, the shortest path from 1 to 5 is 6. The shortest path from 1 to all other vertices are given in the table.

The greedy method to generate shortest paths from source vertex to the remaining vertices is to generate these paths in increasing order of path length.

According to Dijkstra’s algorithm, first we select a source vertex and include that vertex in the set S. To generate the shortest paths from source to the remaining vertices a shortest path to the nearest vertex is generated first and it is included in S. Then a shortest path to the second nearest vertex is generated and so on. To generate these shortest paths we need to determine,

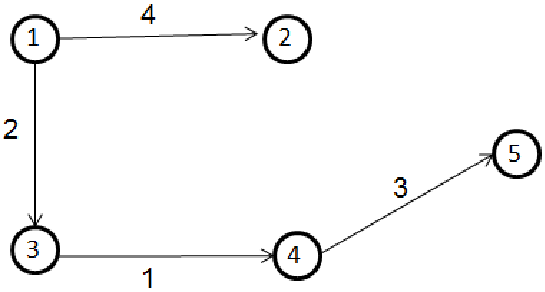
1) The next vertex to which a shortest path must be generated.
2) A shortest path to this vertex.

The first for loop takes O(n) time. Each executin of second for loop requires O(n) time to select the next vertex and again at the for loop to update dist. So that total time for this loop is O(n^2). Therefore time complexity for this algorithm is O(n^2).

interaction	set	Vertex selected	Distance				
			1	2	3	4	5
initial	--	--	0	4	2	∞	8
1	{1}	3	0	4	2	3	8
2	{1,3}	4	0	4	2	3	6
3	{1,3,4}	2	0	4	2	3	6
4	{1,3,4,2}	5					

Tracing the Algorithm

S[1]=0,S[2]=0,S[3]=0,S[4]=0,S[5]=0
dist[1]=cost[1,1]=0
dist[2]=cost[1,2]=4
dist[3]=cost[1,3]=2
dist[4]=cost[1,4]=∞
dist[5]=cost[1,5]=8

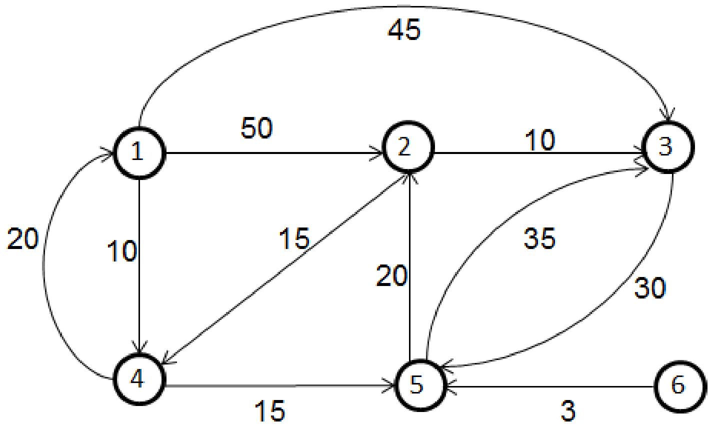


Initially set S is empty. i.e. S={}
as we want to found shortest distance for node 1 to all other nodes the source node i.e node 1 is included in the set S
S={1}
We search for the nearest node from 1 which is node 3.
Node 3 is included in the set i.e. S={1,3}
Now find all adjacent vertices of node 3 other than in set S..
Node 4 is adjacent of node 3

```
if (dist[4] > dist[3]+ cost[3,4]) then
    dist[4]:=dist[3]+cost[3,4];
```

the nearest node is selected and added to S.
S={1,3,4}
Usually this is repeated for all the adjacent vertices other than the nodes in S.
Now find all adjacent nodes of 4 other than the nodes in S.
Node 5 is adjacent
The dis[5] is modified
The node with smallest dist is selected.
Node 2 is selected and added to S
S={1,3,4,2}
The remaining node is 5
S={1,3,4,2,5}
The paths from 1 to all other nodes is shown in the spanning tree.

Example 2) Single source shortest path problem: Find shortest path from node 1 to all other nodes.



Path	Length
1,4	10
1,4,5	25
1,4,5,2	45
1,3	45

Consider the above directed graph. The numbers on the edges are weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is 1-4-5-2. The length of this path is 10 + 15 + 20 = 45. Even though there are three edges on this path it is shorter than the path 1,2 which is of length 50. There is no path from 1 to 6.

To formulate greedy based algorithm to generate shortest paths, we must conceive of a multi stage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated.

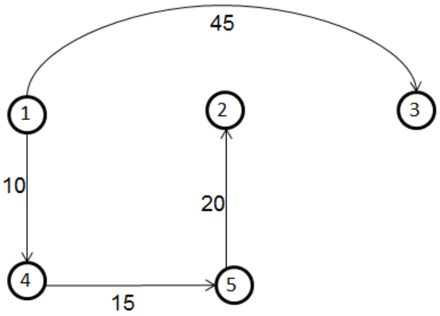
The algorithm known as Dijkstra's algorithm determines the lengths of the shortest paths from V_0 to all other vertices in G. it is assumed that the n vertices are numbered from 1 through n. The set S is maintained as a bit array with S[i]=0 if vertex I is not in S and S[i]=1 if it is.

It is assumed that the graph itself is represented by its cost adjacency matrix with cost[I,j] being the weight of the edge <i,j>. The weight cost[i,j] is set to some large number, ∞, in case the edge <i,j> is not in E(G). For i=j cost[i,j] can be set to nonnegative number without affecting the outcome of the algorithm.

The time taken by the algorithm on a graph with n vertices is $O(n^2)$.

interaction	set	Vertex selected	Distance					
			1	2	3	4	5	6
initial	--	--	0	50	45	10	∞	∞
1	{1}	4	0	50	45	10	25	∞
2	{1,4}	5	0	45	45	10	25	∞
3	{1,4,5}	2	0	45	45	10	25	∞
4	{1,4,5,2}	3	0	45	45	10	25	∞

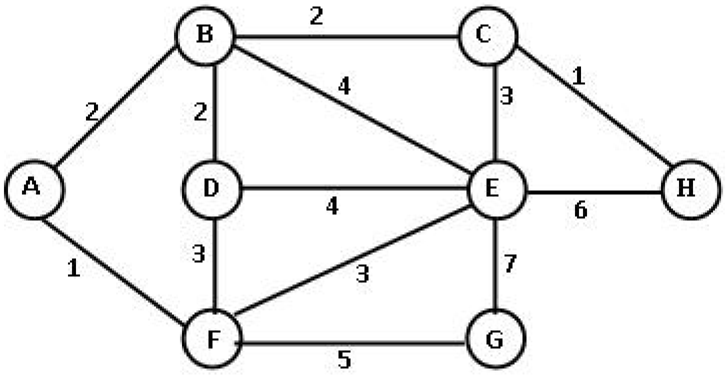
Shortest paths from 1 in increasing order
1-4=10
1-5=1-4-5=25
1-2=1-4-5-2=45
1-3=45



Spanning tree which shows shortest paths from node 1

In Divide and Conquer approach, a problem is divided recursively into sub problems of same kind as the original problem, until they are small enough to be solved and finally the solutions of the sub problems are combined to get the solution of the original problem. In Greedy approach, a problem is solved by determining a subset to satisfy some constraints. If that subset satisfies the given constraints, then it is called as feasible solution, which maximizes or minimizes a given objective function. A feasible solution that either maximizes or minimizes an objective function is called as optimal solution.

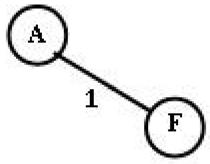
Single Source Shortest Path Problem



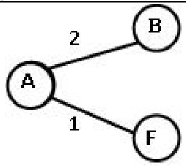
There are many paths from A to H.
For example length of path A F D E H = 1 + 3 + 4 + 6 = 14
A B C E H = 2 + 2 + 3 + 6 = 13
We may further look for a path with length shorter than 13 if exists.

Algorithm:

1. We start with source vertex A.
2. We locate the vertex closest to it. B F are adjacent vertices. Length of AF < length of AB so we choose F.

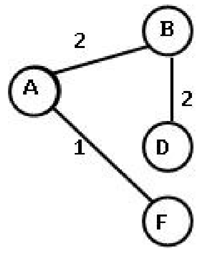


3. Now we look for all the adjacent vertices excluding the just earlier vertex of newly added vertex and the remaining adjacent vertices of earlier vertices, i.e., we have D,E and G (as adjacent vertices of F) and B (as remaining adjacent vertex of A).

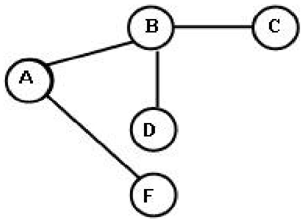
Vertices that may be attached	Path from A	Length	
D	AFD	4	
E	AFE	4	
G	AFG	6	
B	AB	2	

We choose vertex B.

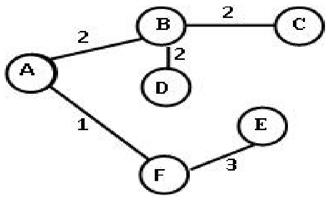
4) We go back to step 3 and continue till we exhaust all the vertices.

Vertices that may be attached	Path from A	Length	
D	ABD AFD	4 4	
G	AFG	6	
C	ABC	4	
E	ABE	6	
B	AFE	4	

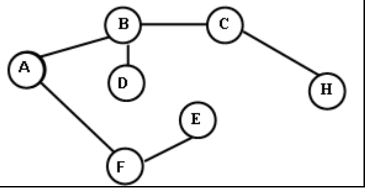
We may choose D, C or E.
We choose say D through B.

Vertices that may be attached	Path from A	Length	
G	AFG	6	
C	ABC	4	
E	AFE ABE BDE	4 6 8	

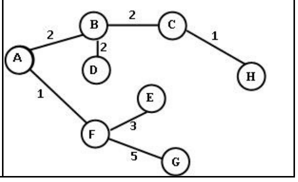
We may choose C or E, choose C.

Vertices that may be attached	Path from A	Length	
G	AFG	6	
E	AFE ABE ABDE ABCE	4 6 8 7	
H	ABCH	5	

We choose E via AFE.

Vertices that may be attached	Path from A	Length	
G	AFG AFEG	6 11	
H	ABCH AFEH	5 10	

We choose H via ABCH.

Vertices that may be attached	Path from A	Length	
G	AFG AFEG	6 11	

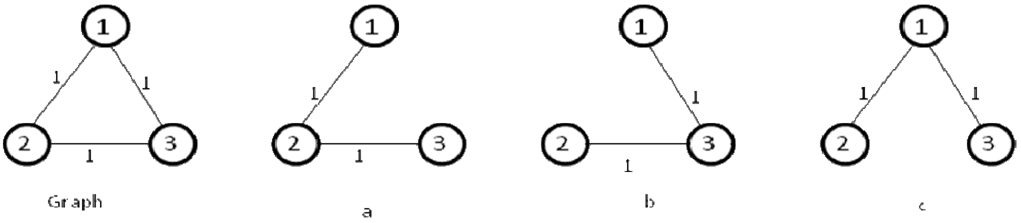
We choose path AFG.
Therefore the shortest paths from source vertex A to all the other vertices are

AB
ABC
ABD
ABCH
AF
AFE
AFG.

S.No	Divide & Conquer Method	Greedy Method
1	Divide and conquer approach is a result oriented approach	By Greedy method, there are some chances of getting an optimal solution to a specific problem
2	The time taken by this algorithm is efficient when compared by greedy method.	The time taken by this algorithm is not that much efficient when compared to divide-and-conquer approach.
3	This approach does not depend on constraints to solve a specific problem	This approach cannot make further move, if the subset chosen does not satisfy the specified constraints.
4	This approach is not efficient for larger problems	This approach is applicable and as well as efficient for a wide variety of problems.
5	As the problem is divided into large number of sub problems, the space requirement is very much large	Space requirement is less when compared to the divide-and –conquer approach.
6	This approach is not applicable to problems which are not divisible. Example Knapsack problem	This problem (Knapsack) is rectified in the greedy method.

Example) Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.

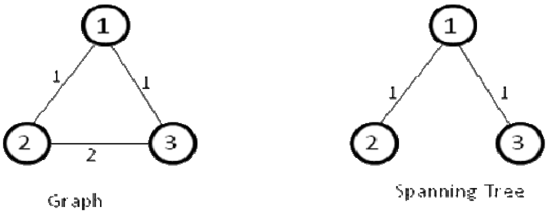
We may get so many spanning trees if weights are equal. If the weights of the connected graph are all distinct, then the minimum spanning tree is unique.



Graph a b c

Three minimum Spanning Trees

If some weights are distinct there may be only one minimum spanning tree as shown below



Graph Spanning Tree

POSSIBLE QUESTIONS

1) Differentiate between Divide and Conquer and Greedy method
2) What is spanning tree? Explain the Prim’s algorithm with an example.
3) Differentiate between Prim’s and Kruskal’s algorithms
4) Write Prim’s algorithm and also analyze its Time Complexity
5) Write Greedy algorithm to generate shortest path
6) Write dijkstra’s algorithm