# Module 3

**Greedy method:** General method, applications - Job sequencing with deadlines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem.
**Dynamic Programming:** General method, applications-Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

## Greedy Method:

The greedy method is perhaps (maybe or possible) the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.
**Feasible solution**:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.
**Optimal solution**: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.
The greedy method suggests that an algorithm 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.

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking).
**Example**: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.
**Application of Greedy Method:**
- Job sequencing with deadline
- 0/1 knapsack problem
- Minimum cost spanning trees
- Single source shortest path problem.

## Algorithm for Greedy method

```
Algorithm Greedy(a,n)
//a[1:n] contains the n
inputs
{
Solution :=0;
For i=1 to n do
{
X:=select(a);
If Feasible(solution, x) then
Solution=
Union(Solution,x)
}
Return solution;
}
```

**Selection** -------------- Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
**Feasible** ------Boolean-valued function that determines whether x can be included into the solution vector.
**Union**-------function that combines x with solution and updates the objective function.

**Greedy algorithms have some advantages and disadvantages:**

1. It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
2. **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

## <u>Knapsack Problem</u>

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

There are two version of knapsack problem

1. 0-1 knapsack problem: Items are indivisible. (either take an item or not) can be solved with dynamic programming.

2. Fractional knapsack problem: Items are divisible. (can take any fraction of an item) It can be solved in greedy method

0-1 KNAPSACK PROBLEM: A thief robbing a store finds n items.

♣ i th item: worth vi value of item and wi weight of item

- W, wi , vi are integers.
- He can carry at most W pounds.

FRACTIONAL KNAPSACK PROBLEM:

A thief robbing a store finds n items.

- i th item: worth vi value of item and wi weight of item W, wi , vi are integers.
- He can carry at most W pounds.

- He can take fractions of items.

**Applications**

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

We are going to discuss about Fractional Knapsack

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***M*** into his knapsack. There are n items available in the store and weight of ***i***$^{th}$ item is $\boldsymbol{w_i}$ and its profit is $\boldsymbol{p_i}$. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

**<u>Fractional Knapsack</u>**

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **i**$^{th}$ item wi>0
- Profit for **i**$^{th}$ item pi>0 and
- Capacity of the Knapsack is **M**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $x_i$ of **i**$^{th}$ item.

$0 \leqslant xi \leqslant 1$

The **i**$^{th}$ item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi in to the total profit.

Hence, the objective of this algorithm is to

$$\text{maximize} \sum_{n=1}^{n} (xi.pi)$$

subject to constraint,

$$\sum_{n=1}^{n} (xi.wi) \leqslant M$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n} (xi.wi) = M$$

In this context, first we need to sort those items according to the value of piwi so that

pi+1 /wi +1 ≤ pi/wi . Here, ***x*** is an array to store the fraction of items.

**<u>Example</u>**

Capacity of the bag M=15

| Item | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Profit | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| Weight | 2 | 3 | 5 | 7 | 1 | 4 | 1 |
| P/W | 5 | 1.6 | 3 | 1 | 6 | 4.5 | 3 |
| X | 1 | 2/3 | 1 | 0 | 1 | 1 | 1 |

Item is choosen based on the increasing order of Pi/Wi

X is determined based on the  pi/wi value hence

6 is maximum Pi/Wi value s

M-wi

15-1=14(item 5 ), 14-2=12(item 1), 12-4=8 (item 6) 8-1=7(item 7) 7-5=2 (item 2)

2-2/3=0(item 2)

For finding weight use: ∑(xi.wi

1*3+2/3*3+1*5+0*7+1*1+1*4.5+3*1=15

Profit is: ∑ (xi.pi) it is 55.5

**Algorithm Knapsack**

```
1. Algorithm GreedyKnapsack(m,n)
2. //p[1:n] and w[1:n] contain the profits and weights respectively of the n objects
   ordered such that p[i]/w[i] >= p[i+1]/w[i+1]
3. M is the knapsack size and x[1:n] is the solution vector
4. {
5. For i=1 to n do x[i]=0.0 //initialize x
6. U=m;
7. For i=1 to n do
8. {if(w[i]>U) then break;
9. X[i]=1.0 ;
10. U=U-w[i];
11. }
12. If(i<=n) then x[i]=U/w[i];
13. }
```

**Analysis**

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***

**MINIMUM COST SPANNING TREES**

**SPANNING TREE**: - A Sub graph ‘n’ of o graph ‘G’ is called as a spanning tree if

(i) It includes all the vertices of ‘G’

(ii) It is a tree

- **Minimum cost spanning tree:** For a given graph ‘G’ there can be more than one spanning tree. If weights are assigned to the edges of ‘G’ then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.
- The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far..

There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

Prim’s Algorithm

Kruskal’s Algorithm

**Prim’s Algorithm**: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

**Kruskal’s Algorithm:** Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

## 1.Kruskal’s algorithm

Kruskal’s Algorithm: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

In Kruskals algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

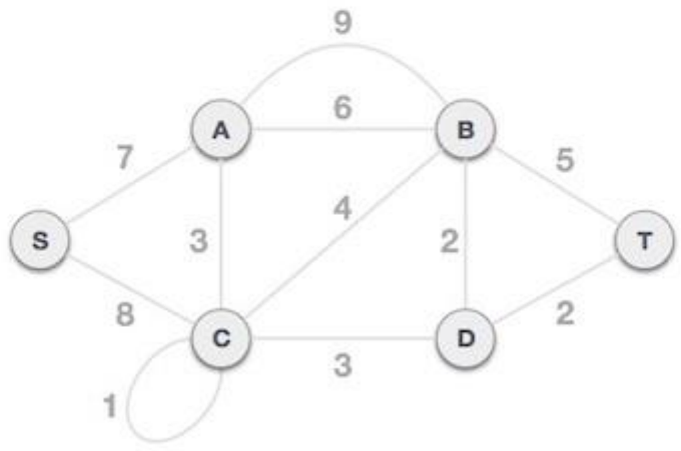
i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infect it is a forest.

iii) At the end if we include ‘n’ vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

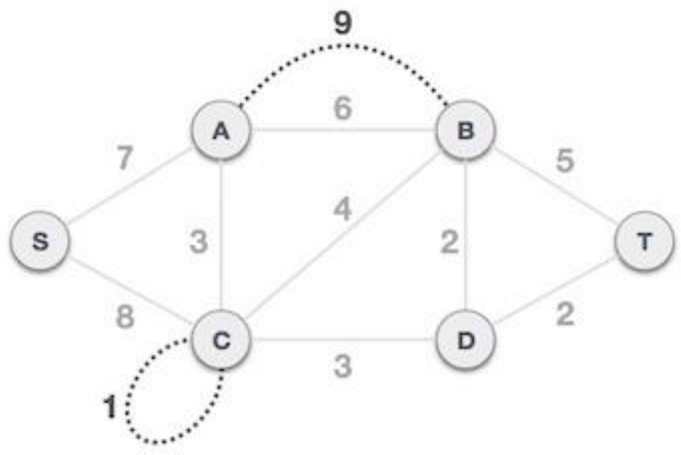
At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.

To understand Kruskal's algorithm let us consider the following example –

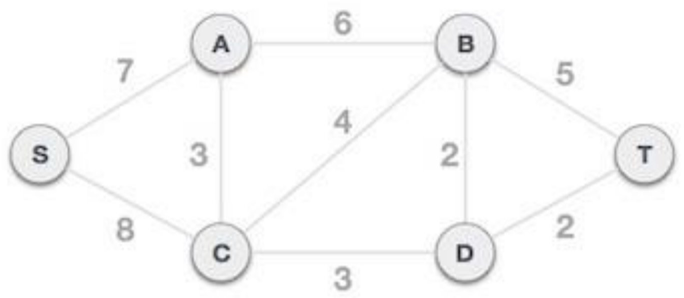


**Step 1 - Remove all loops and Parallel Edges**

Remove all loops and parallel edges from the given graph.

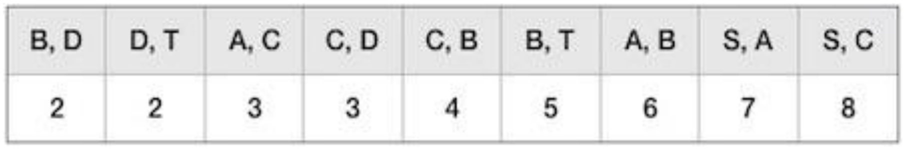


In case of parallel edges, keep the one which has the least cost associated and remove all others.



**Step 2 - Arrange all edges in their increasing order of weight**

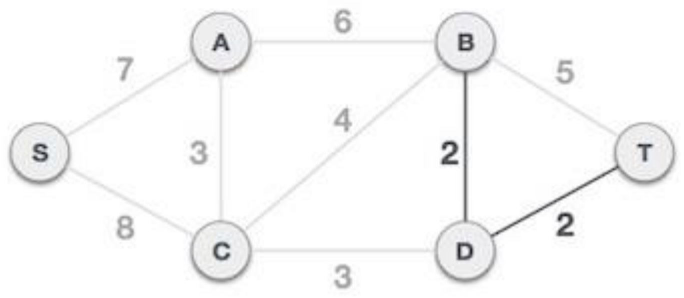
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

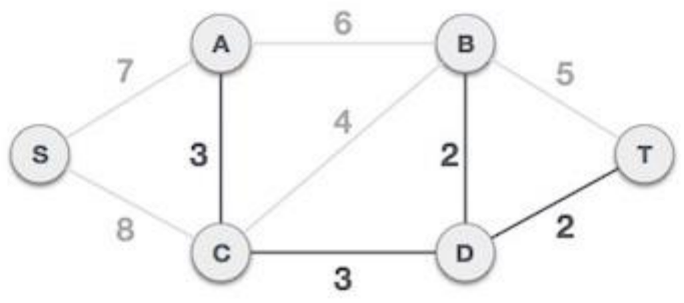
**Step 3 - Add the edge which has the least weightage**

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

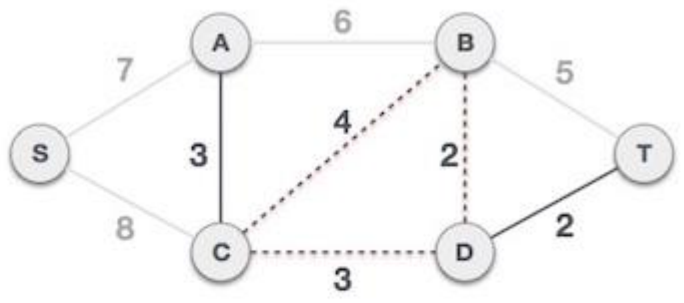


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

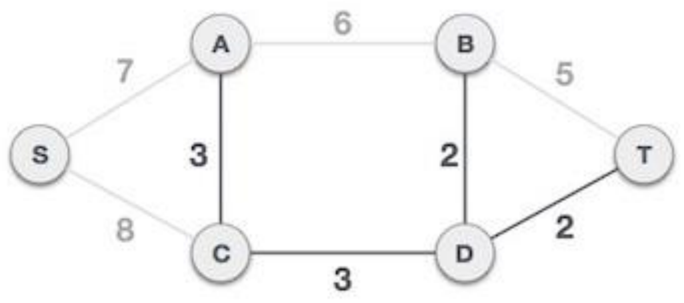
Next cost is 3, and associated edges are A,C and C,D. We add them again −



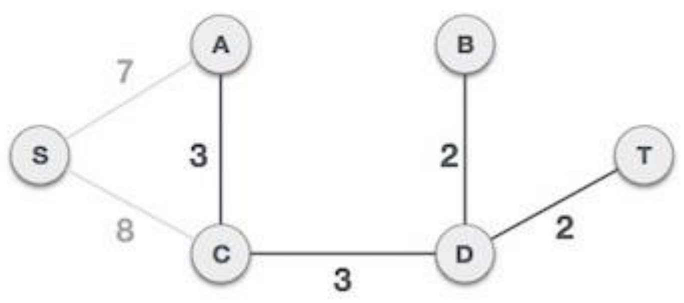
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



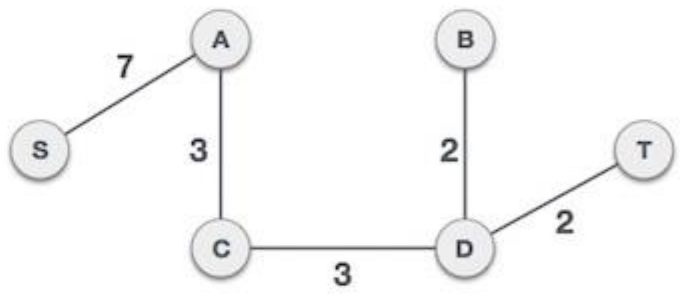
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

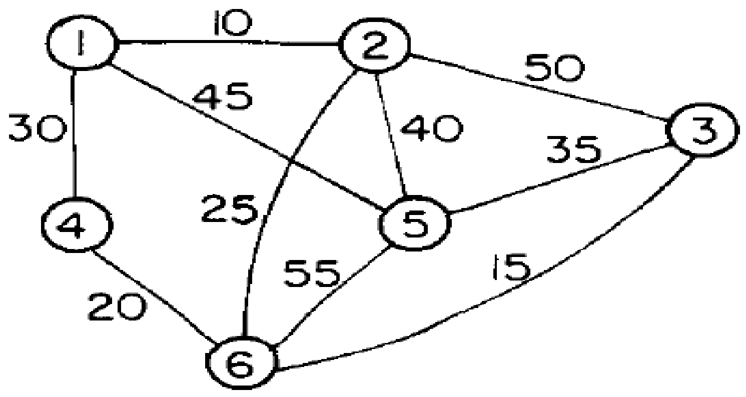


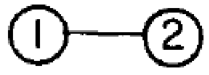
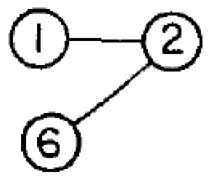
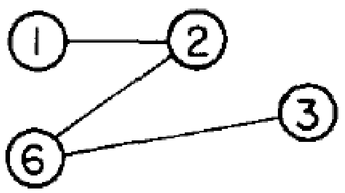
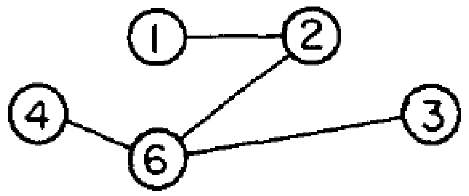
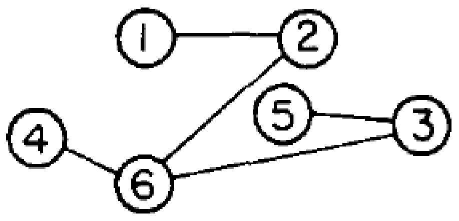
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree

```
1. Algorithm Kruskal (E, cost, n,t)
2. //E is the set of edges in G. 'G' has 'n' vertices
3. //Cost {u,v} is the cost of edge (u,v) t is the set
4. //of edges in the minimum cost spanning tree
5. //The final cost is returned
6. {
7. construct a heap out of the edge costs using heapify;
8. for i:= 1 to n do parent (i):= -1 // place in different sets
9. //each vertex is in different set {1} {1} {3}
10. i: = 0; min cost: = 0.0;
11. While (i<n-1) and (heap not empty))do
12. {
13. Delete a minimum cost edge (u,v) from the heaps; and reheapify using adjust;
14. j:= find (u); k:=find (v);
15. if (j!=k) then
16. { i: = 1+1;
17. t (i,1)=u; t (i, 2)=v;
18. mincost: = mincost+cost(u,v);
19. Union (j,k);
20. }
21. }
22. if (i!=n-1) then write ("No spanning tree");
23. else return mincost;
24. }
```

**Analysis**: - If the no/: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).

**<u>2.PRIM'S ALGORITHM:</u>** Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.



| Edge | Cost | Spanning tree |
|---|---|---|
| (1,2) | 10 |  |
| (2,6) | 25 |  |
| (3,6) | 15 |  |
| (6,4) | 20 |  |
| (1,4) | reject | |
| (3,5) | 35 |  |

Stages in Prim's Algorithm

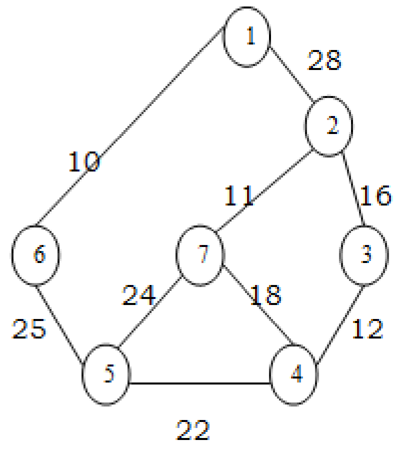
**PRIM’S ALGORITHM**: -

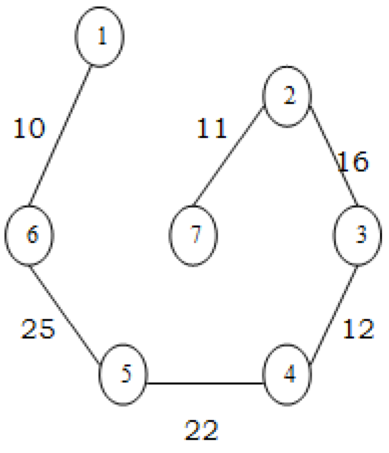
i) Select an edge with minimum cost and include in to the spanning tree.

ii) Among all the edges which are adjacent with the selected edge, select the one with minimum cost.

iii) Repeat step 2 until ‘n’ vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

***Notes:*** - At every state a decision is made about an edge of minimum cost to be included into the spanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.





**Prim's minimum spanning tree algorithm**

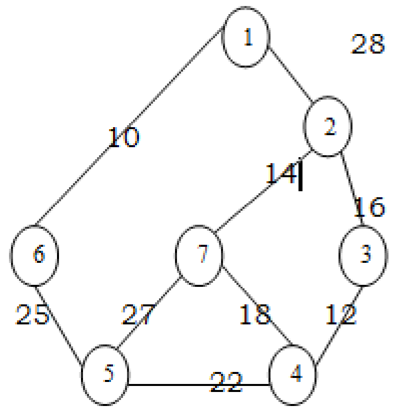
```
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost [1:n, 1:n] is the
// Cost adjacency matrix of an n vertex graph such that
// Cost [i,j] is either a positive real no. or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T[1:n-1,1: 2].
//(t [i, 1], t[i,2]) is an edge in the minimum cost spanning tree. The final cost is
returned
{
Let (k, l) be an edge with min cost in E
Min cost: = Cost [k,l];
t(1,1):= k; t (1,2):= l;
for i:= 1 to n do //initialize near
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
{ //find n-2 additional edges for t
let j be an index such that near (j) !=0 & cost (j, near (i)) is minimum;
t (i,1): = j ;t (i,2): = near (j);
min cost: = Min cost + cost (j, near (j));
near (j): = 0;
for k:=1 to n do // update near ()
if ((near (k) !=0) and (cost {k, near (k)) > cost (k,j)))
then near (k): = j;
}
return mincost;
}
```

The algorithm takes four arguments E: set of edges, cost is nxn adjacency matrix cost of (i,j)= +ve integer, if an edge exists between i&j otherwise infinity. ‘n’ is no/: of vertices. ‘t’ is a (n-1):2matrix which consists of the edges of spanning tree.

E = { (1,2), (1,6), (2,3), (3,4), (4,5), (4,7), (5,6), (5,7), (2,7) }

n = {1,2,3,4,5,6,7)

$$\begin{array}{c|ccccccc}
\text{Cost} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & \alpha & 28 & \alpha & \alpha & \alpha & 10 & \alpha \\
2 & 28 & \alpha & 16 & \alpha & \alpha & \alpha & 14 \\
3 & \alpha & 10 & \alpha & 12 & \alpha & \alpha & \alpha \\
4 & \alpha & \alpha & 12 & \alpha & 22 & \alpha & 18 \\
5 & \alpha & \alpha & \alpha & 22 & \alpha & 25 & 24 \\
6 & 10 & \alpha & \alpha & \alpha & 25 & \alpha & \alpha \\
7 & \alpha & 14 & \alpha & 18 & 24 & \alpha & \alpha
\end{array}$$

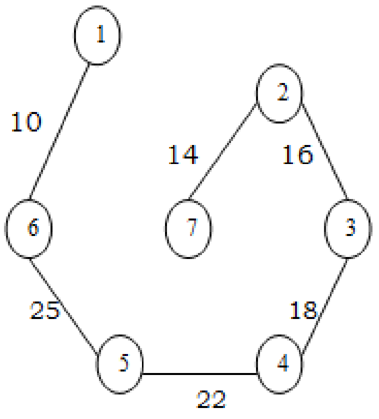


$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 1 & 6 & 5 & 4 & 3 & 2 \\
2 & 6 & 5 & 4 & 3 & 2 & 7
\end{array}$$

Start Vertex

Ending Vertex

Edges of spanning tree



i) The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.

ii) The next edge (i,j) to be added is such that i is a vertex which is already included in the treed and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to ‘i’.

iii) With each vertex ‘j’ next yet included in the tree, we assign a value near ‘j’. The value near ‘j’ represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)

iv) We define near (j):= 0 for all the vertices ‘j’ that are already in the tree.

v) The next edge to include is defined by the vertex ‘j’ such that (near (j)) □ 0 and cost of (j, near (j)) is minimum.

**Analysis**: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph ‘G’ has ‘n’ vertices then the time required by prim’s algorithm is **0(n2)**