1. GENERAL METHOD

Greedy method: It is most straight forward method. It is popular for obtaining the optimized solutions.

Optimization Problem: An optimization problem is the problem of finding the best solution (optimal solution) from all the feasible solutions (practicable of possible solutions). In an optimization problem we are given a set of constraints and an optimization functions. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called optimal solution.

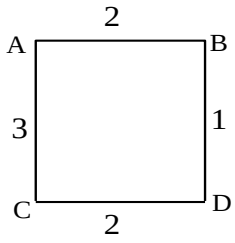
Ex: Problem: Finding a minimum spanning tree from a weighted connected directed graph G.

Constraints: Every time a minimum edge is added to the tree and adding of an edge does not form a simple circuit.

Feasible solutions: The feasible solutions are the spanning trees of the given graph G.

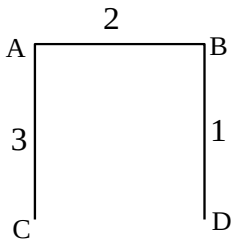
Optimal solution: An optimal solution is a spanning tree with minimum cost i.e. minimum spanning tree.

Q: Find the minimum spanning tree for the following graph.

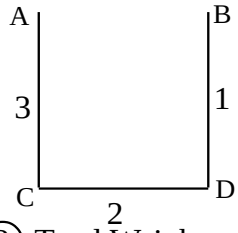


Graph G

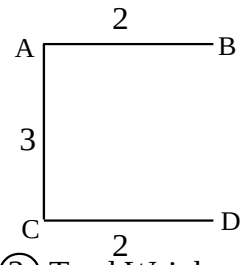
The feasible solutions are the spanning tree of the graph G. Those are



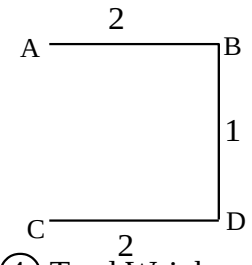
①.Total Weights=6



②.Total Weights=6



③.Total Weights=7



④.Total Weights=5

From the above spanning tree the figure 4 gives the optimal solution, because it is the spanning tree with the minimum cost i.e. it is a minimum spanning tree of the graph G.

The greedy technique suggests constructing a solution to an optimization problem hrough a sequence of steps, each expanding a partially constructed solution obtained so far until a complete solution to the problem is reached to each step, the choice made must be feasible, locally optimal and irrecoverable.

Feasible: The choice which is made has to be satisfying the problems constraints.

Locally optimal: The choice has to be the best local choice among all feasible choices available on that step.

Irrecoverable: The choice once made cannot be changed on sub-sequent steps of the algorithm (Greedy method).

Control Abstraction for Greedy Method:

```
Algorithm GreedyMethod (a, n)
{
      // a is an array of n inputs
      Solution: =Ø;
      for i: =0 to n do
      {
            s: = select (a);
            if (feasible (Solution, s)) then
            {
                  Solution: = union (Solution, s);
            }
            else
                  reject (); // if solution is not feasible reject it.
       }
       return solution;
  }
```

In greedy method there are three important activities.

1. A selection of solution from the given input domain is performed, i.e. *s:= select(a).*
2. The feasibility of the solution is performed, by using feasible '*(solution, s)*' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution.

Q: A child buys a candy 42 rupees and gives a 100 note to the cashier. Then the cashier wishes to return change using the fewest number of coins. Assume that the cashier has Rs.**1**, Rs. **5** and Rs. **10** coins.

This problem can be solved using the greedy method.

2. APPLICATION - JOB SEQUENCING WITH DEADLINES

This problem consists of ***n*** jobs each associated with a deadline and profit and our objective is to earn ***maximum profit***. We will earn profit only when job is completed on or before deadline. We assume that each job will take ***unit time*** to complete.

Points to remember:

- In this problem we have n jobs j1, j2, … jn, each has an associated deadlines are d1, d2, … dn and profits are p1, p2, ... pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.

Example: Consider the following 5 jobs and their associated deadline and profit.

index	1	2	3	4	5
JOB	j1	j2	j3	j4	j5
DEADLINE	2	1	3	2	1
PROFIT	60	100	20	40	20

Sort the jobs according to their profit in descending order.

Note! If two or more jobs are having the same profit then sorts them as per their entry in the job list.

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

Find the maximum deadline value

Looking at the jobs we can say the max deadline value is 3. So, dmax = 3

As dmax = 3 so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

time slot	1	2	3
status	EMPTY	EMPTY	EMPTY

Total number of jobs is 5. So we can write n = 5.

Note!

If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.

Similarly, if we look at job j1 it has a deadline 2. This means we have to complete job j1 on or before time slot 2 in order to earn its profit.

Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit.

Our objective is to select jobs that will give us higher profit.

time slot	1	2	3
Job	J1	J2	J4
Profit	100	60	20

Total Profit is 180

Pseudo Code:

```
for i = 1 to n do
  Set k = min(dmax, DEADLINE(i))  //where DEADLINE(i) denotes deadline of ith job
  while k >= 1 do
    if timeslot[k] is EMPTY then
      timeslot[k] = job(i)
      break
    endif
    Set k = k - 1
  endwhile
endfor
```

Algorithm:

```
int JS(int d[], int j[], int n)
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
// are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
// is the ith job in the optimal solution, 1<=i<=k.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
{
   d[0] = J[0] = 0; // Initialize.
   J[1] = 1; // Include job 1.
   int k=1;
   for (int i=2; i<=n; i++) {
   //Consider jobs in nonincreasing
   // order of p[i]. Find position for
   // i and check feasibility of insertion.
      int r = k;
      while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
      if ((d[J[r]] <= d[i]) && (d[i] > r)) {
         // Insert i into J[].
         for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
         J[r+1] = i; k++;
      }
   }
   return (k);
}
```

Time Complexity = O (n^2)

3. APPLICATION - KNAPSACK PROBLEM

In this problem the objective is to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.

Points to remember:

In this problem we have a Knapsack that has a weight limit W

There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it v1, v2, ..., vn

Our objective is to maximise the benefit such that the total weight inside the knapsack is at most W. And we are also allowed to take an item in fractional part.

$$\max \sum_{0 \le i < n} p_i x_i$$

$$s.t.$$

$$\sum_{0 \le i < n} w_i x_i \le M$$

$$0 \le x_i <= 1$$

$$p_i \ge 0, w_i \ge 0, \; 0 \le i < n$$

Example: Assume that we have a knapsack with max weight capacity, W = 16. Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5

Steps

1. Calculate value per weight for each item (we can call this value density)
2. Sort the items as per the value density in descending order
3. Take as much item as possible not already taken in the knapsack

Compute density = (value/weight)

ITEM	WEIGHT	VALUE	DENSITY
i1	6	6	1.000
i2	10	2	0.200
i3	3	1	0.333
i4	5	8	1.600
i5	1	3	3.000
i6	3	5	1.667

Sort the items as per density in descending order

ITEM	WEIGHT	VALUE	DENSITY
i5	1	3	3.000

i6	3	5	1.667
i4	5	8	1.600
i1	6	6	1.000
i3	3	1	0.333
i2	10	2	0.200

Now we will pick items such that our benefit is maximum and total weight of the selected items is at most W.

Our objective is to fill the knapsack with items to get maximum benefit without crossing the weight limit W = 16.

How to fill Knapsack Table?

is WEIGHT(i) + TOTAL WEIGHT <= W
if its YES
then we take the whole item

How to find the Benefit?

If an item value is 10 and weight is 5
And if you are taking it completely
Then,
benefit = (weight taken) x (total value of the item / total weight of the item)

weight taken = 5 (as we are taking the complete (full) item, no fraction)
total value of the item = 10
total weight of the item = 5

So, benefit = 5 x (10/5) = 10
On the other hand if you are taking say, 1/2 of the item
Then,
weight taken = 5 x (1/2) = 5/2 (as we are taking 1/2 item)
So, benefit = (weight taken) x (total value of the item / total weight of the item)
= (5/2) x (10/5)
= 5

Values after calculation

ITEM	WEIGHT	VALUE	TOTAL WEIGHT	TOTAL BENEFIT
i5	1	3	1.000	3.000
i6	3	5	4.000	8.000

i4	5	8	9.000	16.000
i1	6	6	15.000	22.000
i3	1	0.333	16.000	22.333

So, total weight in the knapsack = 16 and total value inside it = 22.333336

n=3, m=20, (p_1,p_2,p_3)=(25,24,15), (w_1,w_2,w_3)=(18,15,10)

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Algorithm:

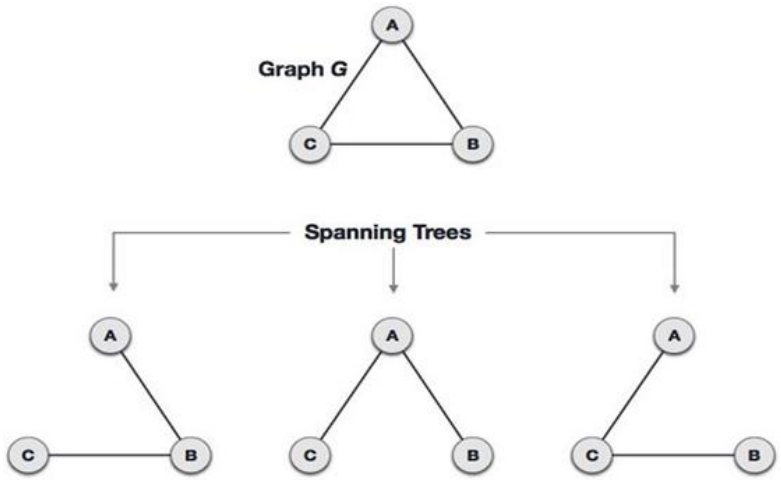
```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
   for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
   float U = m;
   for (i=1; i<=n; i++) {
      if (w[i] > U) break;
      x[i] = 1.0;
      U -= w[i];
   }
   if (i <= n) x[i] = U/w[i];
}
```

Time Complexity = O (n^2)

4. APPLICATION - MINIMUM SPANNING TREE

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

Note: Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

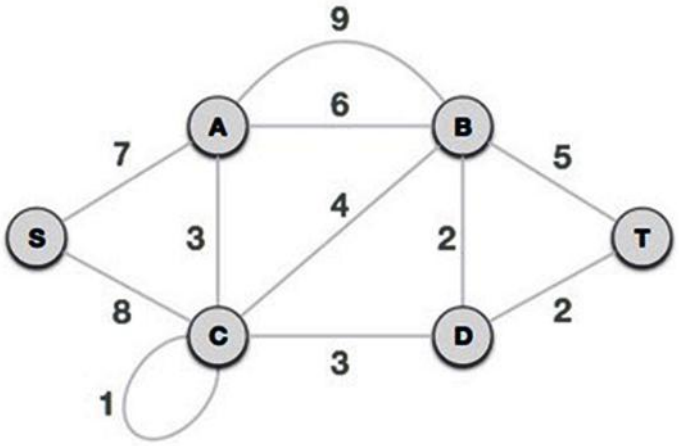
We shall learn about two most important spanning tree algorithms(greedy algorithms):

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

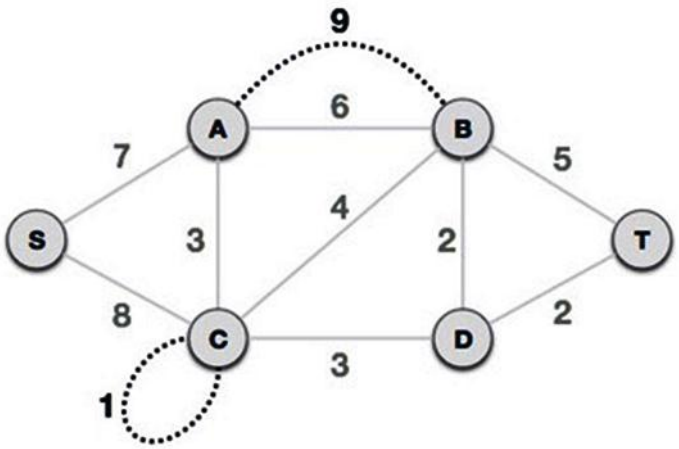
i. Kruskal's Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

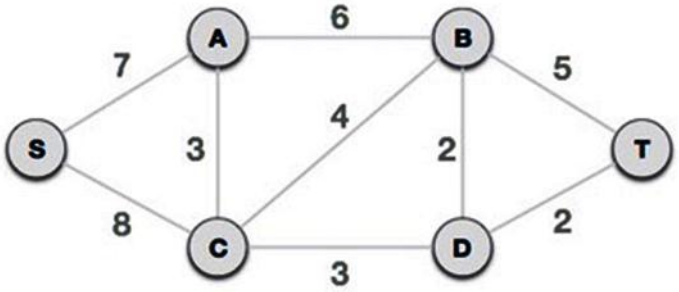
To understand Kruskal's algorithm let us consider the following example:



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.



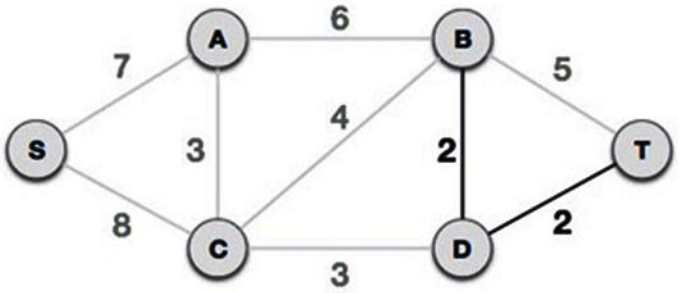
Step 2 - Arrange all edges in their increasing order of weight

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

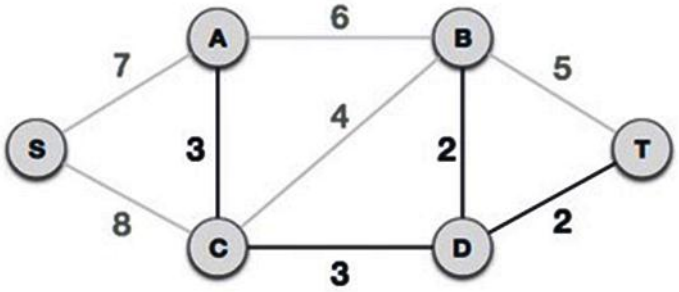
Step 3 - Add the edge which has the least weightage

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.

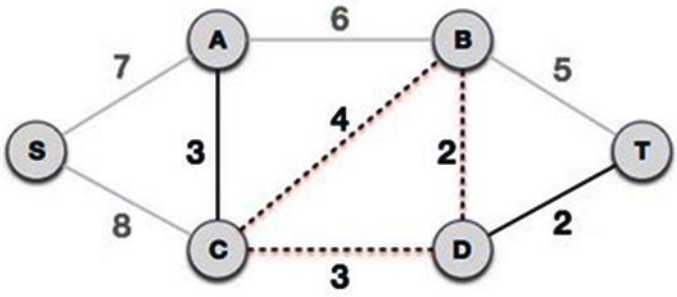


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

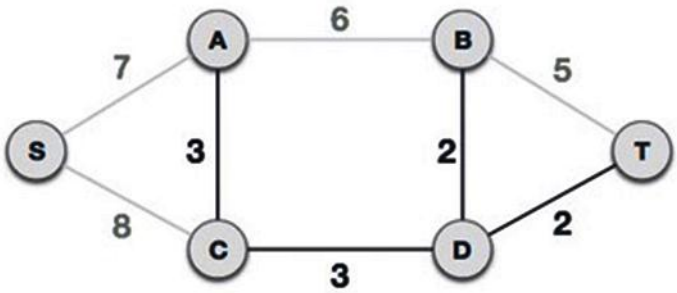
Next cost is 3, and associated edges are A,C and C,D. We add them again −



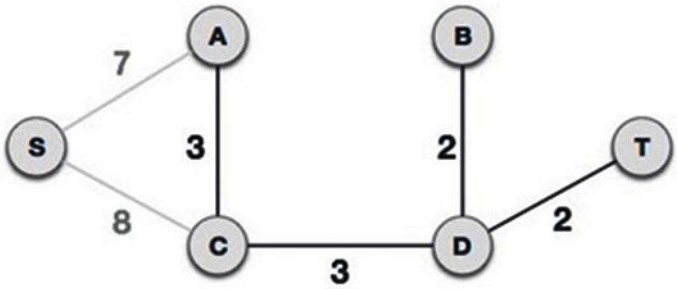
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



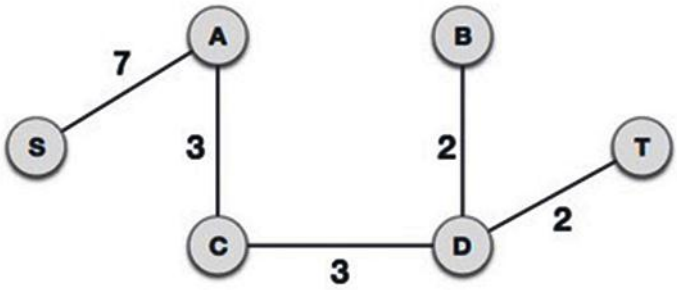
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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.

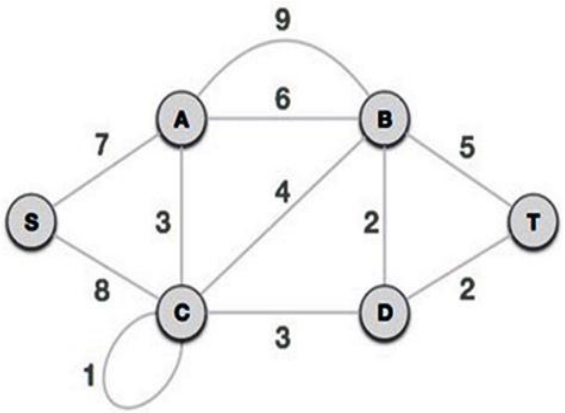
```
float Kruskal(int E[][SIZE],  float cost[][SIZE],  int n,  int t[][2])
{
   int parent[SIZE];
   construct a heap out of the edge costs using Heapify;
   for (int i=1; i<=n; i++)  parent[i] = -1;
   //  Each vertex is in a different set.
   i = 0; float mincost = 0.0;
   while ((i < n-1) && (heap not empty)) {
      delete a minimum cost edge (u,v) from the heap
      and reheapify using Adjust;
      int j = Find(u); int k = Find(v);
      if  (j != k)  {
            i++;
            t[i][1] = u;  y[i][2] = v;
            mincost += cost[u][v];
            Union(j, k);
      }
   }
    if  ( i !=  n-1) cout << "No spanning tree" << endl;
    else  return(mincost);
}
```

Time Complexity = O (|E| log |E|)

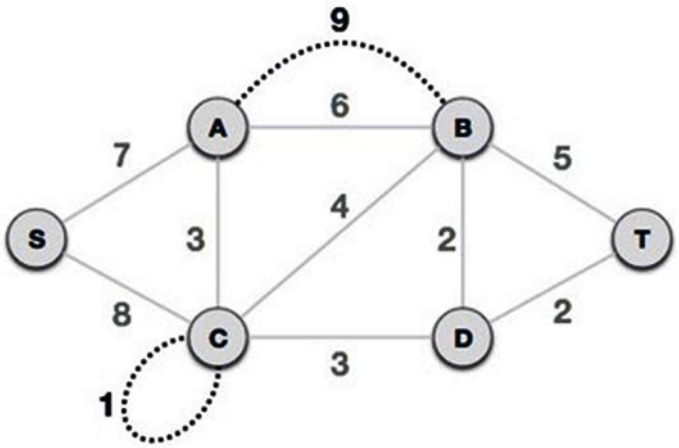
ii. Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

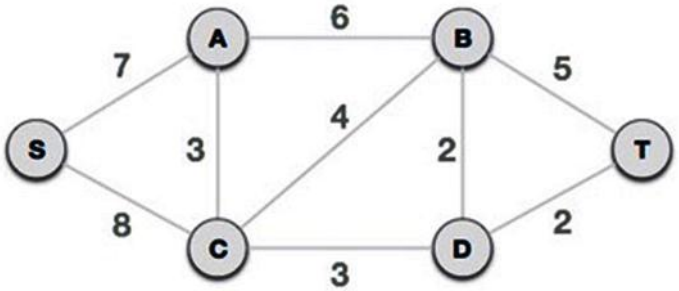
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example.



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.

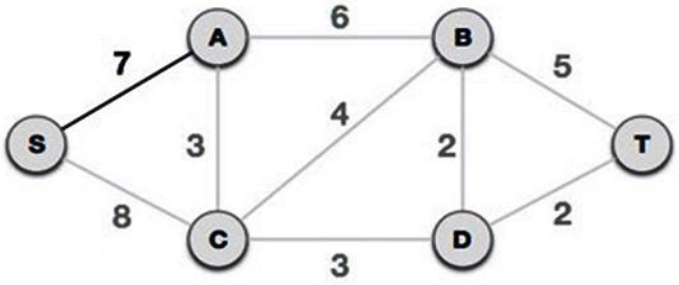


Step 2 - Choose any arbitrary node as root node

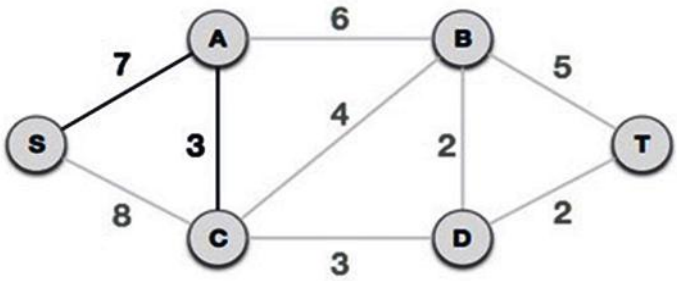
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

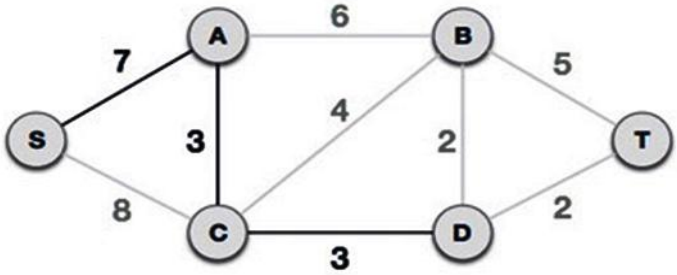
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



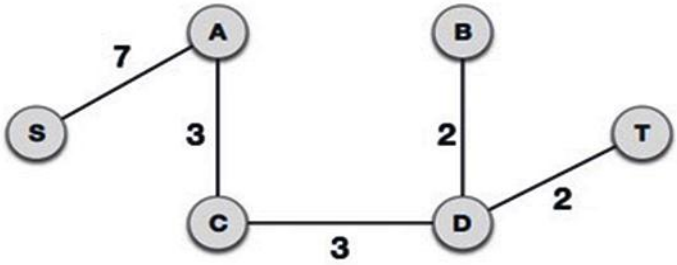
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

```
1   float Prim(int E[][SIZE], float cost[][SIZE],  int n, int t[][2])
11  {
12    int near[SIZE], j, k, L;
13    let (k,L) be an edge of minimum cost in E;
14    float mincost = cost[k][L];
15    t[1][1] = k; t[1][2] = L;
16    for (int i=1; i<=n; i++)  // Initialize near.
17      if (cost[i][L] < cost[i][k]) near[i] = L;
18      else near[i] = k;
19    near[k] = near[L] = 0;
20    for (i=2; i <= n-1; i++) { // Find n-2 additional
21                               // edges for t.
22      let j be an index such that near[j]!=0 and
23      cost[j][near[j]] is minimum;
24      t[i][1] = j; t[i][2] = near[j];
25      mincost = mincost + cost[j][near[j]];
26      near[j]=0;
27      for (k=1; k<=n; k++) // Update near[].
28        if ((near[k]!=0) &&
29            (cost[k][near[k]]>cost[k][j]))
30          near[k] = j;
31    }
32     return(mincost);
33  }
```

Time Complexity = O (n^2)

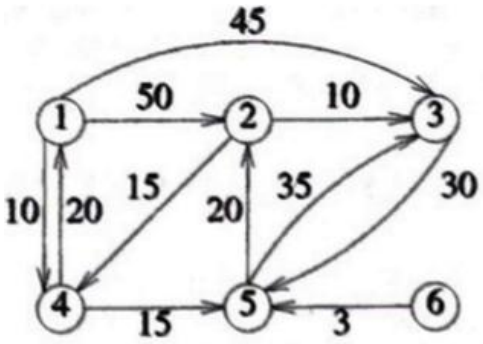
5. APPLICATION - SINGLE SOURCE SHORTEST PATH PROBLEM

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Example:



(a) Graph

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

(b) Shortest paths from 1

Algorithm:

```
ShortestPaths(int v, float cost[][SIZE], float dist[], int n)
{
    int u; bool S[SIZE];
    for (int i=1; i<= n; i++) { // Initialize S.
        S[i] = false; dist[i] = cost[v][i];
    }
    S[v]=true; dist[v]=0.0; // Put v in S.
    for (int num = 2; num < n; num++) {
     // Determine n-1 paths from v.
        choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] = true; // Put u in S.
        for (int w=1; w<=n; w++) //Update distances.
            if (( S[w]=false) && (dist[w] > dist[u] + cost[u][w]))
                dist[w] = dist[u] + cost[u][w];
        }
}
```

Time Complexity = O (n^2)

GREEDY APPROACH	DIVIDE AND CONQUER
1.Many decisions and sequences areguaranteed and all the overlapping subinstancesare considered.	1.Divide the given problem into many subproblems.Find the individual solutions andcombine them to get the solution for themain problem
2. Follows Bottom-up technique	2. Follows top down technique
3.Split the input at every possible pointsrather than at a particular point	3.Split the input only at specific points (midpoint), each problem is independent.
4. Sub problems are dependent on the main Problem	4. Sub problems are independent on the main Problem
5. Time taken by this approach is not that much efficient when compared with DAC.	5. Time taken by this approach efficient when compared with GA.
6.Space requirement is less when compared DAC approach.	6.Space requirement is very much high when compared GA approach.