DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Unit-3

Dr. G. S. Naveen Kumar,
Dean, Quality
Associate Professor, CSE

DAA Syllabus

UNIT- III

- ✓ Greedy method: General method,
- ✓ Applications-

i. Job sequencing with deadlines,
ii. Knapsack problem,
iii. Minimum cost spanning trees,
iv. Single source shortest path problem

Greedy method: General method

- The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving "optimization problems".
- An optimization problem is a problem when it requires either **minimum or maximum results**.
- This technique is basically used to determine the feasible solution that may or may not be optimal.
- 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 **feasible solution** is a subset that satisfies the given criteria.
- The **optimal solution** is the solution which is the best and the most favorable solution in the subset.
- In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Greedy method: General method

✓ What Is Greedy Algorithm?

Greedy algorithm is a problem-solving strategy that makes locally optimal decisions at each stage in the hopes of achieving a globally optimum solution.

We can implement a greedy solution only if the problem statement follows two properties mentioned below:

1. Greedy Choice Property: Choosing the best option at each phase can lead to a global (overall) optimal solution.
2. Optimal Substructure: If an optimal solution to the complete problem contains the optimal solutions to the sub problems, the problem has an optimal substructure.

Greedy method: General method

Steps for Creating a Greedy Algorithm

By following the steps given below, you will be able to formulate a greedy solution for the given problem statement:

- ✓ Step 1: In a given problem, find the best substructure or subproblem.
- ✓ Step 2: Determine what the solution will include (e.g., maximum profit, shortest path).
- ✓ Step 3: Create an iterative process for going over all subproblems and creating an optimum solution.

Greedy method: General method

Applications of Greedy Algorithm

- ✓ It is used in finding the shortest path.
- ✓ It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- ✓ It is used in a job sequencing with a deadline.
- ✓ This algorithm is also used to solve the fractional knapsack problem.

Greedy method: General method

Pseudo code of Greedy Algorithm

```
Algorithm Greedy (a, n)
{
  // a[1:n] contains the n inputs.
  Solution : = ϕ;  // initialize the solution to empty
 for i = 1 to n do
 {
    x: = Select(a);
   if Feasible(solution, x)  then
      solution: = Union(solution , x)
   }
    return solution;
 }
```

- ✓ Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.
- ✓ The function select selects an input from ‘a’, removes it and assigns its value to ‘x’.
- ✓ Feasible is a Boolean valued function, which determines if ‘x’ can be included into the solution vector.
- ✓ The function Union combines ‘x’ with solution and updates the objective function.

Components of Greedy Algorithm

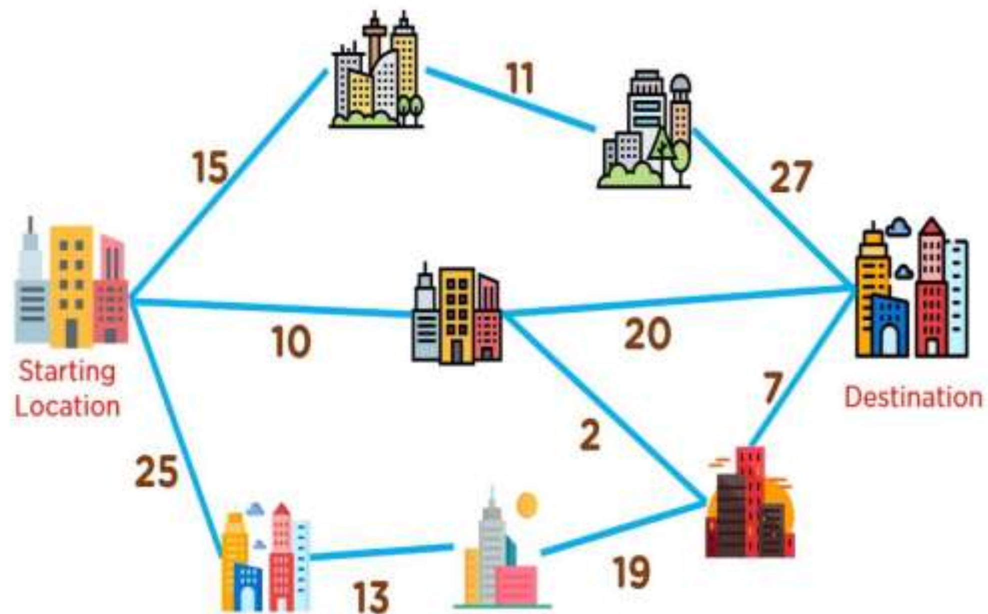
The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Greedy method: General method

Example of Greedy Algorithm

Problem Statement: Find the best route to reach the destination city from the given starting point using a greedy method.



Greedy method: General method

Example of Greedy Algorithm

Greedy Solution:

The steps to generate this solution are given below:

1. Start from the source vertex.
2. Pick one vertex at a time with a minimum edge weight (distance) from the source vertex.
3. Add the selected vertex to a tree structure if the connecting edge does not form a cycle.
4. Keep adding adjacent fringe vertices to the tree until you reach the destination vertex.
5. Paths will be picked up in order to reach the destination city.

Greedy method: Job sequencing with deadlines

Problem Statement:

- Given an array of jobs having a specific deadline and associated with a profit, provided the job is completed within the given deadline. The task is to maximize the profit by arranging the jobs in a schedule, such that only one job can be done at a time.
- Since, the task is to get the maximum profit by scheduling the **jobs**, the idea is to approach this problem greedily. An optimal solution is a feasible solution with **maximum** value.
- The complexity of this algorithm is $O(n^2)$

Greedy method: Job sequencing with deadlines

- When we are given a set of ‘n’ jobs. Associated with each Job i, deadline di > 0 and profit Pi > 0. For any job ‘i’ the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

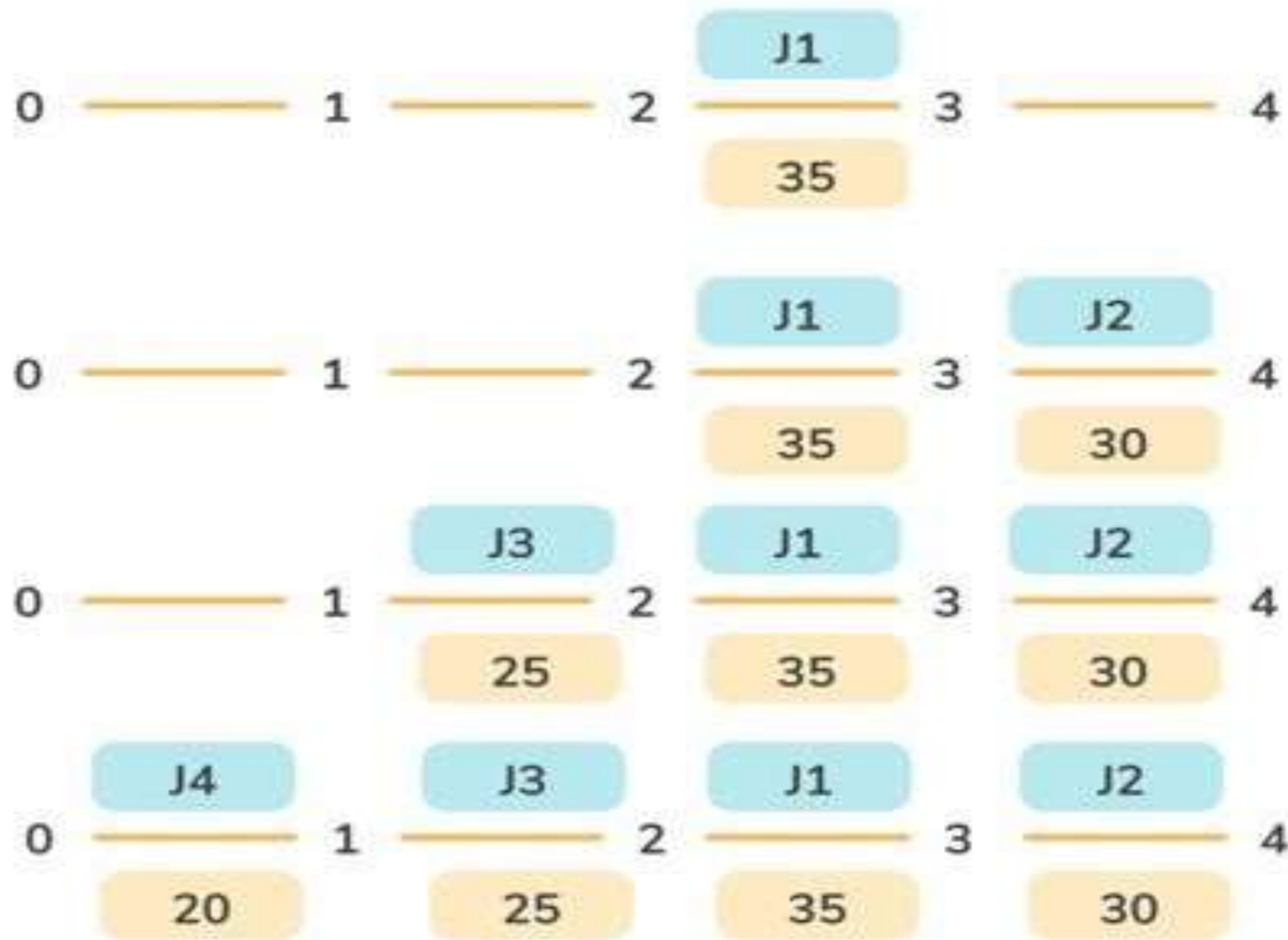
- Sort the jobs in ‘j’ ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \le r \le k$ are the jobs in ‘j’ and $d(j[1]) \le d(j[2]) \le \ldots \le d(j[k])$. To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that

$$d[J[r]] \le r, \quad 1 \le r \le k+1.$$

Examples: 1

n=7

Jobs	J1	J2	J3	J4	J5	J6	J7
Profits	35	30	25	20	15	12	5
Deadlines	3	4	4	2	3	1	2



Solution: J4 J3 J1 J2 **Total profit** : 20 + 25 + 35 + 30 = 110

Examples:2

Let n = 5, (p1,... ,p5) = (20,15,10,5,1) and (d1,...,d5)= (2,2,1,3,3). Using the feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
$\{1\}$	[1, 2]	2	assign to [0, 1]	20
$\{1, 2\}$	[0, 1], [1, 2]	3	cannot fit; reject	35
$\{1, 2\}$	[0, 1], [1, 2]	4	assign to [2, 3]	35
$\{1, 2, 4\}$	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Example:3

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.
Next, J_1 is selected as it gives more profit compared to J_4.
In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
The job J_5 is discarded as it cannot be executed within its deadline.
Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.
Total profit of this sequence is **100 + 60 + 20 = 180**.

Greedy method: Job sequencing with deadlines

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

```
Algorithm GreedyJob (d, J, n)
// J is a set of jobs that can be completed by their deadlines.
 {
J := {1};
for i := 2 to n do
{
if (all jobs in J U {i} can be completed by their dead lines)
then J := J U {i};
}
}
```

Knapsack problem

- The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Let us discuss the Knapsack problem in detail.
- Given a set of items, each with a weight and a profit, 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 profit is as large as possible.
- Problem Scenario

  - A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of ith item is w_i and its profit is 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

Knapsack problem

- Let us apply the greedy method to solve the knapsack problem. We are given ‘n’ objects and a knapsack. The object ‘i’ has a weight w_i and the knapsack has a capacity ‘m’. If a fraction xi, 0 < xi < 1 of object i is placed into the knapsack then a profit of pi xi is earned.
- The objective is to fill the knapsack that maximizes the total profit earned. Since the knapsack capacity is ‘m’, we require the total weight of all chosen objects to be at most ‘m’.
- The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i \quad x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

Knapsack problem

```
Algorithm GreedyKnapsack (m, n)
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so 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 i := 1 to n do x[i]  := 0.0                // initialize x
        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];
}
```

Knapsack problem

- Running time: The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time
- Example: Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15, 10)

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x_1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x_2 = 2/15

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

Knapsack problem

3. Considering the objects in the order of non-decreasing weights w_i.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / x_i. Select the object with the maximum p_i / x_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / x_i ratio, so $x_3 = ½$ and the profit earned is 7.5.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

This solution is the optimal solution.

Knapsack problem, Example 2

$n = 7$
$m = 15$

Knapsack Problem

Objects: O	1	2	3	4	5	6	7
profits: P	10	5	15	7	6	18	3
weights: w	2	3	5	7	1	4	1

$n=7$
$m=15$

Knapsack Problem

Objects: o	1	2	3	4	5	6	7
profits: P	10	5	15	7	6	18	3
weights: w	2	3	5	7	1	4	1
p/w :	5	1.3	3	1	6	4.5	3
x :	1	2/3	1	0	1	1	1

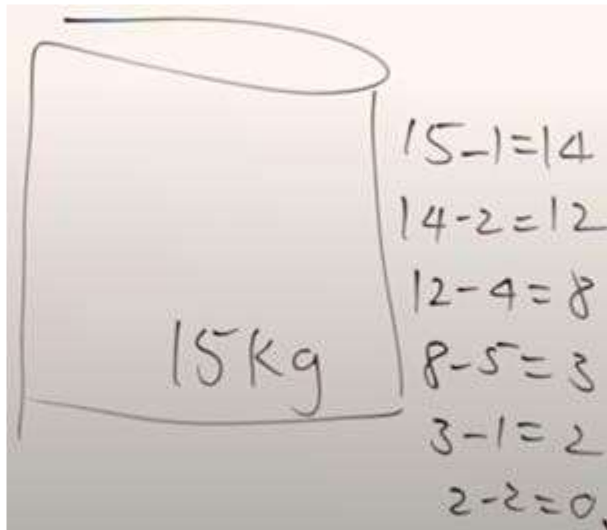
Constraint

$$\sum x_i w_i \le m$$

Objective

$$\max \sum x_i p_i$$

$0 \le x \le 1$



$$\sum x_i w_i = 1\times2 + \tfrac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$$
$$2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

$$\sum x_i p_i = 1\times10 + \tfrac{2}{3}\times5 + 1\times15 + 1\times6 + 1\times18 + 1\times3$$
$$= 10 + 2\times1.3 + 15 + 6 + 18 + 3 = 54.6$$

Minimum cost spanning trees:

- A spanning tree can be defined as the subgraph of an undirected connected graph.
- It includes all the vertices along with the least possible number of edges.
- If any vertex is missed, it is not a spanning tree.
- A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

➢ Properties:

✓ A spanning tree does not have any cycle.

✓ Any vertex can be reached from any other vertex.

✓ If there are **n** number of vertices, the spanning tree should have ***n*** **- 1** number of edges.

What is a spanning tree?

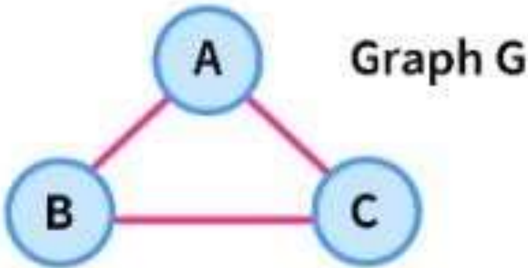
Definition :

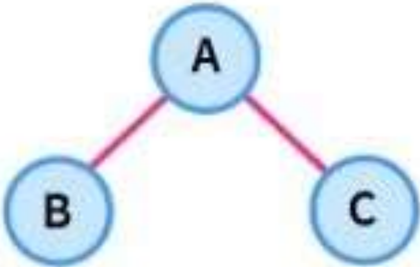
"A spanning tree is a sub-graph that connects all the vertices of the graph with the minimum possible number of edges. Spanning Trees of a given graph G can also be defined as a minimal set of edges that contains all the vertices of G. A spanning tree does not have any cycle and it can never be disconnected. A spanning tree can be weighted or unweighted."

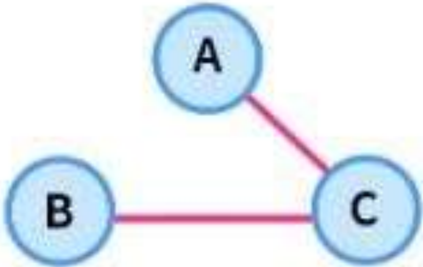
A complete undirected graph can have nn-2 number of spanning trees where n is the number of vertices in the graph.

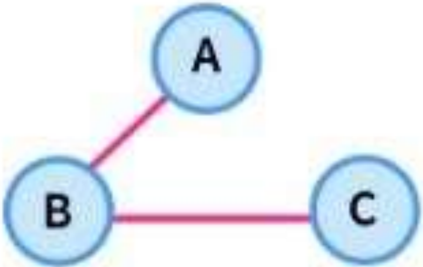
Suppose, if n = 5, the number of maximum possible spanning trees would be 55-2 = 125.

Example of Spanning Tree









Spanning Trees, subgraph of G

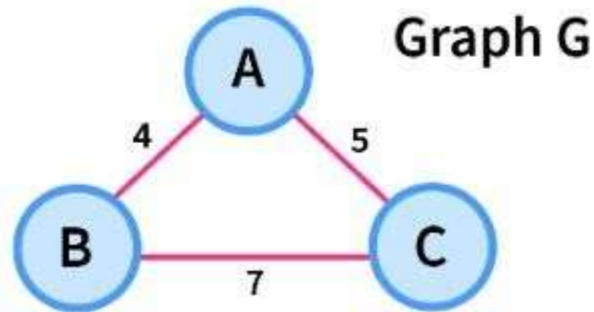
In this picture, we can see that the tree have no cycles and they are minimally connected so they are all the possible spanning trees
of 3 vertices for a given graph G.

What is Minimum Spanning Tree(MST)?

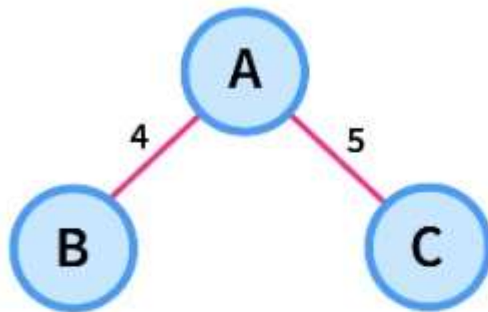
- A minimum spanning tree is the spanning tree that has a minimum cost among all the spanning trees. The cost of the spanning tree is the sum of the weights of all the edges in the tree. In real-life situations, this weight can be measured as distance, cost of transportation, manufacturing cost, traffic load, or any arbitrary value denoted by the edges. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Example

- Consider a weighted graph G with three vertices as shown in the picture below.

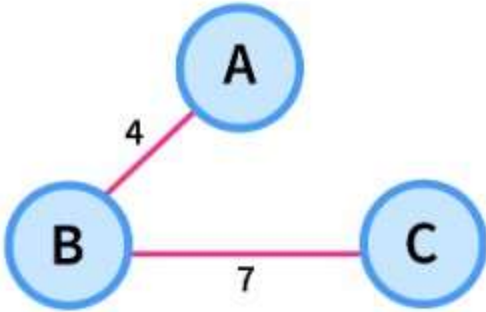


Now let us see some of the spanning trees which are possible with this graph G.

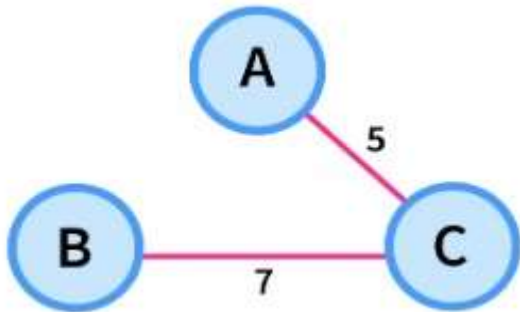


Total Cost=4+5=9

Example



Total Cost=4+7=11



Total Cost=7+5=12

From the above three cases, we can see that among all possible spanning trees **figure 1 has the minimum cost** , So it is the minimum spanning tree among the given spanning trees.

Application of Spanning Tree

Spanning tree has various uses in real life some of the applications are as follows.

- ✓ Spanning Trees are used in designing networks like computer networks, telephone networks, etc.
- ✓ Spanning Trees are used in civil network planning to build the networks.

Application of Minimum Spanning Tree

- ✓ Minimum Spanning Tree is used for designing telecommunication networks and water supply networks.
- ✓ For designing Local Area Networks.
- ✓ For solving the Travelling salesman problem

Properties of spanning-tree

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum n^{n-2} number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

Minimum cost spanning trees:

- To explain how to find a Minimum Spanning Tree, we will look at two algorithms:

1. Kruskals algorithm
2. Prims algorithm.

- ✓ Both algorithms differ in their methodology, but both eventually end up with the MST(Minimum Spanning Tree).
- ✓ Kruskal's algorithm uses edges, and Prim’s algorithm uses vertex connections in determining the MST.

Minimum cost spanning trees:

Kruskal's Algorithm:

- This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows:

- Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Minimum cost spanning trees:

Kruskal's Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

```
Algorithm Kruskal (E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify;
        for i := 1 to n do parent [i] := -1;
                                        // Each vertex is in a different set.
        i := 0; mincost := 0.0;
        while ((i < n -1) and (heap not empty)) do
        {
                Delete a minimum cost edge (u, v) from the heap and
                re-heapify 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;
}
```

Algorithm Steps for Kruskals algorithm:

1. Sort all the edges of the graph in the increasing order of their weight.
2. Pick the edge with the smallest weight.
3. Check if it forms a cycle with the spanning tree formed so far.

   -Include the current edge if it does not form any cycle.

   -Otherwise discard it.
4. Repeat step #3 until there V-1 edges in the spanning tree, where V is the total number of vertices in the graph.

Kruskal's Algorithm: Running time

- For the Adjacency Matrix representation of the graph.

  Time Complexity: $O(V^2)$

  Space Complexity: $O(V^2)$

  In Kruskal's Algorithm, the time required for traversing the matrix is $O(V^2)$ so the overall time complexity is $O(V^2)$. Also to represent the matrix we use a 2-Dimensional array. So, we will require $O(V^2)$ space where V is the number of vertices in graph G.

- For the Edge List representation of the graph.

  Time Complexity: $O(ElogE + ELogV)$

  Space Complexity: $O(V + E)$

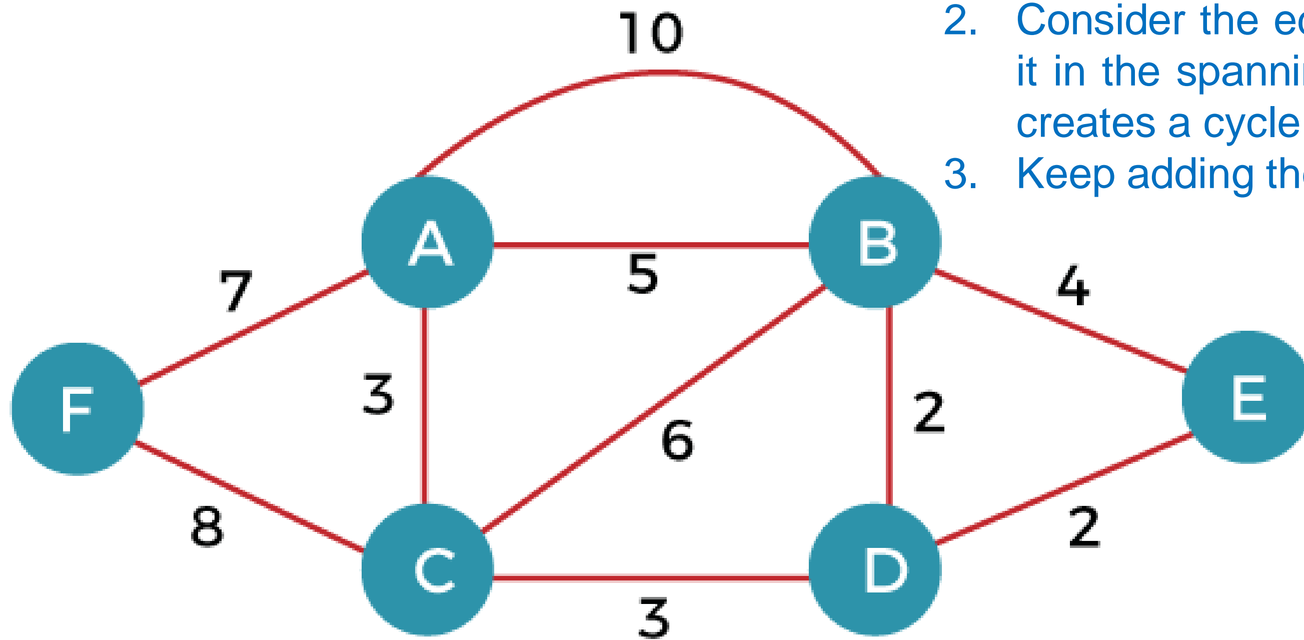
  Where E is the number of edges and V is the number of vertices in graph G.

Minimum cost spanning trees:

Kruskal's Algorithm:

Let's understand through an example:

Consider the below graph.



1. First, sort the edges in the ascending order of their edge weights.
2. Consider the edge which is having the lowest weight and add it in the spanning tree. If adding any edge in a spanning tree creates a cycle then reject that edge.
3. Keep adding the edges until we reach the end vertex.

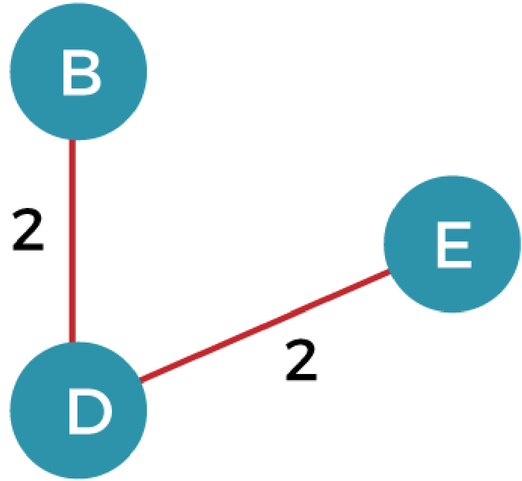
Minimum cost spanning trees:

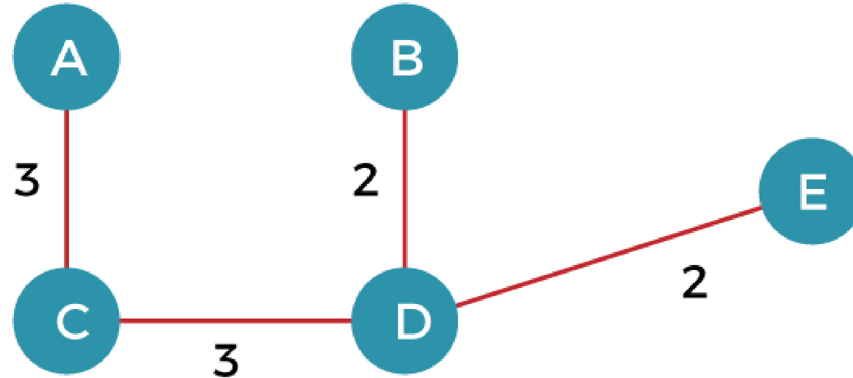
Kruskal's Algorithm:

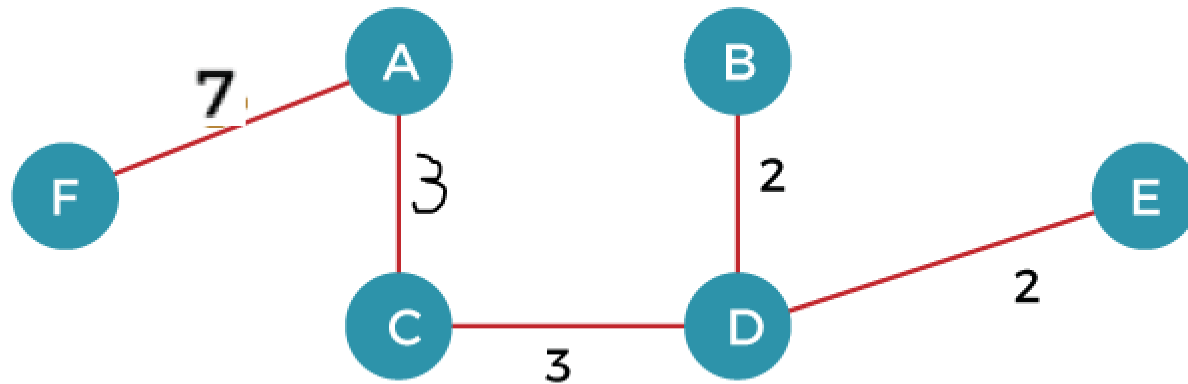
- Once we remove the parallel edge, we will arrange the edges according to the increasing order of their edge weights.
- As we can observe in the above graph, the minimum edge is 2.
- The next minimum edge weight is 3.
- The next edge with a minimum weight is 4.
- The next edge with a minimum weight is 6.
- The next edge with a minimum weight is 7.
- The next edge with a minimum weight is 8.
- Once the edge weights are written in the increasing order, the third step is to connect the vertices according to their weights.

Minimum cost spanning trees:

Kruskal's Algorithm:

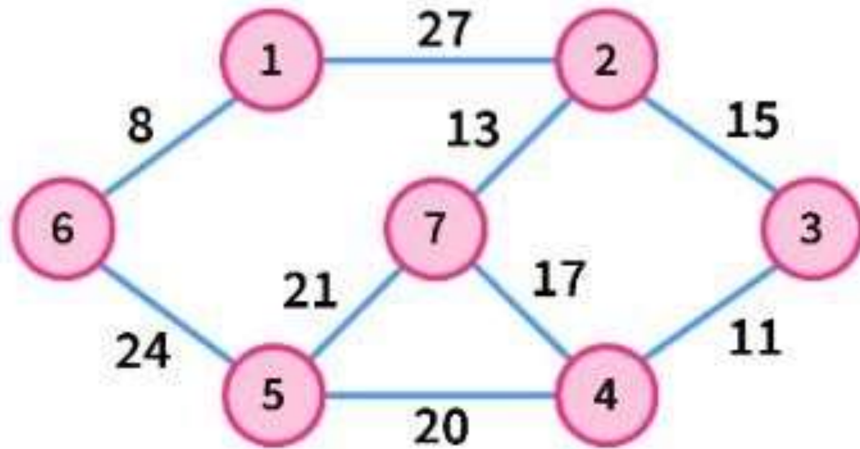




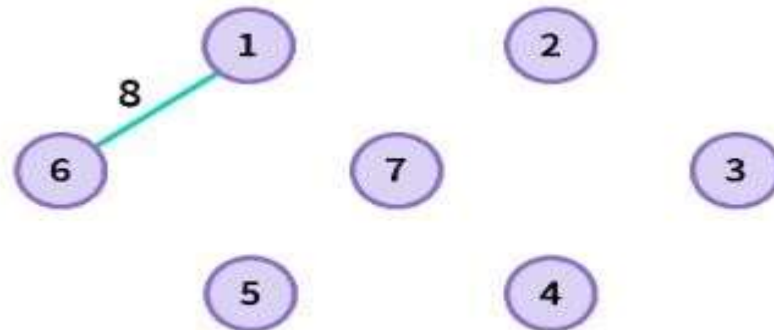


Minimum cost= 17

Let us understand the working of Kruskal's Algorithm with an example.

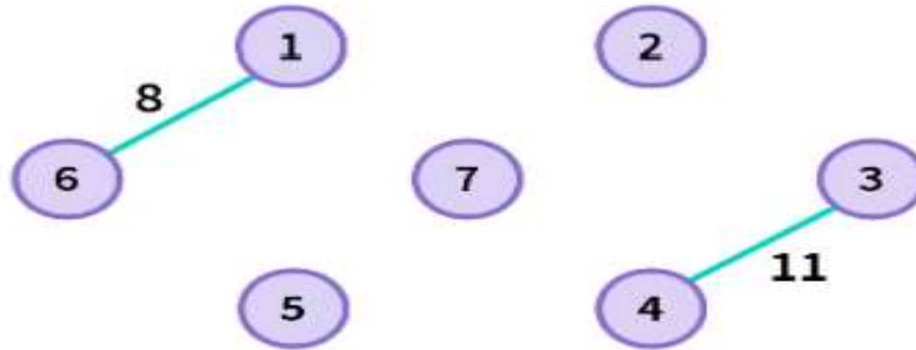


1. Choose the edge with the minimum weight.

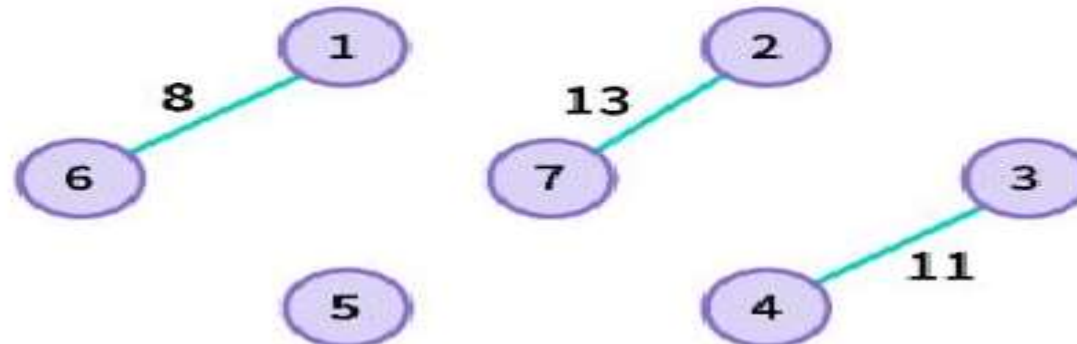


Let us understand the working of Kruskal's Algorithm with an example.

2. Choose the next shortest edge and add it. We can see from the below picture that the next shortest edge need not be connected with the former one.

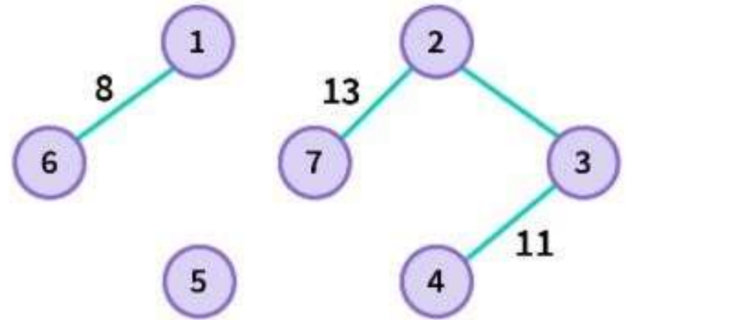


3. Choose the next shortest edge that doesn't create a cycle and add it.

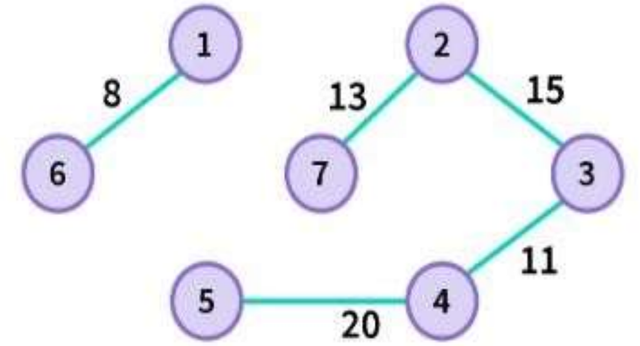


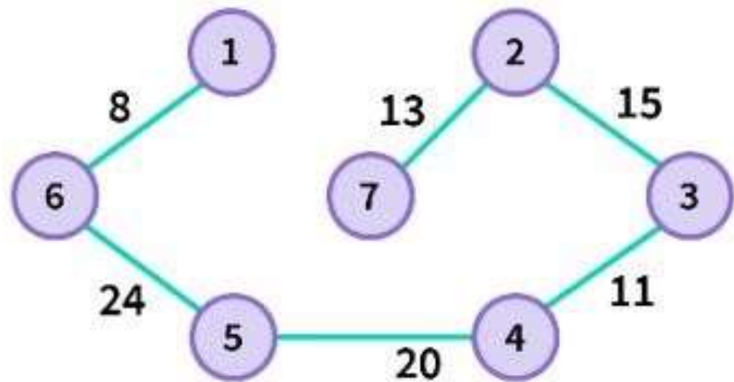
Let us understand the working of Kruskal's Algorithm with an example.

4. Choose the next shortest edge that doesn't create a cycle and add it. If there are more than one such edges, choose anyone from them.



5. Choose the next shortest edge that doesn't create a cycle and add it.





Minimum cost= 91

Prim's Algorithm:

Minimum cost spanning trees:

- ✓ A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.
- ✓ Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST.
- ✓ In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.
- ✓ This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Minimum cost spanning trees:

Prim's Algorithm:

Prim's algorithm is an example of a greedy 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 number or ∞ if no edge (i, j) exists.
 // A minimum spanning tree is computed and stored as a set of
 // edges 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 of minimum cost in E;
        mincost := 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 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                          // Update near[].
                    if ((near [k] ≠ 0) and (cost [k, near [k]] > cost [k, j]))
                          then near [k] := j;
        }
        return mincost;
 }
```

Minimum cost spanning trees:

Prim's Algorithm:

Prim's algorithm is an example of a greedy algorithm.

Algorithm Steps:

1. Select a starting vertex.
2. Select an edge e connecting the tree vertex and fringe vertex that has minimum weight. Fringe vertices are the vertices adjacent to visited vertices but not yet visited.
3. Add the selected edge and the vertex to the minimum spanning tree T
4. Repeat step #2 and step #3 until the vertices adjacent to the visited vertex are unvisited.

Minimum cost spanning trees:

Prim’s Algorithm:

- For the Adjacency Matrix representation of the graph.

  Time Complexity: $O(V^2)$

  Space Complexity: $O(V^2)$

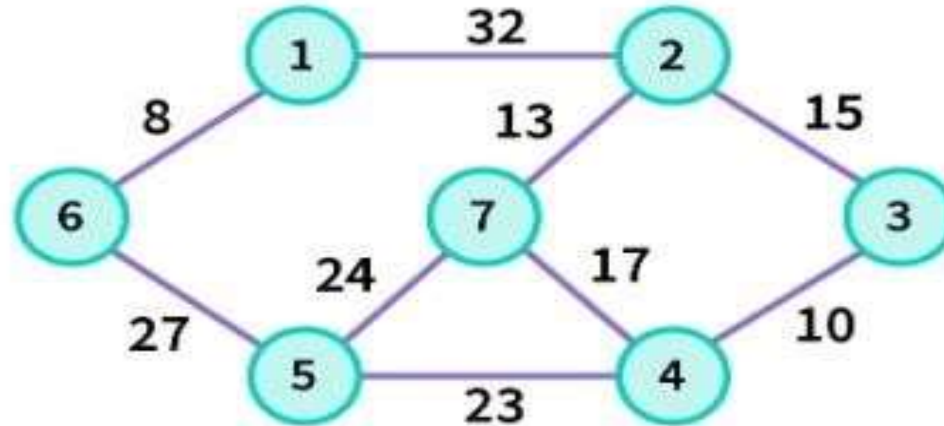
  In Prim's Algorithm, the time required for traversing the matrix is $O(V^2)$ so the overall time complexity is $O(V^2)$. Also to represent the matrix we use a 2-Dimensional array. So, we will require $O(V^2)$ space where V is the number of vertices in graph G.

Prim’s Algorithm:

Example:

- Let us understand the working of Prim's Algorithm with an example. Consider a weighted graph G having seven vertices in the picture below.

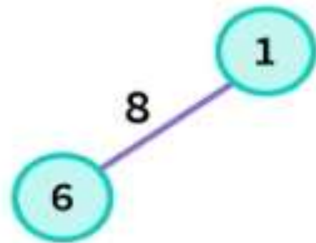
es:



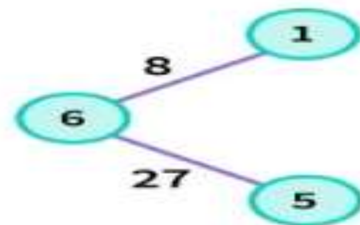
Prim's Algorithm:Examp

Now to find the minimum spanning tree using Prim's Algorithm, follow the steps below.

1. Choose any arbitrary vertex , here we are starting from vertex 1 in the given graph G.



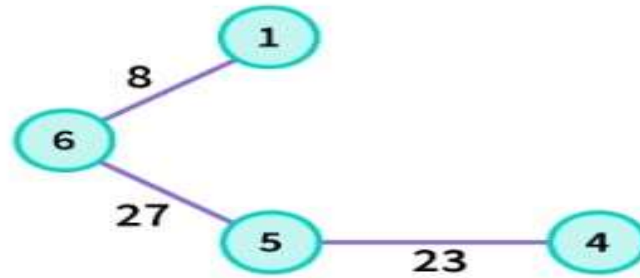
2. Choose the shortest edge from this vertex and add it to the spanning tree so formed.



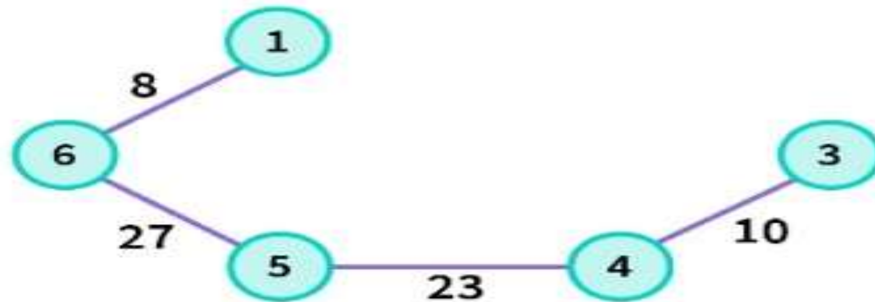
Prim's Algorithm:

Example:

3. Choose the nearest vertex which is not yet included in the solution. In each iteration, we are considering all the respective fringe vertices for a particular vertex.



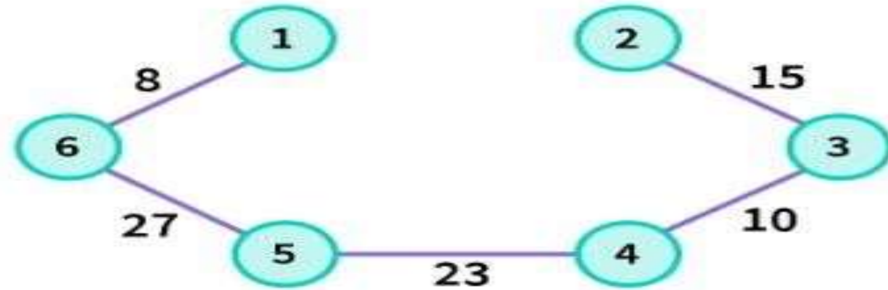
4. Choose the nearest vertex which is not yet included in the solution and does not form any cycle.



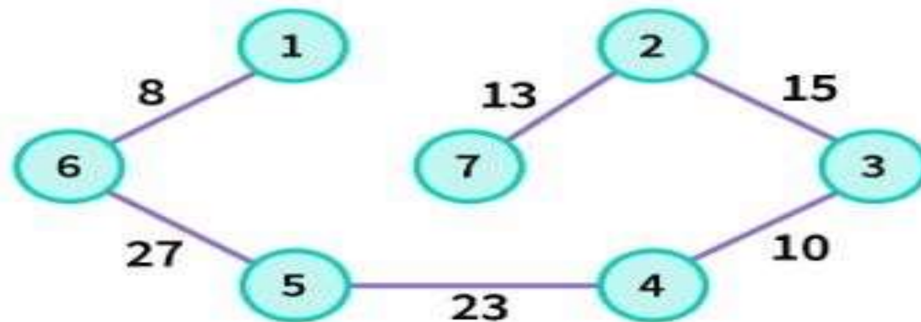
Prim's Algorithm:

Example:

5. Choose the nearest vertex which is not yet included in the solution and does not form any cycle.



6. Repeat the same process until our minimum spanning tree does not have V-1 vertices, where v is the total number of vertices in the in original graph G.



Minimum cost= 96

Single source shortest path problem:

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

- In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.
- In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.
- Dijkstra’s algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra’s algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one.

Single source shortest path problem:

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

- Dijkstra's Algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956. It is a single source shortest paths algorithm. It means that it finds the shortest paths from a single source vertex to all other vertices in a graph. It is a greedy algorithm and works for both directed and undirected, positively weighted graphs (a graph is called positively weighted if all of its edges have only positive weights).
- Dijkstra's Algorithm requires a graph and source vertex to work.The algorithm is purely based on greedy approach and thus finds the locally optimal choice(local minima in this case) at each step of the algorithm.

Single source shortest path problem:

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

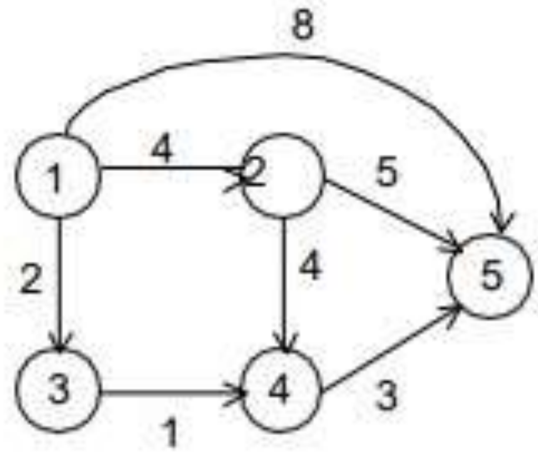
- In this algorithm each vertex will have two properties defined for it-
- **Visited property**:-
  - This property represents whether the vertex has been visited or not.
  - We are using this property so that we don't revisit a vertex.
  - A vertex is marked visited only after the shortest path to it has been found.
- **Path property**:-
  - This property stores the value of the current minimum path to the vertex. Current minimum path means the shortest way in which we have reached this vertex till now.
  - This property is updated whenever any neighbour of the vertex is visited.
  - The path property is important as it will store the final answer for each vertex.

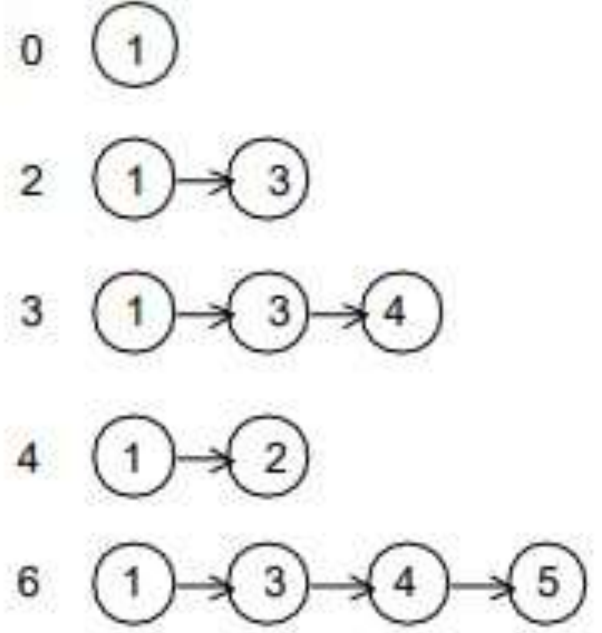
Single source shortest path problem:

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

Dijkstra’s algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.





Single source shortest path problem:

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

Algorithm:

```
Algorithm Shortest-Paths (v, cost, dist, n)
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
        for i :=1 to n do
        {
                S  [i] := false;                              // Initialize S.
                dist [i] :=cost [v, i];
        }
        S[v] := true; dist[v]  := 0.0;                        // Put v in S.
        for num := 2 to n – 1 do
        {
                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 is S.
                for (each w adjacent to u with S [w] = false) do
                        if (dist [w] > (dist [u] + cost [u,  w]) then      // Update distances
                                dist [w] := dist [u] + cost [u, w];
        }
}
```

Single source shortest path problem:

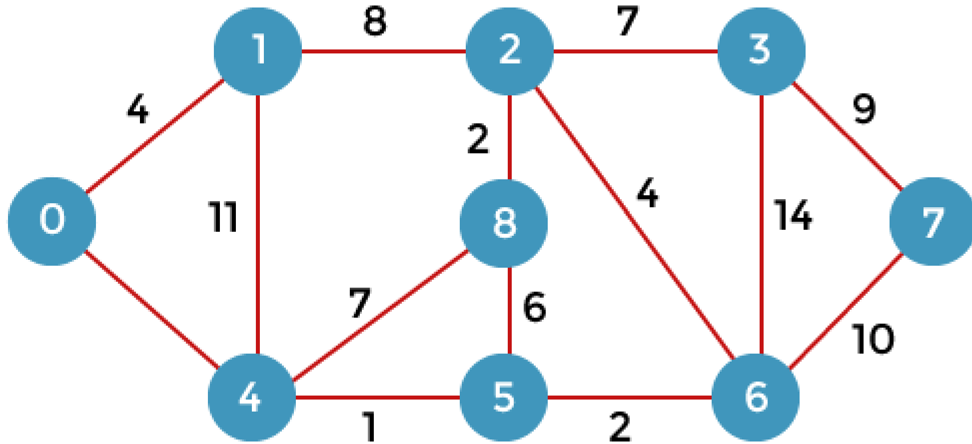
The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHM

Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements — A
- at most $m = |E|$ times decrease the value of an item — mB
- 'n' times select the smallest value — nC
- For array A = O (n); B = O (1); C = O (n) which gives O (n^2) total.
- For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

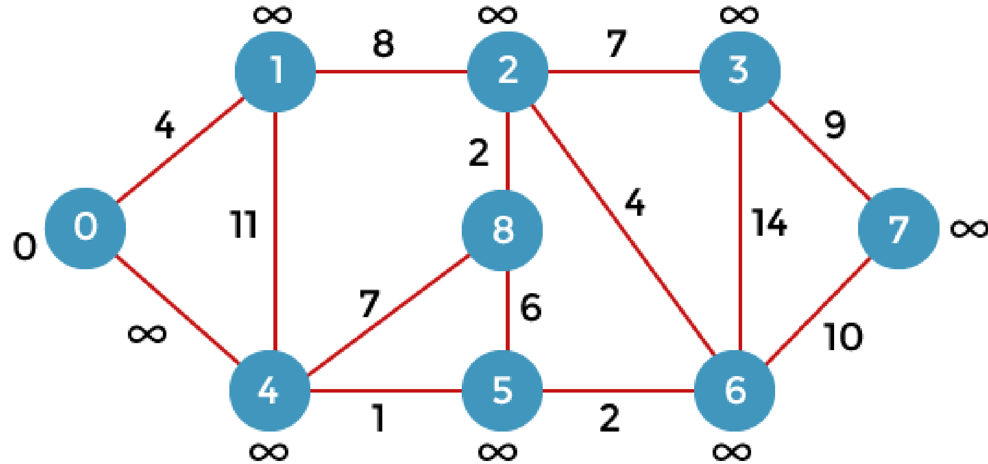
Single source shortest path problem: Example



First, we have to consider any vertex as a source vertex. Suppose we consider vertex 0 as a source vertex.

Here we assume that 0 as a source vertex, and distance to all the other vertices is infinity. Initially, we do not know the distances. First, we will find out the vertices which are directly connected to the vertex 0.

Single source shortest path problem: Example



The formula for calculating the distance between the vertices:

$\{\text{if}(\ d(u) + c(u, v) < d(v))$

$d(v) = d(u) + c(u, v)\ \}$

Let's assume that the vertex 0 is represented by 'x' and the vertex 1 is represented by 'y'. The distance between the vertices can be calculated by using the below formula:

$d(x, y) = d(x) + c(x, y) < d(y)$

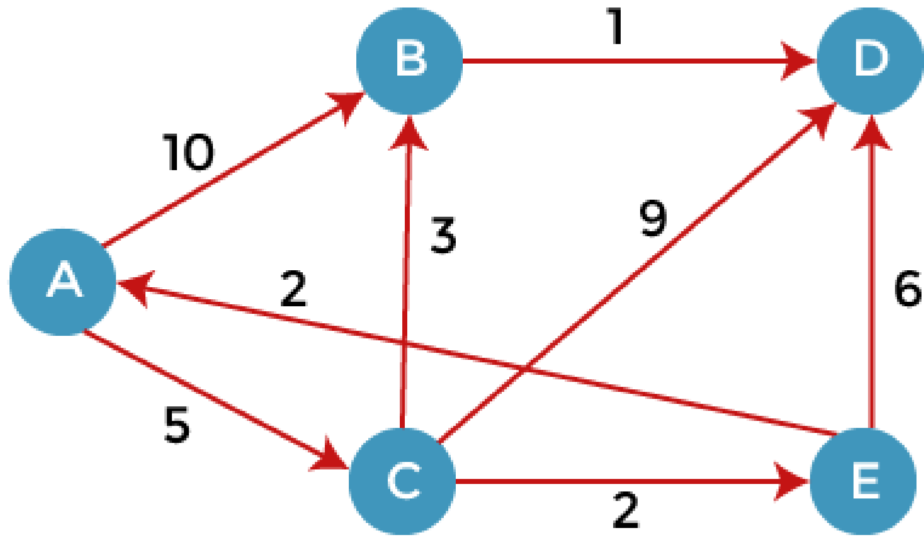
$= (0 + 4) < \infty$

$= 4 < \infty$

Since $4<\infty$ so we will update d(v) from ∞ to 4.

Continue to update distances if the condition is satisfied.

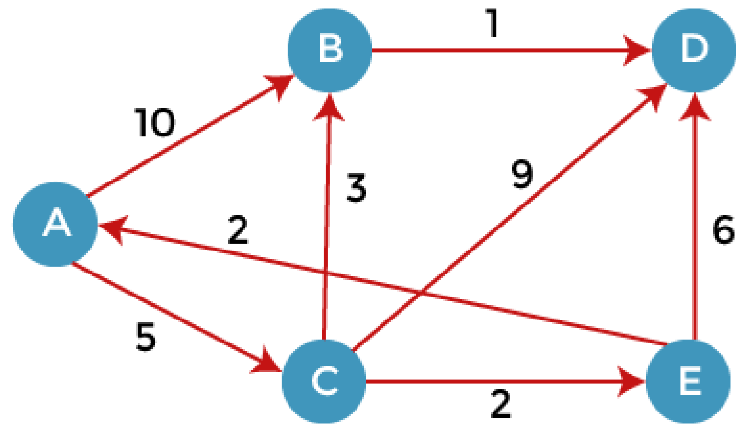
Example for directed graph.



Here, we consider A as a source vertex. A vertex is a source vertex so entry is filled with 0 while other vertices filled with ∞. The distance from source vertex to source vertex is 0, and the distance from the source vertex to other vertices is ∞.

We will solve this problem using the below table:

Example for directed graph.



We will solve this problem using the below table:

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

Thank you