Module 5

Greedy Method

Greedy Method

- There is a problem with ‘n’ inputs
- ***Feasible solution*** : Obtain a subset that satisfies some constraints
- ***Optimal Solution*** : Find a feasible solution that either maximizes or minimizes a given objective function
- This method finds the feasible solutions as per the constraints and **finds the optimal solution** from it. The **optimum value can be a maximum or minimum value** depending upon the problem constraints

Greedy Method

- For Eg : We want to find the tallest person from a group. We have their heights. In this method first we will set a bench mark (Eg: height more than 6 feet). Then remove all entries less than 6 feets. It is the feasible solution list. Then find the optimal one, that is the maximum value.
- Similarly if it is for finding a fastest athlete of a season we will set time like 100 meters in less than 10 seconds. Makes feasible solution and go for athlete with minimum value, right?
- I think you got the idea.

Greedy Method

- At each stage, a decision is made regarding whether a particular is in an Optimal Solution.
- Consider the inputs in an order determined by some selection procedure.
- If the inclusion of the next input into a partially constructed optimal solution will result in an infeasible solution, then that input is not added in the partial solution.
- Otherwise it is added

Greedy Method

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

Knapsack Problem

- It is an optimization problem
- Given a set of items, each with a weight and a value (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 (Capacity of Knapsack or a bag) and the total value (profit) is as large as possible
- Two types :
  - Fractional Knapsack problem
  - 0/1 Knapsack Problem

Knapsack Problem

- In 0/1 knapsack problem we can either add an item fully or not add. Ie. Either take it completely or not take.
- Where as in fractional knapsack problem we can take a fraction of an item. That is , if an item cannot be added completely, a fraction of it can be added.
- **In greedy method, we solve the fractional knapsack problem.**
- 0/1 Knapsack problem will be discussed later (using Backtracking method)

Knapsack Problem (Fractional)

- There are ‘n’ objects and a Knapsack or bag with capacity ‘m’. Object ‘i’ has a wight ‘w_i’ and a profit ‘p_i’
- If a fraction xi , **$0 \leq x_i \leq 1$**, of object ‘i’ can be placed into the knapsack, a profit $p_i x_i$ is earned.
- The objective is to fill the Knapsack that maximizes the total Profit.
- Since the capacity of Knapsack is ‘m’, the total weight of all chosen objects must be at most ‘m’

Knapsack Problem (Fractional)

- The problem can be stated as

$$\textbf{Maximize} \sum_{i=1}^{n} x_i \cdot pi$$

$$\textbf{Subject to the constraint} \sum_{i=1}^{n} x_i \cdot wi \leqslant m$$

Where $\mathbf{0 \leq x_i \leq 1}$

- 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

$$\sum_{i=1}^{n} x_i \cdot wi = m$$

Knapsack Problem (Fractional)

- In order to solve the problem we must sort those objects according to the value of $\mathbf{p_i/w_i}$ such that $\mathbf{p_i/w_i \geq p_{i+1}/w_{i+1}}$(Decreasing order of the ratio)
- Lets check the algorithm.
- Two arrays **p[1:n]** and **w[1:n]** contains profit and weights of respectively
- Here **m** is the size of the Knapsack and there are **n** objects sorted such that $\mathbf{p_i/w_i \geq p_{i+1}/w_{i+1}}$
- **x[1:n]** is the solution vector
- The final profit will be obtained by adding all $\boldsymbol{x_i p_i}$

Knapsack Problem (Fractional)

- **Algorithm Greedy_Knapsack(m,n)**

```
{
    for i=1 to n do
        x[i]=0              //  Initialize x[]
    U = m                   //  U is the Remaining capacity, initially m.
    for i=1 to n do
    {
        If (w[i]>U) then
           break
        x[i] = 1
        U = U-w[i]
    }
    if(i<=n)  then  x[i] = U/w[i]        // Adding the fraction of last object
}
```

Knapsack Problem (Fractional) – Example1

- Let the capacity of Knapsack, m=60, n=4

Item	1	2	3	4
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (pi/wi)	7	10	6	5

- After Sorting

Item	1	2	3	4
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (pi/wi)	10	7	6	5

Knapsack Problem (Fractional) – Example 1

- Initially available space

  U = 60 (U = m)
- Solution Array x = [0,0,0,0]

1) W1= 10 $10 \leq 60$
   - So add 1st item **w1 =10**
   - X=[1,0,0,0]
   - U=60-10 = **50**
2) W2= 40 $40 \leq 50$
   - So add 2nd item W2 = 40
   - X=[1,1,0,0]
   - U = 50 – 40 = **10**
3) W3= 20 20 > 10
   - **So break….**

- **But i<=n**
  - X3 = U/w3
  - ie. X3 = 10/20 = 0.5
  - X=[1,1,0.5,0]
- Let's stop here.
- **Total Profit**

= 1x100+1x280+0.5x120+0x24

= **440**

- **Total Weight**

= 1x10+1x40+0.5x20+0X 24= **60**

Knapsack Problem (Fractional) – Example 2

- Let the capacity of Knapsack, m=20, n=3

Item	1	2	3
Profit	25	24	15
Weight	18	15	10
Ratio (pi/wi)	1.38	1.6	1.5

- After Sorting

Item	1	2	3
Profit	24	15	25
Weight	15	10	18
Ratio (pi/wi)	1.6	1.5	1.38

Knapsack Problem (Fractional) – Example 2

- Initially available space
  U = 20 (U = m)
- Solution Array x = [0,0,0]

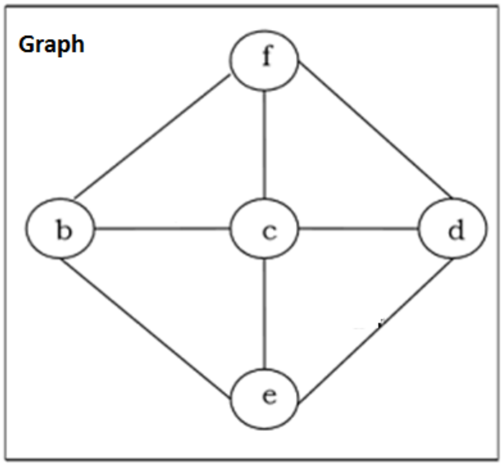
1) W1= 15 15 ≤ 20
   - So add 1st item **w1 =15**
   - X=[1,0,0]
   - U=20-15 = **5**
2) W2= 10 10 **> 5**
   - **So break….**

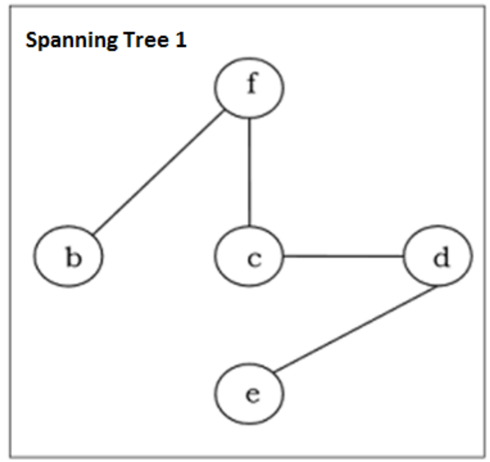
- **But i<=n**
  - X2 = U/w2
  - ie. X2 = 5/10 = 0.5
  - X=[1,0.5,0]
- Let's stop here.
- **Total Profit**
  = 1x24+0.5x15+0x25
  = **31.5**
- **Total Weight**
  = 1x15+0.5x10+0x18 = **20**

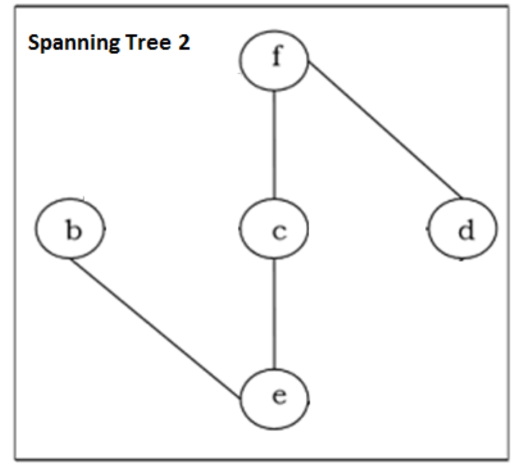
Spanning Trees

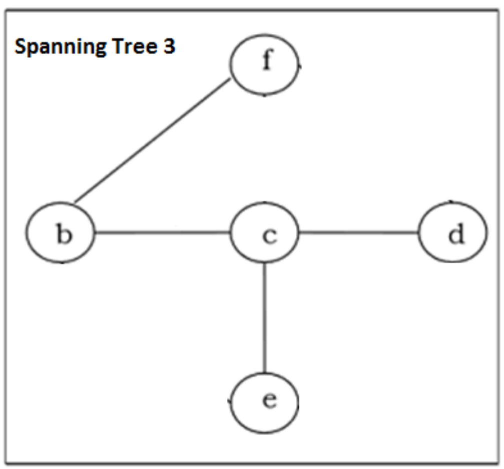
- Let **G=(V,E)** be an **undirected connected graph**. A subgraph **T=(V,E')** of **G** is a spanning tree of **G**, iff. **T** is a Tree
- A **spanning tree** is a subset of Graph **G**, which has all the vertices covered with minimum possible number of edges.
- A **spanning tree** does not have cycles
- Every connected and undirected Graph has at least one spanning tree
- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).

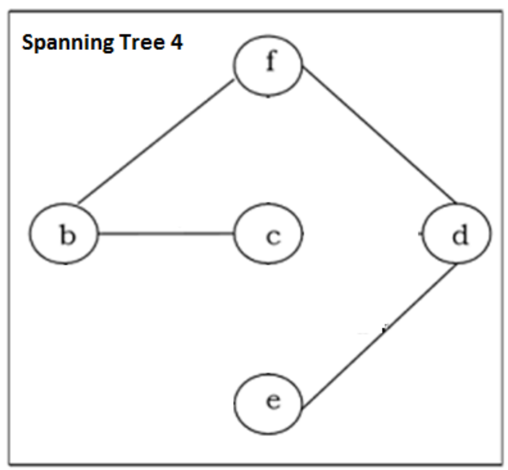
Spanning Trees

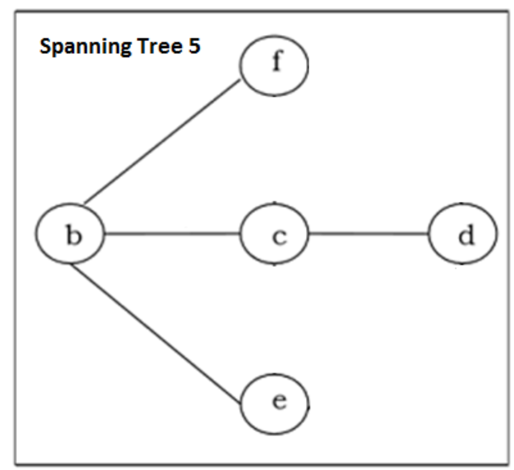












Spanning Trees

- 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**.

Minimum Cost Spanning Tree

- A *minimum spanning tree (MST)* or minimum weight spanning tree for a **weighted, connected and undirected graph** is a spanning tree with **weight less than or equal to the weight of every other spanning tree.**
    - 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.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Cost Spanning Tree

- Minimum Spanning-Tree Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm
- In Greedy method to obtain a minimum cost spanning tree, the tree is built edge by edge.
- The next edge to include is chosen according to some optimization criteria.

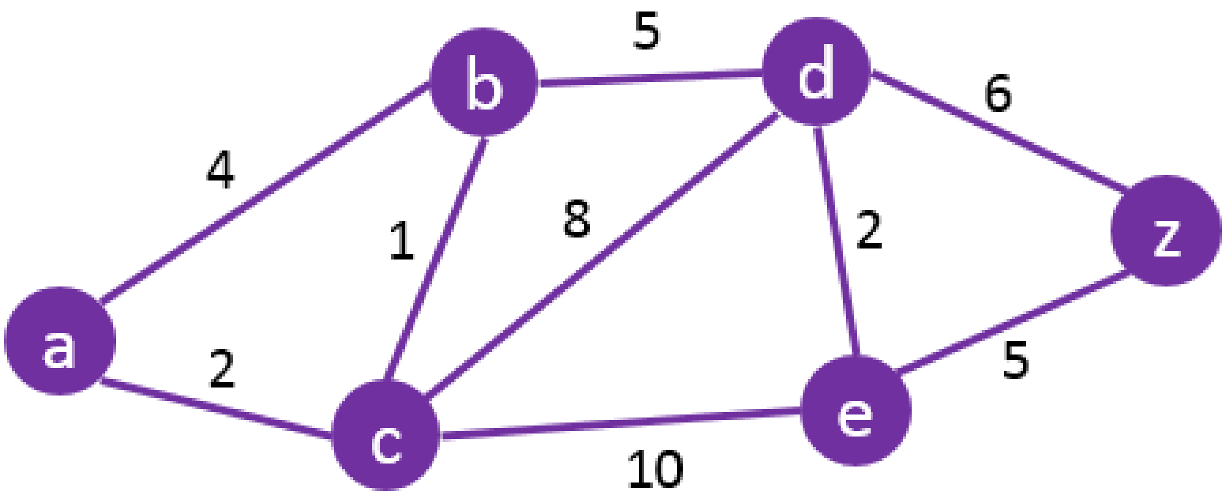
Kruskals Algorithm

- We have a weighted graph and a spanning tree with minimum total weight must be created.
- Here, the edges of the graph are considered in non-decreasing order (Ascending order) of the cost.
    - **First list out all edges in order of their cost**
- Select each edge such that it will not form a cycle. If so, add the edge into the group spanning tree edges.
- If there are ‘n’ vertices, there must be ‘n-1’ edges. So if we the no. of edges reaches ‘n-1’ we can stop.
- **Note : while creating the spanning tree there may be a set of trees (forest). Not necessary to have a single tree. Finally we will get a single spanning tree.**

Kruskals Algorithm

1) Sort all the edges in non-decreasing order of their weight.
2) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
   - If cycle is not formed, include this edge.
   - Else, discard it.
3) Repeat step #2 until there are **V-1 edges** in the spanning tree *(V is the no. of vertices)*.

Kruskals Algorithm - Example



<u>Edges (in the order)</u>

bc, ac, de, ab, bd, ez, dz, cd, ce

Kruskals Algorithm - Example

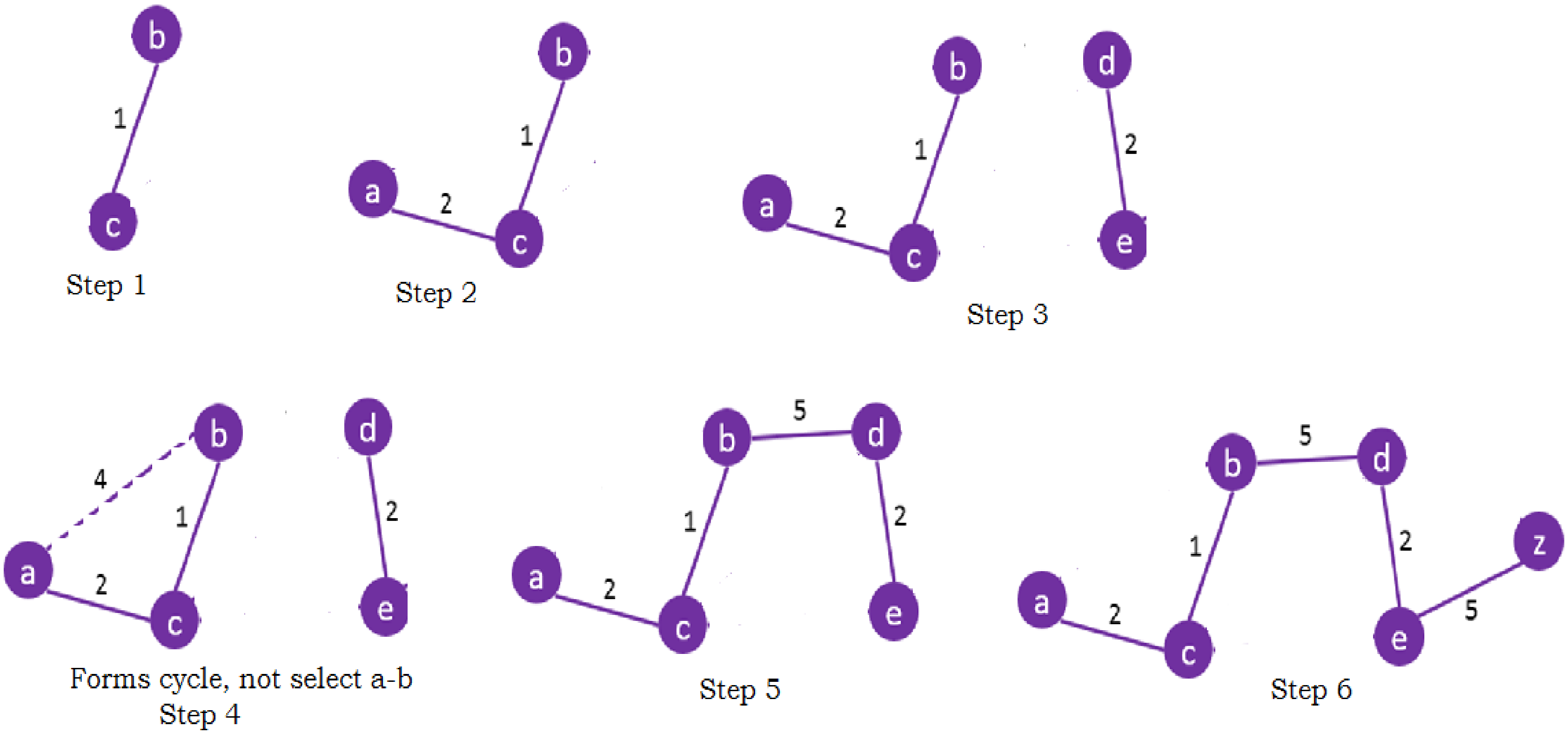
- **<u>Edges (in the order)</u>**
- bc, ac, de, ab, bd, ez, dz, cd, ce

1. Take b-c
2. Take a-c
3. Take d-e
4. Consider a-b

   forms cycle -cannot be added
5. Take b-d
6. Take e-z

   There are only 6 vertices and Now we got 5 edges. we can stop. Minimum cost Spanning tree is created.

Kruskals Algorithm - Example



Minimum Cost = 15

Prims Algorithm

- Here we will start from an arbitrary vertex. It will added to the set of spanning tree vertices. It grows until it includes all the vertices of the graph.
- Let **Vt** be the selected spanning tree vertices so far. Find the minimum edge (u,v) such that ‘***u’ is in Vt*** and ‘**v’ is not in Vt** (v is in set of Vertices of Graph, set V). Then select the adge and add ‘v’ to Vt.
- That is Find the shortest edge from already created spanning tree, which does not create a cycle, and add it to the set of spanning tree vertices

Prims Algorithm

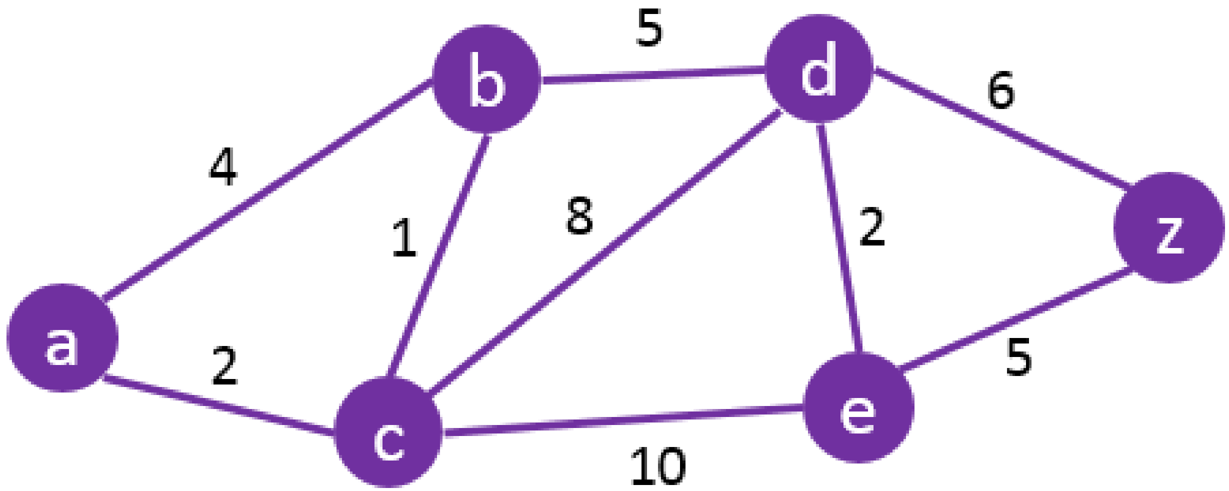
- It will be continued until we include all the vertices of Graph in to Vt.
- **Note: Unlike Kruskals algorithm, here in each stage we will get a single tree**
- Algorithm is given below.
- There is a graph G(V,E) where V is the set of Vertices and E is the set of Edges
- Find a minimum spanning tree G'(Vt,Et) where Vt is the set vertices and Et is the set of Edges in Spanning Tree.
- Finally Vt will become V (Include all vertices), Et will be subset of E (not necessary to have all edges of E)

Prims Algorithm

- Let there be a weighted connected graph G=(V,E)
- Create a Min spanning Tree T=(Vt,Et)

1) Vt = {V_0} // we start from V_{0}, So V_t will be V_o only
2) Et = ϕ // there will not be any edges Et will be empty
3) Mincost = 0
4) For i=1 to |V|-1 do // |V| is no. of vertices
5) {
   - Find a minimum weight edge **e=(u,v)** among all edges, such that **u is in Vt** and **v is in (V-Vt)**
   - Vt = Vt U {v}
   - Et = Et U {e}
   - Mincost = Mincost + Cost (u,v)
6) }
7) Return Et

Prims Algorithm - Example



Prims Algorithm - Example

Start from vertex 'a'

Now Vt = {a}   Et={ }

1. Consider Edges from 'a' and take the minimum

So, a-c is taken   Vt= {a,c}   Et={ac}

2. Consider edges from 'a' and 'c' & find the minimum
   a-b,c-b,c-d,c-e

   So, c-b is taken   Vt={a,c,b}   Et={ac,cb}

3. Consider edges from the nodes in set Vt and Find the minimum (a-b, b-d, c-d, c-e)

Actually a-b is minimum. Already a and b are in Vt, means it forms cycle. So cannot include

So, take b-d   Vt = {a,c,b,d}   Et={ac,cb,bd}

Prims Algorithm - Example

4. Consider edges from the nodes in set Vt and Find the minimum (c-d,c-e,d-e,d-z)

   Here, d-e is taken Vt={a,c,b,d,e} Et={ac,cb,bd,d-e}

5. Consider the next minimum edge from vertices in set Vt

   So e-z is taken Vt = {a,c,b,d,e,z} Et={ac,cb,bd,d-e,e-z}

Now we included all the 6 vertces and 5 edges. So the minimum spanning tree is created

Prims Algorithm - Example

Fig (1)

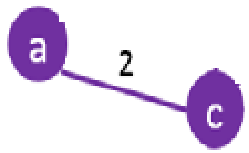


Fig (2)

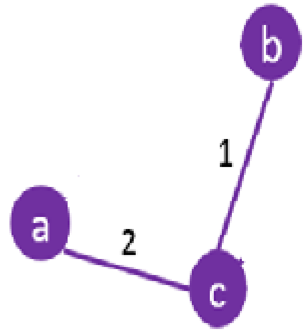


Fig (3)

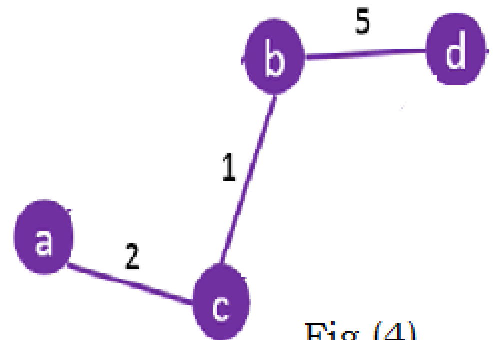


Fig (4)

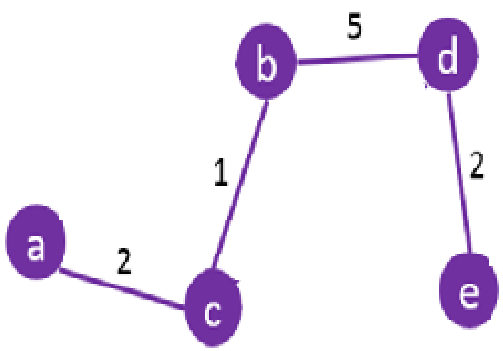


Fig (5)

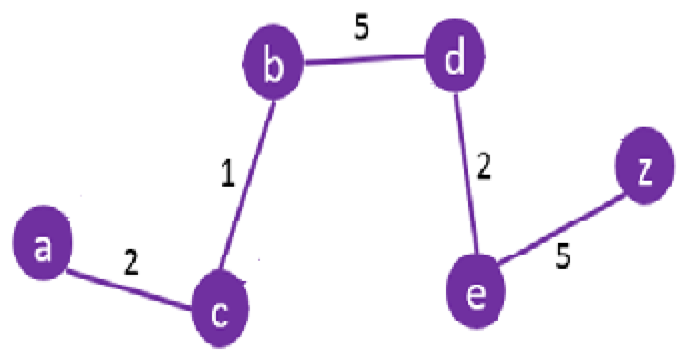


Fig (6)

Minimum Cost = 15