UNIT – V

Greedy Method

Greedy Method: The general method – Optimal Storage on Tapes – Knapsack Problem – Job Sequencing with deadlines – Optimal Merge Patterns – Minimum Spanning Trees – Single Source Shortest Paths

The General Method

What is a 'Greedy algorithm'?

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.

The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given ‗n‘inputs are required us to obtain a subset that satisfies some constraints that is the feasible solution. A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

Greedy choice property

We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

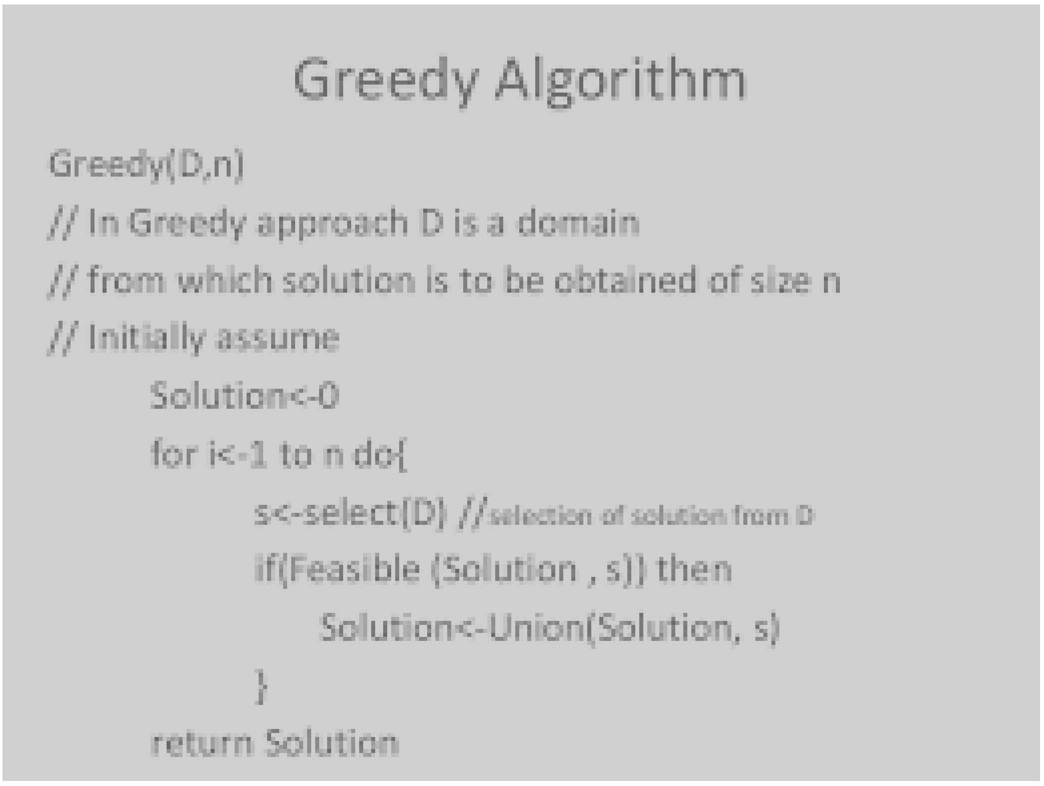
Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.

- **A solution function** − Used to indicate whether a complete solution has been reached.

How to Create a Greedy Algorithm?



Greedy Algorithm

```
Greedy(D,n)
// In Greedy approach D is a domain
// from which solution is to be obtained of size n
// Initially assume
        Solution<-0
        for i<-1 to n do{
                s<-select(D) //selection of solution from D
                if(Feasible (Solution , s)) then
                     Solution<-Union(Solution, s)
                }
        return Solution
```

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

Knapsack Problem

- Given n objects and a knapsack or bag. Object i has a weight w_i , Profit p_i and the knapsack has a capacity W.
- If a fraction x_i, $0\leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- The problem can be stated as

$$\text{maximize}\,\{f(\mathbf{x})\} = \text{maximize}\left\{\sum_{i=1}^{n} x_i p_i\right\} \tag{1}$$

$$\text{subject to}\quad \sum_{i=1}^{n} x_i w_i \leq W \tag{2}$$

$$x_i \in \{0,1\},\ i = 1,2,\ldots,n$$

$$\textbf{and } \mathbf{0 \leq x_i \leq 1,\ 1 \leq i \leq n} \tag{3}$$

- A feasible solution is any set $(x_1,\ldots\ldots x_n)$ satisfying 2 and 3 above. An optimal solution is a feasible solution for which 1 is maximized.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

- If the provided items are already sorted into a decreasing order of pi/wi. The total time including the sort is in ***O(n logn)***.

Example

For the given set of items and knapsack capacity = 20kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Profit
1	18	25
2	15	24
3	10	15

Solution

Step- 01

Compute the profit/ weight ratio for each item.

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

Step- 02

Sort all the items in decreasing order of p_i/ w_i ratio

I2	**I3**	**I1**
1.6	1.5	1.38

Step- 03

Start filling the knapsack by putting the items into it one by one.

Knapsack Capacity	Items in Knapsack	Cost
20	0	0
5	I2	24

Now,

- Knapsack weight left to be filled as 5 but item- 3 has a weight of 10.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items

< I2,(5/10)I3>

Total cost of the knapsack

= 24+(5/10)15

=31.5 units.

Job Sequencing with Deadlines

- Sequencing jobs on a single processor with deadline constraints is called job sequencing with deadlines.
- We are given a set of n jobs.
  - Each job has a defined deadline and a certain profit is associated with it.
  - We get profit for a job only when the particular job is completed within the deadline.
  - A single processor is available to handle all jobs.
  - The processor takes a unit of time to complete a job.
- The problem is stated as follows:

There are n jobs let us say S={1,2,...,n} and each job i has a deadline d_i>=0 and a profit p_i>=0. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i if job i is completed by its deadline but only one machine is available for processing. A feasible solution is a subset of jobs J, such that each job in the subset is completed by its deadline gaining a profit $p_{i.}$ An optimal solution is a feasible solution that maximizes total profit.

Thus, D(i)>0 for $1 \leqslant i \leqslant n$.

Initially, these jobs are ordered according to profit, i.e. $p_1 \geqslant p_2 \geqslant p_3 \geqslant ... \geqslant p_n$

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
   r := k
   while D(J(r)) > D(i) and D(J(r)) ≠ r do
      r := r – 1
   if D(J(r)) ≤ D(i) and D(i) > r then
      for l = k … r + 1 by -1 do
         J(l + 1) := J(l)
         J(r + 1) := i
         k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$.

Example

n= 4, (p1,p2,p3,p4)=(100,10,15,27), (d1,d2,d3,d4)=(2,1,2,1)

Solution

Different feasible solution with the sequencing of jobs and total profit are given below:

Feasible solution	Processing Sequence	Value
1. (1,2)	2,1	110
2. (1,3)	1,3 or 3,1	115
3. (1,4)	4,1	**127 (optimal solution)**
4. (2,3)	2,3	25
5. (3,4)	4,3	42

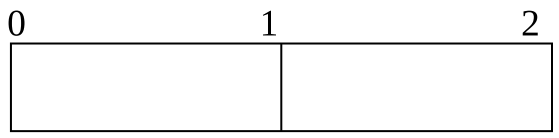
Step - 01

Sort all the given jobs in decreasing order of the profit

Jobs	**J1**	**J4**	**J3**	**J2**
Deadline	2	1	2	1
Profit	100	27	15	10

Step - 02

Value of maximum deadline =2.
So, draw a Gantt chart with maximum time on Gantt chart =2 units as shown-



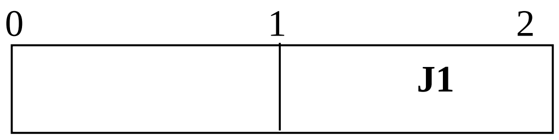
Gantt Chart

Now,

- We take each job one by one in the order they appear in step-01.
- We place the job on Gantt chart as far as possible from 0.

Step – 03

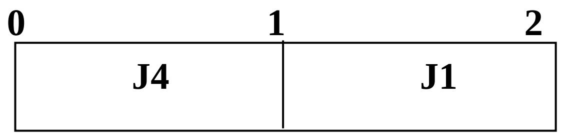
- we take job J1
- since its deadline is 2, so we place it in the first empty cell before deadline 2 as



J={1} is a feasible solution

<u>Step – 04</u>

- we take job J4
- since its deadline is 1, so we place it in the first empty cell before deadline 1 as



- The solution J={1,4} is a feasible solution.
- Next, Job 3 is considered and discarded as J={1,3,4} is not feasible.
- Finally, job 2 is considered for inclusion into J. It is discarded as J={1,2,4} is not feasible.
- Hence, we are left with the solution J={1,4} with value 127. This is the optimal solution for the given problem instance.

Minimum Cost Spanning Trees

Spanning Tree

A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

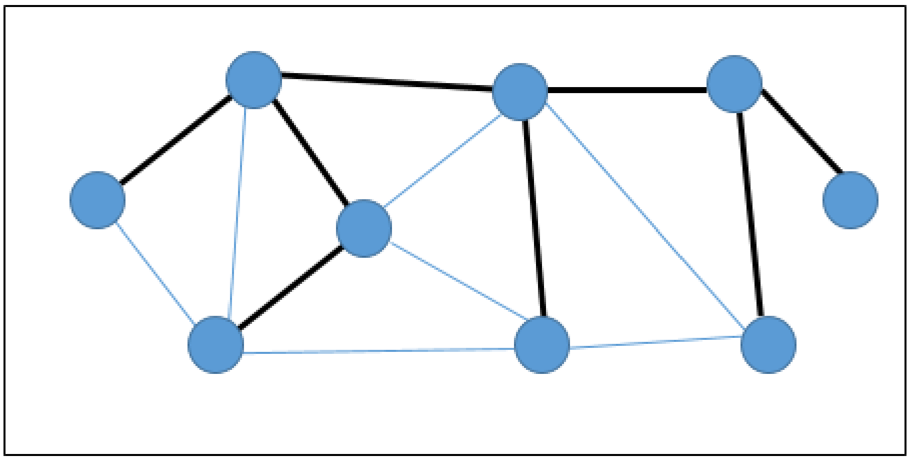
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.



Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. One graph may have more than one spanning tree. If there are **n** numbers of vertices, the spanning tree should have ***n* - 1** number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

Prim's Algorithm

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

Prim's Algorithm Implementation

The implementation of Prim's Algorithm is explained in the following steps-

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

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

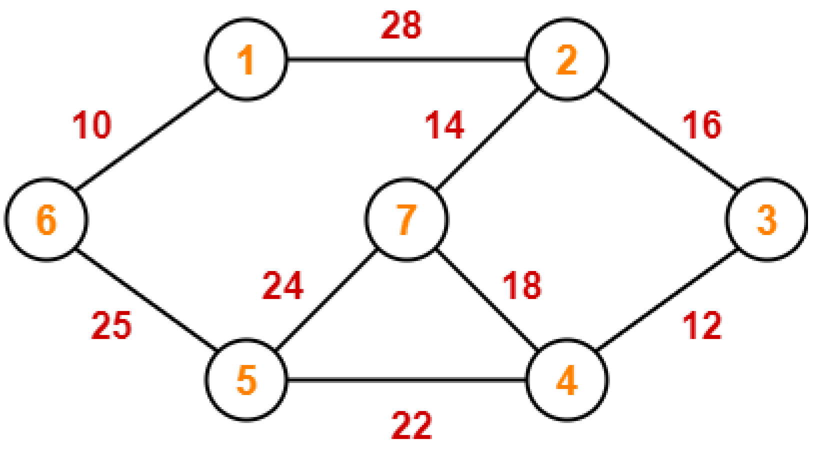
- Keep repeating step-02 until all the vertices is included and Minimum Spanning Tree (MST) is obtained.

Prim’s Algorithm Time Complexity-

Worst case time complexity of Prim’s Algorithm is-

- O(ElogV) using binary heap
- O(E + VlogV) using Fibonacci heap

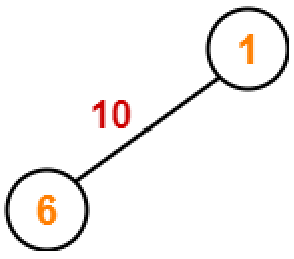
Example



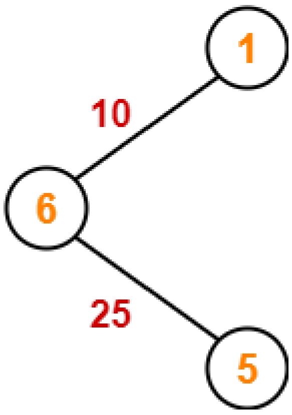
Solution-

The above discussed steps are followed to find the minimum cost spanning tree using Prim’s Algorithm-

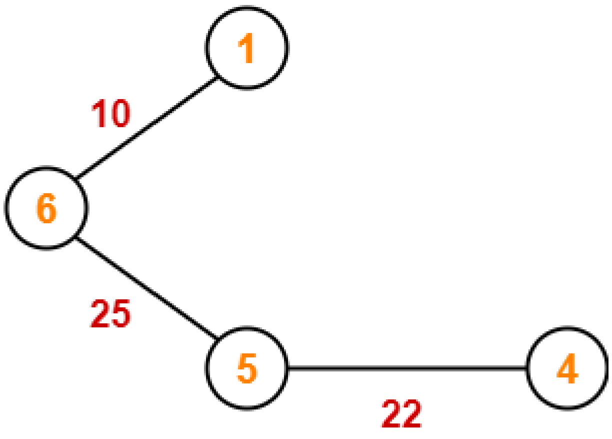
Step-01:



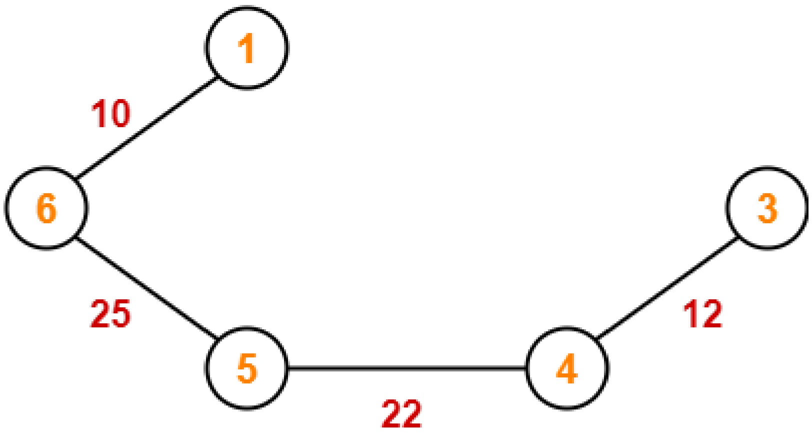
Step-02:



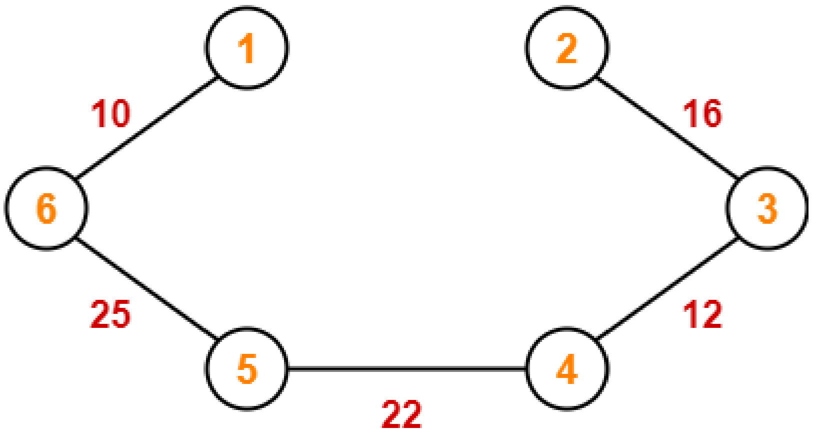
Step-03:



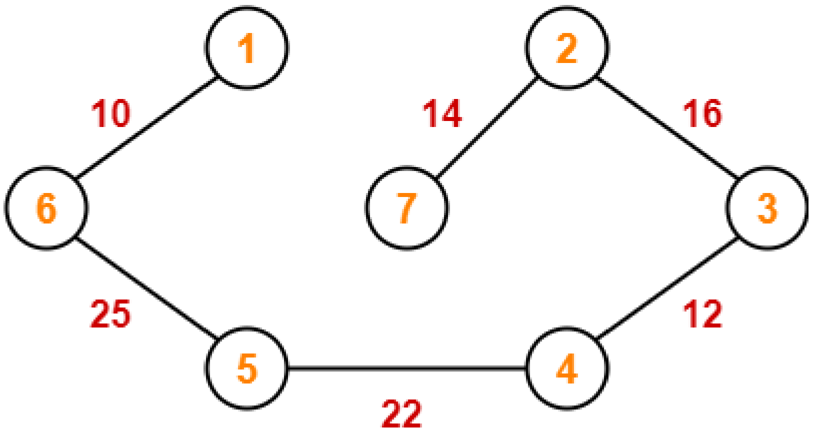
Step-04:



Step-05:



Step-06:



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Kruskal's Algorithm

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm Implementation-

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

The implementation of Kruskal's Algorithm is explained in the following steps-

<u>step-01:</u>

- Sort all the edges from low weight to high weight.

<u>Step-02:</u>

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

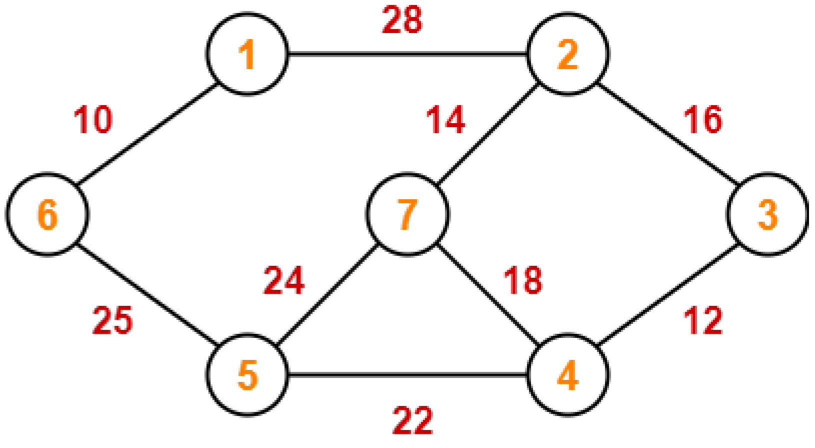
- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Kruskal's Algorithm Time Complexity-

Worst case time complexity of Kruskal's Algorithm

O(ElogV) or O(ElogE)

Example-

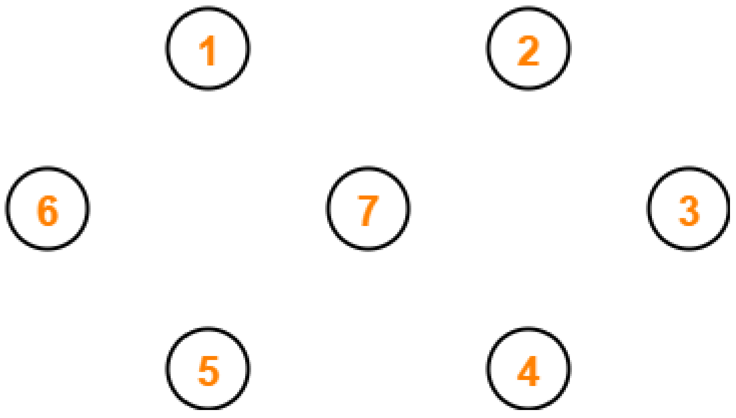


Solution-

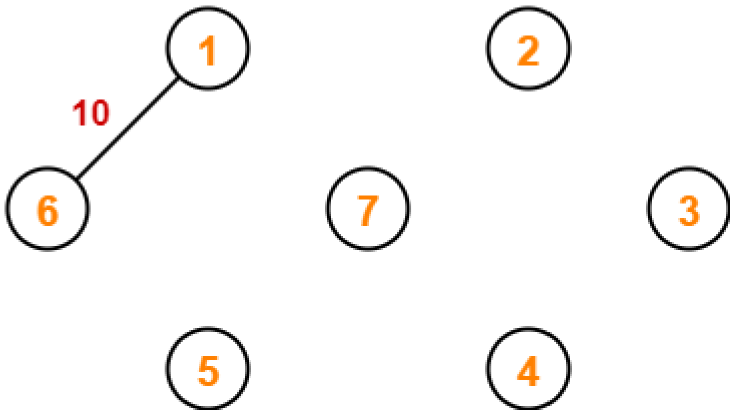
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

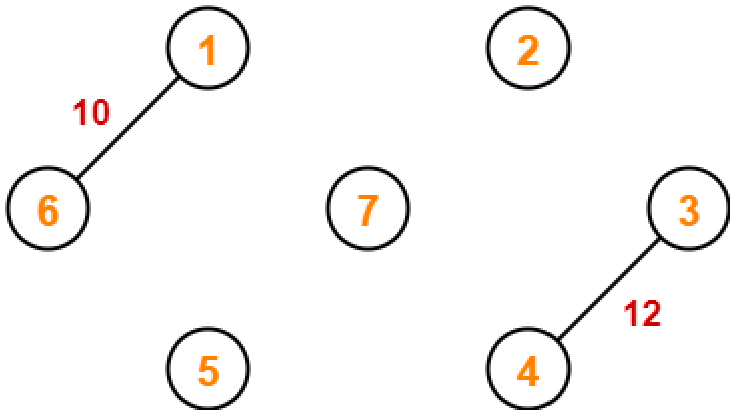
Step-01



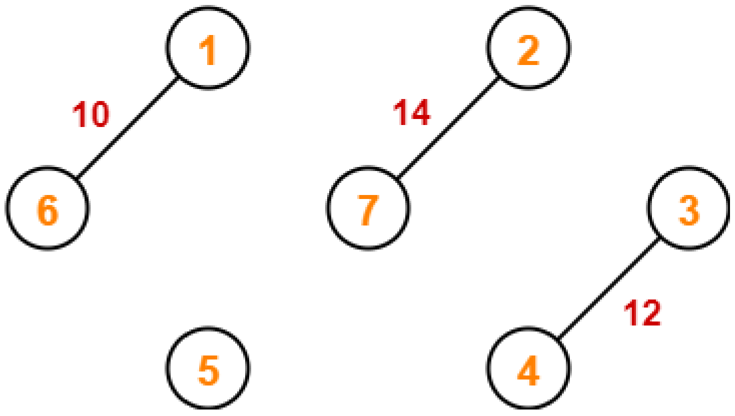
Step-02:



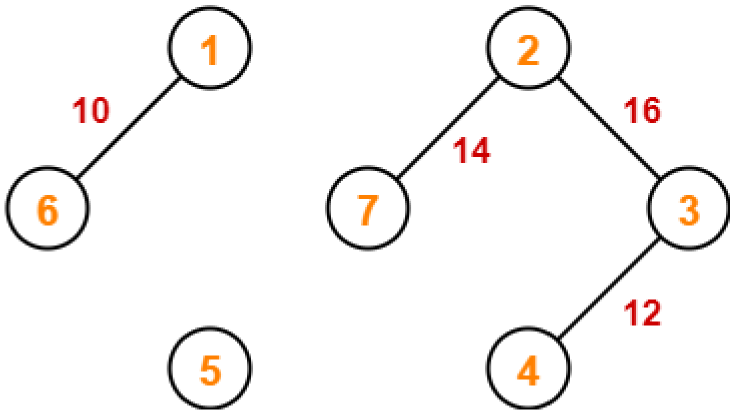
Step-03:



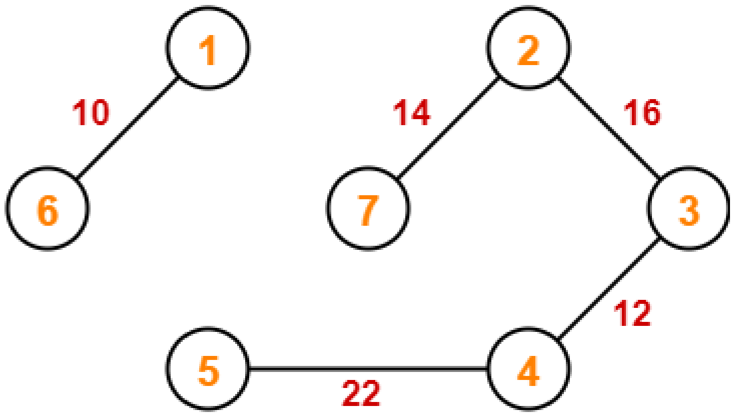
Step-04:



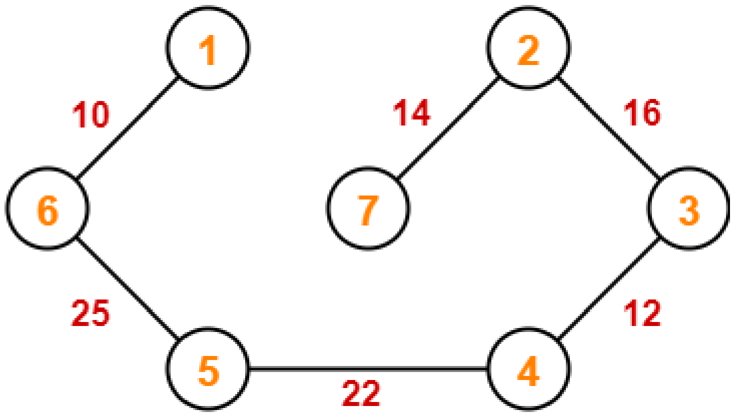
Step-05:



Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Optimal Merge Patterns

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do
   declare new node
   node.leftchild := least (list)
   node.rightchild := least (list)
   node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
   insert (list, node);
return least (list);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence −

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

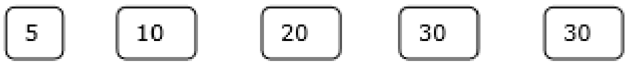
Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

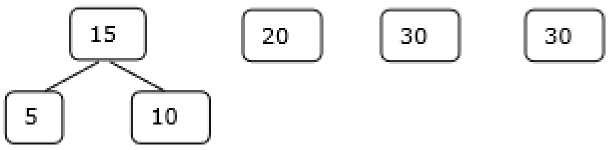
Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

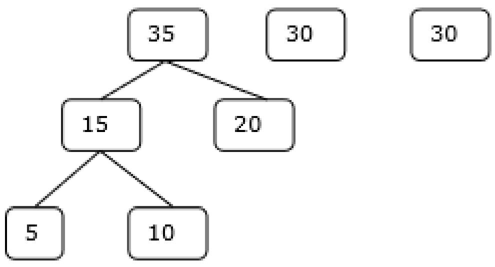
Initial Set



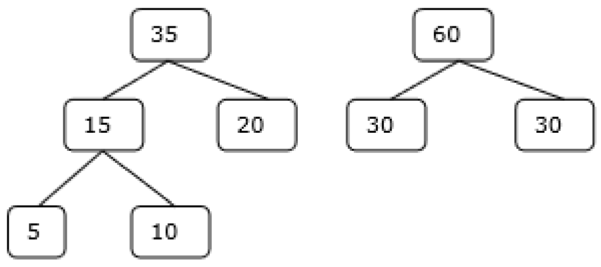
Step-1



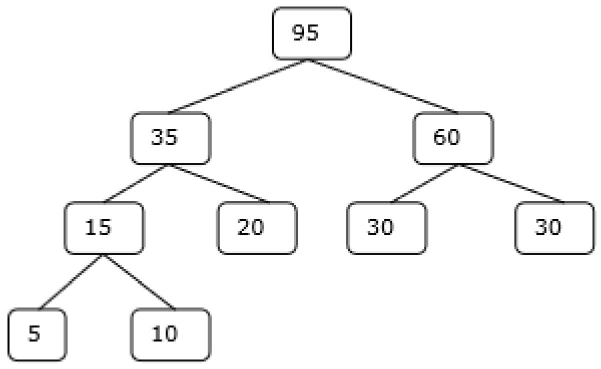
Step-2



Step-3



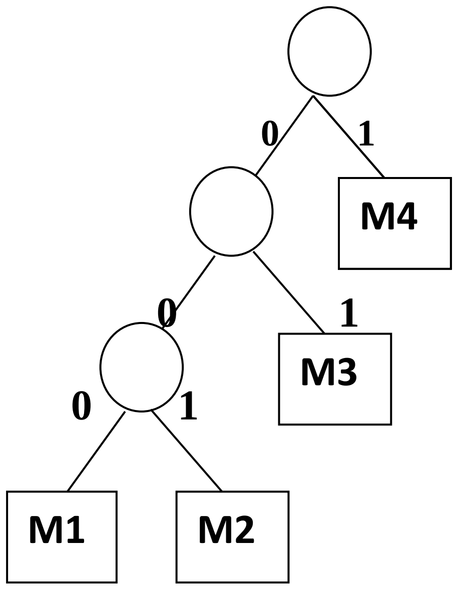
Step-4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Huffman Codes

- Binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages M_1,......,$M_{n+1.}$
- Each code is a binary string that is used for transmission of the corresponding message.
- At the receiving end the code is decoded using a decode tree.
- A decode tree is a binary tree in which external nodes represent messages.
- The binary bits code word for a message determine the branching needed at each level of the decode tree to reach the correct external node.



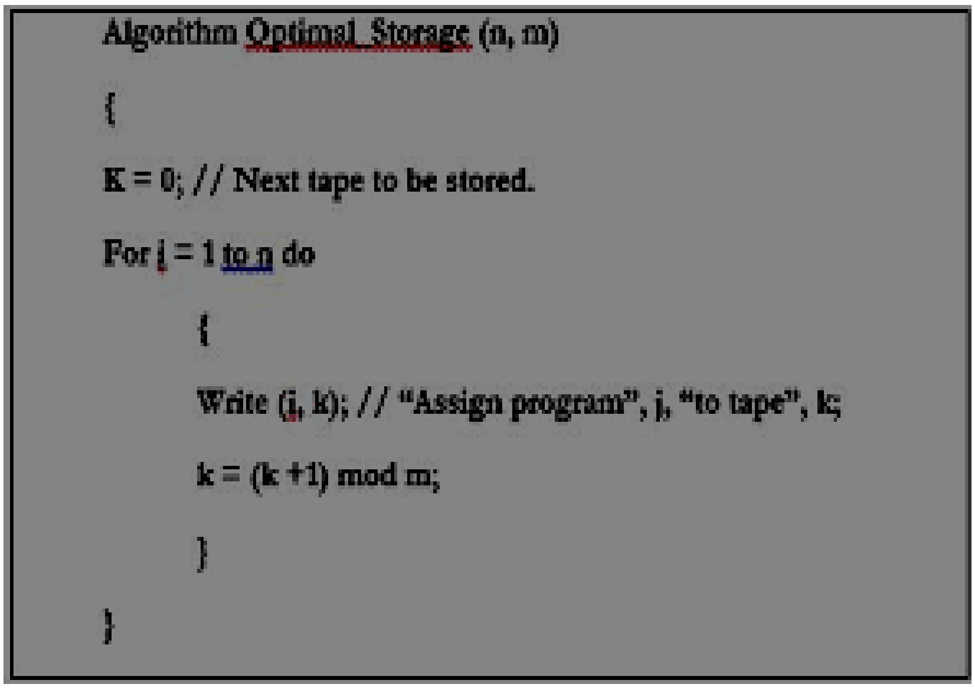
- We interpret a zero as a left branch and a one as a right branch, then the decode tree corresponds to codes 000,001,01 and 1 for messages M_1, M_2, M_3 and M_4 respectively. These codes are called Huffman codes.
- The cost of decoding a code word is propositional to the number of bits in the code.
- This number is equal to the distance of the corresponding external node from the root node.
- If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is$\sum_{1\leq i\leq n+1} q_i d_i$ where d_i is the distance of the external for message Mi from the root node.
- The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path

Optimal Storage on Tapes

1. Given n programs to be stored on tape, the lengths of these programs are i1, i2….in respectively. Suppose the programs are stored in the order of i1, i2…in. 2 We have a tape of length L i.e. the storage capacity of the tape is L. We are also given n programs where length of each program is i is Li.

2. Let Tj be the time to retrieve program ij.
3. It is now required to store these programs on the tape in such a way so that the mean retrieval time is minimum. MRT is the average tome required to retrieve any program stored on this tape.
4. Assume that the tape is initially positioned at the beginning.
5. Tj is proportional to the sum of all lengths of programs stored in front of the program ij.
6. The goal is to minimize MRT (Mean Retrieval Time),(1/n) $\sum_{i=0}^{n} Tj$

I.e., want to minimize $\sum_{j=1}^{n}\sum_{k=1}^{j} Ti$



```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
        {
        Write (i, k); // "Assign program", i, "to tape", k;
        k = (k +1) mod m;
        }
}
```

Example:

```
Input : n = 3 L[] = { 5, 3, 10 }
Output : Order should be { 3, 5, 10 } with MRT = 29/3
```

1. Here, n=3 and (L1, L2, L3) = (5, 10, 3). We can store these 3 programs on the tape in any order but we want that order which will minimize the MRT.

2. Suppose we store the programs in order (L1, L2, L3).
3. Then MRT is given as (5+(5+10)+(5+10+3))/3=38/3
4. To retrieve L1 we need 5 units of time. Because a tape is a sequential device we will have to first pass through entire L1 even if we want to retrieve L2.
5. Hence, retrieval time (RT) is 5 for program 1 and (5+10) for program 2.
6. Similarly, if program 3 is also considered then the total RT becomes 5+ (5+10) + (5+10+3) where (5+10+3) is the RT for program 3.
7. Since we want to find the mean retrieval time we add all the RT and then divide the sum by n.
8. The aim over here is to find the minimum MRT. To do this we consider all the possible orderings of these 3 programs. Since there 3 programs we can have at the most 6(3!) combinations.
9. Consider the below table:

Ordering	**MRT**
L1,L2,L3	5+(5+10)+(5+10+3)/3=38/3
L1,L3,L2	5+(5+3)+(5+10+3)/3=31/3
L2,L1,L3	10+(5+10)+(5+10+3)/3=43/3

Ordering	**MRT**
L2,L3,L1	10+(3+10)+(5+10+3)/3=41/3
L3,L1,L2	3+(5+3)+(5+10+3)/3=29/3
L3,L2,L1	3+(3+10)+(5+10+3)/3=34/3

1. It should be seen that the minimum MRT of (29/3) is obtained in case of (L1, L2, L3). Hence the optimal solution is achieved if the programs are stored in increasing order of their lengths.
2. Hence, a greedy approach to solving the problem is continuously select programs in increasing order of their lengths.
3. If L is an array having program length in ascending order.
4. The time complexity of this algorithm including the time to do sorting is O(n^2).

Single Source Shortest Paths

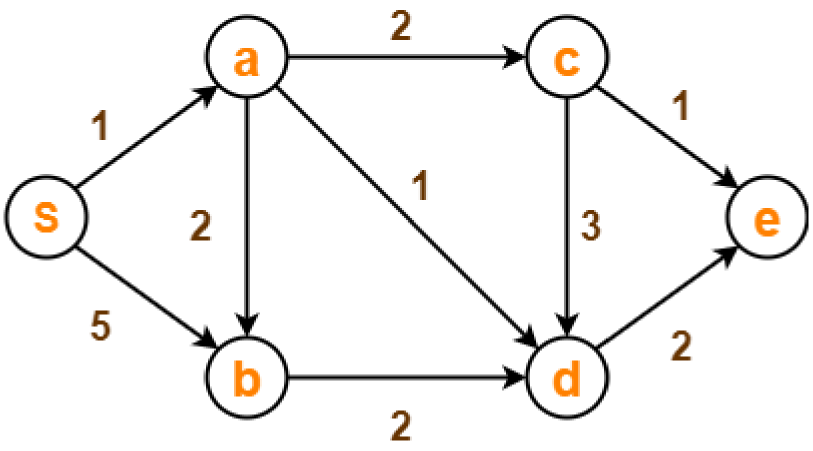
➢ Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.

- The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway
- The length of a path is now defined to be the sum of the weights of the edges on that path.
- The starting vertex of the path is referred to as the source, and the last vertex the destination.
- We are given a directed graph G=(V,E), a weighting function cost for the edges of G, and a source vertex $v_{o.}$
- The problem is to determine the shortest paths from v_0 to all the remaining vertices of G.
- The shortest path between v0 and some other node v is an ordering among s subset of the edges. Hence this problem fits the ordering paradigm.
- To generate the shortest paths in this order, we need to be able to determine
  1. The next vertex to which a shortest path must be generated
  2. A shortest path to this vertex. Let S denote the set of vertices (including v0) to which the shortest paths have already been generated . For w not is S, let dist[w] be the length of the shortest path starting from v0, going through only those vertices that are in S, and ending at w.

```
**Algorithm** ShortestPaths(v, cost, dist, n)
{
for i:=1 to n do
{
        S[i]:=false;
        dist[i]:=cost[v,i];
}
S[v]:=**true;**
dist[v]:=0.0;
**for** num:=2 to n do
{
        Choose u from among those vertices not in S such that dist[u] is
minimum;
S[u]:=true;
**for** (each w adjacent to u with S[w]=**false) do**
        **if**(dist[w]:=dist[u]+cost[u,w];
}
}
```

Example

Find the shortest distance from source vertex 'S' to remaining vertices in the following graph-



Length Adjacency Matrix

$$
\begin{array}{c|cccccc}
 & S & a & b & c & d & e \\
\hline
S & 0 & 1 & 5 & & & \\
A & & 0 & 2 & 2 & 1 & \\
B & & & 0 & & 2 & \\
\end{array}
$$

C				0	3	1
D					0	2
E						0

Also, write the order in which the vertices are visited.

Solution-

Step-01:

The following two sets are created-

- Unvisited set : {S , a , b , c , d , e}
- Visited set : { }

Step-02:

The two variables Π and d are created for each vertex and initialized as-

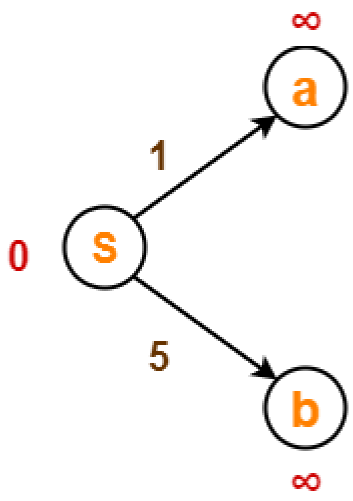
- Π[S] = Π[a] = Π[b] = Π[c] = Π[d] = Π[e] = NIL
- d[S] = 0
- d[a] = d[b] = d[c] = d[d] = d[e] = ∞

Step-03:

Vertex ‘S’ is chosen.

- This is because shortest path estimate for vertex ‘S’ is least.
- The outgoing edges of vertex ‘S’ are relaxed.

Before Edge Relaxation-



Now,

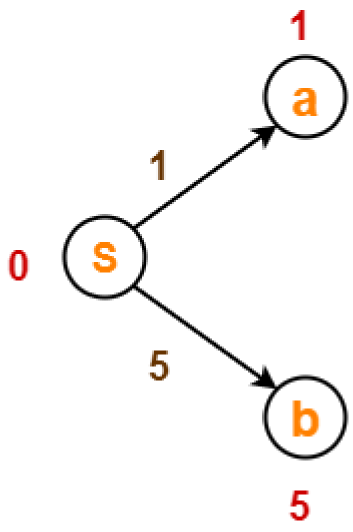
- d[S] + 1 = 0 + 1 = 1 < ∞

∴ d[a] = 1 and Π[a] = S

- d[S] + 5 = 0 + 5 = 5 < ∞

∴ d[b] = 5 and Π[b] = S

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

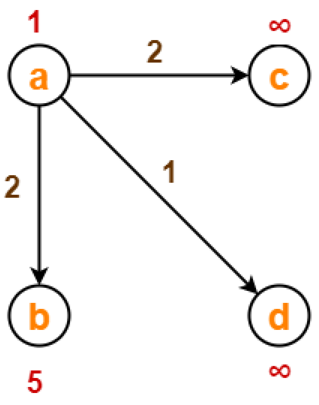
- Unvisited set : {a , b , c , d , e}
- Visited set : {S}

Step-04:

Vertex ‘a’ is chosen.

- This is because shortest path estimate for vertex ‘a’ is least.
- The outgoing edges of vertex ‘a’ are relaxed.

Before Edge Relaxation-



Now,

- d[a] + 2 = 1 + 2 = 3 < ∞

∴ d[c] = 3 and Π[c] = a

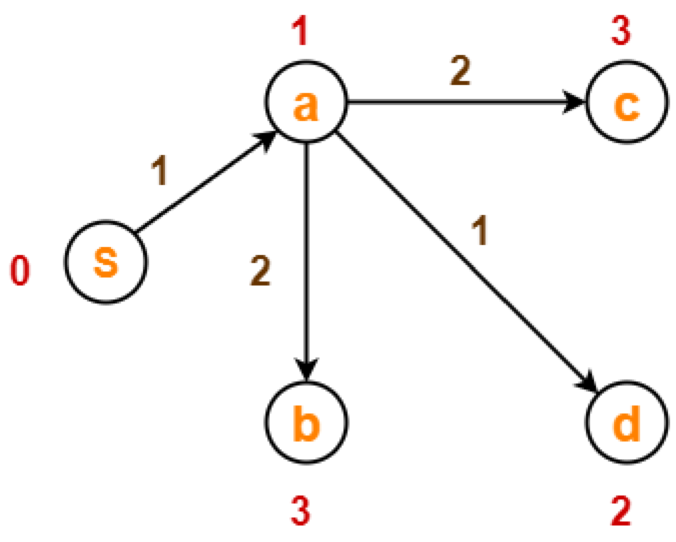
- d[a] + 1 = 1 + 1 = 2 < ∞

∴ d[d] = 2 and Π[d] = a

- d[b] + 2 = 1 + 2 = 3 < 5

∴ d[b] = 3 and Π[b] = a

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

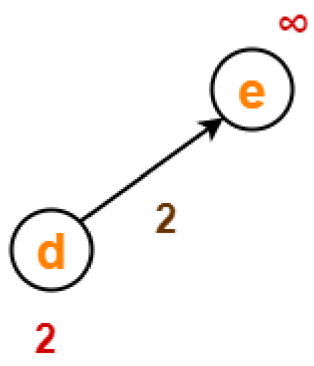
- Unvisited set : {b , c , d , e}
- Visited set : {S , a}

Step-05:

Vertex ‘d’ is chosen.

- This is because shortest path estimate for vertex ‘d’ is least.
- The outgoing edges of vertex ‘d’ are relaxed.

Before Edge Relaxation-

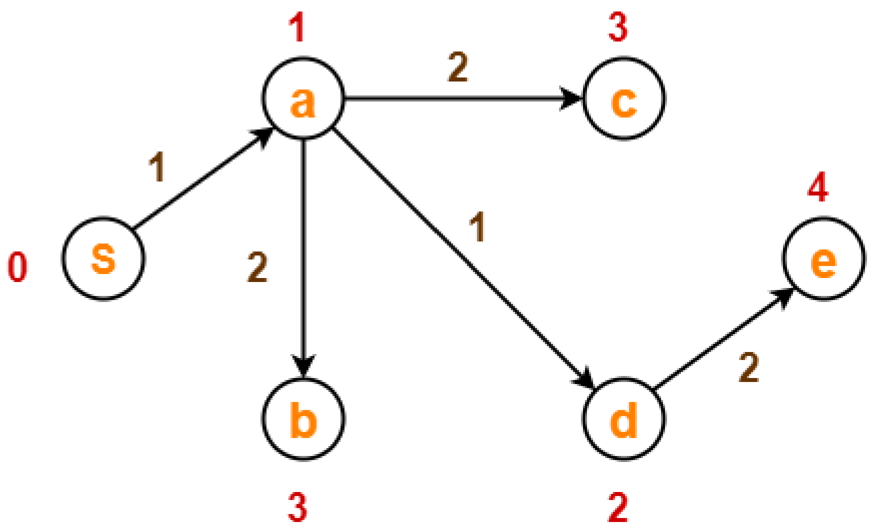


Now,

- d[d] + 2 = 2 + 2 = 4 < ∞

∴ d[e] = 4 and Π[e] = d

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

- Unvisited set : {b , c , e}
- Visited set : {S , a , d}

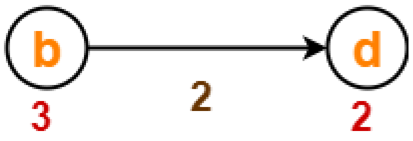
Step-06:

Vertex ‘b’ is chosen.

- This is because shortest path estimate for vertex ‘b’ is least.
- Vertex ‘c’ may also be chosen since for both the vertices, shortest path estimate is least.

- The outgoing edges of vertex ‘b’ are relaxed.

Before Edge Relaxation-



Now,

- d[b] + 2 = 3 + 2 = 5 > 2

∴ No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

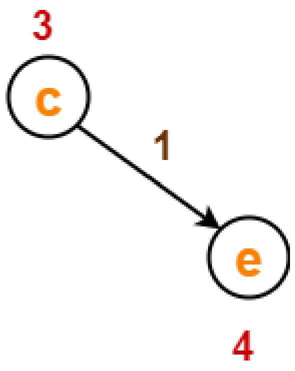
- Unvisited set : {c , e}
- Visited set : {S , a , d , b}

Step-07:

Vertex ‘c’ is chosen.

- This is because shortest path estimate for vertex ‘c’ is least.
- The outgoing edges of vertex ‘c’ are relaxed.

Before Edge Relaxation-



Now,

- d[c] + 1 = 3 + 1 = 4 = 4

∴ No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

- Unvisited set : {e}
- Visited set : {S , a , d , b , c}

Step-08:

Vertex ‘e’ is chosen.

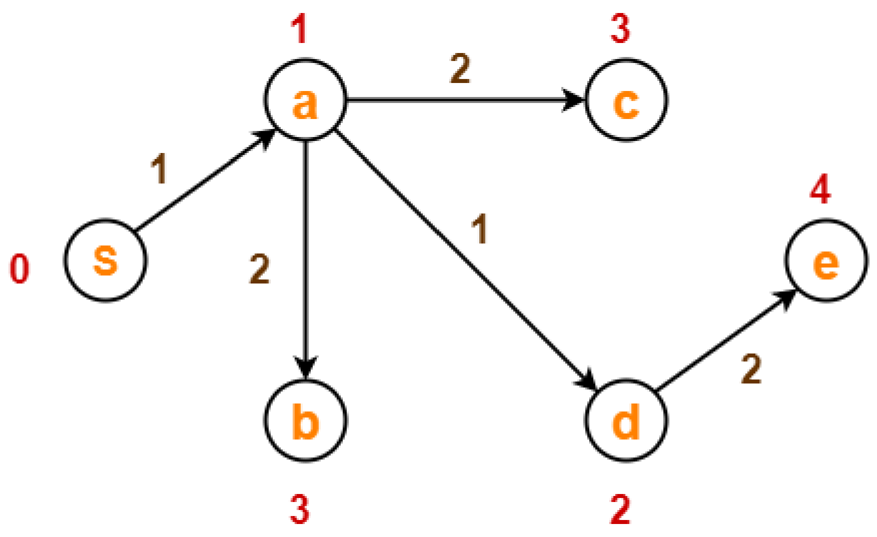
- This is because shortest path estimate for vertex ‘e’ is least.
- The outgoing edges of vertex ‘e’ are relaxed.
- There are no outgoing edges for vertex ‘e’.
- So, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

- Unvisited set : { }
- Visited set : {S , a , d , b , c , e}

Now,

- All vertices of the graph are processed.
- Our final shortest path tree is as shown below.
- It represents the shortest path from source vertex ‘S’ to all other remaining vertices.



Shortest Path Tree

The order in which all the vertices are processed is :

S , a , d , b , c , e.