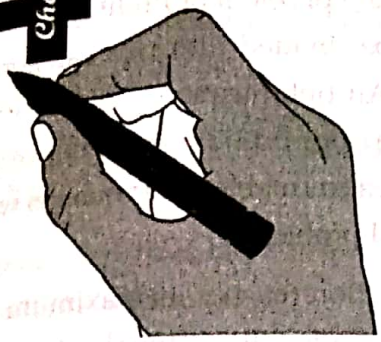
Chapter 4



GREEDY ALGORITHMS

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Optimization Problems and Optimal Solution, Introduction of Greedy Algorithms, Elements of Greedy Strategy.
- Greedy Algorithms: Fractional Knapsack, Job sequencing with Deadlines, Kruskal's Algorithm, Prims Algorithm, Dijkstra's Algorithm and their Analysis
- Huffman Coding: Purpose of Huffman Coding, Prefix Codes, Huffman Coding Algorithm and its Analysis

Optimization Problems and Optimal Solution

In mathematics and computer science, an optimization problem is the problem of finding the best solution from all feasible solutions. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete. An optimization problem with discrete variables is known as a discrete optimization. In a discrete optimization problem, we are looking for an object such as an integer, permutation or graph from a countable set. Problems with continuous variables include constrained problems and multimodal problems.

An optimal solution is a feasible solution where the objective function reaches its maximum (or minimum) value – for example, the most profit or the least cost. A globally optimal solution is one where there are no other feasible solutions with better objective function values. A locally optimal solution is one where there are no other feasible solutions in the vicinity with better objective function values.

Introduction of Greedy Algorithms

In this method, we have to find out the best method/option out of many present ways. In this approach/method we focus on the first stage and decide the output, don't think about the future. This method may or may not give the best output.

Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

- **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
- **Optimal substructure:** Optimal solutions contain optimal sub-solutions. In other words, answers to sub-problems of an optimal solution are optimal.

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Elements of Greedy Strategy

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.

Greedy Algorithms: Fractional Knapsack

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is wi and it worth vi. Any amount of item can be put into the bag i.e. xi fraction of item can be collected, where $0 \le xi \le 1$. Here the objective is to collect the items that maximize the total profit earned. Here we arrange the items by ratio vi/wi.

Algorithm

```
GreedyFracKnapsack (W, n)
{
    for(i=1; i<=n; i++)
        x[i] = 0.0;
    tempW = W;
    for(i=1; i<=n; i++)
    {
        if(w[i] > tempW) then
            break;
        x[i] = 1.0;
        tempW -= w[i];
    }
    if(i<=n)
        x[i] = tempW/w[i];
}
```

Analysis

We can see that the above algorithm just contain a single loop i.e. no nested loops the running time for above algorithm is O(n). However our requirement is that v[1 … n] and w[1 … n] are sorted, so we can use sorting method to sort it in O(n log n) time such that the complexity of the algorithm above including sorting becomes O(n log n).

Example: **Consider five items along with their respective weights and values,**

I = {I1, I2, I3, I4, I5}
w = {5, 10, 20, 30, 40}
v = {30, 20, 100, 90, 160}
The knapsack has capacity W=60, then find optimal profit earned by using fractional knapsack.

Solution: Initially

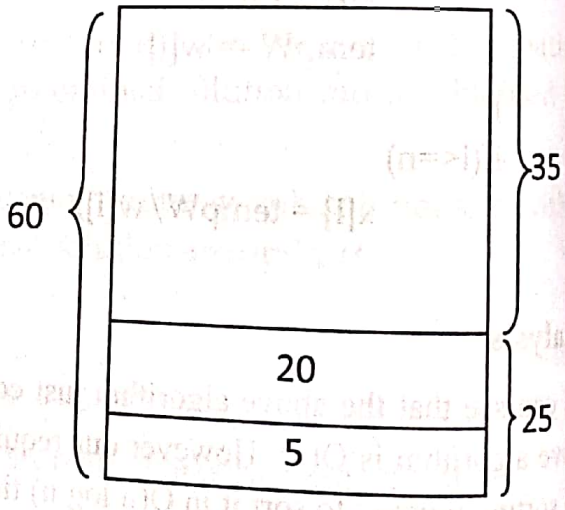
Items	wi	vi
I1	5	30
I2	10	20
I3	20	100
I4	30	90
I5	40	160

Step 2: calculate vi/wi as,

Items	Wi	vi	Pi=vi/wi
I1	5	30	6.0
I2	10	20	2.0
I3	20	100	5.0
I4	30	90	3.0
I5	40	160	4.0

Step 3: Arranging the items with decreasing order of Pi as,

Items	Wi	vi	Pi=vi/wi
I1	5	30	6.0
I3	20	100	5.0
I5	40	160	4.0
I4	30	90	3.0
I2	10	20	2.0



Now filling the knapsack according to decreasing value of Pi

Since, 40 w = 160 vi

1 w = 160/40 = 4vi

35 w = 35 * 4 = 140 vi

Thus, maximum value=v1 + v2+new (v3) = 30+100+140=270

Job sequencing with Deadlines

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of jobs, which will be completed within its deadlines and it should yield a maximum profit.

Let there are a number of jobs J = {j1, j2, j3, j4……..,jn}

Deadline of jobs = {d1, d2, d3, d4, ……dn}

The profit can be earned if job is completed within their deadline = { p1, p2, ……pn}

Here every job can be completed in unit time (i.e. first job begins at time 0 and finished at time 1, the second job begins at time 1 and finished at time 2 and so on.) and we have a single machine (processor). The main aim of this problem is to find the feasible sequence of jobs that maximize the profit earned.

Features of algorithm

- There are n jobs to be processes on a machine
- Each job i has a deadline di ≥ 0 and profit Pi ≥ 0
- Pi is earned iff the job is completed by its deadline
- The job is completed if it is processed on a machine for unit time
- Only one machine is available for processing jobs
- Only one job is processed at a time on the machine

Example 1: 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	J1	J2	J3	J4	J5
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	J2	J1	J4	J3	J5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

Job	Feasible/Non-feasible	Processing sequence	Total profit
J2	Feasible	{J2}	100
J1	Feasible	{J2, J1}	100+60=160
J4	Not feasible	{J2, J1}	160
J3	Feasible	{J2, J1, J3}	160+20=180
J5	Not feasible	{J2, J1, J3}	180

- From this set of jobs, first we select J2, as it can be completed within its deadline and contributes maximum profit.
- Next, J1 is selected as it gives more profit compared to J4.
- In the next clock, J4 cannot be selected as its deadline is over, hence J3 is selected as it executes within its deadline.
- The job J5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J2, J1, J3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is 100 + 60 + 20 = 180.

Example 2: let's assume that there are 4 jobs

n = 4

J = (j1, j2, j3, j4)

D = (d1, d2, d3, d4) = (2, 1, 2, 1)

P = (100, 10, 15, 27)

Find the sequence due to which maximize the profit by using job sequencing with deadline algorithm

Solution:

According to greedy algorithm for this problem, at first sort the jobs on the basis of profit in descending order as,

J = (j1, j4, j3, j2)

D = (d1, d4, d3, d2) = (2, 1, 2, 1)

P = (100, 27, 15, 10)

Job	Feasible/Non-feasible	Processing sequence	Total profit
j1	Feasible	{j1}	100
j4	Feasible	{j4, j1}	100+27=127
j3	Not feasible	{j4, j1}	127
j2	Not feasible	{j4, j1}	127

Thus the optimal profit=127 with the processing sequence= {j4, j1}

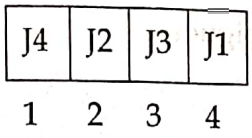
Example 3: Find the optimal schedule for the following seven tasks (jobs) with given value and deadlines.

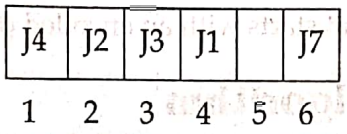
Job	1	2	3	4	5	6	7
Di	4	2	4	3	1	4	6
Vi	70	60	50	40	30	20	10

Solution:

Since the given jobs are already in sorted order of their profit so we do not need sorting these jobs.

Job	Feasible/Non-feasible	Processing sequence	Total profit
j1	Feasible	{j1}	70
j2	Feasible	{j2, j1}	70+60=130
j3	Feasible	{j2, j3, j1}	130+50=180
j4	Feasible	{j4, j2, j3, j1}	180+40=220
j5	Not Feasible	{j4, j2, j3, j1}	220
j6	Not Feasible	{j4, j2, j3, j1}	220
j7	Feasible	{j4, j2, j3, j1, j7}	220+10=230





Thus the optimal profit=230 with the processing sequence= {j4, j2, j3, j1, j7}

Algorithm

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit. Assume, deadline of i^{th} job J_i is D_i and the profit received from this job is P_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

```
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 $O(n^2)$.

Huffman Coding

Huffman Coding also called as Huffman Encoding is a famous greedy algorithm that is used for the lossless compression of data. It uses variable length encoding where variable length codes are assigned to all the characters depending on how frequently they occur in the given text. The character which occurs most frequently gets the smallest code and the character which occurs least frequently gets the largest code.

Prefix Codes

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

Huffman Coding Algorithm

"Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights."

This method is mainly applicable to many forms of data transmission. In 1951, David Huffman found the "most efficient method of representing numbers, letters, and other symbols using binary code". Now standard method used for data compression. In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm.

Initially 2 nodes are considered and their sum forms their parent node. When a new element is considered, it can be added to the tree. Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters. There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Start
2. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

3. Extract two nodes with the minimum frequency from the min heap.
4. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
5. Repeat steps 3 and 4 until the heap contains only one node. The remaining node is the root node and the tree is complete.
6. Stop

Example: Let us take any four characters and their frequencies, and sort this list by increasing frequency. Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

Character	Frequencies
A	03
T	07
E	10
O	05

Now sort these characters according to their frequencies in non-decreasing order.

Character	Frequencies	Code
A	03	00
O	05	01
T	07	10
E	10	11

Here before using Huffman algorithm the total number of bits required is nb=3*2+5*2+7*2+10*2

= 06+10+14+20

= 50 bits

Now using Huffman man algorithm as,

Step 1:

A	3

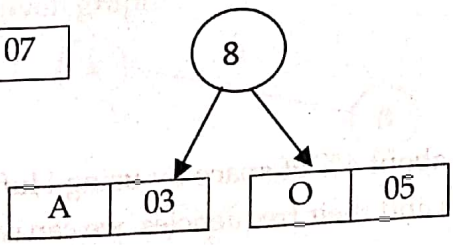
O	5

T	7

E	10

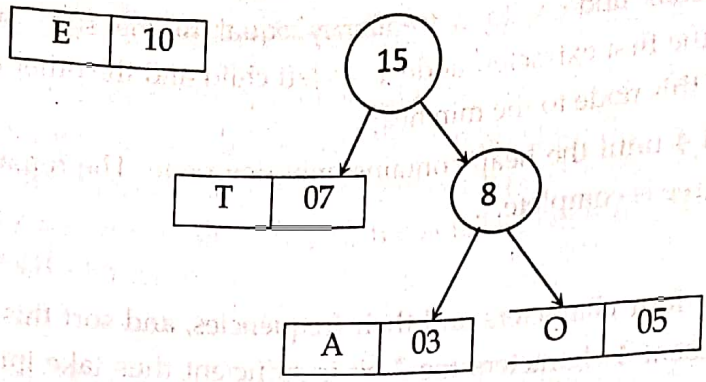
Step 2:

T	07

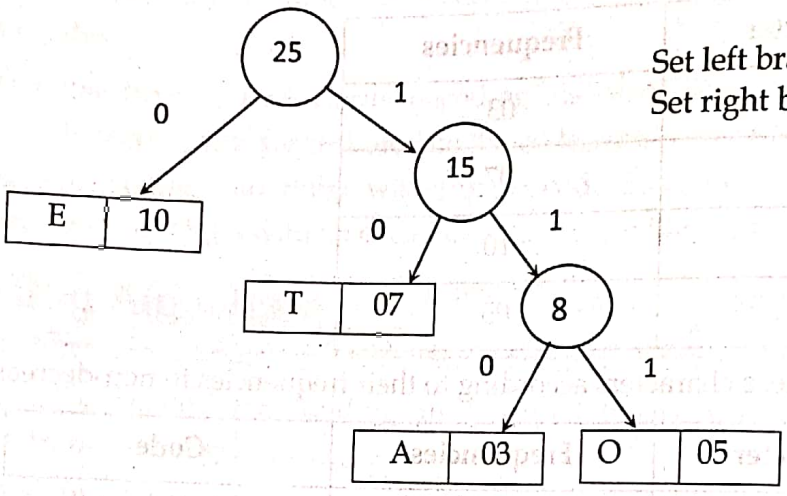


E	10

Step 3:



Step 4:



Set left branch=0 &
Set right branch=1

Now from variable length code we get following code sequence.

Character	Frequencies	Code
A	03	110
O	05	111
T	07	10
E	10	0

Thus after using Huffman algorithm the total number of bits required is:

Number of bits = 3*3+5*3+7*2+10*1

= 09+15+14+10

= 48 bits

= (50-48)/50*100%

= 4%

Since in this small example, we can save about 4% of space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies, we can save a lot of space.

Algorithm

```
Procedure Huffman(C)   // C is the set of n characters and related information
{
    n = C.size
    Q = priority_queue()
    For i = 1 to n
      n = node(C[i])
      Q.push(n)
    End for
    While Q.size() is not equal to 1
      Z = new node()
      Z.left = x = Q.pop
      Z.right = y = Q.pop
      Z.frequency = x.frequency + y.frequency
      Q.push(Z)
    End while
    Return Q
}
```

Analysis

Since we need to sort the given input symbols in ascending order of their frequencies thus sorting takes O(nlogn) time. The for loop executes at most O(n) time also while loop executes at most O(n) time.

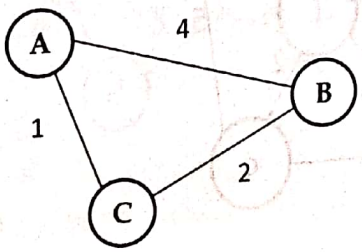
So total time complexity = O(n logn) + O(n) +O(n)

⇨ T(n) = O(n logn)

Kruskal's Algorithm

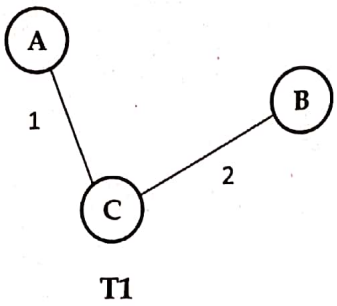
The possible trees constructed from given graph is called spanning trees and the out of many possible spanning trees of given weighted graph choosing one of the spanning tree with minimum possible total weight is called minimum spanning tree.

Example: Find MST of given graph

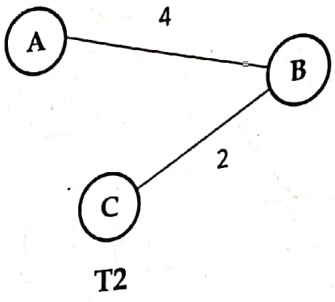


Solution:

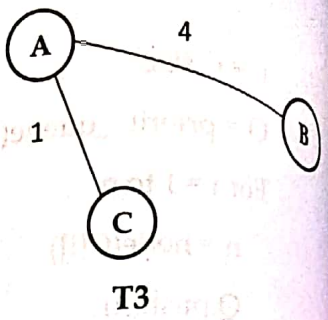
The possible spanning trees of given graph are:



T1



T2



T3

Total weight of T1 = 3

Total weight of T2 = 6

Total weight of T3 = 5

Since T1 has minimum weight out of all possible spanning trees of given graph hence tree T1 acts as minimum spanning tree of given graph.

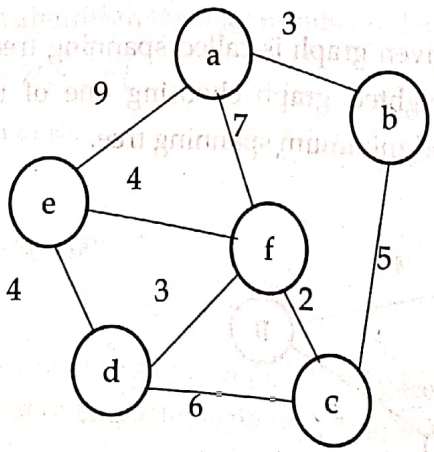
It is a minimum spanning tree construction method of given weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

The idea behind this algorithm is that we put the set of edges form the given graph G = (V, E) in non-decreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will be the minimum. Note that we have G as a graph, V as a set of n vertices and E as set of edges of graph G.

Steps for finding MST using Kruskal's Algorithm

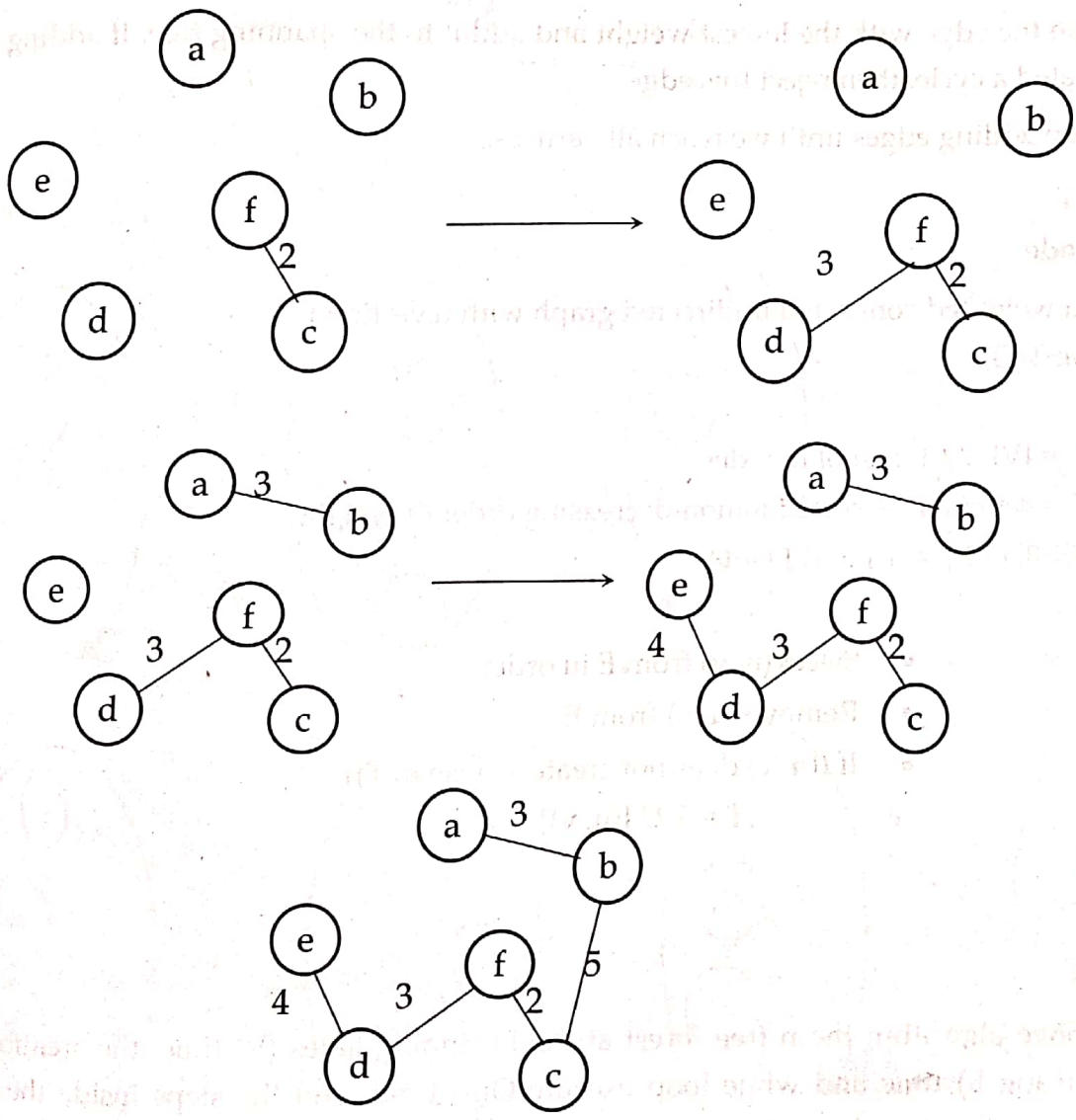
- Arrange the edge of G in order of increasing weight.
- Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.

Example: Find minimum spanning tree of given graph by using Kruskal's algorithm.



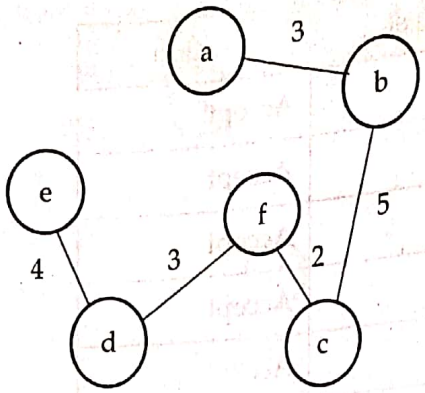
Solution: Edges in sorted order are given below:

Edges	Cost	Action
(f, c)	2	Accept
(f, d)	3	Accept
(a, b)	3	Accept
(d, e)	4	Accept
(b, c)	5	Accept
(d, c)	6	Reject
(a, f)	7	Reject
(a, e)	9	Reject



The edges (d, c), (a, f) and (a, e) forms cycle so discard these edges from the tree.

Thus the minimum spanning tree of weight 17 is;



Algorithm

1. Start
2. Sort all the edges from low weight to high
3. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
4. Keep adding edges until we reach all vertices.
5. Stop

Pseudo code

Let *G* be a weighted connected undirected graph with n vertices)

```
KruskalMST(G)
{
        T = {V}  // forest of n nodes
        E = set of edges sorted in non-decreasing order of weight
        while(|T| < n-1 and E! =Φ)
        {
                • Select (u, v) from E in order
                • Remove (u, v) from E
                • If ((u, v) does not create a cycle in T))
                        T = T U {(u, v)}

        }
}
```

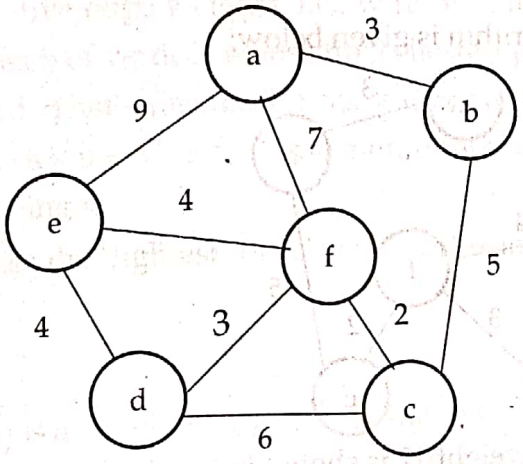
Analysis

In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(E log E) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(E log E) or asymptotically equivalently O(E log V)!.

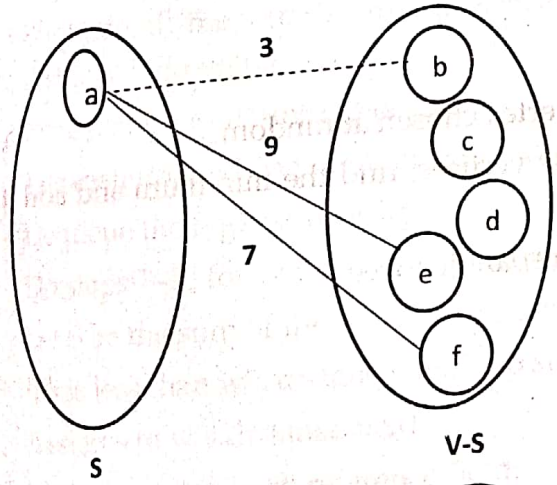
Prim's Algorithm

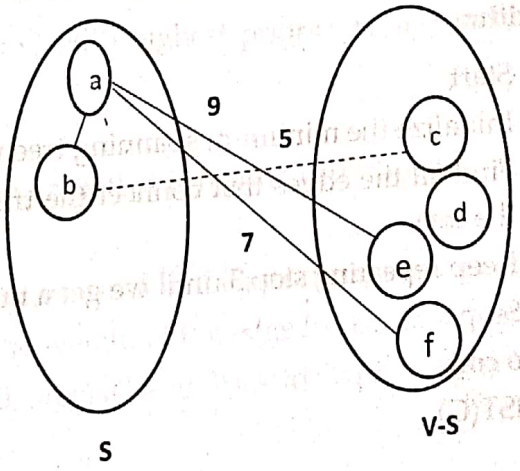
Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. Remember the cycle must be avoided.

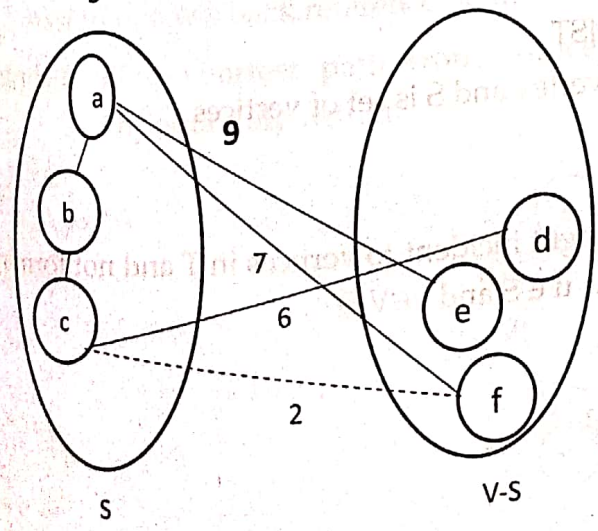
Example: find minimum spanning tree of given graph by using Prim's algorithm

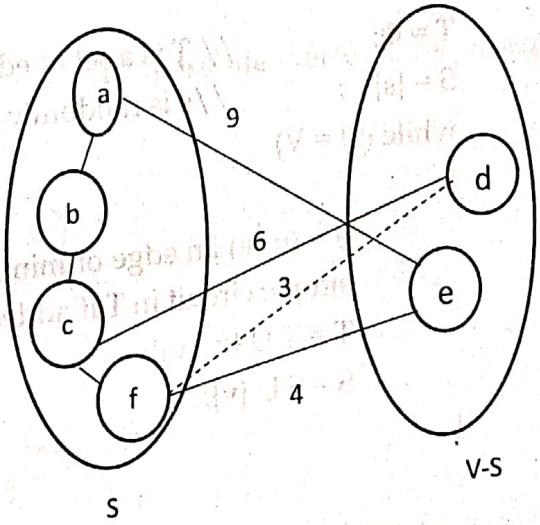


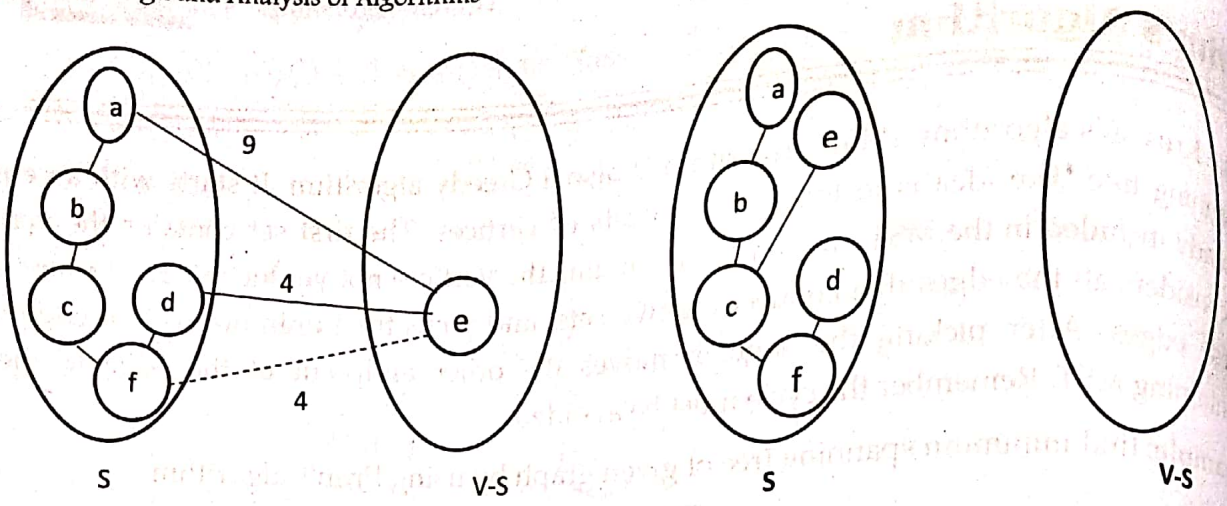
Solution:



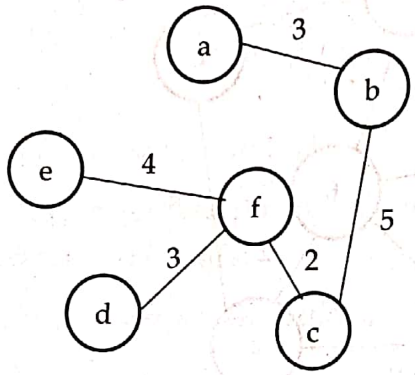








Thus final MST given by Prim's algorithm is given below;



Thus the minimum spanning tree of weight 17 is shown in fig above.

Algorithm

1. Start
2. Initialize the minimum spanning tree with a vertex chosen at random.
3. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
4. Keep repeating step 3 until we get a minimum spanning tree
5. Stop

Pseudo code

```
PrimMST(G)
{
    T = Φ;          // T is a set of edges of MST
    S = {s} ;       //s is randomly chosen vertex and S is set of vertices
    while (S! = V)
    {
        e = (u, v) an edge of minimum weight incident to vertices in T and not forming
        simple circuit in T if added to T i.e. u ∈ S and v∈V-S
        T = T U {(u, v)};
        S = S U {v};
    }
}
```

Analysis

In the above algorithm while loop execute O(V). The edge of minimum weight incident on a vertex can be found in O(E), so the total time is O(EV). We can improve the performance of the above algorithm by choosing better data structures as priority queue and normally it will be seen that the running time of prim's algorithm is O(E log V)!.

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E. Dijkstra's Algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d[v] = δ(s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest path estimate, inserts u into S and relaxes all edges leaving u.

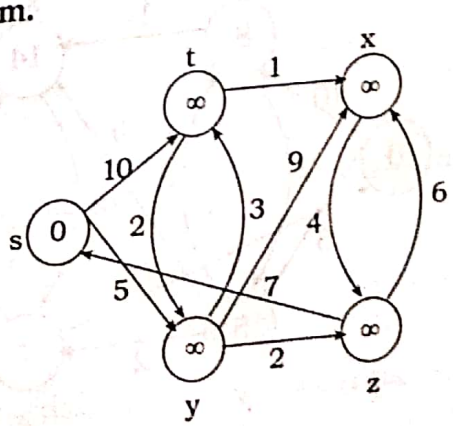
Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the greedy strategy.

Steps

Precondition: G = (V, w) is a weighted graph with initial vertex v_0 then it holds following steps:

1. Initialize the distance field to 0 for v0 and ∞ to for each of the other vertices.
2. Enqueue all the vertices into a priority queue Q with highest priority being the lowest distance field value.
3. Repeat steps 4–10 until Q is empty.
4. The distance and back reference fields of every vertex that is not in Q are correct
5. Dequeue the highest priority vertex into v.
6. Do steps 7–10 for each vertex w that are adjacent to v and in the priority queue.
7. Let S be the sum of the v's distance field plus the weight of the edge from v to w.
8. If s is less than w's distance field, do steps 9–10; otherwise go back to Step3.
9. Assign s to w's distance field.
10. Assign v to w's back reference field.

Example 1: Find shortest path from source vertex s to all possible vertices by using Dijkstra's algorithm.



Solution:

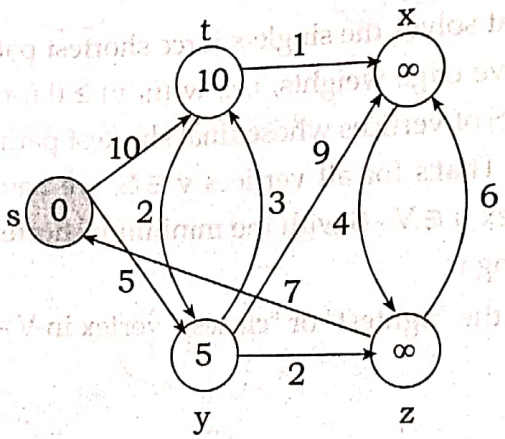
Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices. We make a stack, which contains those vertices which are selected after computation of shortest distance. Firstly we take's' in stack M (which is a source)

M = [S] Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

Adj [s] → t, y



Case i: s → t

d [v] > d [u] + w [u, v]

d [t] > d [s] + w [s, t]

∞ > 0 + 10 [false condition]

Then d [t] ← 10

Case ii: s→ y

d [v] > d [u] + w [u, v]

d [y] > d [s] + w [s, y]

∞ > 0 + 5 [false condition]

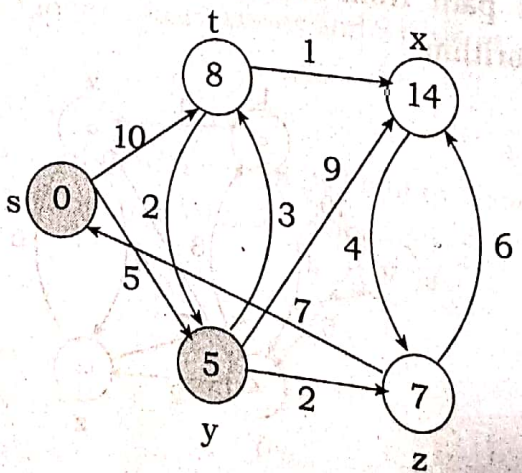
∞ > 5

Then d [y] ← 5

By comparing case (i) and case (ii)

Adj [s] → t = 10, y = 5

y is shortest



Step 3: Now find the adjacent of y that is t, x, z.

Case i: y →t

d [v] > d [u] + w [u, v]

d [t] > d [y] + w [y, t]

10 > 5 + 3

10 > 8

Then d [t] ← 8

Case ii: y → x

d [v] > d [u] + w [u, v]

d [x] > d [y] + w [y, x]

∞ > 5 + 9

∞ > 14

Then d [x] ← 14

Case iii: y → z

d [v] > d [u] + w [u, v]

d [z] > d [y] + w [y, z]

∞ > 5 + 2

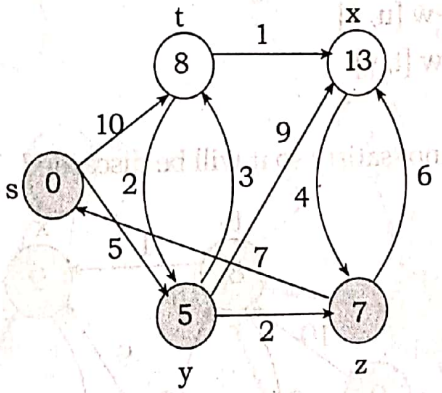
∞ > 7

Then d [z] ← 7

By comparing case (i), case (ii) and case (iii)

Adj [y] → x = 14, t = 8, z =7

z is shortest



Step 4: Now we will find adj [z] that are s, x

Case i: z → x

d [v] > d [u] + w [u, v]

d [x] > d [z] + w [z, x]

14 > 7 + 6

14 > 13

Then d [x] ← 13

Case ii: z → s

d [v] > d [u] + w [u, v]

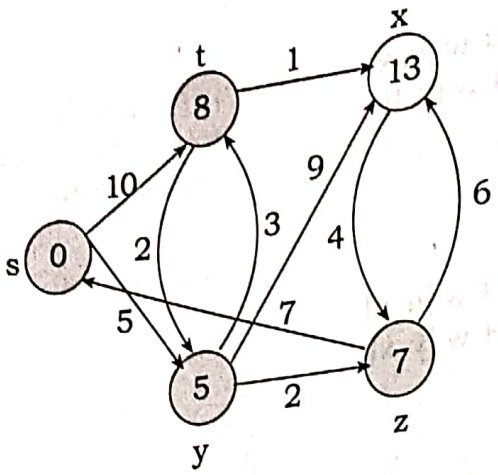
d [s] > d [z] + w [z, s]

0 > 7 + 7

0 > 14

∴ This condition does not satisfy so it will be discarded.

Now we have x = 13.



Step 5: Now we will find Adj [t]

Case i: t → x

d [v] > d [u] + w [u, v]

d [x] > d [t] + w [t, x]

13 > 8 + 1

13 > 9

Then d [x] ← 9

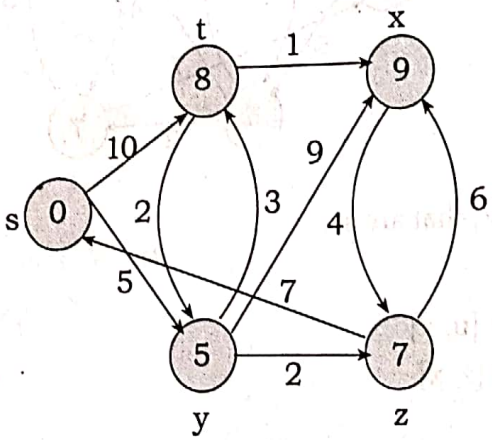
Case ii: t → y

d [v] > d [u] + w [u, v]

d [y] > d [t] + w [t, y]

5 > 10

∴ This condition does not satisfy so it will be discarded.



Thus we get all shortest path vertex as

Weight from s to y is 5

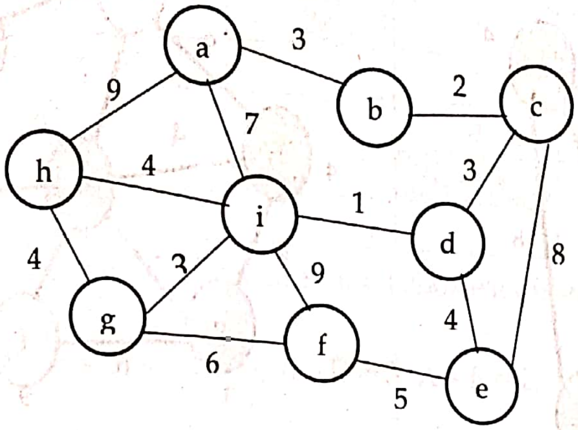
Weight from s to z is 7

Weight from s to t is 8

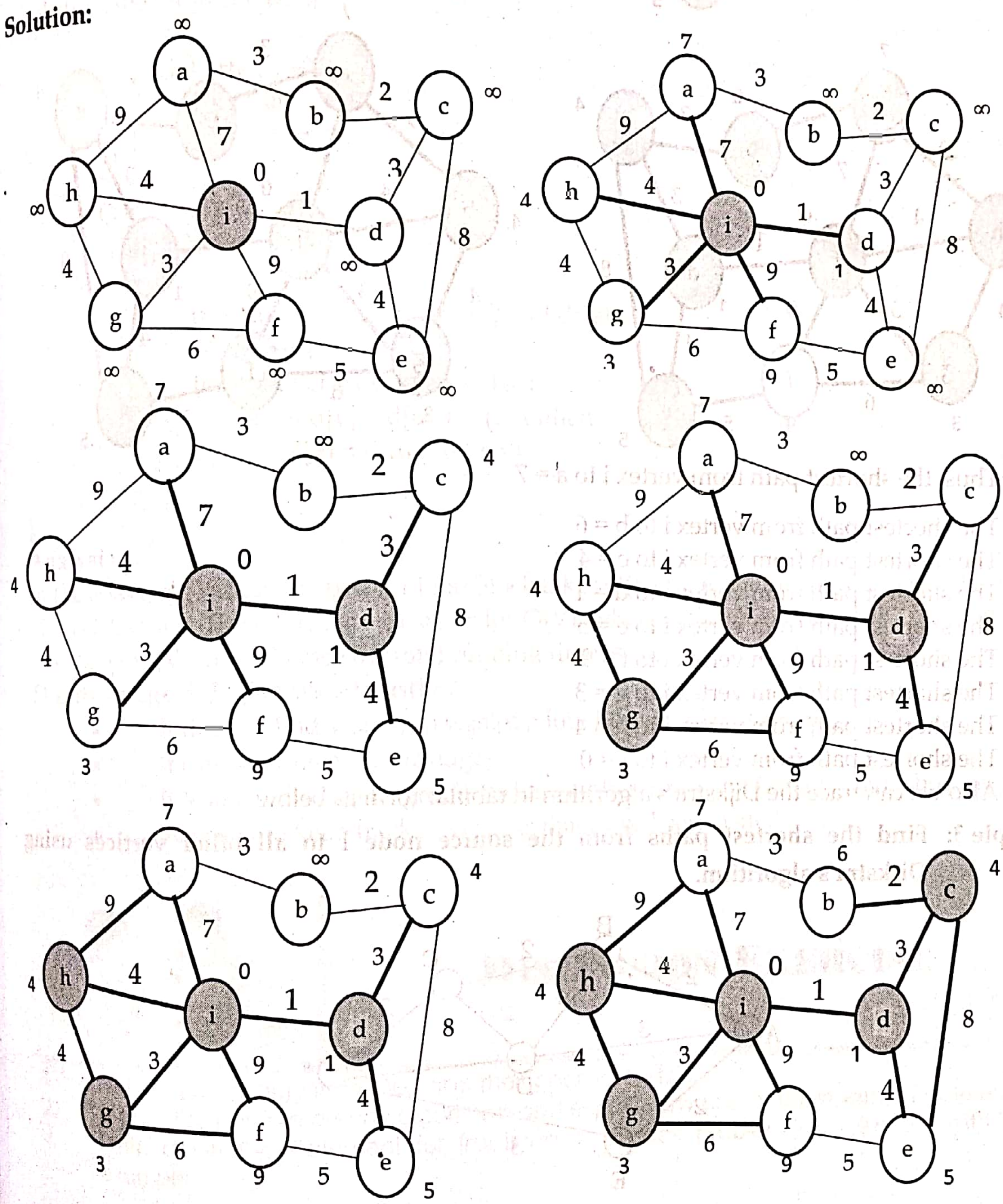
Weight from s to x is 9

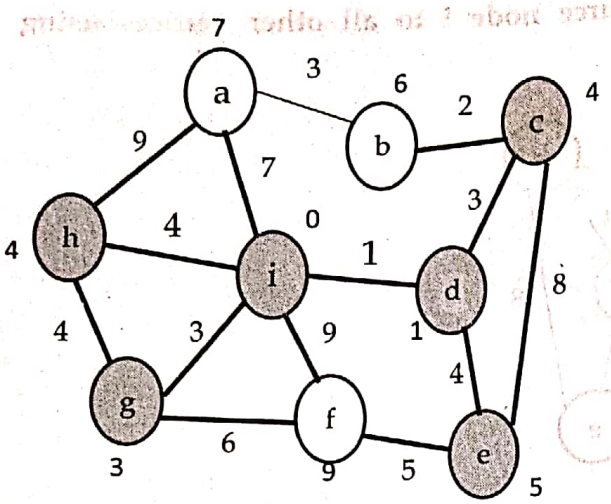
These are the shortest distance from the source's' in the given graph.

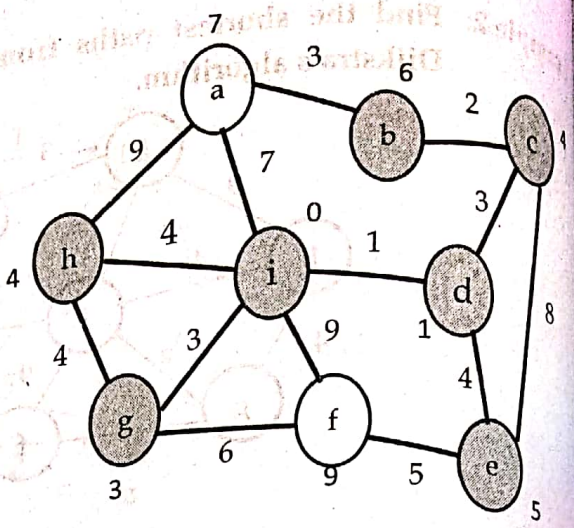
Example 2: **Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.**

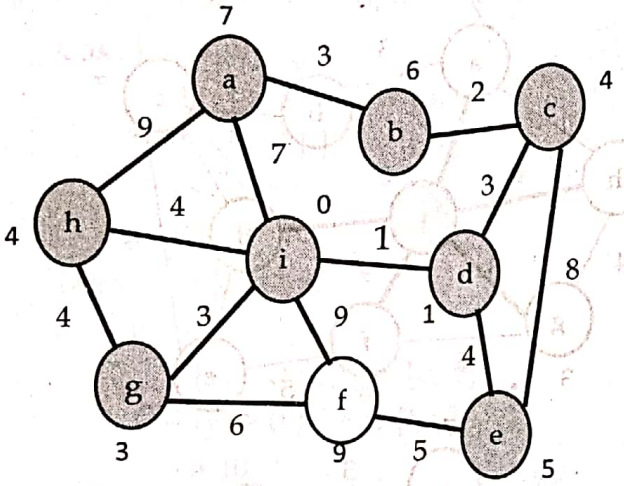


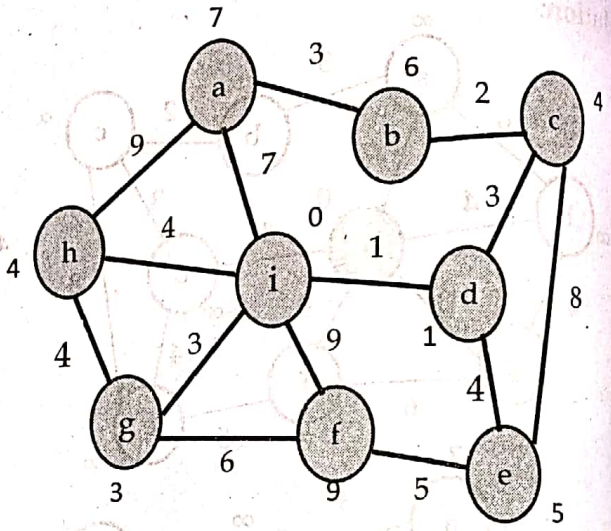
Solution:







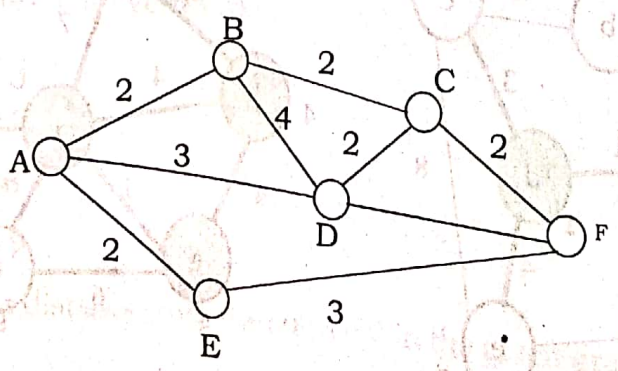




Thus, the shortest path from vertex i to a = 7

The shortest path from vertex i to b = 6
The shortest path from vertex i to c = 4
The shortest path from vertex i to d = 1
The shortest path from vertex i to e = 5
The shortest path from vertex i to f = 9
The shortest path from vertex i to g = 3
The shortest path from vertex i to h = 4
The shortest path from vertex i to I = 0
Also we can trace the Dijkstra's algorithm in tabular form as below,

Example 3: Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.



Marked	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B		2	∞	3	2	∞
E			4	3	2	∞
D			4	3		5
C			4			5
						5

In the above table marked cell are called shortest distances from source vertex A

Thus shortest path from

A – A = 0, A – B = 2, A – C = 4, A – D = 3, A – E = 2, A – F = 5

Algorithm

```
Dijkstra_Algorithm(G, w, s)
{
        for each vertex v ∈V
                d[v] = Φ
        d[s] = 0
        S = Φ
        Q = V
        while (Q!= Φ)
        {
                u = Take minimum from Q and delete.
                S = S U {u}
                for each vertex v adjacent to u
                        if d[v] > d[u] + w(u, v)then
                        d[v] = d[u] + w(u, v)

        }
}
```

Analysis

In the above algorithm, the first for loop block takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times. Hence the total running time is $O(V^2)$.

Disadvantage of Dijkstra's Algorithm

- It does a blind search, so wastes a lot of time while processing.
- It can't handle negative edges.
- It leads to the acyclic graph and most often cannot obtain the right shortest path.
- We need to keep track of vertices that have been visited.

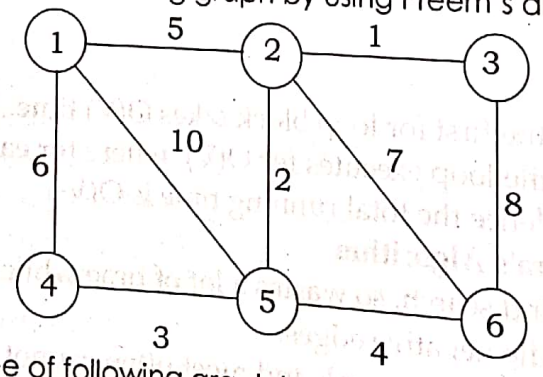
DISCUSSION EXERCISE

1. Define greedy algorithm. Describe their characteristics.
2. List out the possible greedy algorithms and explain any one of them with suitable example.
3. Write down the statement for fractional knapsack problem and explain with suitable example.

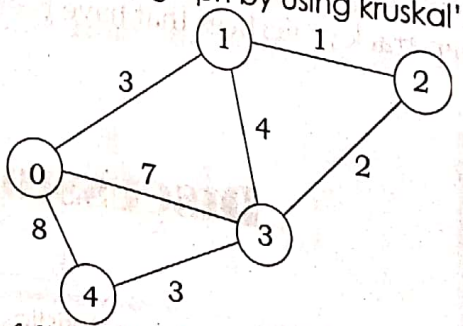
4. State the Greedy Knapsack? Find an optimal solution to the Knapsack instance n = 3, m = 20, {P1, P2, P3) = (25, 24, 15) and (W1, W2, W3) = (18, 15, 10).
5. let's assume that there are 4 jobs
   n = 4
   S = (p1, p2, p3, p4)
   D = (d1,d2,d3,d4)=(2, 1, 3, 4)
   P = (100,10,15,27)
   Find the sequence due to which maximize the profit by using job sequencing with deadlines.
6. Define prefix code. How Huffman algorithm works? Explain with suitable example.
7. What is minimum spanning tree? Write down the working principles of kruskal's method for finding minimum spanning tree of given graph.
8. Differentiate between Preem's algorithm and kruskal's algorithm for finding minimum spanning tree of given graph.
9. Write down the algorithm for Preem's algorithm then analyze it.
10. What is shortest path problem? Explain Dijkstra algorithm with suitable example.
11. How Kruskal's algorithm for finding MST is called greedy algorithm? Explain.
12. Find the shortest paths from the source node i to all other vertices of a weighted graph containing at least 7 vertices and 10 edges by using Dijkstra's algorithm
13. Does Prim's and Kruskal's algorithm work if negative weights are allowed? Explain
14. Trace the Huffman algorithm for following data items and their frequencies

Character	a	b	c	d	e	f
Frequencies	11	23	13	7	87	5

15. Find the optimal schedule for the following jobs with n = 7 profits $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 18, 20, 6, 1, 38)$ and dead lines $(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 3, 4, 1, 2, 1)$.
16. Consider the knapsack instance n = 3 with weight $(w_1, w_2, w_3) = (2, 3, 4)$ and profit $(p_1, p_2, p_3) = (2, 3, 5)$ with total knapsack capacity W=5. Then find optimal solution by using fractional knapsack problem.
17. What are the properties of greedy algorithm? Explain with example.
18. Find minimum spanning tree of following graph by using Preem's algorithm



19. Find minimum spanning tree of following graph by using kruskal's algorithm



20. Find the optimal schedule for the following jobs with n = 5 profits $(p_1, p_2, p_3, p_4, p_5) = (3, 5, 18, 20, 38)$ and dead lines $(d_1, d_2, d_3, d_4, d_5) = (3, 4, 1, 2, 1)$.

❑❑❑