GREEDY TECHNIQUE

Definition:

Greedy technique is a general algorithm design strategy, built on following elements:

- **configurations**: different choices, values to find
- **objective function**: some configurations to be either maximized or minimized

The method:

- Applicable to **optimization problems** ONLY
- Constructs a solution through a sequence of steps
- Each step expands a partially constructed solution so far, until a complete solution to the problem is reached.

  On each step, the choice made must be
- **Feasible:** it has to satisfy the problem's constraints
- **Locally optimal:** it has to be the best local choice among all feasible choices available on that step
- **Irrevocable:** Once made, it cannot be changed on subsequent steps of the algorithm

NOTE:

- Greedy method works best when applied to problems with the **greedy-choice** property
- A globally-optimal solution can always be found by a series of local improvements from a starting configuration.

Greedy method vs. Dynamic programming method:

- **LIKE** dynamic programming, greedy method **solves optimization problems.**
- **LIKE** dynamic programming, greedy method **problems exhibit optimal substructure**
- **UNLIKE** dynamic programming, greedy method problems exhibit the **greedy choice** property -avoids back-tracing.

Applications of the Greedy Strategy:

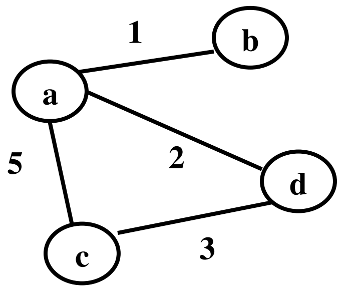
- **Optimal solutions:**
  - Change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- **Approximations:**
  - Traveling Salesman Problem (TSP)
  - Fractional Knapsack problem

Spanning Tree

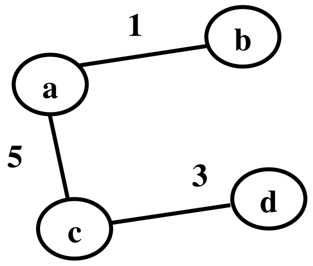
Definition:

Spanning tree is a connected acyclic sub-graph (tree) of the given graph (*G*) that includes all of *G*'s vertices

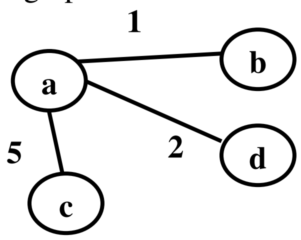
Example: Consider the following graph



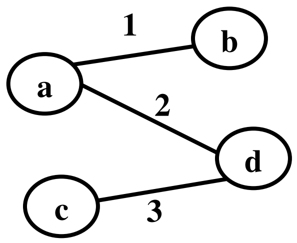
The spanning trees for the above graph are as follows:



Weight (T_1) = 9



Weight (T_2) = 8



Weight (T_3) = 6

Minimum Spanning Tree (MST)

Definition:

MST of a weighted, connected graph *G* is defined as: A spanning tree of *G* with **minimum total weight.**

Example: Consider the example of spanning tree:
For the given graph there are three possible spanning trees. Among them the spanning tree with the minimum weight 6 is the MST for the given graph

Question: Why can't we use **BRUTE FORCE** method in constructing MST ?
Answer: If we use Brute force method-

- Exhaustive search approach has to be applied.
- Two serious obstacles faced:
    1. **The number of spanning trees grows exponentially with graph size.**
    2. **Generating all spanning trees for the given graph is not easy.**

MST Applications:

- **Network design.**
  Telephone, electrical, hydraulic, TV cable, computer, road
- **Approximation algorithms for NP-hard problems.**
  Traveling salesperson problem, Steiner tree
- Cluster analysis.
- Reducing data storage in sequencing amino acids in a protein
- Learning salient features for real-time face verification
- Auto config protocol for Ethernet bridging to avoid cycles in a network, etc

Prim's Algorithm

-to find minimum spanning tree

Some useful definitions:

- **Fringe edge:** An edge which has one vertex is in partially constructed tree Ti and the other is not.
- **Unseen edge:** An edge with both vertices not in Ti

Algorithm:

```
ALGORITHM Prim (G)
//Prim's algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G

// the set of tree vertices can be initialized with any vertex

$V_T \leftarrow \{ v_0 \}$
$E_T \leftarrow \emptyset$
for i ← 1 to |V| - 1 do
    Find a minimum-weight edge e* = (v*, u*) among all the edges (v, u) such
    that v is in $V_T$ and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{ u^* \}$
    $E_T \leftarrow E_T \cup \{ e^* \}$
return $E_T$
```

The method:

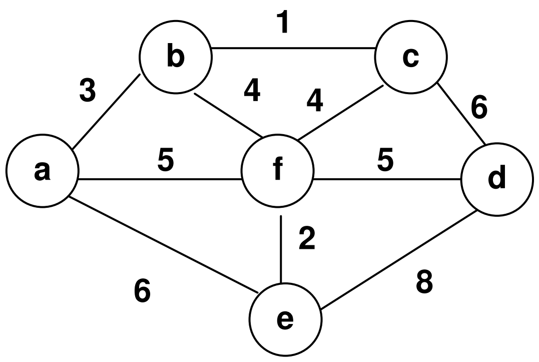
STEP 1: Start with a tree, T_0, consisting of one vertex
STEP 2: "Grow" tree one vertex/edge at a time

- Construct a series of expanding sub-trees T_1, T_2, … T_{n-1}.
- At each stage construct T_{i+1} from T_i by adding the minimum weight edge connecting a vertex in tree (Ti) to one vertex not yet in tree**, choose from "fringe" edges (this is the "greedy" step!)**

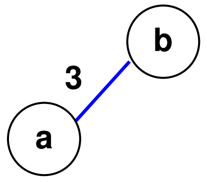
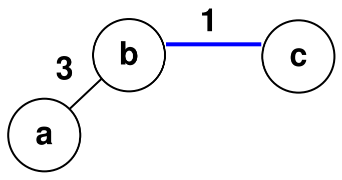
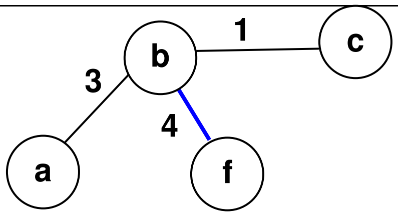
Algorithm stops when all vertices are included

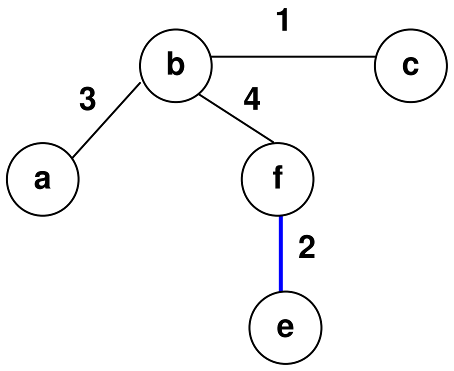
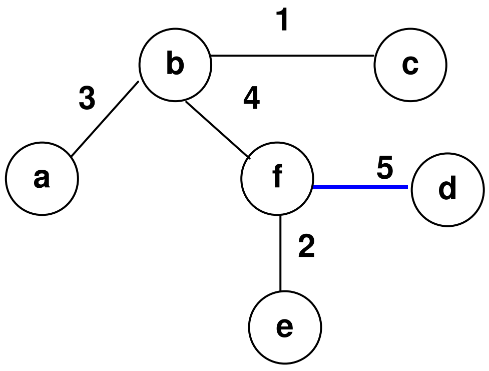
Example:

Apply Prim's algorithm for the following graph to find MST.



Solution:

Tree vertices	**Remaining vertices**	**Graph**
a (-, -)	b (a , 3) c (- , ∞) d (- , ∞) e (a , 6) f (a , 5)	
b (a, 3)	c (b , 1) d (- , ∞) e (a , 6) f (b , 4)	
c (b, 1)	d (c , 6) e (a , 6) f (b , 4)	

f (b, 4)	d (f , 5) e (f , 2)	
e (f, 2)	d (f , 5)	
d(f, 5)		**Algorithm stops since all vertices are included.** The weight of the minimum spanning tree is **15**

Efficiency:

Efficiency of Prim's algorithm is based on data structure used to store priority queue.

- Unordered array: **Efficiency: Θ(n^2)**
- Binary heap: **Efficiency: Θ(m log n)**
- Min-heap: For graph with n nodes and m edges: **Efficiency: (n + m) log n**

Conclusion:

- Prim's algorithm is a "vertex based algorithm"
- Prim's algorithm "Needs priority queue for locating the nearest vertex." The choice of priority queue matters in Prim implementation.
    - Array - optimal for dense graphs
    - Binary heap - better for sparse graphs
    - Fibonacci heap - best in theory, but not in practice

Kruskal’s Algorithm
-to find minimum spanning tree

Algorithm:

ALGORITHM Kruskal (G)

```
//Kruskal’s algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G

Sort E in ascending order of the edge weights

// initialize the set of tree edges and its size
$E_T \leftarrow Ø$
edge_counter $\leftarrow$ 0

//initialize the number of processed edges
k $\leftarrow$ 0

while edge_counter < |V| - 1
        k $\leftarrow$ k + 1
        if $E_T$ U { $e_{i k}$} is acyclic
                $E_T \leftarrow E_T$ U { $e_{i k}$ }
                edge_counter $\leftarrow$ edge_counter + 1
return $E_T$
```

The method:

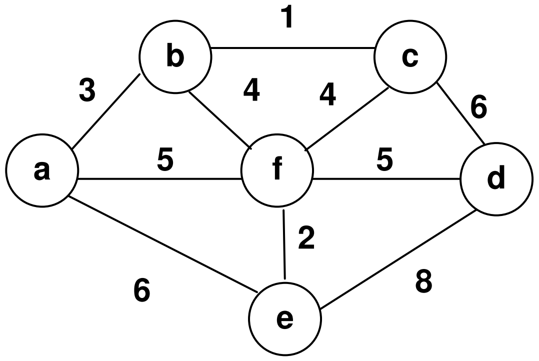
STEP 1: Sort the edges by increasing weight
STEP 2: Start with a forest having |V| number of trees.
STEP 3: Number of trees are reduced by ONE at every inclusion of an edge
At each stage:

- Among the edges which are not yet included, select the one with minimum weight AND which does not form a cycle.
- the edge will reduce the number of trees by one by combining two trees of the forest

Algorithm stops when |V| -1 edges are included in the MST i.e : when the number of trees in the forest is reduced to ONE.

Example:

Apply Kruskal's algorithm for the following graph to find MST.



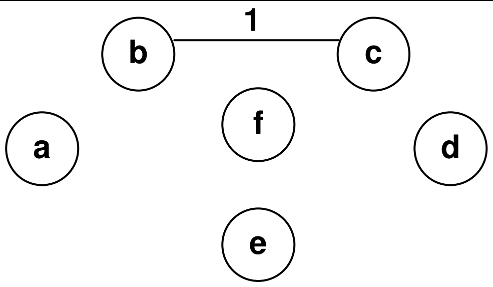
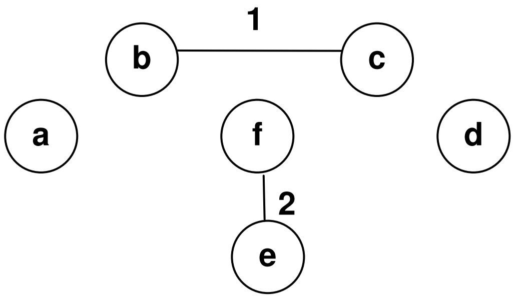
Solution:

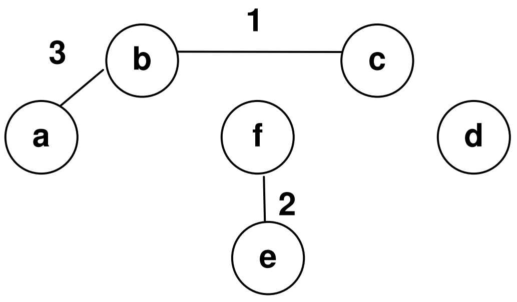
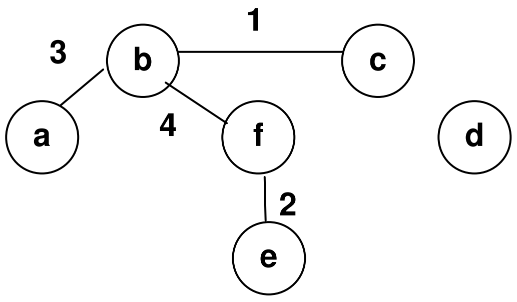
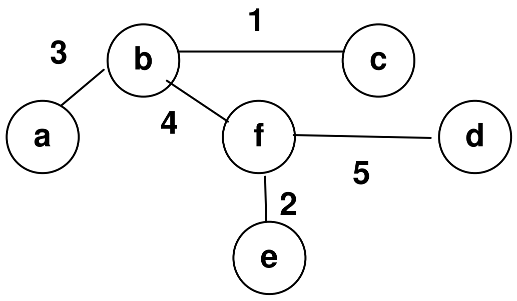
The list of edges is:

Edge	ab	af	ae	bc	bf	cf	cd	df	de	ef
Weight	3	5	6	1	4	4	6	5	8	2

Sort the edges in ascending order:

Edge	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

Edge	bc	
Weight	1	
Insertion status	YES	
Insertion order	1	
Edge	ef	
Weight	2	
Insertion status	YES	
Insertion order	2	

Edge	ab	
Weight	3	
Insertion status	YES	
Insertion order	3	
Edge	bf	
Weight	4	
Insertion status	YES	
Insertion order	4	
Edge	cf	
Weight	4	
Insertion status	**NO**	
Insertion order	-	
Edge	af	
Weight	5	
Insertion status	**NO**	
Insertion order	-	
Edge	df	
Weight	5	
Insertion status	YES	
Insertion order	5	
Algorithm stops as |V| -1 edges are included in the MST		

Efficiency:

Efficiency of Kruskal's algorithm is based on the time needed for sorting the edge weights of a given graph.

- With an efficient sorting algorithm: **Efficiency: $\Theta(|E| \log |E|)$**

Conclusion:

- Kruskal's algorithm is an "edge based algorithm"
- Prim's algorithm with a heap is faster than Kruskal's algorithm.

Dijkstra's Algorithm

- to find Single Source Shortest Paths

Some useful definitions:

- **Shortest Path Problem:** Given a connected directed graph G with non-negative weights on the edges and a root vertex r, find for each vertex x, a directed path P (x) from r to x so that the sum of the weights on the edges in the path P (x) is as small as possible.

Algorithm

- By Dutch computer scientist Edsger Dijkstra in 1959.
- Solves the single-source shortest path problem for a graph with nonnegative edge weights.
- This algorithm is often used in routing.

  E.g.: Dijkstra's algorithm is usually the working principle behind link-state routing protocols

```
ALGORITHM Dijkstra(G, s)
//Input: Weighted connected graph G  and source vertex s
//Output: The length Dv of a shortest path from s to v and its penultimate vertex Pv for
every vertex v in V

//initialize vertex priority in the priority queue
Initialize (Q)

for every vertex v in V do
        $D_v \leftarrow \infty$ ; $P_v \leftarrow$ null    // Pv , the parent of v
        insert(Q, v, $D_v$) //initialize vertex priority in priority queue
$d_s \leftarrow 0$

//update priority of s with ds, making ds, the minimum
Decrease(Q, s, $d_s$)

$V_T \leftarrow \varnothing$
```

```
for i ← 0 to |V| - 1 do
    u* ← DeleteMin(Q)
    //expanding the tree, choosing the locally best vertex
    $V_T$ ← $V_T$ U {u*}
    for every vertex u in V – $V_T$ that is adjacent to u* do
        if Du* + w (u*, u) < Du
            Du ← Du + w (u*, u); Pu ← u*
            Decrease(Q, u, Du)
```

The method

Dijkstra's algorithm solves the single source shortest path problem in 2 stages.

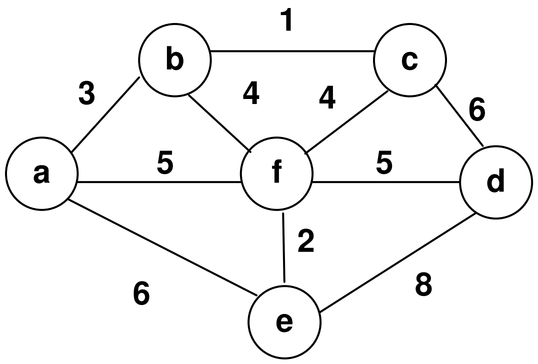
Stage 1: A greedy algorithm computes the shortest distance from **source to all other nodes** in the graph and saves in a data structure.

Stage 2 : Uses the data structure for finding a **shortest path from source to any vertex v.**

- **At each step, and for each vertex x, keep track of a "distance" D(x) and a directed path P(x) from root to vertex x of length D(x).**
- **Scan first from the root and take initial paths P(r, x) = (r, x) with**
  D(x) = w(rx) when rx is an edge,
  D(x) = ∞ when rx is not an edge.
  For each temporary vertex y distinct from x, set
  D(y) = min{ D(y), D(x) + w(xy) }

Example:

Apply Dijkstra's algorithm to find Single source shortest paths with vertex **a** as the source.

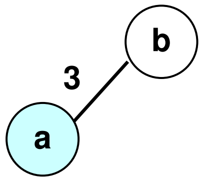


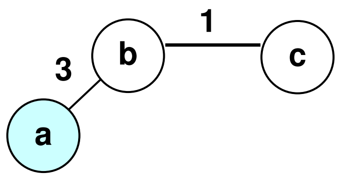
Solution:

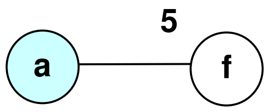
Length Dv of shortest path from source (s) to other vertices v and Penultimate vertex Pv for every vertex v in V:

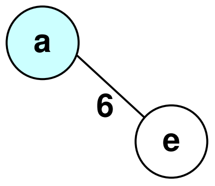
Da = 0 , Pa = null
Db = ∞ , Pb = null
Dc = ∞ , Pc = null
Dd = ∞ , Pd = null
De = ∞ , Pe = null
Df = ∞ , Pf = null

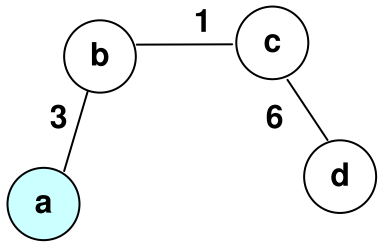
Tree vertices	**Remaining vertices**	**Distance & Path vertex**	**Graph**

a (-, 0)	b (a , 3)
	c (- , ∞)
	d (- , ∞)
	e (a , 6)
	f (a , 5)
		Da = 0 Pa = a
		Db = 3 Pb = [a, b]
		Dc = ∞ Pc = null
		Dd = ∞ Pd = null
		De = 6 Pe = [a, e]
		Df = 5 Pf = [a, f]
			

b (a, 3)	c (b , 3+1)
	d (- , ∞)
	e (a , 6)
	f (a , 5)
		Da = 0 Pa = a
		Db = 3 Pb = [a, b]
		Dc = 4 Pc = [a,b,c]
		Dd = ∞ Pd = null
		De = 6 Pe = [a, e]
		Df = 5 Pf = [a, f]
			

c (b, 4)	d (c , 4+6)
	e (a , 6)
	f (a , 5)
		Da = 0 Pa = a
		Db = 3 Pb = [a, b]
		Dc = 4 Pc = [a,b,c]
		Dd=10 Pd = [a,b,c,d]
		De = 6 Pe = [a, e]
		Df = 5 Pf = [a, f]
			

f (a, 5)	d (c , 10)
	e (a , 6)
		Da = 0 Pa = a
		Db = 3 Pb = [a, b]
		Dc = 4 Pc = [a,b,c]
		Dd=10 Pd = [a,b,c,d]
		De = 6 Pe = [a, e]
		Df = 5 Pf = [a, f]
			

e (a, 6)	d (c, 10)	Da = 0 Pa = a
		Db = 3 Pb = [a, b]
		Dc = 4 Pc = [a,b,c]
		Dd=10 Pd = [a,b,c,d]
		De = 6 Pe = [a, e]
		Df = 5 Pf = [a, f]
			

d(c, 10)			**Algorithm stops since no edges to scan**

Conclusion:

- Doesn't work with negative weights
- Applicable to both undirected and directed graphs
- Use unordered array to store the priority queue: **Efficiency = $\Theta(n^2)$**
- Use min-heap to store the priority queue: **Efficiency = O(m log n)**

Huffman Trees

Some useful definitions:

- **Code word:** Encoding a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bits. This bits sequence is called code word
- **Fixed length encoding:** Assigns to each character a bit string of the same length.
- **Variable length encoding:** Assigns code words of different lengths to different characters.
  Problem:
  How can we tell how many bits of an encoded text represent ith character?
  We can use prefix free codes
- **Prefix free code:** In Prefix free code, no codeword is a prefix of a codeword of another character.
- **Binary prefix code :**
  - The characters are associated with the leaves of a binary tree.
  - All left edges are labeled 0
  - All right edges are labeled 1
  - Codeword of a character is obtained by recording the labels on the simple path from the root to the character's leaf.
  - Since, there is **no simple path to a leaf that continues to another leaf**, no codeword can be a prefix of another codeword

Huffman algorithm:

- Constructs binary prefix code tree
- By **David A Huffman** in 1951.
- Huffman's algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed. Huffman coding uses frequencies of the symbols in the string to build a variable rate prefix code
  - Each symbol is mapped to a binary string
  - More frequent symbols have shorter codes
  - No code is a prefix of another code
- Huffman Codes for Data Compression achieves 20-90% Compression

Construction:

Step 1: Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's weight. (More generally the weight of a tree will be equal to the sum of the frequencies in the tree's leaves)

Step 2: Repeat the following operation until a single tree is obtained.
"Find two trees with smallest weight. Make them the left and right sub-tree of a new tree and record the sum of their weights in the root of the new tree as its weight"

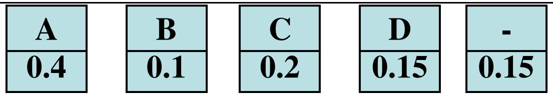
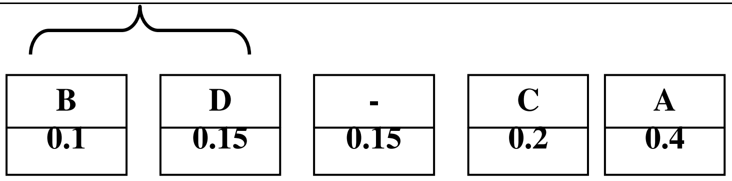
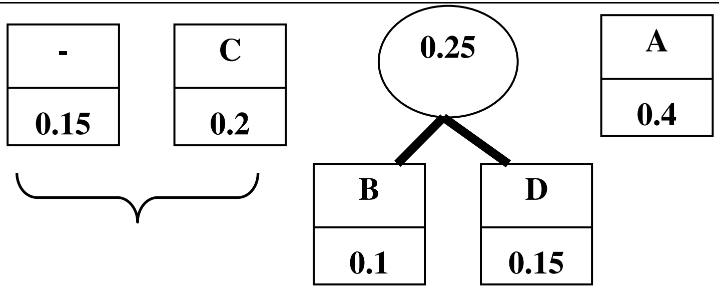
Example:

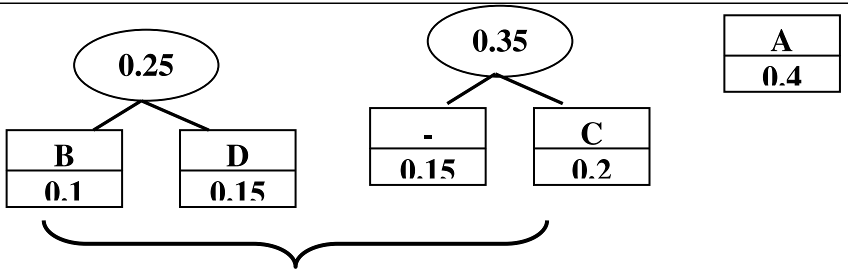
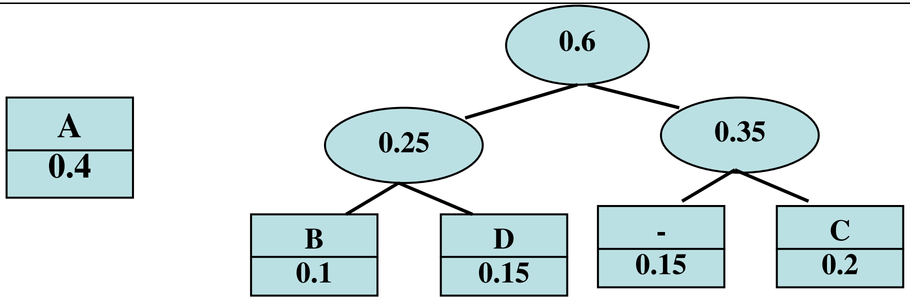
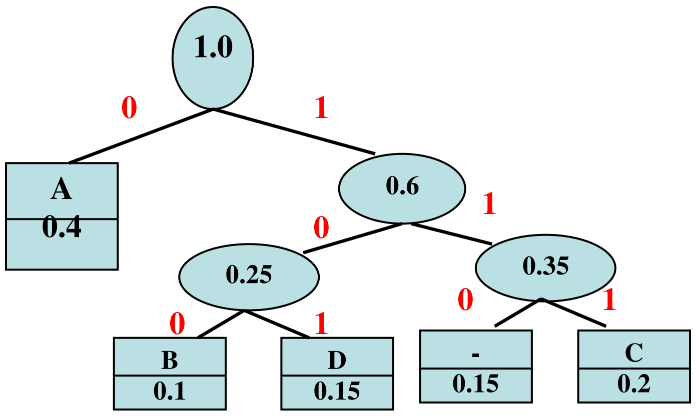
Construct a Huffman code for the following data:

Character	**A**	**B**	**C**	**D**	**-**
probability	0.4	0.1	0.2	0.15	0.15

- Encode the text **ABACABAD** using the code.
- Decode the text whose encoding is **100010111001010**

Solution:

1	
2	
3	

4	
5	 A 0.4 0.6 0.25 0.35 B 0.1 D 0.15 - 0.15 C 0.2
6	 1.0 0 1 A 0.4 0.6 0 1 0.25 0.35 0 1 0 1 B 0.1 D 0.15 - 0.15 C 0.2
Algorithm STOPS as SINGLE TREE is obtained	

Code words are:

Character	**A**	**B**	**C**	**D**	**-**
probability	0.4	0.1	0.2	0.15	0.15
Code word	0	100	111	101	110

Encoded text for ABACABAD using the code words: **0100011101000110**
Decoded text for encoded text 100010111001010 is: **BAD-ADA**
Compute compression ratio:
Bits per character = Codeword length * Frequency
= (1 * 0.4) + (3 * 0.1) + (3 * 0.2) + (3 * 0.15) + (3 * 0.15)
= 2.20
Compression ratio is = (3 – 2.20)/ 3 . 100% = 26.6%