unit- 9

GREEDY TECHNIQUE (9)

This technique is used for designing optimiza-tion problems. Greedy strategy always tries to find the best solution for each sub problem with the hope that this will results in the best solution for a whole problem. That is, greedy approach suggests to construct a solution for a problem through a sequence of steps, each step expanding a partially obtained solution so far, until a complete solution is reached. It is thus important to note that we have to make a choice in each step such that the choice must be-

* feasible - it has to satisfy the constraints of the problem
* locally optimal - it has to be the best local choice among all feasible solutions available at that step.
* irrevocable - Once a choice is made, it can't be changed on subsequent steps of the algorithm.

Chetana Hegde
9448301894

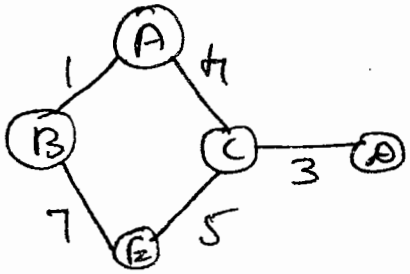
As in every step we look at optimal solution, greedy technique yields best solutions for many of the problems. But, the drawback here is, without aiming at final solution, we will think about the step where we are at. This is just like grabing whatever is available right now without thinking about the future. Thus, for some problems, if we use greedy, then we have satisfy ourselves with only approximate solution.

Consider an example of greedy:
Suppose we have a set of coins as— {1, 1, 1, 1, 1, 5, 5, 5, 10, 10, 25, 25}. We have to give change for 43 paise. Then we will first take the heighest possible value which is less than or equal to 43. i.e 25. For the next choice, we take next-heghest- ie 10. Continuing in this way we get the solution as {25, 10, 5, 1, 1, 1}
Note that, we won't start with 1 here.
The technique used here is nothing but-greedy.

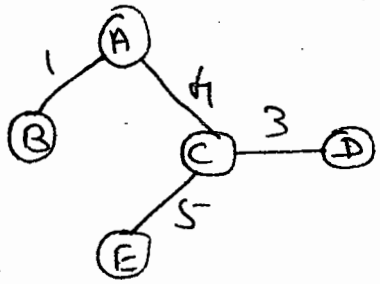
NOTE: Greedy technique is applied in finding minimum spanning tree using the algorithms like Prim's and Kruskal's algorithms.

Def: A connected acyclic subgraph containing all the vertices of a connected graph is known as spanning tree. The minimum spanning tree is defined for weighted connected graph and is the spanning subtree with minimum weight.

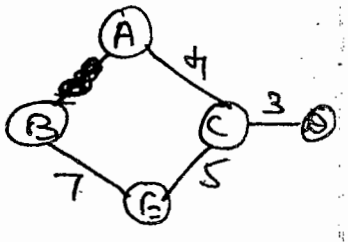
Consider a graph and its spanning trees -



Graph



$W(T_1) = 13$



$W(T_2) = 19$

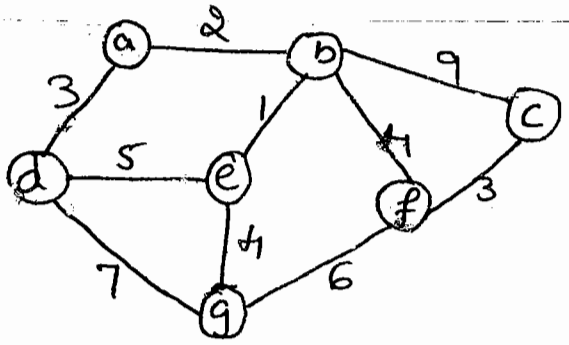
Here, T_1 will be the minimum spanning tree as the weight is less compared to that of T_2. Here, we have used the exhaustive-search technique. That is, first finding all possible spanning trees and then to select the minimum among all these. This method will be time consuming.

Prim's Algorithm :-

This algorithm constructs minimum spanning tree through a sequence of expanding subtrees. The initial subtree is taken as any arbitrary vertex. Then among the other vertices of a graph, the nearest vertex to the existing vertex of the tree is found and is attached to the tree. The procedure is continued till all the vertices are included into the tree. Thus, if V is the set of all vertices of a graph and V_T is that of minimum spanning tree then we have do the following operations -

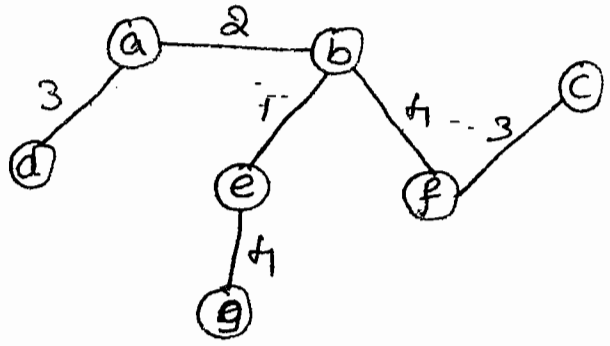
1. Move u^* from the set $V - V_T$ to V_T.
2. For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u respectively.

For illustration, consider a connected weighted graph given below —



Tree vertices	Remaining vertices
a(–, –)	b(a, 2), d(a, 3), c(–, ∞), e(–, ∞), f(–, ∞), g(–, ∞)
b(a, 2)	d(a, 3), e(b, 1), c(b, 9), f(b, 4), g(–, ∞)
e(b, 1)	d(a, 3), c(b, 9), f(b, 4), g(e, 4)
d(a, 3)	c(b, 9), f(b, 4), g(e, 4)
f(b, 4)	c(b, 9), g(e, 4), c(f, 3)
c(f, 3)	g(e, 4)
g(e, 4).	

Thus the minimum spanning tree is –



chetana hegde
9448301894

ALGORITHM Prim(G)

// Prim's algorithm to construct min. spa. tree.

// Input: A connected weighted graph $G(V, E)$.

// Output: E_T, the set of edges composing

// minimum spanning tree of G.

$V_T \leftarrow \{v_0\}$

$E_T \leftarrow \phi$

for $i \leftarrow 1$ to $|V|-1$ do

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that $v \in V_T$ and $u \in V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T.

Analysis:

Note that, in every step of the algorithm, the set of remaining vertices with distances constitute a priority queue. Each time a choice is made, the queue must be updated. Thus, the efficiency of the Prim's algorithm depends on the data structure

selected for the graph and for the priority queue. If we represent a graph by its weight matrix and the priority queue by unordered array. Then the time complexity is of order $\Theta(|V|^2)$.

On the other hand, we can implement priority queue as min-heap, whose efficiency for insertion and deletion is of $O(\log n)$, where n is size of heap. Suppose that the graph is implemented using adjacency linked list. In this case, as the algorithm performs $|V|-1$ deletions and $|E|$ verifications in the time $O(\log n)$, we have

complexity $= (|V|-1+|E|) \cdot O(\log |V|)$

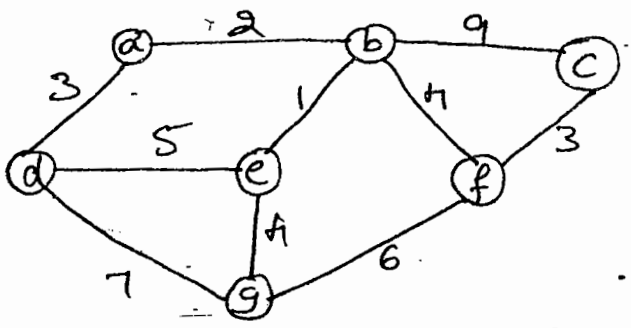
As $|V|-1 \leq |E|$ for connected graph, we have

$C \in O(|E| \cdot \log(|V|))$.

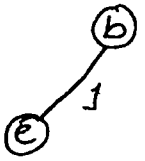
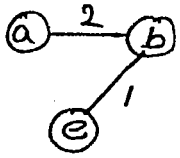
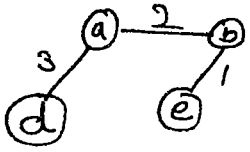
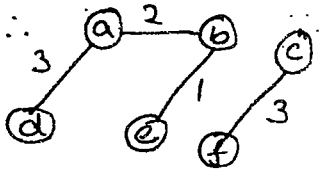
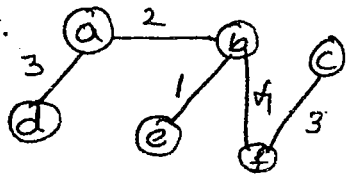
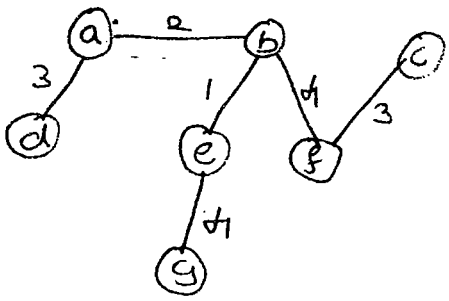
Kruskal's Algorithm:-

This algorithm is also used for finding minimum spanning tree of a connected weighted graph. Here, we will first sort the edges of the graph based on their weights. The edge with minimum weight is added to the empty tree. Then, among the remaining list, next smallest edge is considered and is added to the existing tree only if the addition will not create a cycle. If the addition of any edge to the existing tree results in a cycle, such an edge is just ignored. The procedure is continued till the tree contains all the vertices of a graph.

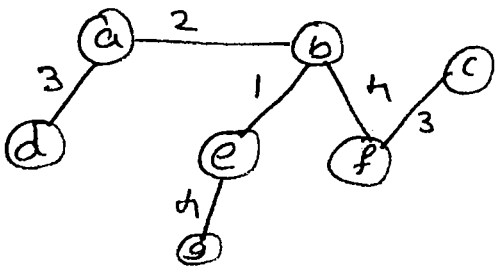
Consider the following graph -



To find the minimum spanning tree, we will proceed as below -

Tree edges	Sorted list of edges
	be ab ad cf bf eg de gf dg bc 1 2 3 3 4 4 5 6 7 9
be	be is inserted. So, 
ab	Next smallest is ab. So, 
ad	Now, ad. ∴ 
cf	Next smallest is cf. ∴ 
bf	Now, bf. ∴ 
eg	Again, eg ⇒ 

Next smallest is de. But this will create a cycle. Similarly, gf, dg, and bc will create cycles. So, we will ignore them.

Thus, required tree is - 

ALGORITHM Kruskal (G)

// For constructing MST by Kruskal's algo.

// Input : Weighted connected graph $G(V, E)$.

// Output : E_T, the set of edges which is MST.

Sort E in non-decreasing order of edge weights $w(e_1) \leq \dots \leq w(e_{|E|})$.

$E_T \leftarrow \phi$

edgecount $\leftarrow 0$

$k \leftarrow 0$

While edgecount $< |V| - 1$ do

$k \leftarrow k + 1$

if $E_T \cup \{e_i\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_i\}$

edgecount $\leftarrow$ edgecount $+ 1$

Return E_T.

NOTE : 1. Even though Kruskal's algorithm seems to be simpler than Prim's algorithm, it is not the fact. Because, before adding every edge to the MST, we have to check whether it results in a cycle or not in Kruskal's algorithm. That is, we have to compare the newly considered selected edge with already selected edges before inserting it in each step.

2. Unlike in Prim's algorithm, the intermediate steps of Kruskal's algorithm may be consists of disconnected edges. At the end only we will get MST here.

3. As Kruskal's algorithm involves sorting the edges, if we use efficient sorting technique to do so, the efficiency of the algorithm will be in $O(|E| \log |E|)$.

Dijkstra's Algorithm

We know that Floyd's algorithm is used to find all-pairs shortest path in a given weighted connected graph. That is, we were interested in a path that starts at any arbitrary vertex and visits all the vertices of a graph within minimum cost. On the other hand, some times, we may be interested in finding a shortest paths from a particular vertex to all other vertices. Such a problem is known as single-source shortest-paths problem. To solve this, Dijkstra's algorithm is used. Here, for a given vertex say, source, we will find shortest path

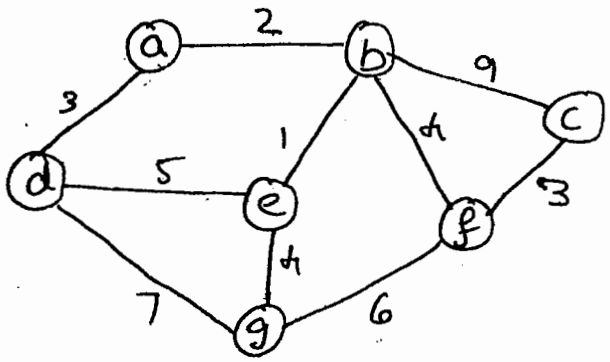
to all other vertices.

In this algorithm, given a source, first find the vertex which is nearer to it. Then find the second nearest & so on. Thus, at the i^{th} iteration, we will find the shortest distance between the source and the i^{th} vertex. At every step, we are going to get a tree of already considered vertices. The set of vertices adjacent to the vertices in T_i is referred to as 'fringe set'. After finding the nearest vertex u^* to the vertex in T_i, we have follow the following steps.

* Move u^* from the fringe set to T_i.
* For each remaining vertex u in fringe set that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$ respectively.

Consider a graph to illustrate Dijkstra's algorithm -

Chetana Hegde
9448301894



Suppose 'a' is given as source vertex. Now we have to find shortest distance of other vertices from a.

Tree vertices	Fringe set
a(-, 0)	b(a, 2), c(-, ∞), d(a, 3), e(-, ∞), f(-, ∞), g(-, ∞)
b(a, 2)	d(a, 3), c(b, 2+9), e(b, 2+1), f(b, 2+4), ~~g(b, 2+1+4)~~ g(-, ∞)
d(a, 3)	c(b, 2+9), e(b, 2+1), f(b, 2+4), g(d, 3+7)
e(b, 3)	c(b, 2+9), f(b, 2+4), g(e, 2+1+4)
f(b, 6)	c(f, 2+4+3), g(e, 2+1+4)
g(e, 7)	c(f, 2+4+3)
c(f, 9)	

Thus, the shortest paths set is given as –

From a to b : $a \rightarrow b$, Dist = 2

a to c : $a \rightarrow b \rightarrow f \rightarrow c$, Dist = 9

a to d : $a \rightarrow d$, Dist = 3

a to e : $a \rightarrow b \rightarrow e$, dist = 3

a to f : $a \rightarrow b \rightarrow f$, dist = 6

a to g : $a \rightarrow b \rightarrow e \rightarrow g$, dist = 7

```
ALGORITHM Dijkstra(G, s)
// Dijkastra's algorithm for single-source-shortest paths
// Input: A weighted connected graph G=(V, E)
//        and its vertex s, which is source.
// Output: The length d_v of a shortest path from
//         s to v and its penaltimate vertex
//         p_v for every vertex v in V.
Initialize(Q)  // An empty vertex priority queue is
               // initialized.
for every vertex v in V do
    d_v ← ∞
    p_v ← null
    Insert(Q, v, d_v)
d_s ← 0
Decrease(Q, s, d_s)  // Update priority of s with d_s
```

$V_T \leftarrow 0$

for $i \leftarrow 0$ to $|V|-1$ do

$u^* \leftarrow$ DeleteMin(Q) //delete min. priority element

$V_T \leftarrow V_T \cup \{u^*\}$

for every vertex u in $V - V_T$ that is adjacent to u^* do

if $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$

$P_u \leftarrow u^*$

Decrease (Q, u, d_u)

Analysis:

The efficiency of Dijkastra's algorithm depends on the data structure used for implementing priority queue and the input graph.

If the priority queue is represented using an array and the graph is represented using weight matrix, then the time complexity of the algorithm will be $\Theta(|V|^2)$.

If the priority queue is represented as a min-heap and the input graph is represented using adjacency linked list, then the time complexity will be. $O(|E| \log |V|)$.

HUFFMAN TREES

In the field of computer science, encoding and decoding a particular text / information is an important aspect, to save the space. Usually, encoding is done by assigning some sequence of bits called 'codeword' to each character of the text. There are mainly two methods of encoding viz. fixed-length encoding and variable-length encoding.

Fixed-length encoding assigns to each character a bit string of same length. If the text to be encoded has 'n' characters, then the number of bits required to encode that text is greater than or equal to $\log_2 n$. For example, if a text contains 8 characters, $\log_2 8 = 3$ bits are sufficient which we can arrange like 000, 001, 010, 011, 100, 101, 110 and 111.

More specifically, we can say that for a text of n characters, the bits required are $\lceil \log_2 n \rceil$.

Variable-length encoding assigns codewords of different lengths to different characters. ~~If a particular~~ Shorter sequence of bits are assigned to frequently occuring characters and longer sequence of bits are assigned to less frequent characters. But, in this method, we will unable to tell how many bits of an encoded text represent an i^{th} character of a text?

To avoid this problem, a method of prefix-free or prefix codes is used. Here, no codeword is a prefix of another codeword. To create a binary prefix code for some ~~alphabet~~ text, associate each character with leaves of binary tree, in which all left edges ar labeled by 0 and right edges are labeled by 1. Now, the sequence of labels found in a simple path from root to any leaf will give the codeword for a character which is at that leaf. Since, there will not be a simple path from one leaf to other leaf, no code can be prefix of other.

Huffman algorithm is used to construct such a tree which will give shorter code for frequent characters and longer code for less frequent characters.

step 1: Initialize n one-node trees and label them with the characters of text. Record the frequency of each character in it's tree's root to indicate tree's weight.

step 2: Find two trees with smallest weights. Make them left and right subtree of a new tree. Put the sum of their weights as a root of this new tree.
Repeat step ② till a single tree is obtained.

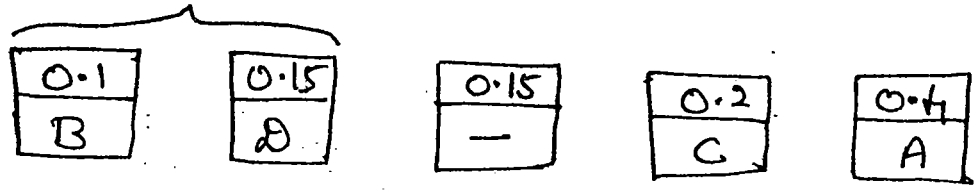
The tree constructed using above algorithm is Huffman tree and the codewords we get is Huffman code.

Ex: Construct a Huffman code for the following data:

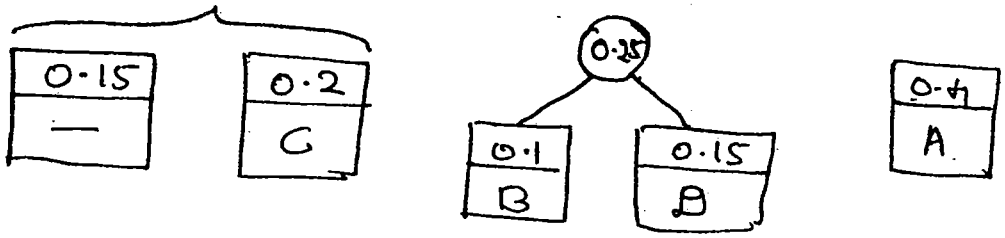
Character :	A	B	C	D	_
Probability :	0.4	0.1	0.2	0.15	0.15

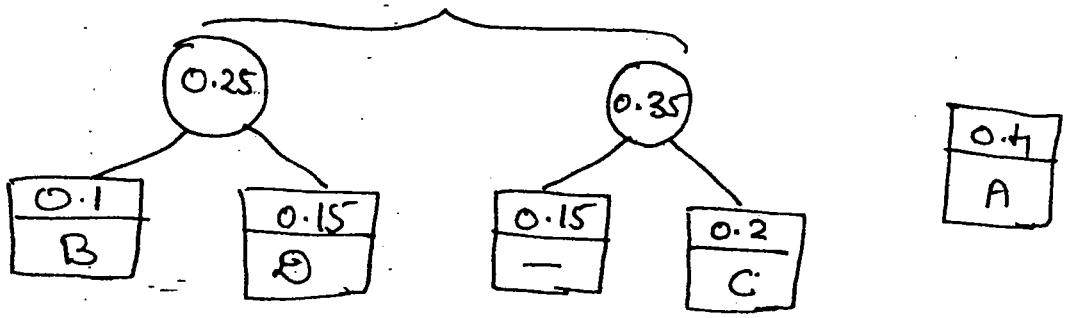
(i) Encode the text ABACABAD using the code.
(ii) Decode the code 100010111001010.

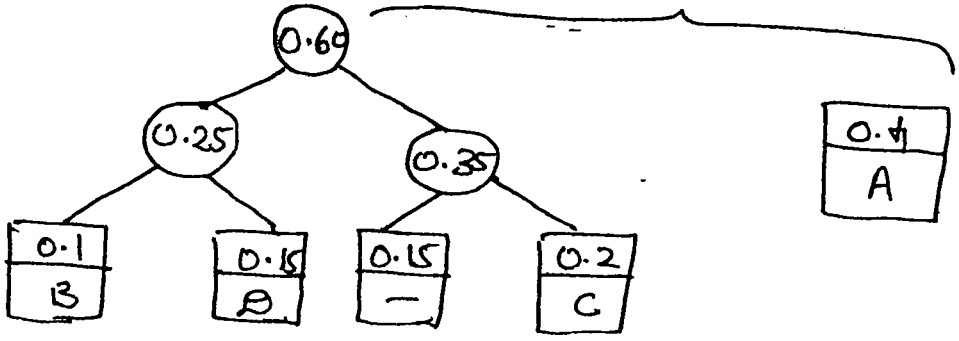
Sol²: In the first step, we create 5 one-node trees as below— (in ascending order of weights)

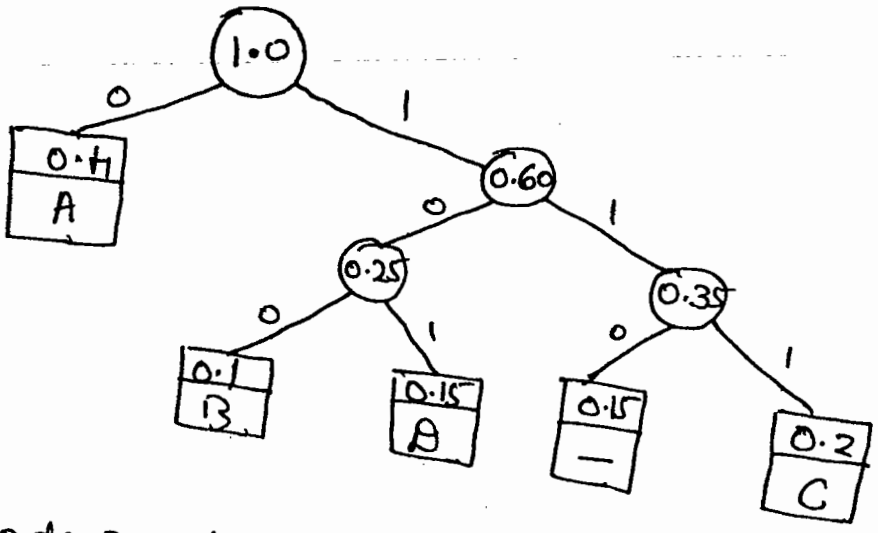


Combining smallest weighted trees—









Now codewords are –

A : 0

B : 100

C : 111

Ø : 101

— : 110

Chetana Hegde
9449830189

So, ABACABAØ can be encoded as –

0 100 0 111 0 100 0 101

And, 100 0 101 110 0 101 0 can be decoded as –

B A Ø — A Ø A