COLLATE

Algorithm Design and Analysis.

Notes

Unit 3

Syllabus

UNIT – III

Greedy Algorithms: Elements of Greedy strategy, overview of local and global optima, matroid, Activity
selection problem, Fractional Knapsack problem, Huffman Codes, A task scheduling problem. Minimum
Spanning Trees: Kruskal's and Prim's Algorithm, Single source shortest path: Dijkstra's and Bellman Ford
Algorithm(with proof of correctness of algorithms).

Unit -III-IV Greedy Algorithms.

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point, in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, &

The two main ingredients of greedy algorithms are:-

(a) Greedy choice property:- This property states that a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when we are considering which choice to make, we make that choice that looks best in the current problem, without considering results from subproblems.

(b) Optimal substructure:- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient

of assessing the applicability of DP as well as greedy algorithms.

Dynamic programming	Greedy Strategy
- bottom up manner.	- top down manner.
- always an optimal solution	- may or may not give an optimal solution.
- choice of a problem at each step depends on the solutions to subproblems	- We make whatever choice seems best at the moment & then solving the subproblem arising after the choice is made.

ACTIVITY SELECTION PROBLEM.

This is a problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a max. size set of mutually compatible activities.

Suppose, we have a set $S = \{a_1, a_2, \ldots a_n\}$ of n proposed activities, that wish to use a common resource, which can only be used by one activity at a time.

Each activity a_i, has a start time s_i & a finish time f_i, where $0 \le s_i < f_i < \infty$.

If selected, a_i takes place during the half open interval $[s_i, f_i)$. The two activities a_i & a_j are said to be compatible if the intervals $[s_i, f_i)$ & $[s_j, f_j)$ do not overlap. (i.e. $s_i \ge f_j$ or $s_j \ge f_i$).

The activity selection problem is to select a maximum size subset of mutually compatible activities.

Example:-

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

① Sort the activities in monotonically increasing order of finish time f_i.

② For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximal subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

→ Recursive Greedy Algorithm

```
Recursive_Activity_Selector(s, f, i, j)
m ← i+1
while m<j & s_m < f_i
    m ← m+1
if m<j
    then return {a_m} ∪ Recursive_Activity_Selector(s, f, m, j)
else return φ.
```

The procedure takes the start & finishing times of the activities, represented as arrays s & f, as well as starting indices i & j of the subproblem in S_{ij} it is to solve. It returns a max. size set of mutually compatible activities in S_{ij}.
the initial call is Recursive_Activity_Selector(s, f, 0, n+1).

sort time = $O(n \lg n)$ time

Recursive_Activity_selector = $O(n)$

total = $O(n) + O(n \lg n)$

$= O(n \lg n)$

→ Iterative greedy algorithm

```
Greedy_Activity_Selector(s,f)
m ← length[s]
A ← {a1}
i ← 1
for m ← 2 to n
    if sm ≥ fi
        A ← A ∪ {am}
        i ← m
return A
```

Assumption for both recursive & ~~greedy~~ iterative approach:-

- I/p activities are ordered by monotonically increasing time. ~~# collects~~

Time = $\Theta(n)$ assuming activities are already sorted.

Task Scheduling Problem.

Given a finite set S of unit time tasks, a schedule S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 & ends at time 2 & so on.

The problem of scheduling unit time tasks with deadlines & penalties for a single processor has the following inputs :-

(a) a set $S = \{a_1, a_2, \ldots a_n\}$ of n unit-time tasks

(b) a set of n integer deadlines, $d_1, d_2, \ldots d_n$, such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i.

(c) a set of n integers or penalties (non-negative), $w_1, w_2, \ldots w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.

We are asked to find a schedule for S that minimises the total penalty incurred for missed deadlines.

Example:-

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Step ① - Arrange tasks in decreasing order of their penalties w_i.

Step ②:- Check for a pairwise comparison:-

In the above example, the greedy algo. selects tasks a_1, a_2, a_3, a_4 & a_7 & then rejects a_5 & a_6.

The final optimal schedule is:-

$\langle a_2, a_4, a_3, a_1, a_7, a_5, a_6 \rangle$

which has a total penalty incurred of $w_5 + w_6 = 30 + 20 = 50$

Ques. Solve the instance of the scheduling problem given above, but with each penalty w_i replaced by $80 - w_i$.

* Huffman Codes.

Huffman codes are a widely used & very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build an optimal way of representing each character as a binary string.

	a	b	c	d	e	f
frequency (in thousands)	45	13	12	16	9	5
fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

Suppose, we have a 1,00,000 character data file that we wish to store compactly. Only six different characters appear in the file & the character 'a' occurs 45,000 times.

There are generally two ways to represent such a file of information:-

(a) fixed length code:- where we need 3 bits to represent six characters. This method requires 300,000 bits to code the entire file.

(b) Variable length code:- this can do considerably better, by giving frequent characters short codes & infrequent characters long codes. This would require:-

$(45\times1 + 13\times3 + 12\times3 + 16\times3 + 9\times4 + 5\times4)\times1000$

$= 2,24,000$ bits

This would save 25% approximately. In fact, this is an optimal character code for this file.

→ Prefix Codes

Codes in which no codeword is also a prefix of some other codeword, are called as prefix codes.

Encoding is always simple for any binary character code; we just concatenate codewords representing each character of the file. Example:- for the variable length prefix code, we code the 3-character file abc as $0.101.100 = 0101100$, where we use '.' to denote concatenation.

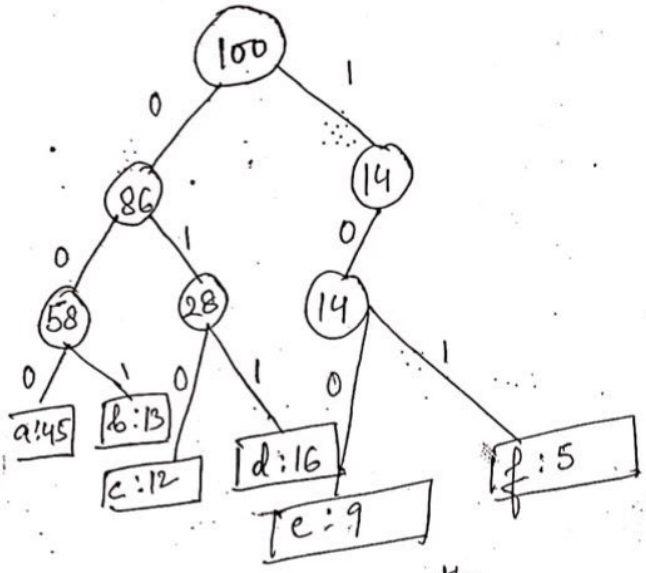
Decoding - prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, & repeat the decoding process on the remainder of the encoded file.

Example- 0 0 1 0 1 1 1 0 1

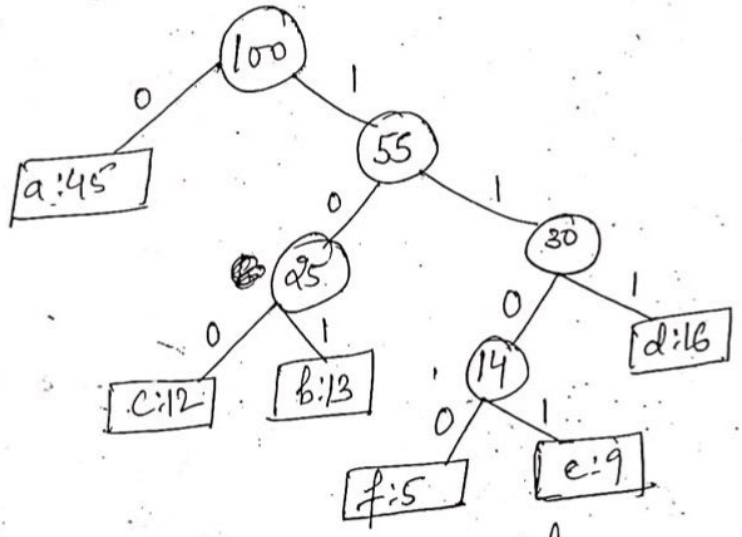
⇓

0.0.101.1101 → aabe

Representation of the decoding process - The decoding process needs a convinient representation of the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where "0" means "go to the left child" & "1" means "go to the right".



(a) fixed length codeword.



(b) Variable length codeword.

Each leaf is labelled with a character & its freq. of occurrence. Each internal node is labeled with the sum of the frequencies in its subtree.

Given a tree T corresponding to a prefix code, it is a simple matter to compute the no. of bits required to encode a file.

For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file & let $d_T(c)$ denote the depth of c's leaf in the tree. $d_T(c)$ is also the length of the codeword for character c.

The no. of bits required to encode a file is thus -

$$B(T) = \sum_{c \in C} f(c)\, d_T(c)$$

which we define the cost of the tree T.

→ Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code - called a Huffman code.

We assume that C is a set of n characters & that each character $c \in C$ is an object with a defined frequency $f[c]$. The algo. builds the tree T corresponding to the optimal code in a bottom up manner.

It begins with a set of |C| leaves & performs a sequence of |C|-1 merging operations to create the final tree.

A min-priority queue, Q, keyed on f, is used to identify the two least frequent objects to merge together. The result of the merger of 2 objects is a new objects whose frequency is the sum of the frequencies of the two objects that were merged.

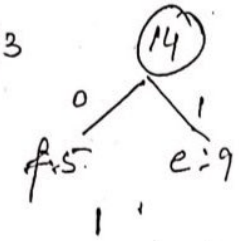
```
HUFFMAN(C)
n ← |C|
Q ← C                    } n times
for i ← 1 to n-1         } n times
    do allocate a new node z
       left[z] ← x ← MIN-EXTRACT(Q)   } lgn times
       right[z] ← y ← MIN-EXTRACT(Q)  } lgn times
       f[z] ← f[x] + f[y]
       INSERT(Q, z)
return MIN-EXTRACT(Q)
```

The min-priority queue, Q is implemented as a binary min-heap & hence takes O(lgn) time for extraction. Therefore, O(nlgn) total time.

(a)

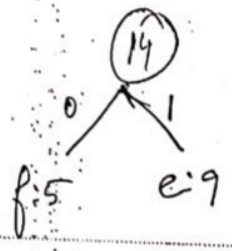
f:5 e:9 c:12 b:13 d:16 a:45

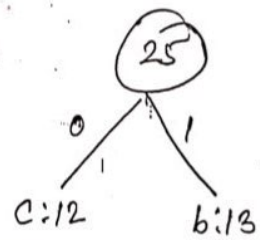
(b)

c:12 b:13 

d:16 a:45

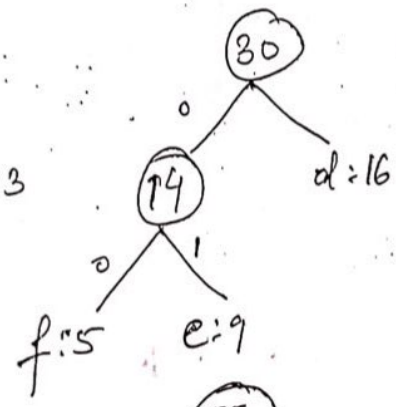
(c)



d:16 

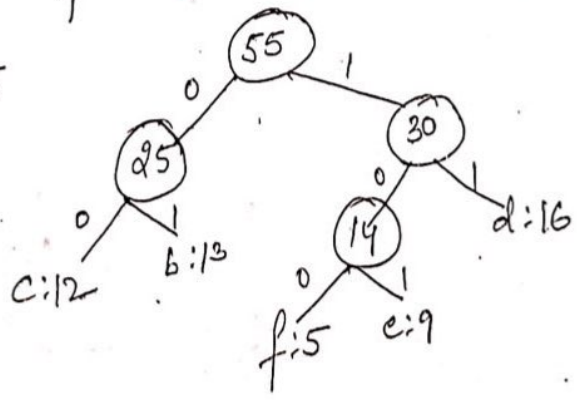
a:45

(d)

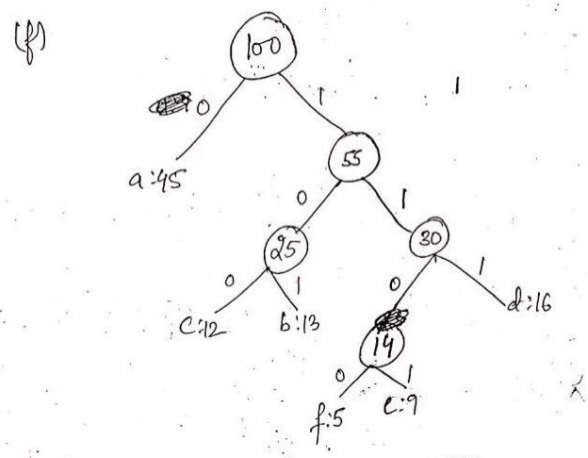


a:45

(e)

a:45 

(f)



Ques. What is an optimal Huffman code for the following frequencies;

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

#. Minimum Spanning Trees

Let us consider a connected, undirected graph $G = (V, E)$, where V is the set of ~~edges~~ vertices & E is the set of edges, and for each edge $(u,v) \in E$ we have a weight $w(u,v)$ specifying the cost to connect u & v. We then wish to find an acyclic subset $T \subseteq E$ that connects all the vertices and whose total weight

$$w(T) \triangleq \sum_{(u,v) \in T} w(u,v) \text{ is minimized.}$$

Since T is acyclic & connects all of the vertices, it must form a tree, which we call a spanning tree since it 'spans' the graph G. We call the problem of determining the tree T the minimum spanning tree problem. (MST)

The two algorithms, for solving the MST problem:-
(a) Kruskal algo. (b) Prim's algo.

The two ~~greedy~~ algorithms are greedy algorithms. The greedy strategy advocates making the choice that is best at the moment. Such a

strategy is not generally guaranteed to find optimal solutions to problems. For the MST problem we can guarantee or prove that certain greedy strategies do yield a spanning tree with minimum weight.

KRUSKAL AGORITHM

```
MST_KRUSKAL(G, w)
A ← φ
for each vertex v ∈ V[G]
    do MAKE_SET(v)
sort the edges of E into increasing order by wt. w
for each edge (u,v) ∈ E taken in increasing order by wt. w
    if FIND_SET(u) ≠ FIND_SET(v)
        then A ← A ∪ {(u,v)}
             UNION(u,v)
return A.
```

Kruskal's algo. is a greedy algo, because at. each step it adds to the forest an edge of least possible weight.

It uses a disjoint set DS to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. The operation

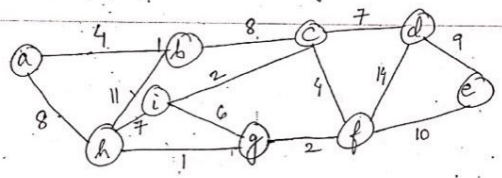
FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u & v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). The combining of trees is accomplished by the UNION procedure.

Running time = $\boxed{O(E \lg E)}$

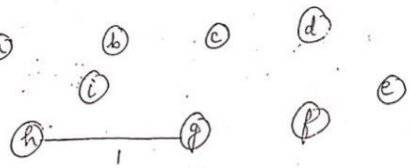
If $\boxed{|E| < |V|^2} \Rightarrow \lg |E| = O(\lg V)$

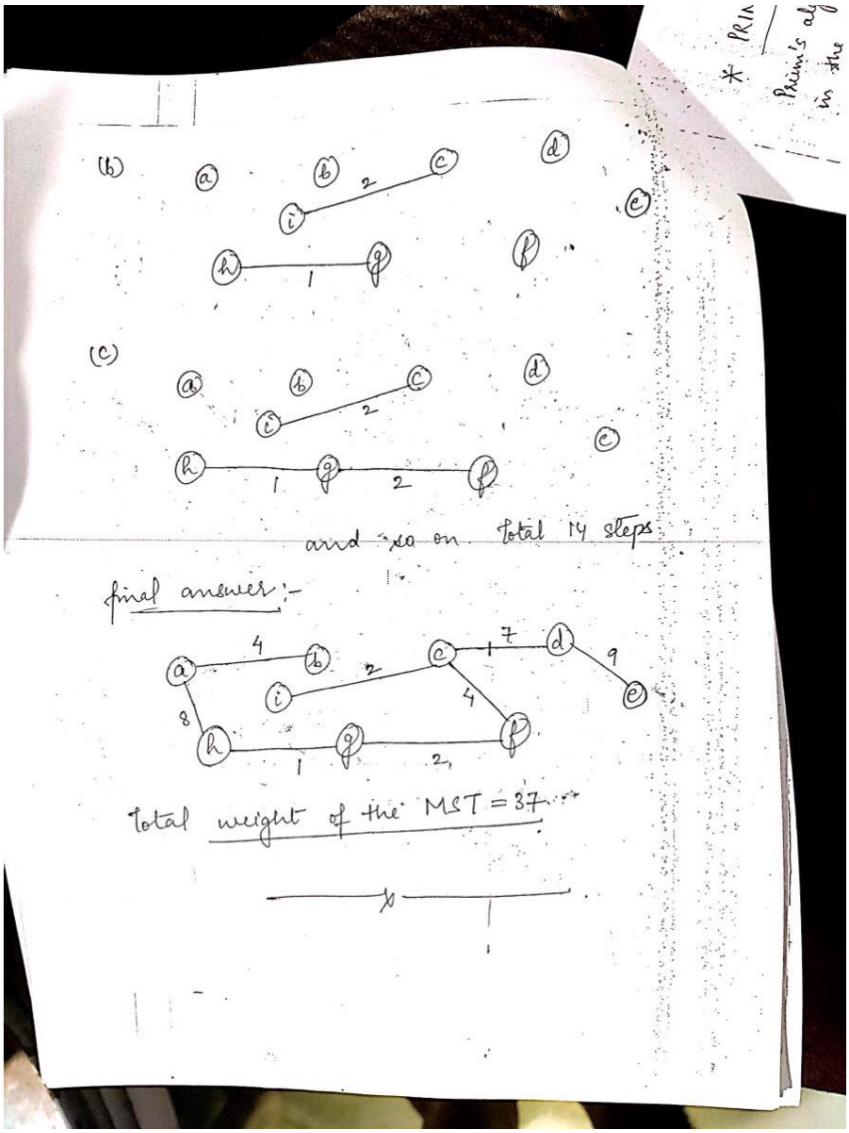
we have, running time = $\boxed{O(E \lg V)}$

Ques.



Procedure:-

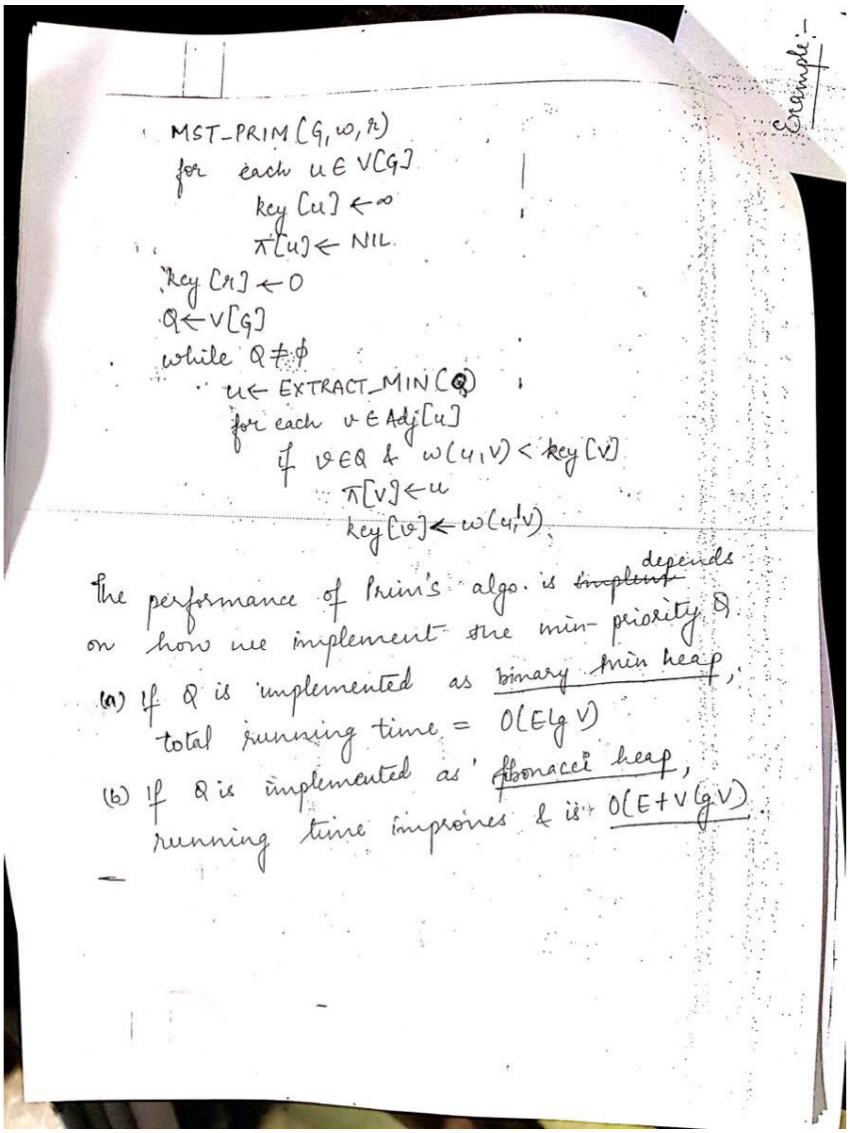




* PRIM'S ALGORITHM.

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r & grows until the tree spans all the vertices in V. When the algo. terminates, the edges in A form a minimum spanning tree. This is a greedy strategy since the tree is augmented at each step with an edge that contributes the minimum ~~also~~ amount possible to the tree's weight.

During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key field. For each vertex v, key[v] is the minimum weight of any edge connecting v to the vertex in the tree; by convention key[v] = ∞ if there is no tree. The field $\pi[v]$ names the parent of v in the tree. When the algorithm terminates, the min-priority queue is empty.



Example:-

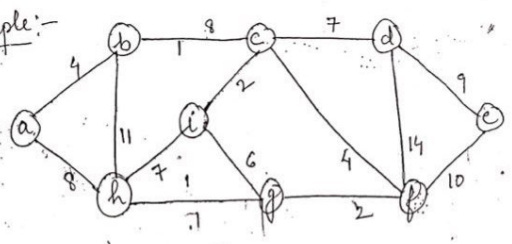
```
MST_PRIM(G, w, r)
for each u ∈ V[G]
    key[u] ← ∞
    π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ ϕ
    u ← EXTRACT_MIN(Q)
    for each v ∈ Adj[u]
        if v ∈ Q & w(u,v) < key[v]
            π[v] ← u
            key[v] ← w(u,v)
```

The performance of Prim's algo. depends on how we implement the min-priority Q.

(a) If Q is implemented as binary min heap, total running time = $O(E \lg V)$

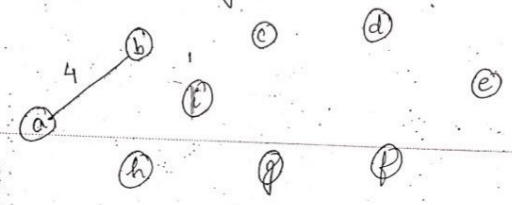
(b) If Q is implemented as Fibonacci heap, running time improves & is $O(E + V \lg V)$.

Example:-

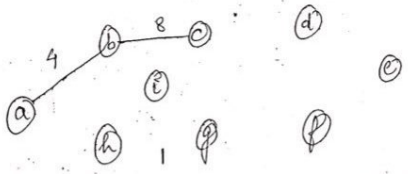


Procedure : choosing r=a

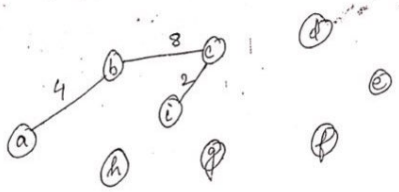
(a)

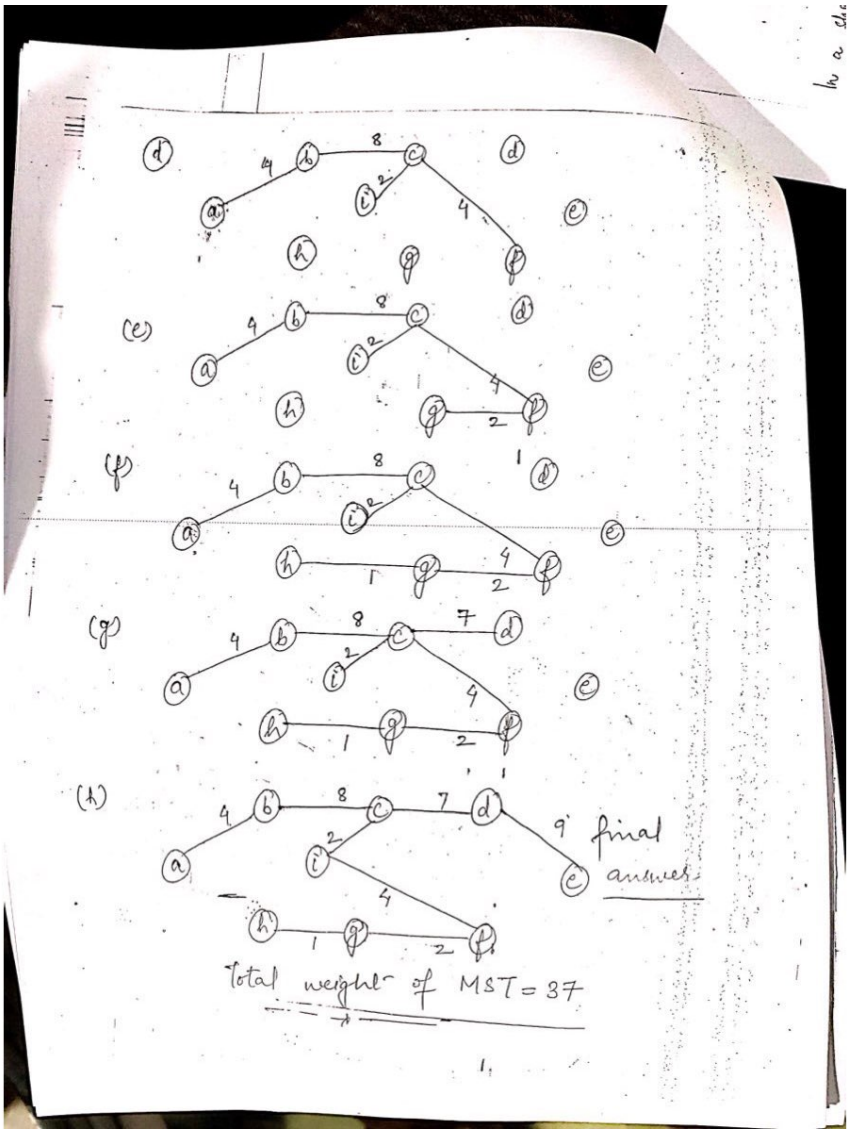


(b)



(c)





Single Source Shortest Paths.

In a shortest paths problem, we are given a weighted directed graph $G=(V,E)$, with weight function $w: E \to R$ mapping edges to real valued weights. The weight of path $p = \langle v_0, v_1, v_2 \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The shortest path weight from u to v is defined by:-

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\}, & \text{if there is a path from } u \text{ to } v \\ \infty & \text{; otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined by as any path p with weight $w(p) = \delta(u,v)$.

→ Variants.

Given a graph $G=(V,E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$. Many other problems can be solved by the algorithm for the single source problem, including the following variants.

(a) Single destination shortest paths problem :- Find a shortest path to a given destination vertex t from each vertex v.
By reversing the direction of each edge in the graph, we can reduce this problem to a single source problem.

(b) Single pair shortest path problem :- Find a shortest path from u to v for given vertices u and v.

(c) All pair shortest path problem :- Find a shortest path from u to v for every pair of vertices u and v.

→ Optimal substructure of a shortest path.

Shortest paths algorithms rely on the property that a shortest path b/w 2 vertices contains other shortest paths within it.

* Given a weighted, directed graph $G=(V,E)$ with weight functions $w: E \rightarrow R$, let $p = \langle v_1, v_2 \dots v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k and, for any i & j such that $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1} \dots v_j \rangle$ be the subpath of p from vertex v_i & vertex v_j. Then p_{ij} the shortest path from v_i to v_j.

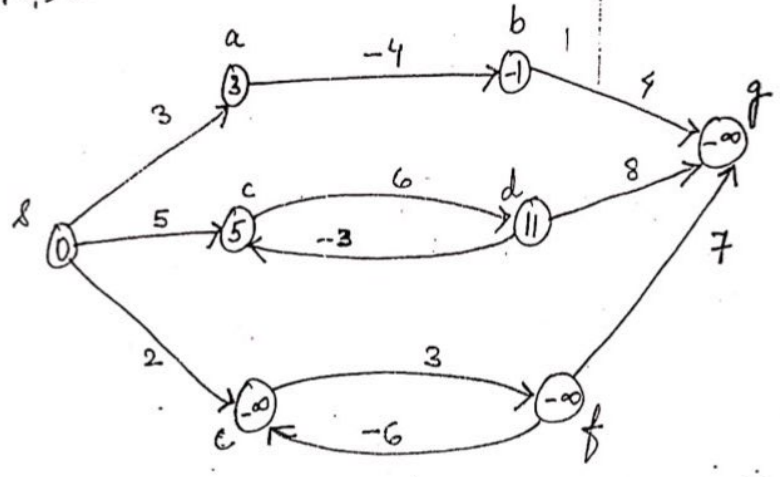
Proof:- If we decompose path p into $v_1 \xrightarrow{p_{ij}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, then we have that $w(p) = w(p_{ij}) + w($

Proof:- If we decompose the path p into $v_1 \xrightarrow{p_{ii}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, then we have that $w(p) = w(p_{ii}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_1 \xrightarrow{p_{ii}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ is a path from v_1 to v_k whose weight $w(p_{ij}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_i to v_k.

→ Negative weight edges.

If the graph $G=(V,E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest path weight $\delta(s,v)$ remains well defined, even if it has a negative value. If there is a negative weight cycle reachable from s, however, shortest path weights are not well defined. No vertex on the cycle can be a shortest path - a lesser weight path can always be found that follows

the "proposed" shortest path & then traverses the negative weight cycle. If there is a negative cycle on some path from s to v, we define $\delta(s, v) = -\infty$.



In the figure above, there is a single path from s to a, i.e. $\langle s, a \rangle$ $\therefore \delta(s,a) = w(s,a) = 3$.

Similarly, there is only one path from s to b, and so $\delta(s,b) = w(s,a) + w(a,b)$

$= 3 + (-4) = -1$.

There are ∞ many paths from s to c: $\langle s, c \rangle$, $\langle s, c, d, c \rangle$, $\langle s, c, d, c, d, c \rangle$, & so on. Because the cycle $\langle c, d, c \rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from s to c is $\langle s, c \rangle$ with weight $w(s,c) = 5$.

The shortest path from s to d is $\langle s, c, d \rangle$, with weight $\delta(s, d) = \delta(s, c) + \delta(c, d)$

$= w(s, c) + w(c, d)$

$= 5 + 6 = 11.$

There are ∞ many paths from s to e:

$\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$ & so on.

The cycle $\langle e, f, e \rangle$ has weight $3 + (-6) = -3 < 0$, therefore there is no shortest path from s to e. So,

$\delta(s, e) = -\infty$.

Algorithms :-

(a) Dijkstra - assumes that all edge weights in the I/P graph are nonnegative

(b) Bellman ford - Allow negative weight edges in the I/P graph & produce a correct answer as long as no negative weight cycles are reachable from the source. Typically, if there is such a negative weight cycle, the algorithm can detect & report its existence.

→ Cycles :-

The shortest path cannot contain a negative weight or positive weight cycle, since removing the

cycle from the path produces a path with the same source & destination vertices & a lower path weight.

→ Representing shortest paths.

Given a graph $G = (V, E)$, we maintain for each vertex $v \in V[G]$, a predecessor $\pi[v]$ that is either another vertex or NIL.

Predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ defines a set of vertices V_π with non NIL predecessors plus the source s:

$$V_\pi = \{ v \in V : \pi[v] \neq \text{NIL} \} \cup \{s\}$$

The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:

$$E_\pi = \{ (\pi[v], v) \in E : v \in V_\pi - \{s\} \}.$$

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to R$, & assume that G has no negative wt. cycles reachable from the source vertex $s \in V$, so that the shortest paths are well defined. A shortest paths tree rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$ such that:

1. V' is the set of vertices reachable from s in G.
2. G' forms a rooted tree with root s.
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique & neither are shortest path trees.

→ Relaxation.

The algorithms use the method of relaxation. For each vertex $v \in V$, the value $d[v]$ is the upper bound on the weight of a shortest path from source s to vertex v.

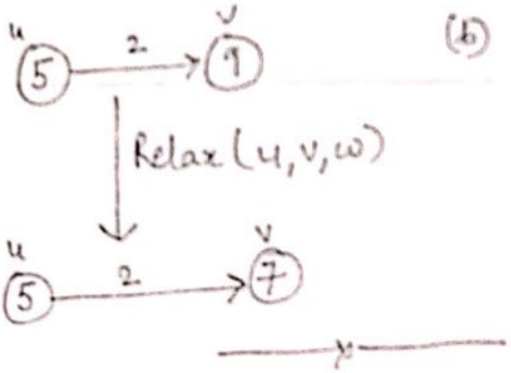
```
INITIALIZE(G,s) = O(V) time.
for each vertex v ∈ V[G]
        d[v] ← ∞
        π[v] ← NIL
d[s] ← 0
```

The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ & $\pi[v]$

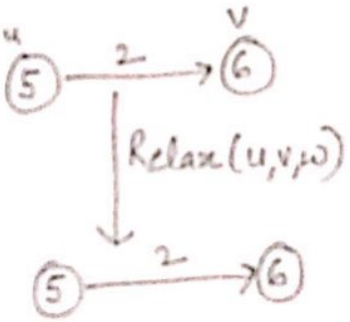
This relaxation step may decrease the value of the shortest path estimate $d[v]$ & update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u,v).

```
RELAX(u, v, w)
if d[v] > d[u] + w(u,v)
    d[v] ← d[u] + w(u,v)
    π[v] ← u.
```

(a)



(b)



DIJKISTRA ALGORITHM

value

DIJKISTRA ALGORITHM

Dijkistra's Algorithm 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. we assume that $w(u,v) \geq 0$ for each edge $(u,v) \in E$.

```
DIJKISTRA (G, w, s)
INITIALIZE (G, s)
S ← ϕ
Q ← V[G]
while Q ≠ ϕ
    u ← Extract_Min (Q)
    S ← S ∪ {u}
    for each vertex v ∈ Adj[u]
        Relax (u, v, w)
```

The algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(u,v)$. The algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest path estimate, inserts u into S, and relaxes all edges leaving u. A priority queue Q is maintained that contains all the vertices in V-S,

keyed by their d values. Graph G is represented by adjacency lists.

Because the algorithm always chooses the closest vertex in V−S to insert into set S, hence it uses a greedy strategy.

ANALYSIS:-

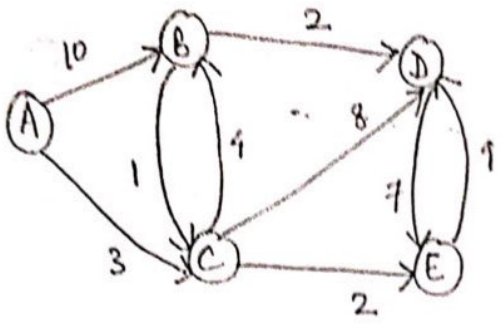
The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, & operation Extract_Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(V^2)$.

When $|E| < |V^2|$, the Graph can be stored efficiently in the form of adjacency lists & using a binary heap or fibonacci heap as a priority queue to implement the Extract_Min function.

Binary heap $O((E+V)\log V)$

Fibonacci heap $O(E + V\log V)$.

Example -



(1)

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	N	N	N	N	N

(2)

Q:	B	C	D	E
A	10	3	∞	∞
0	A	A	N	N

(3)

A	C	B	D	E
0	3	~~10~~ 7	~~∞~~ 11	~~∞~~ 5
N	A	~~A~~ C	~~A~~ C	N C

(4)

A	C	E	B	D
0	3	5	7	11
N	A	C	C	C

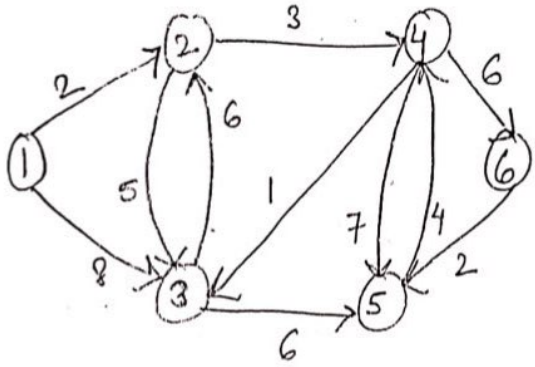
(5)

A	C	E	B	D
0	3	5	7	N 9
N	A	C	C	~~C~~ B

(6)

A	C	E	B	D
0	3	5	7	9
N	A	C	C	B

Ques solve using dijkistra algorithm.



Also, count the number of distance updates.

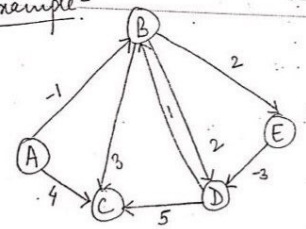
BELLMAN FORD ALGORITHM

Bellman ford algorithm finds all shortest path lengths from a source $s \in V$ to all $v \in V$ or determines that a negative weight cycle exists. Thus, Bellman ford algorithm solves the single source shortest path problem in the more general case in which edge weights can be negative.

Given a weighted, directed graph $G=(V,E)$ with source s & weight function $w: E \to R$, the Bellman ford algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths & their weights. The algorithm returns TRUE if & only if the graph contains no negative weight cycles that are reachable from the source.

```
BELLMAN_FORD (G, w, s)
INITIALIZE (G, s)
for i ← 1 to |V[G]| - 1
        for each edge (u, v) ∈ E[G]
                RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
        if d[v] > d[u] + w(u, v)
                return FALSE
return TRUE
```

Example:-



order of edges:- (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).

(i)

A	B	C	D	E
0/N	∞/N	∞/N	∞/N	∞/N

(ii)

A	B	C	D	E
0	∞	∞	∞	∞
0/N	-1/A	∞/N	∞/N	∞/N

(iii)

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0/N	-1/A	4/A	∞/N	∞/N

(iv)

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0/N	-1/A	2/B	∞/N	∞/N

Content Feedback

Feel free to submit your issues, suggestions and ratings regarding this course content

Click Here

Join & Share WhatsApp Group of COLLATE

(Click the button to join group)

MSIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

MAIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

ADGITM (NIEC)

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

GTBIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BVP

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BPIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	