LECTURE-26 UNIT-III STMatrix ①

Strassen's Matrix Multiplication :

→ It is an application of divide-and-conquer design technique

→ Suppose we want to compute a product $C = AB$ where A & B are $n \times n$ matrices. (assuming that n is exact power of 2)

→ ~~Divide~~ Normal Method let $C = AB$ is as

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix}$$ so matrix Multiplication is computed as:

$$\left.\begin{aligned} r &= ae+bf \\ s &= ag+bh \\ t &= ce+df \\ u &= cg+dh \end{aligned}\right\}$$ Each of these 4 equations specifies ~~two~~ two multiplication of $n/2 * n/2$ matrices & addition of their product

$T(n) = 8T(n/2) + \Theta(n^2)$

Strassen's Method :

Suppose $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

In Strassen's Method 1st compute the 7 $n/2 * n/2$ matrices as:

P, Q, R, S, T, U, V as:

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})B_{11}$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

Then C_{ij}'s are computed as:

$C_{11} = P+S-T+V$

$C_{12} = R+T$

$C_{21} = Q+S$

$C_{22} = P+R-Q+U$

→ P, Q, R, S, T, U, V can be computed using 7 multiplication & 10 addition / subtraction

$\Theta(n^{2.81})$ $T(n) = 7T(n/2) + an^2$

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

Strassen's Matrix Multiplication (02)

Show the strassen's matrix multiplication on the following matrices:

$$A = \begin{bmatrix} 4 & 2 & 0 & 1 \\ 3 & 1 & 2 & 5 \\ 3 & 2 & 1 & 4 \\ 5 & 2 & 6 & 7 \end{bmatrix}_{4\times4} \qquad B = \begin{bmatrix} 2 & 1 & 3 & 2 \\ 5 & 4 & 2 & 3 \\ 1 & 4 & 0 & 2 \\ 3 & 2 & 4 & 1 \end{bmatrix}_{4\times4}$$

Partition Matrices into submatrices as:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

with

$$A_{11} = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}, \quad A_{12} = \begin{bmatrix} 0 & 1 \\ 2 & 5 \end{bmatrix}, \quad A_{21} = \begin{bmatrix} 3 & 2 \\ 5 & 2 \end{bmatrix}, \quad A_{22} = \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 2 & 1 \\ 5 & 4 \end{bmatrix}, \quad B_{12} = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}, \quad B_{21} = \begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix}, \quad B_{22} = \begin{bmatrix} 0 & 2 \\ 4 & 1 \end{bmatrix}$$

Now compute P, Q, R, S, T, U, V as:

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$= \begin{bmatrix} 5 & 6 \\ 9 & 8 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 9 & 5 \end{bmatrix} = \begin{bmatrix} 64 & 45 \\ 90 & 67 \end{bmatrix}$$

likewise

$$Q = \begin{bmatrix} 38 & 29 \\ 67 & 47 \end{bmatrix}, \quad R = \begin{bmatrix} 8 & 4 \\ 7 & 2 \end{bmatrix}, \quad S = \begin{bmatrix} 9 & -5 \\ -20 & 4 \end{bmatrix}, \quad T = \begin{bmatrix} 12 & 11 \\ 24 & 16 \end{bmatrix}$$

$$U = \begin{bmatrix} -5 & -3 \\ 17 & 13 \end{bmatrix}, \quad V = \begin{bmatrix} -22 & -15 \\ -18 & -30 \end{bmatrix}$$

Now compute $C_{11}, C_{12}, C_{21}, C_{22}$ as:

$$C_{11} = P + S - T + V$$

$$= \begin{bmatrix} 64 & 45 \\ 90 & 67 \end{bmatrix} + \begin{bmatrix} 9 & -5 \\ -20 & 4 \end{bmatrix} - \begin{bmatrix} 12 & 11 \\ 24 & 16 \end{bmatrix} + \begin{bmatrix} -22 & -15 \\ -18 & -30 \end{bmatrix}$$

like wise $C_2 = R+T = \begin{bmatrix} 20 & 15 \\ 31 & 18 \end{bmatrix}$, $C_{21} = \begin{bmatrix} 29 & 23 \\ 47 & 51 \end{bmatrix}$

$C_{22} = \begin{bmatrix} 29 & 18 \\ 47 & 35 \end{bmatrix}$

Final Matrix $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} 21 & 14 & 20 & 15 \\ 28 & 25 & 31 & 18 \\ 29 & 23 & 29 & 18 \\ 47 & 51 & 47 & 35 \end{bmatrix}$

LECTURE-28

Greedy Approach:

The greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal ~~solution~~ choice in the hope that this choice will lead to a globally optimal sol^n.

→ This heuristic strategy does not always produce an optimal sol^n, but sometimes it does.

Differences b/w dynamic programming and Greedy Approach

→ In dynamic programming, we make choice at each step, but choice usually depends on the sol^{n}s to the subproblems

→ In greedy, we make whatever choice ~~it~~ seems best at the moment & then solve subproblems arising after choice is made. The choice made by greedy algo may depend choices so far, but it can not depend on any future choices or on the sol^{n}s to subproblems.

→ Dynamic programming solve problem in bottom-up manner while a greedy strategy solves a problem in Top-down fashion, make one greedy choice after another, reducing each given problem to smaller one.

(1) An activity-selection problem:

This is the problem of scheduling several competing activities that requires exclusive use of a common resource, with a goal of selecting a maximum-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 exclusively.

→ Each activity a_i has start time s_i and a finish time f_i

Activity-Selection problem Contd.

→ If an activity a_i selected, then it takes place during the interval (half-open time intervals) $[s_i, f_i)$.

→ Activities a_i and a_j are compatible if the intervals $[s_i, f_i]$ and $[s_j, f_j]$ do not ~~of~~ overlap i.e a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$

→ The activity-selection problem is to select a max. 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

Greedy-Activity-Selector(s, f)

1. n ← length[s]
2. A ← $\{a_1\}$
3. i ← 1
4. for m ← 2 to n
5. do if $s_m \geq f_i$
6. then A ← A ∪ $\{a_m\}$
7. i ← m
8. Return A.

or $\{a_1, a_4, a_8, a_{11}\}$
or $\{a_2, a_4, a_9, a_{11}\}$

Complexity:

$O(n)$

- In fractional knapsack problem, the thief does not need to steal all of an item, but rather items can be broken into smaller pieces and thief can take any fraction of the item. So, the thief may take only a fraction x_i of ith item. where

$$0 \leq x_i \leq 1$$

- ith item contributes the weight $x_i \cdot w_i$ to the total weight in the knapsack and profit $x_i * p_i$ to the total profit.
- Hence the objective of this algorithm is to

$$\text{maximize} \sum_{n=1}^{n} x_i * p_i$$

Subject to constraint

$$\sum_{n=1}^{n} x_i * w_i \leq W$$

→ It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

→ In this context, first we need to sort items according to the value of $\frac{p_i}{w_i}$ (profit per unit)

P.T.O

Algorithm:

Greedy-fractional-knapsack (W[1....n], P[1...n], W)

```
1. For i=1 to n
2.     do P[i] = V[i]/W[i]
3. Sort-Descending(P)
4. For i=1 to n
5.     do x[i]=0;
6. weight=0;
7. For i=1 to n
8.     if weight + W[i] ≤ W then
9.         x[i]=1;
10.        weight = weight + W[i]
11.    Else
12.        x[i] = (W - weight)/W[i]
13.        weight = W
14.        Break;
15. Return x;
```

Example: Solve the given instance of knapsack problem using Greedy approach:

item $I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$

weight $w = \langle 5, 10, 20, 30, 40 \rangle$

profit $v = \langle 30, 20, 100, 90, 160 \rangle$

Knapsack capacity $W = 60$

Soln:

I_i	w_i	v_i	$p_i = v_i/w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

sort →

I_i	w_i	v_i	$p_i = v_i/w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

I_5	35/40
I_3	20
I_1	5

$\Rightarrow \quad x_1 * w_1 + x_3 * w_3 + \frac{7}{8} x_5 * w_5$

Total profit $= 1*30 + 1*100 + \frac{7}{8}*160$

$= 30+100+140 = (270)$

Soln is $= (1, 0, 1, 0, \frac{7}{8})$

Example 2 $I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$

$W = \langle 1, 3, 4, 5 \rangle$

Profit $v_i = \langle 1, 4, 5, 7 \rangle$

$W = 8$

Example3 $n=3$, $W=20$, $\langle P_1, P_2, P_3 \rangle = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

— Solve above instances using Greedy Approach

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

→ It is a variable-length code.

→ Prefix codes: The codes in which no codeword is also a prefix of some other codeword. Such codes are called prefix codes.

→ Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file.

Cost of the Tree: Given a tree T corresponding to a prefix code, we can compute the number of bits required to encode a file. For each character c in alphabet C, let $f(c)$ denote frequency of c and let $d_T(c)$ denote the depth of c's leaf in tree T. $d_T(c)$ also is the length of codeword for c. The number of bits required to encode a file is

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

Which is known as the cost of tree T.

Example:

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

HUFFMAN (C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. For $i \leftarrow 1$ to $n-1$
4. do allocate a new node z
5. left[z] $\leftarrow x \leftarrow$ Extract-Min(Q)
6. Right[z] $\leftarrow y \leftarrow$ Extract-Min(Q)
7. $f[z] \leftarrow f[x] + f[y]$
8. INSERT(Q, z)
9. return Extract-min(Q)

Job sequencing with deadlines: We are given a set of n jobs

Each job i is associated an integer deadline $d_i \geq 0$ and profit $P_i > 0$

For any job i the profit P_i is earned iff the job is completed by its deadline.

→ To complete a job, one has to process the job on a machine for one unit of time. Only one m/c is available for processing the jobs.

→ A Feasible soln for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of profits of jobs in J or $\sum_{i \in J} P_i$.

→ An optimal soln is a feasible soln with max. value.

Example: Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. Feasible sol's ~~are~~ their values are.

	Feasible sol's.	processing sequence	value
1.	(1,2)	2,1	110
2	(1,3)	1;3 or 3,1	115
→ 3	(1,4)	1,4	127
4	(2,3)	2,3	25
5	(3,4)	(4,3)	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Soln. 3 is the optimal soln

Job sequencing algorithm:

```
1. Algorithm Greedy (d, J, n)
2. // J is a set of jobs that can be completed by their
                                                  deadlines
3. {
4.     J := {1};
5.     For i=2 to n do
6.     {
7.         if (all jobs in J U {i} can be completed by
8.         by their deadlines) then J := J U {i};
9.     }
10. }.
```

→ ~~Hear~~ Here jobs are sorted such that ~~$P_1 \leq P_2 \leq P_3 \leq \cdots$~~

$P_1 \geq P_2 \geq P_3 \geq \cdots \cdots P_n$.

Example: Let $n = 5$, $(P_1 \cdots P_5) = (20, 15, 10, 5, 1)$ and $(d_1 \cdots d_5) = (2, 2, 1, 3, 3)$

J	assigned slots	Job considered	Action	Profit
ϕ	none	1	assign to [1,2]	0
{1}	[1,2]	2	assign [0,1]	20
{1,2}	[0,1] [1,2]	3	reject	35
{1,2}	[0,1], [1,2]	4	assign [2,3]	35
{1,2,4}	[0,1], [1,2], [2,3]	5	reject	40

The optimal soln is $J = \{1, 2, 4\}$ with profit = 40

Q.1: What is the soln by job seq. Algo when $n = 7$, $(P_1, P_2 \cdots P_7) = (3, 5, 20, 18, 1, 6, 30)$ and $(d_1, d_2 \cdots d_7) = (1, 3, 4, 3, 2, 1, 2)$?

(1) Minimum Spanning Tree (MST) (Greedy Approach) In the design of electronic circuitry, to interconnect a set of pin (n), we can use an arrangement of (n-1) wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

→ We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections b/w pairs of pins.

→ For each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.

→ Then we wish to find an acyclic set $T \subseteq E$ that connects all of the vertices and whose total weight $w(T)$ is

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

→ Since T is acyclic and connects all of the vertices, it must form a tree, which is called Spanning Tree.

→ We wish to find a minimum weight Spanning Tree, called min-spanning Tree.

(2) Generic-MST(G, w) ← Generic algo for mst

1. $A \leftarrow \emptyset$
2. while A does not form a spanning tree
3. do find ~~one~~ an edge (u, v) that is safe for A.
4. $A \leftarrow A \cup \{(u, v)\}$
5. return A

P.T.O

Algorithms for MST.

① **Kruskal's algo**: Kruskal's algo is based directly on the generic MST algo. It finds a safe edge to add to the growing forest by finding an edge (u,v) of least weight.

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

→ It uses a disjoint-set data structure 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 contain u.

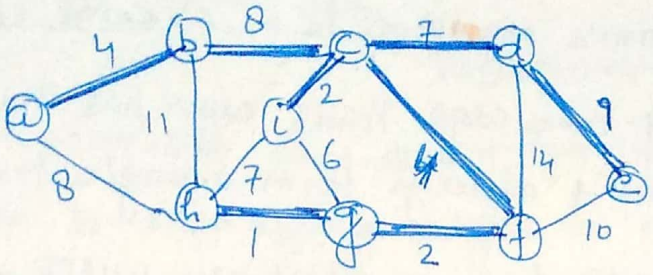
→ We can determine whether two vertices u & v belongs to the same tree by testing whether FIND-SET(u) equals to FIND-SET(v)

→ Combing of Trees is accomplished by the union operation.

Algo:-

```
MST-KRUSKAL(G, w)
1. A ← ϕ
2. For each vertex v ∈ V[G]
3. do MAKE-SET(v)
4. Sort the edges of E into nondecreasing order of weight
5. For each edge (u,v) ∈ E, taken in nondecreasing order by weight
6.    do if FIND-SET(u) ≠ FIND-SET(v)
7.          then A ← A ∪ {(u,v)}
8.               Union(u,v)
9. Return A.
```

Example:



Edges → vertices

↓ {a} {b} {c} {d} {e} {f} {g} {h} {i}

(g,h) {g,h}

(g,f) {f,g,h}

(c,i) {c,i}

(a,b) {a,b}

(c,f) {c,f,g,h}

X (g,i)

(c,d) {c,d,f,g,h,i}

X (h,i)

(b,c) {a,b,c,d,f,g,h,i}

X (a,h)

(d,e) {a,b,c,d,e,f,g,h,i}

X (b,h)

X (d,f)

Total weight $w(T) = 37$

→ Running time of Kruskal's : $O(E \lg V)$

↳ Time to sort the edges: $O(E \lg E)$

↳ Make-SET operations: $|V|$

↳ There are max. $O(E)$ find-set & union operations.

Thus total time $(V+E)\alpha(V)$ where $\alpha(V)$ is a slowly growing function

So $(V+E)\log V\ \alpha(V) = O(\log V) = O(\log E)$

Prim's algorithm:

Prim's algorithm is a special case of the generic min-spanning-tree algo. Prim's algo has the property that the edges in the set A always form a single tree.

→ Here, the tree starts from an arbitrary vertex r and grows until the tree spans all the vertices in V. At each step, a light edge is added to the tree A that connects A to an isolated vertex of G_A.

→ We only adds only safe edge for A;

```
MST-PRIM (G, w, r)
1.  For each u ∈ V[G]
2.      do key[u] ← ∞
3.          π[u] ← NIL
4.  key[r] ← 0
5.  Q ← V[G]
6.  while Q ≠ ϕ
7.      do u ← EXTRACT-MIN(Q)
8.          for each vertex v ∈ Adj[u]
9.              do if v ∈ Q and w(u,v) < key[v]
10.                 then π[v] ← u
11                      key[v] ← w(u,v)
```

Single-source Shortest Paths:

In 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 $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of weights of its ~~constituent~~ constituent edges.

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

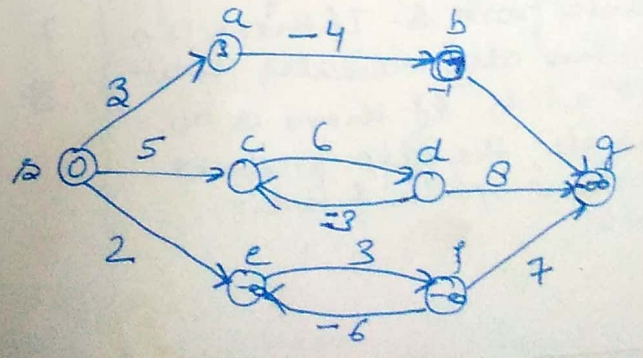
→ The shortest-path weight from u to v is defined as

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

→ Negative-weight edges: In some instances of single-source shortest-path problem, there may be edges whose weights are negative. 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 remains well defined, even if it has the negative value.

↳ If there is a negative-weight cycle reachable from s, shortest-path weights are not well defined.

↳ If there is a negative-weight cycle on some path from s to v, then we define $\delta(u, v) = -\infty$.



Note:-

(1) Shortest-path can not contain a negative-weight cycle.

(2) It also does not contain a positive-weight cycle.

→ It means shortest-path is cycle free.

→ If there are V no of vertices in a graph, then there is atmost $|V|-1$ edges in shortest path

Initialization:

```
INITIALIZE-SINGLE-SOURCE(G,s)
1. For each vertex v ∈ V[G]
2.    do d[v] ← ∞
3.       π[v] ← Nil
4. d[s] ← 0
```

d[v] — It is the upper bound on the weight of a shortest path from s to v.

Relaxation:

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

The Bellman-Ford Algo:

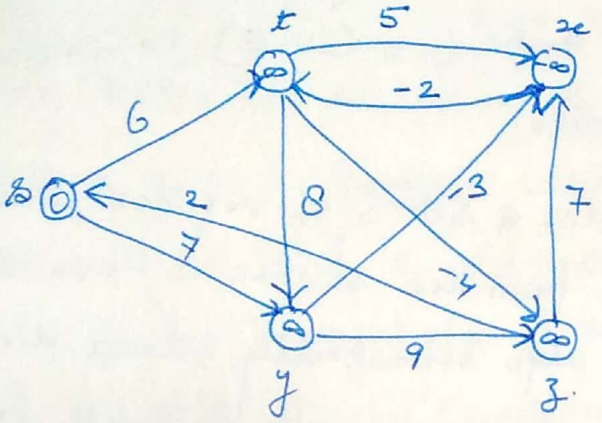
The Bellman-Ford algo. solves the single-source shortest path prob. in general case in which edge weight may be negative. Bellman-Ford algo returns a boolean value indicating whether or not there is a negative-weight cycle reachable from s. If there is a cycle, the algo indicates that no soln exists. If there is no such cycle, the algo produces the shortest-paths & their weights.

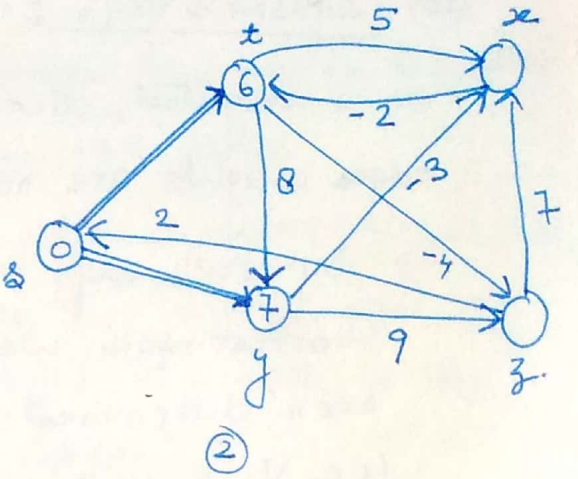
```
Bellman-FORD(G,w,s)
1. INITIALIZE-SINGLE-SOURCE(G,s)
2. For i ← 1 to |V[G]|-1
3.    do for each edge(u,v) ∈ E[G]
4.       do Relax(u,v,w)
5. For each edge(u,v) ∈ E[G]
6.    do if d[v] > d[u] + w(u,v)
7.       then Return False
8. Return True.
```

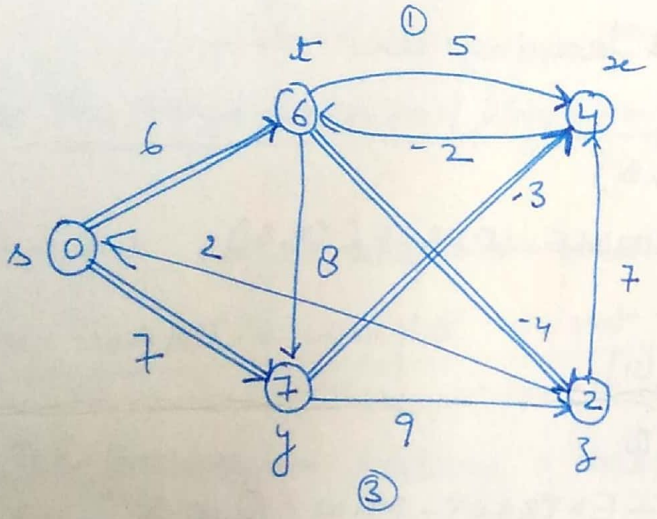
Complexity: $O(VE)$

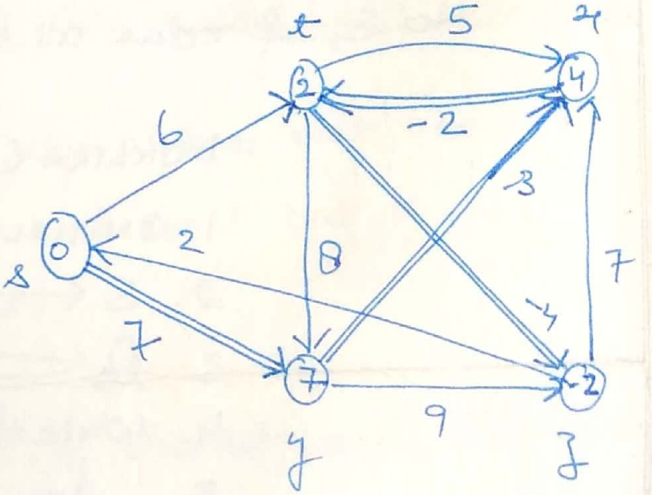
LECTURE-33

Example:

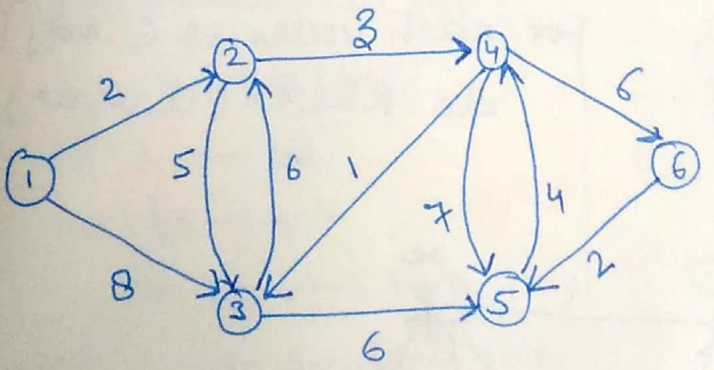








Q.1.



(11) **Dijkstra's algo**: It solves the single-source shortest path on a weighted, directed graph $G = (V, E)$ in which all edges weights are nonnegative.

→ Dijkstra's algo maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algo repeatedly selects the vertex $u \in V - S$ with min. shortest-path estimate, adds u to S, & relax all edges leaving u.

```
DIJKSTRA (G, w, s)
1. INITIALIZE-SINGLE-SOURCE (G, s)
2. S ← ϕ
3. Q ← V[G]
4. while Q ≠ ϕ
5.     do u ← EXTRACT-MIN (Q)
6.        S ← S ∪ {u}
7.        For each vertex v ∈ Adj[u]
8.            do RELAX (u, v, w)
```

Q: