UNIT-III

DAA - Lecture

Graph:- It is a non-linear, non-primitive data structure i.e represented with a set of vertices that are connected by edges i.e. $G = (V, E)$.

Classification of graph:→

1) On the basis of direction.
2) On the basis of weight value.
3) On the basis of connection of edges.
4) On the basis of no. of vertices.
5) " " " " Representation.

Classification of Graph.

- Directed Graph
- Undirected Graph
- Weighted graph
- Unweighted Graph
- Strongly Connected
- Disconnected Graph
- Dense Graph $|E| \leq V^2$
- Sparse Graph $|E| < V^2$
- Planner
- Non-planar

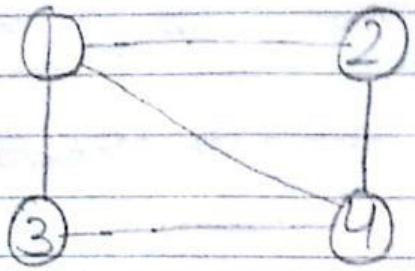
Representation of graphs:- Graph can be represented in the following ways:-

(1) **Adjacency Matrix:-** This representation is normally used for densed graph where no. of edges are quite high. The values can be represented as:-

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E[G] \\ 0 & \text{otherwise.} \end{cases}$$

This matrix will always be square matrix having an order of $n \times n$, where n is the no. of vertices in the graph.

For ex:-

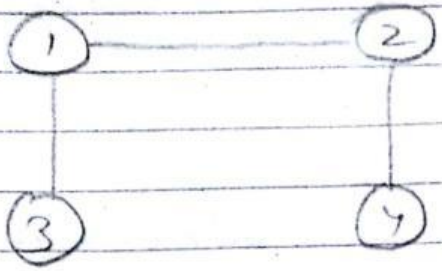


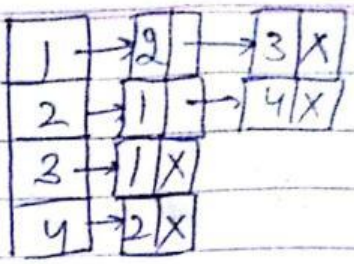
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

(2) Adjacency List:- This representation is particully used for sparse graph where no. of edges are less. In this representation a table of all the vertices is maintained & the corresponding adjacent nodes are represented in the form of linked list.

Since, it occupies larger memory space, Hence, is useful for sparse graph.

for ex :-





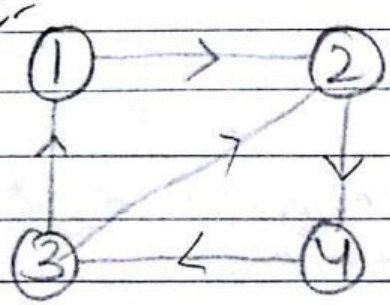
3) Incidence Matrix :-

$$\begin{cases} 1 & \text{if } (i,j) \in E[G] \\ -1 & \text{if } (i,j) \notin E[G] \\ 0 & \text{otherwise} \end{cases}$$

j is source node
i is destination

This matrix is particularly used for representing directed graph where the value of edges can be either 1, -1 or 0 depending on the presence of edges in the graph.

for example :-



$$I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & 1 & 0 & -1 \\ 0 & -1 & +1 & 0 \end{bmatrix} \end{array}$$

The graph is stored as an adjacency matrix G[1...n, 1...n]. All the cycles begin at node 1

x[1...k-1] is a path of (k-1) distinct vertices. If x[k] is equal to zero then no vertex is assigned to x[k] & after execution, next highest no. vertex i.e not in the path is assigned to x[k].

Initially, all the vector of x are equal to x.

UNIT-III

UNIT-3
GRAPH

Graph Traversal Algorithm:-
1) Breadth First Search (BFS)
2) Depth First Search (DFS)

BFS:- It is a graph traversal algorithm that is used for searching a graph. Given a graph G = V, E & source vertex S, this algorithm systematically explores the edges of G to discover every vertex i.e reachable to S.

It computes the distance from S to reach reachable vertex

Basically, this algorithms discovers all the vertices at distance K from S before discovering any vertices at distance K+1.

Some colouring scheme is used to track the node processing criteria. Following 3 colours are used:

1) ○ White :- Initially, all the nodes in the graph are of white color which represents that all the nodes are unexplored & unreachable.

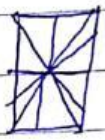
2) ▨ Grey :- This color signifies that node has been discovered but some of its neighbours may or may not be explored.

3) Black ⊗ :- This color signifies that node has been completely discovered & all of its neighbours are explored.

The data structure used in BFS is Queue.

This algorithm is also known as levelled searching Algorithm.

Algorithm of BFS :=>

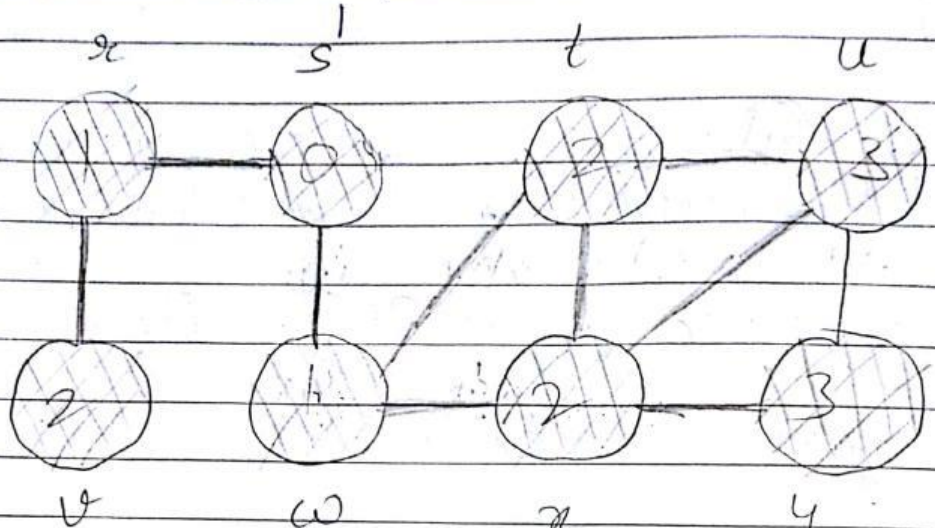


Algorithm BFS (G, S)

```
1  for each vertex v ∈ V[G] - {s}
2  color[u] := WHITE.
3  d[u] := ∞
4  π[u] := NIL
5  color[s] := GRAY
6  d[s] := 0
7  π[s] := NIL
8  Q := ϕ
9  ENQUEUE (Q, s)
10 while (Q ≠ ϕ)
11 {u := DEQUEUE (Q)
12 for each v ∈ Adj (u) in 'G'
```

```
13  { If colour[v] = WHITE
14  color[v] := GRAY
15  d[v] = d[u] + 1
16  Π[v] = u
17  ENQUEUE (Q, v) }
18  color[u] := BLACK. }
```

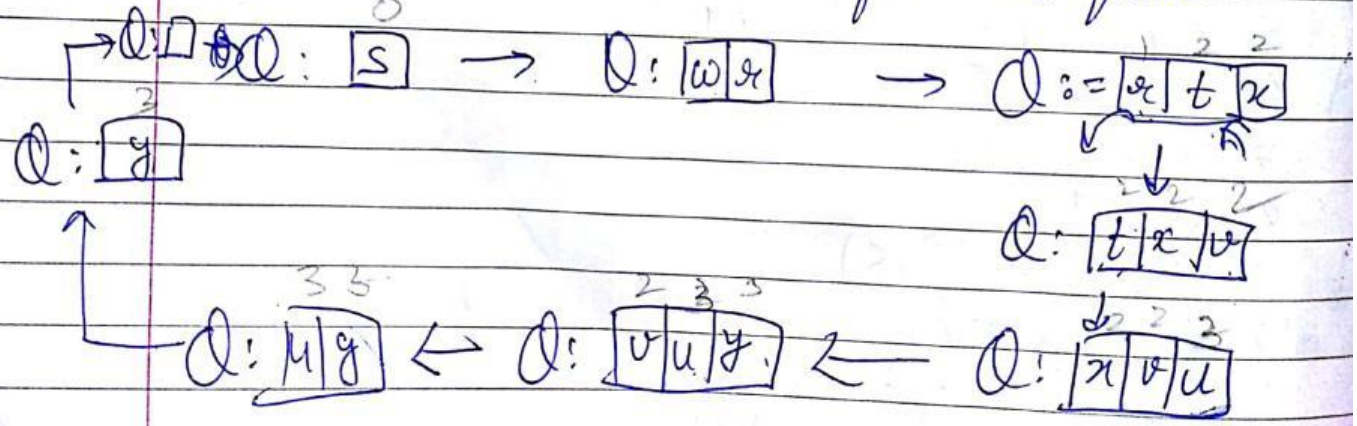
$O[V+E)$



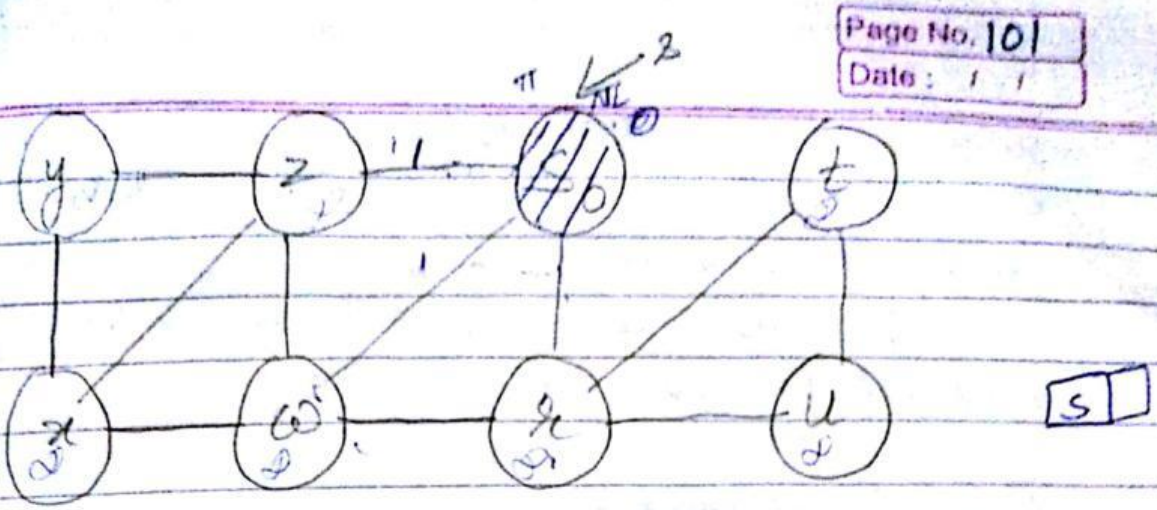
s, w, r, t, u, v, x, y

In this algorithm, G is a graph consisting of V vertices where S denotes the source node, d denotes distance from the source node, Π denotes the parent node from which a particular node will be discovered.

ENQUEUE & DEQUEUE are two procedure that are used to insert & delete an element from the queue.

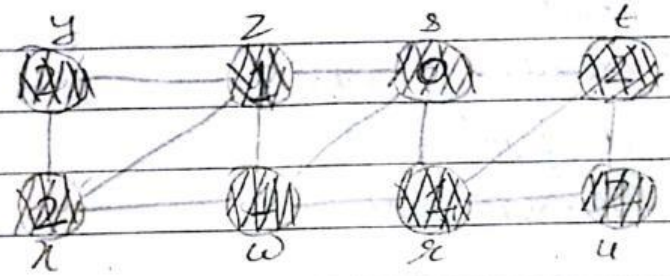


Ques:-



S,

Q := [s] Q := [s]



u := s, s, w, z, t, y, x, y

Q := [r | w | z]

Q := [s] → Q := [r | w | z] → Q := [w | z | t | u] → Q := [z | t | u]

Q := [t | u]

Q := [ ]

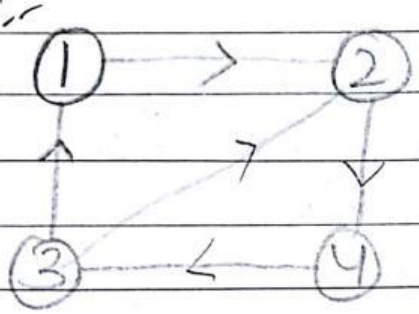
Q := [y] ← Q := [x | y] ← Q := [u | x | y] ← Q := [t | u | x | y]

Q := [s] → Q := [r | w | z] → Q := [w | z | t | u] → Q := [z | t | u | x]

Q := [y] ← Q := [x | y] ← Q := [u | x | y] ← Q := [t | u | x | y]

This matrix is particularly used for representing directed graph where the value of edges can be either 1, -1 or 0 depending on the presence of edges in the graph.

For example:-



$$I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & 1 & 0 & -1 \\ 0 & -1 & +1 & 0 \end{bmatrix} \end{array}$$

Depth First Search:- ① Topological Sort ② Strongly Connected Components — Applications

DFS is a graph traversal algorithm that explodes the edges out of the most recently discovered vertex V

that still has unexplode edges. Once, all the edges of v have been explode, the search backtrack to explore the edges leaving the vertex from which the current node was discovered.

This process continues until all the vertices that are reachable from the original source vertex have been discovered.

If any undiscovered vertex remains then this algorithm selects one of them as a new source & repeats the search from that source.

Besides, traversing the nodes in DF manner, it also tymstamps each node in two ways:-

① Discovery tym at which the node was explode
② Finishing " " " " " " completely processed

These tymstamps provide information about the structure of the graph & the depth of the graph.

DFS also uses a colouring scheme as follows:-

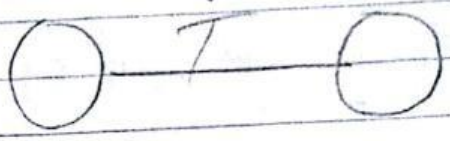
① (I) White :- This colour signifies that node is undiscovered & is yet to be processed.

② (II) GRAY:- It signifies that node has been partially discovered & some of its neighbour may or may not be explore.

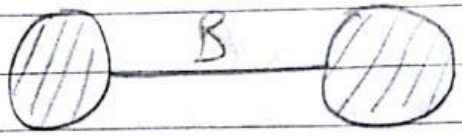
③ (III) BLACK:- This color signifies that node has been completely discovered & all its neighbours have been explore.

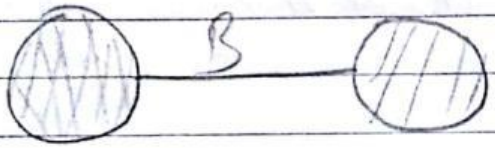
Labelling of Edges:-

① Tree Edge:- Initially all the edges in the given graph are label to be as tree edge.

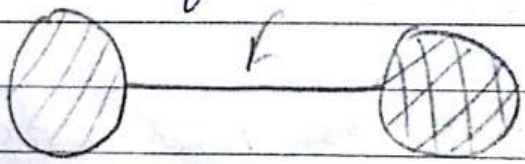


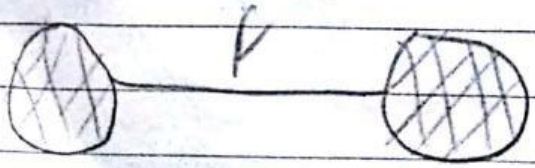
② Back Edge:- If the destination node colour is grey & the source node colour can either be grey or black then the edge is label to be as back edge.



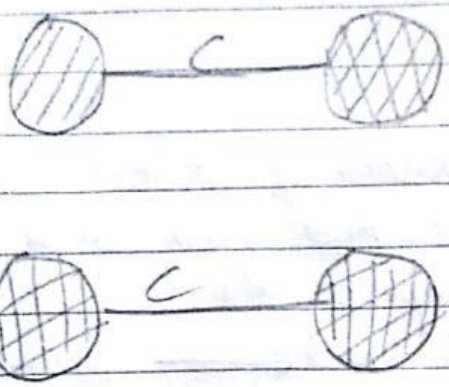


③ Forward Edge:- If source & destination nodes are of same tree i.e have originated from same sources & the color of destination node is black then the edge is label to be as forward edge





Cross edge :- If source & destination node are of from the different tree i.e have originated from different source & colour of destination node is black where source node can either be grey or black then the edge is label to be as cross edge.



UNIT-III

DAA-Lecture.

Algorithm DFS (G)

```
1  for each vertex u ∈ V[G]   // All the nodes are unexplored
2  color u := WHITE
3  π[u] := NIL
4  time := 0
5  for each value u ∈ V[G]
6  If (color[u] = WHITE)
7  DFS-VISIT (G, u)
```

Algo DFS-VISIT (G, u)

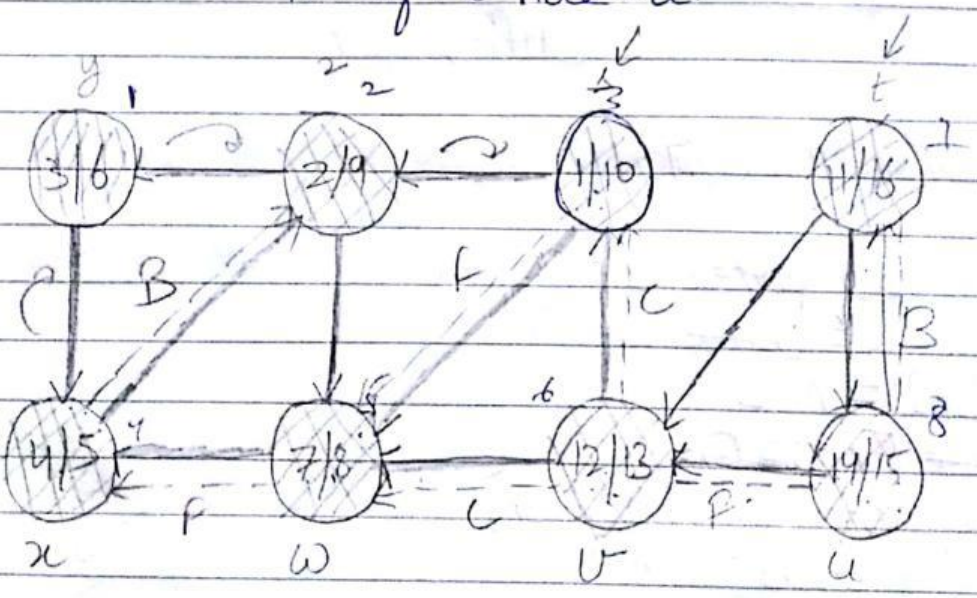
```
1  time := time + 1        // white vertex 'u' has just been discovered
2  d[u] := time
3  color[u] := GRAY
4  for each v ∈ Adj[u] in 'G'   // explore edge (u, v)
5  If (color[v] = WHITE)
```

6 $\Pi[v] := u$
7 DFS-VISIT(G, v)
8 color[u] := BLACK
9 time := time + 1
10 f[u] := time

$O(V+E)$

In this algorithm,

G = is a graph consisting of v vertices where :-
Π = denotes the parent node from which a particular node will be explore.
d[u] & f[v] = denotes the ~~discovery~~ discovery & finishing time of the node u.



s, z, y, x, w, t, v, u

Applications of DFS:-

↳ Topological Sort
↳ Strongly Connected Components.

Topological Sort ⇒ A topological sort of a directed acyclic graph ~~that~~ DAG, G = (V, E) is a linear ordering of all the vertices in such a way such that if G contains

an edge (u,v) then the node u appears before v in the ordering.

DAG can be defined as a graph having directed edges but no cyclic.

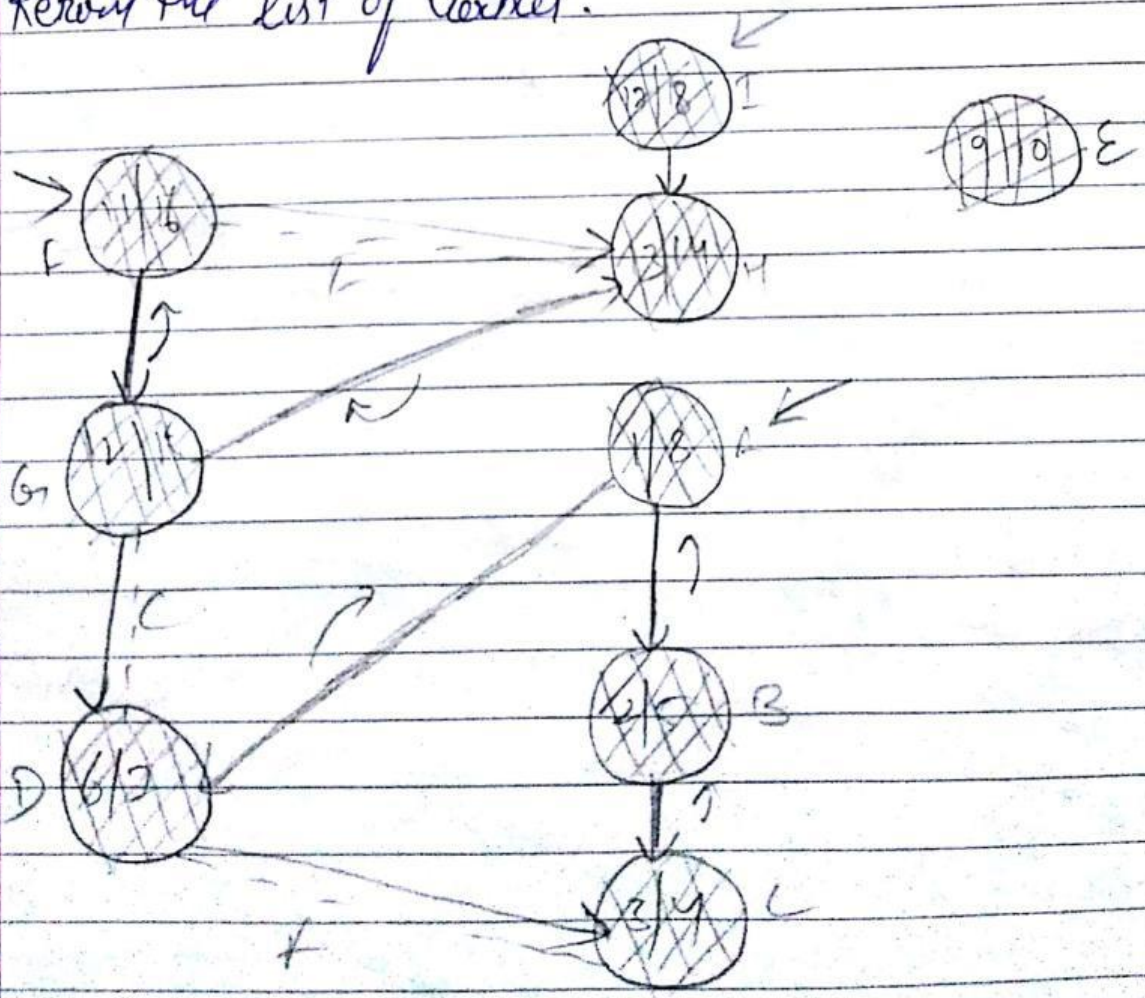
Algorithm:-

Algorithm Topological Sort (G)

1) Call DFS(G) to compute finishing times $f(v)$ for each vertex v. $O(V+E)$

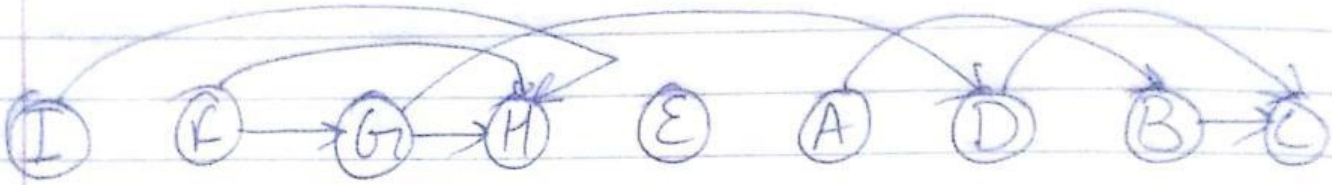
2) As each vertex is finished, insert it onto the front of list.

3) Return the list of vertices.



Page No. 110
Date / /

Now arrange all the nodes in the decreasing order of finishing time to obtain the linear ordering of vertices.



UNIT-III

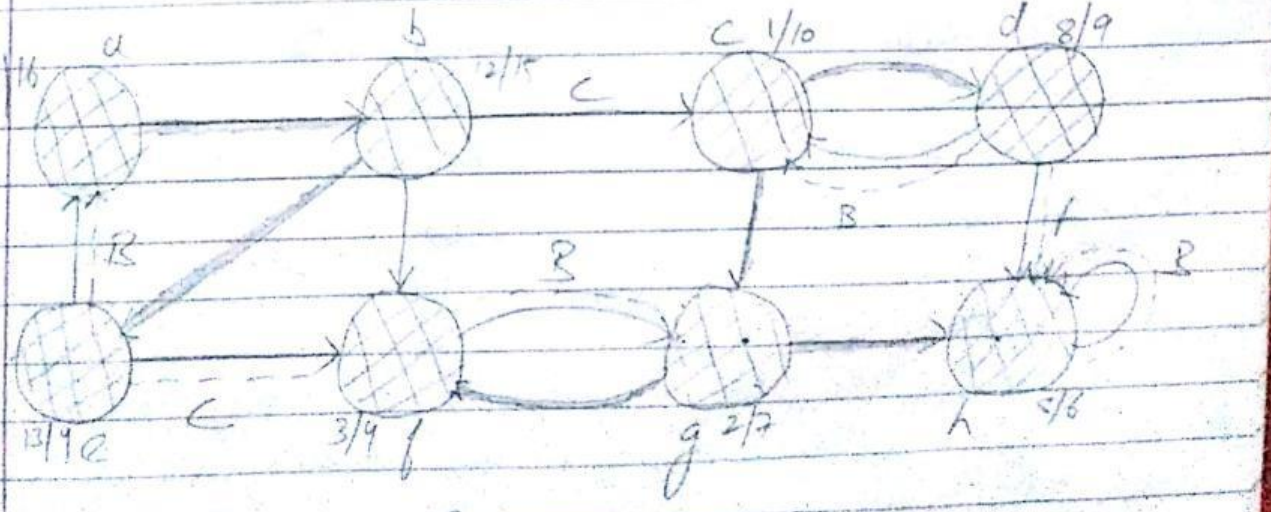
DAA - Lecture

Strongly Connected Components: ⇒ A strongly connected components of a directed graph $G=(V, E)$ is a set of vertices such that C is a subset of V & every vertices in U & V are directly reachable from each other i.e - $u \rightsquigarrow v$ & $v \rightsquigarrow u$ & C denotes the set of strongly connected components

Algorithm:-

Algo. Strongly Connected Components (G_T)

1. Call DFS (G_T) to compute finishing time $f(u)$ for each vertex 'u'.
2. Compute G_T^T. $\Theta[V+E]$
3. Call DFS(G_T^T) & consider the source vertices in the order of decreasing $f[u]$
4. Output the vertices of each-tree as a separate strongly connected component.



G_T

Step 2:- Compute the transpose of given graph:-

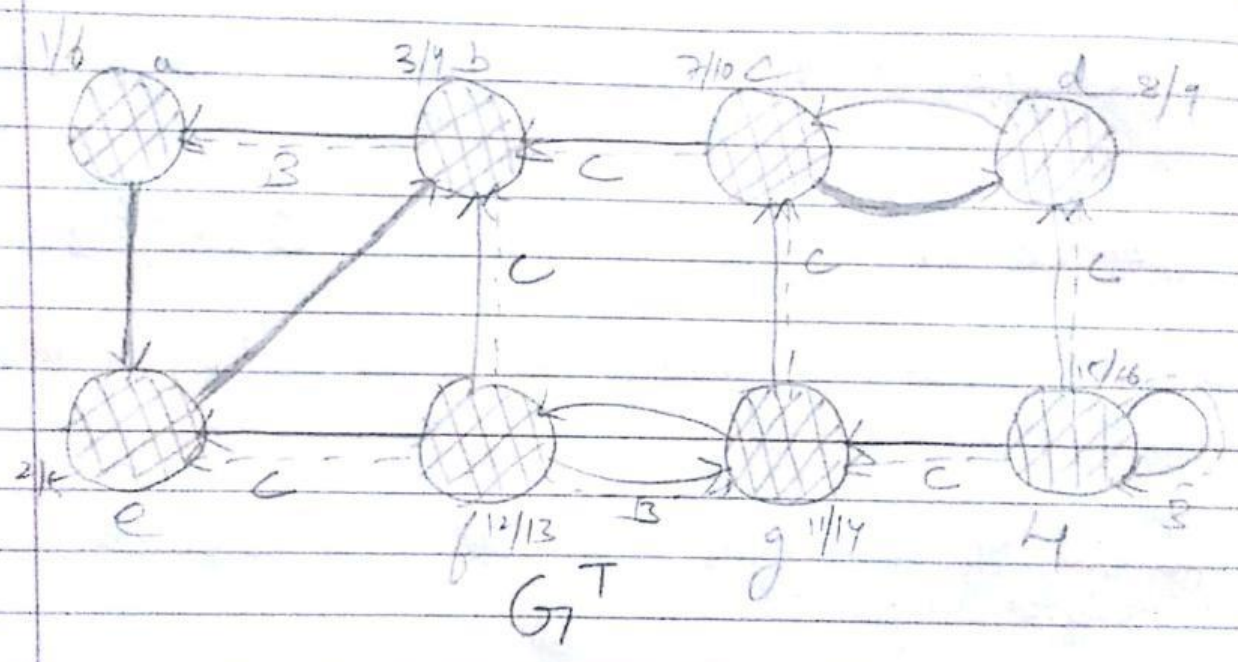
$$G_T^T = (V, E^T)$$

where $E^T = \{ v, u \mid (u, v) \in E \}$

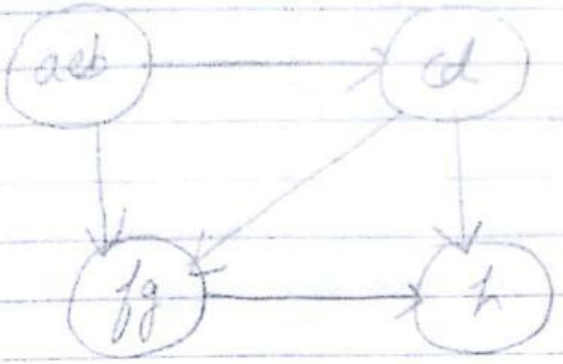
i.e E^T consists of all the edges in the graph with their direction reversed. The source vertex will be considered in the order of highest finishing time in G_T

23/10

Step 2: Soln:-



Source	Components
a	a e b
c	d
g	f g
h	h



UNIT-III

DAA-Lecture

Minimum Spanning Tree:- Tree is an acyclic graph consisting of 'V' vertices & 'E' edges. The spanning tree is defined as the tree which consists of all the vertices as that of the original tree but will contain exactly n-1 edges. A spanning tree is said to be minimum spanning tree if it satisfies the following conditions:-

1) It should be connected i.e all the nodes are reachable from each other.

2) If there are 'N' vertices, then there will be exactly N-1 edges.

3) The minimum cost edge will be added initally in the minimum spanning tree & the process is repeated till there are no cycles & exactly N-1 edges are added to the tree.

There are two methods to determine MST:-

1) Kruskal Algorithm.
2) Prim's "

Kruskal Algorithm:→ In this algorithm, the edges of the graph are considered in non-decreasing order & certain procedures are used to find MST

First Procedure:- Heapify:- This procedure is used to sort the edges in order of the increasing cost

2nd Procedure:- Parent:- This procedure puts all the vertices in different sets.

3rd Procedure:- Ajust:- It is used to rearrange all the vertices in the increasing order of their cost after the minimum cost edge is removed from heap

4th:- Find:- It returns the set of given vertex.

5th:- Union:- It is used to combine two sets into a single set.

E= is the set of edges in the graph
cost= is the weight of edges
n= is the total no. of vertices.
t= is an array that stores the value of edges of MST.

Algorithm of Kruskal Algorithm:-

Algo Kruskal (E, cost, n, t).

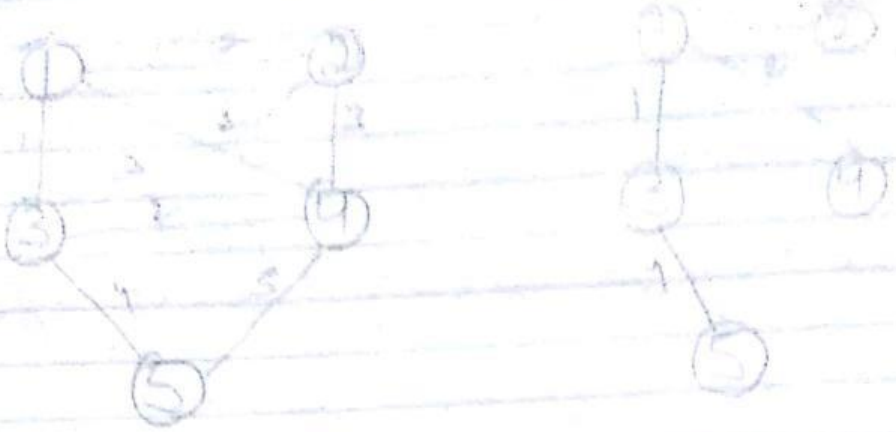
1 Construct a min-heap out of the edge cost using HEAPIFY

2 for i=1 to n.

```
3  do parent[i] := -1  // each vertex is in a different set.
4  i=0, min cost := 0
5  while ((i<n-1) && (heap not empty)) do
6  { Delete a min cost edge (u, v) from the heap & reheapify
   using ADJUST
7  j := FIND(u), k := FIND(v)
8  If (j≠k)
9  { i := i+1
10 t[i,1] := u , t[i,2] := v
11 Min Cost := min cost + cost(u,v)
12 UNION(j,k);
13 }}
14 If (i ≠ n-1) then write("No spanning tree is possible
   else return MinCost
15 }
```

O(E log V)

Example:-



1,3 = 1
1,2 = 2
1,4 = 2
3,4 = 2
3,2 = 3
2,4 = 3
3,5 = 4
4,5 = 5

Step 1:- $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3\}$, $S_4 = \{4\}$, $S_5 = \{5\}$

$u, v = \{1, 3\}$

$j := FIND(U) = FIND(1) = S_1$

$k := FIND[v] = FIND(3) = S_3$

$i = 1$

$S_1' = S_1 \cup S_3 = \{1\} \cup \{3\} = \{1, 3\}$

Step 2:- $j = FIND(U) = FIND(1) = S_1'$ $\quad k = FIND(v) = FIND(2) = S_2$

$i = 2$

$S_1'' = S_1' \cup S_2 = \{1, 3\} \cup \{2\} = \{1, 2, 3\}$

Step 3:- $(u, v) = (1, 4)$

$j = FIND(1) = S_1''$, $\quad k = FIND(4) = S_4$

$i = 3$

$S_1''' = S_1'' \cup S_4 = \{1, 2, 3\} \cup \{4\} = \{1, 2, 3, 4\}$

Step 4. $(u, v) = 3, 4$

$j = FIND(3) = S_1'''$

$k = FIND(4) = S_1'''$

It is not safe to add this edge in M.S.T.

Step 5: $(u,v) = (2,3)$
$j = FIND(2) = S_1'''$
$k = \text{"} (3) = \text{"}$
Unsafe

$(u,v) = (2,4)$
$j = FIND(2) = S_1'''$
$k = \text{"} (4) = \text{"}$
Unsafe

$(u,v) = (3,5)$
$j = FIND(3) = S_1'''$
$k = \text{"} (5) = S_3$
$i = 4$
$S_1''' = S_1''' \cup S_3 = \{1,2,3,4,5\}$

t	1	2
1	1	5
2	1	2
3	1	4
4	3	5

Min Cost = 1+2+2+4
= 9

DAA - Lecture

Prim's Algorithm:- This algorithm is used to obtain a min. cost spanning tree by building a tree i/p. The next edge to include is chosen on the basis of minimum weight. However, while adding the edge, acyclic property of the tree needs to be maintained. The resultant minimum spanning tree is built as according to min. weight of the edge.

Algorithm of prim's algorithm:-

Algorithm Prim $(E, cost, n, t)$

1. Let (k,l) be an edge of minimum cost in E
2. MinCost = $cost(k,l)$ $i = 1$

```
3.  t[i,1] := k  , t[i,2] := l
4.  for i := 1 to n  // initialize, near
5   If (cost[i,k] < cost[i,l])
6   then near[i] := k
7   else near[i] := l }
8   near[k] := near[l] := 0
9   for i = 2 to n-1 do
10  { Let j is an index such that near[j] ≠ 0 & cost[j,
    near[j]] is minimum
11. t[i,1] := j  , t[i,2] := near[j]
12  MinCost := minCost + ~~M~~cost[j, near[j]]
13  near[j] := 0
14  for k := 1 to n do  // update near
15  If (near[k] ≠ 0) && (cost[k, near[k]] > cost[k,j])
16  then near[k] := j }
17  return MinCost }
```

$O[n^2]$

E = denotes the no. of edges in the graph G with n no. of vertices

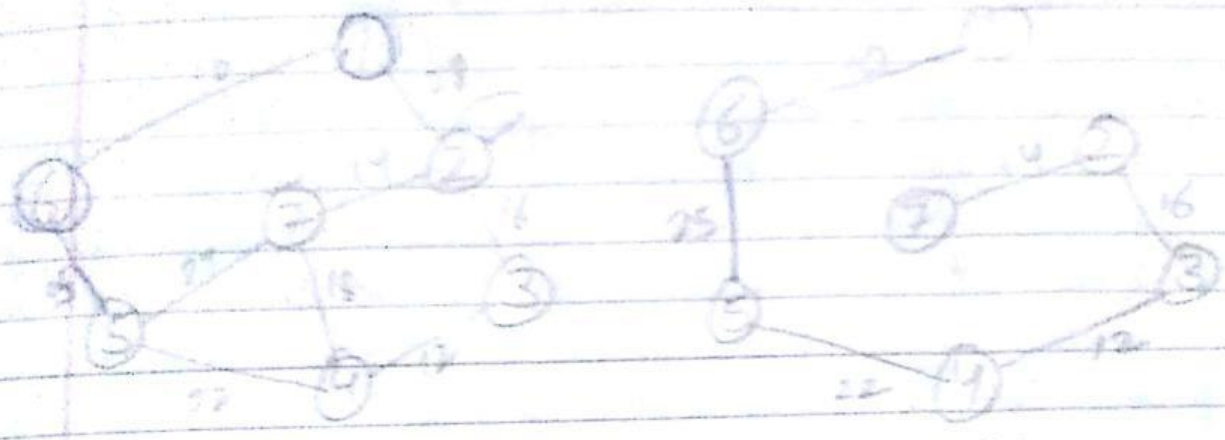
cost = denotes the weight of the edges.

t = is an array i.e used to maintain the edges of minimum spanning tree

t	1	2
1	1	6
2	5	6
3	4	5
4	3	4
5	2	3
6	7	2

MinCost = 10+25+22+12+16+14
= 99

Example:



Step 1:- $(k, l) = (1, 6)$

Initialization Step

cost[1,2] < cost[1,6]

	2	0
	3	4
cost[i,2]	4	7
	5	8
	6	∞
	7	8

$i=1 \Rightarrow$ cost[1,1] < cost[1,6] $\Rightarrow \infty < 10$ X $\Rightarrow$ near[1] = 6

$i=2 \Rightarrow$ $\Rightarrow$ near[2] = 1

$i=3 \Rightarrow$ $\Rightarrow$ n[3] = 6

$i=4 \Rightarrow$ $\Rightarrow$ n[4] = 6

$i=5 \Rightarrow$ X $\Rightarrow$ n[5] = 6

$i=6 \Rightarrow$ X $\Rightarrow$ n[6] = 1

$i=7 \Rightarrow$ $\Rightarrow$ n[7] = 8

$\Rightarrow$ near[1] = near[6] = 0

Step 2:- $i = 2$

$j = 2, 3, 4, 5, 7$

Cost[j, near[j]] :-

$j=2$ cost[2,1] = 28

$j=3$ cost[3,6] = ∞

$j=4$ cost[4,6] = ∞ near[5] = 0

$j=5$ cost[5,6] = 25

$j=7$ cost[7,6] = ∞

[5,7] [1,2]

Page No.
Date

k = 2,3,4,7 update near
cost[k, near[k]] > cost[k, j]

k=2 cost[2,1] ⇒ cost[2,5] ⇒ 28 > ∞, n[2] = 1
3 cost[3,6] > cost[3,5] ⇒ ∞ > ∞, n[3] = 6
4 cost[4,6] > cost[4,5] ⇒ ∞ > 22, n[4] = 5
7 cost[7,6] > cost[7,5] ⇒ ∞ > 24, n[7] = 5

Step-3 i = 3
j = 2,3,4,7

cost[j, near[j]]

j=2 cost[2,1] = 28
j=3 cost[3,6] = ∞
j=4 cost[4,5] = 22
j=7 cost[7,5] = 24
for j = 4, near[4] = 0
k = 2,3,7
c[k, near[k]] > c[k,j]
k=2 c[2,1] > c[2,4] ⇒ n[2] = 1
3 c[3,6] > c[3,4] ⇒ n[3] = 4
7 c[7,5] > c[7,4] ⇒ n[7] = 4

Step 4 i = 4, j = 2,3,7
cost[j, near[j]]

j = 2 cost[2,1] = 28
j = 3 cost[3,4] = 12 ✓
j = 7 cost[7,4] = 18
j = 3, near[3] = 0
k = 2,7

$c[k, near[k]] > c[k, j]$

$k=2 \Rightarrow c[2,1] > c[2,3] = near[2] = 3$
$k=7 \Rightarrow c[7,4] > c[7,3] = near[7] = 4$

Step 5:- $i = 5$
$j = 2, 7$
$cost[j, near[j]]$
$j=2$ $cost[2,3] = 16$ ✓
$j=7$ $cost[7,4] = 18$
$j = 2$, $near[2] = 0$
$k = 7$
$k=7 \Rightarrow c[7,2] > c[7,2] \Rightarrow 18 > 14 \Rightarrow near[7] = 2$

Step 6:- $i = 6$
$j = 7$
$cost[j, near[j]]$
$j=7$ $cost[7,2] = 14$
$j=7$, $near[7] = 0$

UNIT-III

DAA-Lecture

Single Source Shortest path:- The single source shortest path algo. is used to find out the shortest distance of all the nodes of a graph from a given source vertex S.

In this prbm, a weighted graph $G = (V, E)$ & the source vertex V_0 is provided & the prbm is to determine the shortest path from V_0 to the remaining vertices of G. There are 2 algorithm for SSSP:-

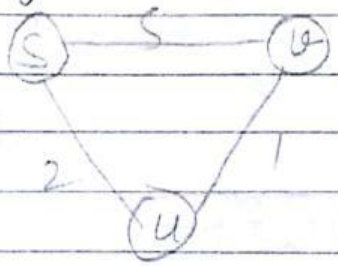
1) Dijkstra's Algo. (+ve)
2) Bellman Ford " (+ve & -ive)

a) INITIALISE-SINGLE SOURCE (s)
1 d[s] ← 0
2 d[v] ← ∞

b) ALGORITHM RELAXATION (u, v, w)
1 If d[v] > d[u] + w[u, v]
2 Then d[v] ← d[u] + w[u, v]

Here d[v] = distance of node v from source S

Here w = denotes the weight of the edges.
s = source node
v = destination node
u = any intermediate node

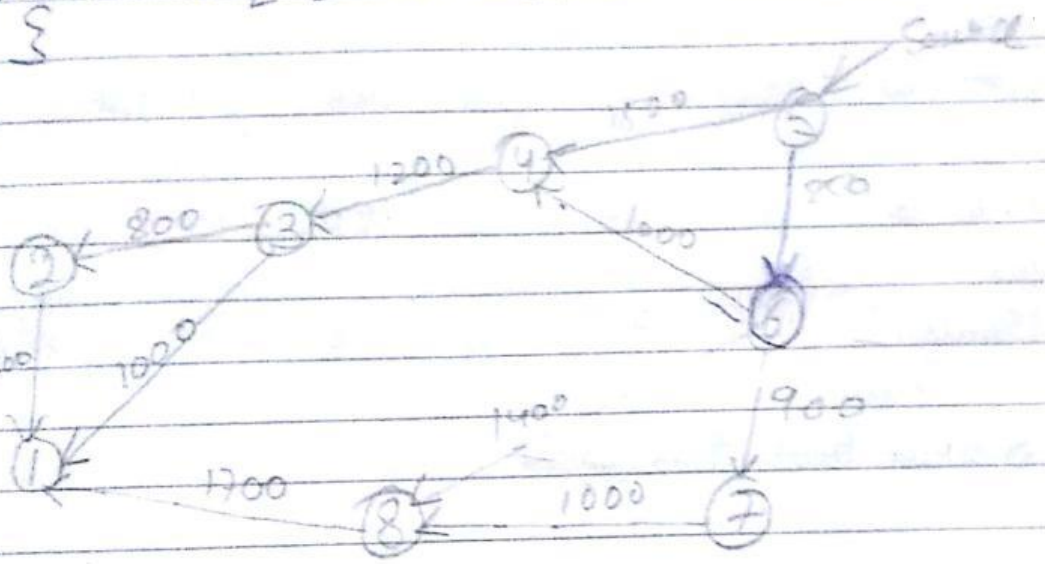


Dijkstra's Algorithm :-

Algorithm Dijkstra's (v, cost, dist, n) $O(n^2)$

1 for i = 1 to n do
2 { s[i] := false, dist[i] := cost[v, i] }
// initialise S.
3 s[v] := true, dist[v] = 0.0
// Put v in S

4 for num := 2 to n-1 do // Determine n-1 paths from v
5 { Choose 'u' among those vertex not in 'S'
such that d[u] is minimum
s[u] = true // Put 'u' in S
6
7 for (each 'w' adjacent to 'u' with s[w] := false)
do // update distance
8 If (d[w] > d[u] + cost[u,w]) then // Relaxation of edges
9 d[w] := d[u] + cost[u,w]
10. }



In this algorithm,
v = source node
u = intermediate node
cost = the weight of the edges
d = Distance ~~node~~ from source node
n = total no. of vertices.

'S' defines the set which is initially empty & the vertices are choosen in set as according to the shortest distance

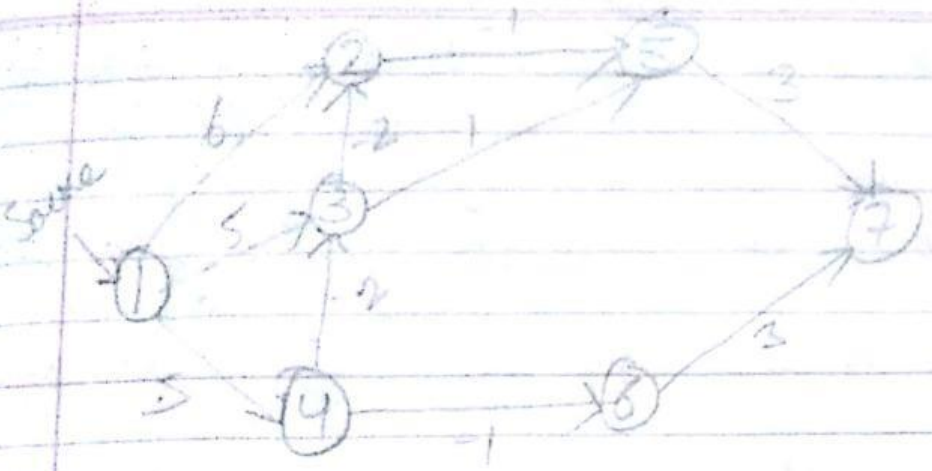
Destinations	S	Vertex Selected	1	2	3	4	5	6	7	8
1			∞	∞	∞	1500	0	250	∞	∞
2										
3										
4										
5										
6										
7										

(2) Bellman ford Algorithm :-

Algorithm Bellmanford (v, cost, dist, n)

```
1  for i := 1 to n do
2  { d[i] := cost[v,i] }
3  for k := 2 to n-1 do
4  { for each 'u' such that u≠v & v has atleast one
     incoming edge do
5  { for each (i,u) in the graph do
6  { if (d[u] > d[i] + cost[i,u]
     then d[u] := d[i] + cost[i,u]
7. }}}
```

$O(n^3)$



This algorithm is used to find the shortest distance from the single source vertex to all the destination nodes.

However, this algorithm is preferably used for the graph having negative edge length.

But their is a condition that their should be no cycle of -ve length.

If their is a -ive length cycle, this algorithm reduces no solution.

When their are no cycles of -ve length, then the shortest path b/w any two vertices of an 'n' vertex graph with atmost (n-1) edges is produced.

No of edges "k"	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

* In this algorithm,

n = total no. of vertices
d = distance from source
v = destination node
cost = weight of edges
a = denotes the total no. of intermediate edges i.e used for SD computations.

UNIT-III

DAA-Lab

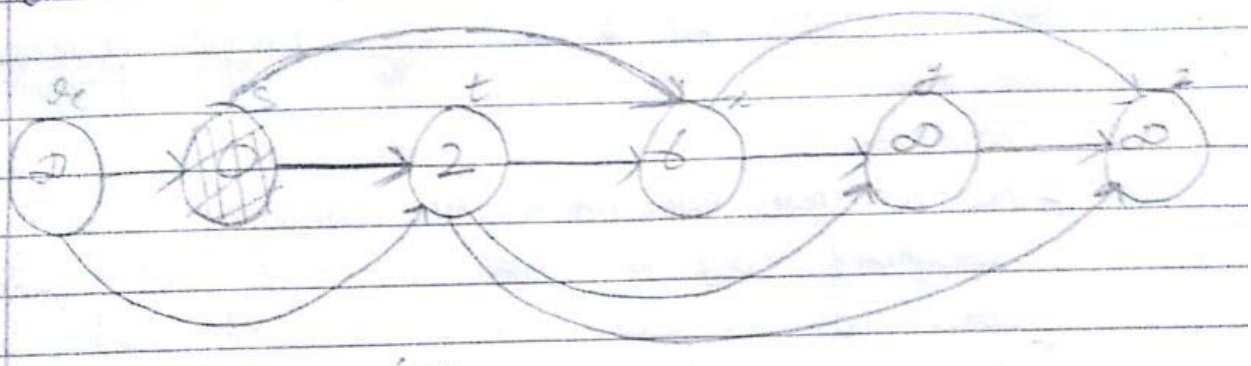
Single Source Shortest Path in DAGs:- In this algorithm the edges of a weighted DAG (Directed Acyclic Graph) $G=(V, E)$ are relaxed & then according to topological sort of its vertices, the shortest path is computed from a single source. The algorithm starts by a topologically sorting the DAG

to impose a linear ordering on the vertices. As we process each vertex, we relaxed each edges that ~~leaves~~ leaves the vertex.

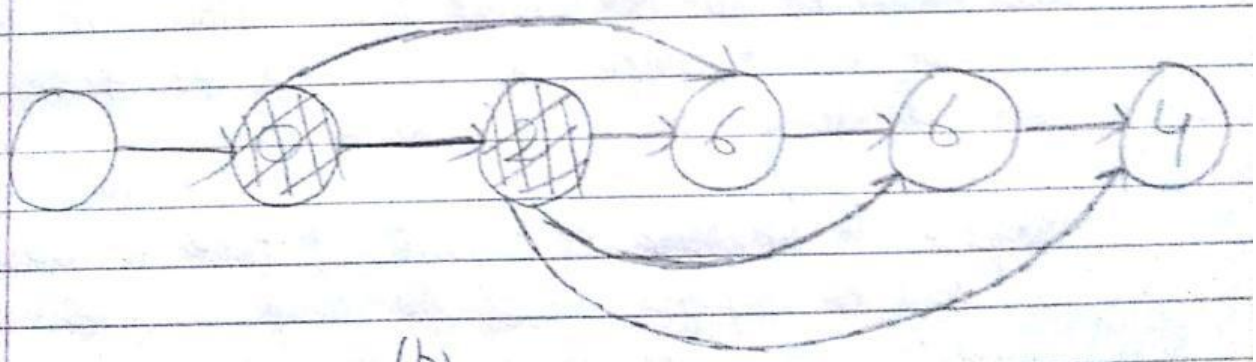
Algorithm:1

Algorithm DAG-SHORTEST-PATHS(G, w, s)

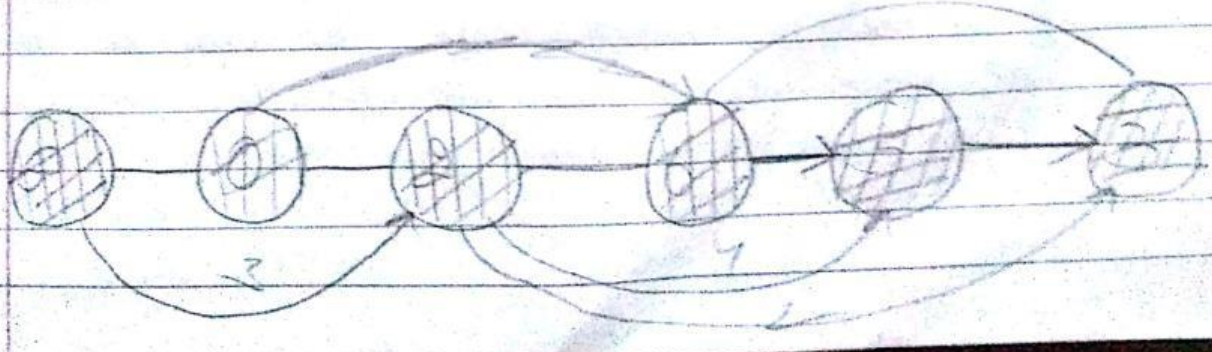
1. Topologically sort the vertices of 'G' $\Theta(V+E)$
2. INITIALISE-SINGLE-SOURCE(G, s)
3. for each vertex u, taken in topologically sorted order
4. for each vertex $v \in Adj[u]$ in graph 'G'
   do RELAX(u, v, w)



(a)



(b)



All-pair shortest path:- This prbm is used to find the shortest path b/w all the pair of vertices in a graph. A weight adjacency matrix is computed for ach 'n' vertex directed graph $G = (V, E)$ such that :-

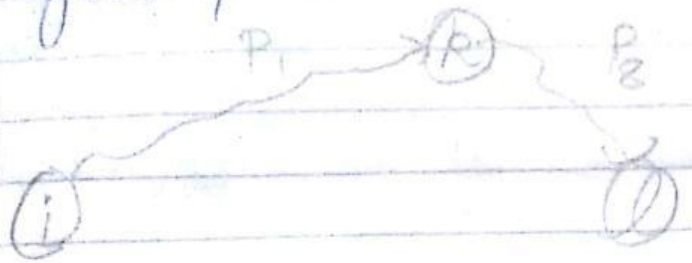
$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \ \& \ (i,j) \in E \\ \infty & \text{if } i \neq j \ \& \ (i,j) \notin E \end{cases}$$

The algorithm i.e used to compute all pair shortest path is known as Floyd - Warshall Algorithm.

Floyd-Warshall Algorithm:- This algorithm is a dynamic programming algorithm i.e used to solve all pair shortest path prbm on a directed graph $G = (V, E)$. In this, the -ive weight edges may be present but their should be no - ve weight cycle. Otherwise, this algorithm will produce no soln. It can be solved using following steps :-

Step 1:- The structure of a shortest path:- The shortest path can be defined with the help of intermediate vertices such that if i is the source node & j is the destination node then any no. of intermediate vertices may be used to determine the shortest path b/w them.

Step 2:- A recursive sol^n to the all-pair shortest path prbm:- A recursive formulation may be defined for the given prbm & can be described as:-



$$d_{ij} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}, d_{ik} + d_j) & \text{if } k \geq 1 \end{cases}$$

Step 3:- Computing the shortest path weights:- A matrix of size $n \times n$ is used to define the distance matrix of shortest path weights w & the algorithm for the following can be defined as following:-

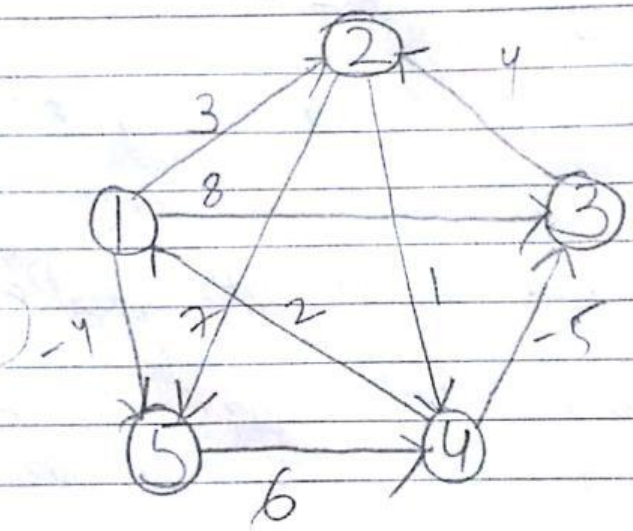
Algorithm:-

Algorithm FLOYD-WARSHALL (w)

```
1  n := rows[w]                                  Θ(n^3)
2  D^(0) := w
3  for k := 1 to n
4  let D^(k) = d_ij^(k) be a new n*n matrix
5  for i := 1 to n
6  for j := 1 to n
7  d_ij^(k) = min (d_ij, d_ik + d_kj)
8  return D
```

Step 4:- Constructing a shortest path:- A Predecessor or Parent matrix (Π) is used to determine the parent node of all the vertices & can be defined as:-

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } i=j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ \& } w_{ij} < \infty \end{cases}$$



$$d^{(0)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & \infty & -5 & 0 & \infty \\ 5 & \infty & \infty & \infty & 6 & 0 \end{array}$$

$$\Pi^{0} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & N & 1 & 1 & N & 1 \\ 2 & N & N & N & 2 & 2 \\ 3 & N & 3 & N & N & N \\ 4 & 4 & N & 4 & N & N \\ 5 & N & N & N & 5 & N \end{array}$$

$$d^{(1)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & \infty & \infty & \infty & 6 & 0 \end{array}$$

$$\Pi^{0} \rightarrow \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & N & 1 & 1 & N & 1 \\ 2 & N & N & N & 2 & 2 \\ 3 & N & 3 & N & N & N \\ 4 & 4 & 1 & 4 & N & 1 \\ 5 & N & N & N & 5 & N \end{array}$$

$d^{(2)}$ =

	1	2	3	4	5
1	0	3	-1	4	-4
2	∞	0	∞	1	7
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$\pi^{(2)}$ =

	1	2	3	4	5
1	N	1	4	2	1
2	N	N	N	2	2
3	4	3	N	2	1
4	4	1	4	N	1
5	N	N	N	5	N

$d^{(3)}$ =

	1	2	3	4	5
1	0	3	-1	4	-4
2	∞	0	∞	1	7
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$\pi^{(3)}$ =

	1	2	3	4	5
1	N	1	4	2	1
2	N	N	N	2	2
3	4	3	N	2	1
4	4	3	4	N	1
5	N	N	N	5	N

$d^{(4)}$ =

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\pi^{(4)}$ =

	1	2	3	4	5
1	N	1	4	2	1
2	4	N	4	2	1
3	4	3	N	2	1
4	4	3	4	N	1
5	4	3	4	5	N

$d^{(5)}$ =

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\pi^{(5)}$ =

	1	2	3	4	5
1	N	3	4	5	1
2	4	N	4	2	1
3	4	3	N	2	1
4	4	3	4	N	1
5	4	3	4	5	N