UNIT - V

Graphs

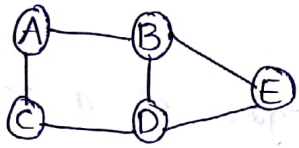
* A graph is a non-linear data structure in which the elements are arranged in the form of random order. A graph is a collection of vertices and edges

*Usually A graph can be represented as $G = (V, E)$

where V = collection of vertices (or) nodes

E = collection of edges

Eg:-



V = A, B, C, D, E

E = {(A,B), (A,C), (B,D), (B,E), (C,D), (D,E)}

Terminology of graphs:-

vertex:-

In a graph every individual element (or) node can be called as a vertex

Edge:- In a graph, a connection link between a pair of vertices is known as an edge.

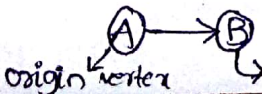
End points (or) End vertices:-

In a graph if there is an edge between a pair of vertices then those vertices are said to be end points (or) end vertices of that edge.

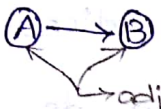
Origin and Destination vertices:-

In a graph if there is a directed edge between a pair of vertices then the starting vertex of the edge can be called as origin vertex and the last vertex of an edge can be called as destination vertex.

Eg:-



Adjacent vertices:- In a graph if there is an edge between the vertices A and B then A and B vertices are said to be adjacent vertices.



adjacent vertices.

Degree of a vertex:- In a graph, the total no of edges are connected to a vertex is said to be degree of that vertex.

* In a directed graph the degree of a vertex can be measured in two ways i) In-degree of a vertex
ii) out-degree of a vertex.

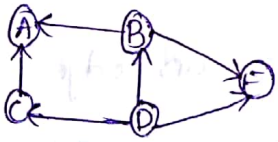
In-degree of a vertex:-

In a graph the total no of edges are coming to a vertex is known as in-degree of that vertex.

Out-degree of a vertex:-

In a graph the total no of edges are going outside from a vertex is known as out-degree of that vertex

Eg:-

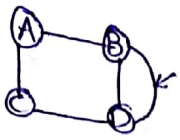


vertex	In-degree	out-degree.
A	2	0
B	1	2
C	1	1
D	0	3

Loop (or) self Loop:- In a graph, if an edge contains same end-points then such edge can be called as loop (or) self loop.

Eg:- 

Parallel edges:- In a graph if a pair of vertices are having more than one edge then those edges are said to be parallel edges.

Eg:- 

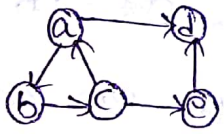
← Parallel edges

Path:- A sequence of consecutive edges from one vertex to another vertex in a graph is said to be a path.

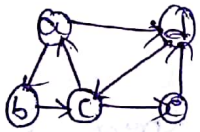
A-D ⇒ A-B-D (or) A-C-D.

Closed path:-

In a graph if a path contains same end points then such path can be called as a closed path.

Eg 

a-b-c-a is closed path

Cycle:- In a graph every closed path is known as a cycle. Eg:- 

a-b-c-a is a cycle
d-c-e-d is a cycle
a-d-c-a is a cycle
a-b-c-e-d-c-a is a cycle

Types of graphs:-

There have several graphs

These are several graphs and among them the graphs are classified into based on the frequently usage. They are.

1. NULL graph
2. Trivial graph
3. Regular "
4. Sub "
5. undirected "
6. directed "
7. Mixed "
8. weighted "
9. Simple "
10. Multi "
11. Complete "
12. Connected "

NULL Graph:- In a graph if there are no more edges between the vertices then such graph can be called a a NULL graph.

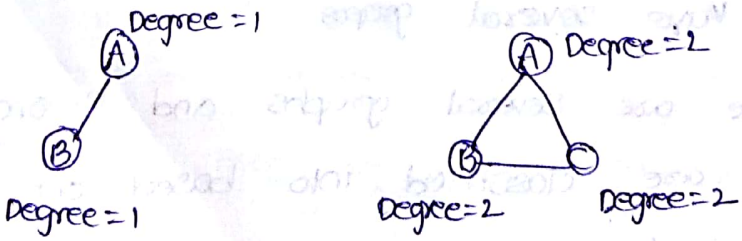
Eg:- (A) (B) (C)

Trivial Graph:- In a graph if there is only one vertex then such graph can be called as a trivial graph.

Eg:- (A)

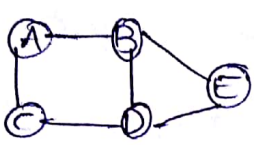
Regular Graph:- In a graph every vertex has same degree then such graph can be called as a regular graph.

Eg:-

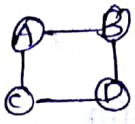


ub-graph:- In a graph a part of the graph can e called as a sub-graph.

Eg:-



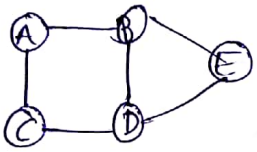
graph



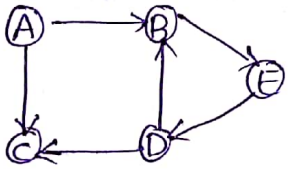
sub graph.

ndirected-graph:- In a graph, all the edges do not ave directions then such graph can be called as an ndirected graph.

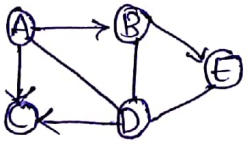
Eg:-



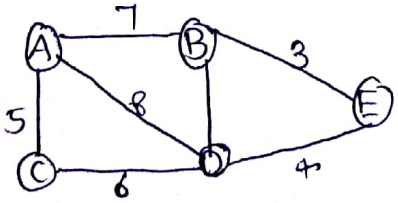
Directed graph:- In a graph, if all the edges have directions then such graph can be called as directed graph



Mixed graph:- In a graph, if some edges have directions and some edges do not have directions then such graph can be called as a mixed graph.

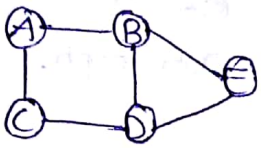


weighted graph:- In a graph all the edges are assigned with some weights (or) values then such graph can be called as weighted (or) label graphs

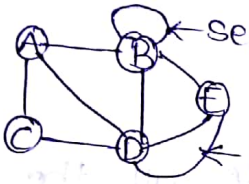


Simple graph:- In a graph, if there is no self loop (or) parallel edges then such graph can be called as simple graph.

Eg:-

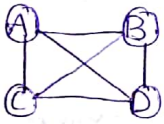


Multi-graph:- In a graph, if there is either self loop (or) parallel edges (or) both then such graph can be called as a multi-graph.

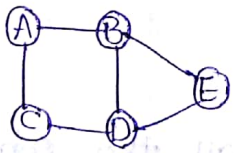


Complete graph:- In a graph, if every vertex is adjacent to other vertices then such graph can be called as a complete graph.

Eg:-



Connected graph:- In a graph, if every pair of vertices is having a path then such graph can be called as a connected graph.



Graph Representation:-

Usually the graphs are represented in two ways

1. adjacency matrix representation
2. Adjacency list representation.

Adjacency Matrix representation:- An adjacency Matrix is a square matrix which stores the value either '0' (or) '1'.

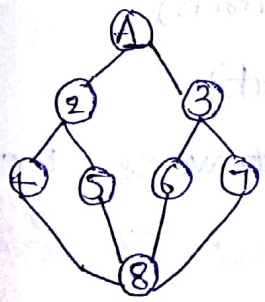
* In the adjacency matrix, if there is an edge from one vertex to another vertex in the graph, then 'one' will be stored otherwise '0' will be stored at a specific position.
* If a graph contains 'n' no.of vertices then the adjacency matrix contains nxn no.of rows & columns

$$Adj_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

where i represents row

j represents column

Eg:- Graph



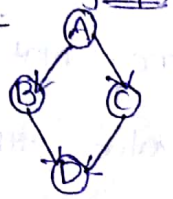
⇒

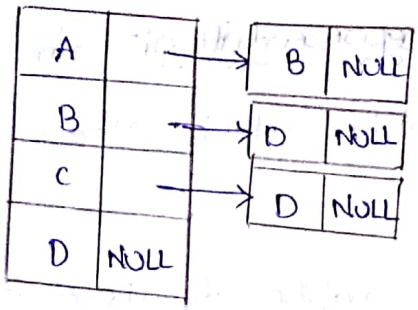
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	1	0	0	0
3	1	0	0	0	0	1	1	0
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	1	0	0	0	0	1
8	0	0	0	1	1	1	1	0

Adjacency List Representation:-

In this representation, the adjacency vertices of a vertex x can be represented in the form of single linked list.

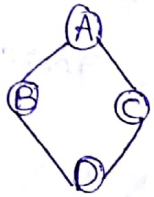
Eg:- graph

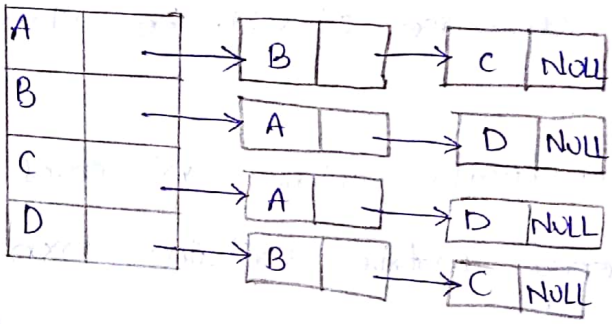




adjacency list.

Eg:-





Graph Traversal:-

The process of visiting and displaying every vertex in a graph is known as graph traversal.

The graph traversal can be performed using two techniques they 1) BFS (Breadth first search)
2) DFS (Depth first search)

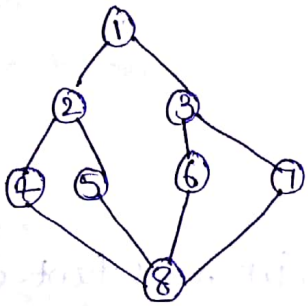
BFS :- this technique uses queue to traverse the given graph.

* In this we first visit any vertex from the graph and then its adjacent unvisited vertices will be visited
* Among the no. of adjacent visited vertices we select one visited vertex and visit its adjacent unvisited

vertices and so on.

* this process will be continued until all the vertices are visited in the given graph.

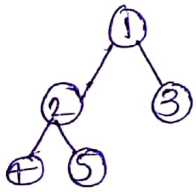
Eg:-



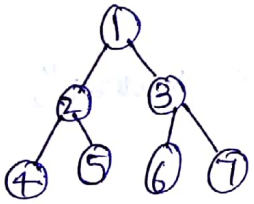
Step (1):- Visit the vertex (1) as the starting vertex.

step (2):- vertex (1) has two unvisited vertices (2) & (3) so that visit both

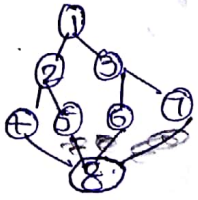
step(3):- vertex (2) has two unvisited vertices (4) & (5) so that visit both



Step(4):- vertex (3) has two unvisited vertices (6) & (7) so that visit both



Step(5):- vertex (4) has only 1 unvisited vertex that is vertex (8) so that visit it.



Step(6):- vertex (5) does not have any unvisited vertices

Step(7):- vertex (6) does not have any unvisited vertices

Step(8):- vertex (7) does not have any unvisited vertice

Step(9):- vertex (8) does not have any unvisited vertice

Program to Implement BFS:-

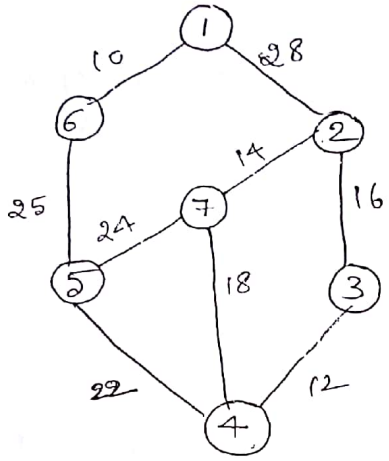
```
# Include <iostream.h>
# include <conio.h >
int adj[10][10], queue[10], visited[10], i,j,k,n,v, front=0, rear=
void main( )
{
clrscr( );
cout<<" Enter the no.of vertices :";
cin>>n;
cout<<"\n Enter the data for the adjacency matrix\n";
for(i =1; i<=n; i ++)
 {
 for (j=1; j<=n; j++)
 {
 cin >>adj [i] [j];
 }
}
cout<<"\n Enter the initial vertex:";
cin >>v;
cout<<"\n Enter the visited vertex order is :";
cout <<v<<" ";
visited [v] = 1;
k=1;
while(k< n)
{
for (j=1 ; j<=n; j++)
```

```
{
if (adj[v][j]!=0 && visited[j]!=1)
  {
   rear++;
   queue[rear]=j;
  }
  front++;
  v=queue[front];
  cout<<v<<"  ";
  visited[v]=1;
   k++;
  }
getch();
 }
```

Minimum Spanning Trees :-

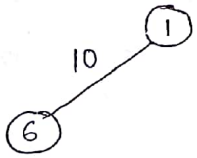
Given a connected weighted graph G it is often desired to create a Spanning Tree T for G such that the sum of the weights of the tree edges in T is as small as possible. Such a tree is called a Minimum Spanning Tree

Krushkal's Algorithm:-

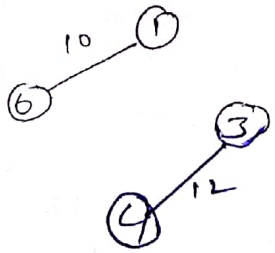


Step 1:- Arrange the weights of all the edges in ascending order 10, 12, 14, 16, 18, 22, 24, 25, 28.

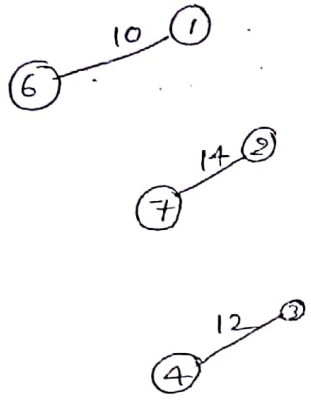
Step 2:- Select the minimum edge from the sorted list ie; edge (1,6)



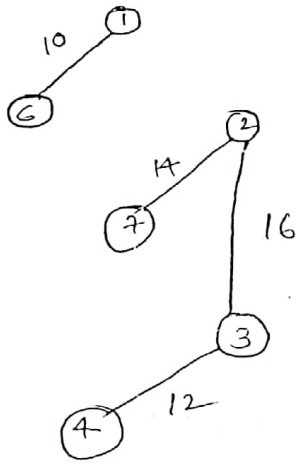
Step 3:- Select the next minimum edge from the sorted list ie; edge (3,4)



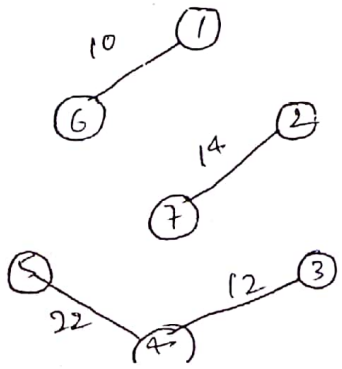
Step 4:- Select the next minimum edge from the sorted list ie; edge (2,7)



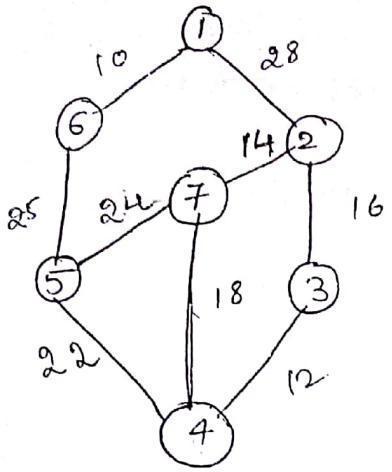
Step 5:- Select the next minimum edge from the sorted list ie edge (2,3)



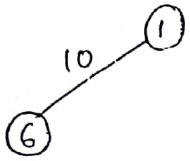
Step 6:- Select the next minimum edge from the sorted list ie edge (7,4) By placing the edge (7,4) a loop is formed so reject the edge (7,4) and Select the next minimum edge ie edge (5,4)



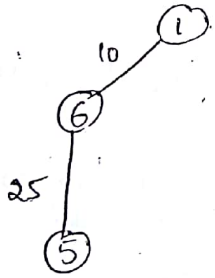
3. Algorithm:-



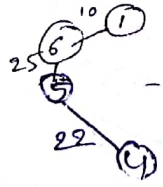
Step 1:- Select the minimum edge from the graph ie; edge (1, 6)



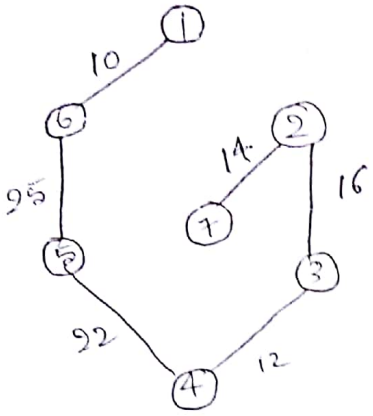
Step 2:- The adjacent edge for vertex ① is edge (1, 2) with cost 28 and the adjacent edge for vertex ⑥ is edge (6, 5) with cost 25 select the minimum edge from the above two edges ie ; edge (6, 5)



Step 3:- The adjacent edge for vertex ① is edge (1, 2) with cost 28 and the adjacent edge for vertex ⑤ is edge (5, 4) with cost 22 select the minimum edge from the above two edges ie ; edge (5, 4)



Step 4:- Select the next minimum edge from the sorted list is edge (5,7) it is rejected and the next minimum edge (5,6) is selected.



The Minimum cost of the spanning tree is

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

Algorithm :-

MST-KRUSKAL(G, W)

1) $A \leftarrow \emptyset$

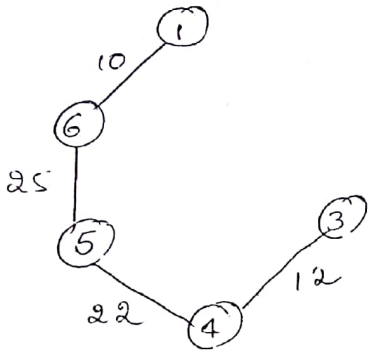
2) for each vertex $V \in V[G]$
do MAKE-SET(V)

3) Sort the edges of E into increasing order by weight W

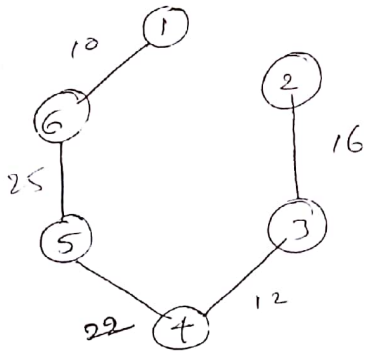
4) for each edge $(u,v) \in E$ taken in increasing order
do if FIND-SET(u) $\neq$ FIND-SET(v)
then $A \leftarrow A \cup \{u,v\}$
UNION(u,v)

5) return A

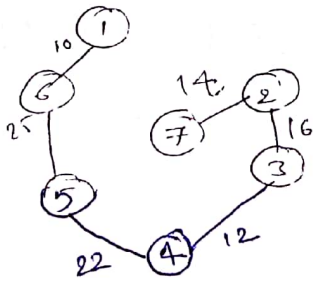
The adjacent edge for vertex ① is edge (1,2) s
th cost 28 and the adjacent edge for vertex ④ irce
edge (4,3) with cost 12 and also edge (4,7)
th cost 18 so select the minimam edge th
rom the above edges ie edge (4,3) ces
e
que
1
nd



tep5:- The adjacent edge for vertex ① is edge (1,2)
vith cost 28 and the adjacent edge for vertex ③
s edge (3,2) with cost 16, select the minimum
dge from the above edges ie; edge (3,2)



Step6:- The adjacent edge for vertex ① is edge (1,2)
vith cost 28 and the adjacent edges for vertex ② is
edge (2,7) with cost 14, select the minimum
dge from the above edges ie edge (2,7)



Minimum costs =
10+25+22+12+16+14
= 99

Algorithm :-

```
Prims_Algorithm(int graph[u][v])
{
 int parent[v]; //Array to store
 int key[v];          constructed MST
 bool mstset[v]; //to represent set of vertices included
                                        in MST
 for(int i=0; i<v; i++)
  key[i]=INT_MAX, mstset[i]=false;
  key[0]=0;
  Parent[0]=-1;
  for(int count=0; count<v-1; count++)
   {
    int u=minkey(key, mstset);
    mstset[u]=true;
    for(int v=0; v<V; v++)
    if(graph[u][v] && mstset[v]==false && graph[u][v
                                    <key[v])
    Parent[v]=u, key[v]=graph[u][v];
   }
    Print MST(parent, V, graph)
  }
```

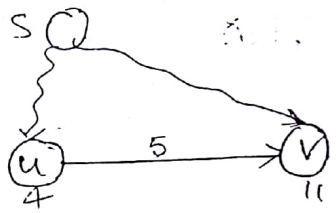
Dijstra's Single Source shortest-path Algorithm

Dijstra's Algorithm is named after a Dutch computer scientist and Dijstra is a greedy algorithm that solves the Single-source shortest path problem for a directed graph G=(V,E) with non-negative edge weights

ra's Algorithm maintains a set S of vertices
se final shortest path weights from the source
have already been determined.

The Algorithm repeatedly selects the vertex with
he minimum shortest path estimate and relaxes
ll the edges from that vertex. The single source
shortest path algorithm is based on a technique
known as relaxation. The process of relaxing
n edge (u,v) consist of testing whether we can
mprove the shortest path from u to v found
ofar going through u.

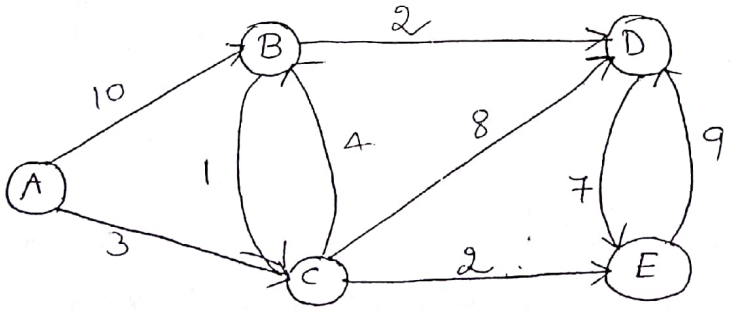
Algorithm for relaxing an edge:-



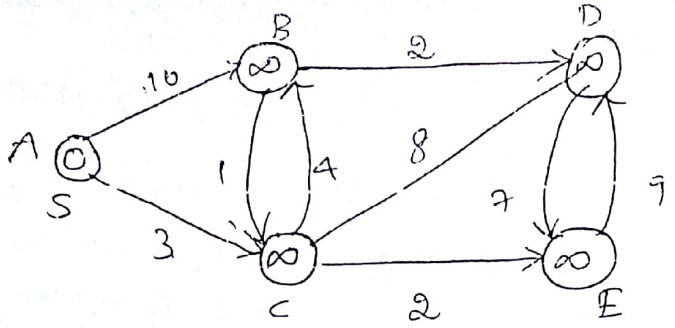

```
Relax(u,v,w)
{
 if  d[v] > d[u] + w(u,v)
 then  d[v] ← d(u) + w(u,v)
   π[v] ← u;
}
```

ρ

Ex:-



Step 1:- Initialize the minimum value to 0 and remaining edges to ∞

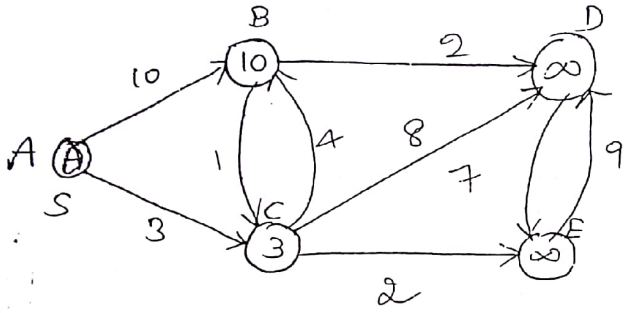


Q :	A	B	C	D
	0	∞	∞	∞

$S : \{A\}$

Step2:- EXTRACT_MIN(Q) [A]

Relax all the edges leaving A



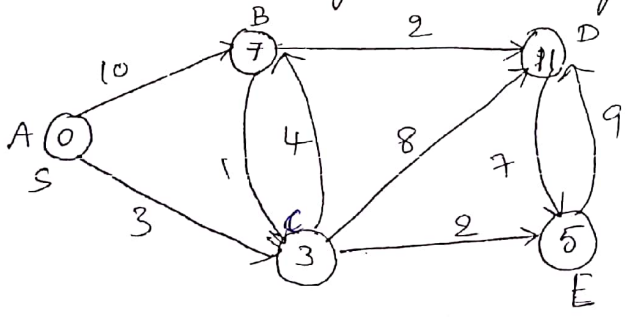
Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S : \{A\}$

Step3:- EXTRACT_MIN(Q) [c]

$S : \{A, C\}$

Relax all the edges leaving C

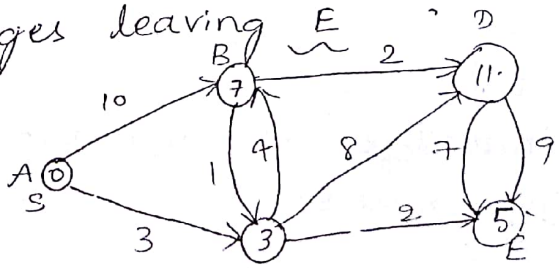


Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

Step4:- EXTRACT_MIN(Q) [E]

Relax all the edges leaving E

$S : \{A, C, E\}$

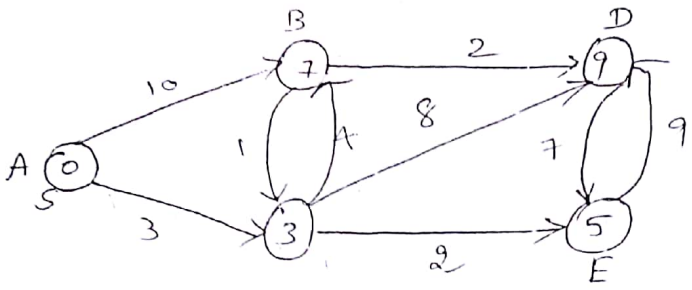


8

5:- EXTRACT-MIN(Q) [B]

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
	0	7	3	9	5

S: {A, C, E, B}



S: {A, C, E, B, D}

The shortest path from a single source A to the remaining vertices : A — B = 7

A — C = 3

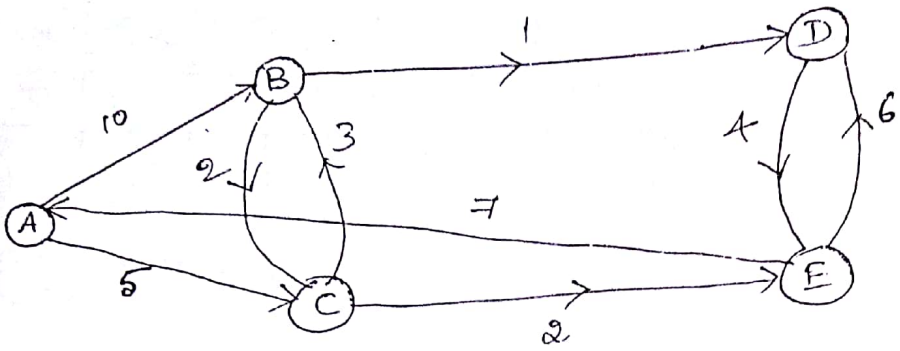
A — D = 9

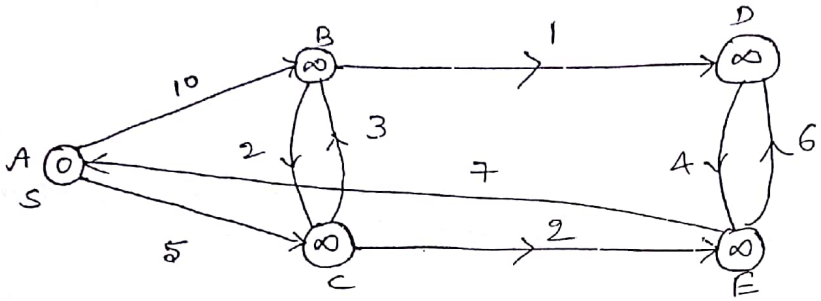
A — E = 5

Algorithm

```
Dijstra (G, w, S)
{
  initialize-Single-Source (G, S)
    S ← ϕ;
    Q ← v[G];
    while Q = ϕ
      do
        u ← Extract-MIN(Q)
        S ← S ∪ {u}
      for each vertex V ∈ adj[u]
        do relax (u, v, w)
}
```

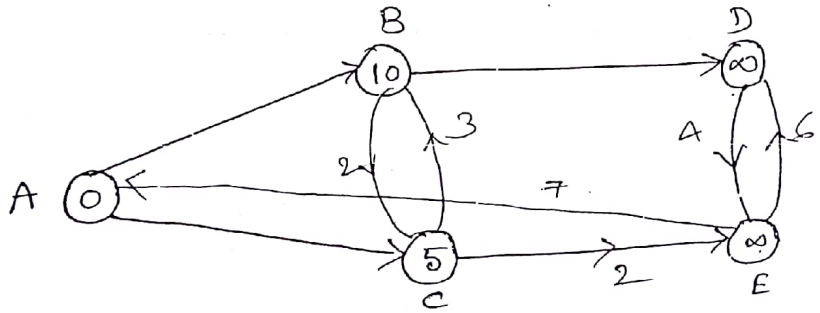
Q:-





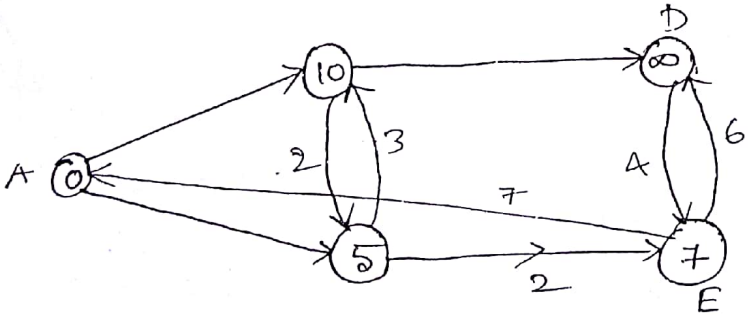
S: {A}

Q	A	B	C	D	E
	0	∞	∞	∞	∞



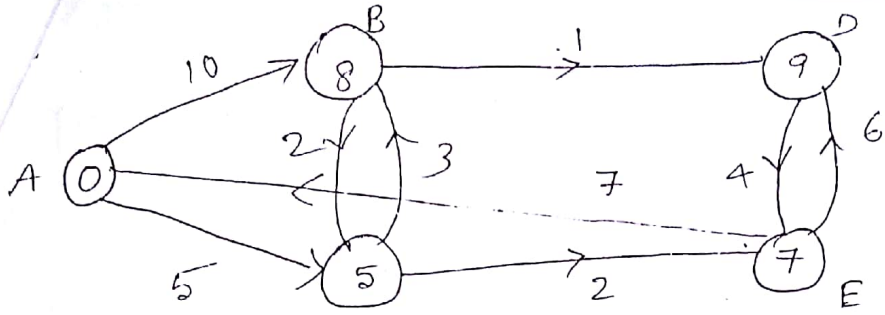
S: {A, C}

Q	A	B	C	D	E
	0	∞	∞	∞	∞
		10	5	∞	∞



S: {A, C, E}

Q	A	B	C	D	E
	0	∞	∞	∞	∞
		10	5	∞	∞
		10	5	∞	



Q	A	B	C	D	E
	0	∞	∞	∞	∞
		10	5	∞	∞
		10	5	∞	7
		10	5	8	7

$S: \{A, C, E, B, D\}$

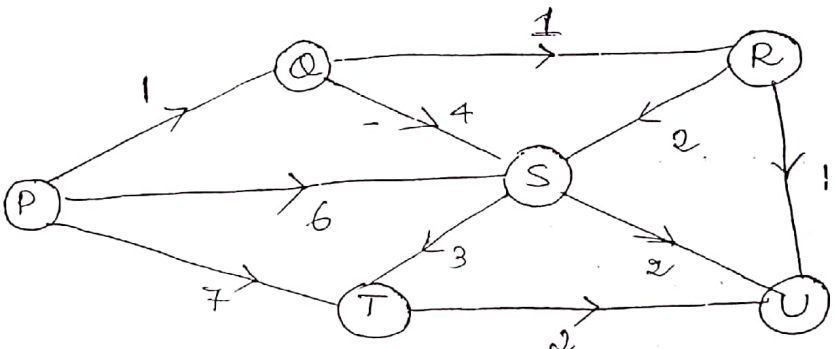
The shortest path from a single source A to the remaining vertices: A — B = 10

A — C = 5

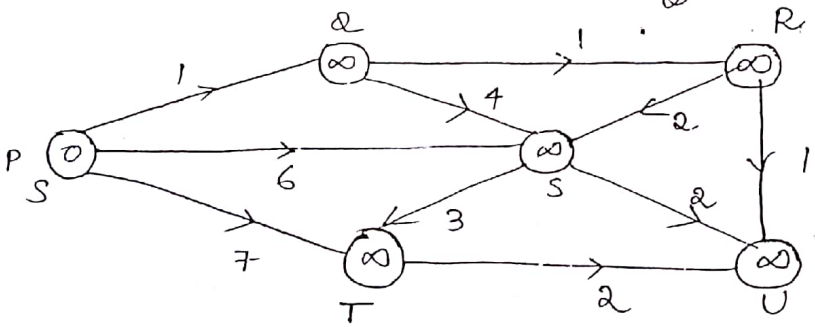
A — D = 8

A — E = 7

eg:-

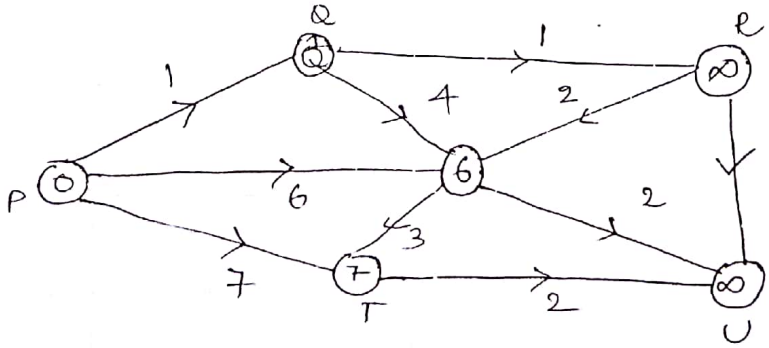


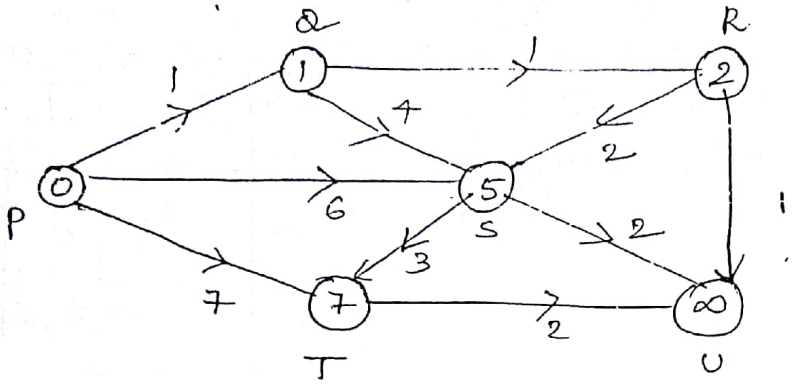
i)



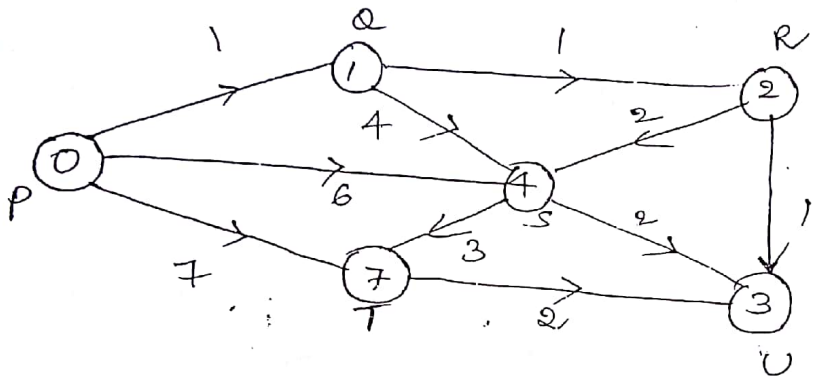
$S: \{P\}$

ii)





S: {P, Q, R}



S: {P, Q, R, U}

S: {P, Q, R, U, S}

S: {P, Q, R, U, S, T}

The shortest path from a single source A to the remaining vertices: P — Q = 1

P − R = 2

P − S = 4

P − T = 7

P − U = 3

All pairs shortest path Algorithm (or) Floyd warshall Algorithm.

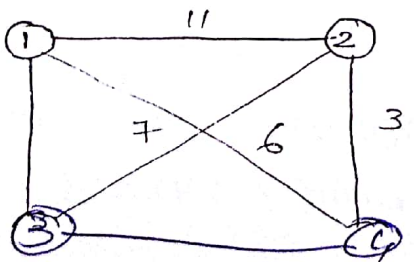
The Algorithm considers the intermediate vertices of a shortest path where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_m \rangle$ is any vertex of p other than v_1 or v_m. The Floydwarshall algorithm is based on the following observation:

let the vertices of G be $V = \{1, 2, 3, \ldots, n\}$ and consider subset $\{1, 2, 3, \ldots, k\}$ of vertices for any pair of vertices $i, j \in V$. consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$ and let p be the minimum weight path from among them. The Floydwarshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$

If k is intermediate vertex of path p and let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$

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

Example:-



$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Algorithm:-

```
FLOYD-WARSHALL(W)
{
 n ← rows[W]
 D(0) ← W
for k ← 1 to n
do for i ← 1 to n
do for j ← 1 to n
do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D(n)
}
```

Step 1:- calculate D^1 {1}

To calculate D^1 fix the first row and first colocumn of the D^0 matrix. Find the remaining elements add the corresponding row and colocumn numbers and replace the position with minimum

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$D^1(2,3) = \min((2,3), (2,1)+(1,3))$$
$$= \min(7, 11+1)$$
$$= \min(7, 12)$$
$$= 7$$

$$D^1(2,4) = \min((2,4), (2,1)+(1,4))$$
$$= \min(3, 11+6) = \min(3, 17) = 3$$

$,2) = \min((3,2), (3,1)+(1,2))$

$= \min(7, 1+11)$

$= \min(7, 12)$

$= 7$

$D'(3,4) = \min((3,4), (3,1)+(1,4))$

$= \min(2, 1+6)$

$= \min(2, 1+6)$

$= 2$

$D'(4,2) = \min((4,2), (4,1)+(1,2))$

$= \min(3, 6+11)$

$= \min(3, 17)$

$= 3$

$D'(4,3) = \min((4,3), (4,1)+(1,3))$

$= \min(2, 6+1)$

$= \min(2, 7)$

$= 2$

$$D' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Step 2:- Calculate $D^2\{1,2\}$

To calculate D^2 fix the second row and secondcoloumn of D' add the corresponding elements from choosen ow and coloumn and update with the minimum value

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^2(1,3) = \min((1,3), (1,2)+(2,3))$
$= \min(1, 11+7)$
$= \min(1, 18)$
$= 1$

$D^2(1,4) = \min((1,4), (1,2)+(2,4))$
$= \min(6, 11+3)$
$= \min(6, 14)$
$= 6$

$D^2(3,1) = \min((3,1), (3,2)+(2,1))$
$= \min(1, 7+11)$
$= \min(1, 18)$
$= 1$

$D^2(3,4) = \min((3,4), (3,2)+(2,4))$
$= \min(2, 7+3)$
$= \min(2, 10)$
$= 2$

$D^2(4,1) = \min((4,1), (4,2)+(2,1))$
$= \min(6, 14)$
$= 6$

$D^2(4,3) = \min((4,3), (4,2)+(2,3))$
$= \min(2, 3+7)$
$= \min(2, 10)$
$= 2$

3:- To calculate $D^3\{1,2,3\}$

Fix the third column and third row from D^2 and add corresponding elements of selected row and column choose the minimum and update the matrix

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 1 & 3 \\ 8 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^3(1,2) = \min((1,2), (1,3)+(3,2))$
$= \min(11, 1+7)$
$= \min(11, 8)$
$= 8$

$D^3(2,1) = \min((2,1), (2,3)+(3,1))$
$= \min(11, 7+1)$
$= \min(11, 8)$
$= 8$

$D^3(1,4) = \min((1,4) (1,3)+(3,4))$
$= \min(6, 1+2)$
$= \min(6, 3)$
$= 3$

$D^3(4,1) = \min((4,1)+(4,3)+(3,1))$
$= \min(6, 2+1)$
$= \min(6,3)$
$= 3$

$D^3(2,4) = \min((2,4), (2,3)+(3,4))$
$= \min(3, 7+2)$
$= \min(3, 9)$
$= 3$

$D^3(4,2) = \min((4,2), (4,3)+(3,2))$

Step 4:- To calculate D^4 {1,2,3,4}

To calculate D^4 fix the fourth row and fourth col of D^3 and add the corresponding elements and select the minimum value and update the matrix.

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^4(1,2) = \min((1,2), (1,4)+(4,2))$
$= \min(8, 3+3)$
$= \min(8,6)$
$= 6$

$D^4(1,3) = \min((1,3), (1,4)+(4,3))$
$= \min(1, 3+2)$
$= \min(1,5)$
$= 1$

$D^4(2,1) = \min((2,1), (2,4)+(4,1))$
$= \min(8, 3+3)$
$= \min(8,6)$
$= 6$

$D^4(2,3) = \min((2,3), (2,4)+(4,3))$
$= \min(7, 3+2)$
$= \min(7,5)$
$= 5$

$D^4(3,1) = \min((3,1), (3,4)+(4,1))$
$= \min(1, 2+3)$
$= \min(1,5)$
$= 1$

$$= \min((3,2), (3,4)+(4,2))$$
$$= \min(7, 2+3)$$
$$= \min(7,5)$$
$$= 5$$

WARSHALL's Algorithm :-

It is used to find the transtive closure of a graph. The transtive closure of a graph G is defined to be a graph G' such that G' has the same nodes as G and there is an edge v_i, v_j in G' whereever there is a path from v_i to v_j in G"

The path matrix P of the graph G is the adjacency matrix of its transtive closure G'

The Recursive definition of transtive closure is,

$$t_{ij}^{0} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \in E \\ 1 & \text{if } i=j \text{ (or) } (i,j) \in E \end{cases}$$

The reachability is called transtive closure of a graph

$$t_{ij}^{(K)} = t_{ij}^{(K-1)} \vee (t_{iK}^{(K-1)} \wedge t_{Kj}^{(K-1)})$$

Transtive closure :- Algorithm

```
Transtive-closure(G)
{
 n ← v[G]
for i ← 1 to n
  do for j ← 1 to n
  do if i=j (or) (i,j) ∈ E[G]
  t        , (0)
```

eva

$t_{ij}^{(0)} \leftarrow 0$

for $k \leftarrow 1$ to n

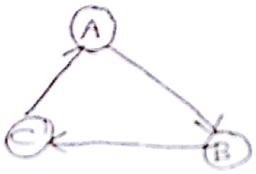
do for $i \leftarrow 1$ to n

do for $j \leftarrow 1$ to n

do

$$t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Q:-



The adjacency matrix for the above graph

$$T^0 = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

Step 1:- calculate T^1 by using T^0

At $k=1,\ i=1 \quad j=1$

$$T_{11}^1 \leftarrow T_{11}^0 \vee [T_{11}^0 \wedge T_{11}^0]$$
$$= 0 \vee [0 \wedge 0]$$
$$= 0$$

$$T^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

At $k=1, i=1, j=2$

$$T_{12}^1 = T_{12}^0 \vee (T_{11}^0 \wedge T_{12}^0)$$
$$= 1 \vee [0 \wedge 1] = 1$$

At $k=1 \quad i=1 \quad j=3$

$$T_{13}^1 = T_{13}^0 \vee (T_{11}^0 \wedge T_{13}^0) = 0 \vee [0 \wedge 0] = 0$$

$k=1,\ i=2,\ j=1$

$$T^1_{21} = T^0_{21} \vee [T^0_{21} \wedge T^0_{11}] = 0 \vee [0 \wedge 0] = 0$$

At $k=1,\ i=2,\ j=2$

$$T^1_{22} = T^0_{22} \vee (T^0_{21} \wedge T^0_{12})$$
$$= 0 \vee [0 \wedge 1] = 0$$

At $k=1,\ i=2,\ j=3$

$$T^1_{23} = T^0_{23} \vee (T^0_{21} \wedge T^0_{13})$$
$$= 1 \vee (0 \wedge 0) = 1 \vee 0 = 1$$

At $k=1,\ i=3,\ j=1$

$$T^1_{31} = T^0_{31} \vee (T^0_{31} \wedge T^0_{11})$$
$$= 1 \vee (1 \wedge 0)$$
$$= 1$$

At $k=1,\ i=3,\ j=2$

$$T^1_{32} = T^0_{32} \vee (T^0_{31} \wedge T^0_{12})$$
$$= 0 \vee [1 \wedge 1] = 1$$

At $k=1,\ i=3,\ j=3$

$$T^1_{33} = T^0_{33} \vee (T^0_{31} \wedge T^0_{13})$$
$$= 0 \vee (1 \wedge 0) = 0$$

Step2:- calculate T^2 by using T^1

At $k=2,\ i=1,\ j=1$

$$T^2_{11} = T^1_{11} \vee (T^1_{12} \wedge T^1_{21})$$
$$= 0 \vee (1 \wedge 0)$$
$$= 0 \vee 0 = 0$$

$$T^2 = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

At $k=2, i=1, j=2$

$$T_{12}^{2} = T_{12}^{1} \vee (T_{12}^{1} \wedge T_{22}^{1})$$
$$= 1 \vee (1 \wedge 0)$$
$$= 1 \vee 0$$
$$= 1$$

At $k=2, i=1, j=3$

$$T_{13}^{2} = T_{13}^{1} \vee (T_{12}^{1} \wedge T_{23}^{1})$$
$$= 0 \vee (1 \wedge 1)$$
$$= 0 \vee (1)$$
$$= 1$$

At $k=2, i=2, j=1$

$$T_{21}^{2} = T_{21}^{1} \vee (T_{22}^{1} \wedge T_{21}^{1})$$
$$= 0 \vee (0 \wedge 0)$$
$$= 0 \vee 0 = 0$$

At $k=2, i=2, j=2$

$$T_{22}^{2} = T_{22}^{1} \vee (T_{22}^{1} \wedge T_{22}^{1})$$
$$= 0 \vee (0 \wedge 0)$$
$$= 0 \vee (0)$$
$$= 0$$

At $k=2, i=2, j=3$

$$T_{23}^{2} = T_{23}^{1} \vee (T_{22}^{1} \wedge T_{23}^{1})$$
$$= 1 \vee (0 \wedge 1)$$
$$= 1 \vee 0 = 1$$

$i=3,\ j=1$

$= T^{1}_{31} \vee (T^{1}_{32} \wedge T^{1}_{21})$

$= 1 \vee (1 \wedge 0)$

$= 1 \vee 0 = 1$

$k=2,\ i=3,\ j=2$

$= T^{1}_{32} \vee (T^{1}_{32} \wedge T^{1}_{22})$

$= 1 \vee (1 \wedge 0)$

$= 1 \vee (0)$

$= 1$

$k=2,\ i=3,\ j=3$

$= T^{1}_{33} \vee (T^{1}_{32} \wedge T^{1}_{23})$

$= 0 \vee (1 \wedge 1)$

$= 0 \vee 1$

$= 1$

Ep 3:- calculate T^3 by using T^2

$$T^3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$k=3,\ i=1,\ j=1$

$= T^{2}_{11} \vee (T^{2}_{13} \wedge T^{2}_{31})$

$= 1 \vee (1 \wedge 1)$

$= 1 \vee 1$

$= 1$

$k=3,\ i=1,\ j=2$

$T^{3}_{12} = T^{2}_{12} \vee (T^{2}_{13} \wedge T^{2}_{32})$

$= 1 \vee (1 \wedge 1)$

$= 1$

$k=3, i=1, j=3$

$T^3 = T^2 \vee (T^{2}_{13} \wedge T^{2}_{33}) = 1 \vee (1 \wedge 1) = 1$

At $k=3, i=2, j=1$

$$T^3_{21} = T^2_{21} \vee (T^2_{23} \wedge T^2_{31})$$
$$= 0 \vee (1 \wedge 1)$$
$$= 1$$

At $k=3, i=2, j=2$

$$T^3_{22} = T^2_{22} \vee (T^2_{23} \wedge T^2_{32})$$
$$= 0 \vee (1 \wedge 1)$$
$$= 1$$

At $k=3, i=2, j=3$

$$T^3_{23} = T^2_{23} \vee (T^2_{23} \wedge T^2_{33})$$
$$= 1 \vee (1 \wedge 1)$$
$$= 1$$

At $k=3, i=3, j=1$

$$T^3_{31} = T^2_{31} \vee (T^2_{33} \wedge T^2_{31})$$
$$= 1 \vee (1 \wedge 1)$$
$$= 1$$

At $k=3, i=3, j=2$

$$T^3_{32} = T^2_{32} \vee (T^2_{33} \wedge T^2_{32})$$
$$= 1 \vee (1 \wedge 1)$$
$$= 1$$

At $k=3, i=3, j=3$

$$T^3_{33} = T^2_{33} \vee (T^2_{33} \wedge T^2_{33})$$
$$= 1 \vee (1 \wedge 1)$$
$$= 1$$

$$T^3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$