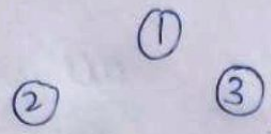
Unit - IV

Graph :→ It can be defined as group of vertices & edges that are used to connect these vertices.

G(V, E)

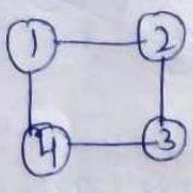
Types of graph :→

1) **Null graph** :→ No edges.

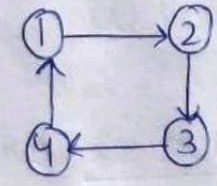


2) **Trivial graph** :→ only one vertex.

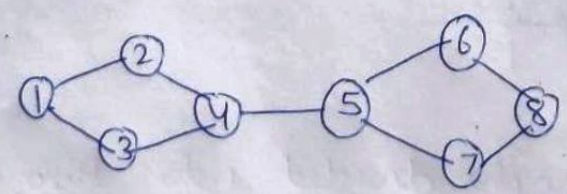
3) **Non-directed** :→ can't determine the starting & ending (i.e. no direction)



4) **Directed** :→ have direction to determine the starting & ending node.

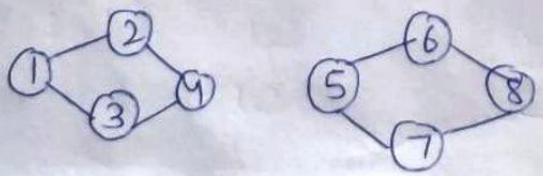


5) **Connected graph** :→ atleast one path between all the vertice



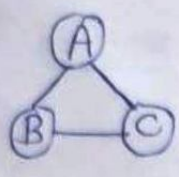
Path :→
1-3-4-5-6-8

6) **disconnected graph** :→ no path.



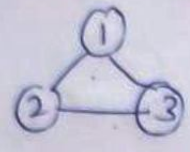
No path from 1 to 8.

7) Regular graph :→ all vertices should have same degree.

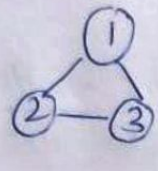


⊙ degree of A = B = C = 2
⊙ It is 2-regular graph.

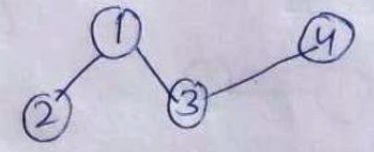
8) Cycle :→ form a cycle.



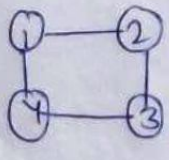
9) Complete :→ all vertices are connected.



10) Acyclic :→ No cycle.

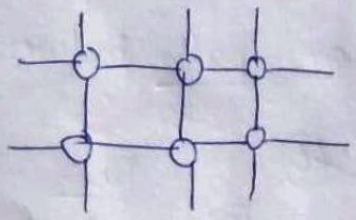


11) Finite :→ No. of edges & vertices are finite

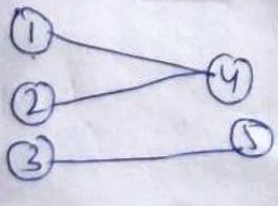


4 - edges
4 - vertices

12) Infinite :- No. of edges & vertices are infinite.

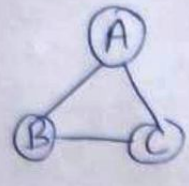


13) Bipartite :→ ⊙ graphs divided into two distinct sets (x & y).
⊙ X should be connected to Y only. (not X to X).



Set X = {1, 2, 3}
Set Y = {4, 5}

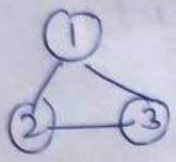
(4) Euler graph :- Vertices have an even degree.



degree A = B = C = 2

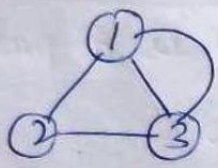
15) Hamilton :- ① visit every vertex of the graph exactly once.

② Starts & ends at same vertex.

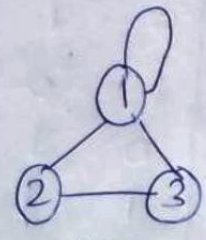


1-2-3-1 ⇒ Hamilton path

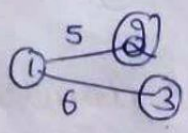
16) Multigraph :- No self-loops, but parallel edges.



17) Pseudo :- No parallel edges, but self-loops



18) Weighted :- some weight on edge.



Terminology used with graph :-

1) Path → Sequence of node to reach terminal node from initial node.

2) Closed path :- Starting = ending node.

3) Node → vertices, represented as circle.

4) Edge → line is a connection between two nodes.

5) Adjacent node

6) degree :- no. of edges connected to that node.

7) Size of graph :- no. of edges in graph.

Size = 2

D(1) = 2

Representation of graph :-

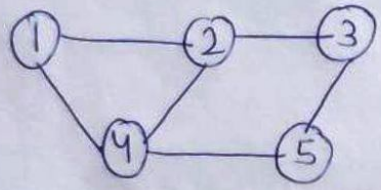
- Sequential Representation OR (Adjacency matrix)
- Linked list Representation OR (Adjacency list)

1) Adjacency Matrix :-

Matrix $A = [a_{ij}]$ can be defined as :-

$a_{ij} = 1$, { if there is a path from V_i to V_j }

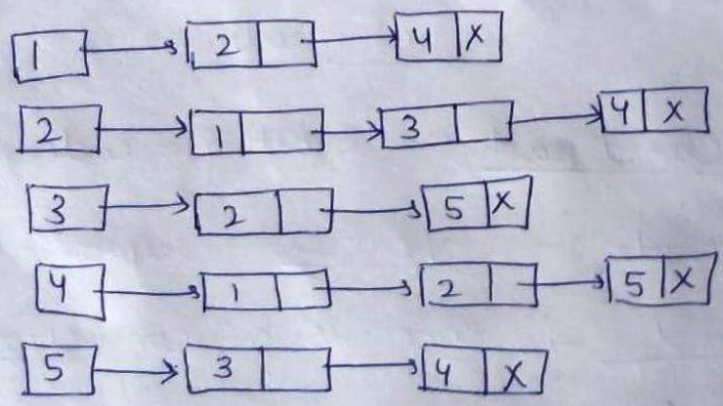
$= 0$, otherwise



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	0	1
4	1	1	0	0	1
5	0	0	1	1	0

* for directed & weighted graph

2) Adjacency list :- Linked list is used for this.

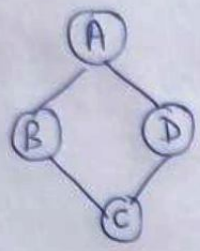


* for directed, see directions & for weighted write down weight also with node. (1) —5→ (2) ⇒ 1 → 2 | 5 | X

Operations on Graphs :→

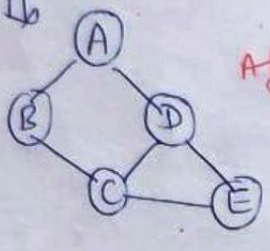
1) **Insertion** :→ a) add a row for new vertex.
b) add a column for new vertex.
c) Make appropriate entries into row & column for new edge.

Eg :-



	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

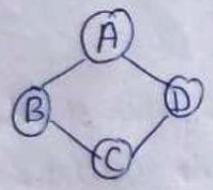
Add new node E ⇓



After insertion →

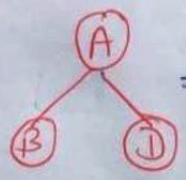
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	0
C	0	1	0	1	1
D	1	0	1	0	1
E	0	0	1	1	0

2) **Deletion** :- a) delete the row & column corresponding to vertex.



	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

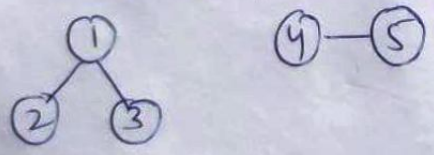
delete C ⟹



⇒

	A	B	D
A	0	1	1
B	1	0	0
D	1	0	0

c) **Merging** :- Merge two graphs.



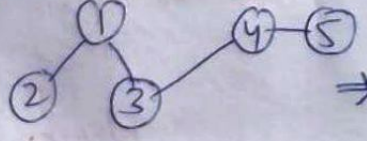
Before Merging

one graph

	1	2	3	4	5
1	0	1	1		
2	1	0	0		
3	1	0	0		
4				0	1
5				0	1

IInd graph

After merging



⇒

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	0	0
3	1	0	0	1	0
4	0	0	1	0	1
5	0	0	0	1	0

Merge

Graph Traversal :- Process of visiting & exploring a graph.

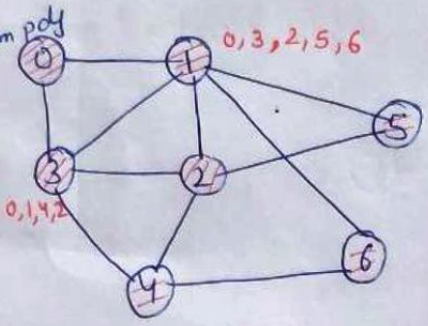
There are two methods

- **BFS** (Breadth first Search) ⇓ Queue is used
- **DFS** (Depth first Search) ⇓ stack is used

During the execution, each node will be one of three states:-

- ⊙ Status = 1 ⇒ Ready state i.e. initial state.
- ⊙ Status = 2 ⇒ Waiting state
- ⊙ Status = 3 ⇒ Processed State

1) **BFS :-** Algo from pdf



Queue :-

0	1	3	2	5	6	4

Result :- 0132564

Starts from 0.

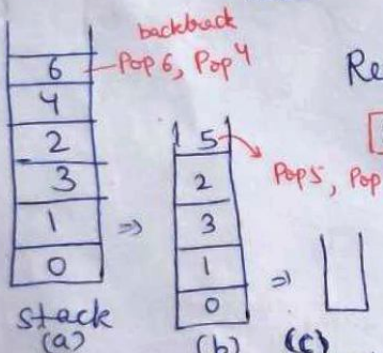
⊙ check adjacent node of each node.

0 → 1, 3 ↓ 0,3,2,5,6 ↓ 0,1,4,2

& solve as FIFO.

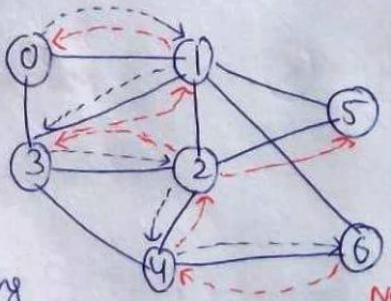
2) **DFS :-** Same eg. as BFS (use stack)

Algo



Result :- 0132465

* Any one of adjacent node will be used & if there will be no node then we backtrack.



Now, backtrack until stack is empty. (use top element & pop it for backtrack.

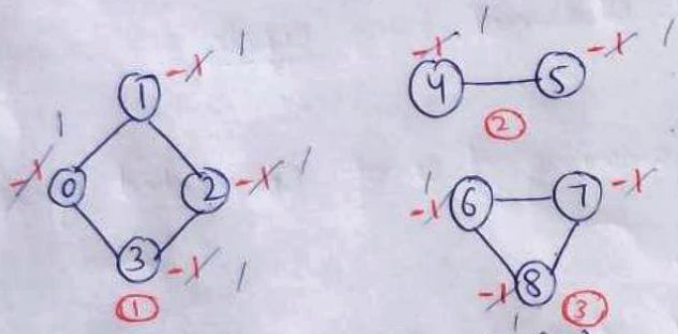
Types of edges in DFS traversal :→

1) Tree edge → member of DFS traversal.
2) Forward edge → $e_1(x,y)$ → where y appears after x.
3) backward edge → $e_1(x,y)$ → where y appears before x.
4) Cross edge → no path from x to y.

Connected Components : →

1) Initialize all vertices as not visited.
2) Do the following for every vertex v:
   a) If v is not visited before, call the DFS & print newline character to print each component in a new line.
      i) Mark v as visited & print v
      (ii) For every adjacent u of v, If u is not visited, then recursively call the DFS.

Eg.



There are three connected components.

N = 9 → No of vertex

```
Connected-Components (G)
{
    for each vertex v ∈ N          N = 9
                                   # initialize every vertex = -1
        flag[v] = -1
        count = 0
    for (int v = 0; v < N; v++)    loop 0 to 8
    {
        if (flag[v] == -1)
        {
            DFS(v, flag)
            count++;
        }
    }
```

```
        printf("%d", count);
    }
}

DFS (int v, int flag)
{
    flag[v] = 1
    printf("%d", v);
    for each adjacent node u of v
        if (flag[u] == -1)
            DFS(u, flag)
}
```

change value of -1 to +1, so it become visited.

Spanning Tree :→

- ⊙ subgraph of undirected connected graph.
- ⊙ includes all the vertices with least possible no. of edges.
- ⊙ It cannot be disconnected.
- ⊙ It doesnot have cycle.

Properties :→

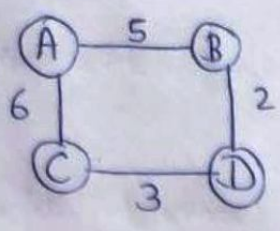
- ⊙ More than one Spanning tree of connected graph G.
- ⊙ No cycles or loop.
- ⊙ Minimally connected. (remove one edge make graph disconnected)
- ⊙ Maximally acyclic. (adding one edge will create loop)
- ⊙ Max. n^{n-2} spanning tree of connected graph.
- ⊙ $n-1$ edges, where n is no. of nodes.

Minimum Spanning Tree (MST) :→

- ⊙ Sum of weights of the edge is minimum.
- ⊙ $G(V,E) \Rightarrow G(V',E') \Rightarrow V'=V, E'<E$

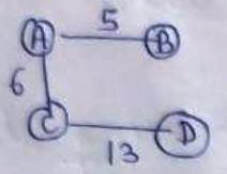
$E' = |V| - 1$

Eg :-

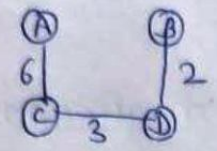


Weight of graph = 16

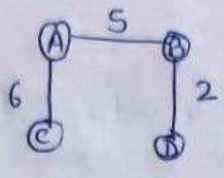
MST :- The possible spanning trees are :-



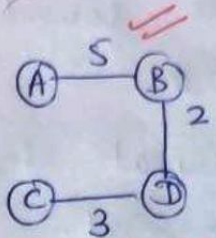
Sum = 14



Sum = 11



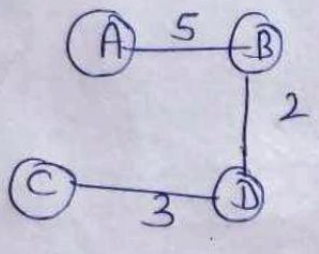
Sum = 13



Sum = 10

Minimum weight = 10
so, it will be selected.

MST will be



Sum = 10

Applications

- Find paths in map.
- design water-supply networks, telecommunication networks, electrical grid.
- Computer network routing protocol. etc.

Algorithm

- Prim's
- Kruskal's

1) Prim's Algorithm :- It is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached.

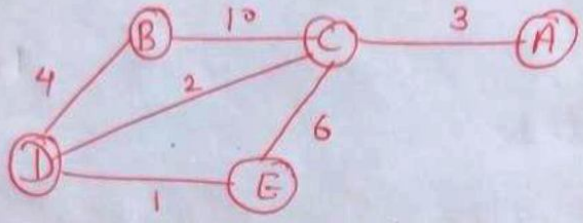
Algo:-

1) First, we have to initialize an MST with the randomly chosen vertex.

2) Now, find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the min. edge & add it to the tree.

3) Repeat step 2 until the minimum spanning tree is formed.

Eg :-

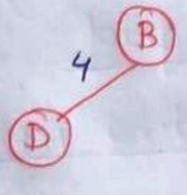


Sol.

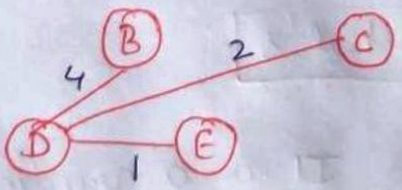
Step-1 :- Choose one vertex.

Step-2 :- find shortest vedge from vertex B.

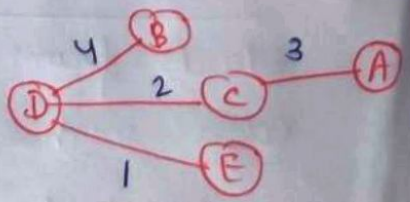
B-C = 10 , B-D = 4 → min. (Choose this)



Step-3 :- D-C = 2, D-E = 1 → min.



Step-4 :- C-A = 3, → min CE = 6



Cost = 4+2+1+3 = 10 units

2) Kruskal's Algorithm :- Find the subset of edges by using which we can traverse each vertex of the graph.

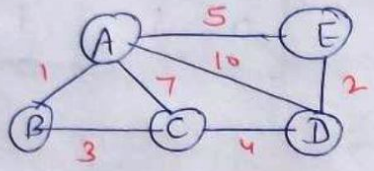
Algo :-

a) First, sort all the edges from low weight to high.

b) Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.

c) Continue to add the edges until we reach all vertices, and a min. spanning tree is created.

Eg :-



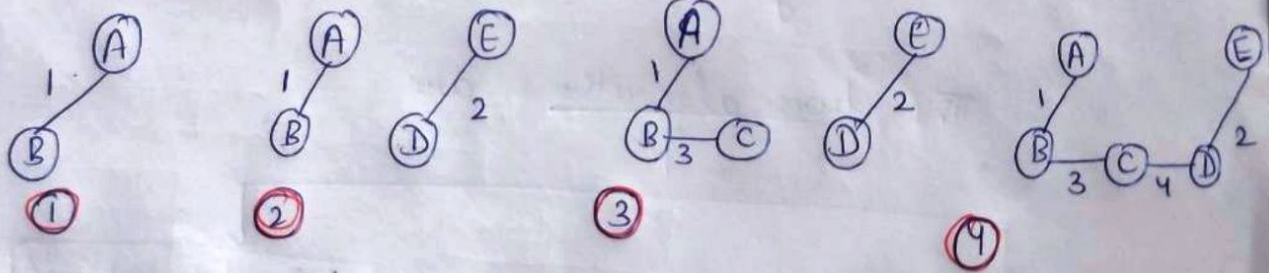
Sol.

Step 1 -

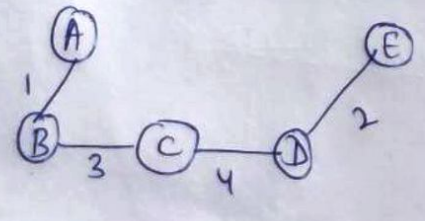
Edge	AB	AC	AD	AE	BC	CD	DE
weight	1	7	10	5	3	4	2

Step 2 - Sort in ascending order.

Edge	AB	DE	BC	CD	AE	AC	AD
weight	1	2	3	4	5	7	10



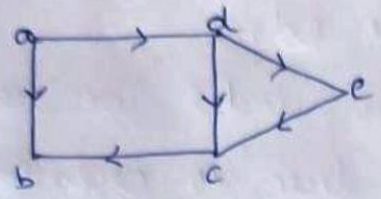
AD, AC will create a cycle, so we will not consider this. Final MST is ⇒



Cost = 10

Transitive closure

Find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the graph. (Reachable mean that there is a path from vertex i to j). The reachability matrix is called the transitive closure of a graph.



	a	b	c	d	e
a	1	1	1	1	1
b	0	1	0	0	0
c	0	1	1	0	0
d	0	1	1	1	1
e	0	1	1	0	1

Shortest Path Algorithm :→

The shortest path from vertex s to vertex t is then defined as any path p with weight $w(p) = \delta(s,t)$

In Single Source Shortest paths problem, we are given a graph $G = (V, E)$, we want to find the shortest path from a given source vertex $s \in V$ to every vertex $v \in V$.

The two algorithms are

- Dijkstra
- Floyd Warshall

Dijkstra Algorithm :-

It is a single-source shortest path algorithm.

There are following steps to solve the problem.

1) Create a set sptset (shortest path tree set) that keeps track of vertices included in the shortest path tree i.e. whose minimum distance from the source is calculated & finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance value as Infinite. Assign the distance value as 0 for the source vertex so that it is picked first.

3) While sptset doesnot include all vertices :-

   a) Pick a vertex 'u' which is not there in sptset & has a min. distance value.

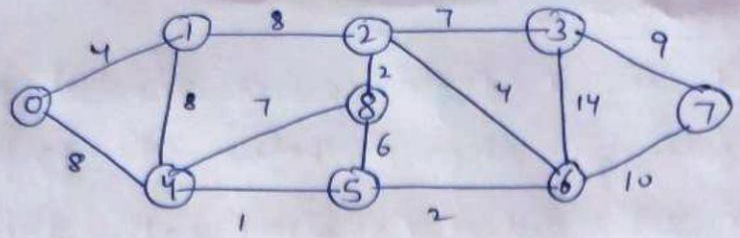
   b) Include u to sptset.

   i) To update the distance values, iterate through all adjacent vertices.

   ii) For every adjacent vertex v, if the sum of the distance value of u (from source) & weight of edge u-v, is less than the distance value of v, then update the distance value of v.

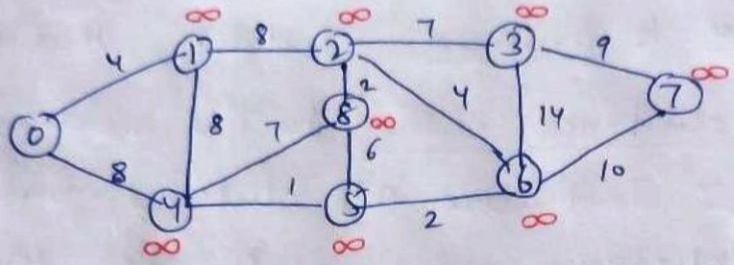
$$d(x,y) = d(x) + c(x,y) < d(y)$$

Eg:-



Sol.

1) Initially take all distance value as infinity.

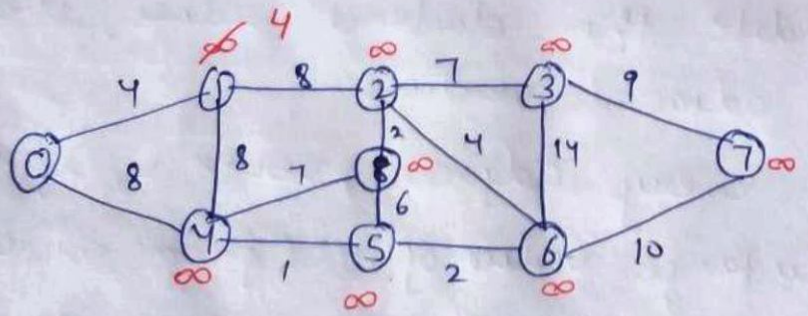


2) Calculate distance from '0' i.e. source vertex. If it is less than the previous one, then update that.

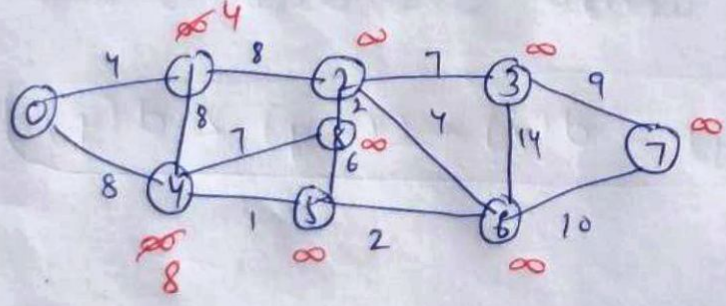
(a) (0-1) ⇒ $d(x,y) = d(x) + c(x,y) < d(y)$

$= (0 + 4) < \infty$

$= 4 < \infty$, then update.



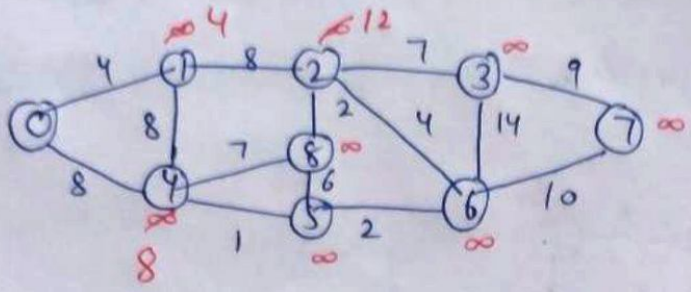
(b) (0-4) = (0+8) < ∞ ⇒ update '8' from 0-4.



'0' is visited.

c) (0-1) & (0-4), (0-1) is lowest. So vertex 1 is selected.

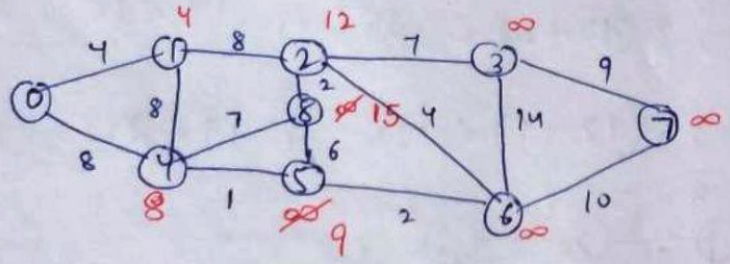
(1-2) ⇒ (4+8) < ∞ ⇒ 12 (update)



d) (1-4) ⇒ (4+8) < 8 ⇒ 12 ≠ 8, (No updation)

1 is visited. Now, find the min. vertex. i.e. 4

e) (4-5) ⇒ (8+1) < ∞ ⇒ 9 < ∞ (update)



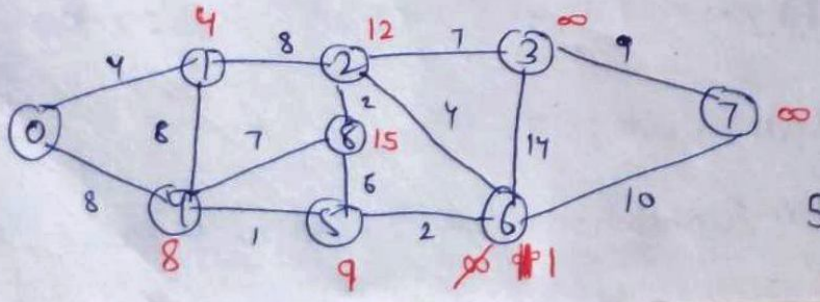
(4-8) ⇒ (8+7) < ∞ ⇒ 15 < ∞ (update)

4 is visited vertex, The min. vertex of distance 9 is vertex 5.

f)

(5-6) ⇒ (9+2) < ∞ ⇒ 11 < ∞ (update)

(5-8) ⇒ (9+6) < 15 ⇒ 15 ≠ 15 (no update)



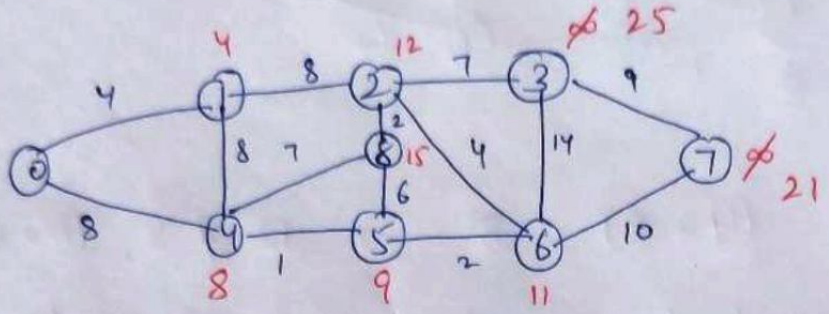
5 is visited

g) Vertex 6 is selected.

(6-2) => (11+4) ≮ 12 => 15 ≮ 12 (No updation)

(6-7) => (11+10) < ∞ => 21 < ∞ (update)

(6-3) => (11+14) < ∞ => 25 < ∞ (update)

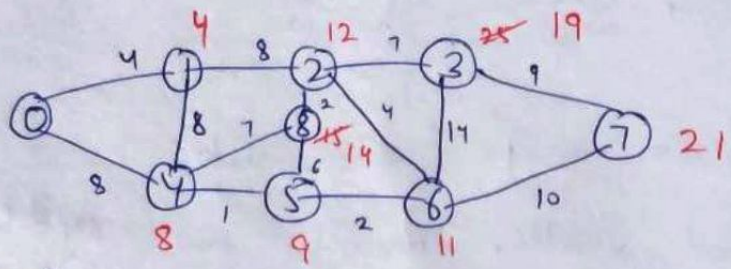


vertex 6 is visited.

h) Vertex 2 is selected.

(2-8) => (12+2) < 15 => 14 < 15, update.

(2-3) => (12+7) < 25 => 19 < 25, update



vertex 2 is visited.

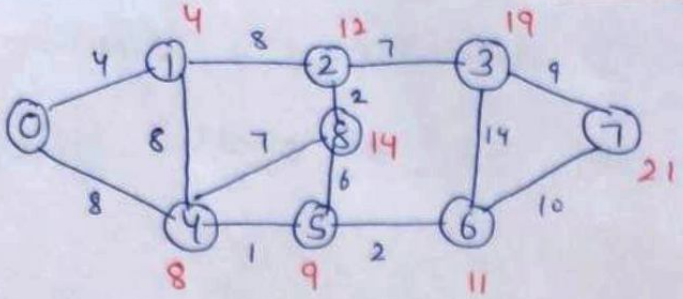
i) vertex 8 is selected, but we cannot update because all adjacent nodes of 8 are visited.

j) Now, select vertex 3.

(3-7) => (19+9) ≮ 21 => 28 ≮ 21 (No updation)

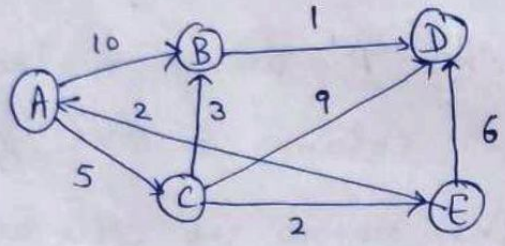
k) Now select vertex 7.

No updation in 7 vertex, because all are visited.



The shortest distance from (0-1) ⇒ 4 , (0-5) ⇒ 9
(0-3) ⇒ 19, (0-4) ⇒ 18 , (0-2) ⇒ 12 , (0-6) ⇒ 11
(0-7) ⇒ 21

For directed graph,



Source vertex = A

Draw a table of all vertex & mark ∞ distance from source vertex to others.

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

→ Select min. one with vertex. i.e. 'A'

① A-B - 10
A-C - 5, Choose C

② C-B = 5+3 = 8 < 10, update
C-D = 5+9 = 14 < ∞, update
C-E = 5+2 = 7 < ∞, update

③ Choose E,
E-D → 7+6 = 13 < 14, update
E-A → X A is visited.

④ Choose B.
B-D ⇒ 8+1 = 9
9 < 13, update

* Shortest distance,
A-B ⇒ 8
A-C ⇒ 5
A-D ⇒ 9
A-E ⇒ 7

A to D Path :- DBCA OR (ACBD) = 5+3+1 = 9

Floyd Warshall :- It computes the shortest path between every pair of vertices of the given graph.

Algo :-

a) Create a matrix A^0 of dimension $n \times n$, where n = no. of vertices.

Each cell A[i][j] is filled with the distance from the i^{th} vertex to j^{th} vertex. If there is no path from i to j, then the cell is left as ∞.

b) Now, create a matrix A^1 using matrix A^0. The elements in the first column & the first row are left as they are. The remaining cells are filled as :-

Let 'k' be the intermediate vertex in the shortest path from i to j, then

$$A[i][j] = A[i][k] + A[k][j] \text{ if } (A[i][j] > A[i][k] + A[k][j])$$

OR

$$A^k[i][j] = \min [A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]]$$

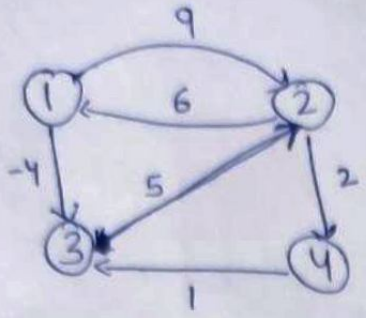
c) Similarly, A^2 is created using A^1. The element in the second column & second row are left as they are. The remaining steps are same as in step b.

d) Similarly, for each vertex other matrices will also be created.

e) The last matrix gives the shortest path between each

pair of vertices.

Eg:-



Sol.

a) A^0 is created of 4×4 vertices, n = 4 with the distance from i to j.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & \infty & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

b) A' is created by using A^0. (leave first row & column as it is, update others if there is min. distance.

$$A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array} \leftarrow \text{leave as it is}$$

$A'[2,3]$ via 1 $= \min[A^0[2,3], A^0[2,1] + A^0[1,3]]$

$= \min[\infty, 6-4] = \min[\infty, 2]$

min is 2, so update 2 in matrix

$A'[2,4] = \min[A^0[2,4], A^0[2,1] + A^0[1,4]]$

$= \min[2, 6+\infty] = \min[2, \infty] = 2$ (No updation)

c) Similarly, create A^2 by using A^1.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

$A^2[1,4] = \min[A^1[1,4], A^1[1,2] + A^1[2,4]]$ via 2

$= \min[\infty, 9+2] = \min[\infty, 11] = \underline{11}$ update.

d) $A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$

e) $A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$

The shortest distance from 1 to 2 → 1
1 to 3 → -4
1 to 4 → 3,

Similarly, 2-1 → 6
2-3 → 2
2→4 → 2 and so on.

Check the last matrix for all the shortest path.

Advantages :-

1) Simple
2) Easy.
3) Calculate distance from every vertex.

Disadvantage :-

1) Slower in some cases.