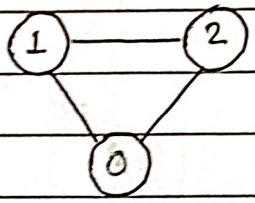
Graphs

Adjacency Matrix :-

An adjacency matrix is a square matrix of N x N size where 'N' is the number of nodes in the graph and its is used to represent the connections between the edges of a graph.

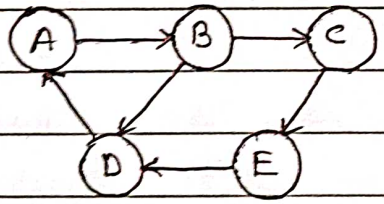


Undirected Graph

	0	1	2
0		1	1
1	1		1
2	1	1	

Row to Column traverse ↑

The diagonal entries of the adjacent matrix will be 0 in undirected graph.



Directed Graph.

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

Representation :-

→ If there exists an edge between vertice V_i and V_j, then $a_{ij} = 1$; Other wise, $a_{ij} = 0$

→ A_{ii} are 0 in absense of self loop.

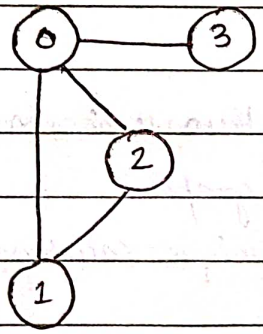
→ Matrix is symmetric for undirected graphs ($a_{ij} = a_{ij}$).

Properties of Adjacency Matrix :-

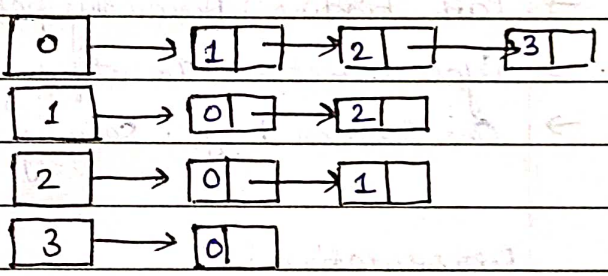
→ Contains 0 and 1's, representing absense of presense of edges.
→ For directed graphs, $A[V_i][V_j] = 1$ if there's an edge from V_i to V_j
→ For undirected graphs, $A[V_i][V_j] = A[V_j][V_i] = 1$ if there an edge between V_i and V_j.
→ For weighted graph instead of 1 ~~and~~ 0's use the given weights.

Adjacency list :-

→ Graph as an array of linked list.
→ Index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Representation in the form of linked list :-



* Index pointing to each index

Pros of Adjacency list :-

→ Storage efficient because we only store values of edges.
→ Identifies all vertices adjacent to a given vertex.

Cons of Adjacency list :-

→ Can be slower than adjacency matrix because all connected nodes must be explored to find adjacent vertices.

Depth First Search (DFS) :-

→ Recursive algorithm for searching all the vertices of a graph
→ Traverses a graph in depthward motion.
→ Uses stack for unvisited nodes.

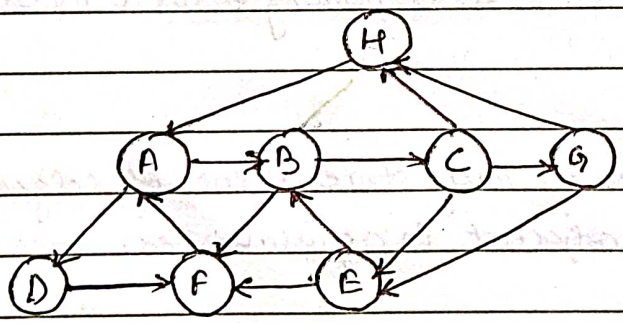
Implementation :-

1. Create a stack with the total number of vertices in the graph.
2. Choose a starting vertex, push into stack.
3. Push a non-visited adjacent vertex to the top of stack.
4. Repeat step 3-4 until no vertex is left to visit from current vertex.
5. If no vertex is left, pop a vertex from the stack.
6. Repeat 2-5 until stack is empty.

Applications :-

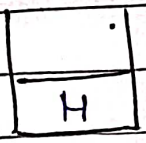
→ Path Finding : Discovering paths between two vertices in a graph
→ Cycle Detection : Identifying cycles in a graph.
→ One-Solution Puzzle : Solving puzzle with unique solution.

Representation :-



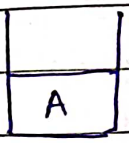
Adjacency list :-

A : B, D
B : C, F
C : E, G, H
D : F
E : B, F
F : A
G : E, H
H : A

1. 

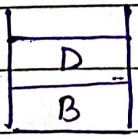
push H

2. pop H and print it, push neighbours of H into TOS (Top of Stack)



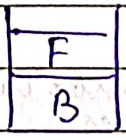
Print (H)

3. Pop A, neighbours of A in stack.



Print (H, A)

4. Pop D, neighbours of D in TOS.

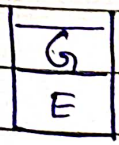


Print (H, A, D)

5. Pop F, neighbour of F in TOS since A is already visited not add back to stack Pop B and neighbours of B into stack.

H, A, D, F, B

6. Print C and add neighbour of C



H, A, D, F, B, C

7. Pop G and add neighbours of G since only F is left unvisited print it.

H, A, D, F, B, C, G, E

Time Complexity :- Worst Case

$O(V+E)$

No. of vertices → V

No. of Edges → E

Space Complexity :-

$O(V)$

Breadth First Search (BFS) :-

→ Traverses the graph in breadthward motion.

→ Uses queue to remember unvisited vertex.

Implementation :-

* Mark each vertex without creating cycles.

1. Put the root of vertex to the back of queue.
2. Take front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of queue.
4. Keep repeating steps 2 and 3 until queue is empty.

Application :-

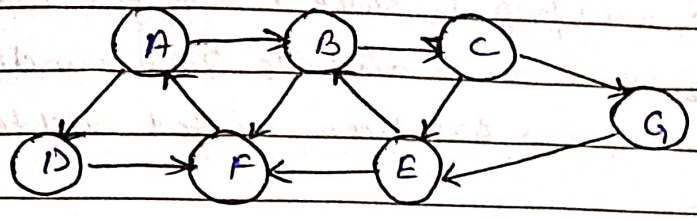
→ GPS Navigation

→ Path finding algorithms.

→ Cycle detection in undirected graph.

→ In minimum spanning tree.

Representation :-



Adjacency list :-

A : B, D
B : C, F
C : E, G
D : F
E : B, F
F : A
G : E

Initialize two queues one visited and one unvisited.

Q1 → unvisited → Insert From → Rear
Q2 → visited. → Delete From → Front

1. Add A to Q1 and null to Q2

$Q_1 = [\ A\ ]$
$Q_2 = [\ null\ ]$

* Follow same rules as DFS search

2. Delete A add to Q_2 and add neighbours of A.

$Q_1 = [\ B, D\ ]$
$Q_2 = [\ A\ ]$

3. Delete B and add neighbours.

$Q_1 = [D, C, F]$
$Q_2 = [A, B]$

4. Insert D to Q_2 and insert neighbours since F is only neighbour so dont add now remove C and Add its neighbours.

$Q_1 = [\ F, E, G\ ]$
$Q_2 = [\ A, B, D, C\ ]$

5. Delete F, add neighbours of F but since A is already visited so dont add, move F to Q_2 now pop E since E has B, F which are already visited in Q_2 so move it also into Q_2, now delete G and since G has E which also has been visited add to Q_2

$Q_1 = [\quad]$

$Q_2 = [A, B, D, C, F, E]$

Time Complexity :- Worst case

$O(V+E)$, $O(V)$, $O(E)$

Space Complexity :-

$O(V)$

Minimum Cost Spanning Tree (MST)

1. Spanning Tree :- A spanning tree is a subgraph of an undirected connected graph.

2. Minimum Spanning Tree :- Minimum Spanning tree can be defined as the spanning tree in which sum of the weights of the edge is minimum.

Krushkal's Algorithm :-

→ Find the MST for connected weighted graph.
→ Identify subset of edges enabling traversal of every graph vertex.
→ Greedy approach makes optimal choice at each vertex — Global Optimum

Implementation :-

1. Start the edge with the lowest weight.
2. Sort all edges from lowest weight to high.
3. Now take lowest weight and add it to the spanning tree.
   → If edge creates cycle reject the edge.
4. Add until we reach all vertices.

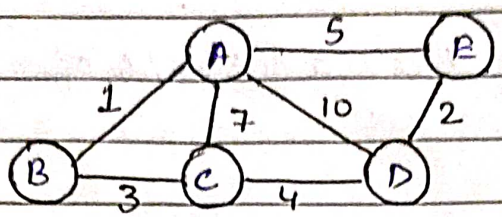
Applications :-

1. Layout electric wiring among cities
2. Lay down LAN connections.

Time Complexity :-

$O(E \log E)$ or $O(V \log V)$

Representation :-



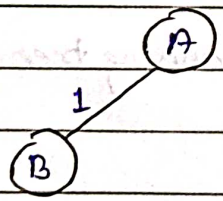
Sort edge and weight :-

Edge :-	AB	AC	AD	AE	BC	CD	DE
Weight :-	1	7	10	5	3	4	2

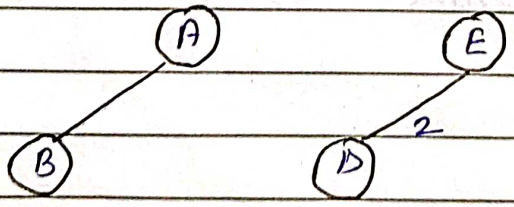
Sort to weight acending (low to high) :-

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

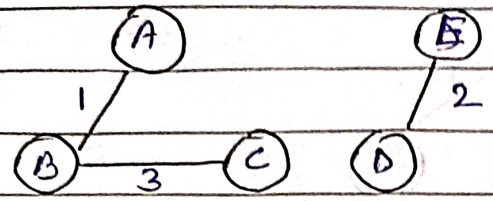
1 Add Edge AB $w=1$.



2. Add DE $w=2$

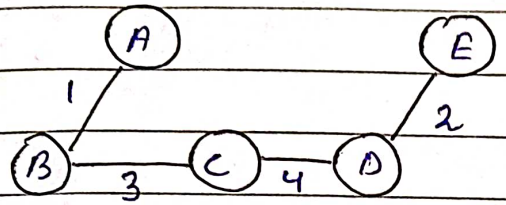


3. Add BC w=3



4. ADd CD w=4

* Adding AC, AE, AD will create a loop so dont add.



← Final Tree

Prim's Algorithm :-

1. Greedy Algorithm that is used to find the minimum spanning tree.
2. Selects edge with minimum weights, ensuring no cycles.
3. Starts with a single node and explores adjacent nodes with connected edges. Chooses edges with min weight.

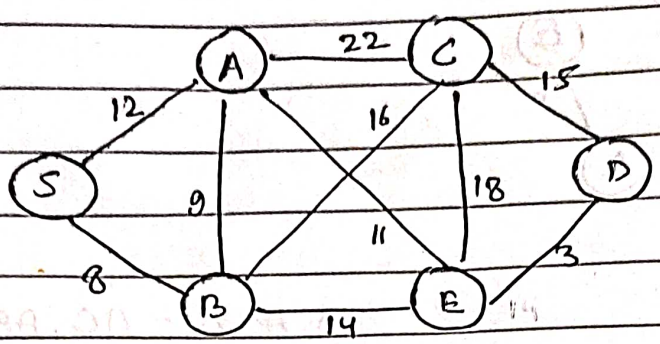
Implementation :-

1. Initialize MST with randomly choosen vertex.
2. From the above choose vertex find neighbours and select minimum vertex.
3. Repeat 2 until tree is formed (MST).

Applications :-

→ Network designing.
→ To make network cycles.
→ Lay down electrical wiring cables.

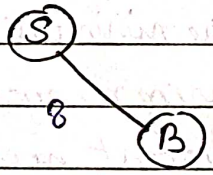
Representation:



1. Select a node be it S.

$V = \{S\}$

2. Find neighbours of S and select the minimum.

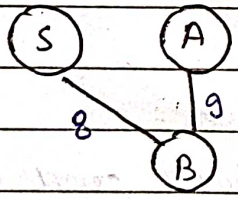


$V = \{S, B\}$

$S \rightarrow A = 12$ ✗

$S \rightarrow B = 8$ ✓

3. Find neighbours of B and select minimum.



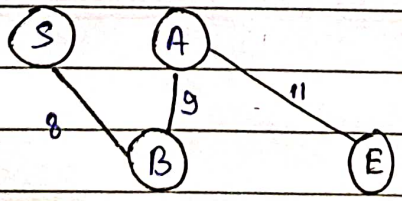
$V = \{S, B, A\}$

$B \rightarrow A = 9$ ✓

$B \rightarrow C = 16$ ✗

$B \rightarrow E = 14$ ✗

4. Do same until cycle is formed.



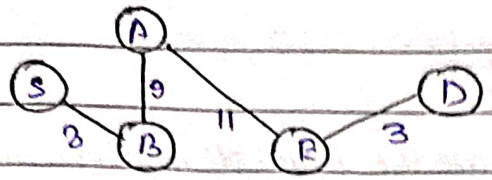
$V = \{S, B, A, E\}$

$A \rightarrow B = 9$ ✗

$A \rightarrow E = 11$ ✓

$A \rightarrow C = 22$

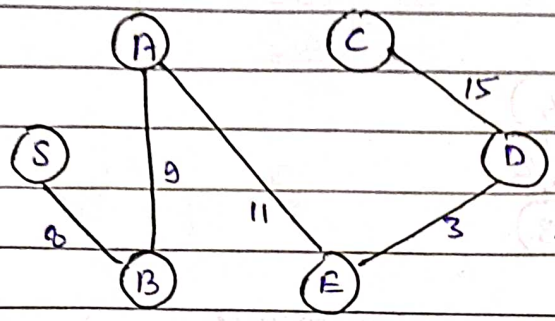
5.



$V = \{S, B, A, E, D\}$

$E \rightarrow D = 3$ ✓
$E \rightarrow C = 12$ ×
$E \rightarrow A = 11$ ×

6.



$V = \{S, B, A, E, D, C\}$

← Final Tree

$D \rightarrow C = 15$ ✓
$D \rightarrow E = 3$ ×

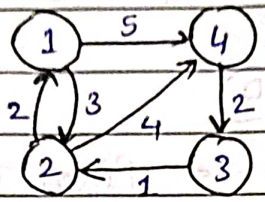
Cost => $8 + 9 + 11 + 3 + 15 = 46$

Time Complexity :- $O(V^2)$ using adjacency list and can be improved to $O(E \times \log V)$ using binary heap.

Auxiliary Space :- $O(V)$

Shortest Path Warshall Algorithm :-

→ Finding the shortest path between all the pair of vertices in a weighted graph.



Create a matrix A^0 with $n \times n$ size (n → no. of vertices)
Row is i and column is j.

1. Each cell is filled with the distance of i^{th} vertex to the j^{th} vertex if there is no path the assign ∞ to the cell.

$$A_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

2. Now create a matrix A' using A^0. The element in the first column and first row are left as original.
Fill the remaining cells using formula :-

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

$$distance[i][j] = \min(distance[i][j], distance[i][R] + distance[R][j])$$

(Vertex values)

→ K=1, when K=1 take first row and first column.

* Refer algorithm on next page →

A^1 =

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

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

$A^1[2][3] = m(dis[2][3], dis[2][1]+dis[1][3])$

$= \min(\infty, 2+5+2)$

$= 9$

A^1 =

	1	2	3	4
1	0	3	∞	5
2	2	0	9	4
3	∞		0	
4	∞			0

$i=2, j=4, k=1$

$A^1[2,4] = \min(dis(2,4), dis((2,1)+(1,4)))$

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

$= \min(4)$

$= 4$

* Calculate all

A^1 =

	1	2	3	4
1	0	3	∞	5
2	2	0	9	4
3	∞	1	0	8
4	∞	∞	2	0

→ Now take K=2 for second vertex and take 2nd row and 2nd column from matrix A^1.

A^2 =

	1	2	3	4
1	0	3		
2	2	0	9	4
3		1	0	
4		∞		0

→

	1	2	3	4
1	0	3	9	5
2	2	0	9	4
3	3	1	0	5
4	∞	∞	2	0

→ Now take k = 3 for next vertex so take 3rd row and third column and similarly for next take k = 4 take 4th row and column from its previous matrix.

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \end{array} \Rightarrow A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \Rightarrow A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

A^4 gives the shortest path between each pair of vertice.

Algorithm :-

n = no. of vertices
A = matrix of dimension n * n
for k = 1 to k = n
    for i = 1 to n
        for j = 1 to n
            $A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$
return A.

Time Complexity :- $O(n^3)$

Space Complexity :- $O(n^2)$

Applications :-

1. Shortest path is a directed graph.
2. To find the transitive closure of directive graph.

P.T.O