CHAPTER 7: GRAPHS

Compiled by: Er. Krishna Khadka

INTRODUCTION

In a graph, each edge is represented by a pair of vertices. Let, v_i and v_j be two vertices and e_i be an edge connecting these two vertices, then we can write,

$$e_i = (v_i, v_j)$$

Here, v_i, and v_j are called end points. The pair of vertices can be ordered or unordered. Depending on these pair, there are two types of graph.

① Directed Graph (di-graph)

② Undirected graph.

If each pair of vertices of a graph is ordered then it is called directed graph and if each pair of

INTRODUCTION

vertices of a graph is unordered then it is called undirected graph. By order pair, we mean that direction is provided for each pair and if no direction is provided it is called unordered pair.

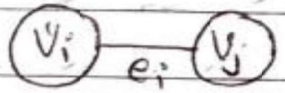


Fig: Unordered pair

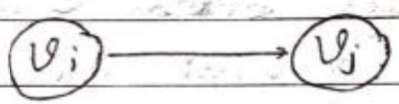


Fig: ordered pair

i.e. For unordered pair: $e_i = (v_i, v_j) = (v_j, v_i)$

for ordered pair: $e_i = (v_i, v_j) \neq (v_j, v_i)$

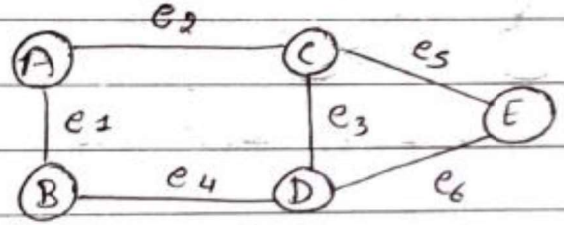


Fig: Undirected Graph

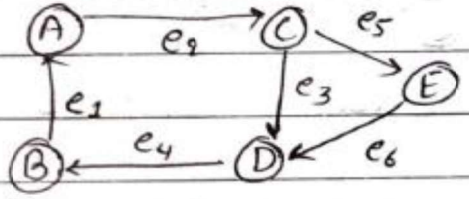


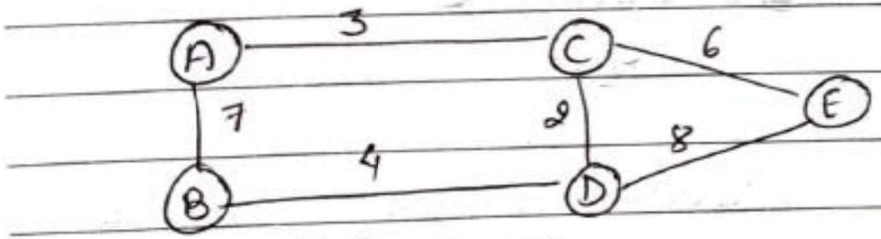
Fig: Directed Graph.

TERMINOLOGY USED

Terminology used:

1. Weighted Graph:

A Graph is said to be weighted graph if all the edges in it are labelled with some number.



2. Connected Graph:

A Graph is said to be connected if there is a path from one vertex to any other node/vertex.

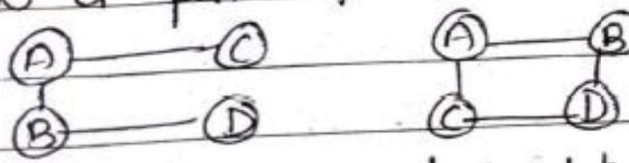


fig: Connected Graph

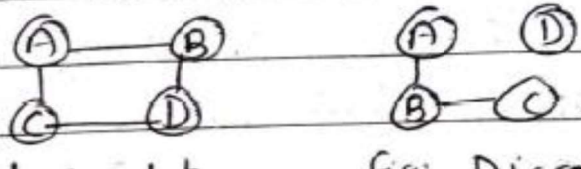


fig: Disconnected Graph

TERMINOLOGY USED

Types:

ⓐ Strongly Connected Graph

ⓑ Weakly Connected Graph.

A connected graph is said to be strongly connected if every vertex is reachable ie. can be visited from every other vertex otherwise it is called weakly connected graph.

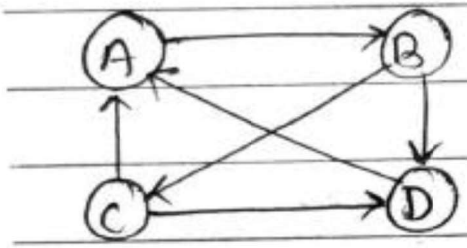


Fig: strongly Connected

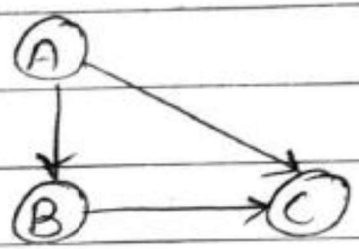
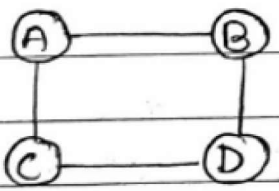


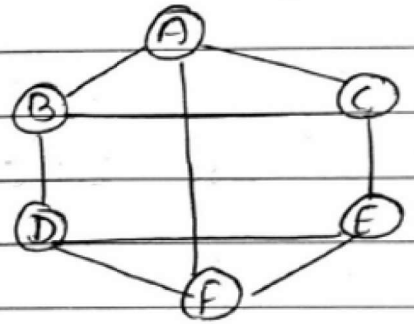
Fig: Weakly Connected

TERMINOLOGY USED

③ Regular Graph:

A Graph is regular if every vertex is adjacent to the same number of vertices.





④ Adjacency:

The relationship between the vertices is called adjacency. The pair of vertices that are directly connected by an arc/edge are called adjacent vertices.

Let, e_i connect two vertices v_i and v_j then, if e_i is directed edge from v_i to v_j, we say v_j is adjacent node of v_i but v_i is not an adjacent node of v_j since, $(v_i, v_j) \neq (v_j - v_i)$. But If

TERMINOLOGY USED

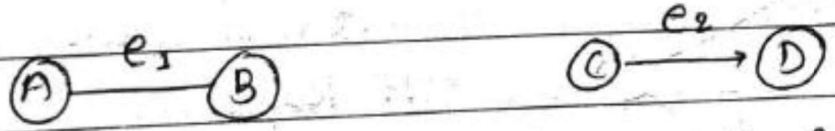
e_i is undirected edge that connect v_i and v_j then we say v_i is adjacent node of v_j and viceversa. Since $(v_i, v_j) = (v_j, v_i)$

- B is adjacent Node of A
- A is not adjacent Node of B since e_1 is directed.
- C is adjacent Node of D
- D is adjacent Node of C since e_2 is undirected.

TERMINOLOGY USED

⑤ Incidence:

Incidence is the relationship between edges and vertices. Let e_i arc/edge connects vertices v_i and v_j then, e_i is said to be incident on both the vertices v_i and v_j. The meaning of incidence remains same for both directed and undirected graph



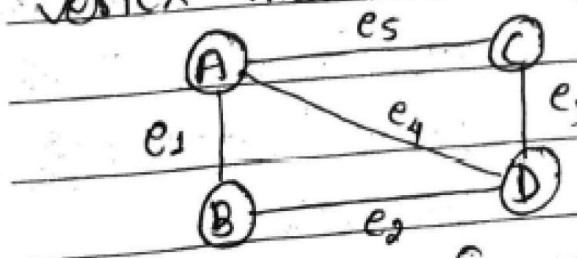
e_1 is incident on A. e_2 is incident on C

e_1 is incident on B. e_2 is incident on D.

TERMINOLOGY USED

⑥ Degree of a vertex:

The total number of edges incident on a particular vertex is called its degree. Let v be a vertex then its degree is denoted by deg(v).



deg(A) = 3.

In case of directed graph, there are two

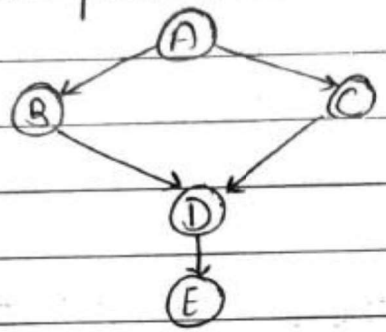
TERMINOLOGY USED

Degree for each node.

i) In degree: Number of edges comming to that vertex.

ii) Out degree: Number of edges going outside of that vertex.

So, for directed graph, the sum of indegree and outdegree of a particular vertex is called its degree.

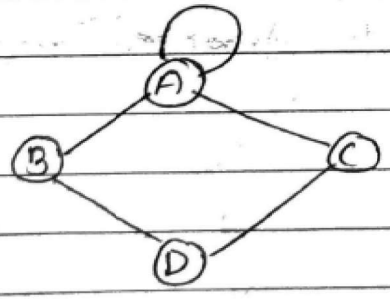


deg(B) = In deg (B) + Out deg(B)
= 1 + 1
= 2

TERMINOLOGY USED

⑦ Self loop/loop:

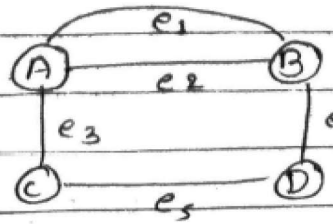
If there is an edge whose starting and end vertices are same then, it is called loop.



→ Vertex A consist of self loop.

⑧ Parallel edges:

If there are more than one edge between the same pair of vertices then they are known as parallel edges.



→ The vertex pair A and B consist of parallel edges e_1 and e_2.

TERMINOLOGY USED

(9) Cycle:

If there is a path containing one or more edges which starts from a vertex and terminates into the same vertex then path is called a cycle. A graph that has cycle is called cyclic graph otherwise acyclic graph.

Eg

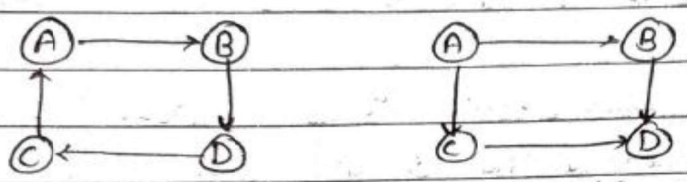


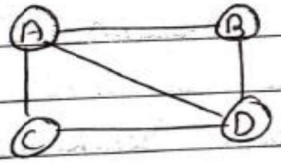
Fig: Cyclic graph

Fig: Acyclic graph.

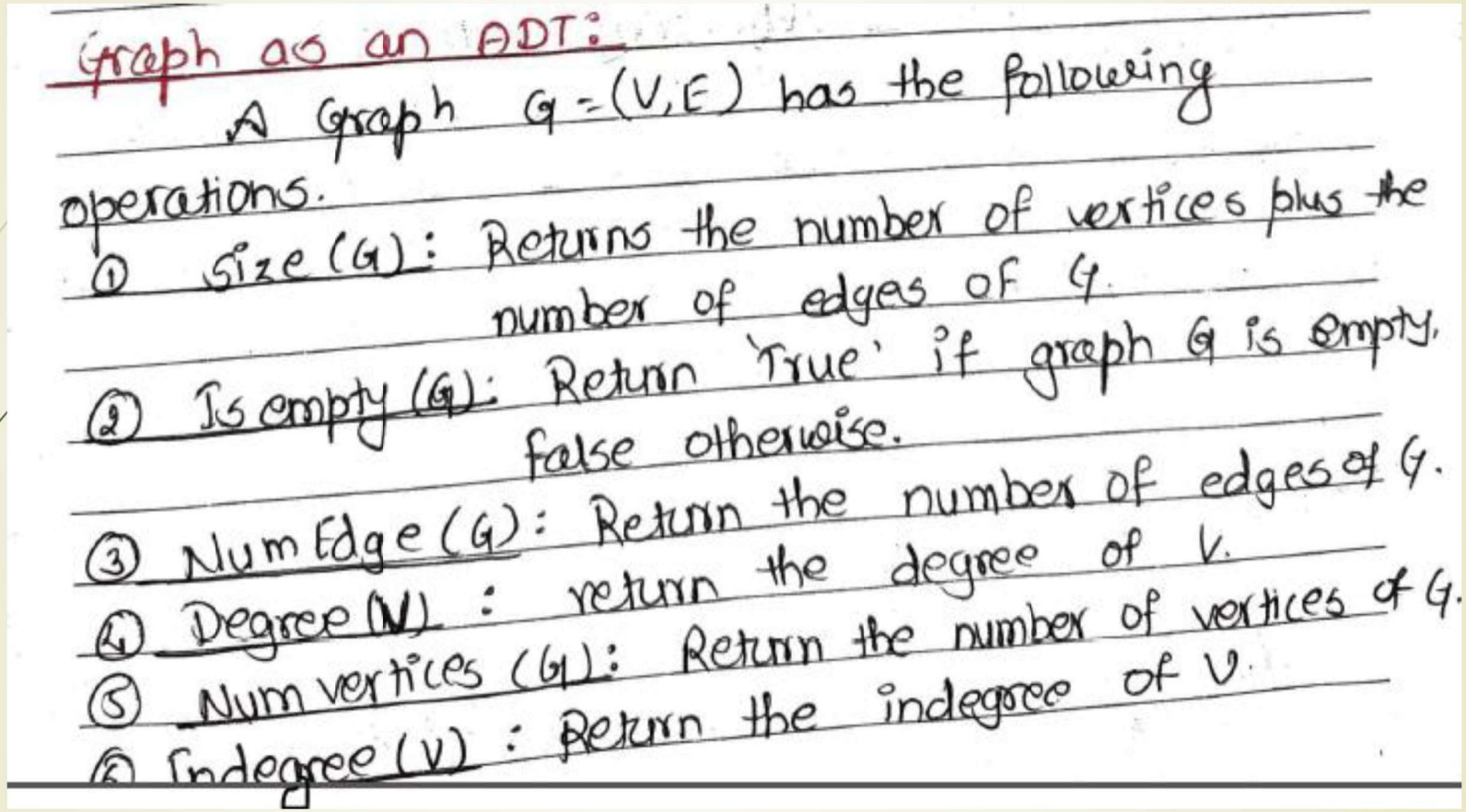
(10) Simple graph:

A graph that consists of no parallel edges between vertices and no-loop is called simple graph.

e.g.



GRAPH AS AN ADT



Graph as an ADT:

A Graph G = (V, E) has the following operations.

① Size (G): Returns the number of vertices plus the number of edges of G.

② Is empty (G): Return 'True' if graph G is empty, false otherwise.

③ NumEdge (G): Return the number of edges of G.

④ Degree (V): return the degree of V.

⑤ Num vertices (G): Return the number of vertices of G.

⑥ Indegree (V): Return the indegree of V.

GRAPH AS AN ADT

(7) Outdegree (V): Return the outdegree of V.

(8) Adjacent Vertices (V): Return the list of vertices which are adjacent to V.

(9) Isdirected (G): Return true if G is directed.

(10) Insertedge (V, U, item): Insert an undirected edge between V and U and store item at this position.

(11) Insertdirected edge (V, U, item): Insert an directed edge between V and U and store item at this position.

(12) Removeedge (e): Remove an edge.

GRAPH REPRESENTATION TECHNIQUE

Graph Representation Techniques :

1) Adjacency matrix → Static Implementation
2) Incidence Matrix
3) Adjacency List → Dynamic implementation

1) Adjacency matrix : Adjacency Matrix :

A matrix formed with the help of vertices is called adjacency matrix. Let A be a matrix of order m x N, then each of the element a_{ij} of the matrix can be represented as :

$$a_{ij} = \begin{cases} 1, & \text{if there exist a direct path bet}^n V_i \& V_j \\ 0, & \text{otherwise} \end{cases}$$

eg 1 :

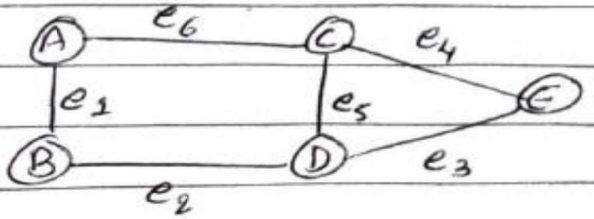
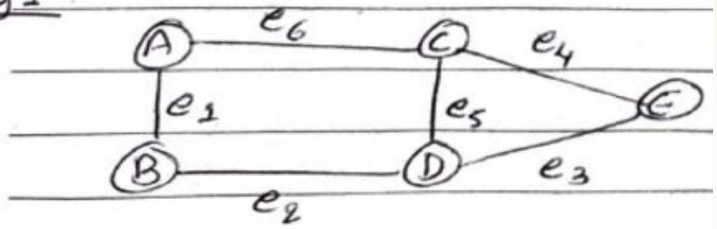


Fig : Undirected Graph.

GRAPH REPRESENTATION TECHNIQUE

Eg 1:



The adjacency matrix of above graph is.

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

Eg: 2

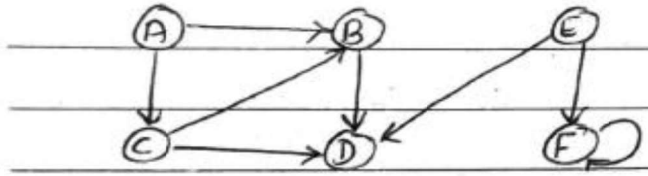


Fig: Directed graph.

NOTE: If graph is undirected then $A = A^T$ where

A = adjacency matrix

A^T = Transpose of A.

GRAPH REPRESENTATION TECHNIQUE

Eg: 2

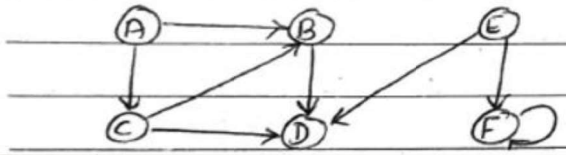


Fig: Directed Graph.

The adjacency matrix is,

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

Q Draw the directed graph corresponding to the below adjacency matrix.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

GRAPH REPRESENTATION TECHNIQUE

② Incidence Matrix:

A matrix formed with the help of vertex and edge is called incidence matrix. Let A be a matrix of order M×N, then each of the element a_{ij} can be represented as.

$$a_{ij} = \begin{cases} 1, & \text{if there exist an edge } e_i \text{ incident on vertex } V_j \\ 0, & \text{otherwise} \end{cases}$$

Eg:

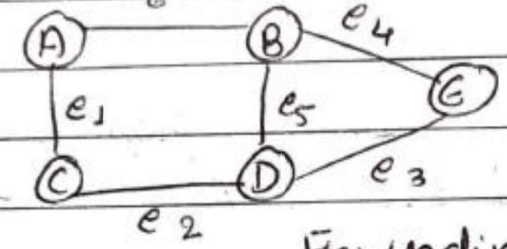


Fig: undirected graph G.

The incidence matrix of G is,

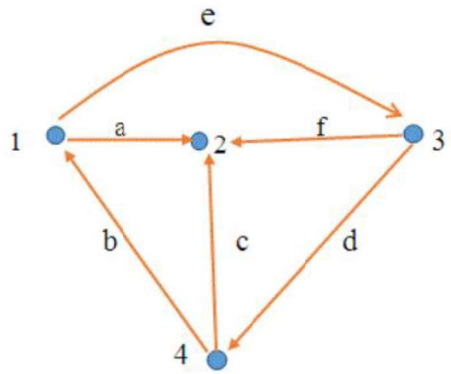
	e_1	e_2	e_3	e_4	e_5	e_6
A	1	0	0	0	0	1
B	0	0	0	1	1	1
C	1	1	0	0	0	0
D	0	1	1	0	1	0
E	0	0	1	1	0	0

GRAPH REPRESENTATION TECHNIQUE

Steps to Construct Incidence Matrix

- If a given k^{th} node has outgoing branch, then we will write +1.
- If a given k^{th} node has incoming branch, then we will write -1.
- Rest other branches will be considered 0.

Examples of Incidence Matrix



For the graph shown above write its incidence matrix.

$[A_C]$ =

nodes \ branches	a	b	c	d	e	f
1	1	-1	0	0	1	0
2	-1	0	-1	0	0	-1
3	0	0	0	1	-1	1
4	0	1	1	-1	0	0

GRAPH REPRESENTATION TECHNIQUE

③ Adjacency list:

It is the dynamic implementation of graph. Pointers are used to implement graph. In adjacency list, a list is formed ~~att~~ For all the vertex of graph. The list consist of all the adjacent nodes of a particular node. Let V be a node, and v_1, v_2, v_3 be its adjacent node then in adjacency list, a list consist of vertices v_1, v_2 and v_3 and connected to V.

eg1:

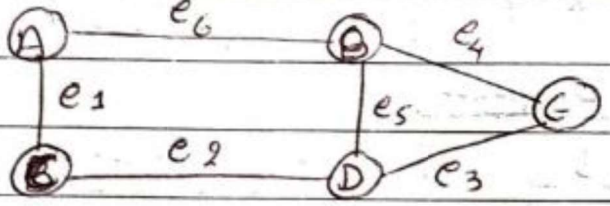


Fig: Undirected Graph G

GRAPH REPRESENTATION TECHNIQUE

Vertex	Adjacency list
A	C, B
B	A, D, E
C	A, D
D	B, C, E
E	B, D

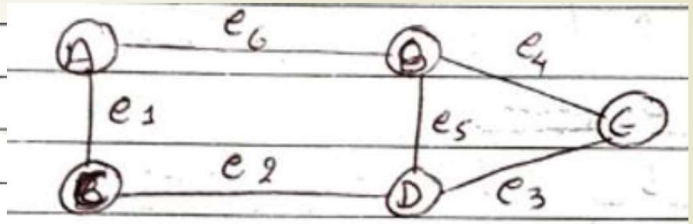
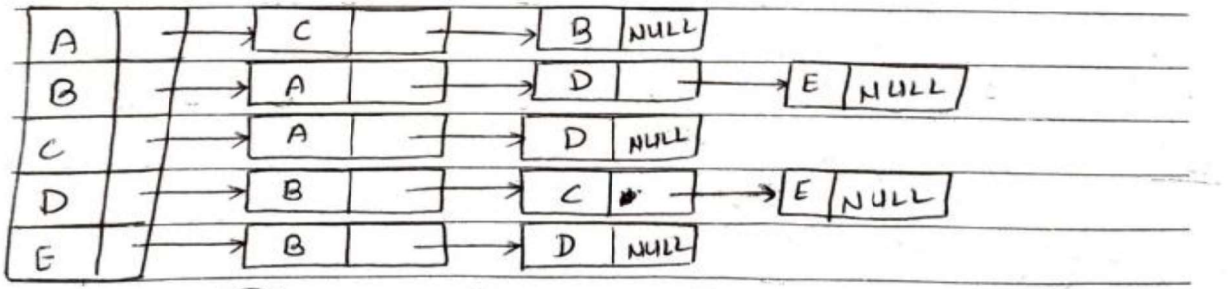
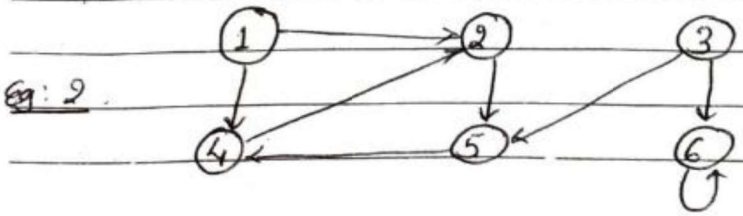


Fig: Undirected Graph G

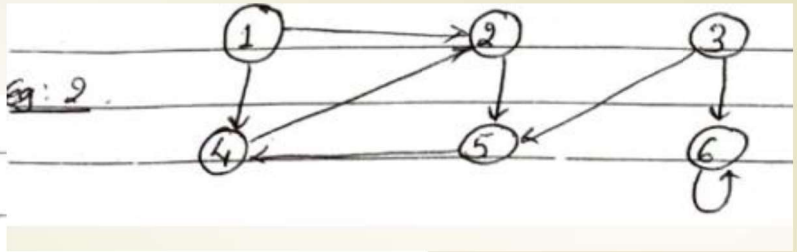
So, the adjacency list representation of this graph is.



Eg: 2.

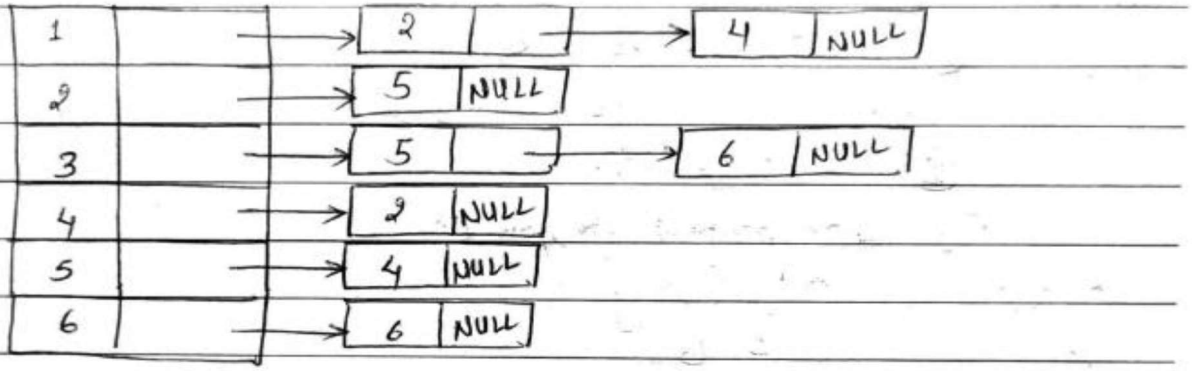


GRAPH REPRESENTATION TECHNIQUE



Vertex	Adjacency list
1	2, 4
2	5
3	5, 6
4	2
5	4
6	6

So, the adjacency list representation of this graph is,



GRAPH REPRESENTATION TECHNIQUE

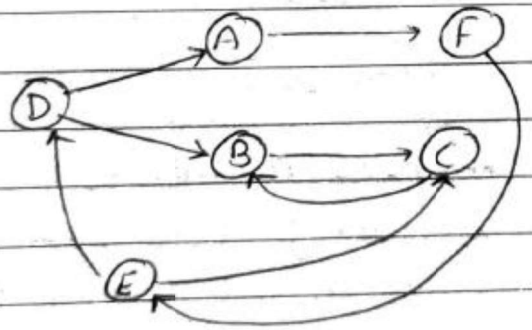
Q. Let, G be a graph represented by this adjacent list

Vertex	Adjacent list
A	F
B	C
C	B
D	A, B
E	C, D
F	E

i) Draw Graph (G).
ii) Is G adjacent & symmetric?
iii) Is G weakly connected?
iv) Give adjacency matrix of G.

GRAPH REPRESENTATION TECHNIQUE

Sol^n: The graph G for the above adjacent list is.



The adjacency matrix of G is.

$A =$

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

GRAPH REPRESENTATION TECHNIQUE

For, Symmetric, $A = A^T$

NOW,

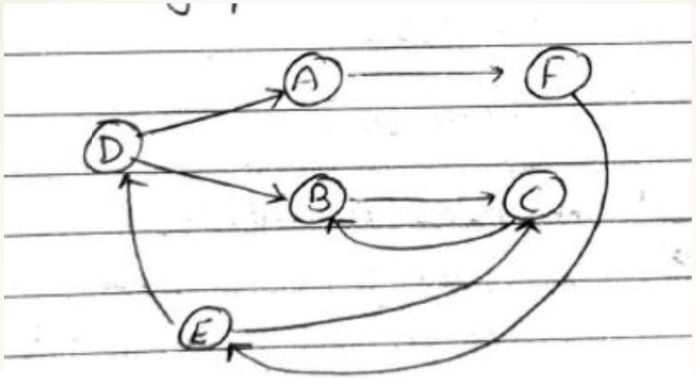
$$A^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Here,

$A \neq A^T$. So the graph is asymmetric.

Since, $A \neq A^T$. hence the graph is directed.

GRAPH REPRESENTATION TECHNIQUE



→ Now, G is not strongly connected i.e. weakly connected because there exists some vertex in G from which we can't visit all the remaining vertex. Eg: B, E etc.

→ Yes, G is cyclic [B → C → B]

[D → A → F → E → D]

Shortest Path:

A path from source vertex s to any vertex V is shortest path if there is no other path from s to V with lower cost. The sum of weights of edge of graph is called cost. The shortest path are not necessarily unique.

One of the algorithm to find shortest path is Dijkstra's shortest path algorithm. It can be used to solve TSM (Travelling sales man) problem.

Greedy Algorithm:

An optimization problem is one in which we want to find, not just a solution, but the best solution. A greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases. At each phase: we take the best we can get right now, without regard for future consequences and we hope that by choosing a local optimum at each step, we will end up at a global optimum.

Dijkstra's Algorithm

Shortest-Paths Problems: Single-Source Shortest Path Problem

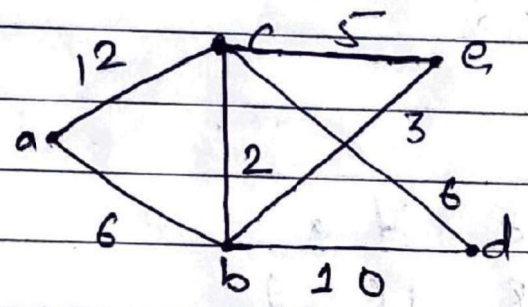
Algorithm:

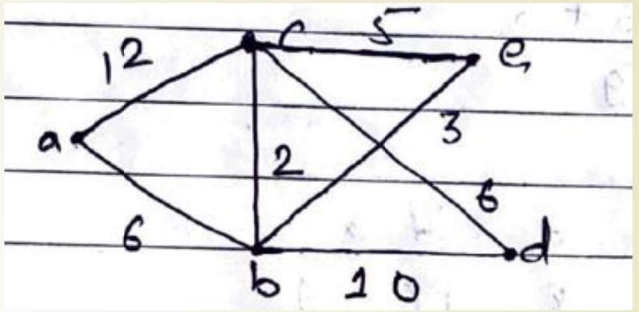
1) Mark all vertices as unknown
2) For each vertex V, keep a distance dv from source to that vertex V.
3) Initially set all vertex to ∞ except for source which is set to ds = 0.
4) Repeat step 5 to 7 until all vertices are known.
5) Select a vertex 'V' which has smallest dv among all the unknown vertices.
6) Mark V as known.
7) For each vertex W, adjacent to V.
   If W is unknown and dv + cost(V,W) < dW,

Date . .

update dw to dv + cost(V,W) i.e. Replace (Relax) all the adjacent vertex of V.

eg. Apply Dijkstra's algorithm to find the shortes path from the vertex a to each of the other vertex of the following ~~wi~~ weighted graph.

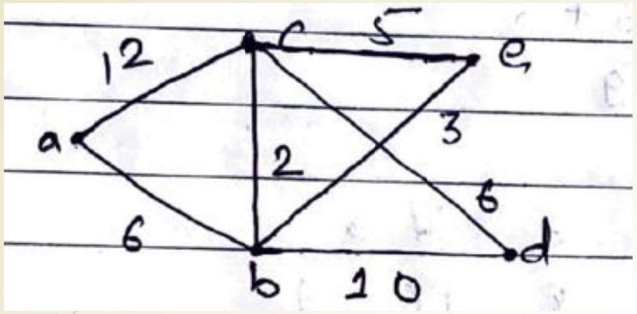




Sol^n:

Step 1: initialize initial vertex (a) to 0 and other remaining vertex to ∞.

Vertex	a	b	c	d	e
Value	0	∞	∞	∞	∞



Step II :

Now, select vertex a. The find out the adjacent vertices of a. Adjacent vertices of a are b & c.

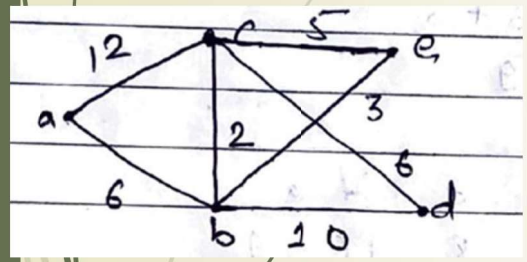
Now, find out the weight of b and c.

wt (b) = wt (a) + wt (a, b) = 0 + 6 = 6

wt (c) = wt (a) + wt (a, c) = 0 + 12 = 12

Update table as:

vertex	a	b	c	d	e
value	0	6 (a,b)	12 (a,c)	∞	∞



Step III : Select vertex b.

Now, vertices adjacent to b are c, e, d

$wt(c) = wt(b) + wt(b,c)$

$= 6 + 2$

$= 8$

$wt(d) = wt(b) + wt(b,d)$

$= 6 + 10$

$= 16$

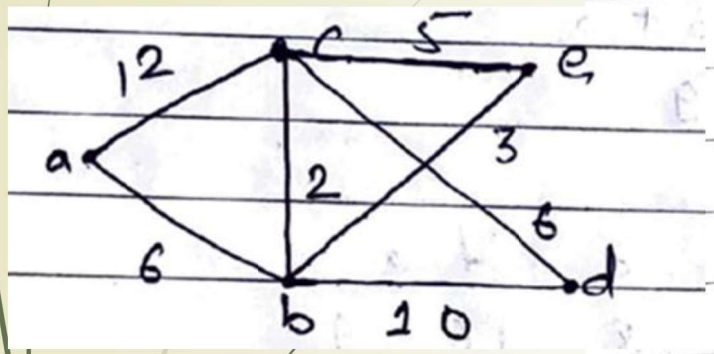
$wt(e) = wt(b) + wt(b,e)$

$= 6 + 3$

$= 9$

Update table as:

vertex	a	b	c	d	e
value	0	6	8	16	9
	(a,b)	(a,b,c)	(a,b,d)	(a,b,e)	



Step IV:

Now,

Select vertex c

$wt(e) = wt(c) + wt(c,e)$

$= 8 + 5$

$= 13$

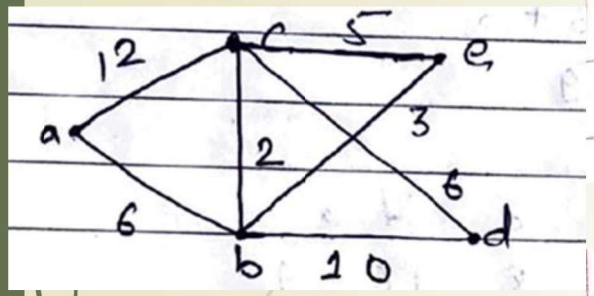
$wt(d) = wt(c) + wt(c,d)$

$= 8 + 6$

$= 14$

Update table as:

Vertex	a	b	c	d	e
value	0	6	8	14	9



Step V:

Select vertex e

~~wt~~ b and c are already selected so ~~we~~ no need to calculate weight.

Step VI:

select vertex d

b and c are already selected so no need to calculate weight.

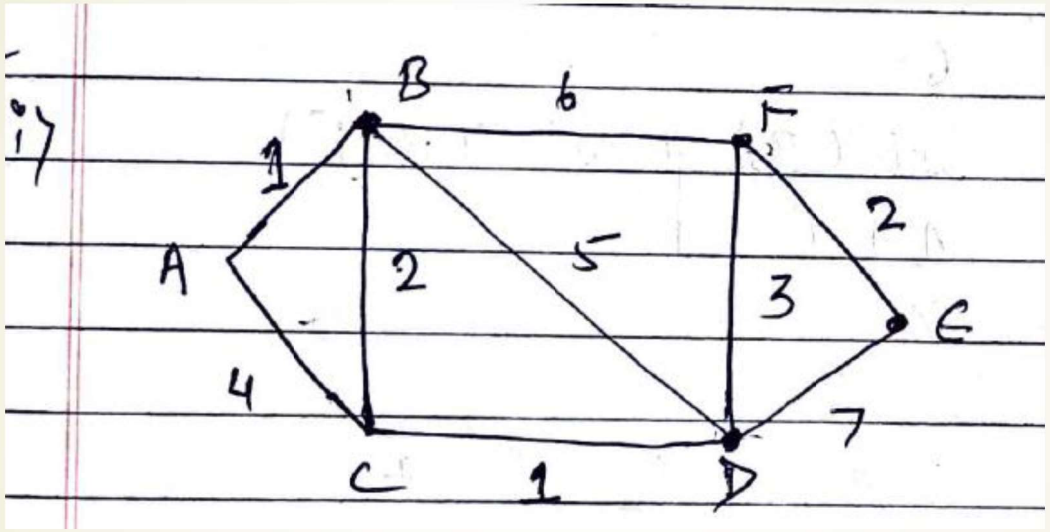
∴ The shortest path from a to c = 8

" " " a to b = 6

" " " a to d = 14

" " " a to e = 9.

Assignment



Find the shortest path from vertex A to others using Dijkstra's Algorithm

Transitive closure and Warshall's Algorithm:

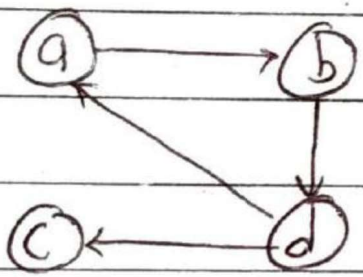
Transitive closure: → The adjacency matrix, A, of graph, G, is the matrix with elements a_{ij} such that $a_{ij} = 1$ implies there is an edge from i to j. If there is no edge then $a_{ij} = 0$.

Let A be the adjacency matrix of D and T be the adjacency matrix of transitive closure of D. T is called the reachability matrix of digraph (directed graph) D due to the property that $T_{ij} = 1$ if and only if v_j can be reached from v_i in D by a sequence of arcs (edges). In simple word its definition is as:

A path exist between two vertices i, j if

1. there is an edge from i to j; or
2. there is a path from i to j going through vertex 1; or
3. there is a path from i to j going through vertex 1 and/or 2;
4. there is a path from i to j going through vertex 1, 2, and/or 3; or
5. there is a path from i to j going through any of the other vertices.

The following figure shows that the digraph (directed graph) its adjacency matrix and its transitive closure.)



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Fig: Digraph | Adjacency matrix | Transitive closure

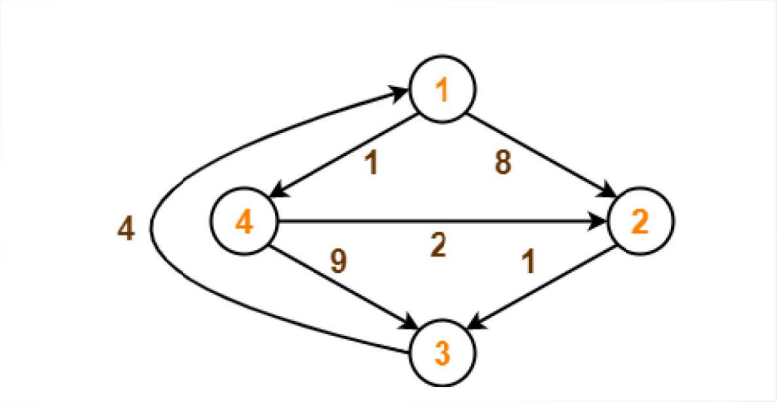
Floyd-Warshall Algorithm/WFI

- **Floyd-Warshall Algorithm/WFI** algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the directed and undirected weighted graphs.
- The graph can include negative weights. It is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm or WFI algorithm.
- It was designed by Stephen Warshall and implemented by Robert W. Floyd and P. Z. Ingerman.

Algorithm:

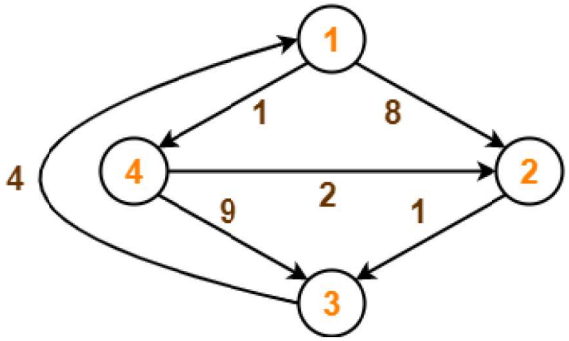
```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j])
return A
```

Example:



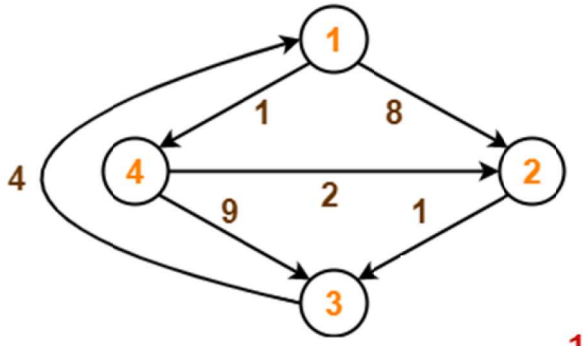
- **Solution-**
- **Step-0:**
- Remove all the self-loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self-edges nor parallel edges.

- **Steps 1:** Create a matrix D_0 of dimension n*n where n is the number of vertices.
- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.



$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

- **Step 2:**
- Now, create a matrix D_1 using matrix D_0.
- The elements in the first column and the first row are left as they are.
- The remaining cells are filled in the following way. Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex.
  - D[i][j] is filled with (D[i][k] + D[k][j]) if (D[i][j] > D[i][k] + D[k][j]).



$D_0 =$

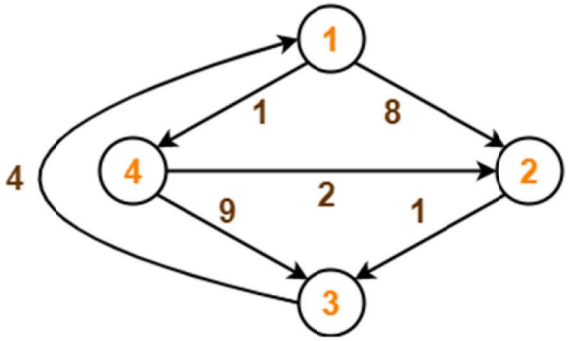
	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	0

$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

- **Step 3**: In a similar way, D_2 is created using D_1. The elements in the second column and the second row are left as they are.

  In this step, k is the second vertex (i.e. vertex 2).



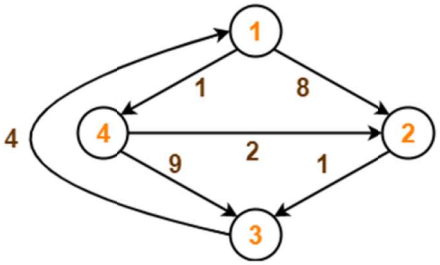
$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

- **Step 4 and 5**: Similarly, D_3 and D_4 is also created

$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0



$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

$D_4 =$

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

- The last matrix D_4 represents the shortest path distance between every pair of vertices. For example, from vertex 1 to vertex 4, cost is 1 and 1 to 3 is 4 and so on.

- **Note:**
- Dijkstra's Algorithm & Bellman Ford's Algorithm are the example of a single-source shortest i.e., given a source vertex it finds shortest path from source to all other vertices.
- Floyd Warshall Algorithm is an example of all-pairs shortest path algorithm, meaning it computes the shortest path between all pair of nodes.

Graph Traversal:

Traversing is the process of visiting each nodes in the graph in some systematic approach. There are two graph traversal methods:

1) Breadth first search (BFS)
2) Depth first search (DFS)

1. Breadth Depth first Search

→ This technique uses queue for traversing all the nodes of the graph.

In this method, first we take any node as starting node. Similar then we take all the nodes adjacent to that starting node. Similar approach we take for all other adjacent node, which are adjacent to the starting node and so on.

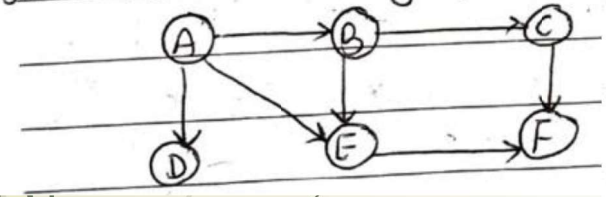
Algorithm:

① Insert starting node into the queue

② Delete front element from queue and insert all its unvisited adjacent nodes into the queue at the end and traverse them. Also, make value of visited array true for visited nodes.

③ Repeat step 2 until queue is empty.

Eg: Consider the graph belo



Eg: ① A

F R

Enqueue(A)

② B D E

F R

Dequeue()
Enqueue(B)
Enqueue(D)
Enqueue(E)
Traversed = A

③ D E C

F R

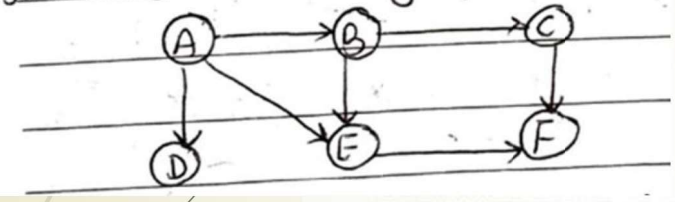
Dequeue()
Enqueue(C)
Traversed = A B

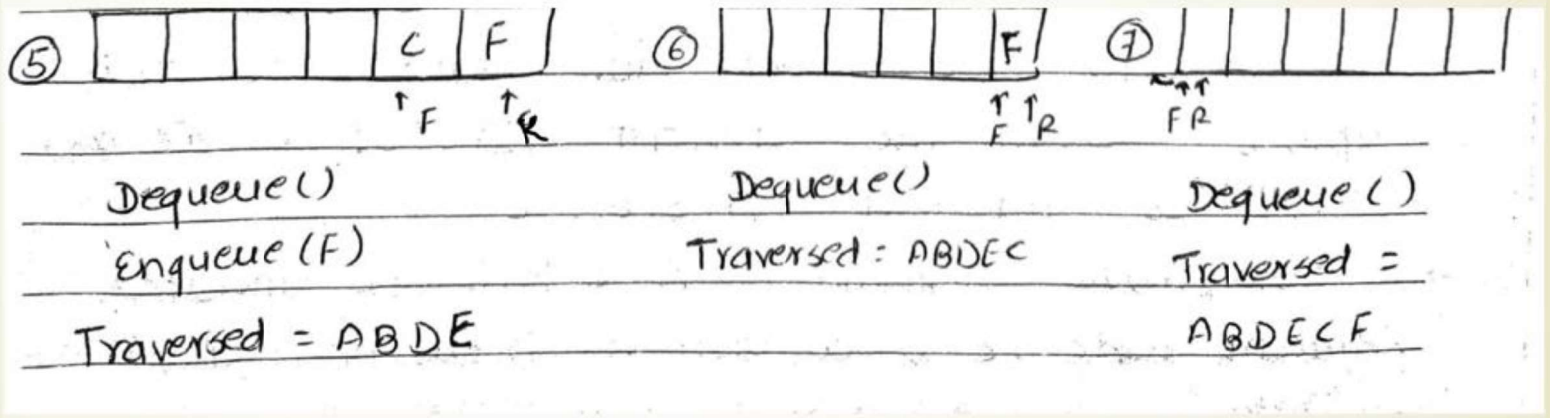
④ E C

F R

Dequeue()
Enqueue(E
Traversed = A B D

Eg: Consider the graph below:





Depth First Search:

This technique uses stack for traversing all the nodes of graph.

This technique make search deeper in the graph, if possible we can take any node as starting node. After that, we traverse starting node, we travel through its adjacent vertex and again adjacent of adjacent of starting node and so on.

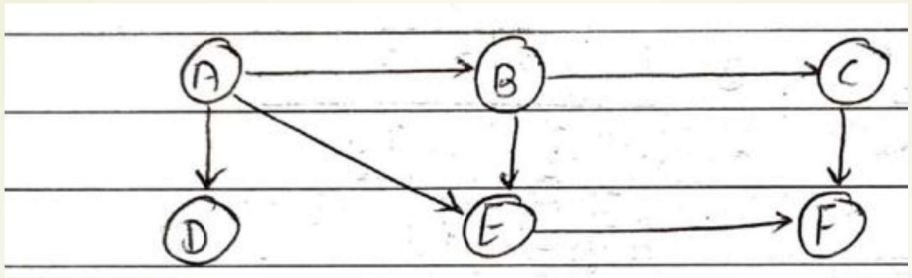
Algorithm:

① Push the starting node into the stack.

② Pop an element from stack. if it has not been traversed then traverse it, if it has already been traversed then ignore it. After traversing make the value of visited array true for this node.

③ Now push all the unvisited adjacent nodes of the popped element on stack. Push the element even if it is already on the stack.

④ Repeat step 2 and 3 until stack is empty.

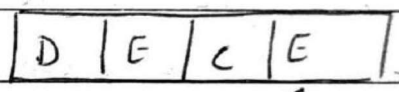


Eg:

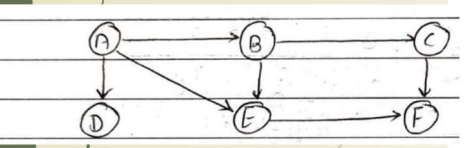
① push(A)

② push(B)
push(E)
push(D)
pop()
Traversed = A

③ pop()
push(C)
push(E)
Traversed = A B



{ pop() and push adjacent nodes of popped element.



④ | D | E | C | F |
↑ TOP

POP()
push adjacent nodes of popped element
i.e. push(F)

Traversed = A B E

⑤ | D | E | C | |
↑ TOP

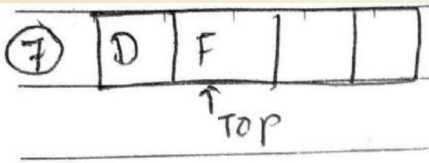
pop()
push adjacent nodes of popped element i.e NULL

Traversed = A B E F

⑥ | D | E | | |
↑ TOP

POP(), push adjacent nodes of popped element i.e. ~~push~~ NULL

Traversed = A B E F C

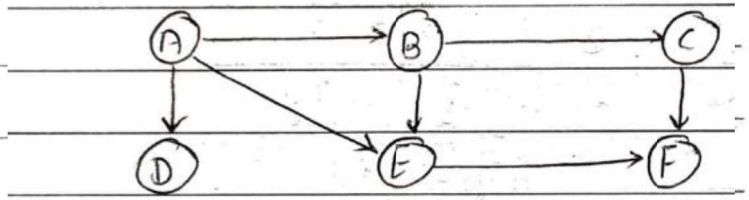
⑦ 

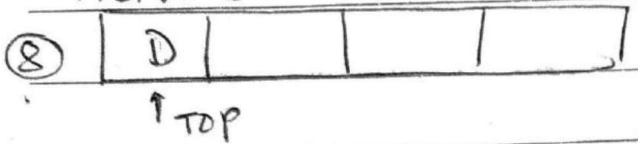
pop() and push adjacent nodes of popped element

i.e. push(F)

Here, E is already visited so ignore it.

Traversed = A B E F C

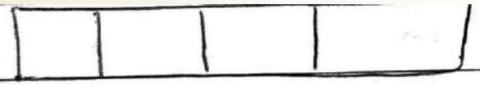


⑧ 

pop() and push adjacent nodes of popped element i.e NULL

push Here, F is already visited so ignore it.

Traversed = A B E F C

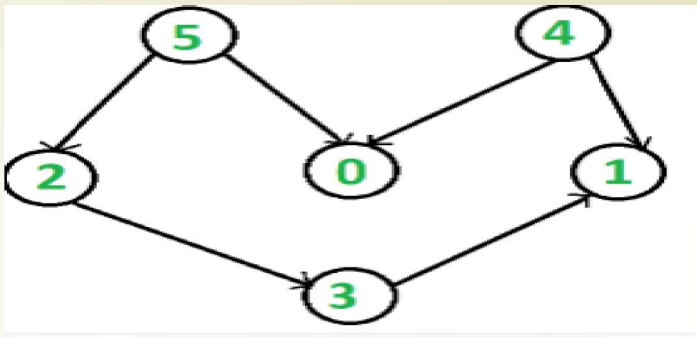
⑨ 

pop() and push adjacent nodes of popped element

i.e. NULL

Traversed = A, B, E, F, C, D

Topological Sort



- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.
- For example, a topological sorting of the following graph is "5 4 2 3 1 0".
- There can be more than one topological sorting for a graph.
- For example, another topological sorting of the following graph is "4 5 2 3 1 0".
- The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

Algorithm:

1 - Compute the indegrees of all vertices

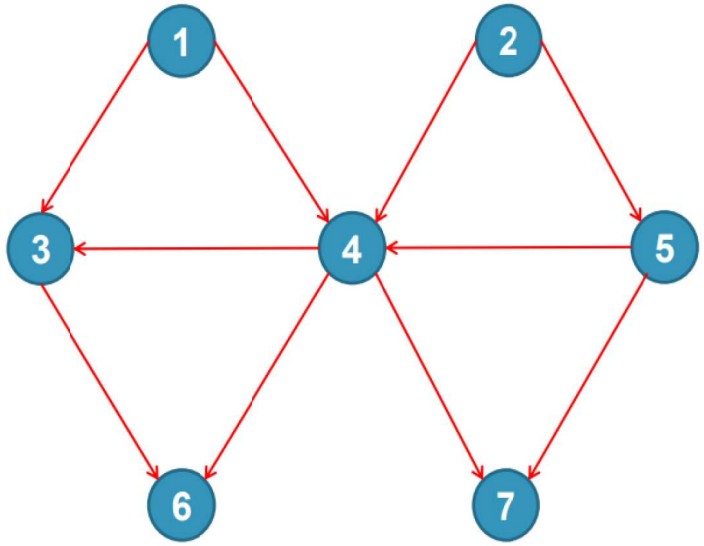
2 - Find a vertex U with indegree 0 and print it (store it in the ordering)If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.

3 - Remove U and all its edges (U,V) from the graph.

4 - Update the indegrees of the remaining vertices.

5 - Repeat steps 2 through 4 while there are vertices to be processed

Example:



Identify nodes having in degree '0'

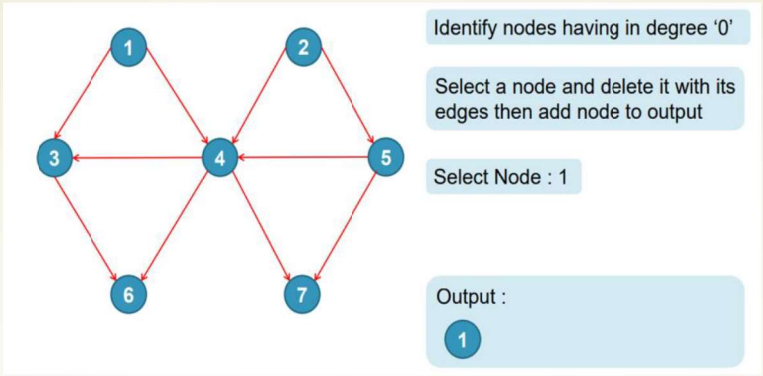
Select a node and delete it with its edges then add node to output

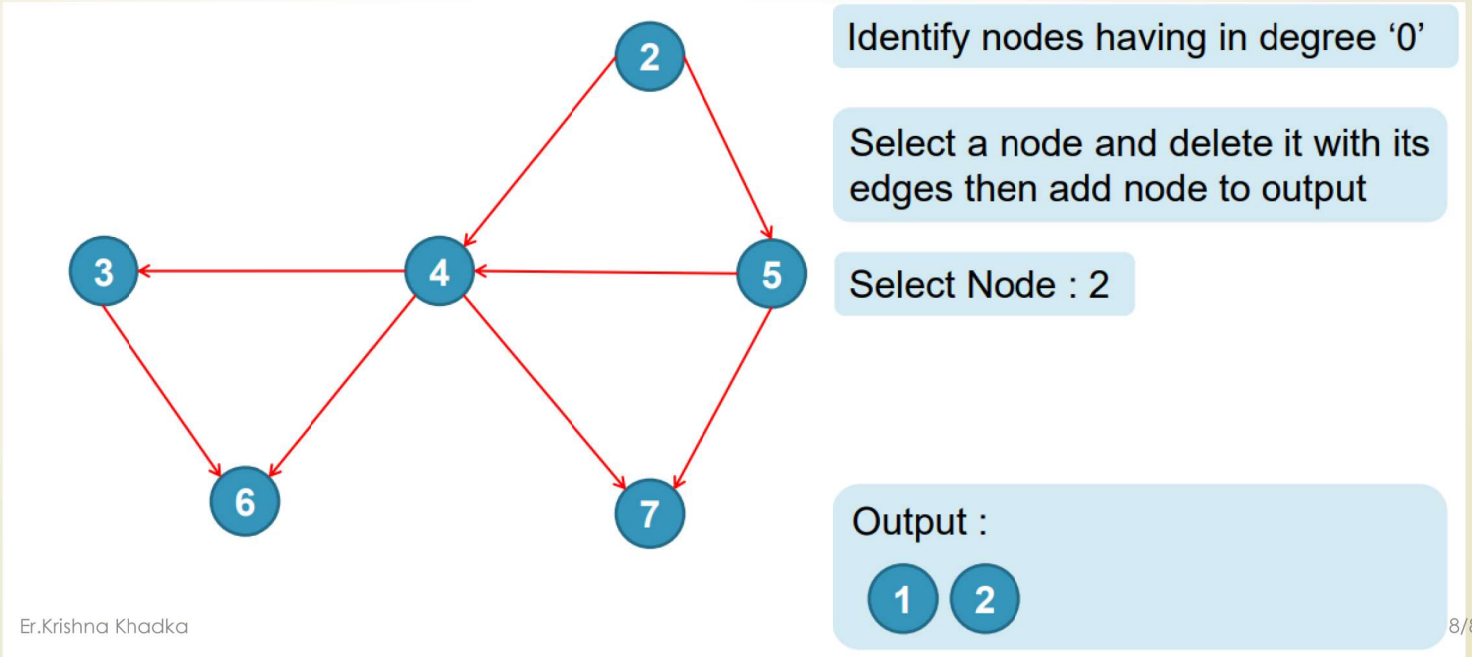
Select Node : 1

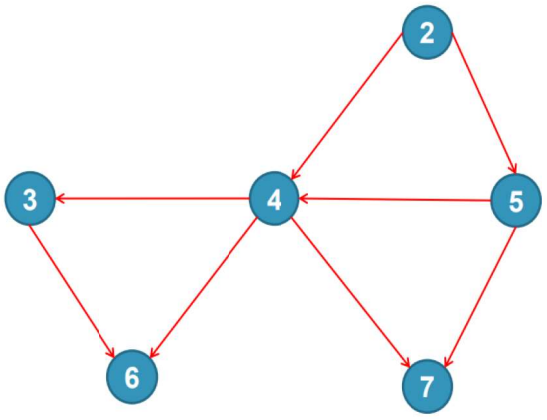
Output :

1

Example:


1
2
3
4
5
6
7
Identify nodes having in degree '0'
Select a node and delete it with its edges then add node to output
Select Node : 1
Output :
1


2
3
4
5
6
7
Identify nodes having in degree '0'
Select a node and delete it with its edges then add node to output
Select Node : 2
Output :
1
2



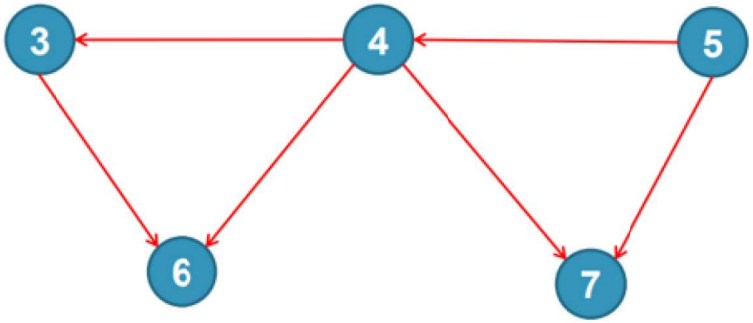
Identify nodes having in degree '0'

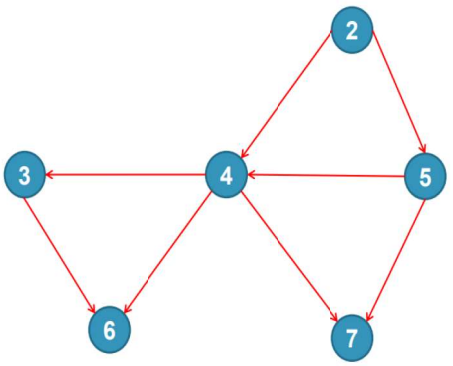
Select a node and delete it with its edges then add node to output

Select Node : 2

Output :

1 2





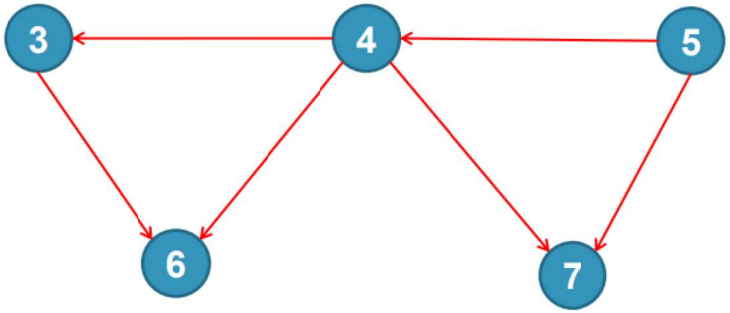
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 2

Output :

1 2



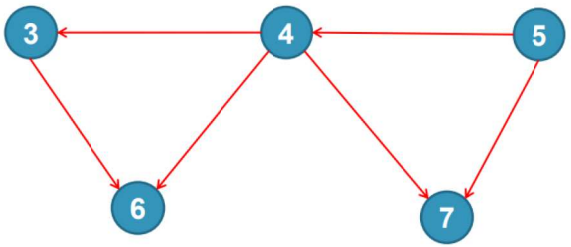
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 5

Output :

1 2 5



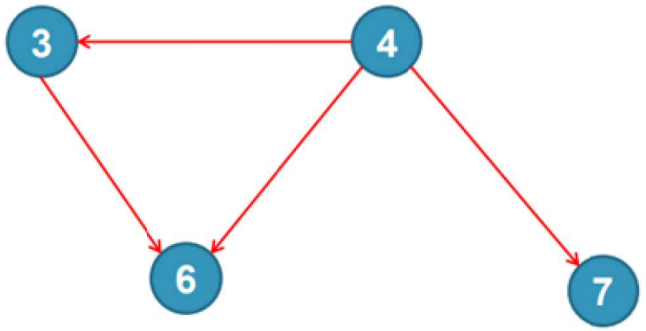
Identify nodes having in degree '0'

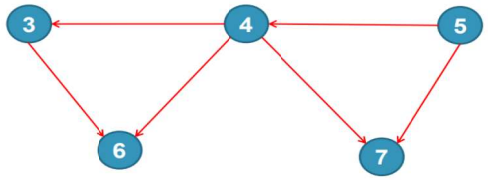
Select a node and delete it with its edges then add node to output

Select Node : 5

Output :

1 2 5





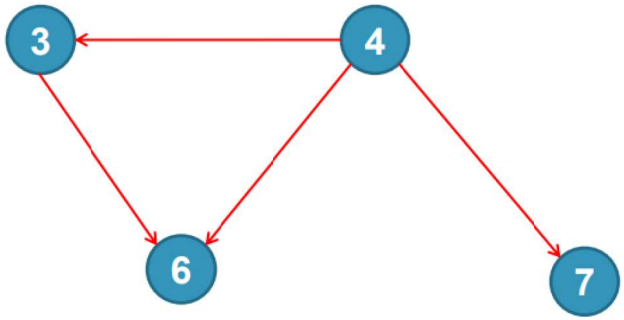
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 5

Output :

1 2 5



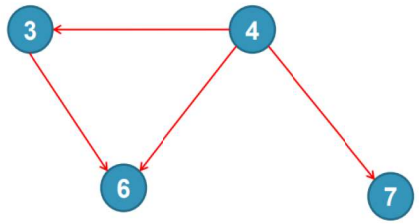
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 4

Output :

1 2 5 4



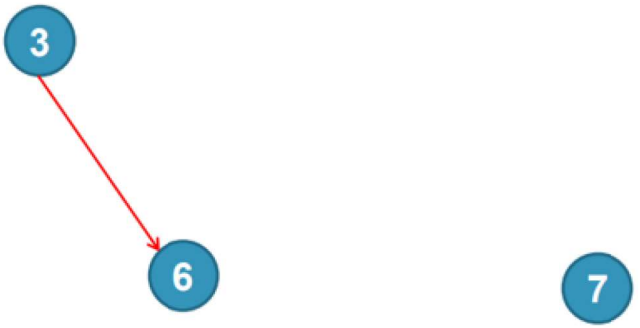
Identify nodes having in degree '0'

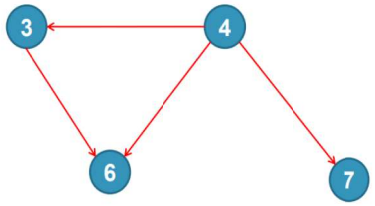
Select a node and delete it with its edges then add node to output

Select Node : 4

Output :

1 2 5 4





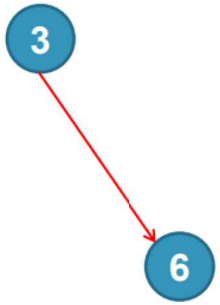
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 4

Output :

1 2 5 4



Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 3

Output :

1 2 5 4 3

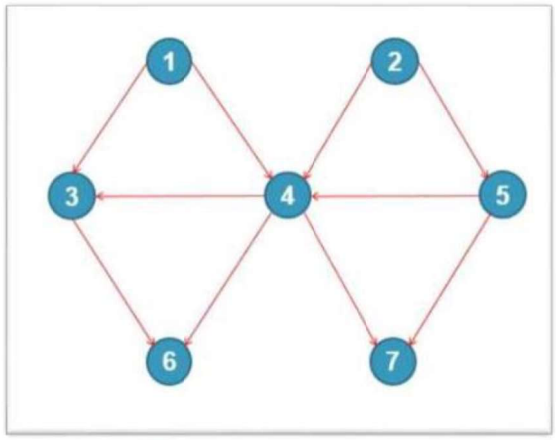
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 3

Output :

1 2 5 4 3



6

7

Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 7

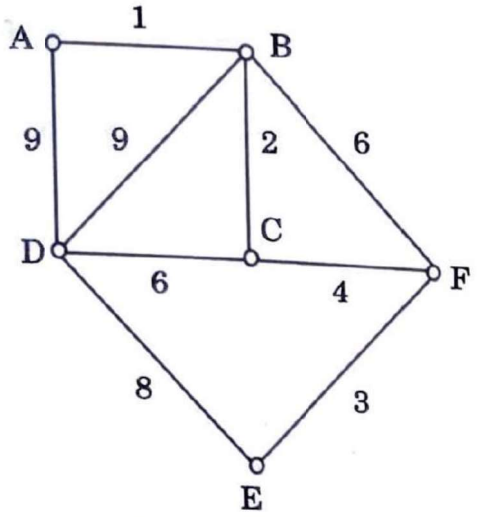
Output :

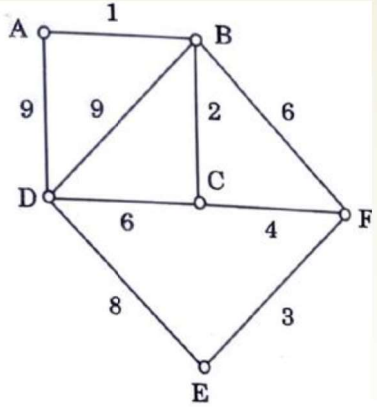
1 2 5 4 3 6 7

Minimal Spanning Trees

Let G be a connected weighted graph. The weight of a spanning tree of G is the sum of the weights of the edges which are included on that spanning tree. Thus, a minimal spanning tree of G is a spanning tree of G with minimum possible weight among all possible spanning trees of that graph. Consider figure which shows six cities and cost of lying

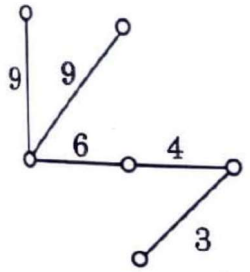
railway links between certain pair of cities. We want to set up railway links between the cities at minimum costs.



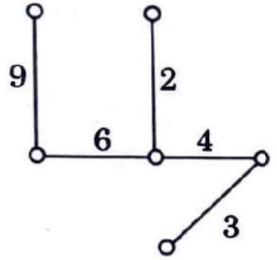


Now, we consider some weighted spanning trees of the weighted graph of figure given below.

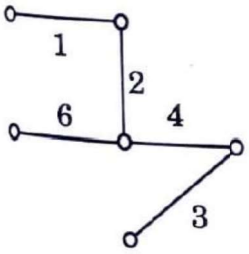
(i)



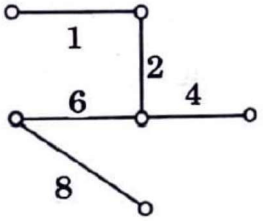
(ii)



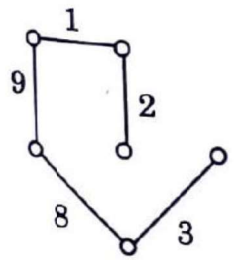
(iii)



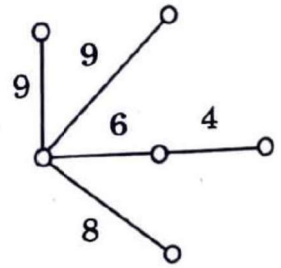
(iv)



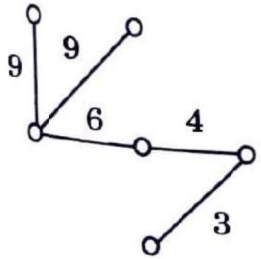
(v)



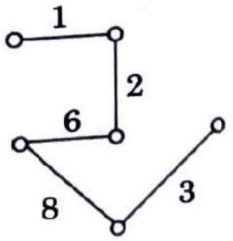
(vi)



(vii)



(viii)



Among the spanning trees in figure, spanning tree (iii) is defined as the minimal spanning tree since its weight is minimum as compared to the weights of the other spanning trees.

It is unnecessary to find the weights of all the possible spanning trees of a given weighted graph to determine its minimal spanning tree. There are several algorithms for determining the minimal spanning tree of the given weighed graphs. However we shall discuss only two algorithms here.

Kruskal's Algorithm

The algorithm involves the following steps:

Step 1: List all the edges of G with non–decreasing order of their weights.

Step 2: Select an edge of minimum weight (if there are more then one edge of minimum weight, arbitrarily choose one of them). This will be the first edge of T.

Step 3: At each stage, select an edge of minimum weight from all the remaining edges of G if it does not form a cycle with the previously selected edges in T. Then add the edge to T.

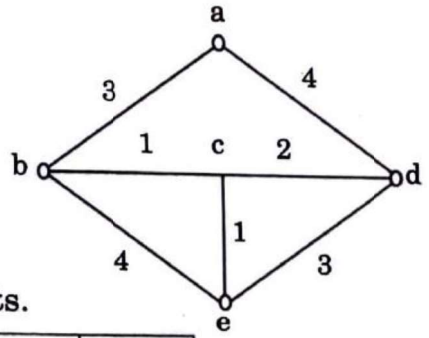
Step 4: Repeat step 3 until n –1 edges have been selected.

Pseudo-code for kurskal's Algorithm

```
KruskalMST(G)
{
    T={V}// Set of vertices
    E= Set of edges sorted in non-decreasing order of their weight
    while(|T| <n-1 and E!=NULL)
    {
        select(u,v) from the E in order
        remove (u,v) from E
        if(u,v) does not create cycle in T
        T=TU{(u,v)
    }
}
```

Example

Show step by step, how Kruskal's algorithm can be used to find a minimal spanning tree for G of figure.



Solution:

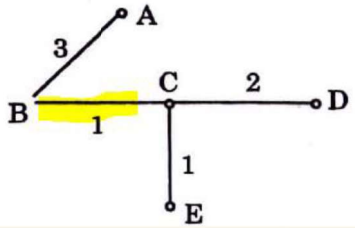
Step 1: List the edges with non–decreasing order of their weights.

Edge	(b, c)	(c, e)	(c, d)	(a, b)	(d, e)	(a, d)	(b, e)
Weight	1	1	2	3	3	4	4

Step 2: The edge (b, c) has the smallest weight, so include it in T.

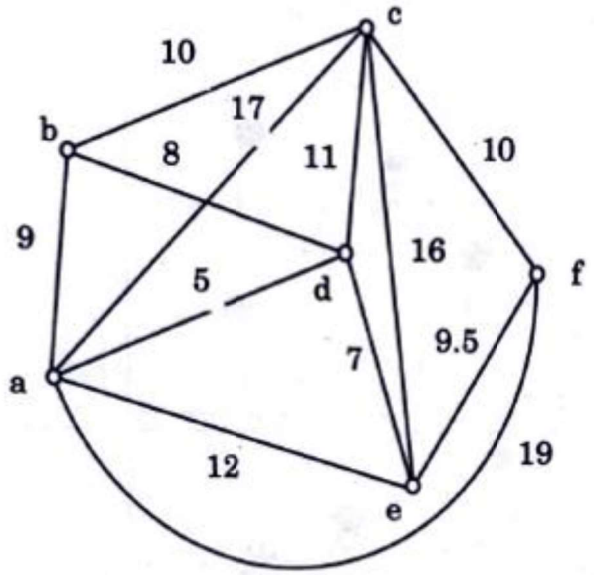
Step 3: An edge with next smallest weight is (c, e), so include it in T.

Step 4: Similarly next smallest weight including edge is (c, d). Since it does not form a cycle with the existing edges in T, include it in T. Similarly we follow this process until T has $5 - 1 = 4$ edges which are shown in figure and weight is: 1+1+2+3=7



Example

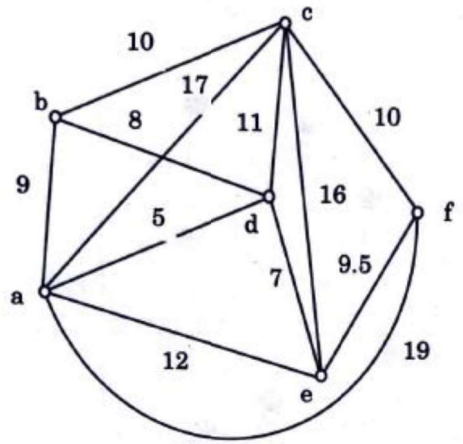
Find a minimal spanning tree of weighted graph G in figure, using Kruskal's algorithm.



Solution:

Example

Find a minimal spanning tree of weighted graph G in figure, using Kruskal's algorithm.

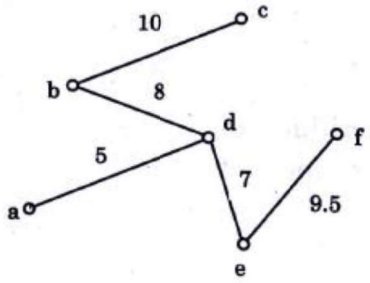


Solution:

The graph G has six vertices. Hence, any spanning tree of G will have $(6 - 1) = 5$ edges.

Edges	(a,d)	(d, e)	(b,d)	(a, b)	(e, f)	(b, c)	(c, f)	(d, c)	(a, e)	(c,e)	(a, c)	(a, f)
Weight	5	7	8	9	9.5	10	10	11	12	16	17	19
Select	Yes	Yes	Yes	No	Yes	Yes	No	No	No	No	No	No

Thus, the minimal spanning tree of G contains the edges {(a, d), (d, e), (b, d), (e, f), (b, c)}. This minimal spanning tree of weight 39.5 is shown in figure.



Example

Prim's Algorithm

This algorithm involves the following steps:

Step 1: Select any vertex in G. Among all the edges which are incident to it, choose an edge of minimum weight. Include it in T.

Step 2: At each stage, choose an edge of smallest weight joining a vertex which is already included in T and a vertex which is not included yet so that it does not form a circuit. Then include it in T.

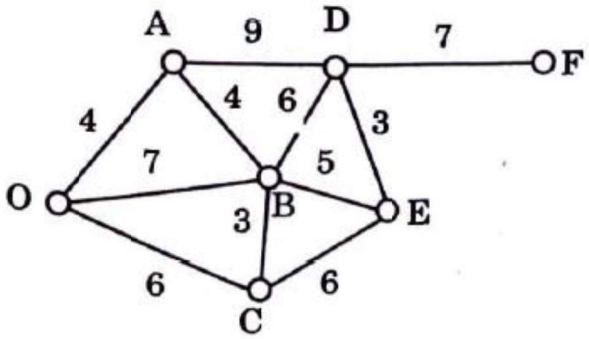
Step 3: Repeat until all the vertices of G are included with n – 1 edges.

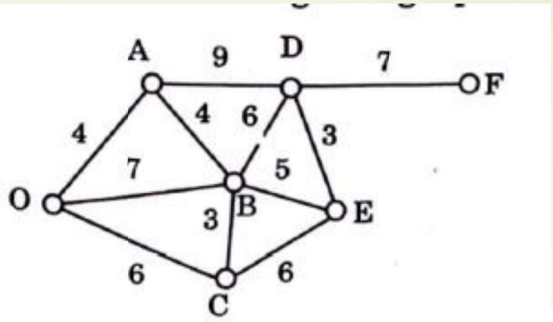
The pseudo-code for prim's algorithm

```
primsMST(G)
{
        T={}      //set of edges of MST
        S={s}     //s is the randomly chosen vertex
        while(S!=V)
        {
          e={u,v} //an edge of miimum weight incident to vertices in s and not forming
        a simple cycle in T.
        T=T U{u,v}
        S=SU{v}
        }
}
```

Example

Find the minimal spanning tree of the weighted graph G of figure using Prim's algorithm.





Solution:

Step 1: We choose a vertex O. Now edge with smallest weight incident on O is OA. So we include OA in T.

Step 2: Now w(OB) = 7, w(OC) = 6, w(AB) = 4 and w(AD) = 9. We choose the edge AB since it is minimum. Then we include it in T.

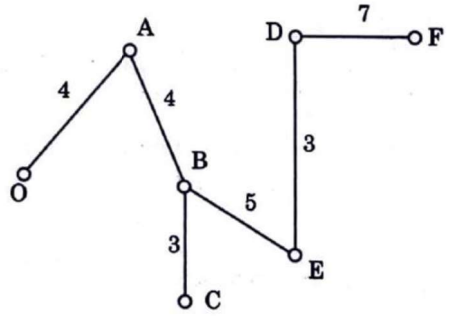
Step 3: Now w(BD) = 6, w(BE) = 5, w(BC) = 3, w(AD) = 9, w(OC) = 6. So we choose B and include it in T.

Step 4: Again w(OC) = 6, w(CE) = 6, w(BE) = 5. So we choose BE and include it in T.

Step 5: Since w(ED) = 3, we include it in T.

Step 6: Now w(DF) = 7, and further we have no choice since if we choose any remaining edge whose weight is less or equal to seven, it forms a circuit. So we include DF in T.

So we have (7 – 1) = 6 edges in a minimal spanning tree, which is shown in figure and weight is: 3 + 3 + 4 + 4 + 5 + 7 = 26



Thank You