Graph

Introduction

A Graph G consists of two things
i) Set of vertices V called nodes.
ii) Set of edge such that each edge is represented by a pair of vertices.

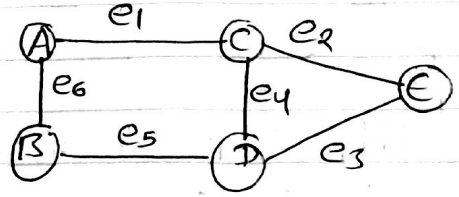


fig I

So A graph G = (V, E) consist of set of vertices V and set of edges E such that each edge in E is a connection between a pair of vertices in V.

fig I, is a graph consisting of 5 vertices (A, B, C, D, E) and 5 edges (e_1, e_2, e_3, e_4, e_5).

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 pairs, there are two types of graph.

1. Directed Graph (diagraph)
2. Undirected Graph

[redacted]:

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

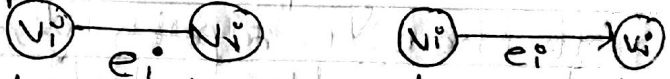
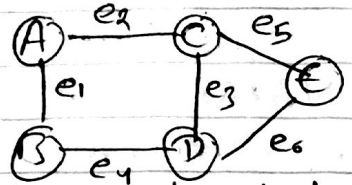


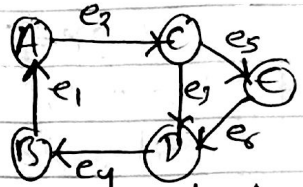
fig: unordered pair fig: Ordered pair

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

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



big: Undirected Graph

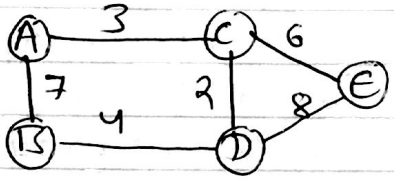


big: Directed Graph

Terminology Used

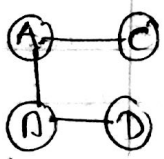
① Weighted Graph

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

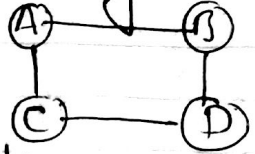


② Connected Graph

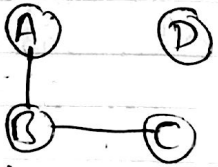
- A graph is said to be connected if there is a path from any node to any other node/vertex.



big: Connected



big: Connected Graph



big: Disconnected Graph

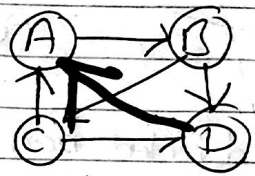
Types

① Strongly connected graph

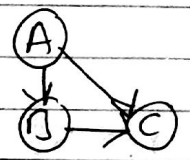
② Weekly connected graph

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

eg:



big: Strongly Connected



big: Weekly Connected

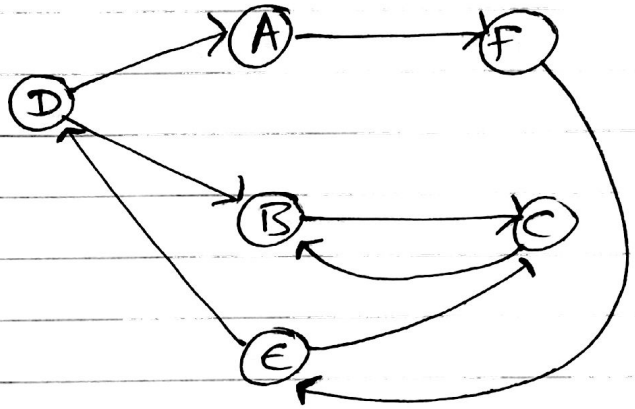
*See this problem after completion of all to terminology and Representation technique.

→ 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 G.
ii) Is G adjarent?
iii) Is G weekly connected?
iv) Is G cyclic?
v) Give adjarency matrix of G.

Soln:- The graph G for the above adjarent list is



#

- The adjarency matrix of G is

$$A = \begin{array}{c|cccccc} & A & B & C & D & E & F \\ A & 0 & 0 & 0 & 0 & 0 & 1 \\ B & 0 & 0 & 1 & 0 & 0 & 0 \\ C & 0 & 1 & 0 & 0 & 0 & 0 \\ D & 1 & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 1 & 1 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

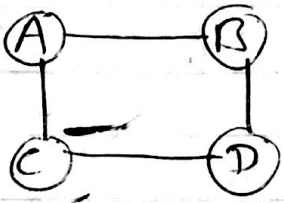
#

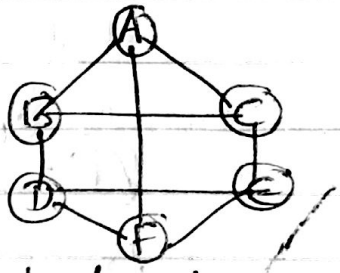
- If the graph is undirected then the adjacency matrix is symmetric i.e. $A = A^T$

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

Since $A \neq A^T$, hence graph is directed.
No, G is not strongly connected i.e is weekly connected because there exists some vertex in G from which we can't visit all the remaining vertex, Example B, E etc.
Yes G is cyclic. [B → C → B, D → A → F → G → D]

③ Regular Graph:
A Graph is Regular if every node is adjacent to the same number of vertices.



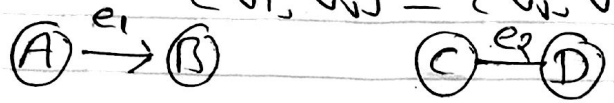


Above graph is Adjacent Graph because each node is adjacent to two nodes / vertices.

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

e_i is undirected edge that connect v_i and v_j then we say v_i is adjacent node of v_j and viceverse 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.

(5) Incidence

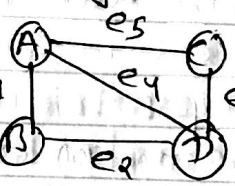
Incidence is the relationship between edges and vertices. Let e_i arc / edge connects vertices v_i and v_j then, e_i is said to be incident on both the vertices v_i and v_j. The meaning of incidence remains same for both directed and undirected graph.

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

(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).

eg:



deg (A) = 3

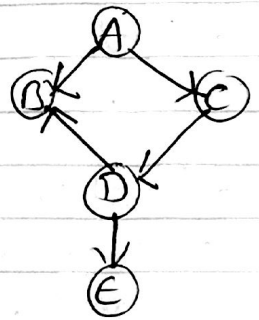
In case of directed graph, there are two degree for each node.

i) In degree – Number of edges coming 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.

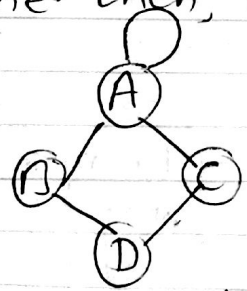
eg:



$$\deg(B) = \text{indeg}(B) + \text{outdeg}(B)$$
$$= 2 + 0$$
$$= 2$$

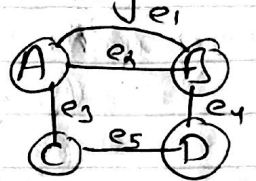
⑦ Self loop / loop:

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.

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

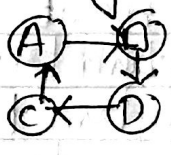


fig: Cyclic Graph

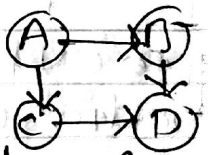
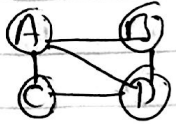


fig: Acyclic Graph.

⑩ Simple Graph:
A graph that consist of no parallel edges between vertices and no loops is called simple graph.

eg: 

The Graph 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 IsEmpty(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 NumVertices(G): Return the number of vertices of G.

⑥ InDegree(V): Return the indegree of V

⑦ OutDegree(V): " " outdegree " "

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 Insertdirectedge(V, U, item): Insert an directed edge betn V and U and store item at this position

⑫ Removeedge(e): Remove an edge e

Graph Representation Techniques

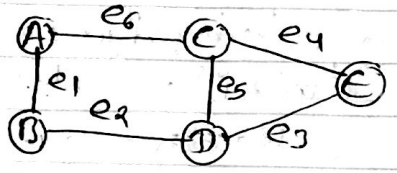
① Adjacency matrix] → static implementation
② Incidence matrix]
③ Adjacency list] → Dynamic implementation

① Adjacency matrix:

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

$$a_{ij} = \begin{cases} 1, & \text{if there exist a direct path between } v_i \text{ to } v_j \\ 0, & \text{otherwise.} \end{cases}$$

eg: 1

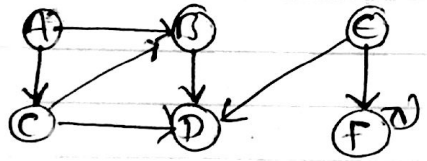


big: Undirected Graph

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



big: Directed Graph

Note: If graph is directed then $A = A^T$ where A = adjacency matrix, A^T = transpose of A.

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

Assn: Draw the directed graph coresponding to the below adjacency matrix.

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

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:

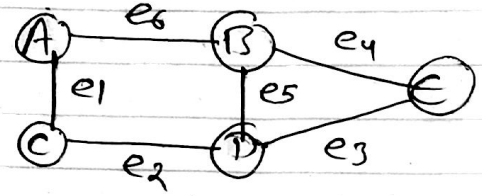
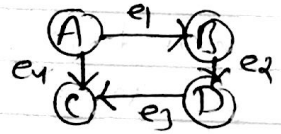


fig: Graph G

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

eg: ii



Incidence matrix =

	e_1	e_2	e_3	e_4
A	1	0	0	1
B	1	1	0	0
C	0	0	1	1
D	0	1	1	0

(3) Adjacency list

It is the dynamic implementation of graph. Pointers are used to implement graph. In adjacency list, a list is burned 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.

eg: 1

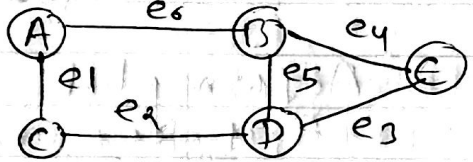
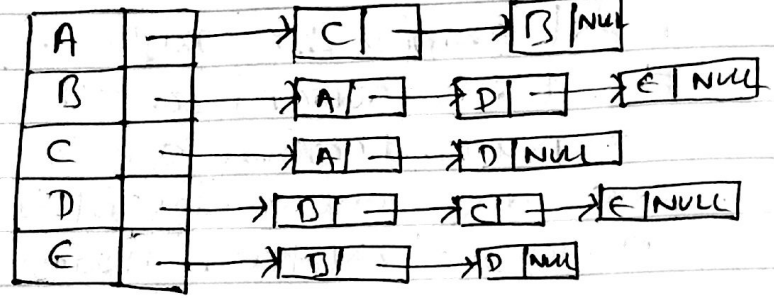


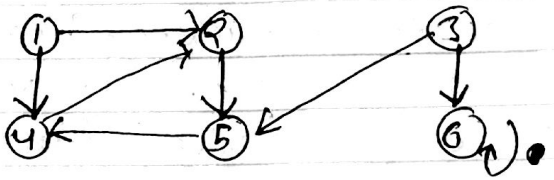
fig: Graph G

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

Note: (A) → (B) then B is adjacent Node of A

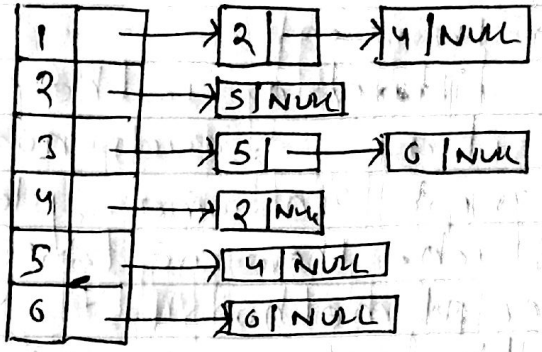
So the adjacency list representation of this graph is



eg: 

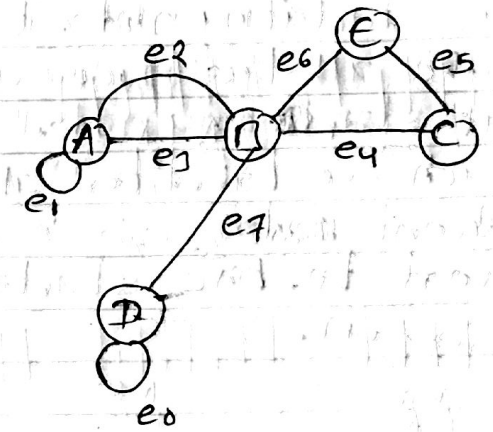
Vertex	Adjacency list
1	2, 4
2	5
3	5, 6
4	2
5	4
6	6

So the adjacency list representation of this graph is



Draw the graph corresponding to the incidence matrix

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$



Graph Traversal

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

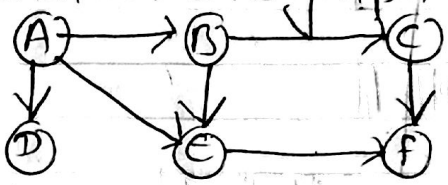
① Breadth first search (BFS)
② Depth First Search (DFS)

① Breadth First Search

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

In this first we take any node as starting node. 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.

eg: Consider the graph below



Let us take A as starting node. So first we traverse A, then we traverse all the adjacent nodes of A in any order. Suppose we traverse in B, D, E order, so our traversal is A, B, D, E. Now we traverse all nodes adjacent to B, then all nodes adjacent to D and finally of E. we should ignore it any node is traversed previously. So our final traversal is

A, B, D, E, C, F

Algorithm: ① Insert starting Node into the queue
② Delete front element from queue and insert all its unvisited neighbours / adjacent nodes into the queue at the end and traverse them. Also make value of visited away true for visited nodes
③ Repeat step 2 until queue is empty.

#①

Enqueue(A)

Traversed = A

②

Dequeue()

Enq-e(B)
Enq-e(D)
Enqueue(E)

Traversed = A

③

Dequeue()
Enqueue()

Traversed = AB

Dequeue()

Traversed = ABD

Dequeue()
Enqueue(F)

Traversed = ABDE

Dequeue()

Traversed = ABDEC

Traversed =

ABDECF ≠

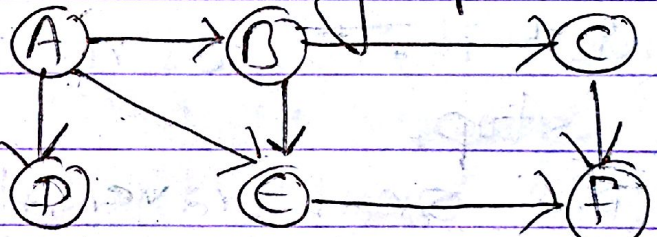
(2)

(2) 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 we traverse starting node, we travel through its neighbour vertex and again neighbour of neighbour of starting node and so on.

eg: Consider the graph below.



The traversel in depth first search is

A, B, E, F, C, D

How? → Select A as starting Node. Traverse it. Visit its adjacent Node. Suppose B is visited, then visit its adjacent node. Suppose E is visited, then visit its adjacent node i.e F.

Traversal is A, B, E, F

Now move backward, F has no adjacent node, E has but already visited, B has C unvisited adjacent node. So visit it. Now check again for C. C has F as adjacent node so stop. Now again move Backward up to B. Now all the ~~node~~ adjacent node of B ~~has~~ has been visited.

Traversal = A, B, E, F, C

Now move backward to A, A has one unvisited adjacent node i.e D so visit it. D has no adjacent node so stop.

So final traversal is

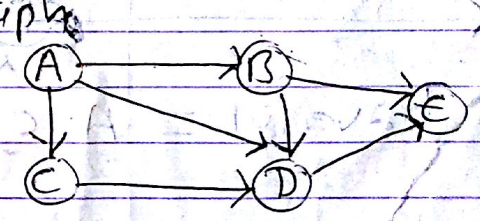
A, B, E, F, C and D //

Algorithm

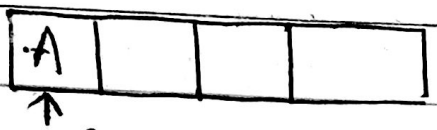
1. Push the starting node into the stack.
2. Pop an element from stack, if it has not been traversed then traverse it, if it has already been traversed then ignore it. After traversing make the value of visited array true for this node.
3. Now push all the unvisited adjacent nodes of the popped element on stack. Push the element even if it is already on the stack.
4. Repeat step 2 and 3 until stack is empty.

Make BFS traverse for following graph

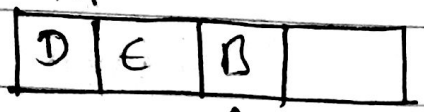
1.



BFS = A, B, C, D, E

① 

push(A)

② 

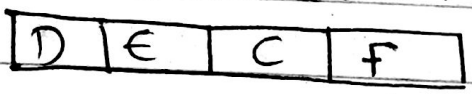
pop(A) and push its adjacent nodes

Traversed = A

③ 

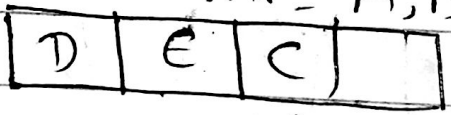
pop(C) and push adjacent nodes of B

Traversed = A, B

④ 

pop(C) and push adjacent nodes of popped element

Traversed = A, B, E

⑤ 

pop(C) and push. (F has NO adj. Nodes)

Traversed = A, B, E, F

⑥ 

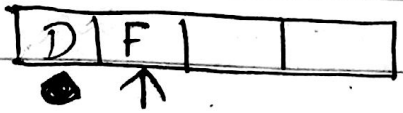
pop C and push A-Node of C. i.e D.

Traversed = A, B, E, F, C

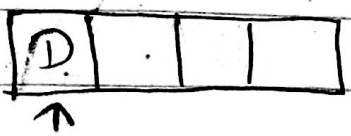
⑦ 

pop F and ignore it

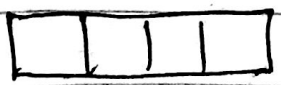
Traversed = A, B, E, F, C

⑧ 

pop E and ignored

⑨ 

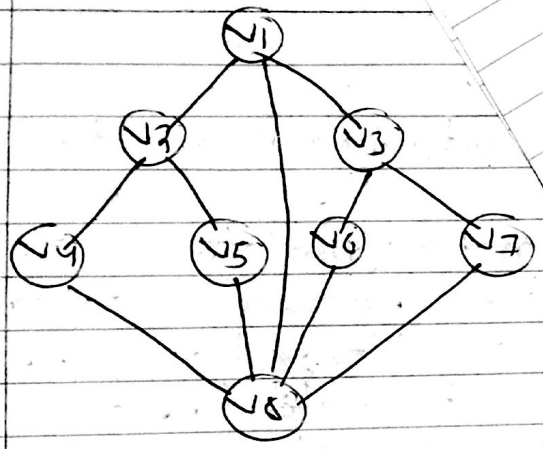
pop F and ignored.

10 
pop D and leave it

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

(5)

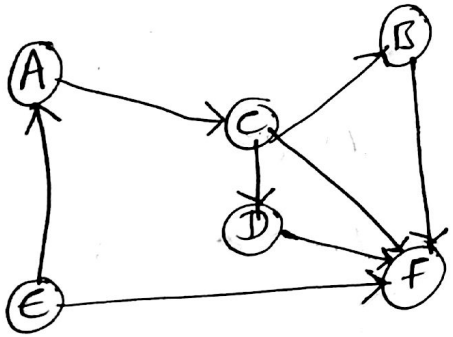
(2)



Let V_1 is starting node.

BFS traversal =

$V_1, V_2, V_8, V_3, V_4, V_5, V_6, V_7$.



Let A be starting node.

BFS = A C B D F

(E can't be visited)

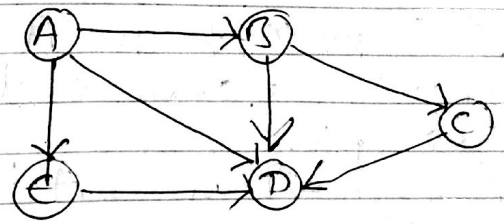
DFS = A C B F D

(E can't be visited)

graph is undirected

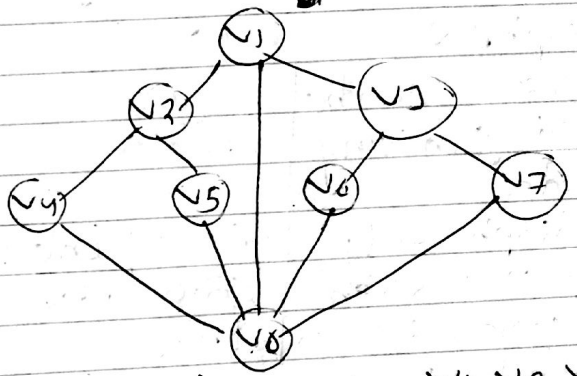
Make DFS traverse for the following graph

(1)



DFS traversal = A, D, E, B, C

(2)



DFS traversal = $V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7$

(There can be several path, its one among them).

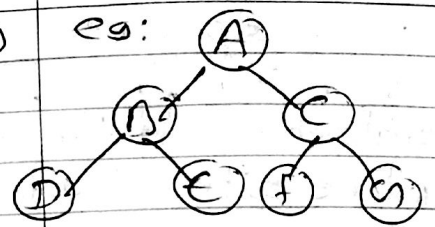
(7) Imp

Graph Vs Tree

	Graph	Tree
(1)	Wile Defination	(1) Wile Defn
(2)	A Graph may or may not consist of cycle.	(2) A Tree doesn't contain cycle.
(3)	A graph without cycle is tree.	(3) A tree with cycle is graph.
(4)	There is no root node or starting node in graph so traversing can be started from any node.	(4) Traversing always start from root node.
(5)	In Graph only those nodes can be traversed which are reachable from starting node.	(5) All the nodes are traversed in tree.
(6)	We can have different sequence of traversal for particular traversal technique.	(6) We have same sequence of traversal for particular traversal technique.

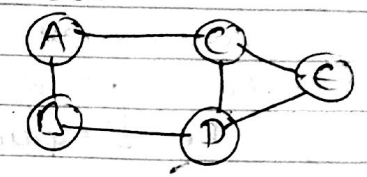
(8)

7) eg:



big: Tree

7) Eg:

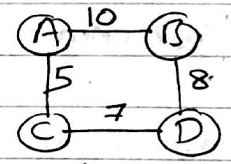


big: Graph

Spanning Tree

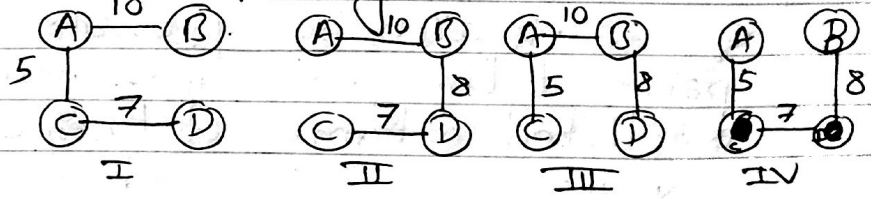
A spanning tree of graph G is a subgraph of G consisting of all the vertex of G and is a tree.

eg:



big: Graph G.

Several spanning tree of G are



In MST, the number of edge will be 1 less than the number of nodes.

Note: If G is not connected graph, then it can't have any spanning tree. In this case it will have spanning forest

We see that multiple spanning trees exists for a single connected graph. Among the several spanning trees, the spanning tree with least cost is called minimum spanning tree. The sum of the weights of the edges of a tree is called its cost.

Algorithms for generating minimum Spanning Tree

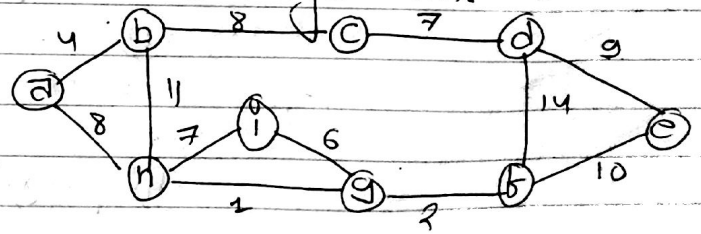
1. Kruskals algorithm
2. Prims Algorithm
3. Round Robin Algorithm

1. Kruskals Algorithm

In kruskals Algorithm, we start with listing of all vertex pairs of the given graph. These pairs are listed in terms of their weight. They are listed in ascending order of their weight i.e The vertex pair with the minimum weight is listed

first, then the vertex pair with the next minimum weight is listed and so on. After that, the vertex pair with the least minimum weight from the list is selected and added to the tree. Then the next vertex pair with minimum weight is selected and added to the tree. During the process of adding vertex pair to the tree, if any vertex pair forms a cycle, we discard it and move to vertex pair with next minimum weight from the list. The process is continued until all the vertex pair from the list are added to the tree. When the list become empty, the resulting tree will be minimum spanning tree.

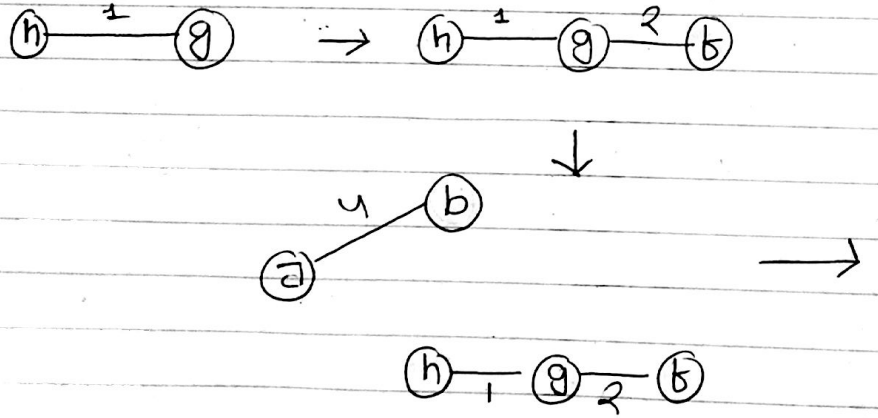
eg: Construct MST of following graph using kruskals Algorithm.



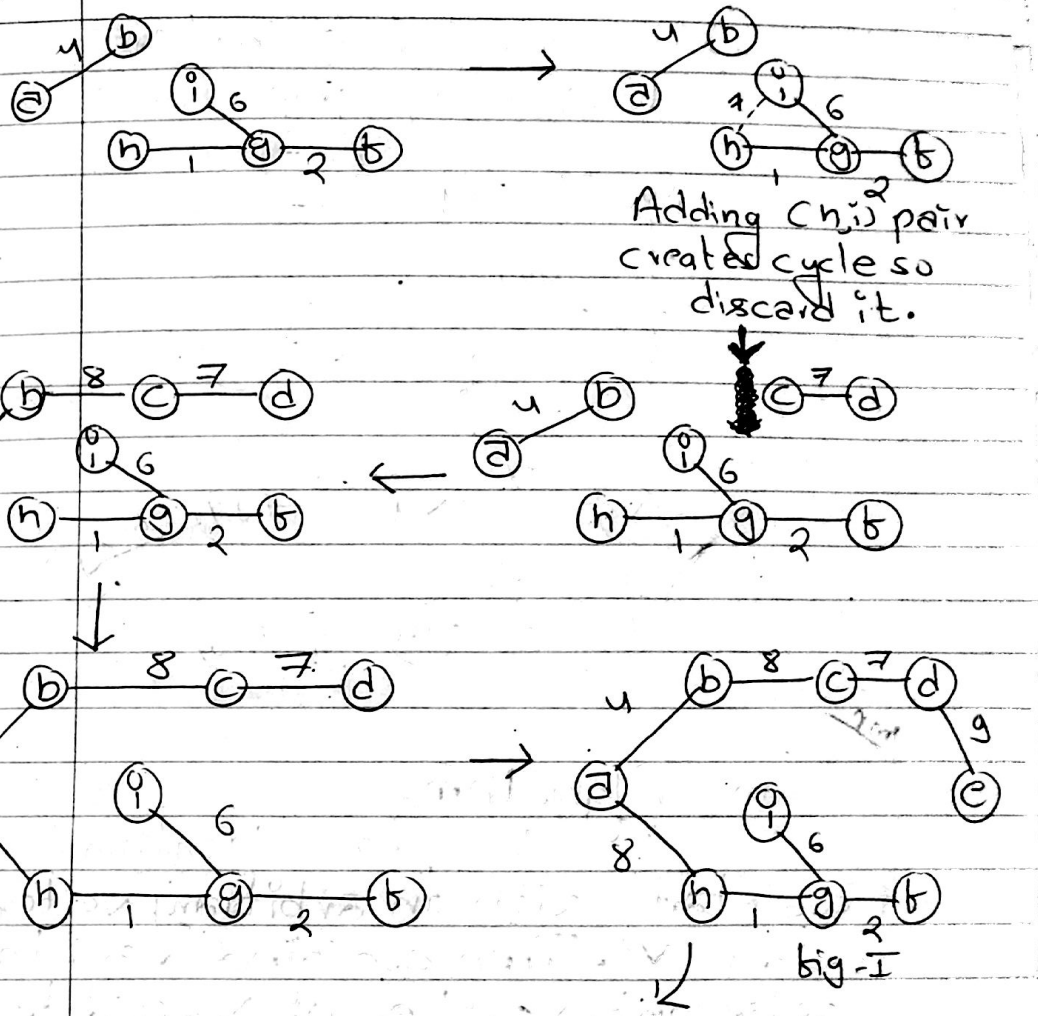
(11)

Soln: The pair of vertices in ascending order of their weights are.

Edge	Weight
(h,g)	1 ✓
(g,b)	2 ✓
(a,b)	4 ✓
(i,g)	6 ✓
(h,i)	7 ✓*
(c,d)	7 ✓
(b,c)	8 ✓
(a,h)	8 ✓
(d,e)	9 ✓
(e,b)	10 *
(b,h)	11 *
(d,b)	14 *



(12)

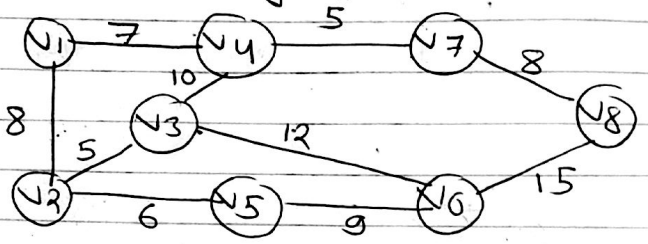


Addition of (e,b), (b,h) and (d,b) all three creates cycle so we discard all these edges. So the final minimum spanning tree is obtained as shown in big I.

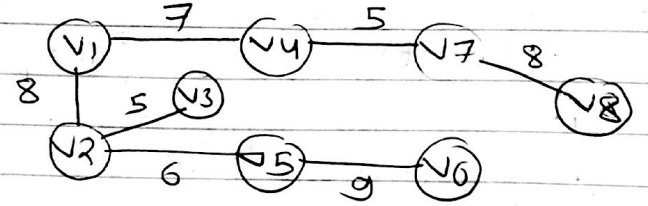
(13)

Use Kruskal's Alg.

~~Construct MST~~ for finding the MST of following graph:



Ans →



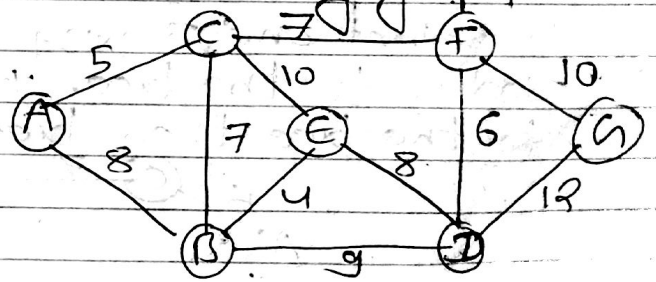
(9) Imp

Prims Algorithms

In prims algorithm we start with an arbitrary vertex say 'v'. Then we find all the adjacent vertices of V, and form a set consisting of these pair of vertices. After that we select the vertex pair with minimum weight ~~for~~ from this set say (V,W). and add it to the tree. Then again we find all the adjacent nodes of W and add these pair of vertices to the previous set and form a new set. From this new set, we select the vertex pair with minimum weight and add it to the tree. During the process of adding vertex pair to the tree, if any pair forms a cycle, we discard it and move to the pair with next minimum weight from the set. The process is continued until all the vertex pairs from the set are added to the tree. When the set becomes empty, the resulting tree will be the minimum spanning tree.

(14)

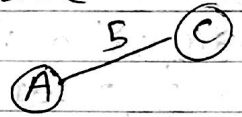
Use prims Algorithm for finding the MST of following graph



Soln: Let our arbitrary vertex be A.
Now we find all the adjacent nodes of A and form a set of these pair of vertices.

$(A, C) = 5$
$(A, B) = 8$

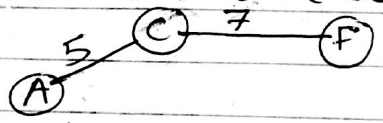
In this set, (A, C) is the pair with minimum weight, so add it to the tree



Now we find the adjacent nodes of C and form a new set as

$(A, B) = 8$
$(C, F) = 7$
$(C, E) = 10$
$(C, B) = 7$

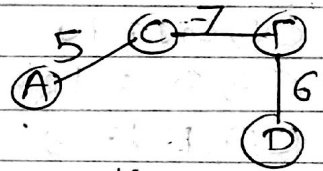
In this set, (C, F) or (C, B) is the pair with minimum weight. So select one among them and add it to the tree. Add (C, F) to the tree then tree will be as



Now, the adjacent nodes of F are C, D, and G. Now the set becomes

$(A, B) = 8$
$(C, E) = 10$
$(C, B) = 7$
$(F, D) = 6$
$(F, G) = 10$

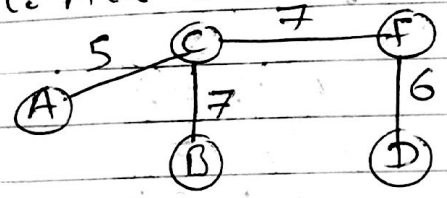
Now $(F, D) = 6$ is the pair with the minimum weight. Add it to the tree then the tree will be as



Now, the adjacent nodes of D are B, E, F and G. Now the set becomes

$(A, B) = 8$
$(C, E) = 10$
$(C, B) = 7$
$(F, G) = 10$
$(D, G) = 13$
$(D, B) = 9$
$(D, E) = 8$

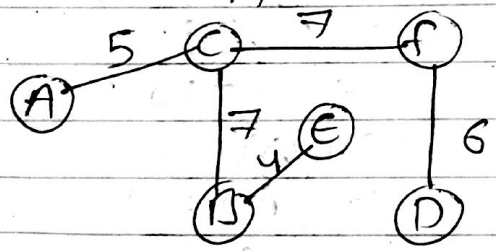
(C, B) = 7 is the pair with min. weight. Add it to the tree.



Now the adjacent nodes of B are C, D, A and E. Now the set become

(A, B) = 8
(C, E) = 10
(F, G) = 10
(D, G) = 12
(D, B) = 9
(D, E) = 8
(B, E) = 4

(B, E) = 4 is the pair with min weight. Add it to the tree.



Now the adjacent nodes of E are B, C, D. The set becomes.

(A, B) = 8
(C, E) = 10
(F, G) = 10
(D, G) = 12
(D, B) = 9
(D, E) = 8

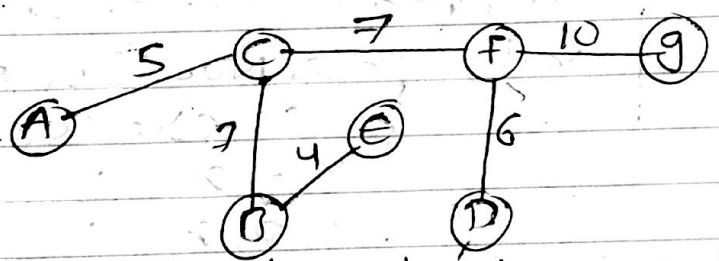
Now, (D, E) = 8 or (A, B) = 8 is the pair with minimum weight. If we add these pair to tree, then it creates cycle. So discard these vertex pairs. Now our set become.

(C, E) = 10
(F, G) = 10
(D, G) = 12
(D, B) = 9

Now (D, B) = 9 is the pair with min. weight. If we add this then it creates cycle so discard it. Now our set become.

(C, E) = 10
(F, G) = 10
(D, G) = 12

Now C(C,E) or C(F,G) are both, pairs with min weight. If we add C(C,E) to tree then it creates cycle so we discard it and add C(F,G)=10 to tree. Our tree become

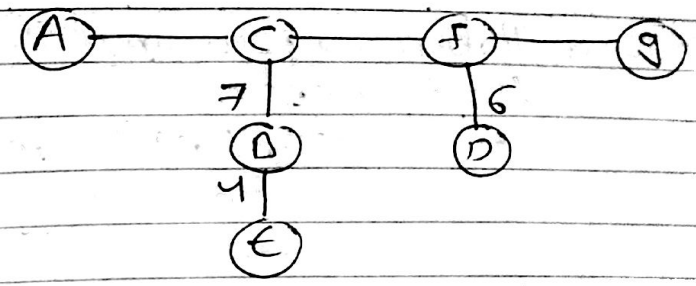


Now the adjacent node of G is F and D. Our set becomes

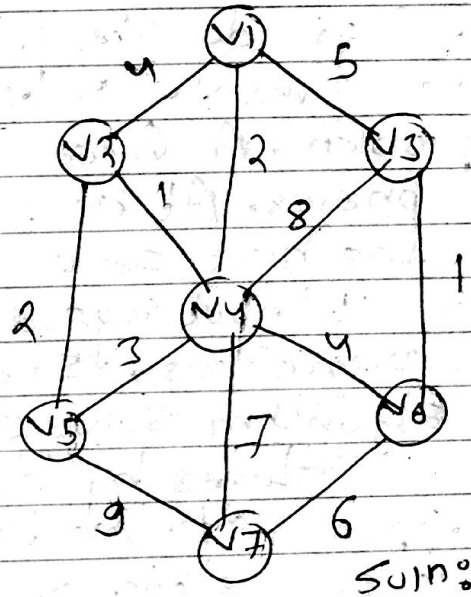
C(D,G) = 13

Finally add C(D,G) to tree, but it creates cycle. hence discard it.

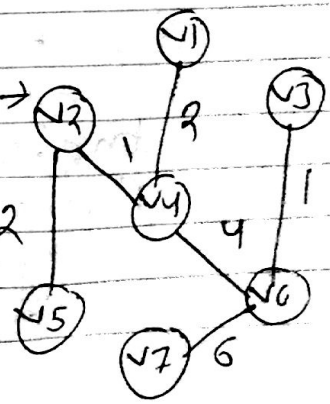
So our final MST is



Find minimum spanning tree using Prims Algorithm.



Soln:→



③ Round Robin Algorithm.
Assignment
See A.M Tanenbaum
Page No: 577

Greedy Algorithm
An optimization problem is one in which we want to find, not just a solution, but the best solution. A greedy algorithm sometime works well for optimization problem. A Greedy algorithm works in phases. At each phase, we take the best we can get right now, without regard for future consequences. So a Greedy algorithm solves a problem in stages by doing what appears to be the best thing at each stage. Some of the examples of Greedy Alg. are Prim's alg., Kruskal's alg. Dijkstra alg etc.

Transitive Closure and Warshall's Algorithm.
The transitive closure of a graph G is defined to be the graph G' such that G' has the same nodes as in G and there is an edge (Vi, Vj) in G' if there is a path from Vi to Vj in G.

The following figure shows a diagraph, whose transitive closure is to be find.

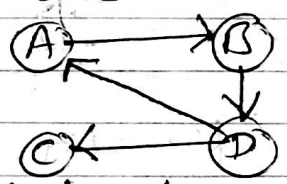


fig: Graph G

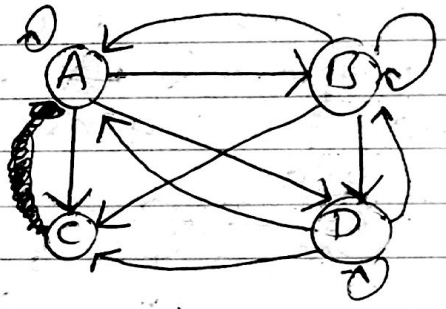
Let A be the adjacency matrix of above diagraph G.

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

Now let T be the adjacency matrix of transitive closure.

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

Let G_1 be the transitive closure of graph G. Then G_1 is



big G_1

→ Warshall's Algorithm for finding transitive closure from diagraph

In Warshall's algorithm, we construct a sequence of Boolean matrices $A = W^{[0]}, W^{[1]}, W^{[2]} \ldots W^{[n]} = T$, where A is the adjacency matrix and T is its transitive closure, This can be done from diagraph as follows

(n is the No. of vertices.)

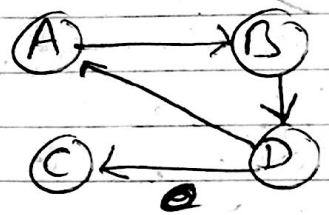
$W^{[0]} = A$

$[W^{[1]}]_{ij} = 1$ if and only if there is a path from v_i to v_j with element of subset of $\{v_1\}$ as intermediate vertices.

$[W^{[2]}]_{ij} = 1$, if and only if there is a path from v_i to v_j with element of subset of $\{v_1, v_2\}$ as intermediate vertices

$T = [W^{[N]}]_{ij} = 1$, if and only if there is a path from v_i to v_j with elements of subsets of $\{v_1, v_2 \ldots v_N\}$ as intermediate vertices.

Ex: Find the Adjacency matrix for transitive closure for the diagraph given below.



Soln: The adjacency matrix A of above graph is

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now

$$Q^{[0]} = A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$Q^{[1]} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

intermediate = A

i-e using A as intermediate we can have path from D to B.

$$Q^{[2]} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

intermediate = {A and B}

$$Q^{[3]} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

intermediate = A, B and C

$$Q^{[4]} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

intermediate = A, B, C and D

Since N = 4

∴ Adjacency matrix for Transitive closure = $Q^{[4]}$ //

Note: How?

Q[1] → 1 means A as intermediate.

Now using A, I can't go from A to A so Q.

" " " " A to C " "

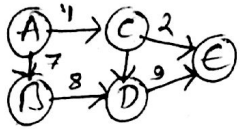
" " " " A to D " "

" " " " B to A

and similar for other.

But I can go from D to B using A hence D to B = 1.

eg



shortest path from A to E is
A, C, E #

Shortest path Algorithm

A path from source vertex s to t is shortest path if there is no path from s to t with lower cost. The sum of the weight of edges of graph is called cost. The shortest path are not necessarily unique.

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

Dijkstra's Shortest path Algorithm.

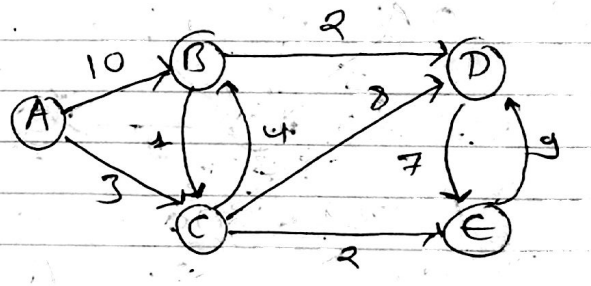
Dijkstra Algorithm is a greedy algorithm that solves the shortest path problem for a directed graph G = (V, E).

Algorithm

1) Mark all vertices as unknown.
2) For each vertex v keep a distance d_v from source vertex to v.
3) Initially set to ∞ except for s which is set to $d_s = 0$.
4) Repeat step 5 to 7 until all vertices are known.
5) Select a vertex v, which has smallest d_v among all the unknown vertices.
6) mark v as known.
7) For each vertex w adjacent to v

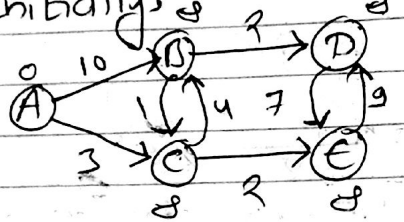
If w is unknown and d_v + cost(v,w) < d_w
update d_w to d_v + cost(v,w) ie
Relax all the adjacent vertices of v.

Find the shortest path for the graph below considering A as source vertex.



dv

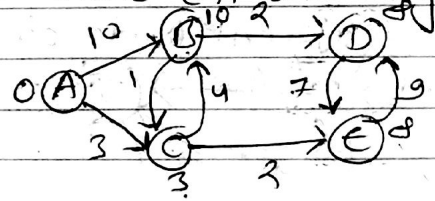
Soln: Initially, set all Node except start node to ∞, and for start Node dv = 0.



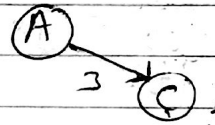
Since dv of A is minimum, so select it.

adjacent vertices of A

Now relax all the ~~edges leaving A~~.

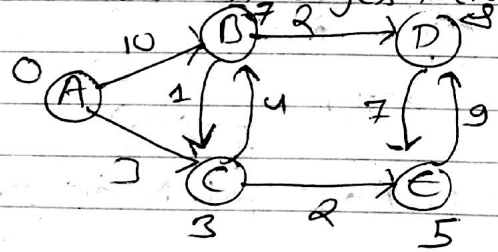


Since dv of C is minimum, select it.

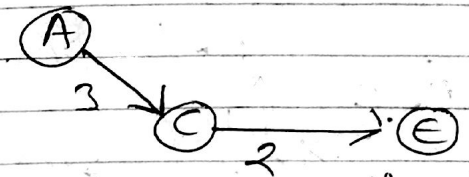


the adjacent vertices of C.

Now relax all ~~edges leaving C~~.

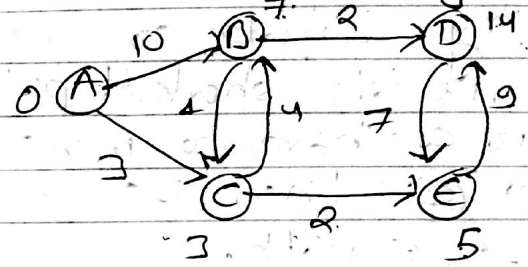


Since dv of E is minimum, so select it

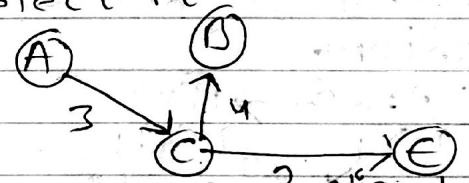


the adjacent vertices of E.

Now relax all ~~edge leaving E~~

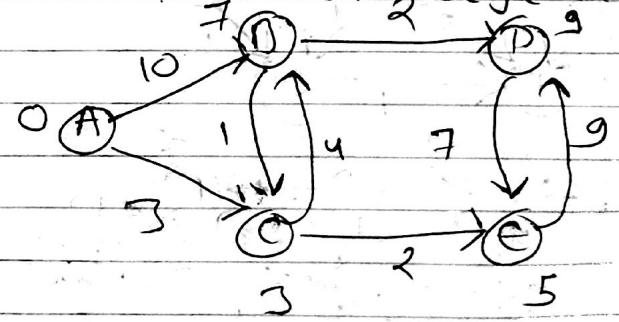


Since dv of B is minimum so select it

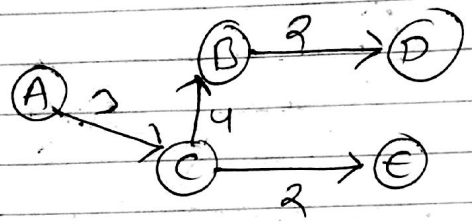


adjacent vertex of B.

Now relax all ~~edge leaving~~

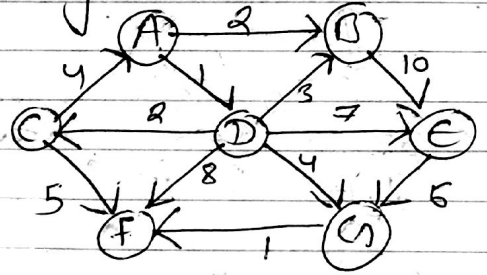


Since D is only one unmarked Node so select it.

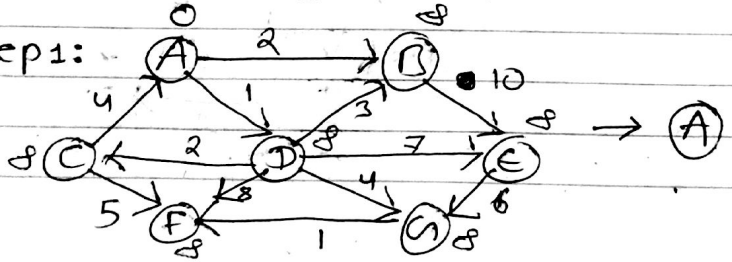


Note: Any subpath of shortest path is also shortest path. So if we have to move from (A) to (E) then we can take A, E, E path.

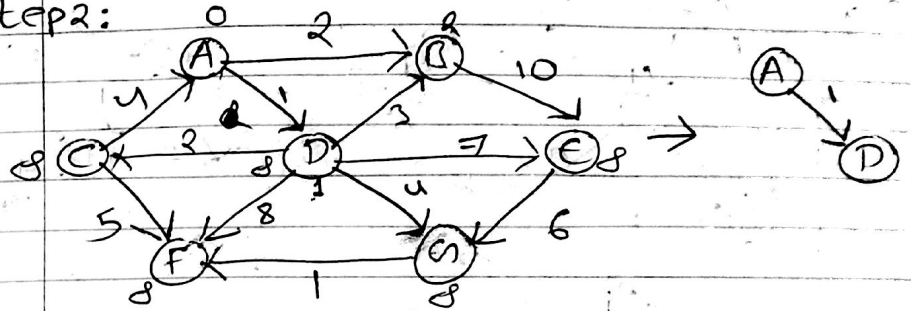
Find shortest path using Dijkstra Algorithm.



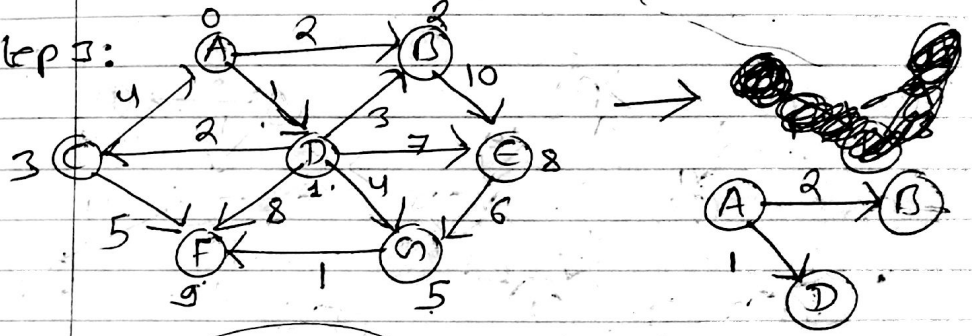
Soln: Step1:



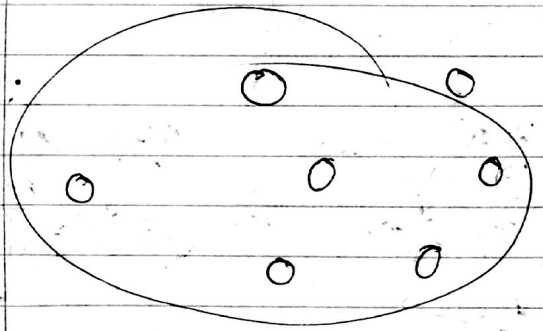
Step2:



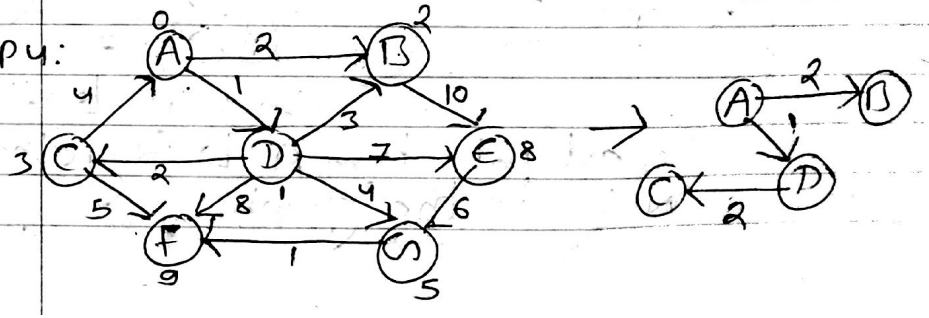
Step3:



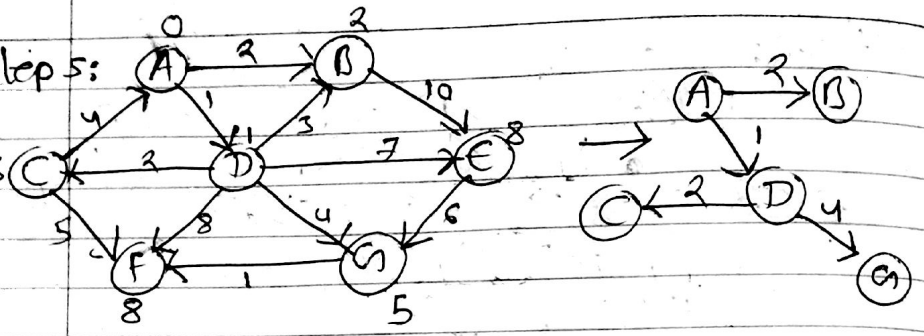
Step4:



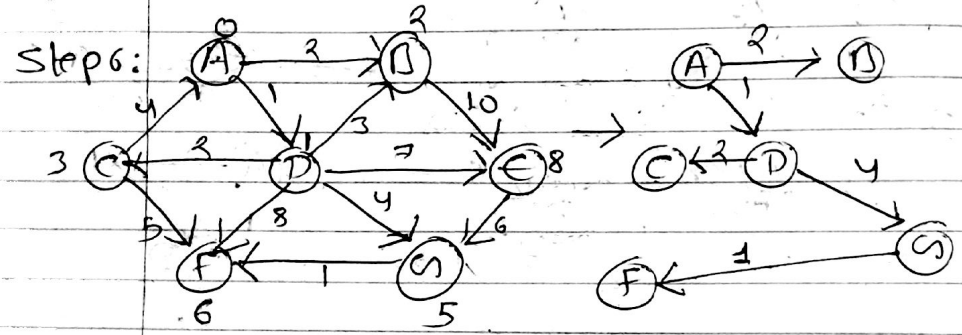
Step4:



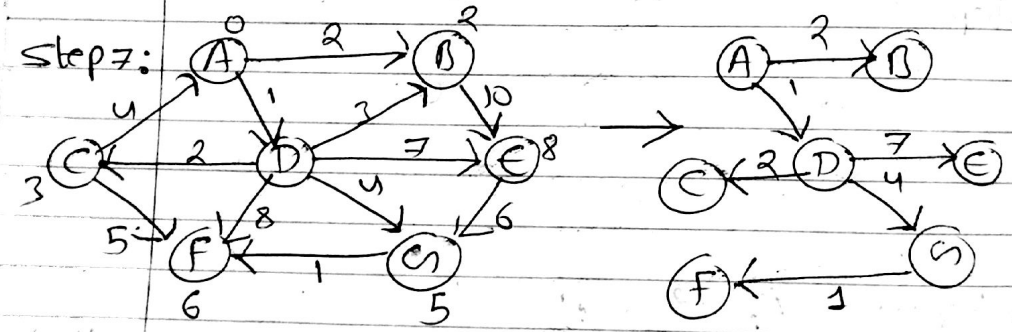
Step 5:



Step 6:



Step 7:



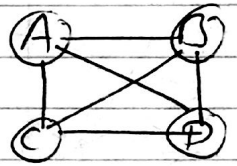
which is final shortest path from A to any other Node in the graph.

Missed Topics

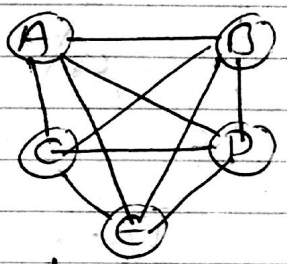
① Complete graph

A graph in which all the vertices are connected to each other is called complete graph. Let n be the total number of vertices of graph G, then complete graph consisting of n vertices is denoted by K_n.

eg:

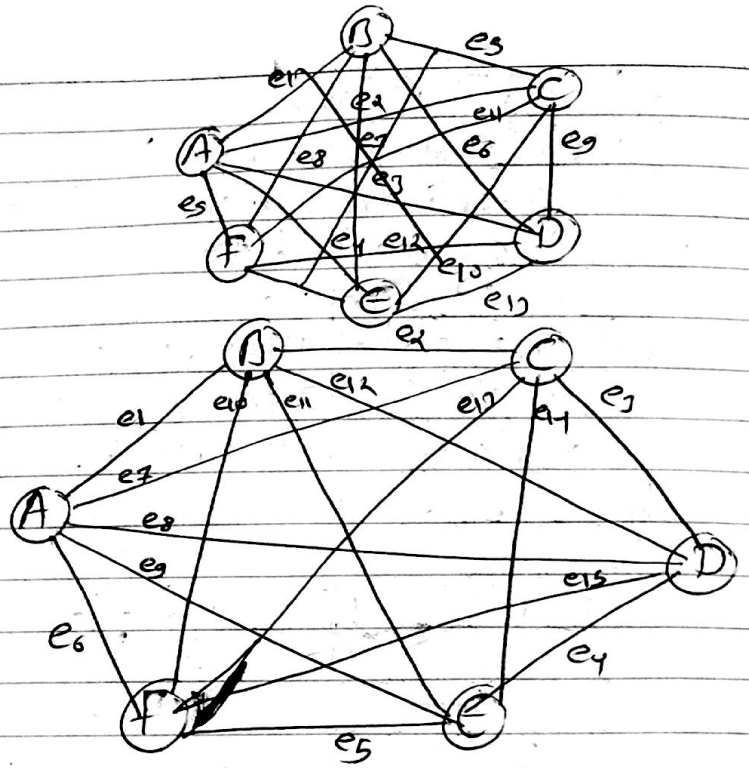


big Graph K_4



big Graph K_5

eg: Draw a complete graph K_6 and write its incidence matrix.



Incidence matrix

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	e13	e14	e15
A	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0
C	0	1	1	0	0	0	1	0	0	0	0	0	1	1	0
D	0	0	1	1	0	0	0	1	0	0	0	1	0	0	1
E	0	0	0	1	1	0	0	0	1	0	1	0	0	1	0
F	0	0	0	0	1	1	0	0	0	1	0	0	1	0	1

(Q) Application of graph

— Representing link structure of website

— In the field of networking

— Routing one of the real world application of graph is to create a network in a city with minimum cost of cabling.

(How? Explain Yourself)

END

Assn:

(1) Write diffn betn Tree and Graph.

(2) Explain strategy for ~~p~~ Null pointer replacement while creating BST to threaded binary tree?

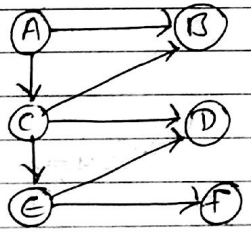
Topological Sort

- A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of $x, y \in V$ such that if $(x, y) \in E$ then x must appear before V in ordering.
- If G is cyclic then no such ordering exists.
- Topological sort can be used to show which vertex to process first and which next in a digraph.

Algorithm

1. Call DFS to compute finishing times $f(v)$ for each vertex $V \in G$.
2. As each vertex is finished, insert it into the front of linked list.
3. Remove from digraph vertex V and all edges incident with V.
4. Finally Display the element of of link list.

Execute topological sort in the following directed acyclic graph.



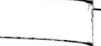
Soln:

Let starting node = A

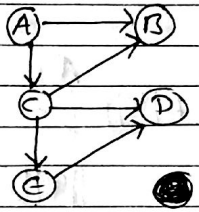
step 1: - Run DFS

A → C → E → F

- Add Node F to front of linked list.

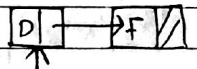


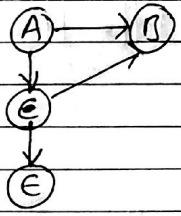
- Remove F



Step 2 - Run DFS

- A → C → E → D
- Add D to front of LL
- Remove D

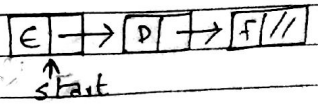




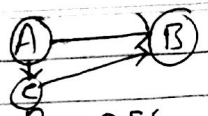
Step 3 - Run DFS

A → C → E

- Add E to linked list



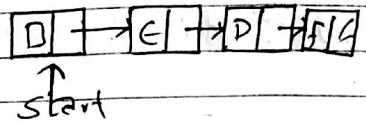
- Remove E



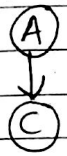
Step 4:- Run DFS

A → C → B

- Add B to linked list



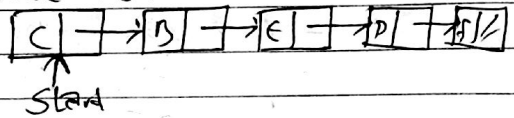
- Remove D



Step 5:- Run DFS

A → C

- Add C to linked list

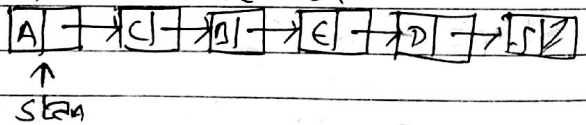


- Remove C

Step 6:- Run DFS

A

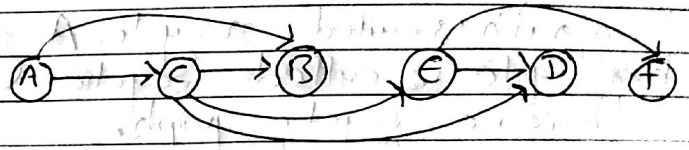
- Add A to linked list



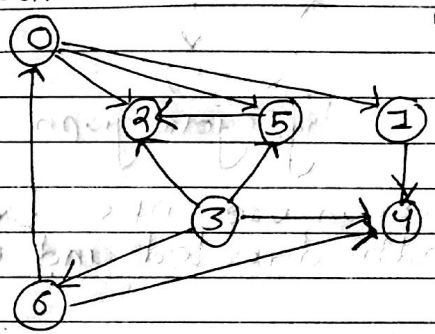
- Remove A

Now we have no more nodes in the graph, so the topological sorting of the graph will be

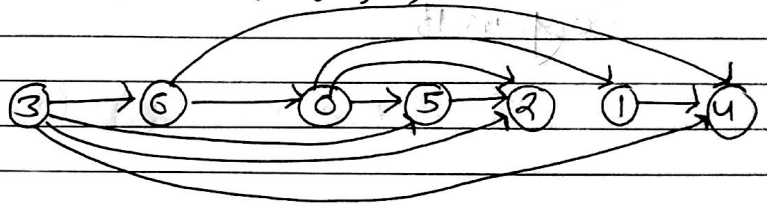
A, C, B, E, D, F



Execute topological sort in the following digraph



Ans: 3, 6, 0, 5, 2, 1, 4



Note: When vertex 4, 1, 2, 5 and 0 are removed then check vertex 3 as you have to visit all vertex in graph.