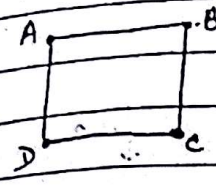
Graph-



$G(V, E)$

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

$E = \{[A,B], [B,C], [C,D], [D,A]\}$

$e = [A, B]$ — square brackets are used for undirected graphs

A & B are neighbours / adjacent nodes

A & B are end points of this edge.

1) Degree – no. of edges incident with a node

eg: deg(A) = 2 deg(D) = 2

2) Isolated node – any node of degree 0

eg. .P

3) Path – Sequence of edges blw any two given nodes

A path of length 'n' has (n+1) nodes

4) Circuit – A closed path ; initial & final node is the same

Also called a cycle

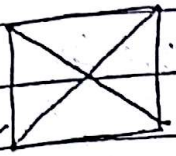
Should have atleast 3 edges / length of path ≥ 3

5) Simple Path – A path where each node occurs only once

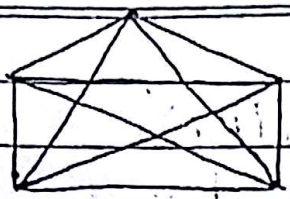
6) Connected Graph – A graph having a path blw any two given nodes

7) Complete Graph – Each node is connected to every other node

A complete graph is always a connected graph but vice-versa is not true.



complete graph of order 4

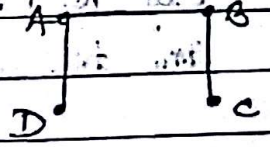


order 5

To draw a complete graph of order n, draw a regular polygon of (n+1) edges & connect each pair of nodes

no. of edges = $\frac{n(n-1)}{2}$

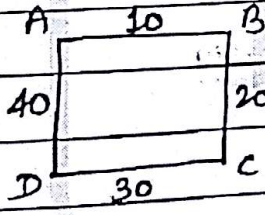
8) Tree graph – A graph devoid of any cycles eg.



9) Label graph – Names are assigned to the edges as well i.e. edges are labelled

10) Weighted graph – all the edges are assigned a non-negative integer value

eg.

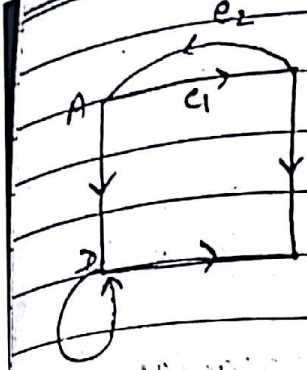


Total weight = 10+20+30+40

11) Multigraph – i) multiple edges b/w pair of nodes
ii) loops

12) Finite graph – Finite no. of edges & vertices

13) Directed graph – Every edge has an assigned direction



$e_1 = \{(A,B)\}$ $\quad e_2 = \{(B,A)\}$

open brackets

Indegree → no. of edges ending at a particular node

Outdegree → " starting from "

Sink – Nodes with outdegree = 0

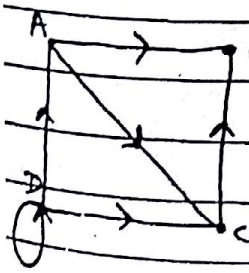
Source – " Indegree = 0

14) Strongly Connected graph – If there exists an directed edge b/w two nodes say N_1 & N_2, then there must exist an directed edge b/w N_2 & N_1 as well and this should hold True for every pair of nodes connected

15) Unilateral graph – Condition of strongly connected does not hold true for every pair of connected nodes.

Graph Representation –

- Sequential
  - Adjacency matrix
  - Path matrix
- Linked



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

Sink → B

Adjacency matrix

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

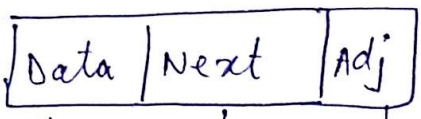
Path matrix

$$A_{ij} = \begin{cases} 0, & \text{if } v_i \text{ is not adjacent to } v_j \\ 1, & \text{if } v_i \text{ is adjacent to } v_j \end{cases}$$

$$P_{ij} = \begin{cases} 0, & \text{no path b/w } v_i \text{ \& } v_j \\ 1, & \text{path between } v_i \text{ \& } v_j \end{cases}$$

Linked Representation

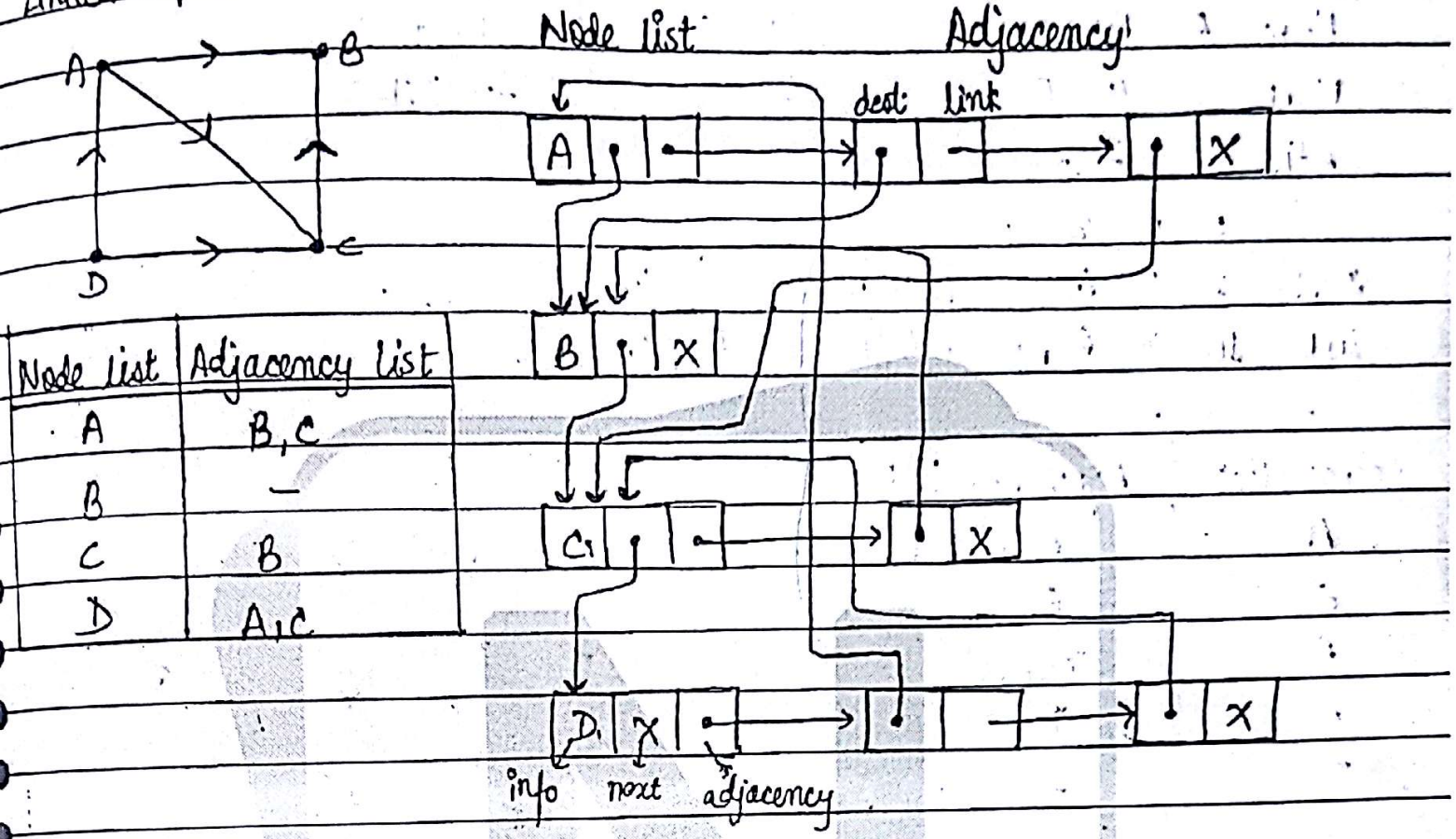
- node list
- edge list



- Data → key value
- Next → ptr to next node
- Adj → ptr to first element in the adjacency list

- Dest → pts to the location in the node list of the dest^n node.
- Link → links together the nodes in the same adjacency

Linked Representation –



Node list	Adjacency list
A	B, C
B	–
C	B
D	A, C

Memory storage –

Start		Info	Next	Adj.
3	1		2	
	2		4	
	3	A	5	2
	4		6	
	5	B	7	0
	6		9	
	7	C	8	7
	8	D	0	8
	9		10	
	10		0	
	11			

Avail		Dest.	link.
1	1		3
	2	5(B)	4
	3		5
	4	7(C)	0
	5		6
	6		9
	7	5( )	0
	8	3(A)	10
	9		11
	10	7(C)	0
	11		0

V. Imp →

Breadth First Search (BFS) – (implemented using queue)

1) Initialize all nodes to the ready state (STATUS = 1)
2) Put starting node N in Queue and change its status to waiting state (STATUS = 2)
3) Repeat steps 4 and 5 ~~while~~ until Queue is empty.
4) Remove front node N. Process N and change its state to processed state (STATUS = 3)
5) Add to rear of queue all neighbours of N that are in ready state and change their state to waiting state. (End of step 3)
6) EXIT

Node List	Adjacent List
A	B, C, F
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

1) Let starting point be A
2) Queue : A

		Front ↓	Rear ↓
		Q: B, C, F	
2)	Process A	Q: C, F, G	
3)	Process B	Q: F, G	
4)	Process C	Q: G, D	
5)	" F	Q: D, E	
6)	" G	Q: E	
7)	" D	Q: J	
8)	" E	Q: K	
9)	" J	Q: Empty	
10)	K		

→ ABCFGDEJK

→ Depth First Search (DFS)– (implemented using stack).

1) Initialize all the nodes to ready state (STATUS=1)
2) Push the starting node N onto the stack and change its status to the waiting state (STATUS=2)
3) Repeat steps 4 & 5 until stack is empty
4) Pop the top node N of the stack. Process N & change its status to processed state (STATUS=3)
5) Push onto the stack all the neighbours of N that are still in the ready state & change their status to waiting state (End of step 3)
6) EXIT

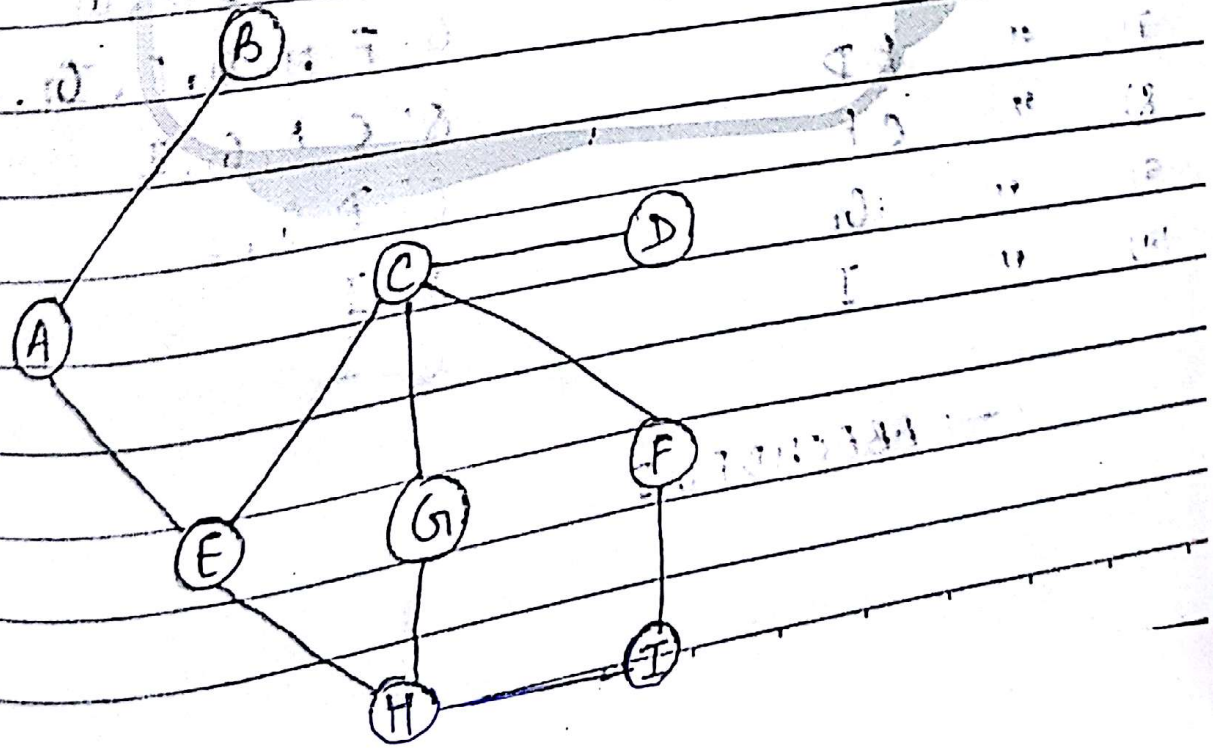
(a) Traverse all the reachable nodes with J as the starting point using DFS algo.

1) Let starting point be J
Stack: J

			Top
2)	Pop & Process	J	S: D, K
3)	"	K	S: D, E, G
4)	"	G	S: D, E, C
5)	"	C	S: D, E, F
6)	"	F	S: D, E
7)	"	E	S: D
8)	"	D	S: Empty

→ JKGCFED

(b) (es)



Node	List. Adjacency List
Starting → A	B, E
B	A
C	D, E, F, G
D	C
E	A, C, H
F	C, I
G	C, H
H	E, G, I
I	F, H

1)	Starting point = A	Q: A
2)	Process A	Q: B, E
3)	" B	Q: E
4)	" E	Q: C, H
5)	" C	Q: H, D, F, G
6)	" H	Q: D, F, G, I
7)	" D	Q: F, G, I
8)	" F	Q: G, I
9)	" G	Q: I
10)	" I	Q: —

→ ABECHDFGI

1)	St: pt → A		S: A
2)	Pop & Process	A	S: B, E
3)	"	E	S: B, C, H
4)	"	H	S: B, C, G, I
5)	"	I	S: B, C, G, F
6)	"	F	S: B, C, G
7)	"	G	S: B, C
8)	"	C	S: B, D
9)	"	D	S: B
10)	"	B	S: —

→ AEHIFGCDB

Shortest Path Algorithms

- find the shortest path in a weighted graph.

Single source shortest path

(*) determines cost of shortest path from u to v

⇒ Dijkstra

all pair shortest path

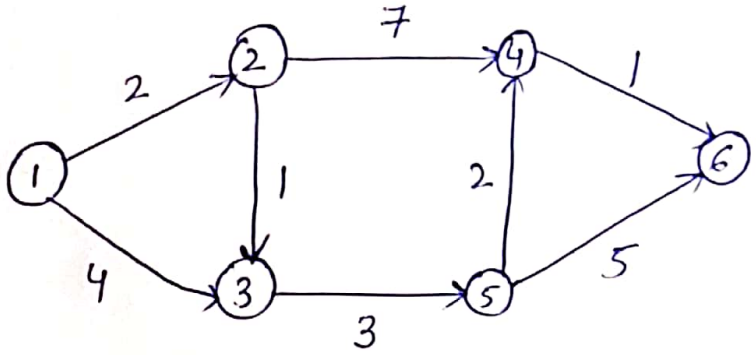
(*) determines the cost of shortest path from each vertex to every other vertex.

⇒ Floyd Warshall

① DIJKSTRA ALGORITHM

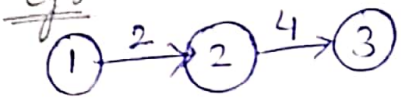
- find the shortest path from one vertex (source) to all other vertices.
- we will use greedy method to find the soln.
  * Greedy method : > take one step at a time
    > take the step which looks best.

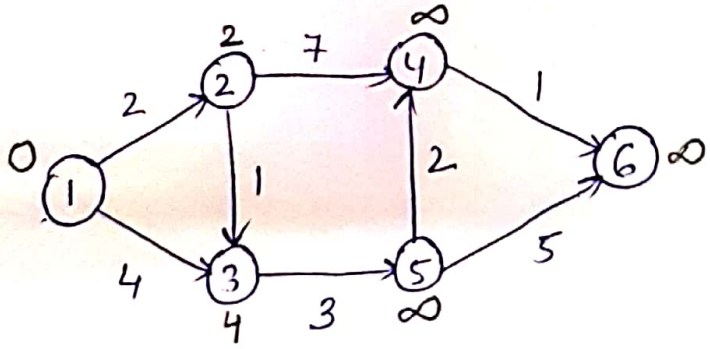
Ques.



Ans. Let us take source = 1.

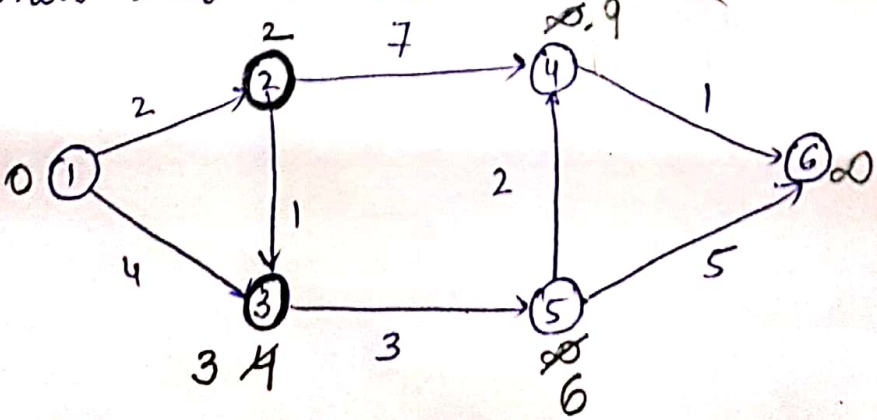
eg.





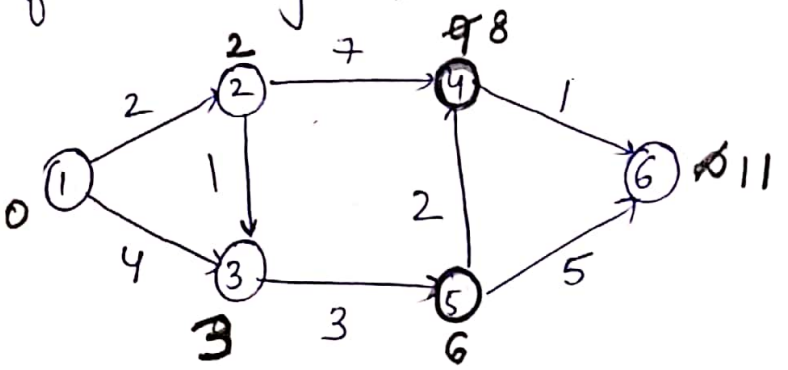
Step 1
(initial step)

- Now select the shortest path (2/4/∞) = 2
- Now perform relaxation (updating the values), check the nodes connected with 2. (3 & 4)
- check their distances now, & update if the new dist < old dist.



∴ Now the smallest is dist 3.
Then select 6.

After selecting 6 distance, which is node 5 :



Now node 4 value < node 6 value
select node 4 (value 8). check
$8 + 1 = 9 < 11$.
∴ Value of node = 9.

Table :

V	dist
2	2
3	3
4	8
5	6
6	9

} Shortest distance from vertex 1.

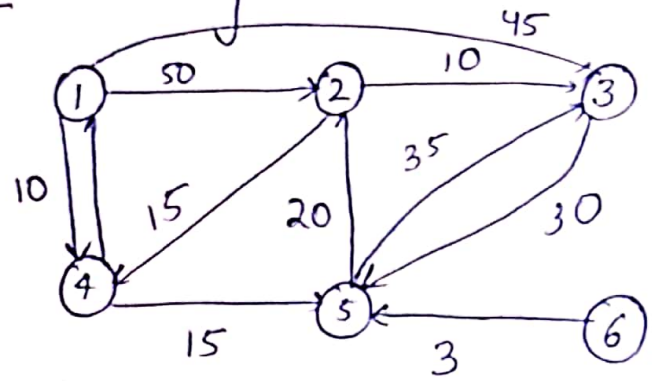
Complexity :

$n = |V| \times |V|$
(total vertices) (vertices to be relaxed)

$= n \times n = n^2$ (in case of complete graph)

∴ Worst case ⇒ $O(n^2)$ / $O(V^2)$

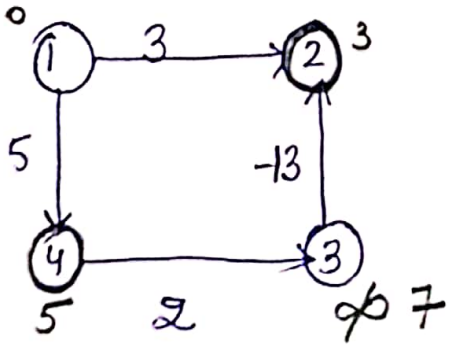
Ques 2. Starting vertex 1.



Selected vertex	2	3	4	5	6
4	50	45	(10)	∞	∞
5	50	45	(10)	(25)	∞
2	(45)	45	(10)	(25)	∞
3	(45)	(45)	(10)	(25)	∞
6	(45)	(45)	(10)	(25)	(∞)

∵ There is no incoming edge on 6.

Drawback



- no need to relax the node which is already selected.
- But, here if check node 3, now node 2 can be updated by $(7-13) = -6 < 3$.
- But dijkstra will not check that path.

(*) Thus, Dijkstra Algo does not work for negative weighted edges.

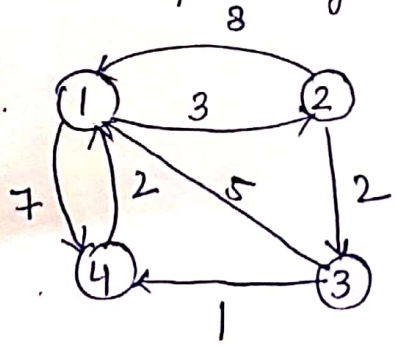
② Floyd Warshall's Algo

- used to find shortest dist. b/w two nodes
- graph should be weighted directed graph
- edges can have +ve / -ve weights.
- There should be no -ve cycle (i.e., a cycle whose edges sum to a -ve value).

Weight matrix

- Elements of this matrix are the weight of the edges b/w any two given nodes.
- takes into account only immediate neighbours.
- For non-adjacent node, weight is taken '∞'.
- If no-self-loop is present, value of each (a x a) cell / every diagonal is '0'.

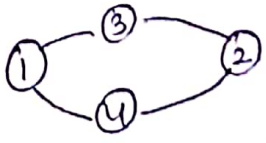
Ques.



$$A_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

- now we'll check the paths via other vertices.

but,



they can be shorter,

$$A_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array}$$

(via vertex 1)

⇒ when vertex 1 is chosen, all the paths belonging to 1 will not change.

$$\therefore A_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix} \end{array}$$

$A_1[2,3]$ = compare

$A^0[2,3]$ $\quad A_0[2,1] + A_0[1,3]$

$= 2 \quad 8 + \infty$

$= (2) < \infty$

So, $A_1[2,4]$: $\quad A_0[2,4] \quad A_0[2,1] + A_0[1,4]$

$\infty \quad 8 + 7$

$\infty > (15)$

Now,

$$A_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & 7 & 0 \end{bmatrix} \end{array}$$

$A_2[2,3]$: $\quad A_1[1,3] \quad A_1[1,2] + A_1[2,3]$

$\infty \quad 3 + 2$

$\infty > (5)$

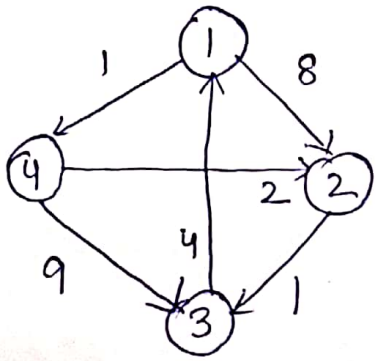
$$A_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

In general,

$$A_K[i,j] = \min \left\{ A_{K-1}[i,j] \;,\; A_{K-1}[i,K] + A_{K-1}[K,j] \right\}$$

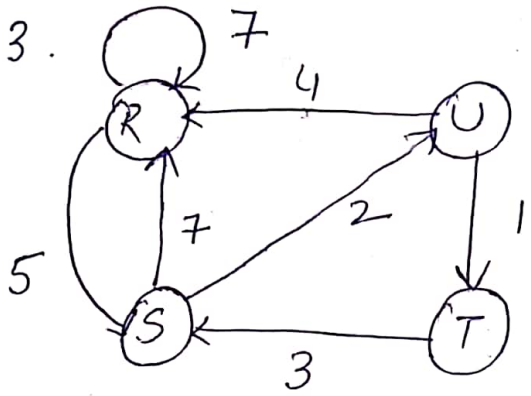
Ques 2.



Ans.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Ques 3.

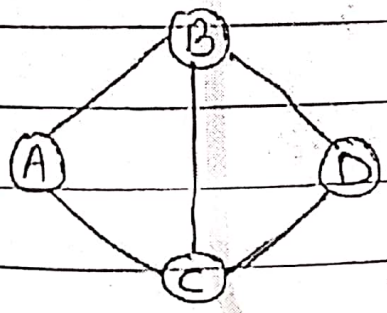


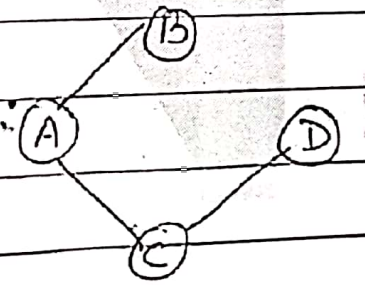
	R	S	T	U
R	7	5	8	7
S	6	6	3	2
T	9	3	6	5
U	4	4	1	6

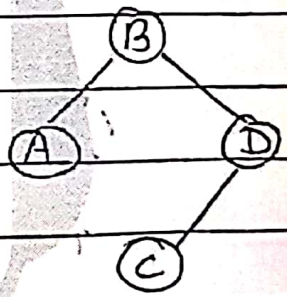
→ Spanning Tree :-

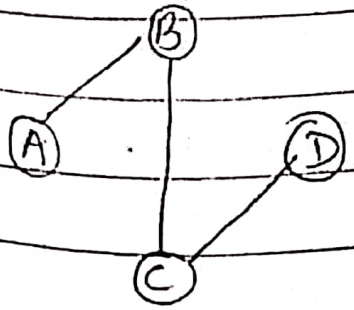
- Trees devoid of any cycle
- Includes all the nodes but not all the edges

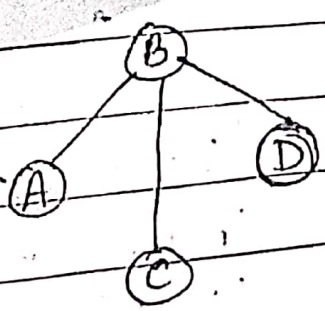
If there are n nodes, the no. of edges in a spanning tree = (n-1).

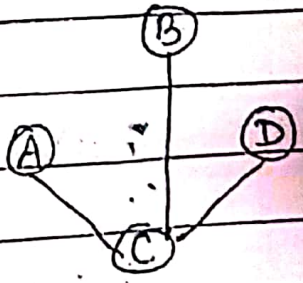










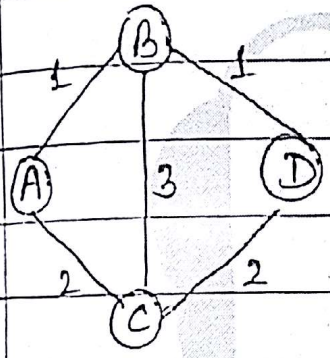


Minimum Spanning Tree-

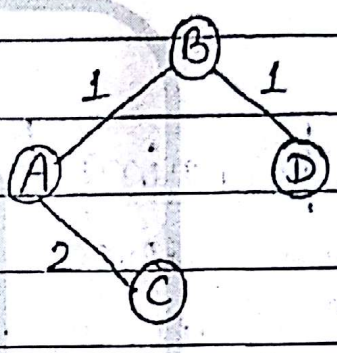
Kruskal's Algo-

Arrange the edges in ascending orders & then draw a spanning tree using these edges in sequence

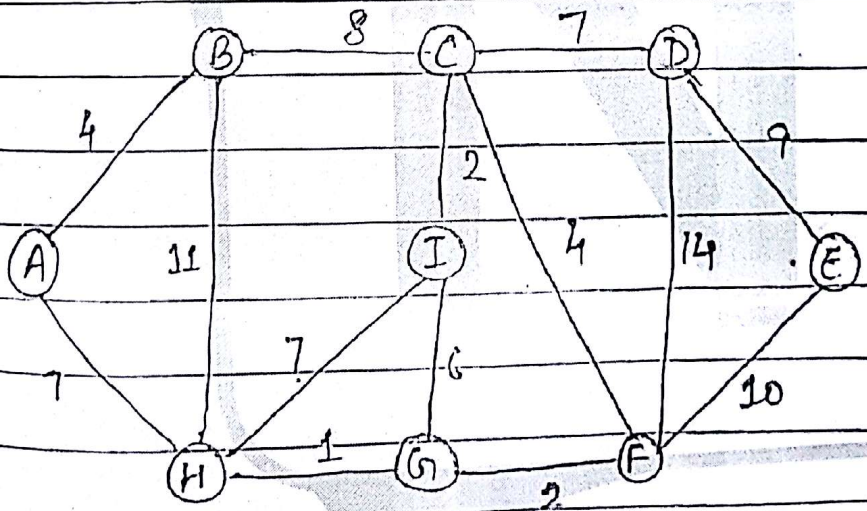
eg.



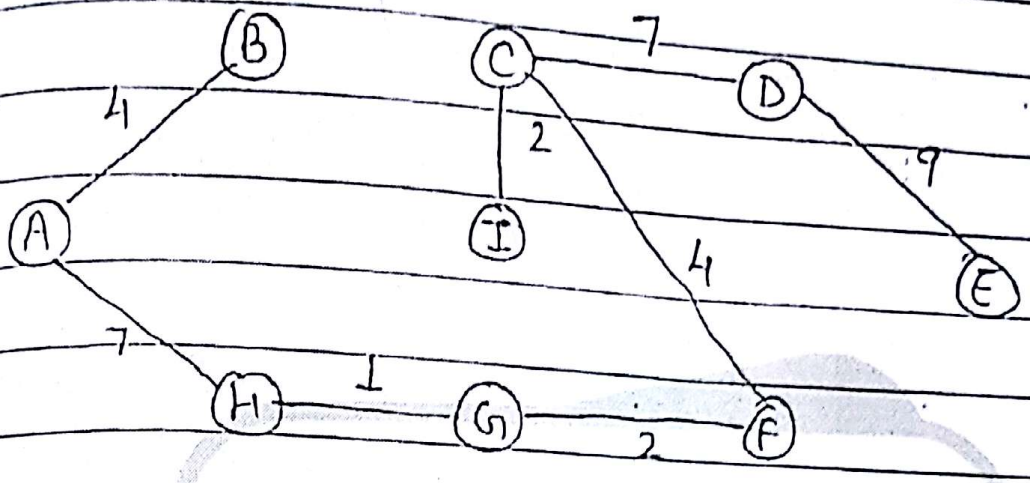
(A,B) - 1 ✓
(B,D) - 1 ✓
(A,C) - 2 ✓
(C,D) - 2 X
(B,C) - 3 X



eg)

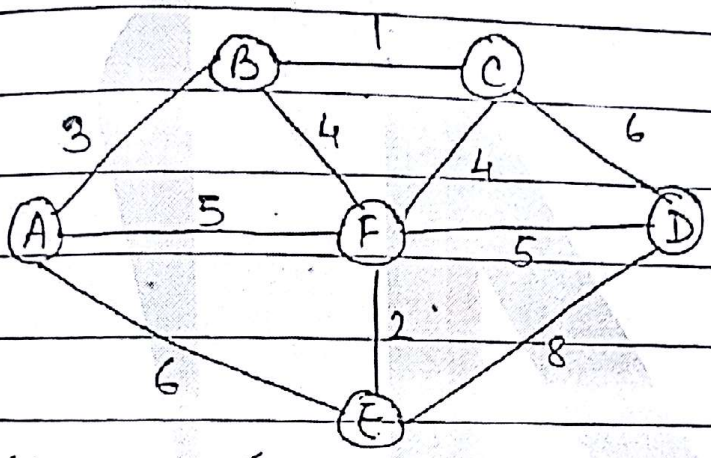


(G,H) - 1 ✓
(C,I) - 2 ✓
(G,F) - 2 ✓
(A,B) - 4 ✓
(C,F) - 4 ✓
(G,I) - 6 X
(A,H) - 7 ✓
(H,I) - 7 X
(C,D) - 7 ✓
(B,C) - 8 X
(D,E) - 9 ✓
(F,E) - 10 X
(B,H) - 11 X
(D,F) - 14 X



36

Ques)



(B, C) - 1. ✓

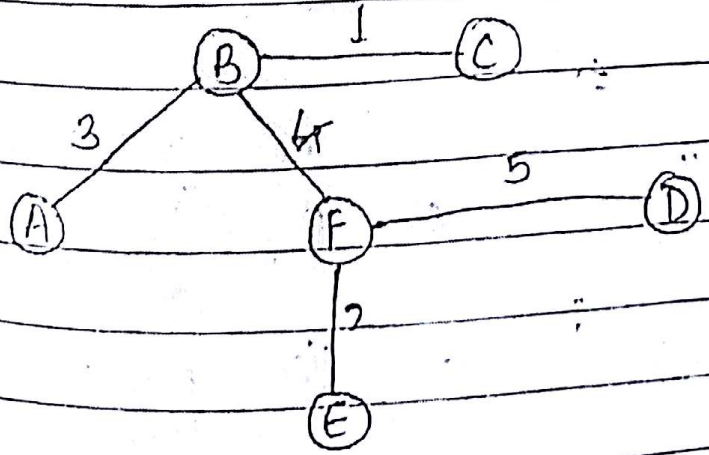
~~(B, A)~~ - (F, E) - 2 ✓ (F, D) - 5 ✓

~~(E, F)~~ - (B, A) - 3 ✓ (C, D) - 6 X

(B, F) - 4 ✓ (A, E) - 6 X

(F, C) - 4 X (E, D) - 8 X

(A, F) - 5 X



1+2+3+4+5 = 15

Prim's Algo

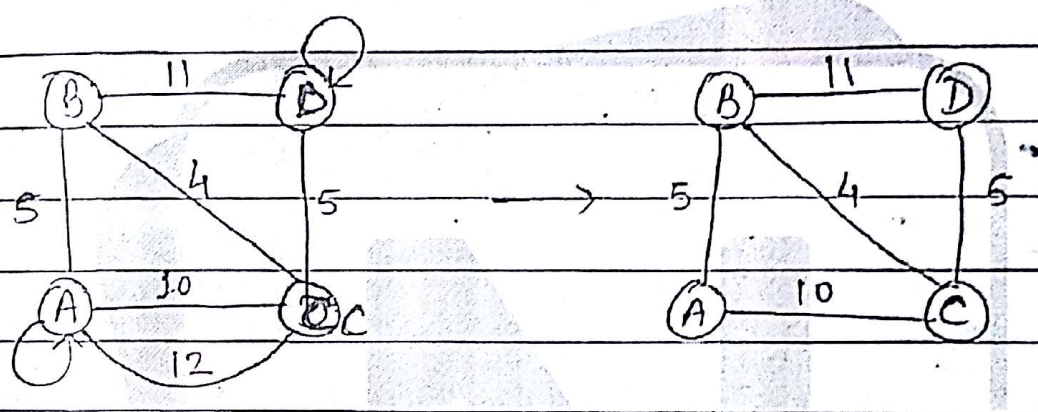
① Select the min cost edge first.

② " " next min edge, but it should be connected to the existing edge.

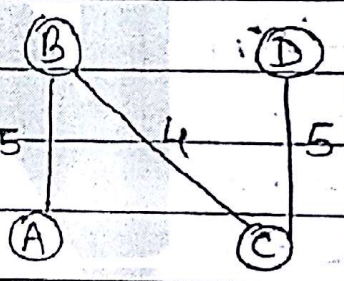
Prim's Algo –

i) Ignore self-loops, if present.
ii) In case of multiple edges, ignore the higher-weighted edge.

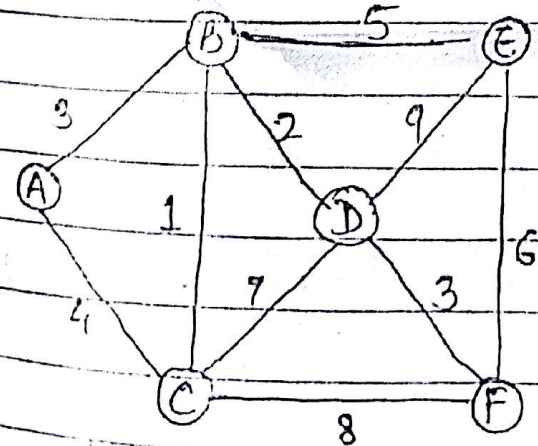
eg.



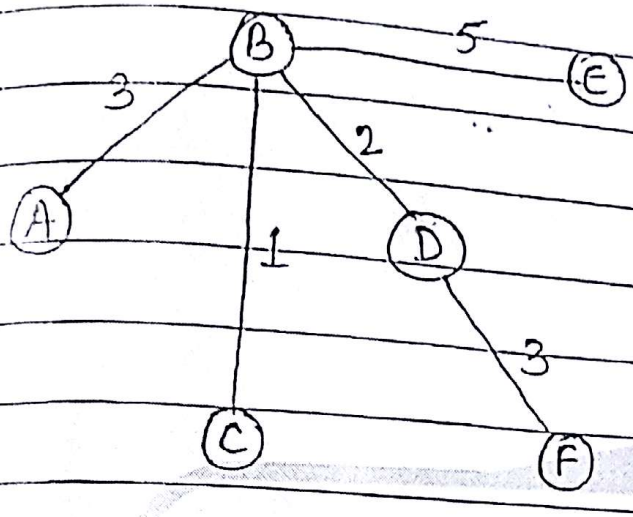
	A	B	C	D
A	0	(5)	10	∞
B	5*	0	(4)	11
C	10	4*	0	(5)
D	∞	11	5*	0



Ques)

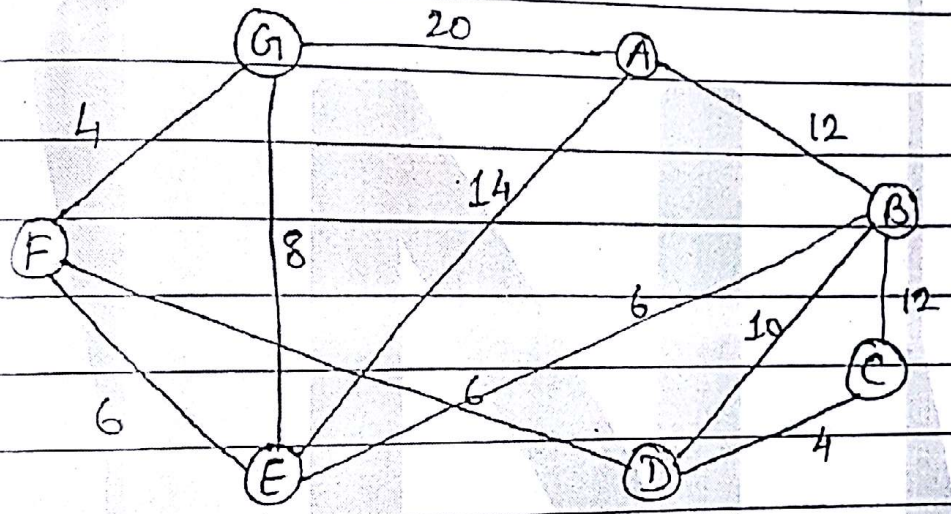


	A	B	C	D	E	F
A	0	(3)	4	∞	∞	∞
B	3*	0	(1)	2	5	∞
C	(4)	1*	0	7	∞	8
D	∞	(2)	7	0	9	3
E	∞	(5)	∞	9	0	6
F	∞	∞	8	3	6	0

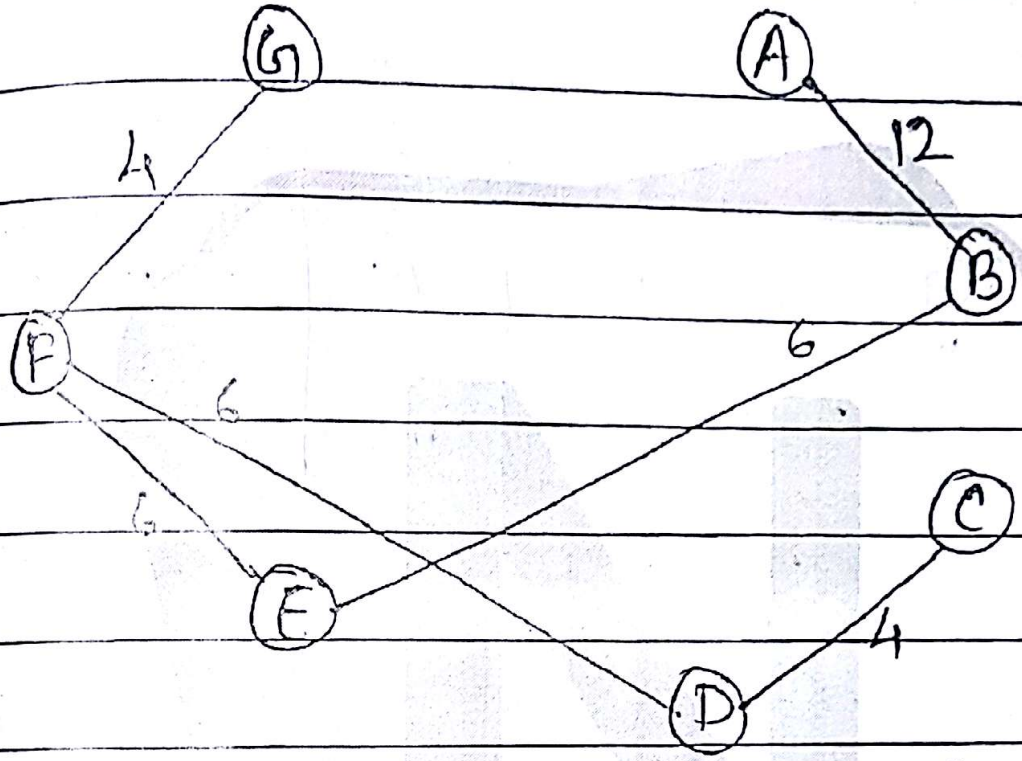


3+1+2+3+5 = 14

Ques)



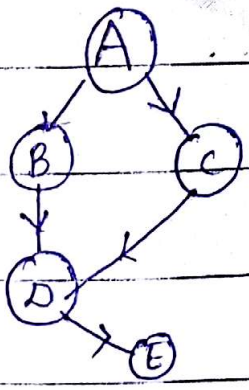
	A	B	C	D	E	F	G
A	0	(12)	∞	∞	14	∞	20
B	12*	0	12	10	(6)	∞	∞
C	∞	12	0	(4)	∞	∞	∞
D	∞	10	4*	0	∞	(6)	∞
E	14	6*	∞	∞	0	6	8
F	∞	∞	∞	6	6	0	(4)
G	20	∞	∞	∞	8	4	0



Topological Sorting

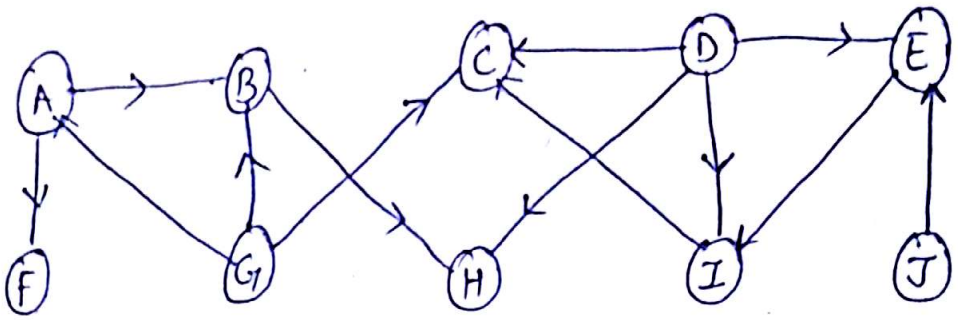
For a directed graph, for every edge (u,v), vertex u comes before v in the ordering.

eg.



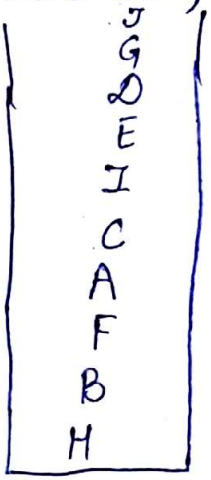
Sorting = ABCDE
= ACBDE

eg.



(1) choose one vertex and visit all its neighbours.

(2) when the neighbours are finished, push it onto the stack.



Steps

•) A → B → H (push H)

A → F (push F)

•) Now go to unvisited C, no neighbours, push C.

•) " " " D.

D → E → I (push I)

push D.

•) Visit G. Done → push

•) Visit J. All neighbours are done. Push

Ans. Print the stack

= J G D E I C A F B H