Graph

→ Non-linear data structure. Consisting of nodes & edges.

→ A graph 'G' is an ordered pair of a set 'V' of vertices and a set 'E' of edges.

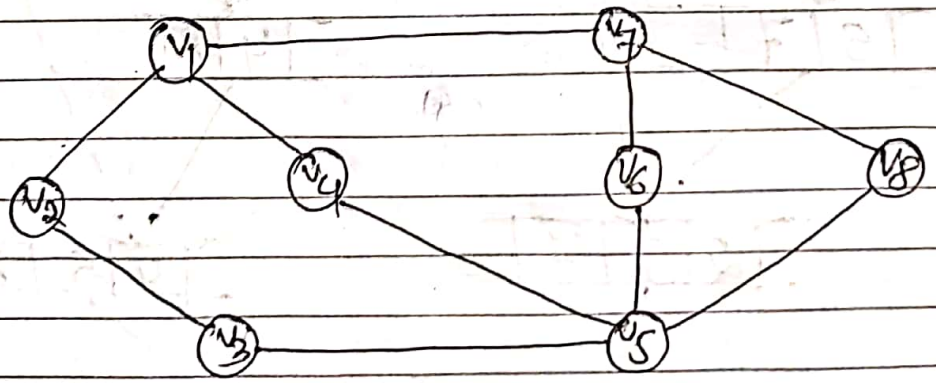
$$G = (V, E)$$

ordered pair :

$$(a, b) \neq (b, a) \text{ if } a \neq b.$$

unordered pair :

$$\{a, b\} = \{b, a\}$$



Graph

$|V|$ → no. of vertices in a graph

$|E|$ → no. of edges in a graph.

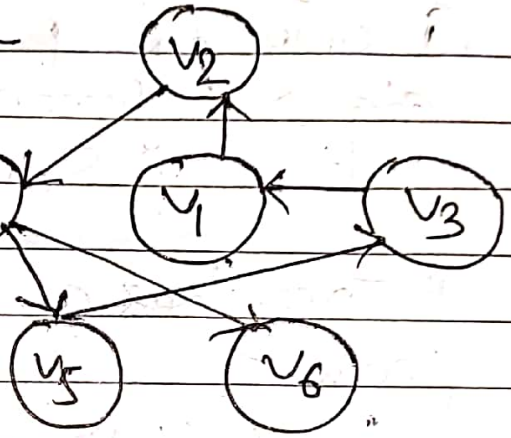
Tree	Graphs
(i) only one path/ edge between two nodes.	(i) multiple paths / edges between two nodes / vertices.
(ii) Tree has a Root node.	(ii) Graphs don't have Root node.
(iii) Tree don't have loops.	(iii) Graphs can have a loops.
(iv) It have $(N-1)$ number of edges. where, $N \rightarrow$ no. of edges	(iv) number of edges not define.
(v) Tree follow Hierarchical model.	(v) It follow network model.

Types of Graph

(i) Directed graph (Digraph)
(ii) undirected graph

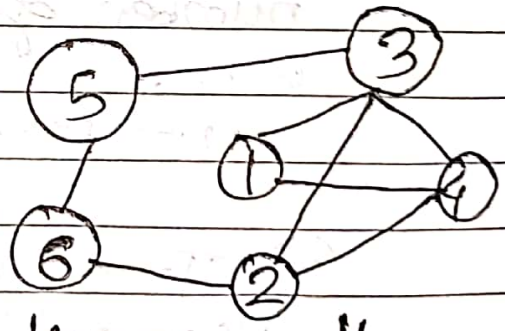
(I) Directed Graph (Digraph)

A directed graph is a set of vertices (nodes) connected by edges, with each node having a direction associated with it.

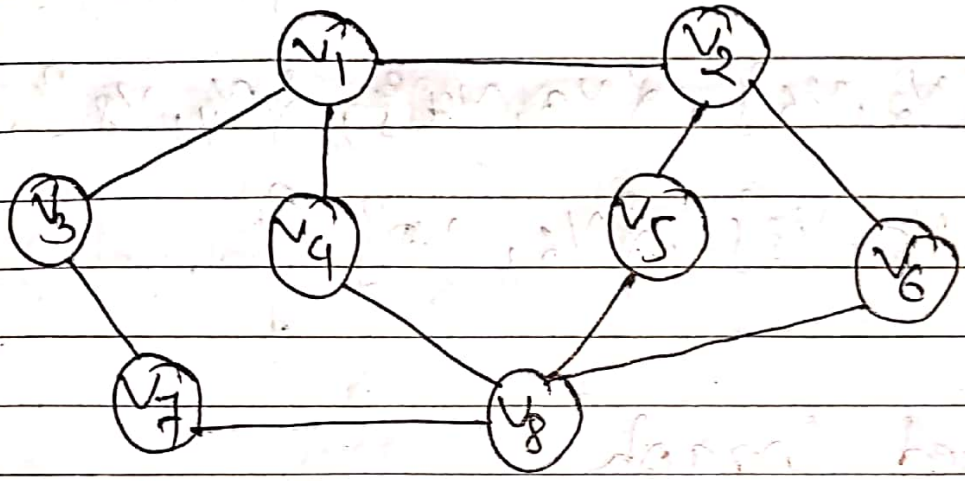


Edges are usually represented by arrows pointing in the direction the graph can be traversed.

(II) Undirected Graph

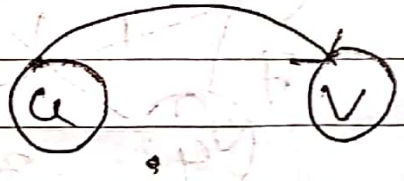


In an undirected graph the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tell us that the graph is undirected.



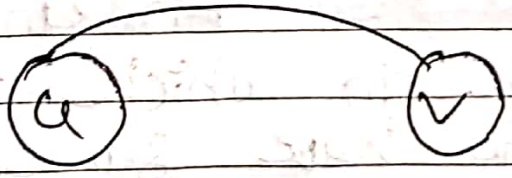
$V = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8 \}$

Edges :- Connection between two vertices.



(u, v)

directed edge

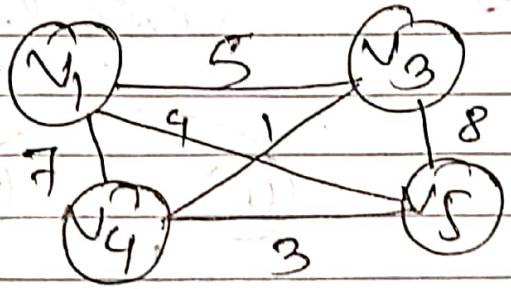


undirected edge (u, v) (v, u)

$E = \{ \{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_5\}, \{v_2, v_6\}, \{v_3, v_7\}, \{v_4, v_8\}, \{v_7, v_8\}, \{v_5, v_8\}, \{v_6, v_8\} \}$.

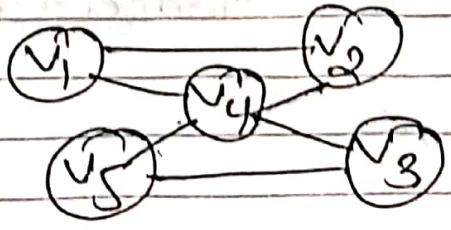
Weighted Graph

A weighted graph is a graph in which each branch is given a numerical weight.



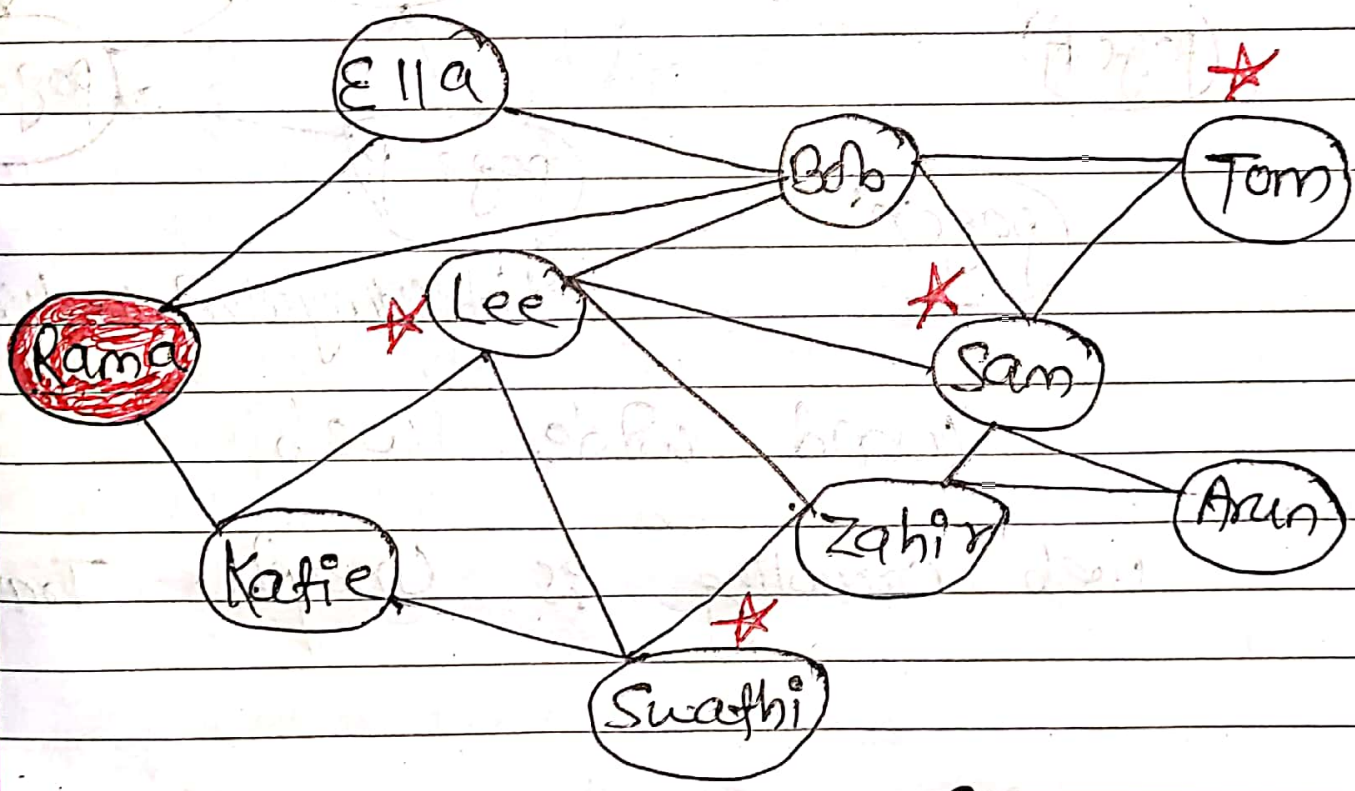
Unweighted Graph

An unweighted graph is a graph in which all edges / paths are considered to have same weight.



Examples of Graphs

1. Social network
2. maps (Google maps) / GPS / Navigation / flight
3. World wide web (WWW)

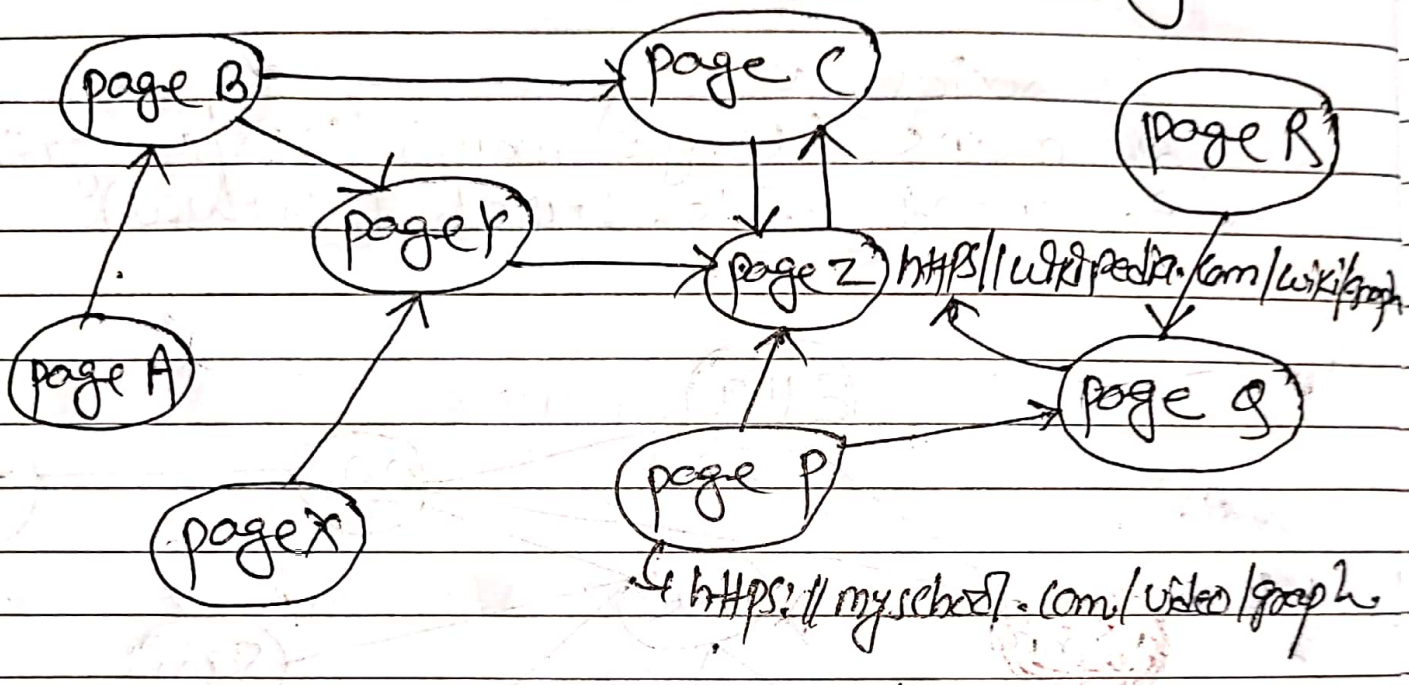


Social network (face book)

Can you suggest some friends to Rama?
→ Lee, Swathi, Sam, Tom.

find all nodes having length of shortest path from Rama equal to 2.

Web Crawling

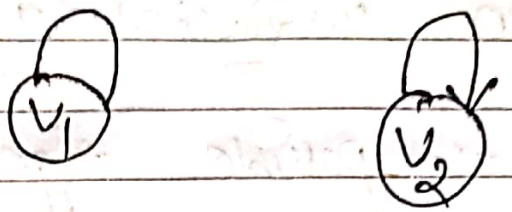


word wide web

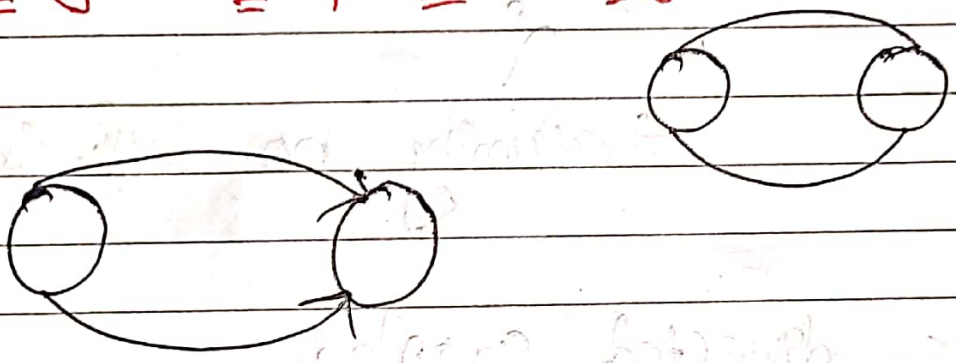
web crawling is Graph Traversal.

Social network → unweighted undirected graph.
WWW → unweighted directed graph.

Self loop :-



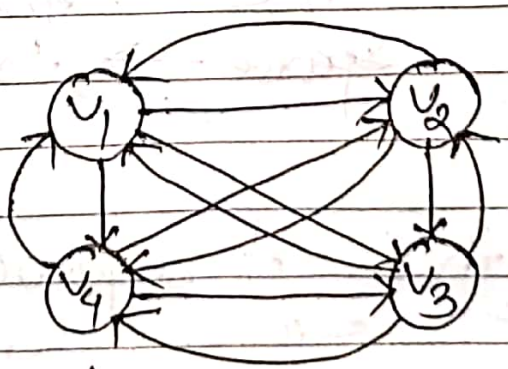
multiedge or parallel edge :-



If there are no self loops or multiedge it's simple graph.

number of edges in a simple graph

In directed graph



max. possible edge :
if $|V| = n$ then,

$0 \le |E| \le n(n-1)$, if directed.

$= n(n-1)$
$= 4(4-1)$
$= 4 * 3 = 12$ edges at max.

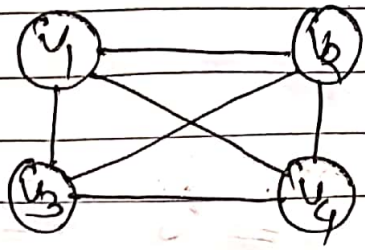
$V = \{v_1, v_2, v_3, v_4\}$
$|V| = 4$

min. possible edge $= \phi$

In undirected graph.

max. possible edges.

$$0 \leq |E| \leq \frac{n(n-1)}{2}$$



↳ assuming no self-loop or multiedge.

for directed graph,

if $|V| = 10$, $|E| \leq 90$

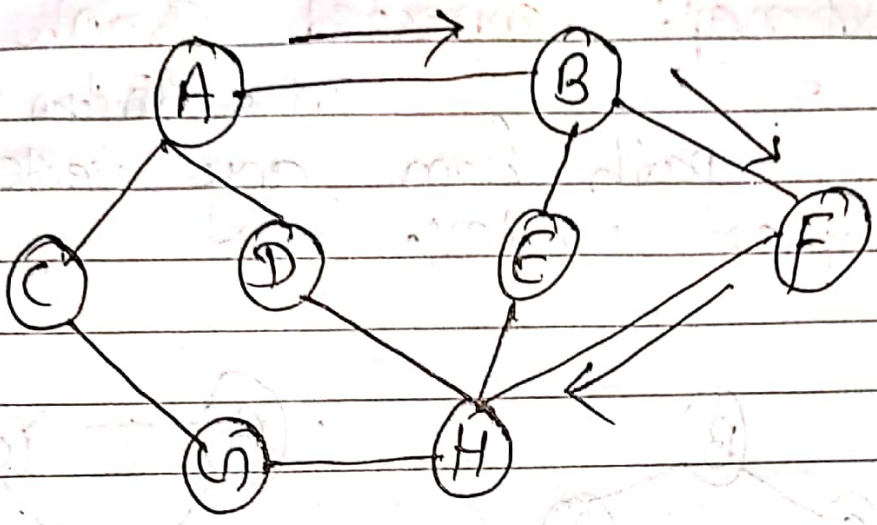
if $|V| = 100$, $|E| \leq 9900$

dense → too many edges

sparse → too few edges

path :- A sequence of vertices where each adjacent pair is connected by an edge.

$< A, B, F, H >$



Simple path :- A path in which no vertices (& thus no edges) are repeated.

Eg $\langle A, B, F, H \rangle$

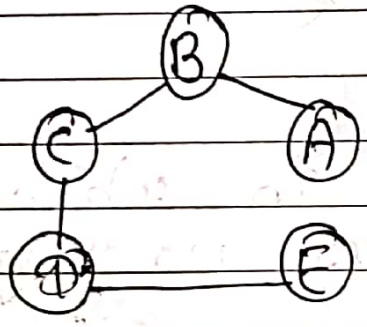
Walk :- A sequence of vertices where each adjacent pair is connected by an edge.

A simple path :- A walk in which no vertices (and thus no edges) are repeated.

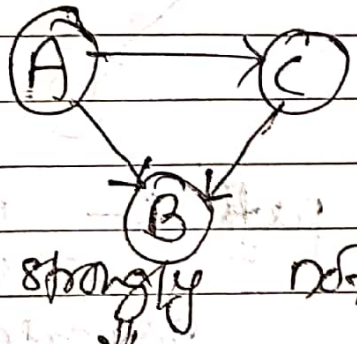
Trail :- A walk in which no edges are repeated.

Strongly Connected Graph

In a directed graph, if there is a path from any vertex to any other vertex.

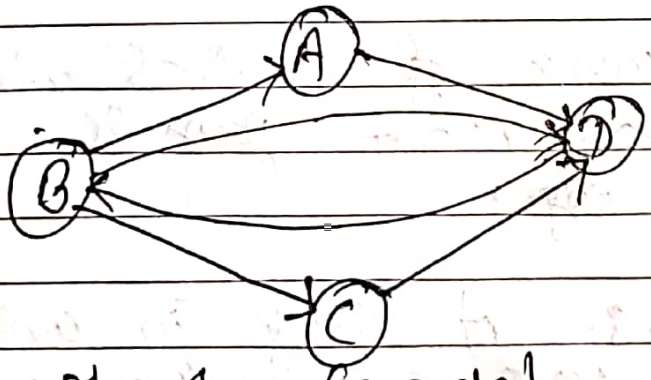


Connected

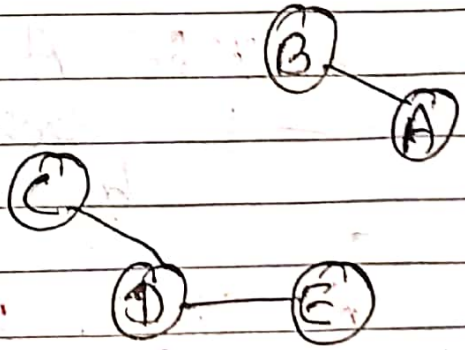


strongly not Connected

weakly Connected



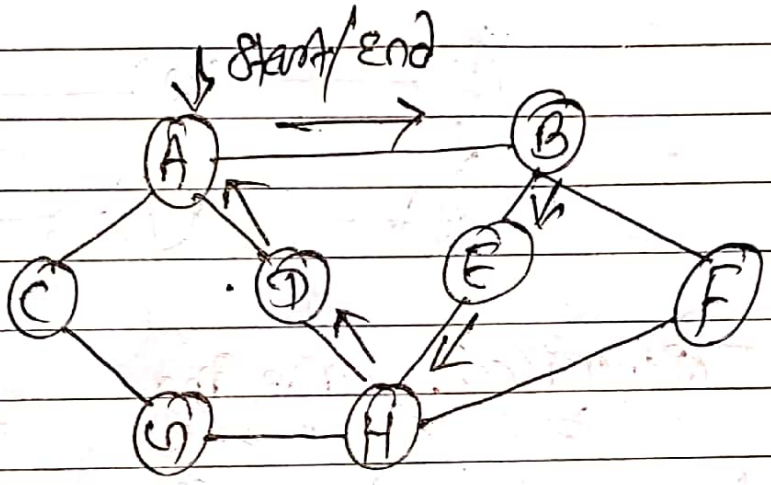
strongly Connected



not Connected

Closed walk :- Starting and ending vertex is same.

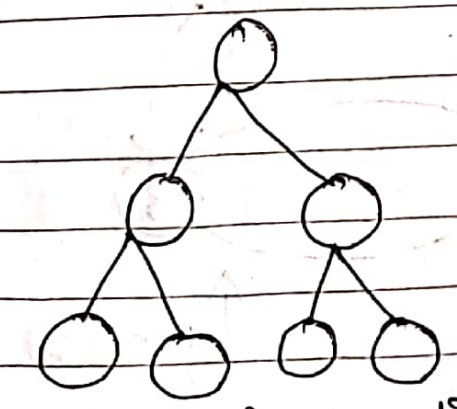
length should be,
$L > 0$



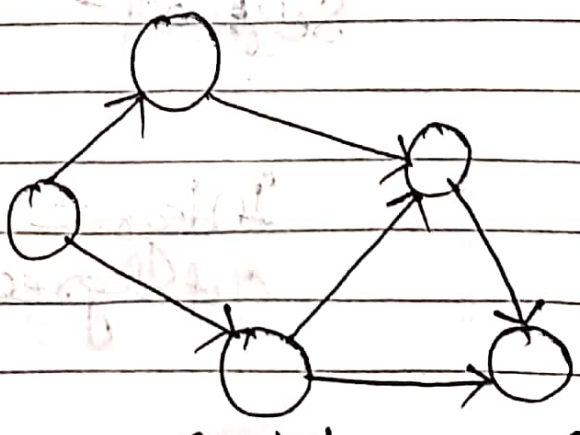
Cycle :- A cycle is a non empty trail in which the only repeated vertices are the first & last vertices.

Simple cycle :- No repetition other than start and end.

Acyclic Graph :- A graph with no cycle.

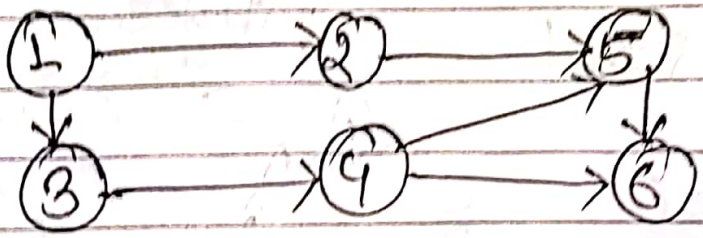


undirected acyclic graph.



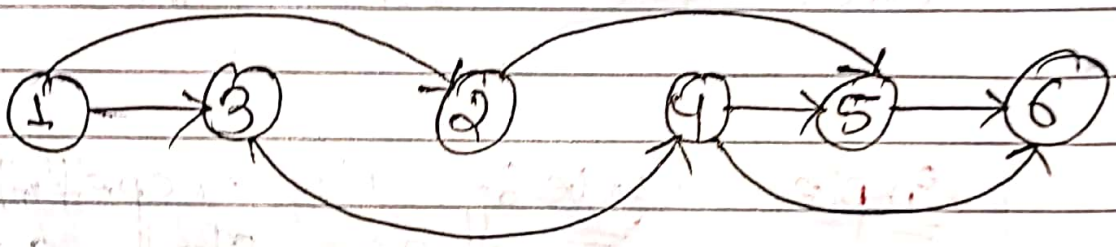
Directed acyclic graph (DAG)

Directed Acyclic graph (DAG)

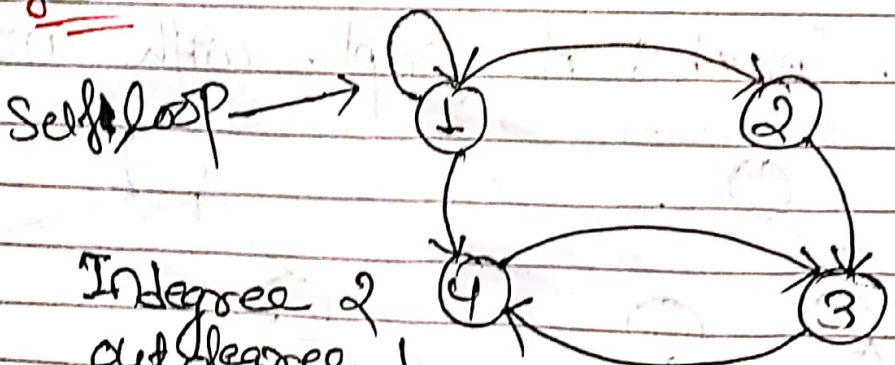


Topological ordering

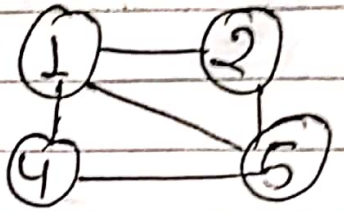
If a DAG can be arranged linearly such that edges are going only in forward direction such graph is called Topological ordering.



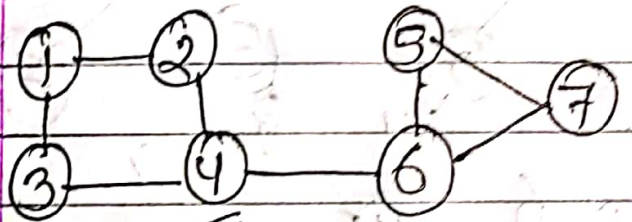
Degree



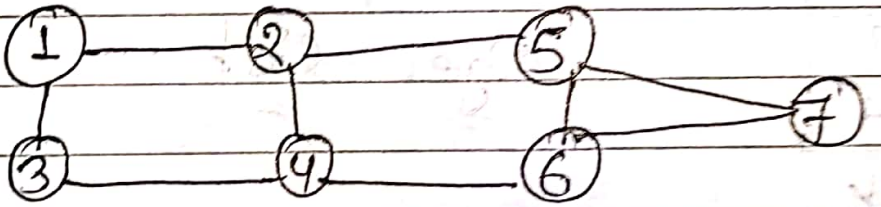
degree = 3



Articulation point :- If there are any vertices whose removal split the graph into 2 components is called articulation point.



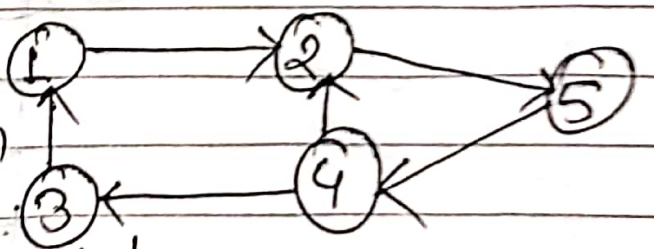
articulation point (4 & 6)



no articulation point.

Strongly Connected

In directed graph, if we can visit all the vertices from any vertices is called strongly connected graph.



Graph Representation

① Edge list

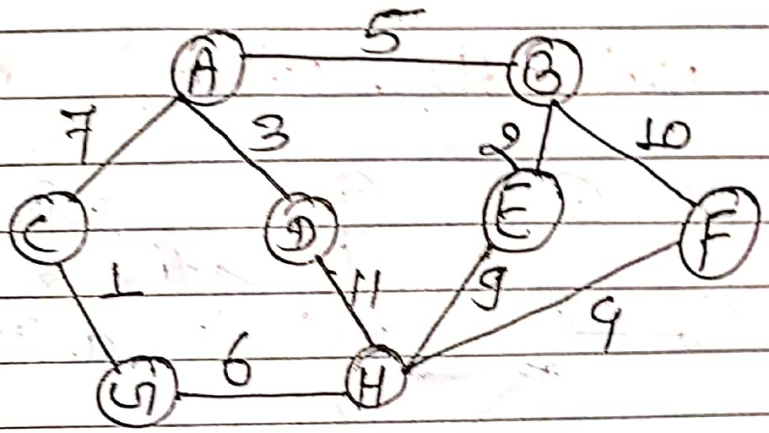
Vertex list

A
B
C
D
E
F
G
H

↓

|V| rows

$O(|V|)$



Edge list

A	B	5
A	C	7
A	D	3
B	E	2
B	F	10
C	G	1
D	H	11
E	H	9
F	H	9
G	H	6

```
struct edge {
        char * start_vertex;
        char * end_vertex;
        int weight;
};
```

OR

```
class Edge {
public:
    string start_vertex;
    string end_vertex;
    int weight;
};
```

```
string vertex_list [MAX_SIZE];
Edge edge_list [MAX_SIZE];
```

Cost :-

(i) space complexity → O(|V| + |E|)

(ii) finding adjacent nodes → O(|E|)

(iii) check if given nodes are connected → O(|E|)

Time Cost of operation → Time complexity.

Memory usage → space complexity.

In simple graph,

if $|V| = n$

then,

$0 \leq |E| \leq n(n-1)$, if directed.

$0 \leq |E| \leq \frac{n(n-1)}{2}$, if undirected.

$|V| = V$

$|E| = V(V-1)$

$= V^2 - V$

$= O(V^2)$

10 → 90

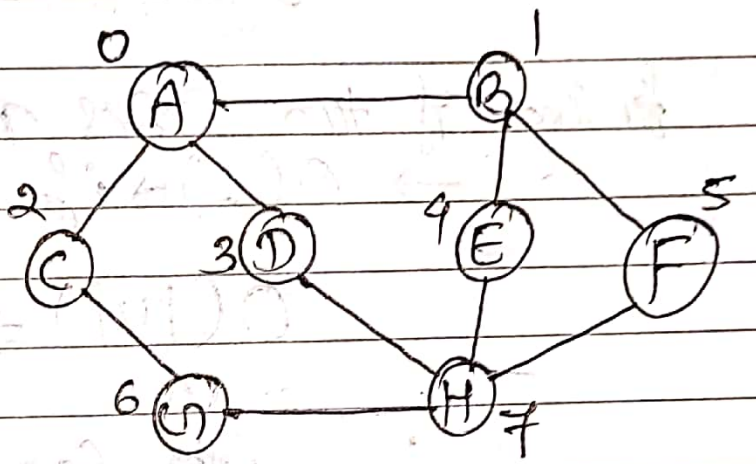
100 → 9900

1000 → 999000

$O(|E|)$ or $O(|V| * |V|)$ → costly

$O(|V|)$ → OK

(ii) Adjacency matrix



Vertex list

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

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

$$A_{ij} = \begin{cases} 1, & \text{if } j \text{ edge from } i \text{ to } j. \\ 0, & \text{otherwise} \end{cases}$$

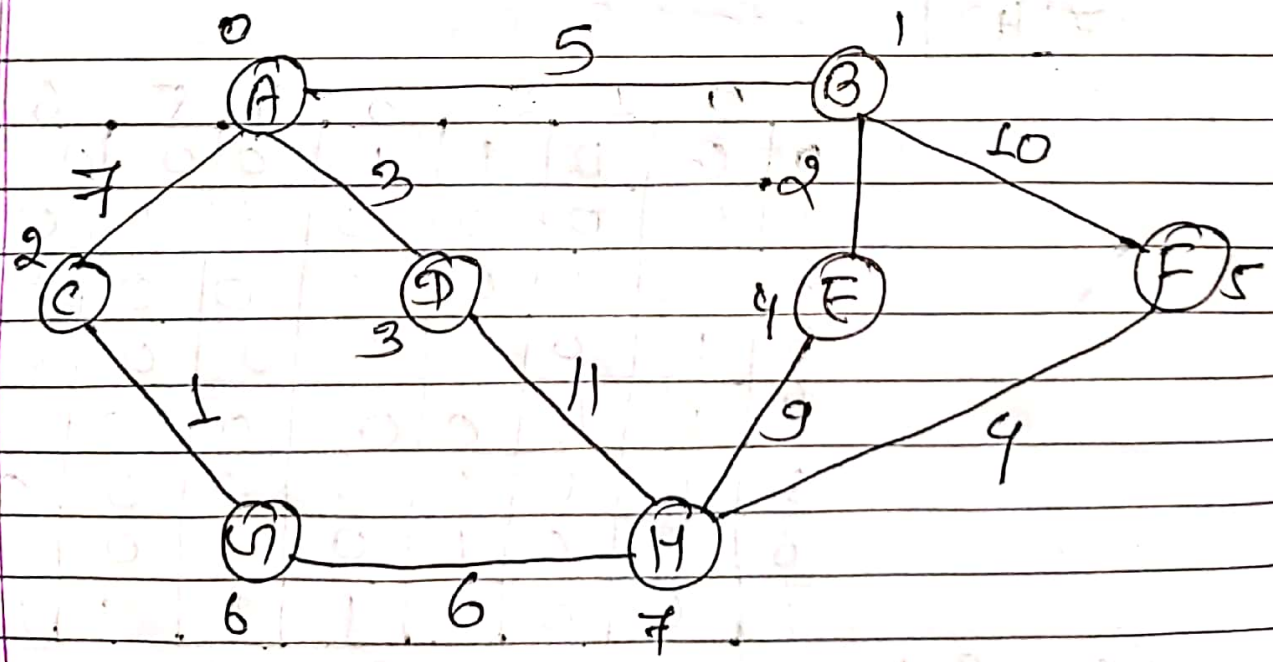
finding adjacent nodes
→ O(|V|) + O(|V|)

finding if two nodes are connected or not.
→ O(1) → if index are given

O(|V|) ⇒ overall

↑
we can use hash table to query flags.

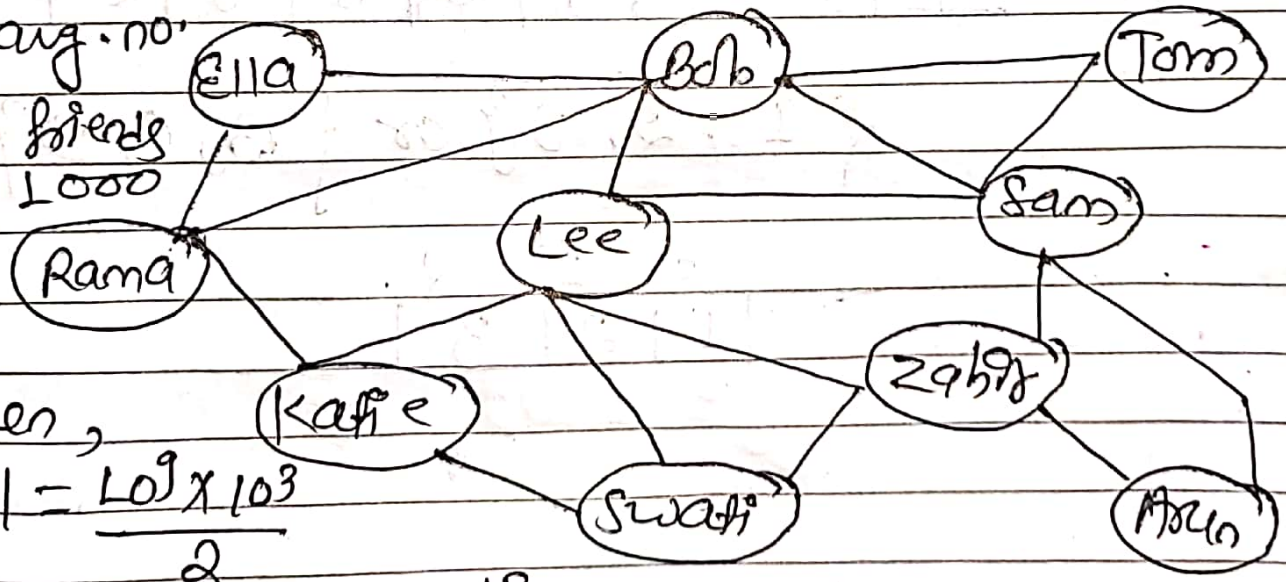
for weighted graph



	0	1	2	3	4	5	6	7	
0	∞	5	7	3	∞	∞	∞	∞	
1	5	∞	∞	∞	2	10	∞	∞	
2	7	∞	∞	∞	∞	∞	1	∞	
3	3	∞	∞	∞	∞	∞	∞	11	$\|V\| = V$
4	∞	2	∞	∞	∞	∞	∞	9	
5	∞	10	∞	∞	∞	∞	∞	4	
6	∞	∞	1	∞	∞	∞	∞	6	10×64
7	∞	∞	∞	6	11	9	4	∞	

↳ $O(V^2)$ space

If $|V| = 10^9$
If avg. no. of friends $= 1000$

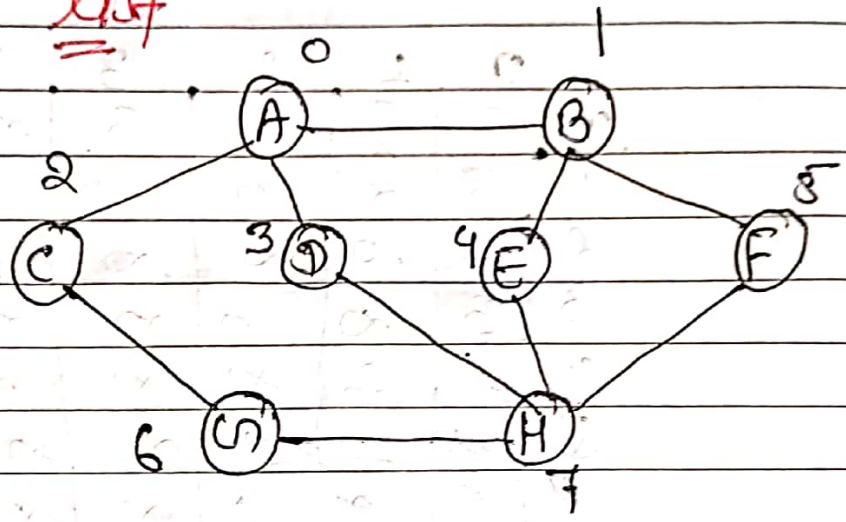


Social network

then,
$|E| = \frac{10^9 \times 10^3}{2}$
$= 5 \times 10^{11} << 10^{18}$

10^{18} bytes $\simeq$ 1000 PB
5×10^{11} bytes $\simeq$ 0.5 TB

(iii) Adjacency List



Connection of node (A)

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

{1, 2, 3} or {B, C, D}

0	1	2
1	2	3

Structure -1

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

Structure -2

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

OR

↓

Space = $O(V^2)$

↓

Time = $O(1)$

(int A[8][8]

int *A[8];

A[0] = new int[3];
A[1] = new int[3];
A[2] = new int[2];
:
A[7] = new int[4];

↓

Space = $O(e)$

$e << v^2$

↓

Time = $O(V)$

↳ linear search

for finding adjacent nodes.

→ $O(V)$ in both matrix.

for a social network with a billion (10^9) users.

If max. no. of friends = 10,000
if machine can scan 10^6 cells / second

Structure - 1

finding adjacents nodes $= \frac{10^9}{10^6}$
$= 1000$ sec
$= 16.66$ min

Structure - 2

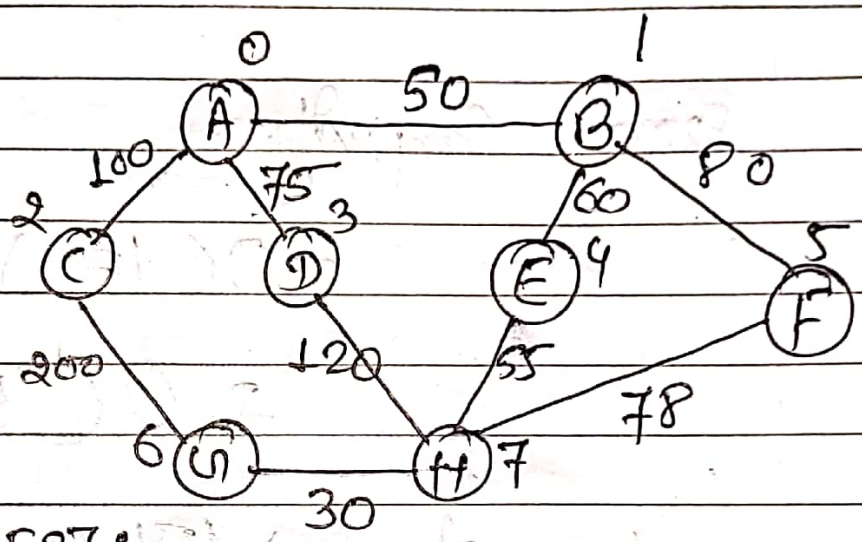
$= \frac{10^9}{10^6}$
$= 10^{-2}$ sec
$= 10$ ms

finding if two nodes are connected,

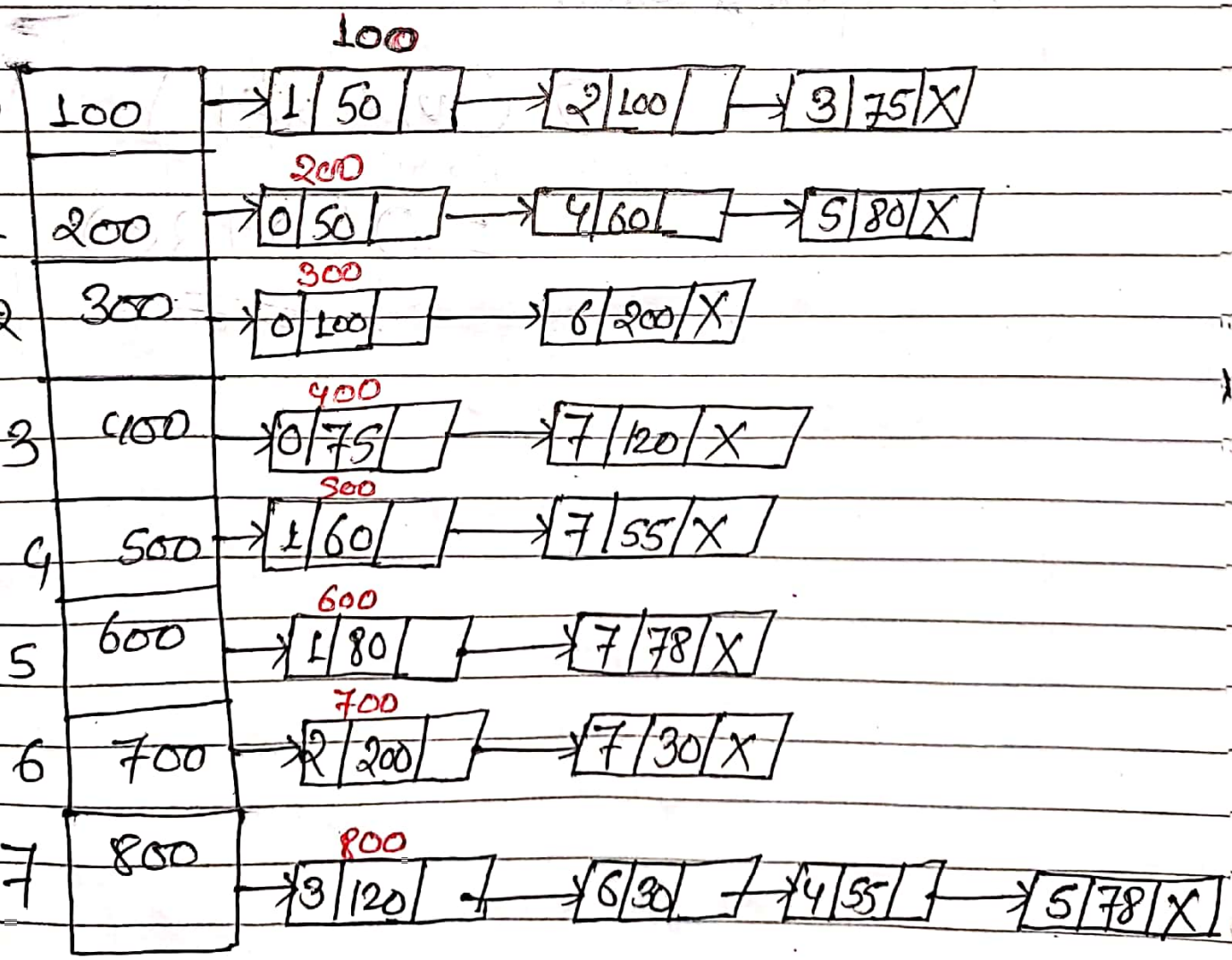
Read A[i][j]
$= \frac{1}{10^6} = 1$ μs

$= \frac{10^9}{10^6}$
$= 10$ ms

```
struct node {
  int data;
  int weight;
  struct node *next;
};
```




```
struct node *A[8];
```



Space Complexity,

$$= O(|E|) + O(|V|)$$

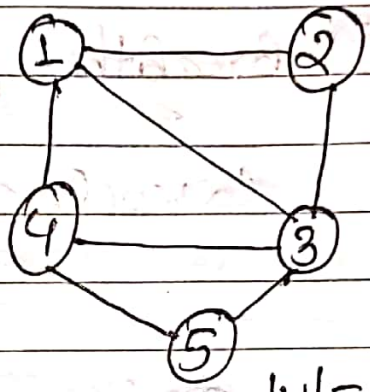
$$= O(|E| + |V|)$$

must graph $\longrightarrow |E| << |V| * |V|$
one sparse with.

$$O(|V| + 2|E|)$$

$$= O(n + 2e)$$

(iv) Compact list



$|V| = n = 5$

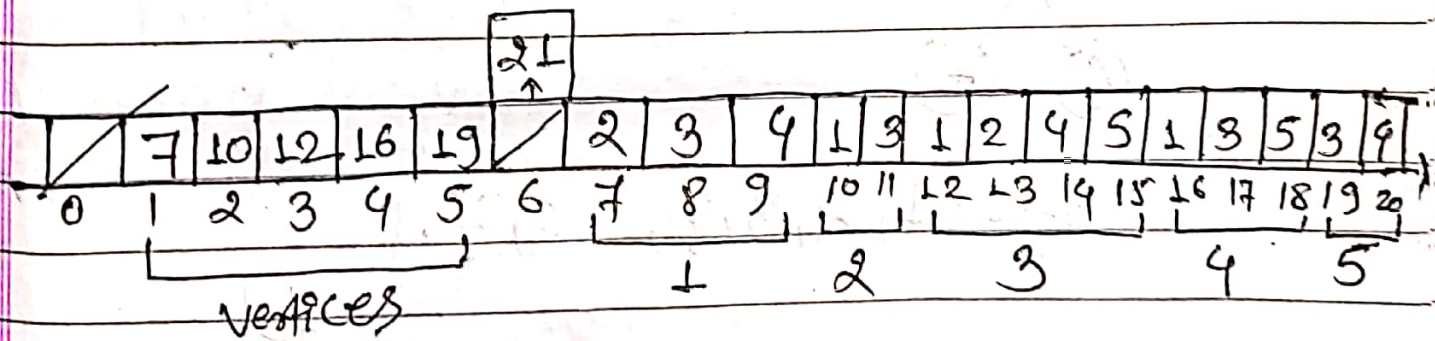
$|E| = e = 7$

$|V| + 2|E| + 1$

$= 5 + 2 \times 7 + 1$

$= 20$

$= 20 + 1 = 21$



$|V| + 2|E|$

$n + 2e$

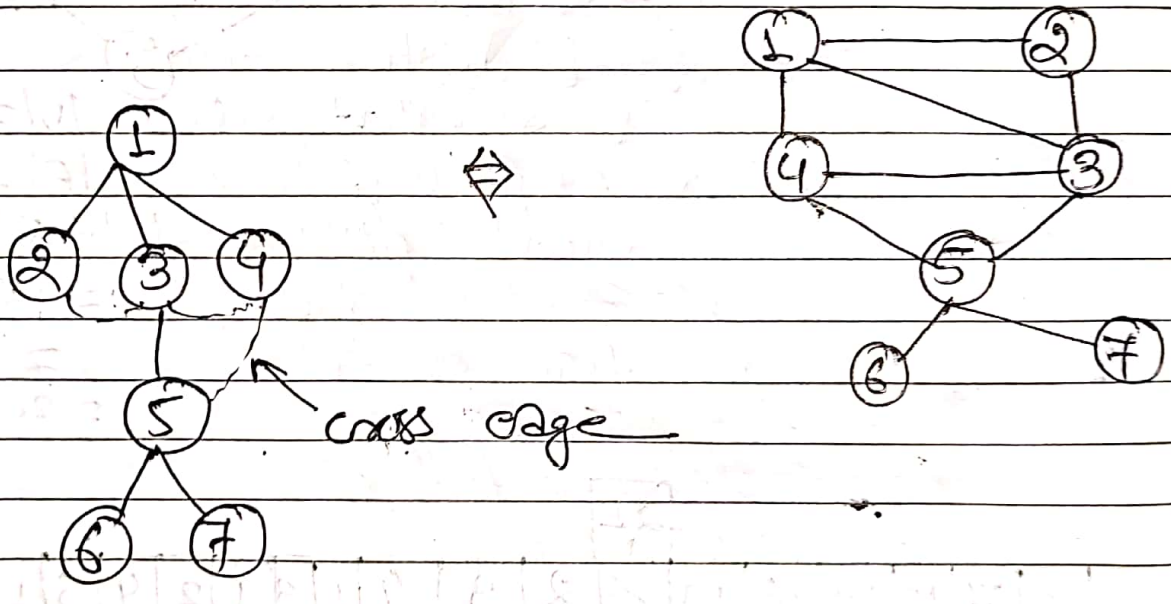
$n + 2n = 3n$

$O(n)$

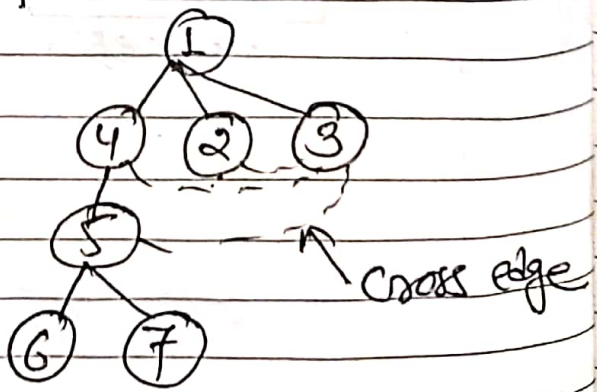
Graph Traversal

(1) Breadth first search (BFS)

↳ level order traversal.



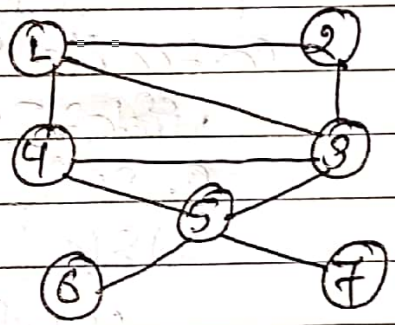
BFS:- 1, 2, 3, 4, 5, 6, 7.



BFS:- 1, 4, 2, 3, 5, 6, 7

Algo

(I) Create queue.
(II) mark all vertices as unvisited.
(III) display, push the starting vertex in queue. then mark starting vertex as visited.
(IV) while queue is not empty perform operation.



A =

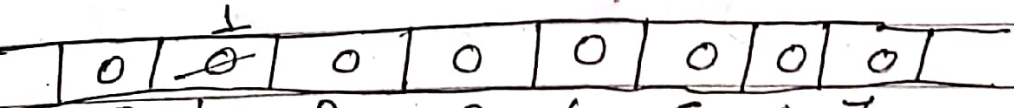
	0	1	2	3	4	5	6	7
0								
1		0	1	1	1	0	0	0
2		1	0	1	0	0	0	0
3		1	1	0	1	1	0	0
4		1	0	1	0	1	0	0
5		0	0	1	1	0	1	1
6		0	0	0	0	1	0	0
7		0	0	0	0	1	0	0

u →

O(n²)

queue

visited



0 1 2 3 4 5 6 7

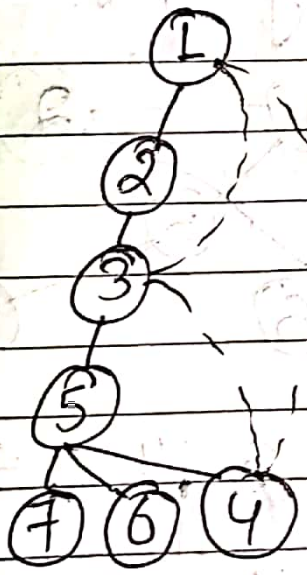
// program for BFS

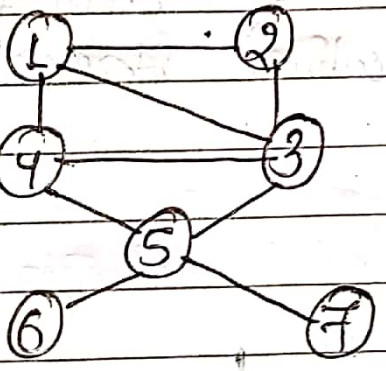
```
void BFS (int Arr[8][8], int starting_vertex, int size){
    int u;
    queue <int> q;
    int visited[8] = {0};
    cout << starting_vertex << " ";
    visited[starting_vertex] = 1;
    q.push (starting_vertex);
    while (!q.empty ()){
        u = q.front ();
        q.pop();
    for (int v=1; v<size; v++){
        if (Arr[u][v]==1 && visited[v]==0){
            cout << v << " ";
            visited[v] = 1;
            q.push (v);
        }
    }
}
```

(11) Depth first Search (DFS)

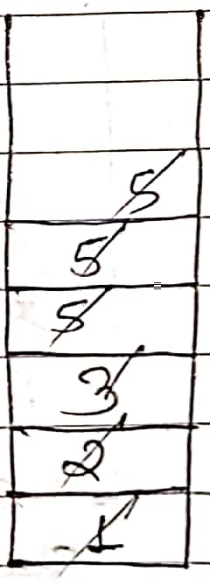
(i) visited

(ii) Exploring.





DFS :- 1, 2, 3, 5, 7, 6, 4



Stack

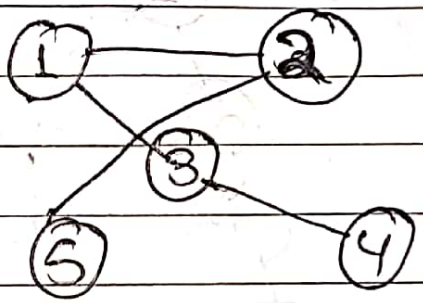
Algo

(i) Create stack.

(ii) mark all vertices as unvisited.

(iii) display, push the starting vertex in stack & mark starting vertex as visited.

(iv) while stack is not empty perform operation



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

4
3
5
2
1
stack

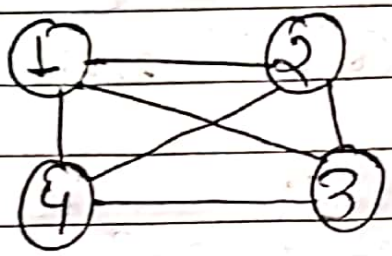
1	1	1	1	1	1
0	1	2	3	4	5

visited

// program for DFS traversal

```
void DFS(int Arr[6][6], int starting_vertex, int size){
    int u;
    stack<int> S;
    int visited[size] = {0};
    cout << starting_vertex << " ";
    visited[starting_vertex] = 1;
    S.push(starting_vertex);
    while(!S.empty()){
        int flag = true;
        u = S.top();
        for(int v=1; v<size; v++){
            if(Arr[u][v]==1 && visited[v]==0){
                flag = false;
                cout << v << " ";
                visited[v] = 1;
                S.push(v);
                break;
            }
        }
        if(flag==true){
            S.pop();
        }
    }
}
```

Spanning tree



In a connected graph, Spanning tree is a sub graph having same number of vertices & 'n-1' edges. where n → vertex.

$G = (V, E)$

$n = |V|$

$e = |E|$

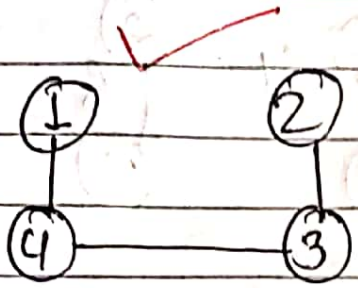
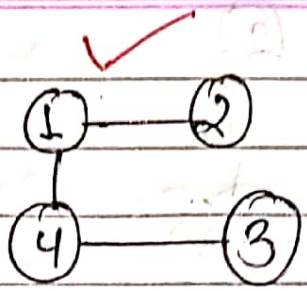
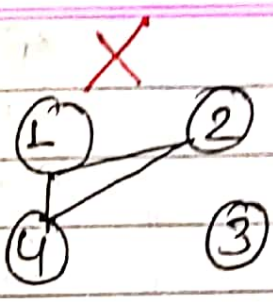
$S \subseteq G$

$S = (V', E')$

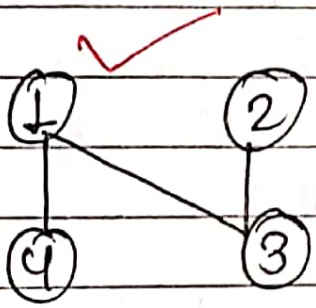
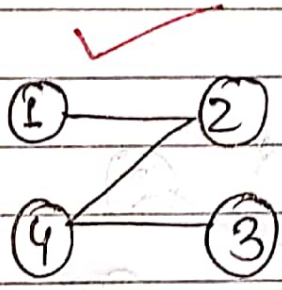
$|V'| = |V|$

$|E'| = |V| - 1$

& there should not be any cycle.

It is having a cycle.

total number of cycle.

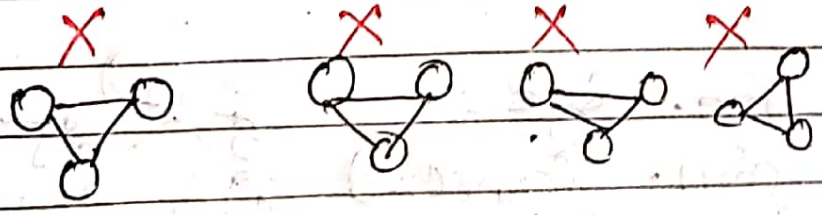
$|V| = 4$

$|E| = 6$

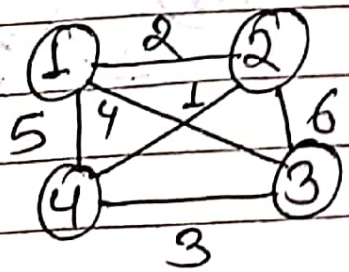
$$C_{|V|-1} = {}^{6}C_{4-1} = {}^{6}C_{3} = 6C_{-4}$$

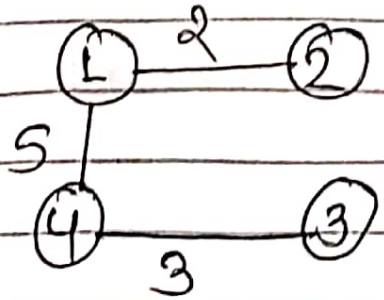
$= 20 - 4$

$= 16$ ways

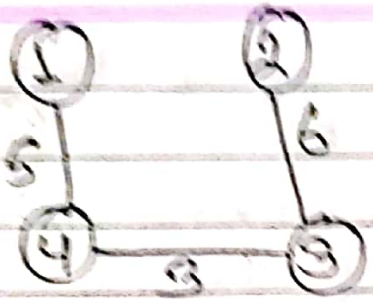


minimum cost spanning tree

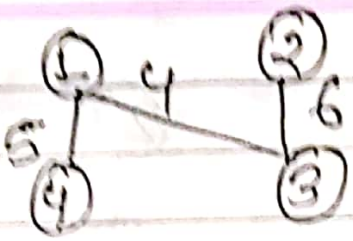




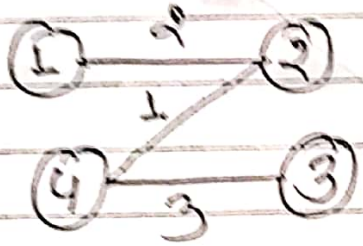
Cost = 10



Cost = 14



Cost = 15

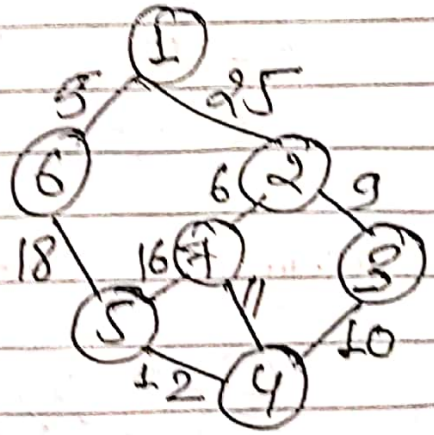


cost = 6

Prim's minimum cost spanning tree

Algo

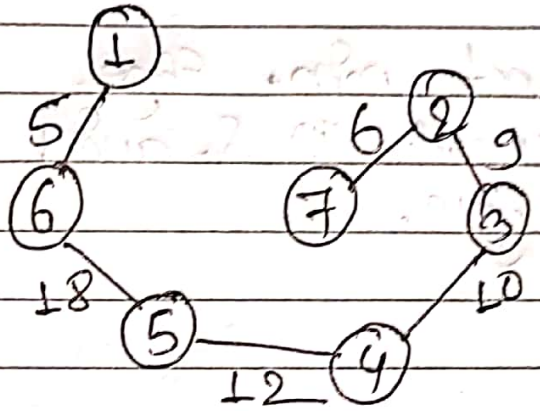
(I) Select min. edge
(II) Select min. connected edge from (I).
(III) repeat (I) & (II).



$(|V|-1)(|E|)$

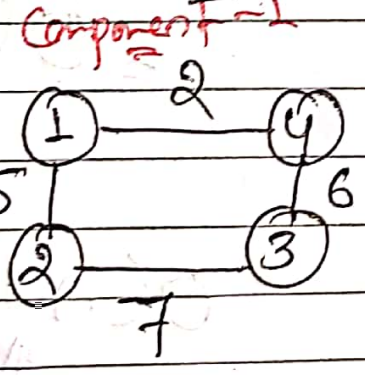
$n \quad e = n*n$

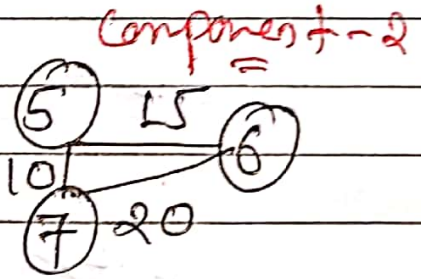
$O(n^2)$



cost = 5+18+12+10+9+6 = 60

for non-Connected graph

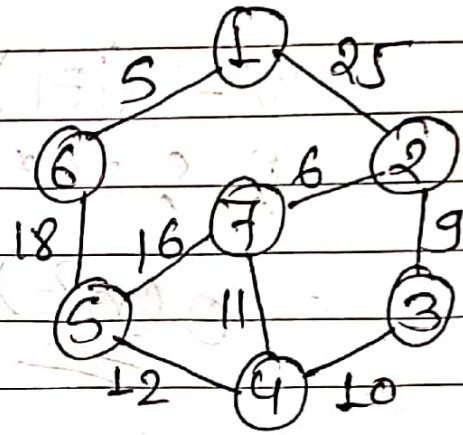




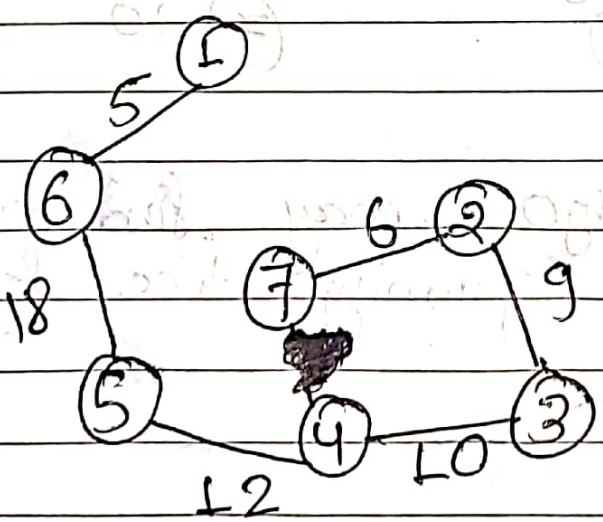
so, prim's algo. may find out min. cost spanning spanning tree for one component.

Kruskal's minimum cost spanning tree

(I) Select min. edge
(II) repeat (I) & avoid cycle.



$(|V|-1)\,|E|$
$= O(n^2)$

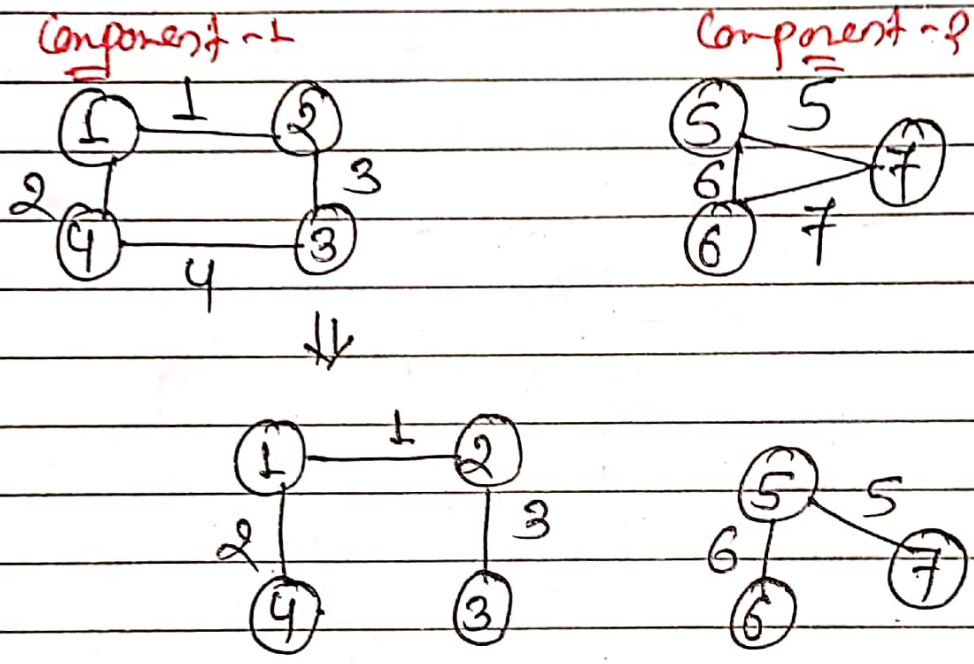


$O(n \log n)$

↑
if Heap is used.

Cost = 60

for non connected graph



Kruskal's algorithm can find spanning tree for individual component.