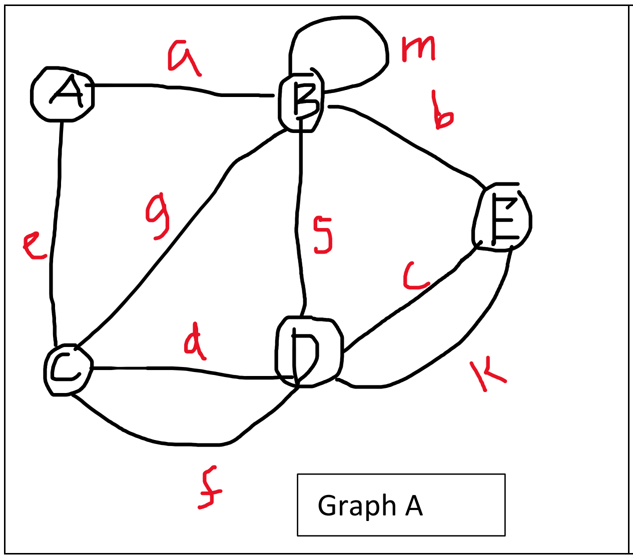
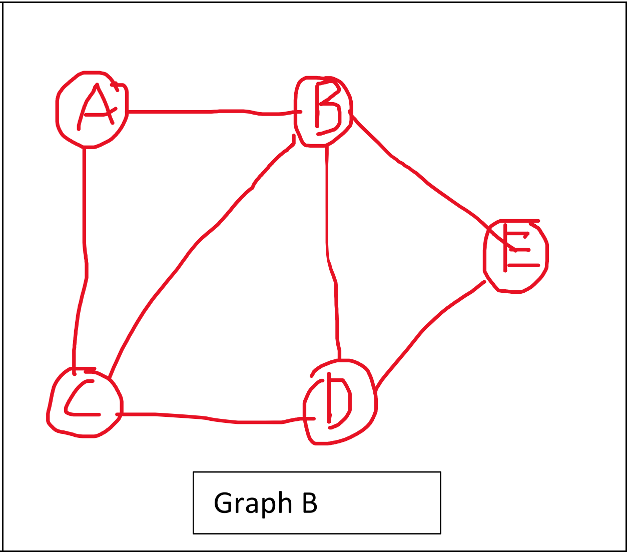
Lesson-16-Graphs Introduction
Slide-7- Exercise





Q.No 1-9 for Graph A.

1. **|V| = 5**
2. **|E| = 10**
3. Name the end points of e and f.

   e = (A,C) f = (C,D)
4. Does A and D are adjacent? No
5. Does B and C are adjacent? Yes
6. deg(A) = 2 , deg(D) = 5

7. Name the Parallel edges in the given graph.

   c,

8. Name the Self loop in the given graph A. m

9. Does a Graph A is simple graph ? No

10. Does a Graph B is simple graph ? Yes

<ins>**Slide-10**</ins>

Property 1

$\sum_v \deg(v) = 2m$

Proof: Let E_v = {e | e incident to v}.

- E_v is a set, which represents the set of edges incident to a specific vertex v in a graph.
- **e** stands for edges.
- | This symbol means "such that" or "where." It is used to specify the condition that the elements in the set E_v must satisfy.

- Condition: an edge is said to be "incident" to a vertex if that vertex is one of the endpoints of the edge.

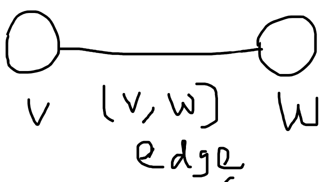
In summary, the set E_v is a collection of edges, where each edge is incident to the vertex v. For every vertex v in the graph, you can create a corresponding set E_v that contains all the edges that are connected to that vertex.

Then $\sum_v \deg(v) = \sum_v |E_v|$

- $\sum_v \deg(\mathbf{v})$ - sum of the degrees of all vertices in the graph

- $\sum_v |E_v|$ - This represents the sum of the sets E_v for all vertices v in the graph. In other words, it is the sum of the number of edges incident to each vertex.

Notice every edge (v,w) belongs to just two of these sets: E_v and E_w. So, every edge is counted exactly twice.



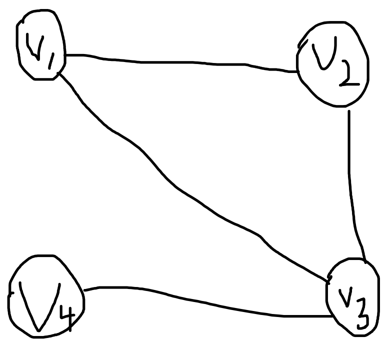
$$E_v = \{vw\}$$

$$E_w = \{vw\}$$

So, **$\sum_v \deg(v) = 2m$ is proved.**

Example: **$\sum_v \deg(v) = 2m$**

From the graph below sum of deg(v1) + deg(v2) + deg(v3) + deg(v4)
= 2 + 2 + 3 + 1 = 8 = 2 * number edges
= 8 = 2 * 4
$\sum_v$ deg(v) = 2m



Summary: Counting Edges by Vertex Degrees

- Let G be an undirected graph with n vertices and m edges.
- The degree of a vertex in a graph is the number of edges incident to that vertex.
- If we sum up the degrees of all vertices in the graph, it should count each edge exactly twice because each edge contributes to the degree of two vertices (one at each end of the edge)

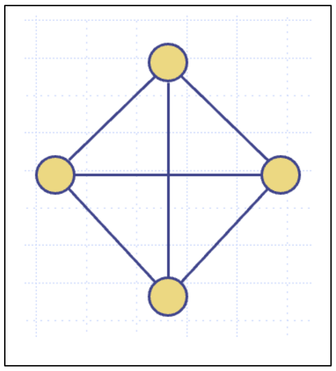
Property – 2
$m \le n(n-1)/2$
Proof: max number of edges is
$C(n, 2) = C_{n,2} = n(n-1) / 2$

Why 2? Two vertices can form an edge.

Easy to understand the proof.

- If a graph has n vertices it might have n-1 edges. Each vertex is connected to each other except itself.
- Counting the total number of edges are n * (n-1)
- As per the Property one, each edge is counted twice. To get the exact number of edges divide the total by 2. That is n * (n-1) / 2

Example 1: Complete graph K_4



Number of vertices n = 4

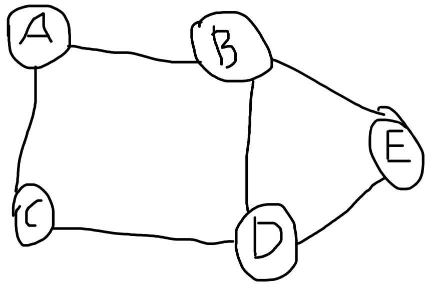
M = n*n-1)/2

Number of edges m = n (n-1) / 2

= 4 (4-1)/2

= 12/2 = 6

Example 2:



Number of vertices n = 5

Number of edges = 6

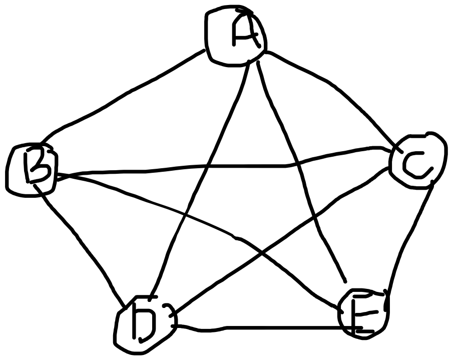
Number of edges m = n (n-1) / 2

= 5 (5-1)/2

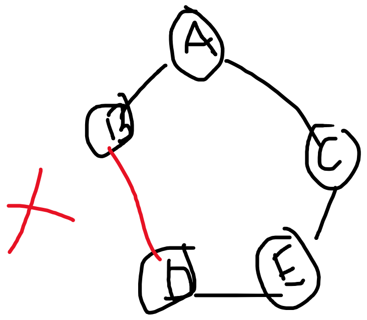
= 20/2 = 10

6<=10

Complete graph denoted using Kn, 4 nodes --> K_4. 5 nodes --> K_5

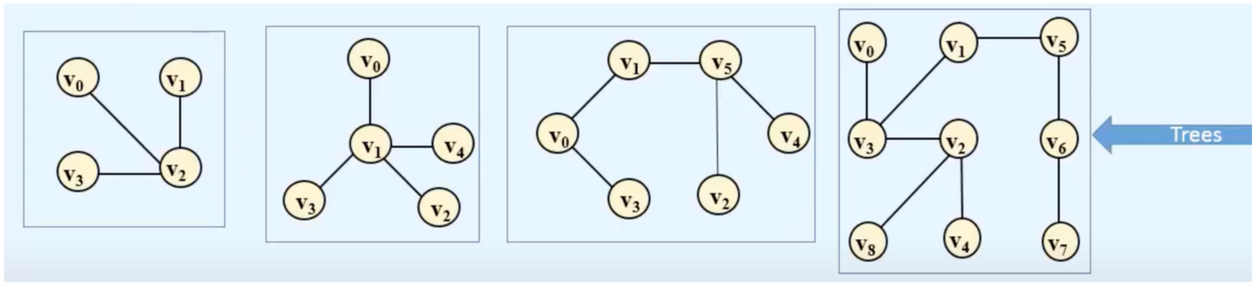


Spanning Tree

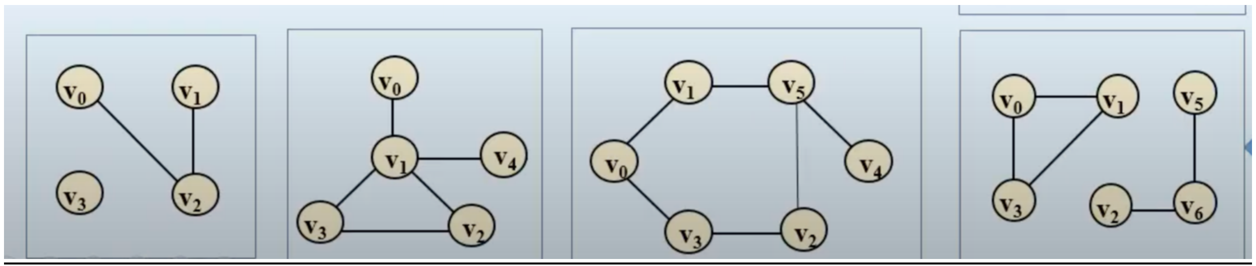


Slide-13

Valid Trees



Invalid Trees



BFS & DFS Algorithm

DFS

```
Algorithm: Depth First Search (DFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.
Initialize a stack S  – O(1)
Pick a starting vertex s and mark it as visited
S.push(s) – O(1)
while S ≠ ∅ do – O(n)
    v ← S.peek()  – O(1)
    if some vertex adjacent to v not yet visited then
        w ← next unvisited vertex adjacent to v
        mark w - O( deg(v)) – O(2m)
        push w onto S  O(1)
    else //if can't find such a w, backtrack
        S.pop()-- O(1)
```

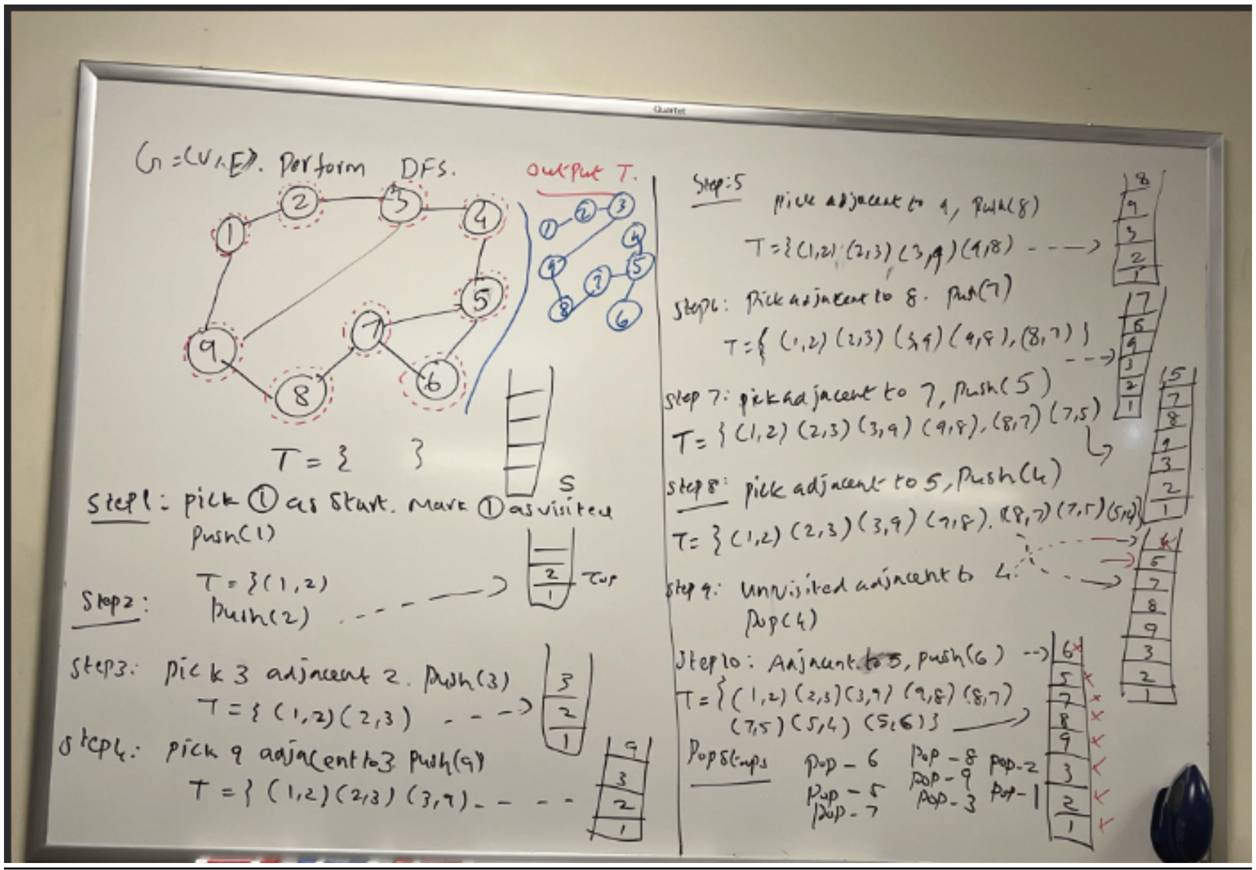
BFS

```
Algorithm: Breadth First Search (BFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.
Initialize a queue Q – O(1)
Pick a starting vertex s and mark s as visited
Q.add(s)  – O(1)
while Q ≠ ∅ do  – O(n)
        v ← Q.dequeue() – O(1)
O(deg(v))for each unvisited w adjacent to v do
            mark w
            Q.add(w) - O(1)
// Efficient to use adjacency list
```

In both algorithm, implemented using Adjacency Matrix is O(n^2), To discover all neighbors of a vertex, it must traverse the entire row which leads to a O(n^2) time complexity.
Adjacency list O(n+m)

n = number of vertices and m = number of edges.

<u>DFS Example</u>



G=(V,E). Perform DFS.

T = { }

Step1: pick ① as Start. Mark ① as visited
push(1)

Step2: T = {(1,2)
push(2)

Step3: pick 3 adjacent 2. Push(3)
T = { (1,2)(2,3)

Step4: pick 9 adjacent to 3 Push(9)
T = { (1,2)(2,3)(3,9)

Step:5 pick adjacent to 9, Push(8)
T = {(1,2)(2,3)(3,9)(9,8)

Step6: Pick adjacent to 8. push(7)
T = { (1,2)(2,3)(3,9)(9,8),(8,7) }

Step 7: pick adjacent to 7, Push(5)
T = { (1,2)(2,3)(3,9)(9,8),(8,7)(7,5)

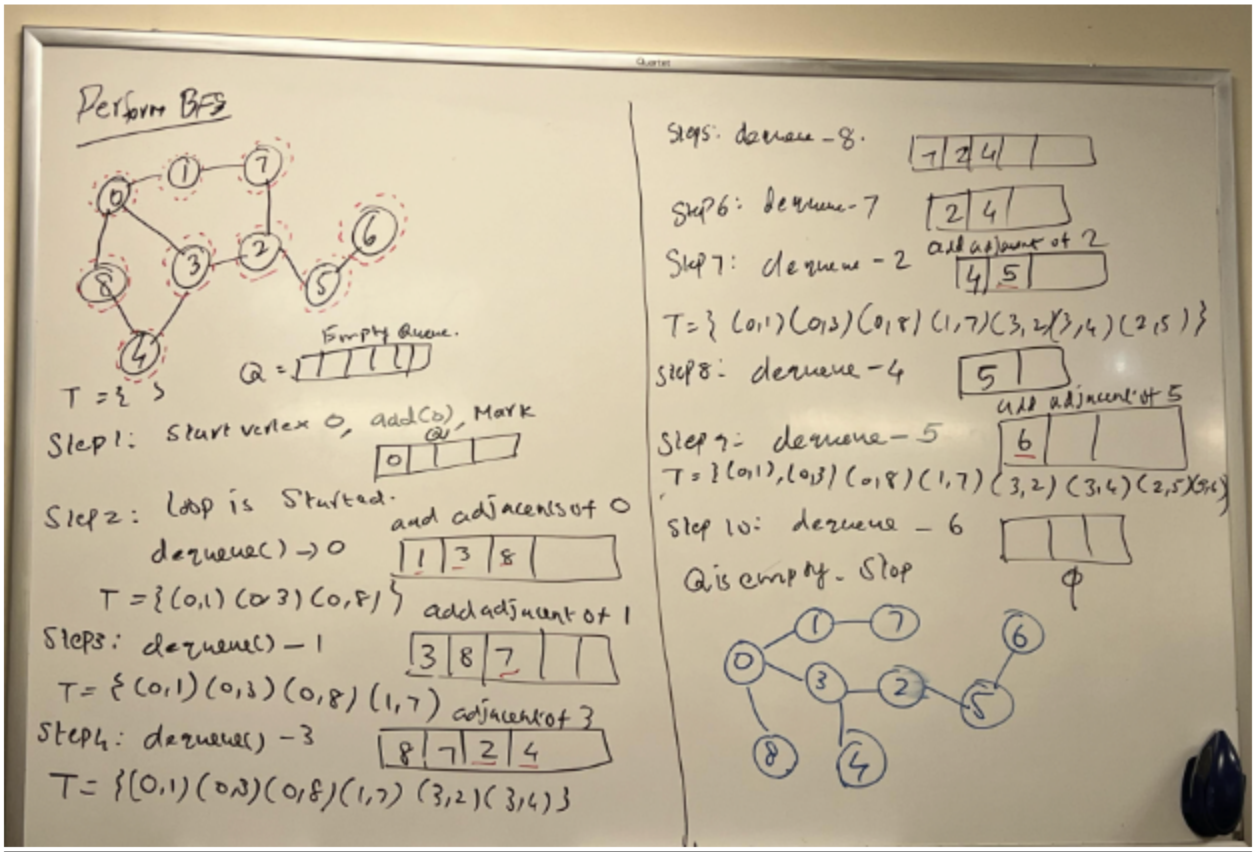
Step8: pick adjacent to 5, Push(4)
T = { (1,2)(2,3)(3,9)(9,8).(8,7)(7,5)(5,4)

Step 9: unvisited adjacent to 4:
Pop(4)

Step10: Adjacent to 5, push(6)
T = { (1,2)(2,3)(3,9)(9,8)(8,7)
(7,5)(5,4)(5,6) }

Pop Steps: Pop – 6, Pop – 5, Pop – 7, Pop – 8, Pop – 9, Pop – 3, Pop – 2, Pop – 1

BFS Example



Perform BFS

Q = Empty Queue.

T = { }

Step 1: Start vertex 0, add(0), Mark Q

Step 2: Loop is Started.

dequeue() → 0 and adjacents of 0

T = {(0,1) (0,3) (0,8)}

Step 3: dequeue() − 1 add adjacent of 1

T = {(0,1) (0,3) (0,8) (1,7)

Step 4: dequeue() − 3 adjacent of 3

T = {(0,1) (0,3) (0,8) (1,7) (3,2) (3,4)}

Step 5: dequeue − 8.

Step 6: dequeue − 7

Step 7: dequeue − 2 add adjacent of 2

T = {(0,1) (0,3) (0,8) (1,7) (3,2) (3,4) (2,5)}

Step 8: dequeue − 4

Step 9: dequeue − 5 add adjacent of 5

T = {(0,1), (0,3) (0,8) (1,7) (3,2) (3,4) (2,5) (5,6)}

Step 10: dequeue − 6

Q is empty. Stop

φ

Review Questions

1. Slide-2-15 every slide has unique concepts. Need to go over all the slides for the definition, properties and terminologies.
2. Able to know how to represent a graph adjacency using adjacency matrix and adjancency List. When to use each.
3. What is sparce and dense graph.
4. Able to know the working principles of BFS and DFS.
5. Uses of stack and Queue graph searching algorithms.