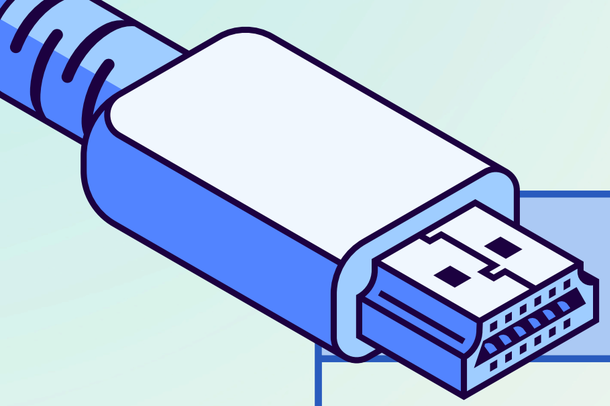
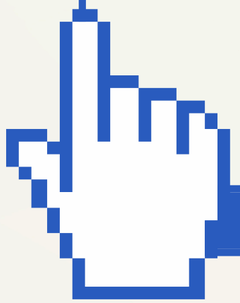


CASE STUDY

DATA STRUCTURES

TEACHER INCHARGE: MSNV JITENDRA



TEAM MEMBERS

KOUSHIK VULLI - 2023001661

K JAYA SAI DEEP - 2023000196

M VASU VARMA-2023001083

J SANJAY - 2023000715

Topic : Social Network Graph for Influencer Detection

Executive Summary

This case study explores the use of graph representations to analyze user connections within a social media platform. The primary goal is to identify influencers users with the most connections using Breadth
First Search (BFS) and Depth-First Search (DFS) techniques. The findings demonstrate that BFS is more effective for identifying influencers, while DFS excels at discovering user communities. Additionally, the report compares the performance of adjacency lists versus adjacency matrices in terms of efficiency and scalability.

TASKS TO PERFORM :

- Model the social network as an undirected graph where nodes represent
users and edges represent friendships.
- Implement Breadth-First Search (BFS) to find the most connected node (user with the most direct connections).
- Implement Depth-First Search (DFS) to traverse the graph and identify
clusters (communities) of users.
- Compare BFS and DFS in terms of their use cases and efficiency for finding influencers and communities.
- Discuss how the graph representation
(adjacency list vs adjacency matrix) impacts performance.

CASE DESCRIPTION

Graph Representation The social network is represented as an undirected graph:

Nodes : Represent users.
Edges : Represent friendships between users.

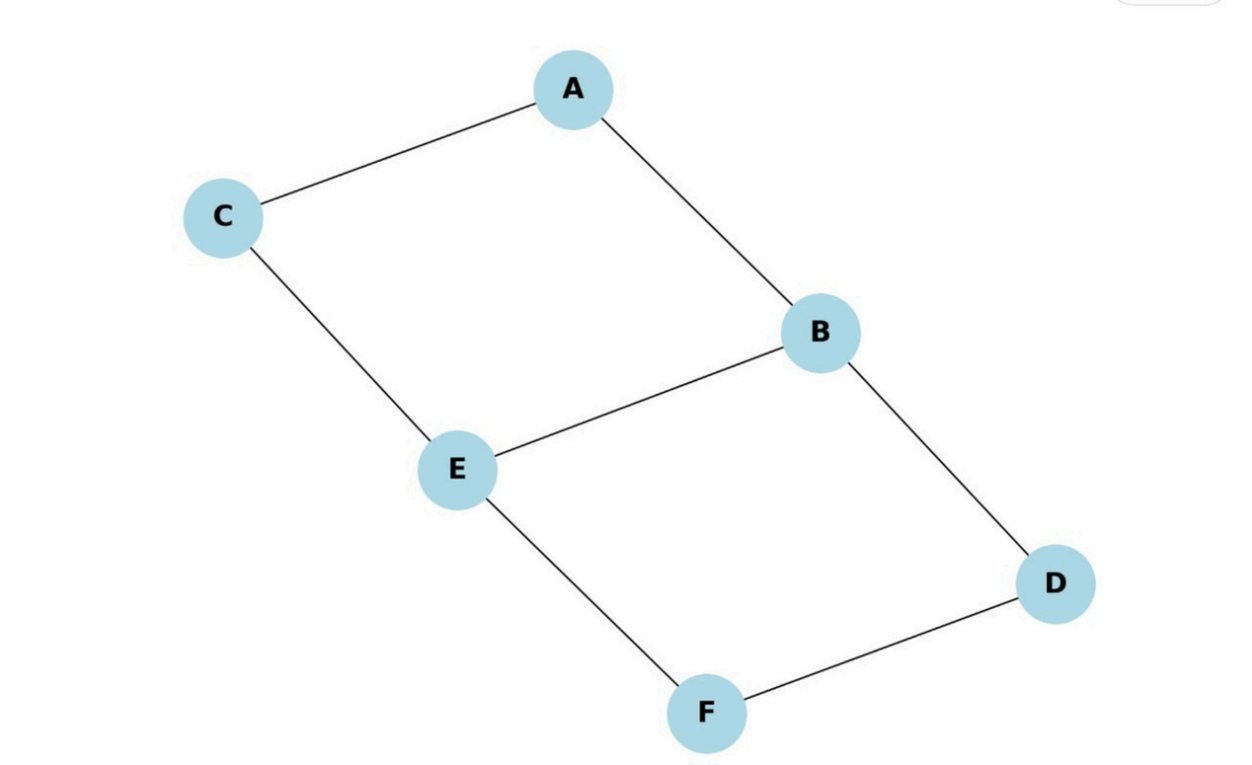
An adjacency list is used for efficient space utilization, especially when the graph is sparse.

This setup allows for the analysis of direct connections and community detection

SCENARIO :

A social media platform wants to analyze its user connections to identify influencers who have the most connections and reach. Users are represented as nodes, and friendships as edges between them.

Graph Visualization:



- Nodes: Each node (A, B, C, D, E, F) represents a
  user on the social media platform.
- Edges: The lines connecting the nodes represent friendships between users.

For example:

- User A is connected to Users B and C.
- User B is connected to Users A, D, and E.
- User C is connected to Users A and E, and so on.

Count Connections:

For each user, count the number of direct connections

- User A: 2 connections (B, C)
- User B: 3 connections (A, D, E)
- User C: 2 connections (A, E)
- User D: 2 connections (B, F)
- User E: 3 connections (B, C, F)
- User F: 2 connections (D, E)

Identify Influencers:

Users B and E, with the highest number of connections (3), can be considered the influencers in this scenario.

DFS - DEPTH FIRST SEARCH

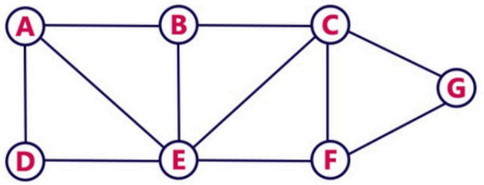
Depth-First Search (DFS) is a graph traversal algorithm that explores as far down a branch as possible before backtracking, systematically visiting all nodes in a graph or tree structure.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.
Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
Step 5 - Repeat steps 3 and 4 until queue becomes empty.
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

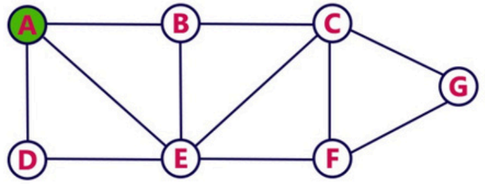
Example

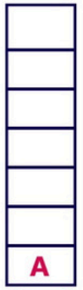
Consider the following example graph to perform DFS traversal



Step 1:

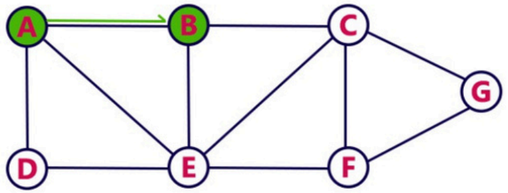
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.





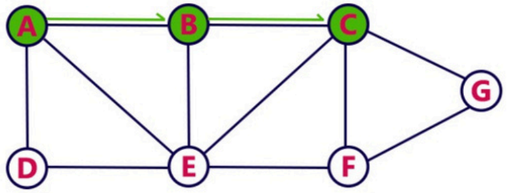
Step 2:

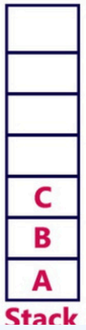
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

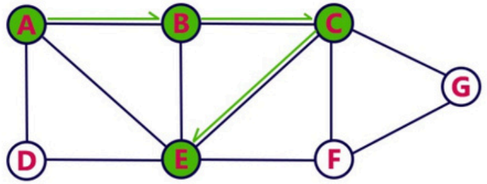
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





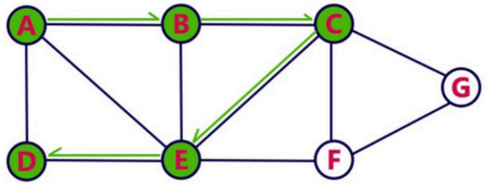
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



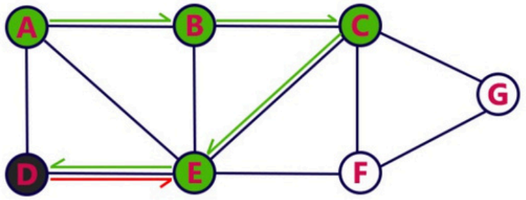
Step 5:

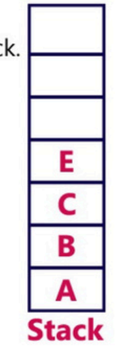
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

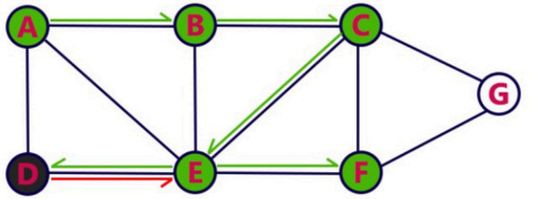
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

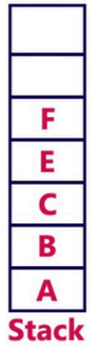




Step 7:

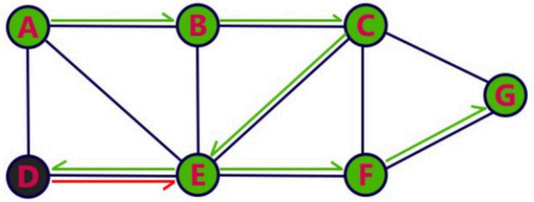
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





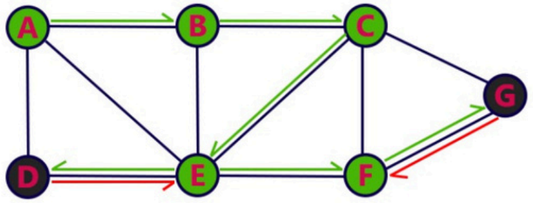
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



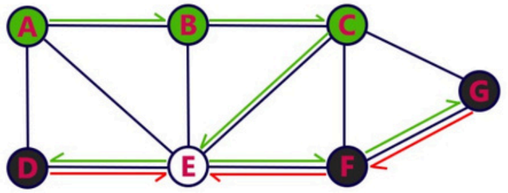
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



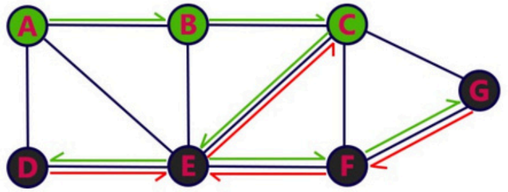
Step 10:

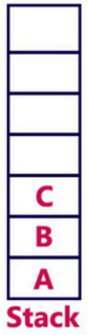
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

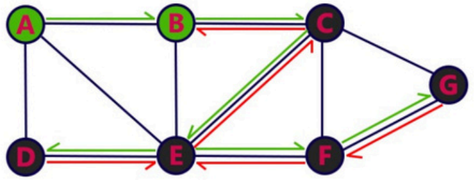
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

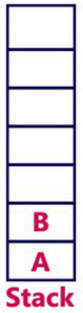




Step 12:

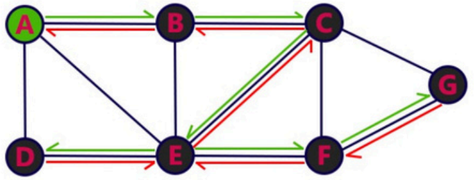
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





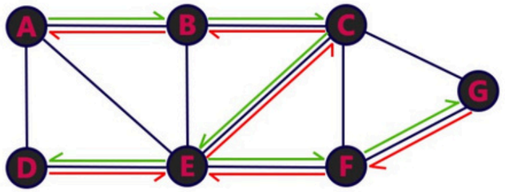
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

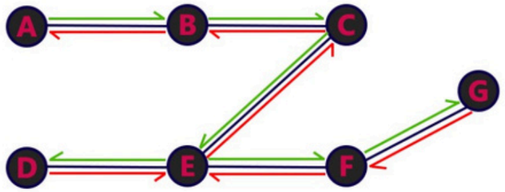


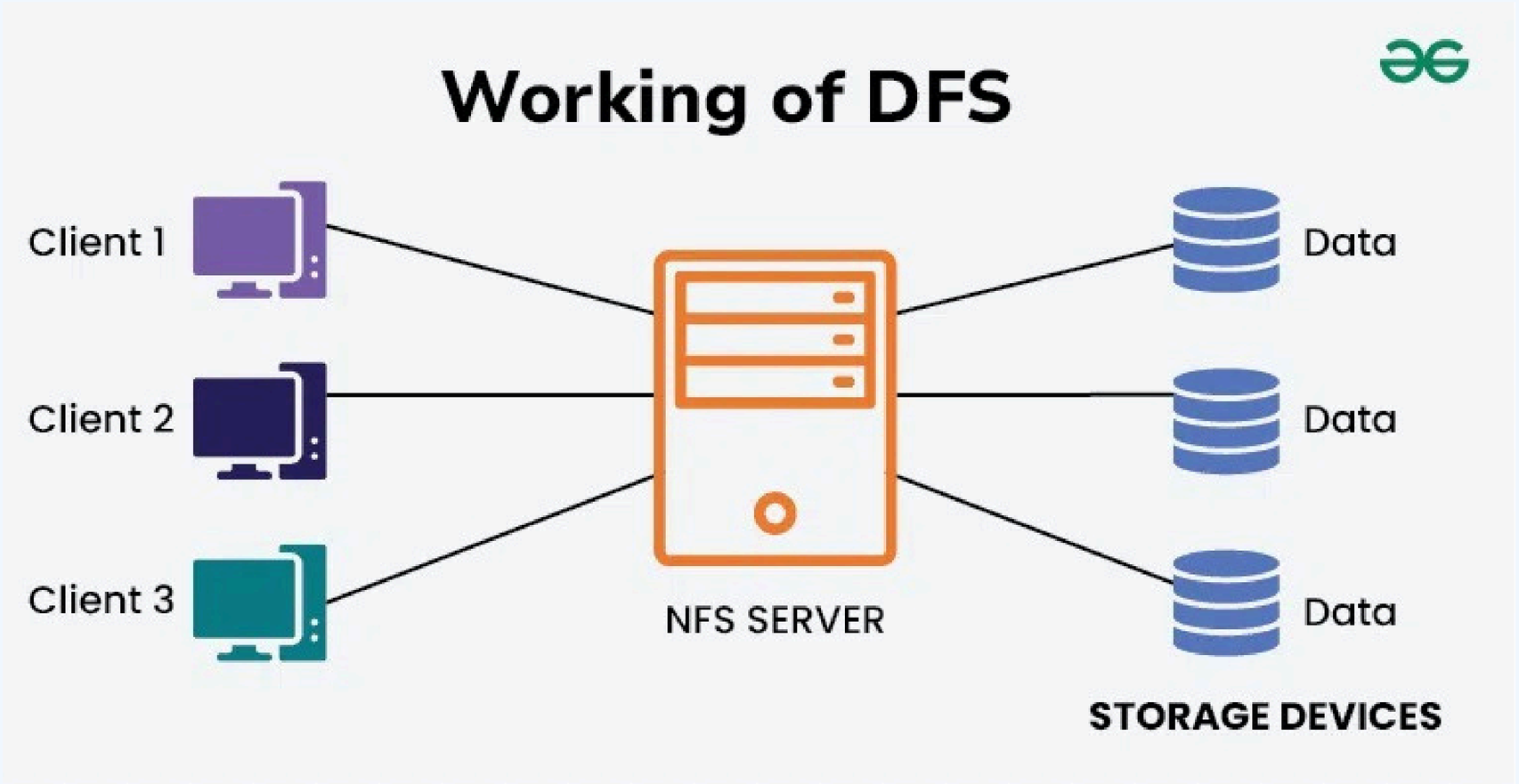
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.





WORKING OF DFS

BFS - BREADTH FIRST SEARCH

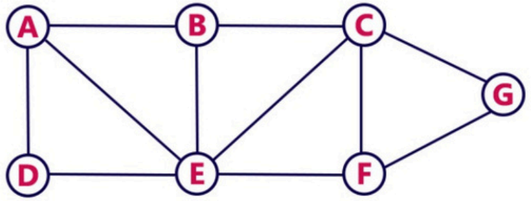
Breadth-First Search (BFS) is a graph traversal algorithm that explores all neighbors of a node before moving on to the neighbors of those neighbors, systematically visiting nodes level by level.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.
Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
Step 5 - Repeat steps 3 and 4 until queue becomes empty.
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

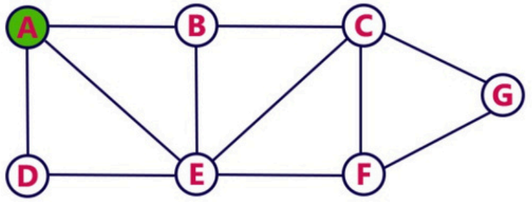
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

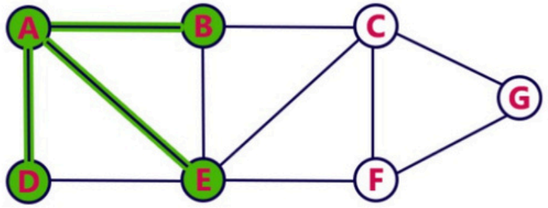


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

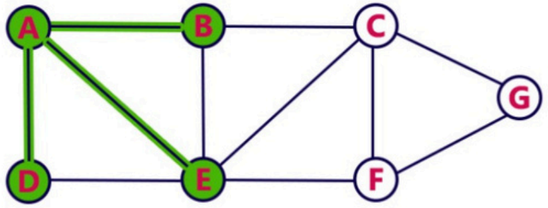


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

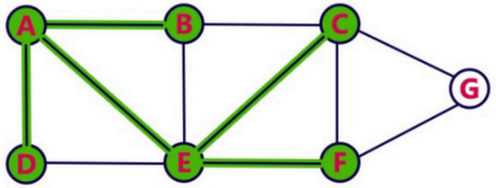


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

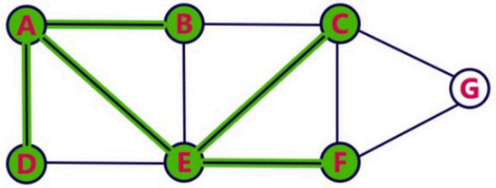


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

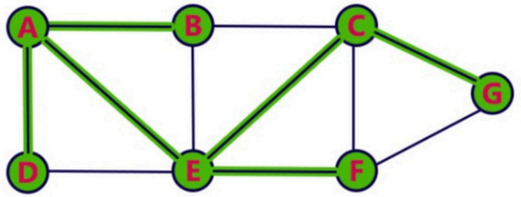


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

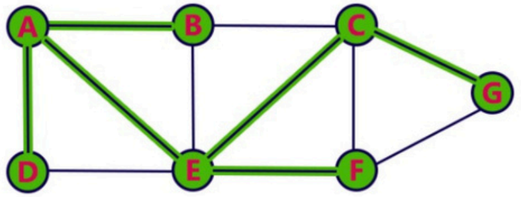


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

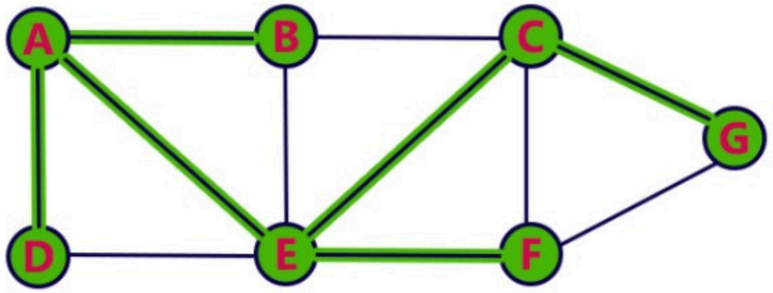


Queue

						G

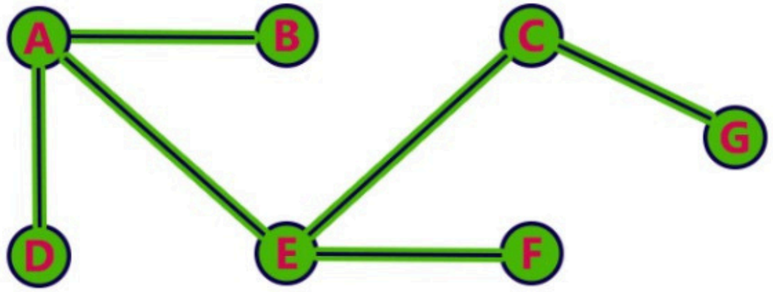
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



CODE ;

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#define MAX_USERS 100

typedef struct Node {
    int user;
    struct Node* next;
} Node;

Node* adjList[MAX_USERS];
bool visited[MAX_USERS] = { false };

// Function to add an undirected edge between two users
void addEdge(int u, int v) {
    Node* newNode = malloc(sizeof(Node));
    newNode->user = v;
    newNode->next = adjList[u];
    adjList[u] = newNode;

    newNode = malloc(sizeof(Node));
    newNode->user = u;
    newNode->next = adjList[v];
    adjList[v] = newNode;
}
```

```c
// Function to find the user with the most direct connections
int findMostConnected(int n) {
    int maxConnections = 0, influencer = 0;
    for (int i = 0; i < n; i++) {
        int connections = 0;
        Node* curr = adjList[i];
        while (curr != NULL) {
            connections++;
            curr = curr->next;
        }
        if (connections > maxConnections) {
            maxConnections = connections;
            influencer = i;
        }
    }
    return influencer;
}

// Depth-First Search to traverse and identify clusters
void DFS(int user) {
    visited[user] = true;
    printf("%d ", user);

    Node* temp = adjList[user];
    while (temp) {
        if (!visited[temp->user]) {
            DFS(temp->user);
```

} C Ln

```
    Node* temp = adjList[user];
    while (temp) {
        if (!visited[temp->user]) {
            DFS(temp->user);
        }
        temp = temp->next;
    }
}

// Function to find and print all clusters (communities) of users
void findClusters(int n) {
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            printf("Cluster: ");
            DFS(i);
            printf("\n");
        }
    }
}
```

```
int main() {
    // Example graph with 5 users (0 to 4) and some friendships
    int n = 5;
    addEdge(0, 1);
    addEdge(1, 2);
    addEdge(2, 3);
    addEdge(3, 4);
    addEdge(1, 4);

    int influencer = findMostConnected(n);
    printf("User with the most connections is: %d\n", influencer);

    // Reset visited array for DFS
    for (int i = 0; i < n; i++) visited[i] = false;
    printf("Communities in the network:\n");
    findClusters(n);

    return 0;
}
```

OUTPUT :

Run

```
User with the most connections is: 1
Communities in the network:
Cluster: 0 1 4 3 2
```

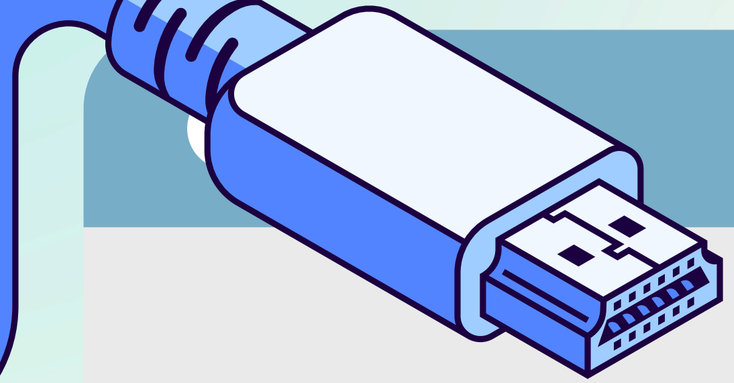
COMPARISON OF BFS AND DFS

Aspect	Breadth-First Search (BFS)	Depth-First Search (DFS)
Traversal Method	Explores all neighbors at the current depth before moving deeper.	Explores as far down a branch as possible before backtracking.
Use Cases	- Finding the shortest path in unweighted graphs. - Identifying connected components in a level-wise manner. - Suitable for finding influencers based on direct connections.	- Detecting cycles in graphs. - Topological sorting. - Identifying communities through deep exploration of connections.
Finding Influencers	Effective for identifying the most connected user due to level-wise exploration of direct connections.	Less effective for influencer detection, as it may prioritize deeper nodes over direct connections.
Finding Communities	Can identify clusters but may not reveal all connections deeply.	Effective for community detection, revealing clusters by exploring all connections exhaustively.
Time Complexity	$O(V + E)$	$O(V + E)$
Space Complexity	Higher due to storing multiple levels in a queue ($O(V)$).	Lower in some cases but can reach $O(V)$ in deep recursions due to the call stack.

IMPACT OF GRAPH REPRESENTATION ON PERFORMANCE

Aspect	Adjacency List	Adjacency Matrix
Space Complexity	$O(V + E)$ — Efficient for sparse graphs, only stores existing edges.	$O(V^2)$ — Inefficient for sparse graphs, stores all possible edges.
Time Complexity	- Adding Edges: $O(1)$ - Traversing Neighbors: $O(k)$ - Checking Edge Existence: $O(k)$	- Adding Edges: $O(1)$ - Traversing Neighbors: $O(V)$ - Checking Edge Existence: $O(1)$
Traversal Efficiency	More efficient for BFS and DFS in sparse graphs.	Slower in sparse graphs, requires iterating over all vertices.
Use Cases	Best for sparse, dynamic graphs (e.g., social networks).	Suitable for dense graphs, quick edge existence checks.

CONCLUSION



- This case study examined the use of graph theory to analyze user connections on a social media platform for identifying influencers and communities. By modeling the social network as an undirectedgraph, we implemented Breadth-First Search (BFS) to effectively pinpoint the most connected users, demonstrating its efficiencyin recognizing influencers. Conversely, Depth-First Search (DFS) was utilized to uncover user clusters, revealing important community structures within the network.
- The analysis highlighted that BFS is ideal for identifying influencers due to its breadth-first approach, while DFS excels in community detection. Additionally, the choice between an adjacencylist and an adjacency matrix significantly affects performance, with the adjacency list being more efficient for sparse graphs typical in social networks. Overall, these findings emphasize the value of graph representations and traversal algorithms in enhancing user engagement strategies on social media platforms.

REFERENCES

B.TECH SMARTCLASS

https://www.btechsmartclass.com/data_structures/insertion-sort.html

GEEKSFORGEEKS

https://www.geeksforgeeks.org/

W3 SCHOOLS

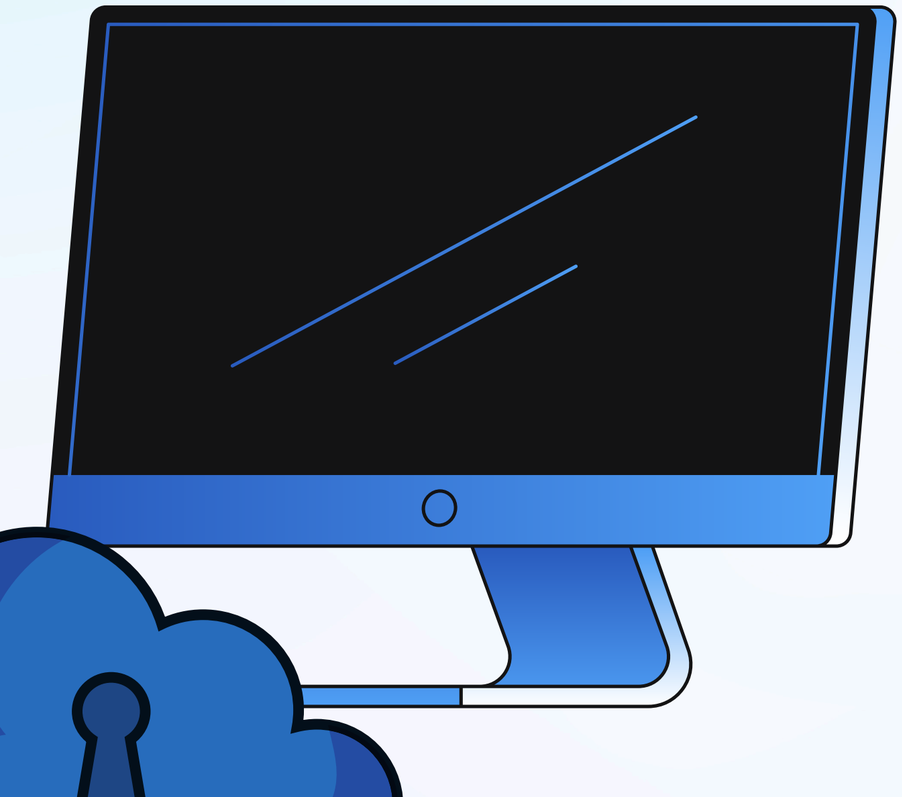
www.w3schools.com/dsa

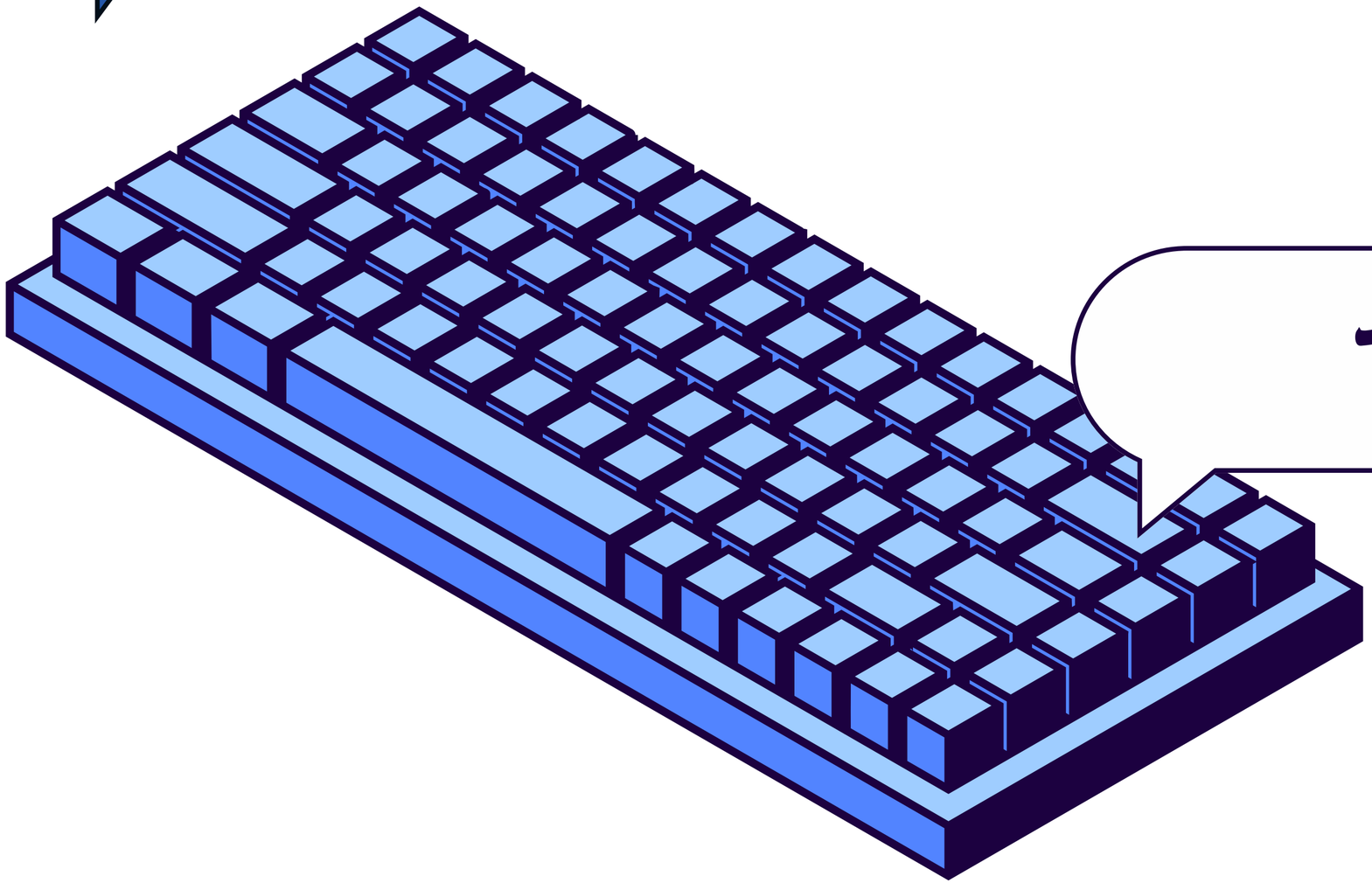
CHAT GPT

https://openai.com/index/chatgpt/

YOUTUBE

https://www.youtube.com/





Thank You!!!