**Exp No. 7**

## Title:

Implement a program to represent a graph using adjacency list or adjacency matrix data structure and perform BFS and DFS traversal algorithms.

## Aim:

To represent a graph using adjacency matrix and adjacency list structures and to traverse the graph using BFS and DFS algorithms.

## Objectives:

• Understand graph representation using adjacency matrix and adjacency list.  
• Implement BFS and DFS traversals using queue and recursion respectively.  
• Learn to handle disconnected graphs.  
• Explore traversal order differences between BFS and DFS.

## Theory:

A graph consists of a finite set of vertices (nodes) and edges connecting them. Graphs can be represented using:  
  
1. Adjacency Matrix: A 2D array `G[V][V]` where G[i][j] = 1 if there is an edge between vertex i and j, else 0.  
 - Suitable for dense graphs.  
 - Simple but consumes more space.  
  
2. Adjacency List: Each vertex maintains a list of its adjacent vertices.  
 - Efficient for sparse graphs.  
 - Saves memory.  
  
Traversal Techniques:  
• Breadth-First Search (BFS): Level-wise traversal using a queue.  
• Depth-First Search (DFS): Recursive or stack-based traversal exploring as deep as possible before backtracking.

## Algorithm:

Breadth-First Search (BFS):

1. Start from a source vertex.  
2. Mark it visited and enqueue it.  
3. While queue is not empty:  
 - Dequeue vertex v.  
 - For each unvisited adjacent vertex u of v, mark visited and enqueue u.  
4. Continue until all vertices are visited.

Depth-First Search (DFS):

1. Start from a source vertex.  
2. Mark it visited.  
3. Recursively visit all its unvisited adjacent vertices.  
4. Continue until all vertices are visited.

## Program Code (C Language): Graph Representation and Traversal

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 20

// --- Adjacency List Node Structure ---

typedef struct AdjListNode {

    int dest;

    struct AdjListNode \*next;

} AdjListNode;

// --- Adjacency List Header (Graph) Structure ---

typedef struct Graph {

    int V; // Number of vertices

    AdjListNode \*array[MAX\_VERTICES];

    bool visited[MAX\_VERTICES];

} Graph;

// --- Queue for BFS ---

int queue[MAX\_VERTICES];

int front = -1, rear = -1;

void enqueue(int item) {

    if (rear == MAX\_VERTICES - 1) {

        // Queue is full (should not happen if MAX\_VERTICES is large enough)

        return;

    }

    if (front == -1) front = 0;

    queue[++rear] = item;

}

int dequeue() {

    if (front == -1 || front > rear) {

        // Queue is empty

        front = rear = -1;

        return -1;

    }

    return queue[front++];

}

bool is\_queue\_empty() {

    return (front == -1 || front > rear);

}

// --- Graph ADT Functions ---

// Function to create a new adjacency list node

AdjListNode\* new\_adj\_list\_node(int dest) {

    AdjListNode \*new\_node = (AdjListNode \*)malloc(sizeof(AdjListNode));

    if (new\_node == NULL) exit(EXIT\_FAILURE);

    new\_node->dest = dest;

    new\_node->next = NULL;

    return new\_node;

}

// Function to create a graph with V vertices

Graph\* create\_graph(int V) {

    Graph \*graph = (Graph \*)malloc(sizeof(Graph));

    if (graph == NULL) exit(EXIT\_FAILURE);

    graph->V = V;

    // Initialize array and visited flags

    for (int i = 0; i < V; i++) {

        graph->array[i] = NULL;

        graph->visited[i] = false;

    }

    return graph;

}

// Function to add an edge to the graph

void add\_edge(Graph \*graph, int src, int dest, bool directed) {

    // Add edge from src to dest

    AdjListNode \*new\_node = new\_adj\_list\_node(dest);

    new\_node->next = graph->array[src];

    graph->array[src] = new\_node;

    if (!directed) {

        // For undirected graph, add an edge from dest to src as well

        new\_node = new\_adj\_list\_node(src);

        new\_node->next = graph->array[dest];

        graph->array[dest] = new\_node;

    }

}

// --- Traversal Algorithms ---

// Breadth-First Search (BFS)

void BFS(Graph \*graph, int start\_vertex) {

    // Reset visited flags for all nodes

    for (int i = 0; i < graph->V; i++) {

        graph->visited[i] = false;

    }

    front = rear = -1; // Reset queue

    printf("\nBFS Traversal (Start %d): ", start\_vertex);

    graph->visited[start\_vertex] = true;

    enqueue(start\_vertex);

    while (!is\_queue\_empty()) {

        int v = dequeue();

        printf("%d ", v);

        // Get all adjacent vertices of v

        AdjListNode \*current = graph->array[v];

        while (current != NULL) {

            int u = current->dest;

            if (!graph->visited[u]) {

                graph->visited[u] = true;

                enqueue(u);

            }

            current = current->next;

        }

    }

    printf("\n");

}

// Depth-First Search (DFS) (Recursive)

void DFS\_util(Graph \*graph, int v) {

    // Mark the current node as visited and print it

    graph->visited[v] = true;

    printf("%d ", v);

    // Recur for all the vertices adjacent to this vertex

    AdjListNode \*current = graph->array[v];

    while (current != NULL) {

        int u = current->dest;

        if (!graph->visited[u]) {

            DFS\_util(graph, u);

        }

        current = current->next;

    }

}

void DFS(Graph \*graph, int start\_vertex) {

    // Reset visited flags for all nodes

    for (int i = 0; i < graph->V; i++) {

        graph->visited[i] = false;

    }

    printf("DFS Traversal (Start %d): ", start\_vertex);

    DFS\_util(graph, start\_vertex);

    printf("\n");

}

// Function to free graph memory

void free\_graph(Graph \*graph) {

    for (int i = 0; i < graph->V; i++) {

        AdjListNode \*current = graph->array[i];

        AdjListNode \*next;

        while (current != NULL) {

            next = current->next;

            free(current);

            current = next;

        }

    }

    free(graph);

}

int main() {

    int V, E, directed\_flag, u, v, start\_v;

    printf("--- Graph Traversal (Adjacency List) ---\n");

    printf("Enter number of vertices: ");

    scanf("%d", &V);

    Graph \*graph = create\_graph(V);

    printf("Enter number of edges: ");

    scanf("%d", &E);

    printf("Is graph directed? (1 for Yes, 0 for No/Undirected): ");

    scanf("%d", &directed\_flag);

    bool directed = (directed\_flag == 1);

    printf("Enter edges (source destination):\n");

    for (int i = 0; i < E; i++) {

        scanf("%d %d", &u, &v);

        if (u >= 0 && u < V && v >= 0 && v < V) {

            add\_edge(graph, u, v, directed);

        } else {

            printf("Invalid vertex entered. Skipping edge.\n");

            i--;

        }

    }

    printf("Enter start vertex for traversal: ");

    scanf("%d", &start\_v);

    // Perform traversals

    if (start\_v >= 0 && start\_v < V) {

        BFS(graph, start\_v);

        DFS(graph, start\_v);

    } else {

        printf("Invalid start vertex.\n");

    }

    // Clean up memory

    free\_graph(graph);

    return 0;

}

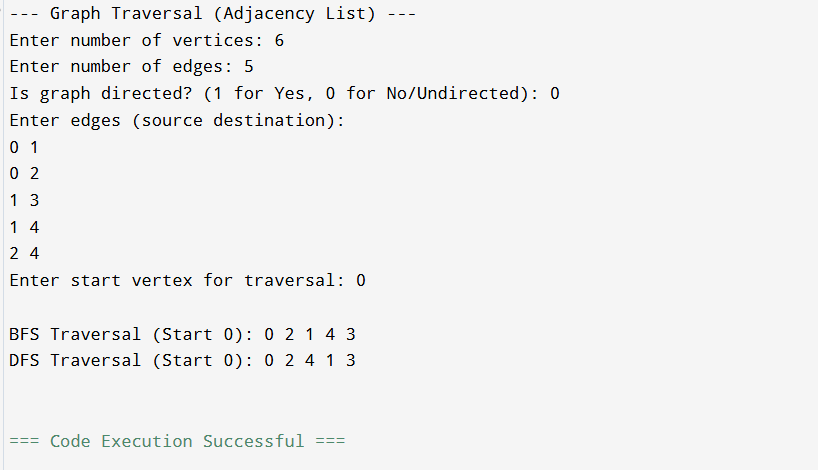
## Sample Input/Output:

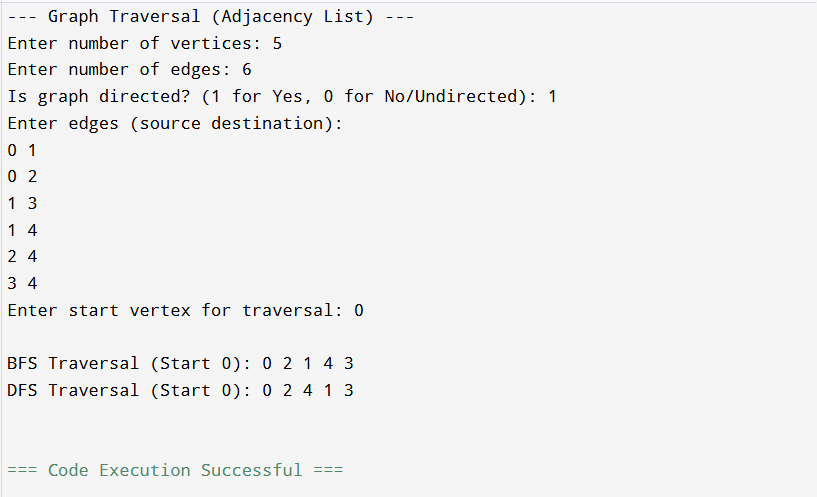
Input:  
Enter number of vertices: 5  
Enter number of edges: 6  
Is graph directed? 0  
Enter edges:  
0 1  
0 2  
1 3  
1 4  
2 4  
3 4  
Enter start vertex: 0  
  
Output:  
BFS Traversal: 0 1 2 3 4  
DFS Traversal: 0 1 3 4 2

## Result:

Successfully represented a graph using adjacency matrix and performed BFS and DFS traversals.

Program Output:





## Conclusion:

Graph traversal techniques enable systematic exploration of nodes. BFS ensures level-wise traversal using a queue, while DFS explores deeper paths recursively. Adjacency matrix is suitable for dense graphs, while adjacency list is memory-efficient for sparse graphs.

## Post-Lab Problem (Easy): Connected Components in an Undirected Graph

Objective:

Modify the DFS program to count and print all connected components in an undirected graph.

Description:

Traverse all vertices; if a vertex is unvisited, perform DFS and increment the component count.

Sample Output:

Connected Components: 2  
Component 1: 0 1 2  
Component 2: 3 4

Hint:

Reset visited[] after each DFS call and count the total number of components.

Post Lab Program Code:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 20

// --- Adjacency List Node Structure (Reused) ---

typedef struct AdjListNode {

    int dest;

    struct AdjListNode \*next;

} AdjListNode;

// --- Graph Structure (Reused) ---

typedef struct Graph {

    int V;

    AdjListNode \*array[MAX\_VERTICES];

    bool visited[MAX\_VERTICES];

} Graph;

// --- Graph ADT Functions (Reused) ---

AdjListNode\* new\_adj\_list\_node(int dest) {

    AdjListNode \*new\_node = (AdjListNode \*)malloc(sizeof(AdjListNode));

    if (new\_node == NULL) exit(EXIT\_FAILURE);

    new\_node->dest = dest;

    new\_node->next = NULL;

    return new\_node;

}

Graph\* create\_graph(int V) {

    Graph \*graph = (Graph \*)malloc(sizeof(Graph));

    if (graph == NULL) exit(EXIT\_FAILURE);

    graph->V = V;

    for (int i = 0; i < V; i++) {

        graph->array[i] = NULL;

        graph->visited[i] = false;

    }

    return graph;

}

void add\_edge\_undirected(Graph \*graph, int src, int dest) {

    // src to dest

    AdjListNode \*new\_node = new\_adj\_list\_node(dest);

    new\_node->next = graph->array[src];

    graph->array[src] = new\_node;

    // dest to src (undirected)

    new\_node = new\_adj\_list\_node(src);

    new\_node->next = graph->array[dest];

    graph->array[dest] = new\_node;

}

// --- Post-Lab DFS Function ---

// DFS utility function that prints the component

void DFS\_component(Graph \*graph, int v) {

    graph->visited[v] = true;

    printf("%d ", v);

    AdjListNode \*current = graph->array[v];

    while (current != NULL) {

        int u = current->dest;

        if (!graph->visited[u]) {

            DFS\_component(graph, u);

        }

        current = current->next;

    }

}

// Main function to find and print connected components

void find\_connected\_components(Graph \*graph) {

    int component\_count = 0;

    // Ensure all visited flags are reset initially

    for (int i = 0; i < graph->V; i++) {

        graph->visited[i] = false;

    }

    for (int i = 0; i < graph->V; i++) {

        if (!graph->visited[i]) {

            component\_count++;

            printf("Component %d: ", component\_count);

            // Start DFS from unvisited vertex i

            DFS\_component(graph, i);

            printf("\n");

        }

    }

    printf("\nConnected Components: %d\n", component\_count);

}

// Function to free graph memory (Reused)

void free\_graph(Graph \*graph) {

    for (int i = 0; i < graph->V; i++) {

        AdjListNode \*current = graph->array[i];

        AdjListNode \*next;

        while (current != NULL) {

            next = current->next;

            free(current);

            current = next;

        }

    }

    free(graph);

}

int main() {

    int V, E, u, v;

    printf("--- Post-Lab: Connected Components in Undirected Graph ---\n");

    printf("Enter number of vertices: ");

    scanf("%d", &V);

    Graph \*graph = create\_graph(V);

    printf("Enter number of edges: ");

    scanf("%d", &E);

    printf("Enter edges (source destination):\n");

    // Example for 2 components: 5 vertices, 2 edges (0 1, 3 4)

    // Vertices: 0, 1, 2, 3, 4

    // Edges: (0, 1), (3, 4). Vertex 2 is isolated.

    // Components: {0, 1}, {2}, {3, 4} -> Total 3 components.

    // Example for 2 components: 5 vertices, 3 edges (0 1, 1 2, 3 4)

    // Components: {0, 1, 2}, {3, 4} -> Total 2 components.

    for (int i = 0; i < E; i++) {

        scanf("%d %d", &u, &v);

        if (u >= 0 && u < V && v >= 0 && v < V) {

            add\_edge\_undirected(graph, u, v);

        } else {

            printf("Invalid vertex entered. Skipping edge.\n");

            i--;

        }

    }

    find\_connected\_components(graph);

    // Clean up memory

    free\_graph(graph);

    return 0;

}

Post Lab Output:

