NAME	Sujal Dingakar
UID no	2024301005
Experiment No	8

AIM:	Implement Breadth Frst Search and Depth First Search Traversal for given Graph. Graph should be dynamic , means it should accept number of vertices and edges , dynamically. (It should not be fixed in program) Submission status
THEORY:	**Why Do We Use Graphs?** Graphs are used to represent complex relationships and structures that are not easily handled by simpler data structures. They are particularly useful in scenarios involving connections and interactions. Here are some key reasons for using graphs: • **Modeling Relationships**: Graphs can represent various relationships, such as social networks (people and their connections), transportation networks (cities and routes), and dependency graphs (tasks and their dependencies). • **Dynamic Data Handling**: Graphs excel in scenarios where data is dynamic and frequently changing. Adding or removing connections can be done efficiently, making them suitable for applications like social media platforms. • **Pathfinding and Navigation**: Graphs are used in algorithms for finding the shortest paths, such as Dijkstra's algorithm, making them essential in mapping and navigation applications. **What is a Graph?** A **graph** is a data structure that consists of a set of vertices (or nodes) connected by edges (or arcs). Graphs can represent various real-world structures and relationships, making them incredibly versatile in computer science and mathematics. **Key Characteristics of Graphs**: • **Vertices**: The fundamental units in a graph, representing entities (e.g., cities in a transportation network). • **Edges**: Connections between pairs of vertices, which can represent relationships (e.g., roads between cities). • **Directed and Undirected Graphs**: In directed graphs, edges have a direction (e.g., one-way streets), while in undirected graphs, they do not (e.g., two-way streets). • **Weighted and Unweighted Graphs**: In weighted graphs, edges have associated weights (e.g., distances or costs), while unweighted graphs treat all edges equally. • **Cyclic and Acyclic Graphs**: A graph is cyclic if it contains at least one cycle (a path that starts and ends at the same vertex); otherwise, it is acyclic.

Operations that Can Be Performed on Graphs :-
Graphs support various operations, including:

- **Traversal**:
    - **Breadth-First Search (BFS)**: Explores neighbors before going deeper into the graph, useful for finding the shortest path in unweighted graphs.
    - **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking, useful for exploring all possible paths.
- **Pathfinding**: Algorithms like Dijkstra's and A* help find the shortest or most efficient path between two vertices in a graph.
- **Cycle Detection**: Algorithms exist to determine if a graph contains cycles, which is crucial in many applications.
- **Minimum Spanning Tree**: Algorithms like Kruskal's and Prim's help find the minimum spanning tree of a graph, connecting all vertices with the least total edge weight.

Applications of Graphs :-
Graphs are widely used across various fields and applications:

1. **Social Networks**: Represent relationships among users, enabling functionalities like friend suggestions and connection recommendations.
2. **Transportation Networks**: Model cities, roads, and routes, allowing for navigation and route optimization in GPS systems.
3. **Web Page Linking**: The structure of the internet can be modeled as a graph, with web pages as vertices and hyperlinks as edges, facilitating search engines' crawling and indexing.
4. **Network Routing**: Used in telecommunications and computer networks to manage the data flow between different nodes, optimizing routes for data packets.
5. **Dependency Resolution**: In software development, graphs can represent dependencies between modules, helping with tasks like build automation and package management.
6. **Recommendation Systems**: Used in e-commerce and streaming services to suggest products or content based on user connections and preferences.

What graph does over other dataStructure.?
Graphs are unique among data structures because they excel at representing complex relationships and interactions that are not easily modeled with simpler structures like arrays, linked lists, or trees.

Graphs offer significant advantages over other data structures due to their ability to represent complex relationships and dynamic connectivity. Unlike trees or arrays, graphs can model non-hierarchical data with multiple pathways between nodes, making them ideal for applications like social networks and transportation systems. They efficiently handle sparse data and support various algorithms for tasks such as pathfinding and cycle detection.

	Overall, graphs are essential for scenarios requiring intricate interconnections and interactions that simpler structures cannot effectively manage.
ALGORITHM:	**Graph Representation:** • Define a Node structure to represent each vertex's adjacency list. • Define a List structure to represent the linked list for each vertex. • Define a Graph structure that contains the number of vertices and an array of adjacency lists. **Queue Data Structure:** • Define a Queue structure for use in BFS traversal, including methods to enqueue and dequeue elements. **Graph Initialization:** • Create a graph using the createGraph function, which allocates memory for the graph and initializes the adjacency lists. **Adding Edges:** • Define the addEdge function to add directed edges between vertices. If you want an undirected graph, the same edge is added in both directions. **BFS Traversal:** • Initialize a queue to manage the BFS process. • Maintain an array to track the level of each vertex. • Use a visited array to keep track of visited vertices. • Start BFS from a specified vertex: o Dequeue a vertex, mark it as visited, and record its level. o Enqueue all unvisited adjacent vertices. • Print the BFS traversal order and levels of each vertex. **DFS Traversal:** • Initialize arrays to keep track of visited vertices, start times, and end times of each vertex. • Start DFS from a specified vertex: o Mark the vertex as visited, record its start time, and store it in the DFS traversal order. o Recursively visit all unvisited adjacent vertices. o Record the finish time for each vertex upon returning from the recursive call. • Print the DFS traversal order, start times, and end times for each vertex. **Input Handling:** • Get the number of vertices and edges from the user. • Read edges from the user to build the graph and adjacency matrix for BFS. • Ask the user for the starting vertex for both DFS and BFS. **Output:** • Print the adjacency list representation of the graph. • Print the BFS and DFS traversal orders and relevant timing information. **Memory Cleanup:** • Free allocated memory for the adjacency lists and the graph structure at the end of the program.

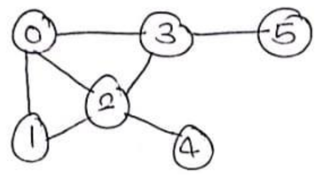
PROBLEM SOLVING:

1. BFS TRAVERSAL

Subject - Data Structure
Topic - Graph bfs & dfs implementation.
Name - Sujal Sandeep Dinganker
Date - 23-10-2024.

BFS Implementation :-

Step 1 : Initially we have a graph for BFS traversal



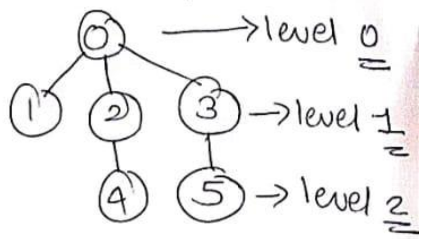
we represent the above graph in Adjacency Matrix Representation :-

There edge between 0 and 3 vertex. so it is undirected graph we mark edge in matrix from both direction 0→3 also and 3→0. like that we make matrix for remaining edge.

No of edges : (0,3) (0,2) (0,1) (3,5) (2,4) (1,2) (2,3)

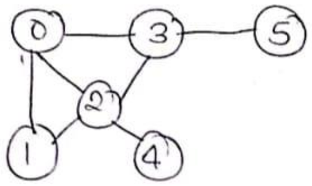
0	1	1	1	0	0
1	0	1	0	0	0
1	1	0	1	1	0
1	0	1	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0

start Vertex is 0



This matrix representation.

Step 1 :-



Visited

0					

queue

0					

↑ Front

push 0 into the queue and mark as a visited

Step 2 :- Remove 0 from the queue front and visit the unvisited neighbours of 0 and push into the queue.

Visited:

0	1	2	3	

queue

~~0~~	~~3~~	~~2~~	~~1~~	

↑ F

0	1	2	3	

↑ Front

print :- 0

Step 3 : Remove 1 from the queue front and visited the unvisited neighbours of 1 now 1 have no unvisited neighbours.

Visited :

0	1	2	3	

print : ⇒ 0, 1

queue :

0	1	2	3	

↑ Front

Step 4 :- Remove 2 from the queue front and visit the unvisited neighbours of 2 where 2 has 1 unvisited neighbours which is 4 push in queue

Visited :

0	1	2	3	4	

print :- 0, 1, 2

queue :

0	1	2	3	4	

Front point to now 2

Step 5 :- Remove 3 from the queue front and visit the unvisited neighbours of 3 where 3 has 1 unvisited neighbours which is 5. push into queue.

Visited : | ~~0~~ | ~~1~~ | ~~2~~ | ~~3~~ | 4 | 5 |

queue : | 0 | 1 | 2 | 3 | 4 | 5 |

↑ Front

print : 0,1,2,3

mark 3 has visited.

Step 6 :- Remove 4 from the queue front and visit the univisited neighbours of 4 where 4 has all node already visited, simply mark as visited and print it.

Visited : | ~~0~~ | ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | 5 |

queue : | 0 | 1 | 2 | 3 | 4 | 5 |

↑ Front

print :- 0,1,2,3,4

Step 7 :- Remove 5 from the queue front and visit the unvisited neighbours of 5 where 5 have no unvisited neighbours. simply mark and print it.

Visited : | ~~0~~ | ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | ~~5~~ |

queue : | 0 | 1 | 2 | 3 | 4 | 5 |

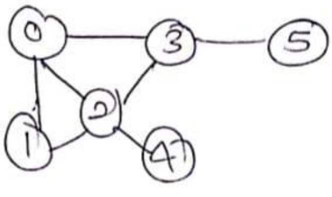
print :- 0,1,2,3,4,5

BFS Traversal :- 0,1,2,3,4,5

2. DFS TRAVERSAL

DFS Implementation :-

Step 1 :- Initially we represent Graph in adjacency list,,



↓

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

Nowa we traversed from vertex 0,, we use stack for backtracking the element.

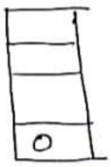
Initially we used a stack empty and boolean array to check visited or not.

bool | F | F | F | F | F | F |

↓

| T | F | F | F | F | F |

stack

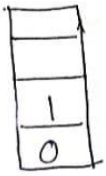


start from 0 mark as visited in boolean array as True then push in stack for backtrack

2> Now we have 3 choice to go 1, 2, 3 we explore first 1,

bool | T | T | F | F | F | F |

Stack



Mark 1 as visited.

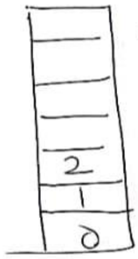
Now push 1 into stack for backtrack

3> Now we have to go explore 2 and mark as visited,

bool | T | T | T | F | F | F |

Mark 2 as visited,

Stack

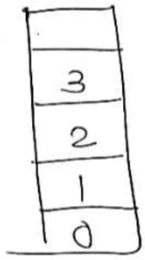


Now push 2 into stack for backtrack.

4> Now we explore node 3 and mark as visited.

bool | T | T | T | T | F | F |

Mark 3 as visited,



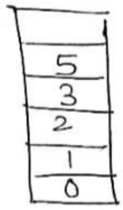
Now push 3 into stack for backtrack

5) Now we explore node 5 after 3 and mark 5 as a visited.

bool

T	T	T	T	T	T

Mark 5 as visited,



Now push 5 into stack for backtrack

6) Now we explore further from 5 but there is no node after 5 so now we backtrack using stack

bool

T	T	T	T	T	T

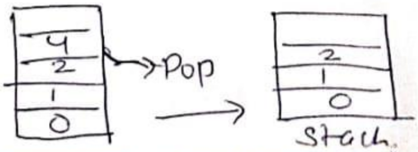
Stack Stack

- pop 5 from stack to go 3 backtrack
- pop 3 from stack to go 2 backtrack
- Now we check there is a Node 4 which is not explore so push into stack

7) Now we explore further from 4 but there is no node after 4 so we pop the element 4 to back track to 2,,

bool

T	T	T	T	T	T

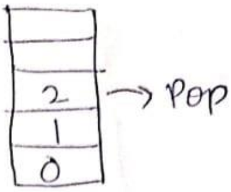


Stack.

8) Now we check Node 2 again but there is no unvisited node so pop 2 also from stack backtrack.

bool array

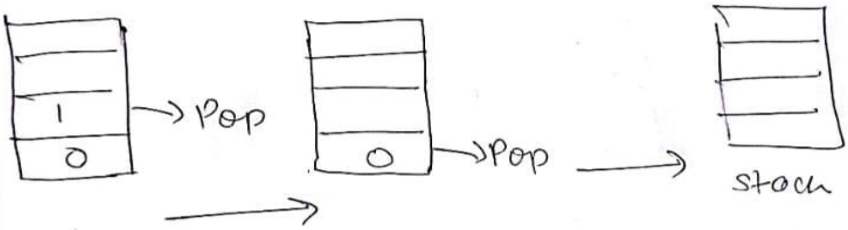
T	T	T	T	T	T



9) Now we check 1 as pop it then we check 0 and also pop it.

bool array

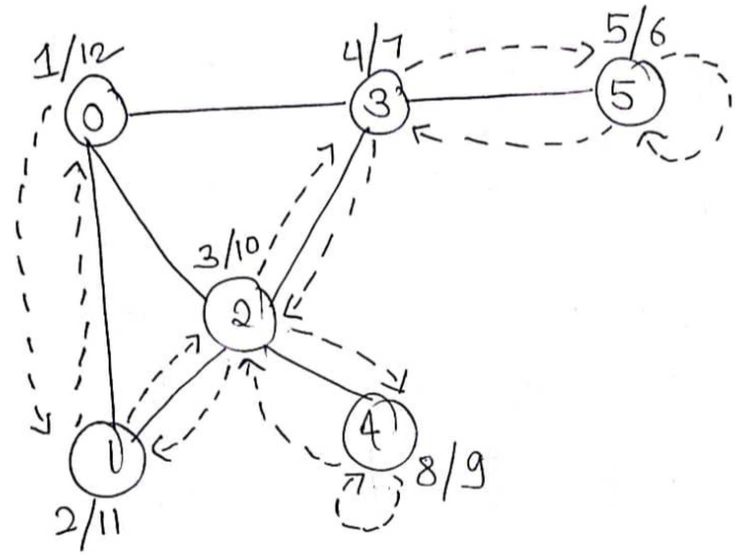
T	T	T	T	T	T



stack is empty.

Final DFS traversal → 0,1,2,3,5,4

DFS → Start and End Time Traversal Manual



Vertex	Start Time	End Time
0	1	12
1	2	11
2	3	10
3	4	7
4	8	9
5	5	6

PROGRAM:

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

#define MAX 100

int level[MAX]; // BFS levels
int dfsTraversal[MAX]; // To store DFS traversal order
int dfsIndex = 0; // To keep track of DFS traversal index

struct Node {
   int data;
   struct Node* next;
};
```

```
struct List {
    struct Node* head;
};

struct Graph {
    int vertices;
    struct List* array;
};

struct Queue {
    int size;
    int front;
    int rear;
    int *arr;
};

// Global time variable for DFS
int time = 0;

// Function to enqueue an element to the queue
void enqueue(struct Queue *q, int i) {
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear = (q->rear + 1) % q->size;
    q->arr[q->rear] = i;
    printf("Enqueued %d\n", i);
}

// Function to dequeue an element from the queue
int dequeue(struct Queue *q) {
    if (q->front == -1) {
        return -1;
    }
    int x = q->arr[q->front];
    printf("Dequeued element: %d\n", x);
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % q->size;
    }
    return x;
}

// BFS function
void bfs(int adj[MAX][MAX], int V, int s) {
    struct Queue q;
    q.size = MAX;
```

```
    q.front = -1;
    q.rear = -1;
    q.arr = (int *)malloc(q.size * sizeof(int));

    bool visited[MAX] = {false};
    int bfsTraversal[MAX];
    int index = 0;

    // Initialize levels
    for (int i = 0; i < MAX; i++) {
        level[i] = -1; // Set all levels to -1 initially
    }
    level[s] = 0; // Starting vertex is at level 0

    visited[s] = true;
    enqueue(&q, s);

    while (q.front != -1) {
        int curr = dequeue(&q);
        bfsTraversal[index++] = curr;

        // Print the current node and its level
        printf("Node: %d, Level: %d\n", curr, level[curr]);

        for (int j = 0; j < V; j++) {
            if (adj[curr][j] == 1 && !visited[j]) {
                visited[j] = true;
                level[j] = level[curr] + 1; // Set the level of the adjacent vertex
                enqueue(&q, j);
            }
        }
    }

    printf("\nBFS Traversal Array: ");
    for (int i = 0; i < index; i++) {
        printf("%d ", bfsTraversal[i]);
    }
    printf("\n");
    free(q.arr);
}

// DFS function with push/pop operations
void DFS(struct Graph* graph, int vertex, bool visited[], int startTime[], int
endTime[]) {
    printf("Push: %d\n", vertex); // Push operation
    visited[vertex] = true;
    startTime[vertex] = ++time;  // Record the start time

    dfsTraversal[dfsIndex++] = vertex; // Store the traversal order
```

```
    struct Node* currentNode = graph->array[vertex].head;
    while (currentNode) {
        int adjacentVertex = currentNode->data;
        if (!visited[adjacentVertex]) {
            DFS(graph, adjacentVertex, visited, startTime, endTime);
        }
        currentNode = currentNode->next;
    }

    endTime[vertex] = ++time;  // Record the finish time
    printf("Pop: %d\n", vertex); // Pop operation
}

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->vertices = vertices;
    graph->array = (struct List*)malloc(vertices * sizeof(struct List));

    for (int i = 0; i < vertices; i++) {
        graph->array[i].head = NULL;
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Uncomment the following code to make the graph undirected
    newNode = createNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Function to print the adjacency list of the graph
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->vertices; v++) {
        struct Node* temp = graph->array[v].head;
```

```
        printf("Adjacency list of vertex %d: ", v);
        while (temp) {
            printf("-> %d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

// Function to perform DFS traversal from a specified starting vertex
void DFSTraversal(struct Graph* graph, int startVertex) {
    bool* visited = (bool*)malloc(graph->vertices * sizeof(bool));
    int* startTime = (int*)malloc(graph->vertices * sizeof(int));
    int* endTime = (int*)malloc(graph->vertices * sizeof(int));

    for (int i = 0; i < graph->vertices; i++) {
        visited[i] = false;
        startTime[i] = 0;
        endTime[i] = 0;
    }

    printf("DFS traversal starting from vertex %d: \n", startVertex);
    DFS(graph, startVertex, visited, startTime, endTime);

    // Print the DFS traversal after push/pop operations
    printf("\n\nDFS Traversal Order: ");
    for (int i = 0; i < dfsIndex; i++) {
        printf("%d ", dfsTraversal[i]);
    }

    // Print the start and end times of all vertices
    printf("\n\nVertex\tStart Time\tEnd Time\n");
    for (int i = 0; i < graph->vertices; i++) {
        printf("%d\t\t%d\t\t%d\n", i, startTime[i], endTime[i]);
    }

    free(visited);
    free(startTime);
    free(endTime);
}

int main() {
    int V, edges, startVertex;

    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);
```

	`` // Create a graph for DFS struct Graph* graph = createGraph(V); // Input for both DFS and BFS int adj[MAX][MAX] = {0}; for (int i = 0; i < edges; i++) { int u, v; printf("Enter edge (u v): "); scanf("%d %d", &u, &v); // Add edge for DFS graph addEdge(graph, u, v); // Add edge for BFS adjacency matrix adj[u][v] = 1; adj[v][u] = 1; } printf("******************** DFS TRAVESAL RESULT ************************\n"); // Step 1: Print adjacency list representation for DFS printf("\nAdjacency List Representation for DFS Traversal:\n"); printGraph(graph); // Step 2: Ask for DFS starting vertex printf("\nDFS starting from which vertex? "); scanf("%d", &startVertex); // Step 3: Perform and print DFS traversal with push/pop DFSTraversal(graph, startVertex); // Step 4: Print adjacency matrix representation for BFS printf("******************** BFS TRAVESAL RESULT ************************\n"); printf("\nAdjacency Matrix Representation:\n"); for (int i = 0; i < V; i++) { for (int j = 0; j < V; j++) { printf("%d ", adj[i][j]); } printf("\n"); } // Step 5: Ask for BFS starting vertex printf("\nBFS starting from which vertex? "); scanf("%d", &startVertex); // Step 6: Perform BFS traversal and print operations printf("\nBFS starting from vertex %d:\n", startVertex); ```

	`` bfs(adj, V, startVertex); // Free memory used for DFS graph for (int i = 0; i < V; i++) { struct Node* current = graph->array[i].head; while (current) { struct Node* temp = current; current = current->next; free(temp); } } free(graph->array); free(graph); return 0; } ```
RESULT :-	1. DFS TRAVERSAL

```
Enter the number of vertices: 6
Enter the number of edges: 7
Enter edge (u v): 0
3
Enter edge (u v): 0
2
Enter edge (u v): 0
1
Enter edge (u v): 3
5
Enter edge (u v): 2
4
Enter edge (u v): 1
2
Enter edge (u v): 2
3
******************** DFS TRAVESAL RESULT ************************

Adjacency List Representation for DFS Traversal:
Adjacency list of vertex 0: -> 1 -> 2 -> 3
Adjacency list of vertex 1: -> 2 -> 0
Adjacency list of vertex 2: -> 3 -> 1 -> 4 -> 0
Adjacency list of vertex 3: -> 2 -> 5 -> 0
Adjacency list of vertex 4: -> 2
Adjacency list of vertex 5: -> 3
```

```
DFS starting from which vertex? 0
DFS traversal starting from vertex 0:
Push: 0
Push: 1
Push: 2
Push: 3
Push: 5
Pop: 5
Pop: 3
Push: 4
Pop: 4
Pop: 2
Pop: 1
Pop: 0


DFS Traversal Order: 0 1 2 3 5 4

Vertex  Start Time      End Time
0               1               12
1               2               11
2               3               10
3               4               7
4               8               9
5               5               6
```

2. BFS TRAVERSAL

```
******************** BFS TRAVESAL RESULT ************************

Adjacency Matrix Representation:
0 1 1 1 0 0
1 0 1 0 0 0
1 1 0 1 1 0
1 0 1 0 0 1
0 0 1 0 0 0
0 0 0 1 0 0

BFS starting from which vertex? 0

BFS starting from vertex 0:
Enqueued 0
Dequeued element: 0
Node: 0, Level: 0
Enqueued 1
Enqueued 2
Enqueued 3
Dequeued element: 1
Node: 1, Level: 1
Dequeued element: 2
Node: 2, Level: 1
Enqueued 4
Dequeued element: 3
Node: 3, Level: 1
Enqueued 5
Dequeued element: 4
Node: 4, Level: 2
Dequeued element: 5
Node: 5, Level: 2

BFS Traversal Array: 0 1 2 3 4 5
```