Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name: Gautam D. Chaudhari
Roll No: 04
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim : Implementation of DFS and BFS traversal of graph.

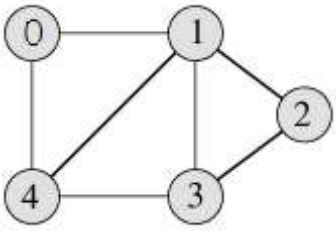
Objective:

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

Theory:

A graph is a collection of nodes or vertices, connected in pairs by lines referred to as edges. A graph can be directed or undirected.

One method of traversing through nodes is depth first search. Here we traverse from the starting node and proceed from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



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

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

Input: V is a starting vertex

Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

1. if gptr = NULL then
   print "Graph is empty" exit
2. u=v
3. OPEN.PUSH(u)
4. while OPEN.TOP !=NULL do
   u=OPEN.POP()
   if search(VISIT,u) = FALSE then
   INSERT_END(VISIT,u)
   Ptr = gptr(u)
   While ptr.LINK != NULL do

Vptr = ptr.LINK

OPEN.PUSH(vptr.LABEL)

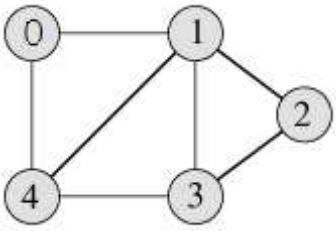
End while

End if

End while

5. Return VISIT
6. Stop

BFS Traversal



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

BFS Traversal – 0 1 4 2 3

Algorithm

Algorithm: DFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

push(j)

}

i=pop()

print("Visited vertex i")

visited[i]=1

count++

Algorithm: BFS()

i=0

count=1

visited[i]=1

print("Visited vertex i")

repeat this till queue is empty or all nodes visited

repeat this for all nodes from first till last

if(g[i][j]!=0&&visited[j]!=1)

{

enqueue(j)

}

i=dequeue()

print("Visited vertex i")

visited[i]=1

count++

Code:

DFS

```
#include <stdio.h>

#define MAX 5


void depth_first_search(int adj[][MAX], int visited[], int start) {

   int stack[MAX];

   int top = -1, i;

   printf("%c–", start + 65);

   visited[start] = 1;

   stack[++top] = start;
```

```
    while (top != -1) {
        start = stack[top];
        for (i = 0; i < MAX; i++) {
            if (adj[start][i] && visited[i] == 0) {
                stack[++top] = i;
                printf("%c–", i + 65);
                visited[i] = 1;
                break;
            }
        }
        if (i == MAX)
            top--;
    }
}

int main() {
    int adj[MAX][MAX];
    int visited[MAX] = {0};
    int i, j;

    printf("\n Enter the adjacency matrix: ");
    for (i = 0; i < MAX; i++)
        for (j = 0; j < MAX; j++)
            scanf("%d", &adj[i][j]);

    printf("DFS Traversal: ");
```

```
    depth_first_search(adj, visited, 0);
    printf("\n");
    return 0;
}
```

BFS

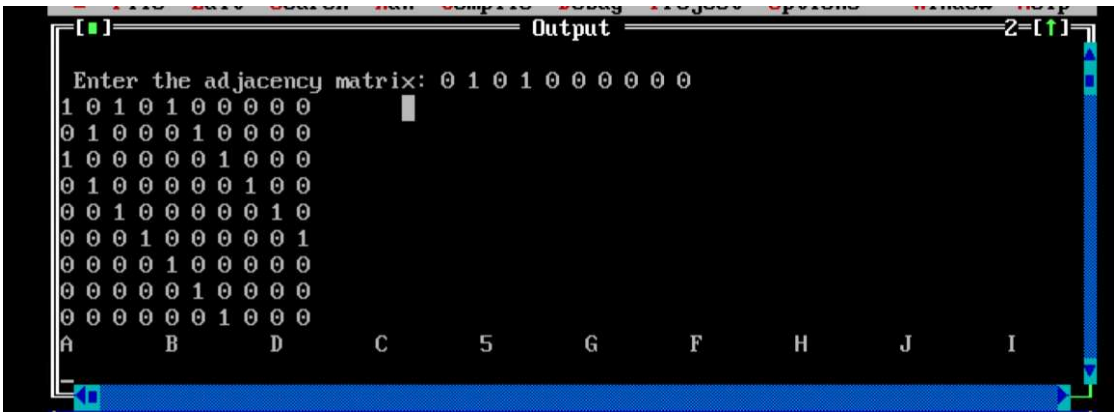
```
#include <stdio.h>
#define MAX 10

void breadth_first_search(int adj[][MAX], int visited[], int start) {
    int queue[MAX];
    int rear = -1, front = -1, i;
    queue[++rear] = start;
    visited[start] = 1;

    while (rear != front) {
        start = queue[++front];

        if (start == 4) {
            printf("5\t");
        } else {
            printf("%c\t", start + 65);
        }

        for (i = 0; i < MAX; i++) {
            if (adj[start][i] == 1 && visited[i] == 0) {
                queue[++rear] = i;
                visited[i] = 1;
            }
        }
    }
}
```

```
int main() {
    int visited[MAX] = {0};
    int adj[MAX][MAX];
    int i, j;

    printf("\nEnter the adjacency matrix:\n");
    for (i = 0; i < MAX; i++) {
        for (j = 0; j < MAX; j++) {
            scanf("%d", &adj[i][j]);
        }
    }

    breadth_first_search(adj, visited, 0);
    return 0;
}
```

Output:

DFS

BFS



Conclusion:

Write the graph representation used by your program and explain why you choose that.
BFS and DFS using the adjacency matrix representation
Adjacency Matrix:

- In an adjacency matrix, a 2D array is used to represent a graph.
- The rows and columns of the matrix represent the graph's vertices.
- A '1' in matrix[i][j] indicates that there is an edge between vertex i and vertex j.
- This representation is suitable for dense graphs (where most of the vertex pairs have edges) and is relatively space-efficient in such cases.

Write the applications of BFS and DFS other than finding connected nodes and explain how it is attained?
Breadth-First Search (BFS) and Depth-First Search (DFS) are fundamental graph traversal algorithms with various applications beyond finding connected nodes:
BFS Applications :
Shortest Path: BFS finds the shortest path between two nodes in an unweighted graph by exploring nodes level by level.
Minimum Spanning Tree (MST): In a connected graph with equal edge weights, BFS can discover the Minimum Spanning Tree, a vital concept in network design and optimization.

Web Crawling: Search engines like Google use BFS for web crawling, systematically discovering and indexing web pages starting from a seed URL.
Social Network Analysis: BFS is used to analyze social networks, identifying degrees of separation, recommending friends, and studying information or disease spread.
Puzzle Solving: It's applied to solve puzzles like the sliding tile puzzle or Rubik's Cube by exploring possible moves in a systematic manner.

DFS Applications :
Topological Sorting: DFS finds a topological ordering in Directed Acyclic Graphs (DAGs). This ordering is crucial for scheduling tasks with dependencies.
Cycle Detection: DFS can detect cycles in graphs. It's vital in compilers to identify circular dependencies and prevent infinite loops in code generation.
Maze Solving: DFS is used to solve mazes. It explores paths deeply before backtracking, systematically searching for solutions.
Game Strategies: In game development, DFS explores various game states deeply, allowing it to determine the best strategy. Games like chess or tic-tac-toe benefit from this.
Arborescence Detection: DFS is used to identify arborescences (rooted, directed trees) within a graph, aiding in network design and optimization