ASSIGNMENT 3

AIM:

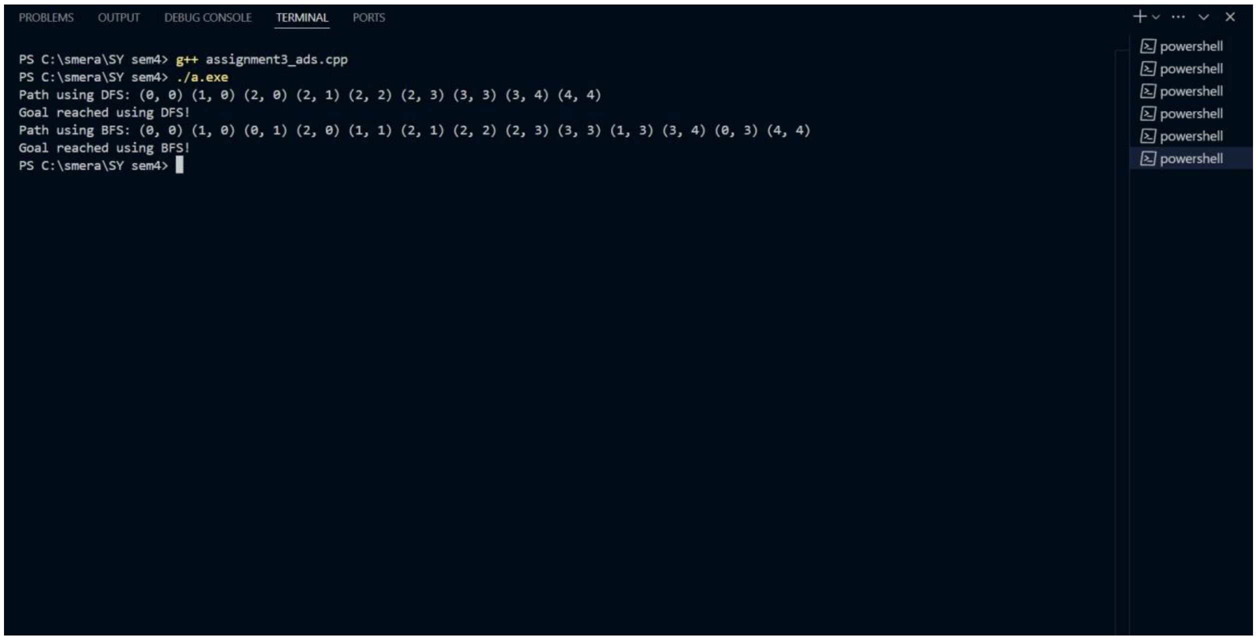
Consider the scenario of a maze. The maze is represented as a grid of cells, where each cell can be either open or blocked. Each cell in the maze represents a vertex. The goal is to find a path from the starting point to the goal within a given maze using DFS and BFS.

SOURCE CODE:

```
#include <iostream>
#include <queue>
using namespace std;

const int ROWS = 5;
const int COLS = 5;

// Directions: down, right, up, left
int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};

// Check if the cell is valid
bool isSafe(int maze[ROWS][COLS], bool visited[ROWS][COLS], int x, int y) {
    return x >= 0 && x < ROWS && y >= 0 && y < COLS && maze[x][y] == 0 &&
!visited[x][y];
}
```

```
// DFS implementation
bool dfs(int maze[ROWS][COLS], bool visited[ROWS][COLS], int x, int y) {
    if (x == ROWS - 1 && y == COLS - 1) {
        cout << "(" << x << ", " << y << ") ";
        return true;
    }

    if (isSafe(maze, visited, x, y)) {
        visited[x][y] = true;
        cout << "(" << x << ", " << y << ") ";

        for (int i = 0; i < 4; i++) {
            if (dfs(maze, visited, x + dx[i], y + dy[i])) {
                return true;
            }
        }

        // Backtrack
        visited[x][y] = false;
    }
    return false;
}

// BFS implementation
void bfs(int maze[ROWS][COLS]) {
    bool visited[ROWS][COLS] = {false};
    queue<pair<int, int>> q;
    q.push({0, 0});
    visited[0][0] = true;

    while (!q.empty()) {
        pair<int, int> cell = q.front();
        q.pop();
        int x = cell.first;
        int y = cell.second;

        cout << "(" << x << ", " << y << ") ";

        if (x == ROWS - 1 && y == COLS - 1) {
            cout << "\nGoal reached using BFS!" << endl;
            return;
        }

        for (int i = 0; i < 4; i++) {
            int newX = x + dx[i];
            int newY = y + dy[i];

            if (isSafe(maze, visited, newX, newY)) {
                visited[newX][newY] = true;
                q.push({newX, newY});
            }
        }
    }

    cout << "\nNo path found using BFS." << endl;
}

int main() {
    int maze[ROWS][COLS] = {
        {0, 0, 1, 0, 0},
        {0, 0, 1, 0, 1},
        {0, 0, 0, 0, 1},
        {1, 1, 1, 0, 0},
        {0, 0, 0, 1, 0}
    };

    bool visited[ROWS][COLS] = {false};

    // DFS
    cout << "Path using DFS: ";
    if (dfs(maze, visited, 0, 0)) {
        cout << "\nGoal reached using DFS!" << endl;
    } else {
        cout << "\nNo path found using DFS." << endl;
    }

    // BFS
    cout << "Path using BFS: ";
    bfs(maze);
    return 0;
}
```

OUTPUT:



```
PS C:\smera\SY sem4> g++ assignment3_ads.cpp
PS C:\smera\SY sem4> ./a.exe
Path using DFS: (0, 0) (1, 0) (2, 0) (2, 1) (2, 2) (2, 3) (3, 3) (3, 4) (4, 4)
Goal reached using DFS!
Path using BFS: (0, 0) (1, 0) (0, 1) (2, 0) (1, 1) (2, 1) (2, 2) (2, 3) (3, 3) (1, 3) (3, 4) (0, 3) (4, 4)
Goal reached using BFS!
PS C:\smera\SY sem4>
```

CONCLUSION:
The maze-solving problem modeled as a graph traversal task demonstrates the practical application of search algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS). Representing the maze as a grid of vertices allows for an efficient exploration of paths. DFS explores deeper paths first and can be useful for scenarios requiring complete path exploration, whereas BFS explores all neighboring nodes level by level, making it ideal for finding the shortest path in an unweighted maze. By implementing both algorithms, we gain insight into their strengths and trade-offs in terms of time, space, and traversal patterns. This approach not only highlights foundational concepts in graph theory but also provides a logical and structured solution to real-world pathfinding problems.