DEPARTMENTOF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

Project Report:

Pathfinding Prowess: Solving Mazes with Shortest Paths Using DFS and BFS

Subject Name – Design and Analysis of Algorithms

Subject Code – 23CSH-301

Submitted To:
Er. Mohammad Shaqlain
(E17211)

Submitted By:
Name: Shivam Kumar Tiwari
UID: 23BCS12755
Section: KRG_2-B

1. Aim

To solve Mazes with Shortest Paths Using Depth-First Search and Breadth-First Search

2. Objective

Create an algorithm to solve mazes by finding the shortest path from the start to the exit, using either Depth-First Search or Breadth-First Search.

3. Apparatus Used

- Programming Language: C++
- IDE Used: Code:: VS Code / Visual Studio

4. Procedure / Algorithm

A) Algorithm: Maze Solver Using BFS (Shortest Path)

Input:

- A 2D maze represented as a grid (`0` = open path, `1` = wall).
- Start coordinates `(Sx, Sy)` and Exit coordinates `(Ex, Ey)`.

Output:

- Shortest path from start to exit (if exists).

Stepwise Algorithm

1. **Initialize structures**
   - Create a queue `Q` to hold positions to explore.
   - Create a 2D `visited` array to mark visited cells.
   - Create a `parent` dictionary to trace the path.
2. **Add starting point**
   - Enqueue the start cell `(Sx, Sy)` into `Q`.
   - Mark the start cell as visited.
3. **Define movement**
   - Possible movements: `up`, `down`, `left`, `right`.
4. **Start BFS loop**
   - While the queue `Q` is not empty:
     - Dequeue the current cell `(x, y)`.
     - If `(x, y)` is the exit `(Ex, Ey)`, stop BFS (exit found).
     - For each valid neighboring cell `(nx, ny)`:
       - Check if `(nx, ny)` is inside maze boundaries.
       - Check if `(nx, ny)` is open (`0`) and not visited.
       - Enqueue `(nx, ny)` into `Q`.
       - Mark `(nx, ny)` as visited.

- Store `(x, y)` as the parent of `(nx, ny)`.

5. **Trace back the path**
   - **If the exit is reached:**
     - Initialize an empty list `path`.
     - Start from exit cell `(Ex, Ey)`.
     - Move backward using the `parent` dictionary until reaching the start.
     - Reverse the `path` to get the correct order from start to exit.
6. **Output the result**
   - If `path` exists, print the shortest path coordinates.
   - Else, print "No path exists."

CODE:

```
#include <bits/stdc++.h>

using namespace std;

struct Point {

   int x, y;

};

// Directions: up, down, left, right

int dx[] = {-1, 1, 0, 0};

int dy[] = {0, 0, -1, 1};
```

```
bool isValid(int x, int y, int rows, int cols, vector<vector<int>> &maze,
vector<vector<bool>> &visited) {
    return (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 0 &&
!visited[x][y]);
}
vector<Point> bfsMazeSolver(vector<vector<int>> &maze, Point start, Point end) {
    int rows = maze.size();
    int cols = maze[0].size();
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));
    map<pair<int,int>, pair<int,int>> parent;
    queue<Point> q;
    q.push(start);
    visited[start.x][start.y] = true;
    while (!q.empty()) {
        Point curr = q.front(); q.pop();
        if (curr.x == end.x && curr.y == end.y) {
            // Trace back path
            vector<Point> path;
            pair<int,int> p = {end.x, end.y};
            while (!(p.first == start.x && p.second == start.y)) {
                path.push_back({p.first, p.second});
```

```
            p = parent[p];
         }
         path.push_back(start);
         reverse(path.begin(), path.end());
         return path;
      }
      for (int i = 0; i < 4; i++) {
         int nx = curr.x + dx[i];
         int ny = curr.y + dy[i];
         if (isValid(nx, ny, rows, cols, maze, visited)) {
            visited[nx][ny] = true;
            q.push({nx, ny});
            parent[{nx, ny}] = {curr.x, curr.y};
         }
      }
   }
   return {}; // No path found
}
```

B) Algorithm: Maze Solver Using DFS (Pathfinding)

Input:

- 2D maze (`0` = open, `1` = wall)
- Start `(Sx, Sy)` and Exit `(Ex, Ey)`

Output:

- A path from start to exit (may not be shortest).

Stepwise Algorithm

1. Initialize structures
   - Create a 2D `visited` array to mark cells as visited.
   - Create a `path` list to store the current path.
2. Define movement
   - Possible directions: up, down, left, right.
3. Recursive DFS function
   - Function `dfs(x, y)`:
     1. If `(x, y)` is outside maze or is a wall or visited, return `False`.
     2. Add `(x, y)` to `path` and mark visited.
     3. If `(x, y)` is the exit, return `True`.
     4. For each neighbor `(nx, ny)`:

- Recursively call `dfs(nx, ny)`.
- If recursive call returns `True`, propagate `True`.

5. If no path found, backtrack: remove `(x, y)` from `path` and return `False`.

4. Call DFS
   - Start DFS from `(Sx, Sy)`.
5. Output
   - If DFS succeeds, `path` contains the route from start to exit.
   - Else, print "No path exists."

CODE:

```
#include <bits/stdc++.h>

using namespace std;


struct Point {

    int x, y;

};


int dx[] = {-1, 1, 0, 0};

int dy[] = {0, 0, -1, 1};
```

```
bool isValid(int x, int y, int rows, int cols, vector<vector<int>> &maze,
vector<vector<bool>> &visited) {

    return (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 0 &&
!visited[x][y]);

}

bool dfsMazeSolver(vector<vector<int>> &maze, Point curr, Point end,
vector<vector<bool>> &visited, vector<Point> &path) {

    int rows = maze.size();

    int cols = maze[0].size();

    if (!isValid(curr.x, curr.y, rows, cols, maze, visited)) return false;

    path.push_back(curr);

    visited[curr.x][curr.y] = true;

    if (curr.x == end.x && curr.y == end.y) return true;

    for (int i = 0; i < 4; i++) {

        Point next = {curr.x + dx[i], curr.y + dy[i]};

        if (dfsMazeSolver(maze, next, end, visited, path)) return true;

    }

    path.pop_back(); // backtrack

    return false;

}
```

6. Complexity Analysis

Algo	Best Case	Worst / Average Case
BFS	O(1) – exit found immediately	O(R × C) – explores all cells in the maze
DFS	O(1) – exit found immediately	O(R × C) – may traverse all cells before finding exit

7. Result

- **BFS successfully finds the shortest path from start to exit.**
- **DFS finds a path (may not be shortest), demonstrating recursive backtracking.**

8. Conclusion

- **BFS is ideal when the shortest path is required.**
- **DFS is simpler but may not yield the shortest path, useful for exploring all possible routes.**