# DESIGN AND DEVELOPMENT OF A TERMINAL-BASED

# MAZE PATH FINDER

Bachelor of Technology

in

**Computer science and Engineering**

Submitted by

**Mohammed Jissan(PRN:24070721022) and Sai ram Reddy(24070721010)**

For the Fulfillment

of

**Data Structures and Algorithms Course**

SYMBIOSIS
INSTITUTE OF TECHNOLOGY,
HYDERABAD

**Symbiosis Institute of Technology, Hyderabad**

Survey Number:292, Banglore Highway, Modallaguda Village, Nandigama Mandal, Ranga Reddy DIST, Near Hyderabad, Telangana- 509217.

# Contents

# 1. Introduction

This project focuses on the classic problem of finding the **shortest path** from a start point ('S') to a finish point ('E') within a two-dimensional grid, or maze. The challenge demonstrates the application of graph traversal techniques to solve pathfinding problems in an unweighted grid environment. The grid is implicitly treated as a graph where each cell is a node, and movement to adjacent cells represents edges.

# 2. Objectives

The primary goals of this project are:

- To successfully model the 2D maze as an **unweighted graph**.
- To implement the **Breadth-First Search (BFS)** algorithm, which guarantees finding the globally shortest path in terms of steps.
- To implement a **path tracing mechanism** to reconstruct the sequence of steps from start to finish.
- To provide a clear output showing the sequence of steps and a visual map of the path.

# 3. Technologies Used

- **C Programming Language**: Chosen for its efficiency and direct control over memory and data structures (like arrays and custom structs).
- **Queue Data Structure**: Essential for the BFS algorithm, ensuring nodes are explored in a layer-by-layer manner (FIFO).
- **Custom Structs (`Point`, `PathNode`)**: Used to encapsulate cell coordinates and pathfinding metadata (distance, parent index).

# 4. Implementation Details

## 4.1. Algorithm: Breadth-First Search (BFS)

**BFS** is employed because it guarantees the **shortest path** in terms of the number of steps in an unweighted grid. It explores the graph level by level, ensuring the goal is found via the minimum number of edges.

## 4.2. Key Structures

- **`PathNode` Struct**: Stores the cell coordinates, its distance (`dist`) from the start, and a crucial `parent_index`.

- **Path Tracing**: The `parent_index` stores the array index of the preceding node in the queue. Once the end node is found, the final path is **reconstructed by tracing backward** from the end to the start.
- `visited` **Array**: A 2D boolean array is used to mark cells already processed, preventing cycles and ensuring an optimal time complexity of $O(R \cdot C)$.

# 5. Features and Demonstration

- **Shortest Path Guarantee**: Achieved via the inherent properties of BFS on an unweighted graph.
- **Clear Output**: Provides the total shortest path length and the sequence of coordinates for the path.
- **Visual Representation**: The maze is reprinted with the shortest path marked by the * character.

# 6. Source Code

The complete C implementation for the Maze Path Finder using BFS is provided below.

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

#define N 5

int maze[N][N] = {
    {0,0,1,0,0},
    {1,0,1,0,1},
    {0,0,0,0,0},
    {0,1,1,1,0},
    {0,0,0,1,0}
};

int visited[N][N], parent[N][N][2];
int qx[100], qy[100], front = 0, rear = 0;

int dx[4] = {1,-1,0,0};
int dy[4] = {0,0,1,-1};

void printPath(int ex, int ey) {
    int x = ex, y = ey;
    printf("\nShortest Path (reversed):\n");
    while (!(parent[x][y][0] == -1 && parent[x][y][1] == -1)) {
        printf("(%d,%d) ", x, y);
        int px = parent[x][y][0];
        int py = parent[x][y][1];
        x = px; y = py;
    }
    printf("(S)\n");
}

void bfs(int sx, int sy, int ex, int ey) {
```

```
    visited[sx][sy] = 1;
    parent[sx][sy][0] = parent[sx][sy][1] = -1;
    qx[rear] = sx; qy[rear++] = sy;

    while (front < rear) {
        int x = qx[front], y = qy[front]; front++;

        if (x == ex && y == ey) {
            printf("\nPath Found!\n");
            printPath(ex, ey);
            return;
        }

        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i], ny = y + dy[i];

            if (nx>=0 && nx<N && ny>=0 && ny<N && maze[nx][ny] == 0 && !visited[nx][ny]) {
                visited[nx][ny] = 1;
                parent[nx][ny][0] = x;
                parent[nx][ny][1] = y;

                qx[rear] = nx;
                qy[rear++] = ny;
            }
        }
    }
    printf("\nNo Path Found!\n");
}

int main() {
    int sx=0, sy=0, ex=4, ey=4;

    printf("MAZE:\nS=Start, E=End\n\n");
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            if(i==sx && j==sy) printf("S ");
            else if(i==ex && j==ey) printf("E ");
            else printf("%d ", maze[i][j]);
        }
        printf("\n");
    }

    bfs(sx, sy, ex, ey);
    return 0;
}
```

Listing 1: Maze Path Finder (BFS) Code

# 7. Sample Output

Based on the defined sample maze in the `main()` function (Start at (1, 1), End at (5, 5)), the BFS algorithm finds the following shortest path.

### 7.1. Expected Terminal Output

The path length is the number of steps taken from S to E, which is 7.

```
MAZE:
S 0 1 0 0
1 0 1 0 1
0 0 0 0 0
0 1 1 1 0
0 0 0 1 E

Path Found!
Shortest Path (reversed):
(4,4) (3,4) (2,4) (2,3) (2,2) (1,1) (0,0) (S)
```

## 8. Code Analysis and Explanation

The Maze Path Finder implementation uses the **Breadth-First Search (BFS)** algorithm to guarantee finding the shortest path in the unweighted grid. The code is structured into logical blocks for clarity and efficiency.

### 8.1. Header Files and Structures

This initial block sets up the environment and defines the custom data types required for the search.

- **Constants**: Define the maximum grid size (`MAX_SIZE`) and the size of the static queue (`QUEUE_CAPACITY`).
- `Point` **Struct**: Stores simple `row` and `col` coordinates for a cell.
- `PathNode` **Struct**: The core BFS structure, storing the current location, the distance (`dist`) from the start, and the vital `parent_index`, which points to the predecessor node in the queue for path reconstruction.
- **Global Queue**: The pathfinding queue is implemented using a global array (`path_queue`) and two indices (`head` and `tail`) for standard FIFO operations.

### 8.2. Queue Operations

These functions abstract the basic queue management required for BFS:

- `is_empty()`: Returns true when the queue has been fully processed.
- `enqueue()`: Adds a new node to the back of the queue.
- `dequeue()`: Retrieves the next node to be processed from the front of the queue.

### 8.3. BFS Algorithm Core (`find_shortest_path`)

This function executes the main search logic.

- **Direction Arrays**: `dr` and `dc` define the $\Delta$row and $\Delta$col needed to check the four neighboring cells (Up, Down, Left, Right).
- **Initialization**: A 2D boolean array (`visited`) is used to track cells that have already been queued, preventing cycles and redundant processing. The starting node is enqueued, and its position is marked true in the `visited` array.
- **Traversal Loop**: The `while` loop processes nodes in layers. For the dequeued node, it checks all four neighbors.
- **Validity Check**: A neighbor is processed only if it is within maze bounds, has not been visited, and is not a wall (`#`). Valid neighbors are marked visited, and a new `PathNode` is created with its `parent_index` set to the current node's index.

### 8.4. Path Tracing and Reconstruction

This section handles path extraction after the goal is found.

- **Goal Identification**: Once the end point ('E') is dequeued, its index is saved as `end_node_index`, and the BFS loop breaks.
- **Backtracking**: If the path was found, a `while` loop starts from `end_node_index` and iteratively uses the `parent_index` to move backward through the queue, retrieving the coordinates of the shortest path.
- **Path Storage**: Coordinates are stored in the output arrays (`path_R`, `path_C`) from the end to the start. The function returns the `path_length` (total steps + 1).

### 8.5. Display and Main Execution

These are the utility and driver functions.

- `print_maze_with_path()`: A utility function that copies the original maze, iterates through the calculated path, and marks the path cells with a `*` for visual demonstration.
- `main()`: Defines the sample maze, locates the start and end points, calls `find_shortest_path()`, and prints the final output, including the path length, the sequence of coordinates, and the visual maze.

## 9. References


- Standard C Library Documentation.
- Introduction to Graph Theory and Breadth-First Search Algorithm.