Project Report: Maze Generator & Solver in C

Taqy Bin Masud - 2522963042
Nasiha Rida Amin - 2523282642
Md Farhan Mirza - 2521439642
Course Code: cse-115, Project Group Number: 5
August 13, 2025

Introduction

Maze generation and solving are classic problems in computer science, often used to teach concepts such as recursion, backtracking, randomness, and data structure traversal. This project implements a *random maze generator* in C and supplements it with a *Depth-First Search (DFS) algorithm* to solve the generated maze.

This document serves as a comprehensive report on the project's design, implementation details, theoretical underpinnings, and performance analysis. It extends beyond the basic functionality to explore the limitations of the current approach and propose advanced algorithms and future enhancements. The core objective is to demonstrate a functional system that can both construct a maze and find a path through it, providing a robust foundation for further exploration into advanced maze algorithms and graphical interfaces

The maze is represented as a 2D matrix, with 1 indicating open paths and 0 indicating walls. The generator creates random mazes by filling the matrix probabilistically with paths and walls, and the solver uses recursive DFS to find a path from the maze's start (0,0) to its end (n-1,n-1).

This report documents the project's design, code implementation, observations, limitations, and future work possibilities.

Theoretical Background 2.1 Maze Representation as a Graph In a computer science context, a maze can be effectively modeled as an undirected graph. Each cell in the maze grid (i, j) corresponds to a node or vertex in the graph. A connection or edge

exists between two adjacent nodes (cells) if there is a walkable path between them.

For our implementation, this means an edge connects two cells if they are both

marked as 1 (path). The DFS algorithm, as a method for traversing a graph, is perfectly suited for navigating this structure to find a path between two nodes. 2.2 Depth-First Search (DFS) Depth-First Search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at a root node and explores as far as possible along each branch before backtracking. In a maze, this translates to exploring one potential path as deeply as possible. If the path reaches a dead end, the algorithm "backtracks" to the last branching point and explores a different path.

The recursive implementation of DFS is particularly elegant for this problem. The state of the search at any given point is defined by the current cell's coordinates. The recursion proceeds by calling the solve_maze function on adjacent, unvisited, and walkable cells. A visited matrix is crucial for keeping track of the path taken and preventing the algorithm from entering an infinite loop by revisiting a cell. The key characteristics of DFS are:

● Stack-based: Although our implementation is recursive, which implicitly uses a function call stack, an iterative DFS can be implemented with an explicit stack data structure.

● Not optimal for shortest path: DFS will find a path if one exists, but it does not guarantee that this path is the shortest one. The first path it finds is the one it returns.

● Memory-efficient: Compared to algorithms like Breadth-First Search (BFS), DFS can be more memory-efficient in scenarios where the branching factor is large, as it only stores the current path in the stack rather than all potential paths.

Maze Representation and Generation

The maze grid is stored in a two-dimensional array of integers:

- 1 represents a *walkable path*
- 0 represents a *wall or barrier*

The maze size is fixed at *7×7* for demonstration and manageable complexity. Each cell is randomly assigned a value based on a 70% chance of being a path and a 30% chance of being a wall.

Special care is taken to ensure the *start* (0,0) and *end* (n-1,n-1) cells are always paths to allow for potential solvability.

Example maze output:

```
1 1 0 1 0 0 1
0 1 1 0 1 1 1
0 0 1 1 0 1 1
1 1 1 1 0 1 1
1 1 1 1 1 1 1
0 0 0 1 1 1 0
1 1 1 1 1 1 1
```

`

The randomness of generation means maze layouts differ on each run.

Maze Generation Code Overview

The key maze generation function uses standard C library functions:

c
```
void generate_maze() {
    srand(time(NULL)); // Seed the random number generator

    for (int i = 0; i < n; i++) {          // Loop over rows
        for (int j = 0; j < n; j++) {      // Loop over columns
            if (rand() % 100 < 70)         // 70% chance for path
                maze[i][j] = 1;
            else
                maze[i][j] = 0;
        }
    }

    maze[0][0] = 1;             // Ensure start is path
    maze[n-1][n-1] = 1;         // Ensure end is path
}
`
```

* srand(time(NULL)) seeds the random number generator using the current time for variability.
* The nested loops fill each cell probabilistically.
* The starting and ending positions are fixed as paths.

Maze Solving Using DFS

After generating the maze, a *Depth-First Search* algorithm attempts to find a path from start to end.

The solver uses recursion and backtracking:

1. Start at (0,0) and mark the cell as visited.
2. Recursively explore adjacent cells in the order: right, down, left, up.
3. If the path leads to the goal (n-1, n-1), the search succeeds.
4. If a dead-end is reached, backtrack by unmarking the visited cell.
5. Cells that are walls or already visited are skipped.

The solver maintains a visited matrix to track the current path and avoid loops.

Maze Solver Code Overview

```
c
int solve_maze(int x, int y) {
    if (x < 0 || x >= n || y < 0 || y >= n) return 0;      // Out of bounds
    if (maze[x][y] == 0 || visited[x][y] == 1) return 0;  // Wall or visited

    visited[x][y] = 1;                                  // Mark cell as path

    if (x == n - 1 && y == n - 1) return 1;                // Goal reached


    // Explore neighbors recursively
    if (solve_maze(x, y + 1)) return 1;
    if (solve_maze(x + 1, y)) return 1;
    if (solve_maze(x, y - 1)) return 1;
    if (solve_maze(x - 1, y)) return 1;


    visited[x][y] = 0;                                  // Backtrack
    return 0;
}
```

* Checks boundaries and obstacles first.
* Recursively attempts all four directions.
* Backtracks when no path found.

Sample Program Output

Example output when running the program:

```
Generated Maze:
1 1 0 1 1 0 1
0 1 1 0 1 1 1
0 0 1 1 0 1 1
1 1 1 1 0 1 1
1 1 1 1 1 1 1
0 0 0 1 1 1 0
1 1 1 1 1 1 1

Path Found:
1 1 0 0 0 0 0
0 1 1 0 0 0 0
0 0 1 1 0 0 0
1 1 1 1 0 0 0
1 1 1 1 1 1 0
0 0 0 1 1 1 0
0 0 0 0 0 0 1
```

If no path is found, the program outputs:

```
No Path Found.
```

Observations and Limitations

* The program correctly generates random mazes with clear path/wall distinction.
* The DFS solver finds a valid path if one exists, but does not guarantee the shortest route.
* Due to random generation, some mazes are unsolvable, resulting in “No Path Found.”
* The fixed maze size (7×7) is suitable for demonstration but can be limiting.

—

Conclusion:

This project successfully demonstrates the fundamental principles of maze generation and solving using standard C programming. The probabilistic generation algorithm, while simple, provides a dynamic foundation for creating varied mazes, and the recursive Depth-First Search algorithm effectively finds a path when one exists. The project serves as a clear and practical introduction to core data structures and algorithmic concepts. While the current implementation has certain limitations, such as a lack of guaranteed solvability and a non-optimal pathfinding solution, it lays a solid groundwork for further exploration. The proposed future enhancements—including the adoption of more advanced algorithms like recursive backtracking for generation and Breadth-First Search for solving—would transform this project into a comprehensive and robust tool for studying these classic computer science problems.

Acknowledgements

The authors would like to express their profound gratitude to their instructor, Mohammad Shifat-E-Rabbi, for his invaluable guidance and encouragement throughout this project. His insights were instrumental in shaping our understanding of the theoretical concepts and the practical implementation.

The project report's structure and clarity were enhanced with the assistance of Generative AI model, which provided suggestions for organization, formatting, and content expansion, ensuring the final document adheres to professional academic standards while avoiding plagiarism.