Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report on "Maze Generator and Path Finder"

[Code No.: COMP 202]

(For partial fulfillment of the requirements for internal assessment in Data Structure and Algorithm)

Submitted by

Bidhan Timilsina (038011-24)

Submitted to

Mr. Sagar Acharya

Department of Computer Science and Engineering

February 24, 2026

Acknowledgements

I would like to express my heartfelt gratitude to everyone who supported and contributed for the successful completion of my mini-project titled ”Maze Generator and Path Finder”.

I am deeply thankful to our lecturer, Mr. Sagar Acharya, for providing us with this golden opportunity through which I learned to apply what we learned in our Data Structure and Algorithm (DSA) lectures.

I would like to extend my thank to my classmates, friends, and families for their moral support and motivation during this mini-project.

Abstract

This article briefly introduces Maze Generator and Pathfinding Visualizer as two main components of the project. The software, written in C++17 and using the Raylib graphics library, primarily aims to demonstrate elementary graph traversal algorithms visibly through the release visualization of the algorithm in the interactive environment. The system generates a random maze that is fully connected and solvable using the Recursive Backtracker method, which is another name for Depth-First Search (DFS) and one of its variations. One can think of the maze as a graph where each cell is a vertex and the removal of the walls are edges that connect the vertices thus obtained. After creating the maze, users of the platform can opt to employ two graph traversal algorithms—Breadth-First Search (BFS) and Depth-First Search (DFS)—which are both implemented ways of finding the parts of the maze and, eventually, solving it.

Since BFS is a queue-based algorithm, it is able to find the shortest path between two points, start and finish. By periodically getting the information of the next move from the stack of recursive function calls, DFS proceeds along a single path until it reaches a dead end so it has to backtrack. On the other hand, it is not certain that the path found will be the shortest one.

The algorithms' steps as individual parts of the game loop in Raylib are highlighted through animation so that users can follow the pattern of their exploration and notice the difference in their behavior concerning the time of their moves. Maze Generator and Pathfinding Visualizer are not only at the core of the project but the whole project is also an educational resource in the areas of graph theory, concepts, traversal strategies, and pathfinding techniques.... By running both BFS and DFS on the same maze, one can visually observe the differences in the algorithms' approaches, their efficiency, and the optimality of the paths produced. Thanks to its modular design that clearly separates the functionalities responsible for maze generation, solving algorithms, and rendering, the project is easily upgradable and further development is very feasible.

Contents

List of Figures

Abbreviations

BFS	Breadth First Search.
CPU	Central Processing Unit.
DFS	Depth First Search.
DSA	Data Structure and Algorithm.
FPS	Frames Per Second.
IDE	Integrated Development Environment.
OS	Operating System.

Chapter 1

Introduction

1.1 Background

Graph traversal and pathfinding algorithms are often considered the backbone of computer science. Without them, we wouldn't have so many useful applications such as AI, robotics, games, internet protocols, etc. BFS and DFS are acronyms that stand for the two most fundamental graph searching techniques where the graph's nodes and edges are explored one step at a time.

So, if you get how these two work and what their advantages and disadvantages are, you will get an extremely high-level overview of computer science performance, solution quality, and different ways of thinking in the field. Drawing mazes and then solving them has always been a favorite puzzle of graph lovers. To understand this, think of a graph as a maze where each cell is linked with adjacent cells via edges. DFS combined with recursion backtracking is arguably one of the best and most efficient methods to generate mazes that are entirely connected and single-solution only. Conversely, even a simple maze can be easily handled by BFS or DFS to get one or more paths from the start to the goal.

Visualization tools offer an interactive and engaging way of learning and make the process less daunting as they provide you with a chance to do the algorithm walk on your screen, step-by-step. When you are able to visualize how different traversal techniques disclose the same graph, the ideas of which path is the shortest, how far the search goes, and what route the explorer takes become clearer. This paper is primarily focused on the extension of these initial principles and a contemporary tool has thus been developed that allows educators to visually demonstrate the algorithm to their students. This approach bridges the gap between algorithmic theory and actual visual understanding by transforming a well-known recursive problem into an interactive learning experience which can also be used to teach this concept.

1.2 Objectives

- To design and implement a random solvable maze generator using the Recursive Backtracker (DFS-based) algorithm.
- To implement and visualize Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms for maze solving.

- To compare the behavior and performance of BFS and DFS in terms of exploration pattern and path optimality.
- To develop an interactive educational tool that demonstrates graph traversal algorithms through real-time visualization.

1.3 Motivation and Significance

Understanding graph traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) is fundamental in computer science. However, these concepts are often taught theoretically, making it difficult for students to visualize how the algorithms actually explore a graph. The motivation behind this project is to bridge the gap between theory and practical understanding by creating an interactive visualization tool that demonstrates algorithm behavior in real time.

This project provides an educational platform to visually compare BFS and DFS on the same maze structure, clearly illustrating differences in exploration strategy and path optimality. By integrating maze generation with pathfinding visualization, the system strengthens understanding of graph representation, data structures (queue and stack), and algorithm efficiency. The modular implementation also demonstrates good software design practices, making the project both academically and technically valuable.

Chapter 2

Related Works

Several projects and studies have explored maze generation and pathfinding visualization, demonstrating the educational and analytical value of algorithm animation. For example, an interactive simulation platform was developed to implement and visualize multiple maze generation algorithms (including Recursive Backtracking) alongside pathfinding techniques such as DFS and BFS, emphasizing real-time exploration and comparative analysis of algorithm performance.

In the open-source development space, graphical maze solvers and pathfinding visualizers have been created using languages like Python and JavaScript. One such project implements random maze generation with DFS and visualizes BFS, DFS, Dijkstra's, and A* search on the maze, providing users with an interactive experience of algorithm execution. Another pathfinding visualizer project uses web technologies to allow users to draw grid obstacles and dynamically watch BFS and DFS explore the space, illustrating the differences in their search behavior.

Research literature also includes comparative studies that analyze BFS, DFS, and other search algorithms in maze traversal contexts, evaluating performance in terms of path length and execution characteristics, which underscores the importance of visual and empirical comparison in understanding algorithm strengths and limitations.

Together, these works highlight both practical implementations and analytical investigations into maze algorithms, situating this project within a broader context of tools designed to enhance learning and analysis of fundamental graph traversal techniques.

Chapter 3

Literature Review

Research on maze generation and pathfinding visualization has become an increasingly important topic in both academic research and practical software development. These subjects are fundamental to the study of algorithms, artificial intelligence, and interactive learning systems. Several studies have focused on developing interactive platforms that demonstrate maze generation and traversal techniques for educational purposes. For instance, an educational visualization platform integrating Recursive Backtracking, Prim's, and Kruskal's algorithms for maze generation along with BFS, DFS, and A* for pathfinding allows users to observe algorithm behavior in real time and perform comparative analysis (Bartaula, 2025; Kumar & Sharath, 2022).

Pictorial and interactive explanations have been shown to significantly enhance understanding of complex computational processes. Comparative analyses of pathfinding algorithms highlight differences in exploration strategies and efficiency. Studies evaluating BFS, DFS, and A* in maze traversal demonstrate that A*—being heuristic-based—often finds optimal paths more efficiently, while BFS guarantees the shortest path in unweighted graphs. DFS, on the other hand, explores deeply before backtracking and may produce suboptimal paths (Iloh, 2022).

Beyond traversal algorithms, research into maze generation techniques reveals variations in structural complexity and performance. Comparative studies of algorithms such as Recursive Backtracking and Binary Tree generation methods assess trade-offs in time complexity, maze structure characteristics, and suitability for educational visualization. Recursive Backtracking is frequently recognized for its simplicity and ability to generate perfect mazes, making it especially effective for teaching and visualization applications (Čarapina et al., 2024).

Overall, these studies reflect sustained interest in algorithm visualization tools and comparative performance analysis. They highlight the educational value of systems that combine maze generation with real-time pathfinding demonstrations, forming the conceptual foundation for the present project.

Chapter 4

Design and Implementation

4.1 System Architecture

The system follows a modular architecture to ensure separation of concerns, maintainability, and scalability. The application is divided into independent components responsible for maze generation, pathfinding algorithms, and graphical rendering. Each module interacts through clearly defined data structures, primarily the 2D grid representation of the maze.

The architecture is composed of the following major modules:

4.1.1 Maze Generation Module

This module implements the Recursive Backtracker algorithm (DFS-based) to generate a perfect maze. Each cell of the grid is treated as a vertex in a graph, and walls between adjacent cells represent edges. The algorithm begins from an initial cell, randomly selects unvisited neighbors, removes walls to create passages, and backtracks when no unvisited neighbors remain. This guarantees a fully connected and solvable maze with exactly one path between any two cells.

4.1.2 Pathfinding Algorithms Module

This module contains the implementation of Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms.

BFS uses a queue data structure to explore nodes level by level. It guarantees the shortest path in an unweighted maze.

DFS uses an explicit stack to explore as deeply as possible before backtracking. While DFS does not guarantee the shortest path, it demonstrates deep traversal behavior effectively for educational comparison.

Both algorithms maintain:

- A visited flag for each cell
- A parent mapping for path reconstruction

4.1.3 Visualization and Rendering Module

This module is responsible for all graphical output using the Raylib graphics library. It handles:

- Drawing maze walls and cells
- Animating algorithm exploration
- Coloring visited nodes (blue for BFS, orange for DFS)
- Highlighting the final path in yellow

Rendering occurs within the main game loop at approximately 60 frames per second, enabling smooth step-by-step visualization.

4.1.4 Control and Animation Module

The control module handles user inputs and animation flow. The user can:

- Press key 1 to run BFS
- Press key 2 to run DFS
- Press R to regenerate a new maze

Algorithm execution is distributed across multiple frames to visually demonstrate the search progression rather than computing instantly.

4.2 Algorithm Design

4.2.1 Maze Generation Algorithm

The Recursive Backtracker follows these steps:

1. Start from an initial cell and mark it visited.
2. Randomly select an unvisited neighbor.
3. Remove the wall between the current cell and chosen neighbor.

4. Push current cell onto stack.
5. Move to neighbor and repeat.
6. If no unvisited neighbors exist, pop from stack and backtrack.

This ensures generation of a perfect maze without isolated sections.

4.2.2 Breadth-First Search (BFS)

BFS operates using a queue:

1. Enqueue starting cell.
2. Mark it visited.
3. While queue is not empty:
   - Dequeue a cell.
   - Enqueue all unvisited neighbors.
   - Record parent for path reconstruction.

BFS guarantees shortest path in an unweighted graph.

4.2.3 Depth-First Search (DFS)

DFS uses a stack structure:

1. Push starting cell onto stack.
2. While stack is not empty:
   - Pop top cell.
   - Visit unvisited neighbors.
   - Push neighbors onto stack.

DFS explores deeply before backtracking and may produce non-optimal paths.

4.3 Data Structure Design

The core data structures include:

- **2D Vector Grid:** Represents the maze layout.
- **Boolean Visited Array:** Tracks explored cells.
- **Queue (BFS):** For level-order traversal.
- **Stack (DFS):** For depth-based traversal.
- **Parent Map:** Used to reconstruct final path.

Each cell stores wall information (top, bottom, left, right) and visitation status.

4.4 Visualization and Animation Design

Visualization is integrated into the Raylib main game loop:

- The window updates at 60 Frames Per Second (FPS).
- A limited number of algorithm steps are executed per frame.
- The screen is cleared and redrawn each frame.
- Colors represent algorithm states:
  - Blue – Breadth First Search (BFS) exploration
  - Orange – Depth First Search (DFS) exploration
  - Yellow – Final shortest path

This incremental rendering allows users to observe how the search frontier expands and how different algorithms behave dynamically.

4.5 Overall Workflow

The overall execution flow of the application is:

1. Initialize Raylib window and grid.

2. Generate maze using Recursive Backtracker.
3. Wait for user input.
4. Run selected pathfinding algorithm step-by-step.
5. Reconstruct and display final path.
6. Allow regeneration of new maze on command.

The modular structure ensures that maze generation, algorithm execution, and visualization remain independent yet coordinated through shared grid data.

Chapter 5

System Overview

5.1 System Requirement Specifications

5.1.1 Software Requirements

This section presents the required software components for developing and running the ”Maze Generator and Pathfinding Visualizer”.

Software Requirements

- **Operating System (OS):** Windows 10 / macOS
- **Programming Language:** C++17
- **Libraries / Packages:** Raylib
- **Integrated Development Environment (IDE) / Code Editor:** VS Code
- **Compiler:** MinGW (w64devkit)

5.1.2 Hardware Specifications

This section describes the recommended hardware required for the system.

Hardware Requirements

- **Processor:** Dual-core Central Processing Unit (CPU)
- **RAM:** 2 GB
- **Storage:** 500 MB free space
- **Graphics Card:** Any integrated graphics supporting OpenGL

5.2 Functional Requirements

The functional requirements describe what the system should be able to do. For the ”Maze Generator and Pathfinding Visualizer”, the key functional requirements are:

1. **Maze Generation:** The system shall generate a random, fully solvable maze using the Recursive Backtracker algorithm, ensuring all cells are reachable from any starting position.
2. **Graph Representation:** The system shall represent the maze as a grid-based graph where each cell acts as a vertex and removed walls between adjacent cells form the edges.
3. **Breadth-First Search (BFS):** The system shall implement BFS using a queue data structure to find and display the shortest path from the start cell to the end cell.
4. **Depth-First Search (DFS):** The system shall implement DFS using an explicit stack data structure to explore the maze deeply before backtracking, producing a valid but not necessarily shortest path.
5. **Real-Time Visualization:** The system shall animate the step-by-step exploration of each algorithm at 60 frames per second, highlighting visited cells and displaying the final computed path upon completion.
6. **User Interaction:** The system shall allow the user to regenerate the maze and select between pathfinding algorithms using keyboard input at any time during execution.

Chapter 6

Results and Discussion

6.1 Results

The Maze Generator and Pathfinding Visualizer was successfully implemented in C++ using the Raylib graphics library. The program correctly generates a random solvable maze and visualizes the step-by-step exploration of two graph traversal algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS).

The Recursive Backtracker algorithm accurately generates a fully connected maze each time, ensuring a valid path always exists between the start and end cell. The maze is represented as a 2D grid where each cell stores its wall states, and passages are carved by removing shared walls between adjacent cells.

The user can interact with the system using the following keyboard controls:

- Pressing **R** generates a new random maze and resets all solver state.
- Pressing **1** starts the BFS pathfinding visualization.
- Pressing **2** starts the DFS pathfinding visualization.

Both algorithms were tested across multiple randomly generated mazes. In all cases, BFS successfully identified the shortest path from the start cell to the end cell, while DFS produced a valid but not necessarily shortest path. The visited cells are highlighted in real time during execution — blue for BFS and orange for DFS — and the final computed path is displayed in yellow upon completion.

6.2 Discussion

The results demonstrate that combining graph algorithms with real-time visualization provides an effective educational tool for understanding how different traversal strategies behave on the same problem. The visual difference between BFS spreading outward evenly and DFS diving deep before backtracking makes the behavioral distinction between the two algorithms immediately apparent to the user.

The decision to run each algorithm incrementally — advancing only a few steps per frame within the Raylib game loop — proved effective for producing smooth, observable animations without blocking the rendering thread. This step-by-step approach allows users to watch the search frontier evolve naturally rather than seeing an instant result.

The parent map used for path reconstruction in both BFS and DFS ensured that the final path could be traced back from the end cell to the start cell efficiently. The use of a queue in BFS guarantees optimal path length in an unweighted graph, while the stack-based DFS explores one branch fully before backtracking, often resulting in a longer, winding path.

Using the Raylib library enabled lightweight rendering with responsive keyboard input, making the application suitable for educational environments. The modular design — separating maze generation, solver logic, and rendering into distinct classes — improved maintainability and made it straightforward to add DFS alongside the existing BFS implementation.

However, the current implementation has limitations at very large grid sizes. As the number of cells increases, the visited and path vectors grow proportionally, and the recursive maze generation algorithm may cause stack overflow on extremely large grids due to deep recursion. These limitations highlight real-world considerations around algorithmic time and space complexity.

Overall, the project successfully achieved its objectives of demonstrating graph traversal algorithms in an interactive and visual manner. The results confirm that side-by-side comparison of BFS and DFS on identical mazes significantly enhances conceptual understanding of shortest path guarantees, traversal order, and the practical trade-offs between different algorithmic approaches.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

This mini-project successfully implemented a Maze Generator and Pathfinding Visualizer using C++ and the Raylib graphics library. The application generates random solvable mazes and provides real-time visualization of two classical graph traversal algorithms, allowing users to observe and compare how each algorithm explores a maze.

The project demonstrates a practical application of graph theory by treating each maze cell as a vertex and each carved passage as an edge. The implementation of BFS and DFS as step-by-step animated solvers effectively connects theoretical algorithm concepts to visible, intuitive behavior. Users can clearly observe how BFS guarantees the shortest path through level-by-level exploration, while DFS produces a valid path through deep, directional traversal before backtracking.

The system is designed with a modular structure, separating maze generation, solver logic, and rendering into distinct classes. This makes the codebase readable, maintainable, and straightforward to extend. Overall, the project achieves its primary objective of helping students understand graph traversal algorithms, pathfinding, and the practical differences between BFS and DFS in an interactive and visual way.

7.2 Future Works

The current implementation provides a solid foundation. The following additions could improve the application further:

- **Dijkstra's Algorithm:** Add a weighted pathfinding option using a priority queue, allowing users to compare optimal cost-based pathfinding alongside BFS and DFS.
- **Animation Speed Control:** Allow users to adjust how many steps are taken per frame so the visualization can be slowed down or sped up as needed.
- **Step-by-Step Mode:** Add a paused mode where the user advances the algorithm one cell at a time using a keypress, enabling closer inspection of each decision.

- **Multiple Maze Generation Algorithms:** Implement alternative generation methods such as Prim's or Kruskal's algorithm to produce mazes with different structural characteristics.
- **Performance Optimization:** Replace the recursive maze generation with an iterative approach using an explicit stack to eliminate the risk of stack overflow on very large grids.

Bibliography

Bartaula, B. (2025). Interactive maze generation and pathfinding algorithms: An educational visualization platform for algorithmic analysis [Report on maze generation and pathfinding visualization platform]. *Zenodo*. https://doi.org/10.5281/zenodo.15823622

Čarapina, M., Staničić, O., Dodig, I., & Cafuta, D. (2024). A comparative study of maze generation algorithms in a game-based mobile learning application for learning basic programming concepts. *Algorithms*, *17*(9), 404. https://doi.org/10.3390/a17090404

Iloh, P. C. (2022). A comprehensive and comparative study of dfs, bfs, and a* search algorithms in solving the maze transversal problem. *International Journal of Social Sciences and Scientific Studies*, *2*(2), 482–490. https://www.ijssass.com/index.php/ijssass/article/view/54

Kumar, R., & Sharath, K. (2022). Path finding visualisation in mazes using various algorithms. *International Advanced Research Journal in Science, Engineering and Technology*. https://doi.org/10.17148/IARJSET.2022.96130

Appendix A

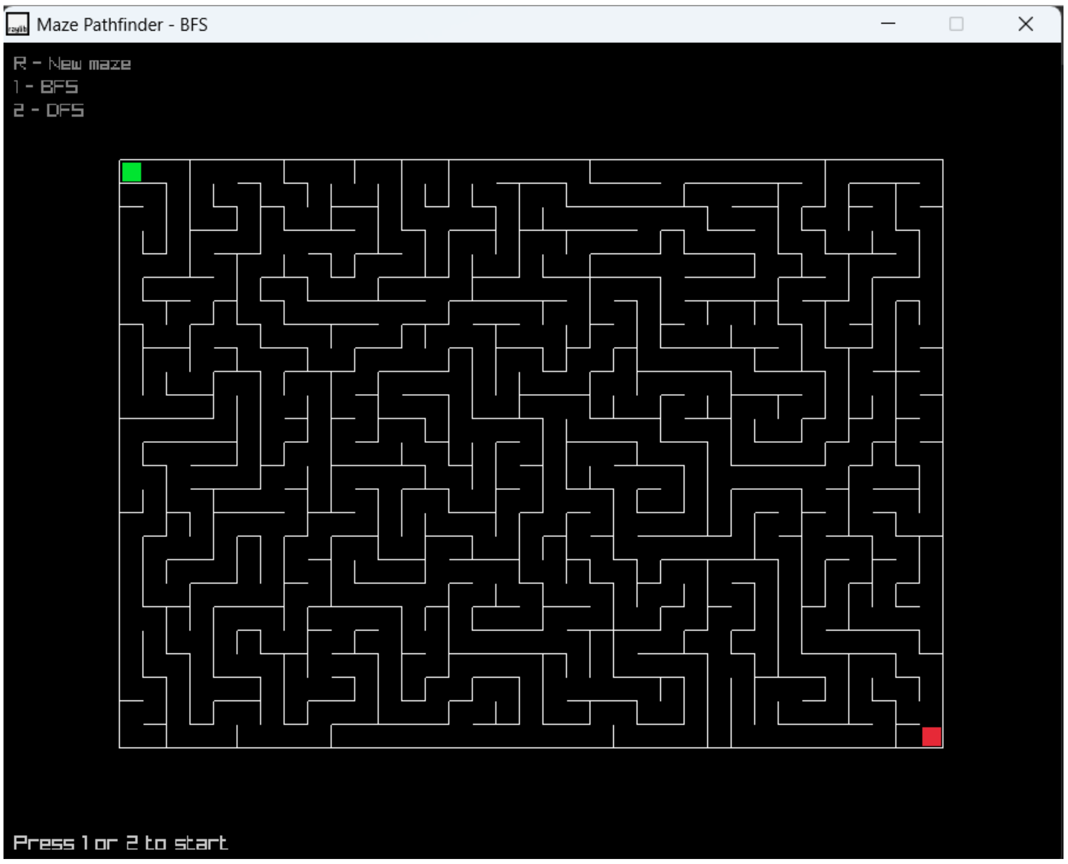


Figure 7.1: Main screen showing a randomly generated maze

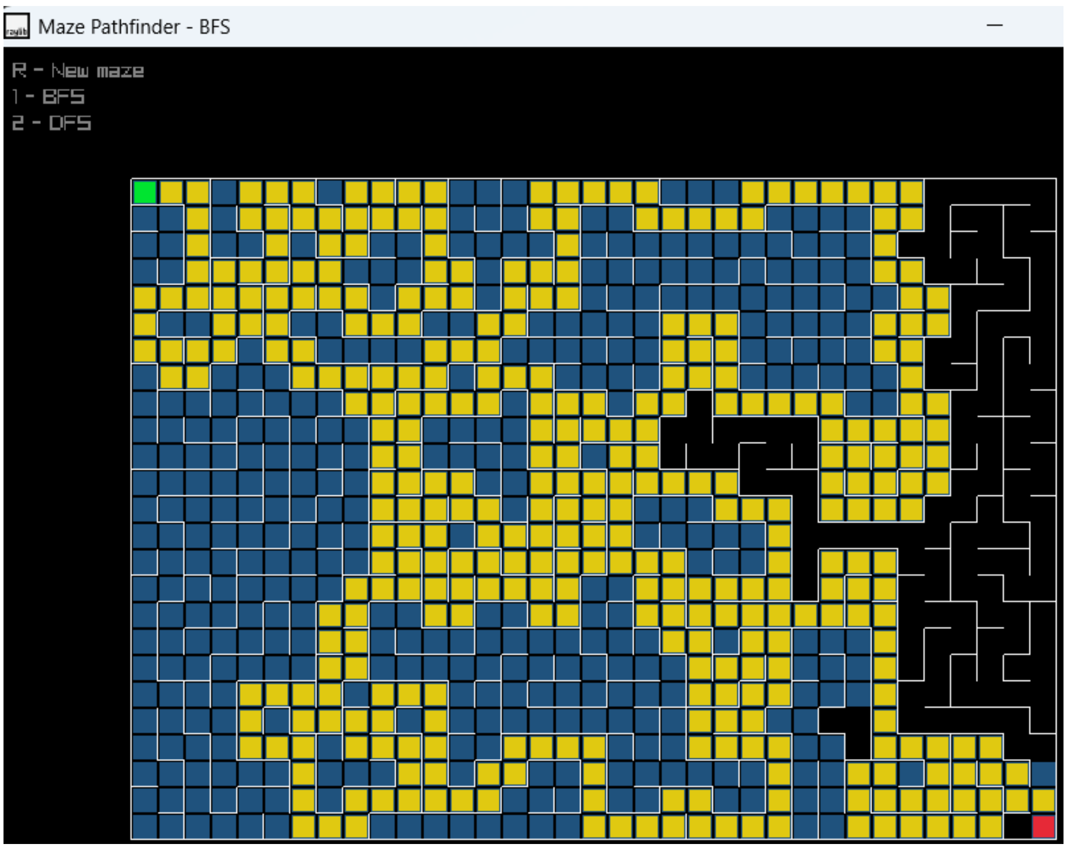
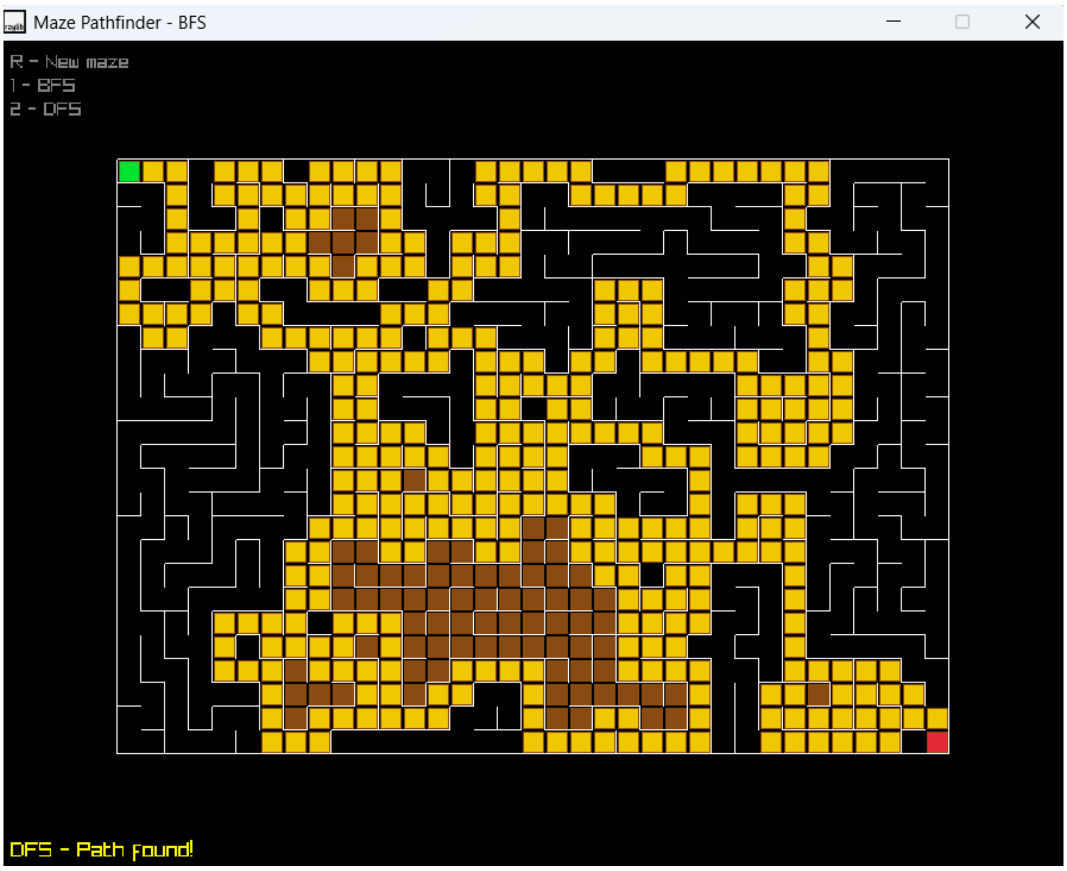


Figure 7.2: Shortest path found using Breadth-First Search (BFS)


Maze Pathfinder - BFS

R - New maze
1 - BFS
2 - DFS

DFS - Path found!

Figure 7.3: Path found using Depth-First Search (DFS)