PROJECT REPORT ON

The Maze Runner Autonomous Robot Analysis Suite

ASSINGMENT
Of
CODING SKILLS - I

(B. TECH CSE – 'D')

Submitted by: -

MOHAMMAD ALI	–	AP24110010311
SACHIN KUMAR	–	AP24110010317
ADITYA KUMAR	–	AP24110010333
SANKALP TIWARI	–	AP24110010356
PRADYUN	–	AP24110010325

Submitted on: - 08 December 2025

Prepared in the partial fulfillment of the
Project Based Learning of Course CSE 201 – CODING SKILLS I

SRM UNIVERSITY, AP

DECEMBER 2025

CERTIFICATE

This is to certify that the project entitled **“The Maze Runner – Autonomous Robot Pathfinding Simulation”** has been successfully completed by **Mohammad Ali (AP24110010311), Sachin Kumar (AP24110010317), Pradyun Nimaje (AP24110010325), Sankalp Tiwari (AP24110010356),** and as a required academic component of **CSE 201**: **Coding Skills 1** during Semester 3 of the Academic Year 2025–26 in the Department of Computer Science and Engineering, SRM University–AP.

This work was carried out under the guidance of the undersigned and has satisfactorily met all course requirements as of 08/12/2025.

(Signature of the Course Instructor)

Mr. Prakash,
Course Instructor (Campus Corporate Connect),
Department of Computer Science and Engineering
SRM University, AP

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to **Dr. T. R. Paarivendhar**, Chancellor, SRM University–AP, and **Prof. Ch. Satish Kumar**, Vice-Chancellor (In-Charge), SRM University–AP, for providing an excellent academic environment that encourages innovation and experiential learning.

I extend my heartfelt thanks to **Mr. Prakash, Course Instructor (Campus Corporate Connect),** Department of Computer Science and Engineering, SRM University–AP, for their invaluable guidance, continuous support, and insightful feedback throughout the development of our project titled **"The Maze Runner – Autonomous Robot Pathfinding Simulation."** Their mentorship was instrumental in helping us understand and implement algorithmic concepts such as maze generation, Depth-First Search (DFS), and Breadth-First Search (BFS), and apply them effectively in a practical simulation environment.

I also express my gratitude to the **Dean, School of Engineering and Sciences**, and the **Head of the Department, Computer Science and Engineering**, for their constant academic encouragement and support.

Finally, we would like to thank our peers and friends for their cooperation and motivation during the execution of this project. This work, prepared in partial fulfilment of the Project-Based Learning component of the course **CSE 201 – Coding Skills 1**, has been collaboratively completed by:

• **Mohammad Ali (AP24110010311)**
• **Sachin Kumar (AP24110010317)**
• **Pradyun Nimaje (AP24110010325)**
• **Sankalp Tiwari (AP24110010356)**

Their collective effort and teamwork contributed significantly to the successful completion of this report.

ABSTRACT

This report presents the successful development and implementation of the **“Maze Runner – Autonomous Robot Pathfinding Simulation”** project. This work was collaboratively undertaken by the student team comprising **Mohammad Ali (AP24110010311), Sachin Kumar (AP24110010317), Pradyun Nimaje (AP24110010325), and Sankalp Tiwari (AP24110010356)**. The project was completed under the expert supervision of **Mr. Prakash, Course Instructor (Campus Corporate Connect)**, Department of Computer Science and Engineering, SRM University–AP.

This initiative was executed as a key component of the **Project-Based Learning (PBL)** requirement for the course **CSE 201 – Coding Skills 1** during the Academic Year **2025–2026**. The primary objective of this project is to design and implement a simulation that generates random mazes and enables an autonomous virtual robot to navigate them using classical pathfinding algorithms. The system tackles the computational challenge of maze traversal by integrating **Recursive Backtracking** for maze generation, **Depth-First Search (DFS)** for exploratory navigation, and **Breadth-First Search (BFS)** for shortest-path discovery. These algorithms collectively demonstrate fundamental concepts of recursion, search strategies, and grid-based problem solving.

The project highlights the practical application of algorithmic design and structured programming to create an interactive visual simulation. Through real-time console visualization and comparative performance analysis, the work successfully reinforces core learning outcomes related to algorithm behavior, traversal efficiency, and data structure utilization. This project not only fulfills the course requirements but also reflects the experiential learning philosophy of SRM University–AP.

TABLE OF CONTENTS

Contents

1. Introduction

This report presents a detailed account of the design, development, and implementation of the project titled **"The Maze Runner – Autonomous Robot Pathfinding Simulation."** The project was undertaken as part of the **Project-Based Learning (PBL)** curriculum for the course **CSE 201 – Coding Skills 1**.

The system was collaboratively developed by the student team—**Mohammad Ali, Sachin Kumar, Pradyun Nimaje, and Sankalp Tiwari**—and successfully completed in **December 2025** under the guidance of **Mr. Prakash Sir**, Department of Computer Science and Engineering, SRM University–AP.

The primary aim of this project is to explore fundamental algorithmic techniques by simulating autonomous robot navigation inside a computer-generated maze. The system integrates three major components:

- **Random Maze Generation** using recursive backtracking,
- **Depth-First Search (DFS)** for exploratory navigation with backtracking, and
- **Breadth-First Search (BFS)** for shortest-path discovery through level-order traversal.

This project provides a practical environment to analyze, compare, and visualize classical search strategies, reinforcing essential concepts in algorithm design and problem solving.

1.1 Problem Statement

Navigating through a maze is a classic computational problem that demonstrates the need for efficient search strategies and algorithmic decision-making. The challenge lies in enabling a robot to autonomously traverse a complex grid of walls and passages, starting from a fixed entry point and reaching the designated exit while avoiding dead ends and unnecessary exploration.

The problem addressed in this project is to generate a random, solvable maze and design a simulation that allows a virtual robot to find a path through it using two fundamental search algorithms—Depth-First Search (DFS) and Breadth-First Search (BFS). The system must:

- Construct a valid maze using a reliable, algorithmic generation method.
- Navigate the maze using DFS, illustrating deep exploration and backtracking.
- Navigate the maze using BFS, demonstrating systematic, layer-by-layer search and shortest-path identification.
- Provide real-time visual feedback so that the behavior of each algorithm can be observed and analyzed.
- Compare both algorithms based on traversal steps and search efficiency.

The project must be completed within the academic timeframe of Semester 3 (AY 2025–26) as part of the CSE 201 – Coding Skills 1 coursework, fulfilling the Project-Based Learning requirement.
This problem statement reflects the need to transform theoretical learning into a practical, working simulation that deepens the understanding of algorithm design, problem-solving techniques, and performance evaluation.

1.2 Project Objectives

The primary objectives of this project are centered around understanding and applying fundamental algorithmic strategies to solve a structured pathfinding problem. The system aims to simulate the behavior of a robot navigating through a randomly generated maze using two classical search algorithms. The key objectives are:

• To generate a random and solvable maze
Design and implement a maze-generation algorithm using recursive backtracking that produces unique maze structures while ensuring the existence of a valid path between the start and end points.

• To implement Depth-First Search (DFS) for exploratory navigation
Develop a recursive DFS-based solver that demonstrates deep-path exploration, backtracking, and state marking, enabling the robot to traverse the maze even through complex dead-end patterns.

• To implement Breadth-First Search (BFS) for shortest-path discovery
Construct a BFS solver using a queue-based approach that systematically explores the maze layer by layer and guarantees the shortest path to the destination.

• To visualize algorithmic behavior in real time
Integrate an interactive console visualization that updates the maze dynamically, allowing users to clearly observe robot movement, visited paths, decision-making, and algorithm behavior.

• To compare performance characteristics of DFS and BFS
Analyze and record the number of nodes or steps explored by each algorithm, highlighting differences in efficiency, traversal pattern, memory usage, and optimality.

• To apply structured problem-solving and modular design
Organize the project into clearly defined functional modules, ensuring readability, maintainability, and demonstrating systematic application of programming and algorithmic concepts.

2. Project Layout/System Architecture

The architecture of the **Maze Runner – Autonomous Robot Pathfinding Simulation** is designed to provide a structured and modular framework for generating mazes, executing search algorithms, and visualizing robot navigation. The system is organized into distinct functional layers, each responsible for a critical aspect of the overall simulation. This modular architecture ensures clarity, maintainability, and ease of testing.

The system follows a **linear execution flow**, starting from maze creation, progressing through algorithmic navigation, and concluding with performance analysis. Each module interacts through well-defined data structures and shared global arrays, enabling efficient communication and state management.

2.1 Architectural Components

The system architecture consists of the following core components:

2.1.1. Maze Representation Layer

- Stores the maze as a **2D grid** of characters.
- Distinguishes between walls, paths, visited cells, start point, end point, and solution path.
- Uses two arrays:

  maze[][]– active maze used during algorithms

  originalMaze [][] – preserved copy for resetting before BFS
- Provides the foundational data structure required for all other modules.

2.1.2 Maze Generation Module

- Implements **recursive backtracking** to carve tunnels in a grid of walls.
- Randomizes direction order to ensure unique maze patterns.
- Ensures maze solvability and proper boundary walls.
- Assigns **S** (Start) and **E** (End) points automatically.

This module forms the first stage of the project workflow.

2.1.3 DFS Navigation Module

- Utilizes **Depth-First Search** with recursion and backtracking.
- Explores one direction deeply before reconsidering alternatives.
- Marks each step with symbols (R, ., *) to visualize robot movement.

- Constructs the solution path when the end cell is reached.

This module demonstrates exploratory searching and recursive problem solving.

2.1.4 BFS Navigation Module

- Uses a **queue** to explore the maze level by level.
- Guarantees the shortest path in terms of number of steps.
- Maintains a visited[][] array to prevent redundant processing.
- Updates the maze display periodically for real-time visualization.

This module contrasts the behavior of DFS and showcases systematic, optimal search.

2.1.5 Visualization & Animation Layer

- Responsible for rendering the maze and robot movement in the console.
- Uses:
    - ANSI color codes
    - timed refresh (SLEEP_MS())
    - screen-clearing commands (system(CLEAR_SCREEN))
- Allows users to observe algorithm decisions and traversal patterns dynamically.

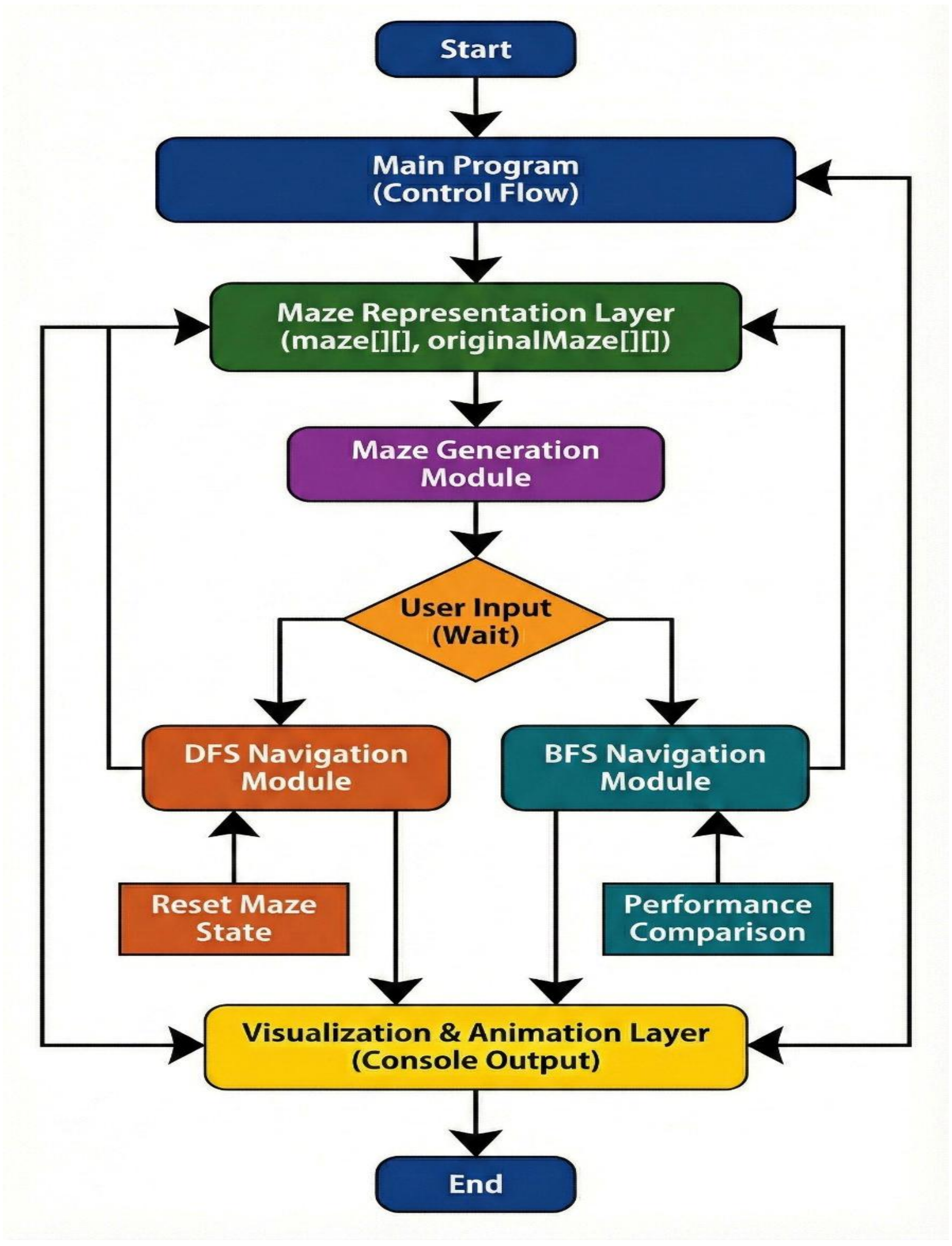
2.1.6 Control Flow (Main Program Layer)

The main() function orchestrates the execution sequence:

1. Initialize and generate the maze
2. Wait for user input
3. Run DFS solver and display results
4. Reset maze state
5. Run BFS solver and display results
6. Print final performance comparison

This layer integrates all modules into a single cohesive workflow.

2.2. Flow Chart


Start
Main Program (Control Flow)
Maze Representation Layer (maze[][], originalMaze[][])
Maze Generation Module
User Input (Wait)
DFS Navigation Module
BFS Navigation Module
Reset Maze State
Performance Comparison
Visualization & Animation Layer (Console Output)
End

3. Modules Explanation

The Maze Runner system is organized into multiple functional modules, each responsible for a specific stage of the simulation—from maze construction to algorithmic navigation and visualization. This modular approach enhances readability, maintainability, and allows each component to be independently understood and improved. The following subsections provide a detailed explanation of each module implemented in the project.

3.1 Maze Representation Module

This module provides the foundational data structure for the entire system. Key elements include:

- A 2D character array **maze[ROWS][COLS]** that stores the current state of the maze.
- A backup array **originalMaze[ROWS][COLS]** used to reset the maze before running BFS.
- Symbolic constants such as '#' for walls, ' ' for paths, 'S' for start, 'E' for end, 'R' for the robot, '.' for visited cells, and '*' for the final solution path.

This module ensures consistent representation of maze structure and enables other modules to operate on a unified grid model.

3.2 Maze Generation Module

This module is responsible for constructing a **random, solvable maze** using a **recursive backtracking algorithm**.

Key Steps:

1. Start with a grid filled entirely with walls.
2. Recursively carve paths by:
    - Moving two steps at a time,
    - Breaking intermediate walls,
    - Randomizing direction order to ensure unique maze patterns.
3. Assign the start (S) and end (E) positions.
4. Store a copy of the maze for later use in BFS.

This module guarantees that every execution produces a unique maze while maintaining solvability.

3.3 DFS Navigation Module

This module implements **Depth-First Search (DFS)** to simulate a robot navigating through the maze using recursion and backtracking.

Core Functionalities:

- Recursively explore one direction at a time.
- Mark the robot's movement using 'R', visited dead-ends using '.', and the final solution path using '*'.
- Backtrack when dead-ends are encountered.
- Terminate successfully when the robot reaches the end point.

This module demonstrates deep exploration behavior and highlights DFS's inability to guarantee the shortest path.

3.4 BFS Navigation Module

This module implements **Breadth-First Search (BFS)** to find the shortest path from the start to the end.

Major Components:

- A queue (Node queue[]) to manage level-order exploration.
- A visited[][] matrix to prevent revisiting cells.
- Retrieval and expansion of nodes until the target is reached.
- Periodic visualization updates after every few steps.

This module illustrates BFS's ability to find the shortest path and contrasts its behavior with DFS.

3.5 Visualization & Animation Module

This module handles real-time visual updates of the maze and robot movement.

Responsibilities:

- Clear and redraw the console using platform-specific commands (cls or clear).
- Display walls, paths, start/end markers, visited nodes, and robot position using ANSI colors.
- Introduce timed delays (SLEEP_MS()) to create smooth animation.
- Show exploration patterns for both DFS and BFS clearly.

By making algorithm execution visible, this module transforms theoretical concepts into intuitive visual behavior.

3.6 Performance Analysis Module

At the end of the simulation, this module compares DFS and BFS based on:

- Total steps or nodes explored
- Efficiency of search patterns

- Ability to find the shortest path
- Memory usage characteristics (stack vs queue)

A summary report is printed to help understand the practical differences between the two algorithms.

Summary of Modules

Module	Purpose
Maze Representation	Stores maze structure and maintains start/end points
Maze Generation	Creates random, solvable mazes using recursion
DFS Navigation	Performs deep, recursive traversal with backtracking
BFS Navigation	Finds shortest path using queue-based traversal
Visualization	Animates robot movement and algorithm behavior
Performance Analysis	Compares DFS and BFS in metrics and behavior

4. Features Implemented

The Maze Runner simulation integrates multiple functional and algorithmic components to create a complete, interactive environment for maze generation and pathfinding. The following features were successfully implemented as part of the project:

4.1 Random Maze Generation

- A dynamic maze is generated at runtime using a **recursive backtracking algorithm**.
- Ensures unique maze structures in every execution due to randomized direction ordering.
- Maintains solvability by carving a continuous path between the start and end points.
- Assigns the **Start (S)** and **End (E)** positions automatically within the maze.

4.2 Depth-First Search (DFS) Navigation

- Implements DFS to simulate a robot exploring the maze through **deep, recursive traversal**.
- Supports backtracking when dead ends are encountered.
- Marks cells as:
  - **R** → robot's current position
  - **.** → visited/dead-end
  - ***** → final solution path

- Provides step counting through the **dfs_steps** variable.

4.3 Breadth-First Search (BFS) Navigation

- Implements BFS using a **queue-based structure** for systematic exploration.
- Guarantees discovery of the **shortest path** in terms of number of steps.
- Uses a **visited matrix** to prevent reprocessing the same cell.
- Periodically updates the maze display to show exploration progress.
- Tracks the number of visited nodes via the **bfs_steps** counter.

4.4 Real-Time Console Visualization

- Animated display using platform-specific screen clearing (cls/clear).
- Color-coded maze elements for enhanced readability:
  - Walls
  - Paths
  - Start/End nodes
  - Robot
  - Solution path
- Smooth animation achieved using timed delays (SLEEP_MS).
- Allows users to clearly observe algorithmic behavior step-by-step.

4.5 Maze Reset Mechanism

- Before running BFS, the original maze is restored using **resetMaze()**.
- Ensures DFS and BFS operate on identical maze conditions for fair comparison.
- Preserves visual integrity and avoids interference from DFS markings.

4.6 Algorithm Performance Report

- At the end of execution, the system prints a detailed comparison of:
  - Total steps explored by DFS
  - Total nodes visited by BFS
  - Strengths and limitations of each algorithm
- Helps users understand the **trade-offs** between deep-first exploration and level-order search.

4.7 User Interaction Support

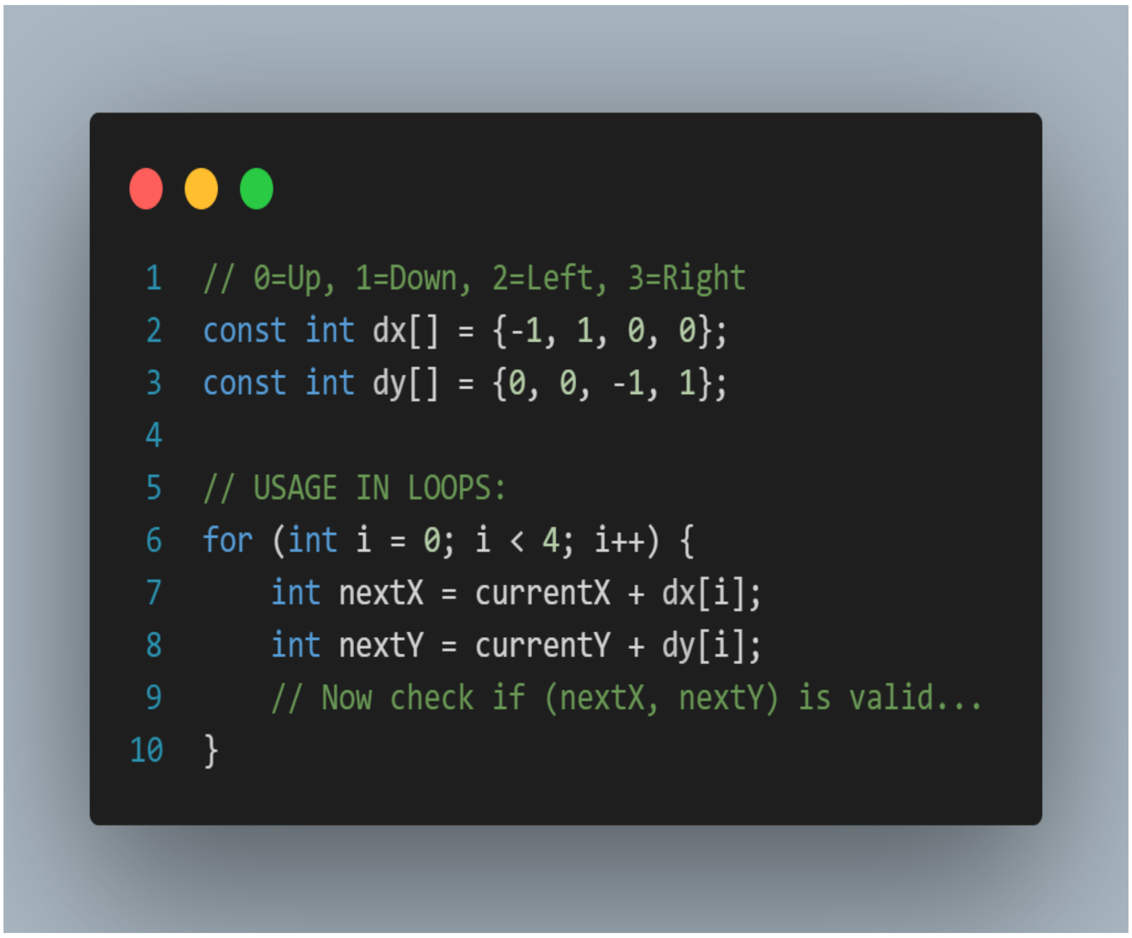
- User-triggered execution phases using **ENTER** key prompts.
- Allows controlled sequential progression:
    1. Generate maze
    2. Run DFS
    3. Reset maze
    4. Run BFS
    5. Display report

This ensures a smooth and understandable simulation workflow

5. Code Snippets

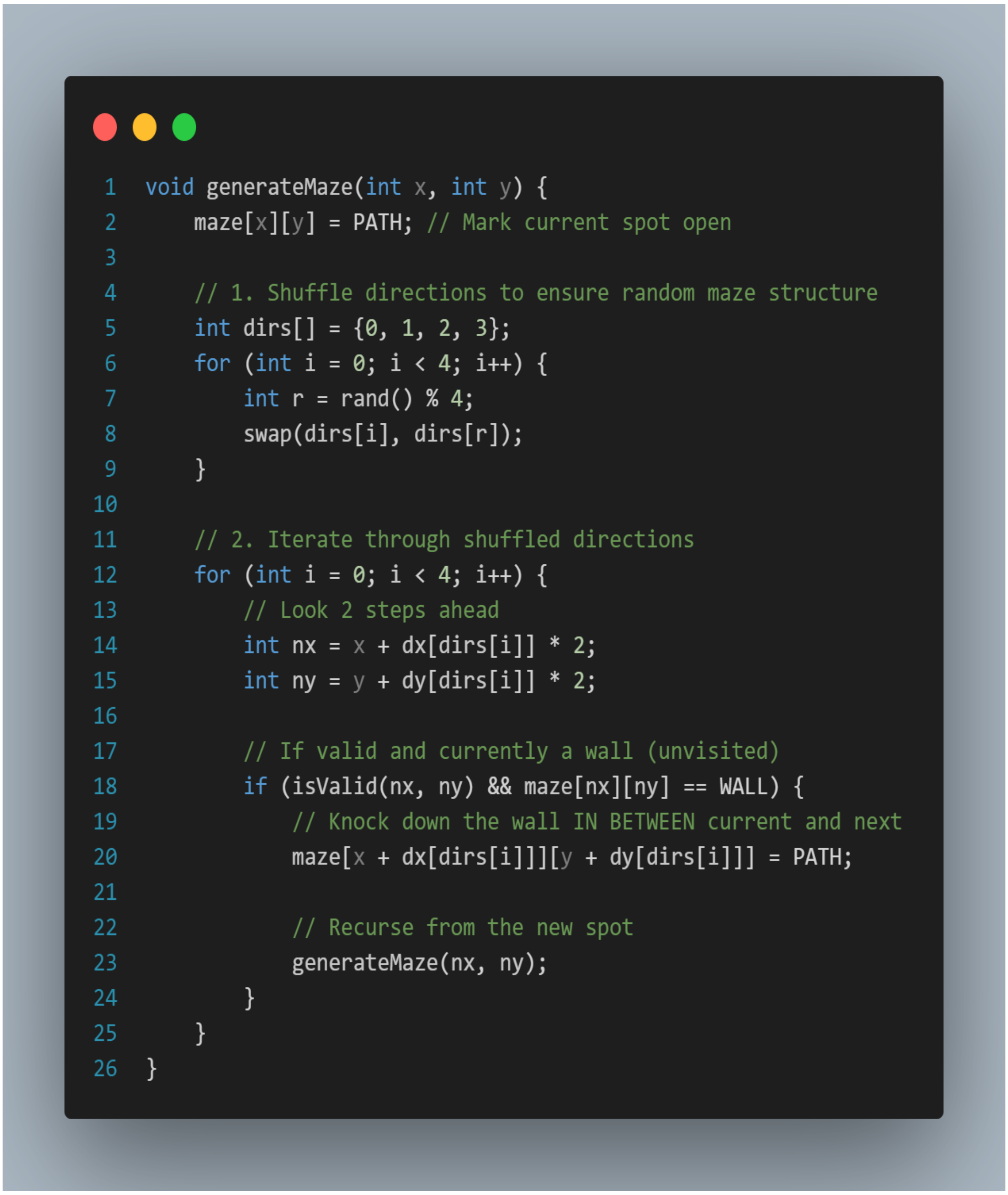
Code Snippet 1: 1. The "Compass" (Direction Arrays)

Instead of writing four separate if statements (for Up, Down, Left, Right), we use arrays to mathematically calculate neighbor coordinates.



```
// 0=Up, 1=Down, 2=Left, 3=Right
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};

// USAGE IN LOOPS:
for (int i = 0; i < 4; i++) {
    int nextX = currentX + dx[i];
    int nextY = currentY + dy[i];
    // Now check if (nextX, nextY) is valid...
}
```

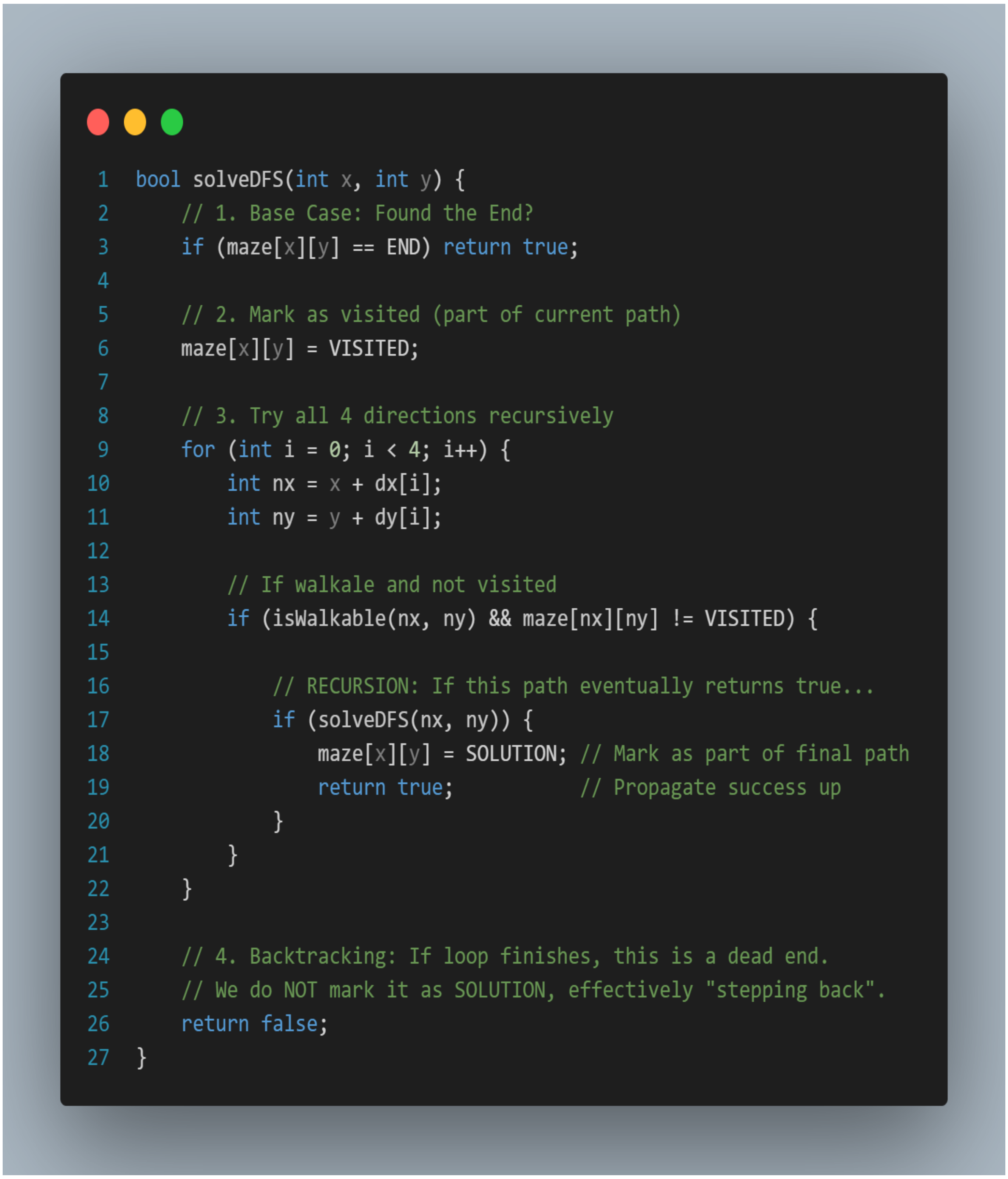
Code Snippet 2: Maze Generation (Randomized Prim's / Recursive Backtracking)
Logic: Start at a cell, pick a random direction, jump two steps (to leave a wall thickness), knock down the wall in between, and recurse.



```
void generateMaze(int x, int y) {
    maze[x][y] = PATH; // Mark current spot open

    // 1. Shuffle directions to ensure random maze structure
    int dirs[] = {0, 1, 2, 3};
    for (int i = 0; i < 4; i++) {
        int r = rand() % 4;
        swap(dirs[i], dirs[r]);
    }

    // 2. Iterate through shuffled directions
    for (int i = 0; i < 4; i++) {
        // Look 2 steps ahead
        int nx = x + dx[dirs[i]] * 2;
        int ny = y + dy[dirs[i]] * 2;

        // If valid and currently a wall (unvisited)
        if (isValid(nx, ny) && maze[nx][ny] == WALL) {
            // Knock down the wall IN BETWEEN current and next
            maze[x + dx[dirs[i]]][y + dy[dirs[i]]] = PATH;

            // Recurse from the new spot
            generateMaze(nx, ny);
        }
    }
}
```

3. DFS Solver (The "Stack" Approach)

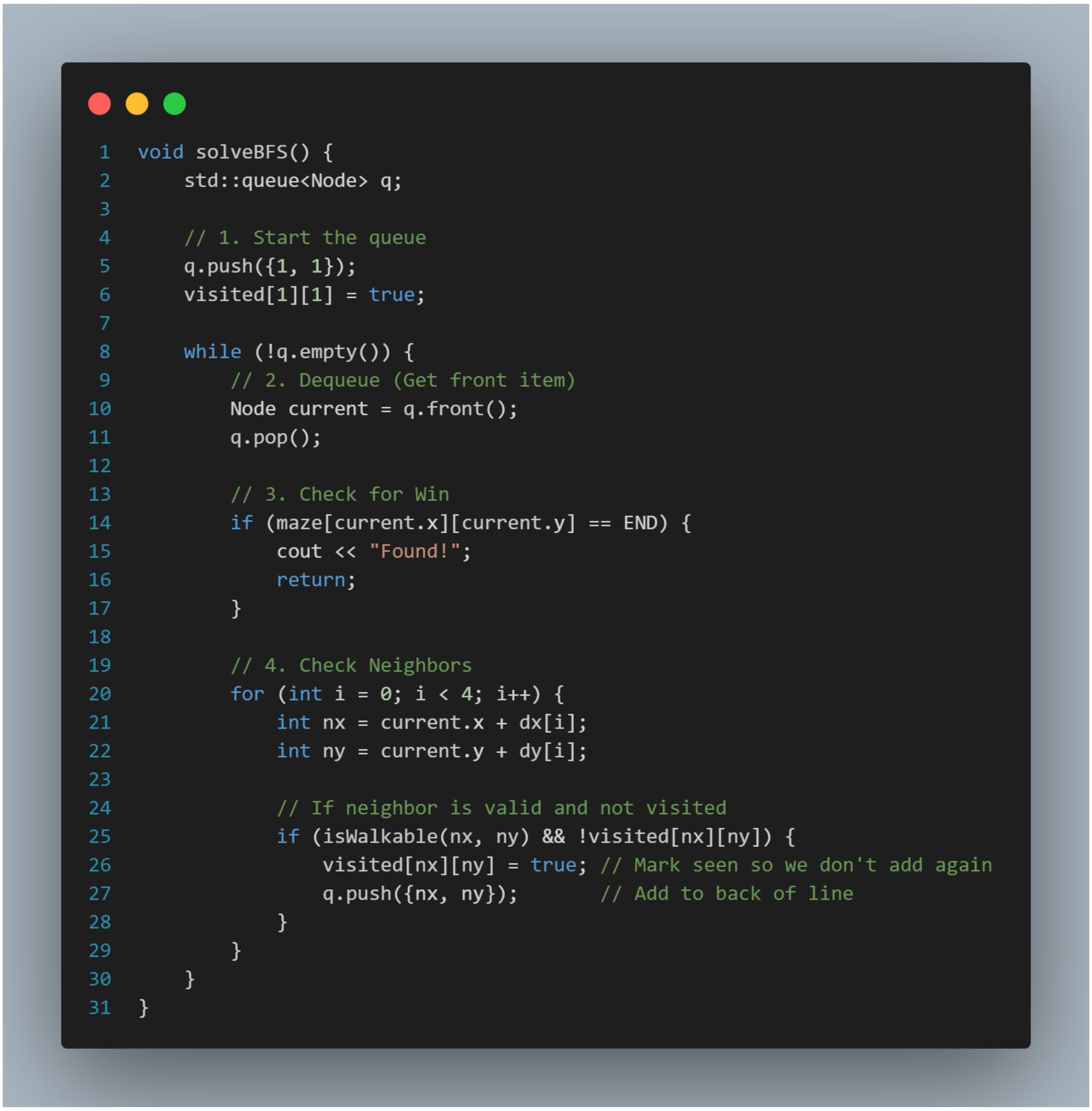
Logic: Go as deep as possible. If you hit a dead end, return false and backtrack (step back) to the previous junction.



```
bool solveDFS(int x, int y) {
    // 1. Base Case: Found the End?
    if (maze[x][y] == END) return true;

    // 2. Mark as visited (part of current path)
    maze[x][y] = VISITED;

    // 3. Try all 4 directions recursively
    for (int i = 0; i < 4; i++) {
        int nx = x + dx[i];
        int ny = y + dy[i];

        // If walkale and not visited
        if (isWalkable(nx, ny) && maze[nx][ny] != VISITED) {

            // RECURSION: If this path eventually returns true...
            if (solveDFS(nx, ny)) {
                maze[x][y] = SOLUTION; // Mark as part of final path
                return true;           // Propagate success up
            }
        }
    }

    // 4. Backtracking: If loop finishes, this is a dead end.
    // We do NOT mark it as SOLUTION, effectively "stepping back".
    return false;
}
```

4. BFS Solver (The "Queue" Approach)

Logic: Add the starting node to a Queue. While the Queue isn't empty, take the front item, look at its neighbors, and add unvisited neighbors to the back of the Queue. This creates a "ripple" effect.



```cpp
void solveBFS() {
    std::queue<Node> q;

    // 1. Start the queue
    q.push({1, 1});
    visited[1][1] = true;

    while (!q.empty()) {
        // 2. Dequeue (Get front item)
        Node current = q.front();
        q.pop();

        // 3. Check for Win
        if (maze[current.x][current.y] == END) {
            cout << "Found!";
            return;
        }

        // 4. Check Neighbors
        for (int i = 0; i < 4; i++) {
            int nx = current.x + dx[i];
            int ny = current.y + dy[i];

            // If neighbor is valid and not visited
            if (isWalkable(nx, ny) && !visited[nx][ny]) {
                visited[nx][ny] = true; // Mark seen so we don't add again
                q.push({nx, ny});       // Add to back of line
            }
        }
    }
}
```

6.Conclusion

The **Maze Runner – Autonomous Robot Pathfinding Simulation** successfully demonstrates how classical search algorithms operate within a dynamically generated maze environment. By integrating **Recursive Backtracking**, **Depth-First Search (DFS)**, and **Breadth-First Search (BFS)** into a unified simulation, the project provides a practical and visual understanding of algorithmic behavior, traversal strategies, and problem-solving methodologies.

The system effectively showcases the contrasting characteristics of DFS and BFS. DFS illustrates deep exploration and backtracking, often leading to longer or non-optimal paths, whereas BFS consistently identifies the shortest route by exploring the maze layer by layer. The animated console visualization further enhances comprehension by making each algorithm's decision-making process transparent and intuitive.

Through modular design, structured implementation, and real-time visualization, this project bridges the gap between theoretical concepts and hands-on application. It reinforces essential ideas in recursion, queue-based traversal, grid navigation, and algorithmic efficiency. Moreover, the final performance report provides meaningful insights into the trade-offs between search depth, optimality, and computational overhead.

Overall, the simulation fulfills its intended objectives and serves as an effective educational tool, offering clear, practical exposure to fundamental pathfinding algorithms and their real-world implications

7. Future Enhancements

While the current version of the Maze Runner simulation effectively demonstrates maze generation and pathfinding using DFS and BFS, several enhancements can further improve its functionality, performance, and user experience. Potential future developments include:

1. Shortest Path Reconstruction in BFS

Although BFS finds the shortest route, the program currently visualizes only the exploration process. A future enhancement could include:

- Storing parent nodes during BFS
- Drawing the exact shortest path using a dedicated symbol
- Providing path length and reconstruction statistics

This would make the comparison between DFS and BFS more comprehensive.

2. Variable Maze Sizes and Difficulty Levels

Allowing users to choose:

- Maze dimensions (small, medium, large)
- Difficulty settings (density of walls, branching complexity)

would make the simulation more interactive and adaptable to different learning goals.

3. Implementation of Additional Algorithms

The system can be extended by integrating more advanced pathfinding algorithms, such as:

- **A*** (A-Star Search)
- **Dijkstra's Algorithm**
- **Greedy Best-First Search**

These additions would showcase heuristic-based navigation and enable deeper algorithmic comparison.

4. Graphical User Interface (GUI)

Replacing the console-based display with a GUI using tools such as SDL, OpenGL, or a simple desktop framework would bring:

- Smooth animations
- Visual controls (Start, Pause, Speed Control)
- Better maze representation

This would significantly enhance user engagement and visualization clarity.

5. Real-Time Speed Control & Step-by-Step Mode

Introducing:

- Adjustable animation speed
- A *step-by-step exploration mode*
- Ability to pause and inspect algorithm state

would make the simulation a more powerful teaching tool.

8. Reference

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). **Introduction to Algorithms (3rd Edition)**. MIT Press.

   Referenced for theoretical foundations of DFS, BFS, and graph traversal algorithms.
2. Sedgewick, R., & Wayne, K. (2011). **Algorithms (4th Edition)**. Addison-Wesley.

   Used for understanding maze generation techniques and recursive backtracking.
3. Lafore, R. (2017). **Data Structures and Algorithms in C++**. Pearson Education.

   Referred for conceptual clarity on recursion, queues, and grid-based problem solving.
4. Patel, R. (2010). **Programming in C**. McGraw-Hill.

   Consulted for syntax, memory management, and modular program structure in C.
5. Online Resource: **GeeksforGeeks – Graph and Maze Algorithms**.

   Retrieved from https://www.geeksforgeeks.org

   Used for reference on BFS, DFS, and algorithm visualization concepts