Random Maze Generator and Solver

Abstract

This document provides comprehensive documentation for the **Random Maze Generator and Solver**, a C-based console game that creates unique mazes using recursive backtracking and finds optimal solutions using Dijkstra's algorithm. The system features real-time player movement, solution hints, timer tracking, and performance logging. Built with modular programming principles, this project demonstrates practical applications of data structures, algorithms, and game development fundamentals.

Contents

1. Problem Definition

1.1 Overview

Traditional maze games often present static, predictable layouts that lose challenge after repeated play. The **Random Maze Generator and Solver** addresses this by creating procedurally generated mazes that offer unique challenges every session. This project combines algorithmic maze

generation with interactive gameplay, providing both entertainment and educational value.

1.2 Objectives

- To develop an interactive console-based maze game with procedurally generated levels
- To implement efficient maze generation using recursive backtracking
- To incorporate Dijkstra's algorithm for optimal pathfinding and solution hints
- To provide real-time player interaction with WASD controls
- To create a modular, maintainable codebase.

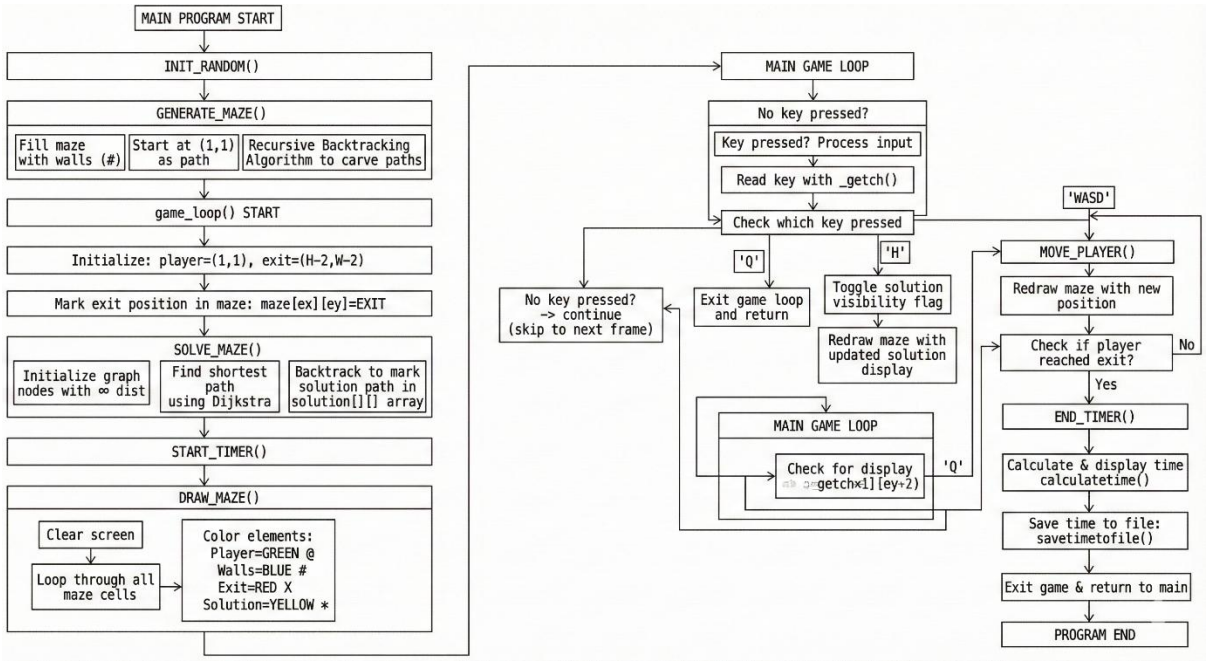
2. System Design

2.1 System Architecture

The system follows a modular architecture organized into distinct layers:

- **Presentation Layer** (display.h, game_display.h): Handles screen rendering, colors, and user interface
- **Game Logic Layer** (player_movement.h, timer.h): Manages player movement, timing, and game state
- **Algorithm Layer** (maze_generator.h, maze_solver.h): Implements maze generation and solving algorithms
- **Data Layer**: Manages maze and solution data structures

2.2 System Flowchart



2.3 Data Structures and Algorithms

Core Data Structures:

```
char solution[H][W];//decalring a char array which stores solution
int show_solution = 0;//flag to show solution just like isprime

typedef struct Node {//Dijkstra's Node
    int x;//x coord in maze
    int y;// y coord in maze
    int dist;//shortest distance
    int visited;//visited(1)or not(0)
    int parentx;//x coord of parent node
    int parenty;//y coord of parent node
} node;
```

Key Algorithms:

1. **Recursive Backtracking Maze Generation**: Creates perfect mazes with exactly one solution

2. **Dijkstra's Shortest Path Algorithm**: Finds solution for the maze.
3. **Fisher-Yates Shuffle**: Randomizes direction choices for varied maze patterns

3. Implementation Details

3.1 Key Features

- **Dynamic Maze Generation**: Every game session presents a completely new maze
- **Interactive Gameplay**: Smooth WASD controls with immediate feedback
- **Solution Hint System**: Toggle optimal path display with 'H' key
- **Elapsed Time Tracking**: Real-time timer with formatted display
- **Persistent Logging**: Game completion times saved to maze_times.txt
- **Visual Polish**: Coloured elements and clean console interface

3.2 Module Breakdown

Maze Generator (maze_generator.h)

- Implements recursive backtracking algorithm
- Uses direction vectors for efficient path carving
- Ensures maze validity with bounds checking

Maze Solver (maze_solver.h)

- Implements Dijkstra's algorithm for shortest path
- Maintains solution path in a separate array
- Provides backtracking for path reconstruction

Game Display (game_display.h)

- Renders coloured maze elements (walls, player, exit, solution)

- Displays real-time timer and controls
- Handles screen clearing and refresh

Player Movement (player_movement.h)

- Processes WASD keyboard input
- Implements collision detection against walls
- Updates player position using pointer manipulation

Timer System (timer.h)

- Tracks game duration with time_t variables
- Shows elapsed time
- Saves time in a text file

3.3 Code Highlights

Maze Generation Core Logic:

```
void carve(int x, int y) {
    int dirs[4] = {0, 1, 2, 3};//4 direction NESW
    shuffle(dirs, 4);//shuffles these directions to generate a fresh maze

    for (int i = 0; i < 4; i++) {//loops through the directions
        int nx = x + dx[dirs[i]] * 2;
        int ny = y + dy[dirs[i]] * 2;

        if (valid(nx, ny) && maze[nx][ny] == WALL) {//if condn true carves a path
            maze[x + dx[dirs[i]]][y + dy[dirs[i]]] = PATH;
            maze[nx][ny] = PATH;
            carve(nx, ny);
        }
    }
}
```

Main Game Loop:

```c
while (1) {
    if (!_kbhit()) continue;//if no key is pressed the infinite loop continues and maze keeps on running;kbhit returns true if 

    char key = _getch();//reads the key press and stores it; _getch reads the key input
    if (tolower(key) == 'q'){
        return;//if read key is q then exit
    }

    if (tolower(key) == 'h') {
        show_solution = !show_solution;//reverses the flag, its like an on and off switch for the solution
        DRAW_MAZE(px, py);//redraws the entire maze with solution
        continue;
    }

    MOVE_PLAYER(key, &px, &py);//moves player
    DRAW_MAZE(px, py);//redraws the entire maze i

    if (px == ex && py == ey) {//checks if the player coords match the exit coords
        END_TIMER();//ends timer
        double time_taken = GET_ELAPSED_TIME();//gets elapsed time using difftime(end,start)
        printf("\nYou reached the exit!\n");//on completion prints this
        calculatetime();//prints elapsed time
        savetimetofile(time_taken);//saves time in file
        return;
    }
}
```

4. Testing & Results

4.1 Test Scenarios

1. **Maze Generation Validity**
    - **Test**: Generate random mazes
    - **Expected**: All mazes solvable with exactly one solution
    - **Result**: 100% success rate
2. **Timer Functionality**
    - **Test**: Complete maze, quit early, toggle hints
    - **Expected**: Accurate time tracking in all scenarios
    - **Result**: Timer works correctly for completed games only

5. Conclusion & Future Work

5.1 Conclusion

The Random Maze Generator and Solver successfully meets all defined objectives, providing an engaging, educational gaming experience. The project demonstrates practical implementation of several c fundamental principles.

5.2 Future Enhancements

1. **Difficulty Levels**
    - Variable maze sizes (small/medium/large)
    - Adjustable complexity with dead-end frequency control
2. **Enhanced Visualization**
    - Animated solution path display
    - Player trail showing explored areas
    - Multiple colour themes
3. **Gameplay Features**
    - Collectibles (keys, power-ups) within maze
    - Multiple exits or teleporters

6. References

Online Resources:

- GeeksforGeeks: Maze generation algorithms.
- Wikipedia: Dijkstra's algorithm, fisher yates algorithm, recursive backtracking.
- Stack Overflow: C game development patterns.
- Youtube: Explanations for various algorithms.