SPL-1 Project Report

Maze Solver Game

Submitted By
Tarek Jamil Adnan
BSSE Roll No.: 1520
BSSE Session: 2022-23

Submitted To
Dr. Mohammed Shafiul Alam Khan
Professor

Institute of Information Technology

University of Dhaka

Institute of Information Technology
University of Dhaka

25-03-2025

Project Name

Maze Solver Game

Supervised by

Dr. Mohammed Shafiul Alam Khan
Professor

Institute of Information Technology

University of Dhaka

Supervisor Signature:

Submitted by

Tarek Jamil Adnan
BSSE Roll No.: 1520
BSSE Session: 2022-23

Signature:

Table of Contents

List of Tables

List of Figures

1. Introduction

The "Maze Solver Game" is a software project developed as part of the Software Project Lab-1 (SPL-1) course at the Institute of Information Technology, University of Dhaka. This project falls within the broad domain of interactive educational software and algorithmic visualization. It aims to create an engaging graphical user interface (GUI) application where users can navigate a maze manually or observe automated maze-solving algorithms in action. The primary objective is to demonstrate fundamental programming concepts such as data structures, algorithms, and GUI programming while providing an intuitive and educational experience.

The project combines a player-driven game mode with an automated solver mode, featuring multiple maze-solving algorithms (Breadth-First Search, Depth-First Search, and A*). It uses a grid-based maze structure and supports both keyboard and mouse interactions. The software is designed with simplicity and functionality in mind, making it an effective tool for learning and experimentation.

1.1 Significance of Maze-Solving Algorithms

Maze-solving algorithms form the intellectual backbone of this project, rooted in graph theory and search strategies. Breadth-First Search (BFS), as described by Cormen et al., systematically explores all nodes at each depth, ensuring the shortest path in unweighted graphs. Depth-First Search (DFS), conversely, prioritizes depth over breadth, offering a contrasting approach. The A* algorithm, introduced by Hart et al., enhances efficiency with heuristics like Manhattan distance, optimizing path-finding. These algorithms, widely studied in computer science curricula, provide a practical lens through which the project demonstrates their mechanics, drawing from decades of algorithmic research.

1.2 Evolution of GUI in Educational Tools

The graphical user interface (GUI) is central to the project's interactivity, reflecting advancements in educational software design. Early GUIs, such as those in Xerox's Alto , laid the groundwork for visual programming, evolving into tools like the BGI library used here . Studies by Sedgewick highlight how GUIs in algorithm visualization—e.g., animated sorting or graph traversal—enhance learning by making processes observable. The Maze Solver Game leverages this history, employing C/C++ with BGI to render grid-based mazes and dynamic paths, offering a modern take on a classic educational approach that balances simplicity with visual clarity.

1.3 Educational and Practical Objectives

The project's objectives are twofold: educational and technical. It aims to illustrate core programming concepts—data structures (e.g., queues for BFS, stacks for DFS), algorithms, and GUI programming—while providing an engaging learning experience. Historically, maze problems have been educational staples; for instance, Shannon's maze-solving machine demonstrated mechanical problem-solving, inspiring software analogs . Practically, the project supports keyboard and mouse inputs on a 16x21 grid, ensuring accessibility. By prioritizing simplicity and functionality, it mirrors minimalist educational tools like "Logo", aiming to empower users to experiment with and understand computational logic in an intuitive environment.

2. Background of the Project

This chapter outlines the theoretical and practical foundations that inform the "Maze Solver Game" project. It explores maze solving as a fundamental problem in computer science, the key concepts and algorithms involved, and the educational context that shapes its development. By building on established principles and tools, the project serves as a bridge between theoretical knowledge and interactive application, tailored for educational purposes at the Institute of Information Technology, University of Dhaka.

2.1 Maze Solving in Computer Science

Maze solving is a classic and enduring problem in computer science, centered on finding a viable path from a designated starting point to a goal within a structured grid or graph. This problem is frequently employed as a teaching mechanism to illustrate the application of algorithms and data structures, including stacks, queues, and graphs. According to Cormen et al. (2009), maze-solving algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) are foundational applications of graph traversal techniques, offering systematic methods to navigate complex environments [1]. These techniques have been pivotal since the early days of computing, with applications ranging from robotics to network routing. The Maze Solver Game leverages this rich heritage, adapting maze solving to demonstrate algorithmic efficiency and graph-based problem-solving in an accessible format.

2.2 Related Concepts

The project is grounded in several key concepts that are essential to understanding its design and functionality:

- **Grid-Based Maze**: A maze is represented as a 2D array, typically with a fixed size (e.g., 16x21 in this project), where each cell is assigned a value: walls (1), paths (0), or the goal (2). This representation simplifies the maze into a discrete structure, enabling algorithmic traversal and visualization. Such grids are standard in computational studies for their clarity and ease of manipulation.

- **BFS (Breadth-First Search)**: BFS is a graph traversal algorithm that methodically explores all neighbouring nodes at the current depth before advancing to deeper levels. This breadth-first approach ensures the shortest path in unweighted graphs, making it ideal for educational demonstrations of optimal path-finding. Its reliance on a queue data structure further reinforces foundational programming concepts.

- **DFS (Depth-First Search)**: In contrast, DFS explores as far as possible along each branch before backtracking, often implemented using a stack or recursion. While it may not guarantee the shortest path, its straightforward logic and exploratory nature make it a valuable teaching tool for understanding depth-based search strategies.

- **A* Algorithm**: Introduced by Hart et al. (1968), A* is a heuristic-based search algorithm that enhances efficiency by incorporating a cost function—typically combining path cost (g) and an estimated cost to the goal (h), such as Manhattan distance. This optimization makes A* a sophisticated choice for demonstrating advanced path-finding techniques. Together, these concepts provide the algorithmic backbone of the Maze Solver Game, showcasing a spectrum of search strategies.

2.3 Educational Context

The Maze Solver Game builds on these concepts to create an interactive tool that merges entertainment with education, drawing inspiration from both classic maze games and modern algorithmic visualizations. Traditional maze games, such as "Pac-Man" (1980), introduced players to navigation challenges, while educational software like "The Incredible Machine" (1993) popularized interactive problem-solving. This project adapts these influences for academic purposes, targeting students learning low-level programming and algorithm design. The use of C/C++—a language known for its control over system resources—and the Borland Graphics Interface (BGI) library enhances its suitability for teaching fundamental programming skills, as noted by Sedgewick (2011) in his emphasis on visual tools for algorithm education [2]. By providing a hands-on platform to explore BFS, DFS, and A*, the project aligns with pedagogical approaches that prioritize experiential learning, making abstract concepts tangible and engaging for BSSE students at the University of Dhaka.

3. Description of the Project

3.1 Overview

The Maze Solver Game is a desktop application designed to offer an interactive and educational experience through two distinct primary modes, each catering to different user interactions and learning objectives. Developed as a standalone program, it combines user engagement with algorithmic demonstration, running on a graphical interface that highlights both manual and automated problem-solving approaches.

1. **Game Mode:** In this mode, players actively navigate a randomly generated maze using intuitive controls—either arrow keys for directional movement or mouse clicks for point-and-click navigation. This mode emphasizes user participation, challenging players to find their way from the starting point to the goal through a dynamic, grid-based layout that varies with each session.

2. **Solver Mode:** Complementing the game mode, the solver mode employs automated algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), and A*—to solve the maze, providing a step-by-step visualization of the process. This mode is designed to illustrate the mechanics of each algorithm, offering users a clear, real-time view of how computational strategies unfold to reach the goal efficiently.
   Together, these modes create a versatile tool that not only entertains but also educates, leveraging a consistent maze structure to explore both human intuition and algorithmic precision**.**

3.2 Features

This table shows about the features and the description of all these features. The table is given below

Table 1: Features of Maze Solver Game

Feature	Description
Maze Generation	Randomly generates grid-based mazes
Manual Navigation	Supports keyboard and mouse input
Algorithm Visualization	Displays BFS, DFS, and A* solving processes
Save/Load Functionality	Allows saving and loading maze layouts
Hint System	Shows solution path on demand in game mode

3.3 Project Workflow

Purpose

The Project Workflow Diagram illustrates the high-level process flow of the Maze Solver Game, showing how the user interacts with the system and how the program processes inputs to generate outputs. It captures the sequence of operations from initialization to completion.

Components

1. **Start**: The entry point of the program, where execution begins. This initial step involves setting up the graphical environment, initializing variables, and preparing the system for user interaction, marking the launch of the Maze Solver Game application.

2. **Main Menu**: The initial user interface presented to the user, offering a selection of options to navigate the program's features. It displays choices such as playing the game, selecting a solver algorithm, or managing maze files, serving as the central hub for all subsequent actions.

3. **Game Mode**: The interactive gameplay branch where users take control. Players engage with a randomly generated maze, using keyboard or mouse inputs to explore and navigate from start to goal, testing their problem-solving skills in a dynamic, hands-on environment.

4. **Solver Mode**: The automated solving branch, featuring algorithm selection and execution. This mode allows users to choose from BFS, DFS, or A* algorithms, then observe as the selected method systematically solves the maze, highlighting computational approaches to path-finding.

5. **Maze Generation**: A common step for both modes, responsible for creating the maze structure. It uses a randomized algorithm to generate a grid-based layout with walls, paths, and a goal, ensuring each maze is unique and solvable for consistent gameplay and analysis.

6. **User Input**: Encompasses player navigation in game mode or algorithm selection in solver mode. In game mode, it processes keyboard or mouse commands to move the player; in solver mode, it captures the user's choice of algorithm to initiate the solving process.

7. **Path Finding**: The automated solving process exclusive to solver mode. This step executes the chosen algorithm (BFS, DFS, or A*), systematically exploring the maze to identify a path from start to goal, demonstrating the algorithm's logic and efficiency in action.

8. **Display Output**: The rendering of the maze and results for both modes. It visually presents the maze layout, player movements, or algorithm progress on-screen, using graphical elements like coloured cells and paths to provide clear, real-time feedback to the user.

9. **Save/Load**: File operations for maze persistence, enabling users to store and retrieve maze layouts. The save function writes the current maze to a file, while the load function reads a saved maze, offering flexibility to revisit or share specific configurations.

10. **End**: The program termination point, concluding the user's session. This step occurs after completing gameplay, observing a solution, or exiting via the menu, closing the graphical interface and releasing system resources to finalize the application's run

The project follows a modular design, as illustrated below:

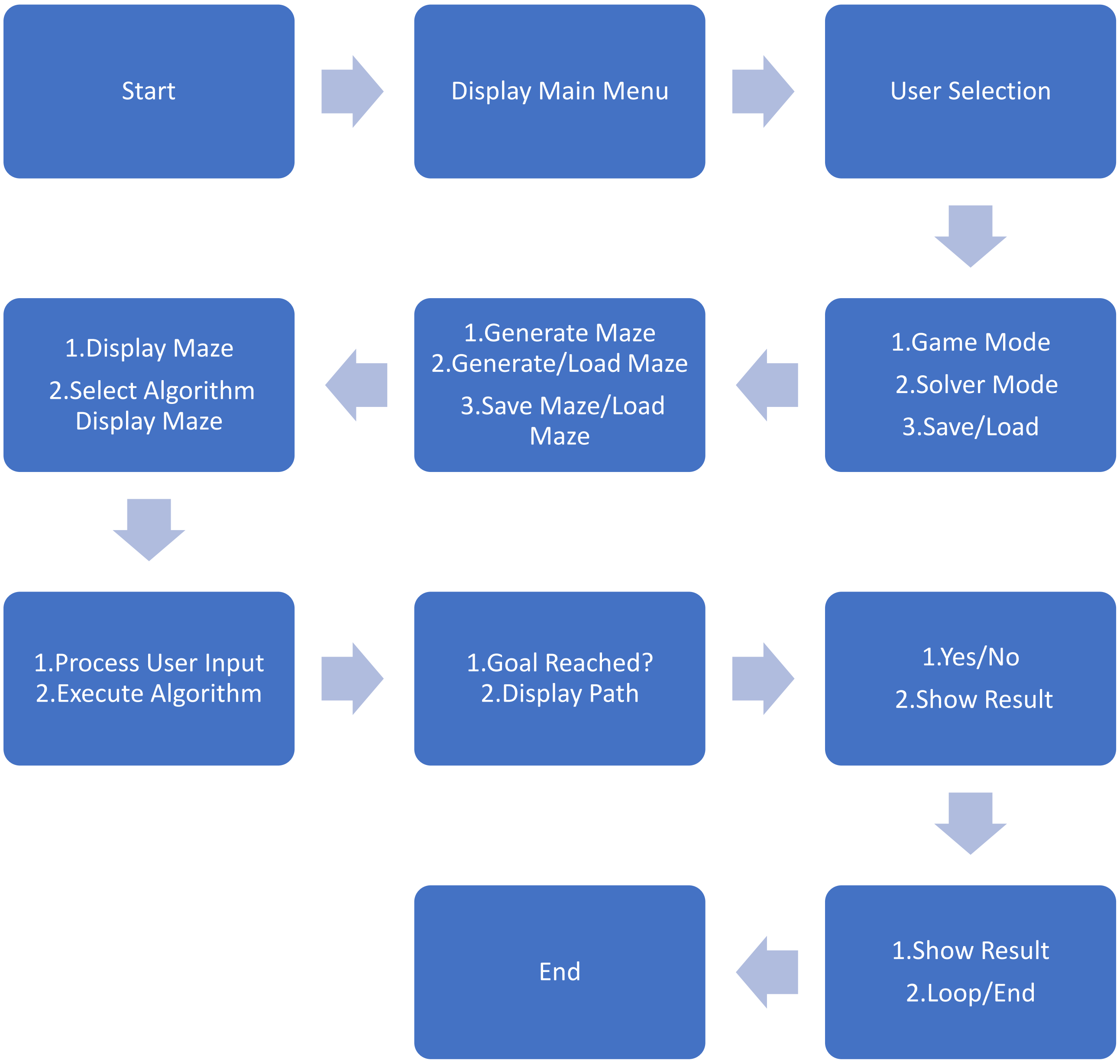


Figure 1: Project Workflow Diagram

4. Implementation and Testing

4.1 Implementation Details

4.1.1 Development Environment

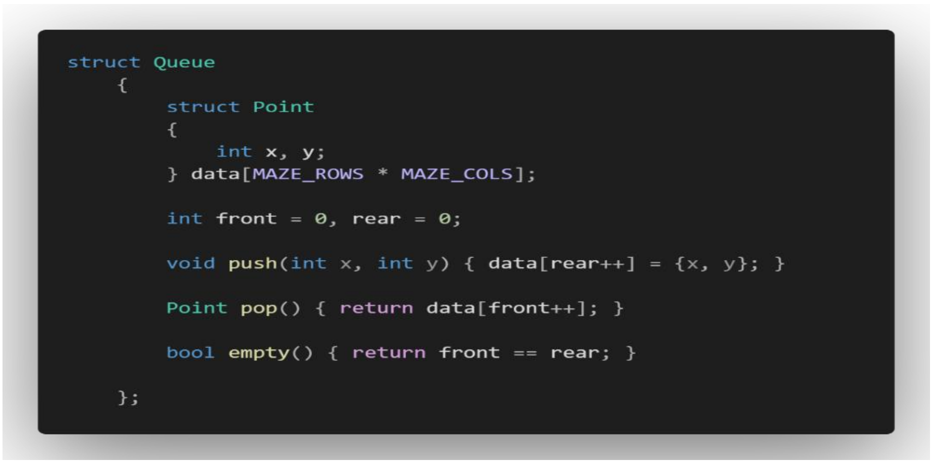
Table 2: Development Tools and Languages

Tool/Language	Purpose
C/C++	Core programming language
Visual Studio Code	IDE for coding
BGI Graphics	GUI rendering
GitHub	Version control

4.1.2 Key Components

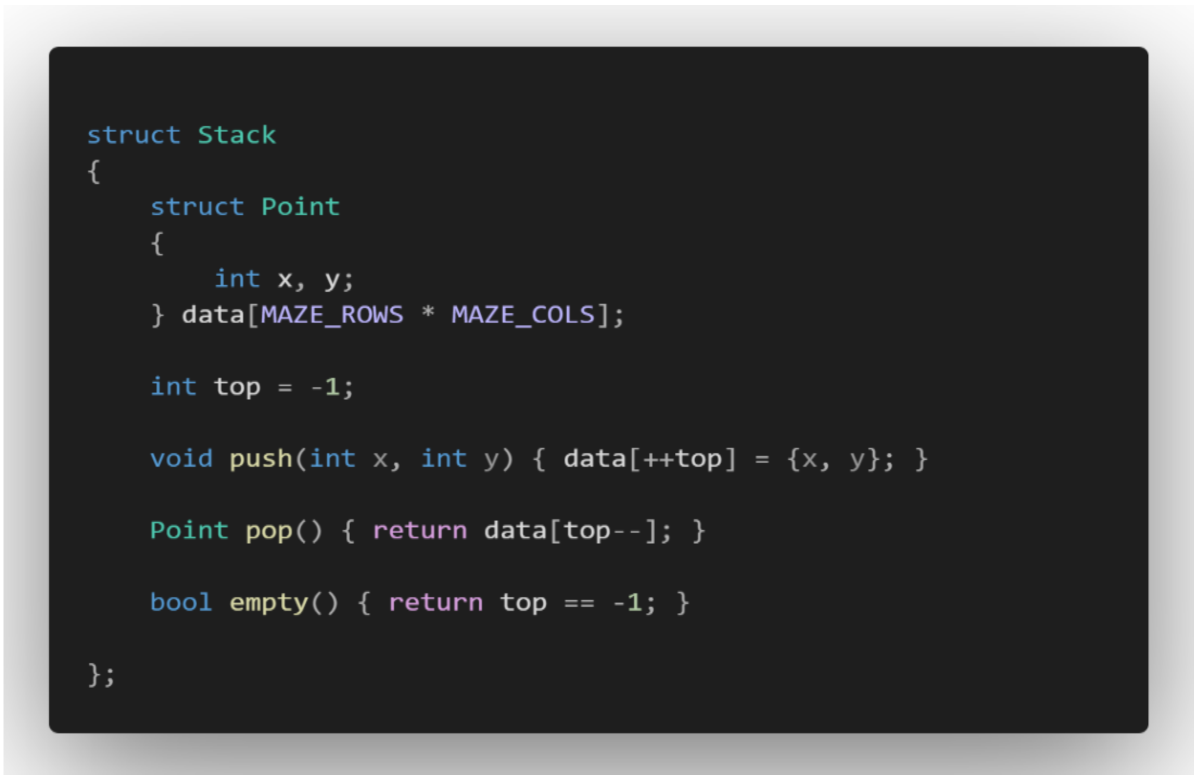
- **Maze Generation**: Uses rand() with a probability-based approach to create walls.
- **Path Finding**: Implements BFS using a queue, DFS using a stack, and A* with a priority-based node list.
- **GUI**: Utilizes BGI library functions like bar(), circle(), and outtextxy().

4.1.3 Code Snippet



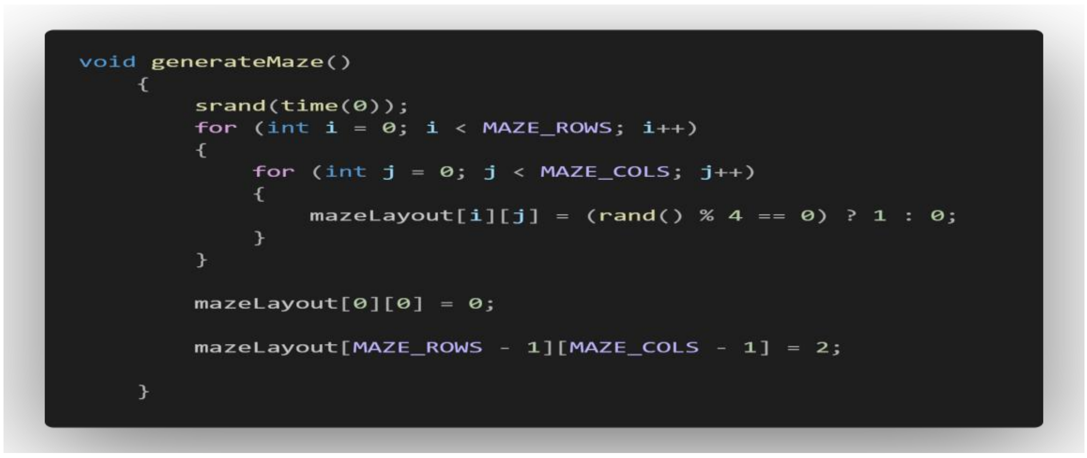
```
struct Queue
    {
        struct Point
        {
            int x, y;
        } data[MAZE_ROWS * MAZE_COLS];

        int front = 0, rear = 0;

        void push(int x, int y) { data[rear++] = {x, y}; }

        Point pop() { return data[front++]; }

        bool empty() { return front == rear; }

    };
```

Fig: Implementing Queue



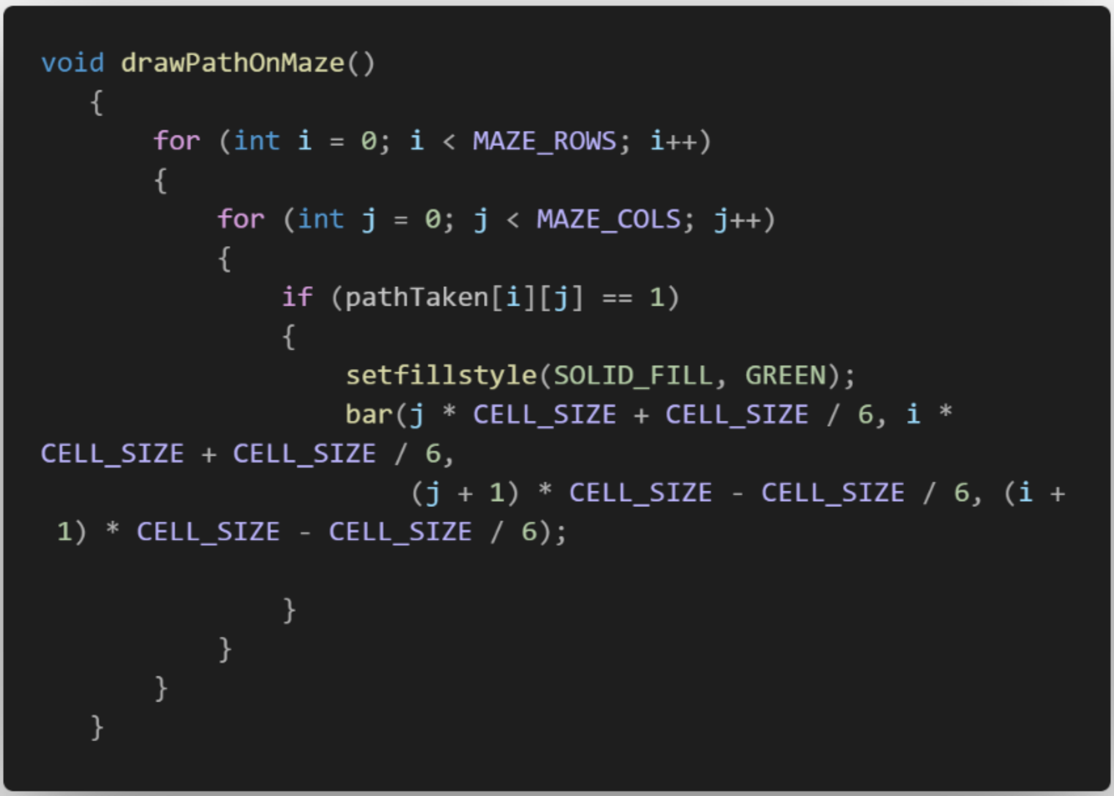
```
struct Stack
{
    struct Point
    {
        int x, y;
    } data[MAZE_ROWS * MAZE_COLS];

    int top = -1;

    void push(int x, int y) { data[++top] = {x, y}; }

    Point pop() { return data[top--]; }

    bool empty() { return top == -1; }

};
```

Fig: Implementing Stack



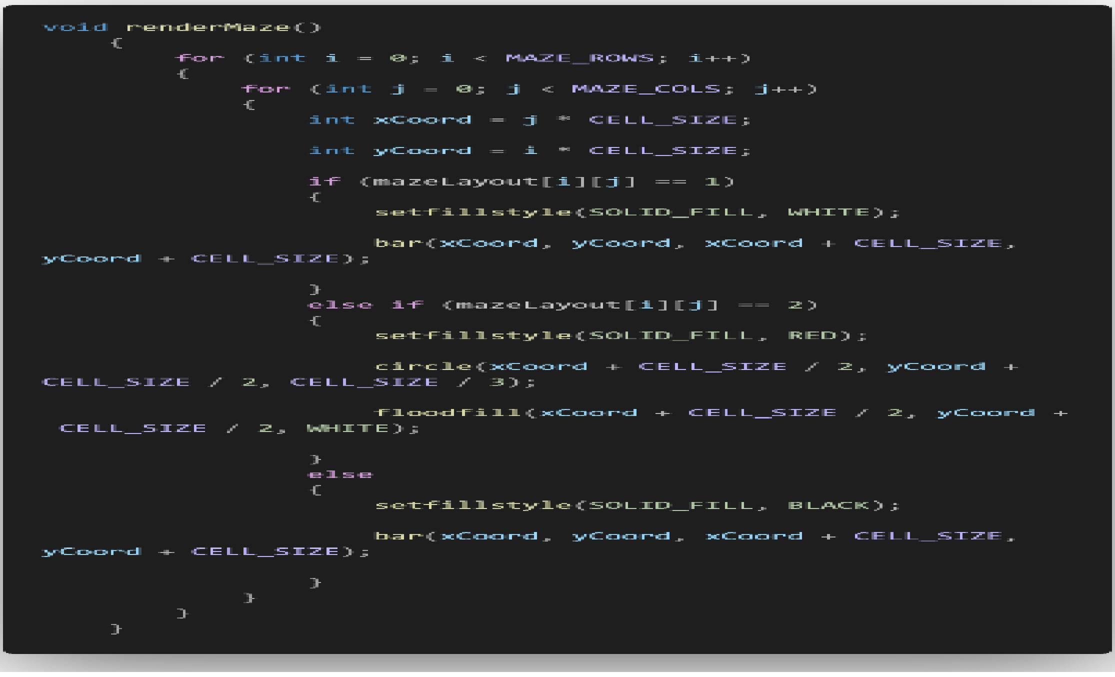
```
void generateMaze()
    {
        srand(time(0));
        for (int i = 0; i < MAZE_ROWS; i++)
        {
            for (int j = 0; j < MAZE_COLS; j++)
            {
                mazeLayout[i][j] = (rand() % 4 == 0) ? 1 : 0;
            }
        }

        mazeLayout[0][0] = 0;

        mazeLayout[MAZE_ROWS - 1][MAZE_COLS - 1] = 2;

    }
```

Fig: Generating Maze



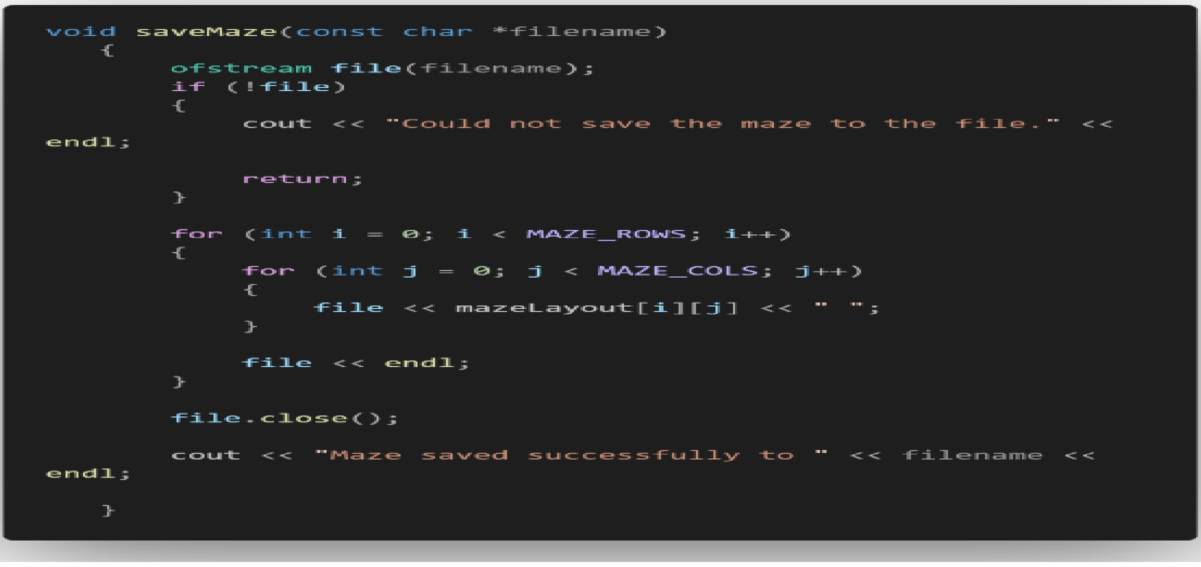
```
void drawPathOnMaze()
   {
       for (int i = 0; i < MAZE_ROWS; i++)
       {
           for (int j = 0; j < MAZE_COLS; j++)
           {
               if (pathTaken[i][j] == 1)
               {
                   setfillstyle(SOLID_FILL, GREEN);
                   bar(j * CELL_SIZE + CELL_SIZE / 6, i *
CELL_SIZE + CELL_SIZE / 6,
                       (j + 1) * CELL_SIZE - CELL_SIZE / 6, (i +
 1) * CELL_SIZE - CELL_SIZE / 6);

               }
           }
       }
   }
```

Fig: Drawing Path on Maze



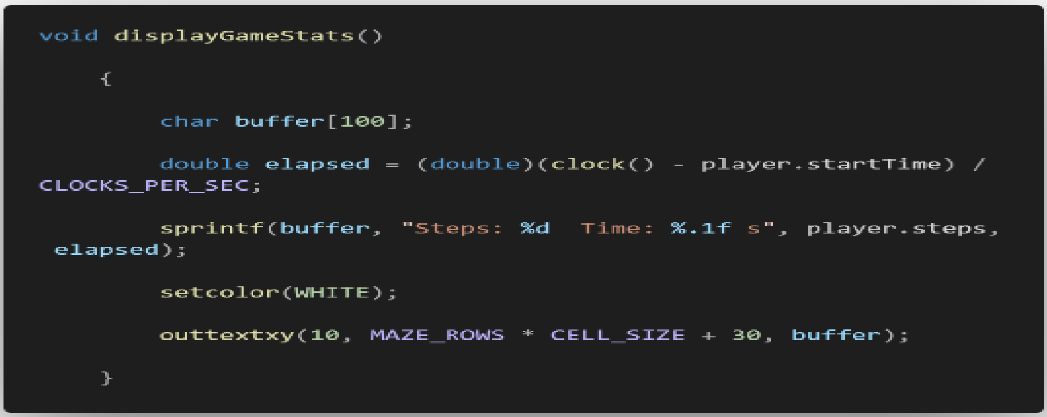
```
void renderMaze()
    {
        for (int i = 0; i < MAZE_ROWS; i++)
        {
            for (int j = 0; j < MAZE_COLS; j++)
            {
                int xCoord = j * CELL_SIZE;
                int yCoord = i * CELL_SIZE;
                if (mazeLayout[i][j] == 1)
                {
                    setfillstyle(SOLID_FILL, WHITE);
                    bar(xCoord, yCoord, xCoord + CELL_SIZE,
yCoord + CELL_SIZE);
                }
                else if (mazeLayout[i][j] == 2)
                {
                    setfillstyle(SOLID_FILL, RED);
                    circle(xCoord + CELL_SIZE / 2, yCoord +
CELL_SIZE / 2, CELL_SIZE / 3);
                    floodfill(xCoord + CELL_SIZE / 2, yCoord +
 CELL_SIZE / 2, WHITE);
                }
                else
                {
                    setfillstyle(SOLID_FILL, BLACK);
                    bar(xCoord, yCoord, xCoord + CELL_SIZE,
yCoord + CELL_SIZE);
                }
            }
        }
    }
```

Fig: Render Maze



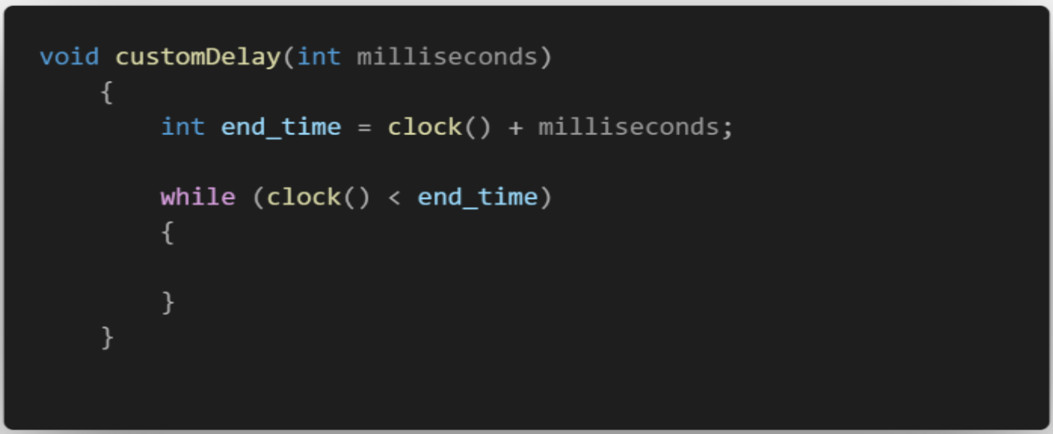
```
void saveMaze(const char *filename)
    {
        ofstream file(filename);
        if (!file)
        {
            cout << "Could not save the maze to the file." <<
endl;
            return;
        }
        for (int i = 0; i < MAZE_ROWS; i++)
        {
            for (int j = 0; j < MAZE_COLS; j++)
            {
                file << mazeLayout[i][j] << " ";
            }
            file << endl;
        }
        file.close();
        cout << "Maze saved successfully to " << filename <<
endl;
    }
```

Fig: Saving the maze



```
void displayGameStats()

    {

        char buffer[100];

        double elapsed = (double)(clock() - player.startTime) /
CLOCKS_PER_SEC;

        sprintf(buffer, "Steps: %d  Time: %.1f s", player.steps,
 elapsed);

        setcolor(WHITE);

        outtextxy(10, MAZE_ROWS * CELL_SIZE + 30, buffer);

    }
```

Fig: Displaying Game Stat



```
void customDelay(int milliseconds)
    {
        int end_time = clock() + milliseconds;

        while (clock() < end_time)
        {

        }
    }
```

Fig: Showing the custom Delay

4.2 Testing

4.2.1 Test Cases

This table show us The test case for Maze Generation and Path Finding.

Table 3: Test Cases for Maze Generation and Path Finding

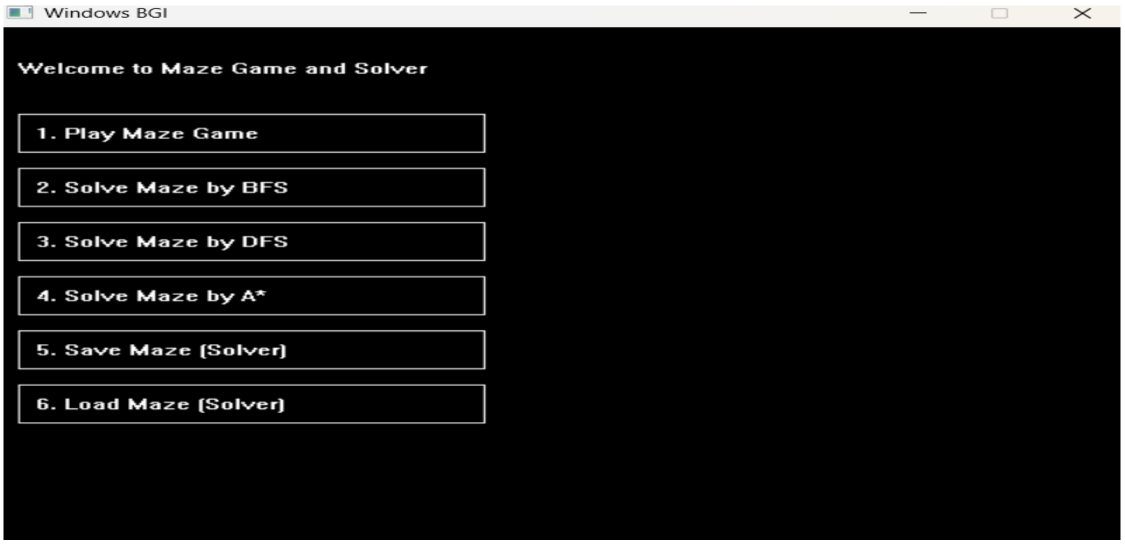
Test Case	Input	Expected Output	Result
Maze Generation	Random seed	Valid maze with start/goal	Pass
BFS Path Finding	Generated maze	Shortest path to goal	Pass
DFS Path Finding	Generated maze	Valid path to goal	Pass
A* Path Finding	Generated maze	Optimal path to goal	Pass

4.2.2 Methodology

The testing methodology for the Maze Solver Game encompassed a comprehensive approach to ensure reliability and functionality. Unit tests were conducted on individual components, such as maze generation, path-finding algorithms (BFS, DFS, A*), and GUI rendering, to verify their correctness in isolation. Integration tests followed, evaluating the seamless interaction of these components within the full system, including transitions between game and solver modes. Additionally, edge cases were rigorously tested, such as unsolvable mazes, to assess the program's robustness and error handling. This multi-tiered strategy confirmed the software's stability, accuracy, and ability to handle diverse scenarios effectively.

5. User Interface

5.1 Main Menu



Welcome to Maze Game and Solver

Figure 3: Main Menu Screenshot

5.2 Gameplay

Players use arrow keys to move the yellow square from the top-left corner to the goal.

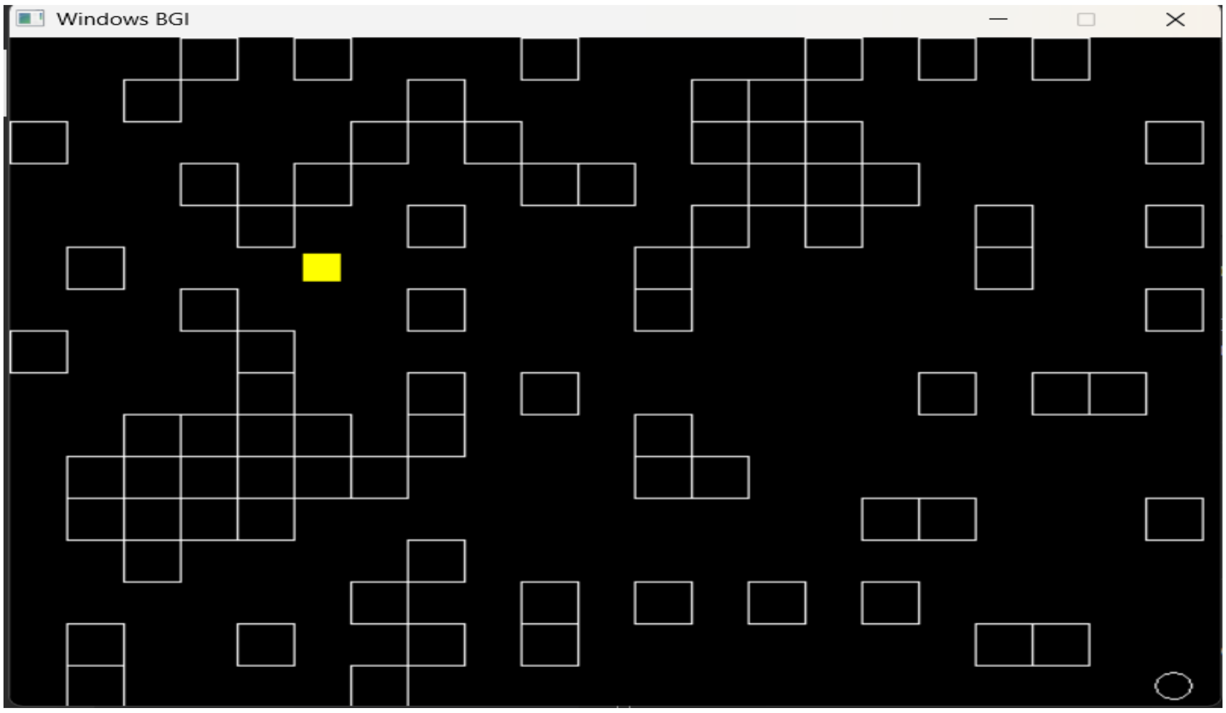


Figure 4: Gameplay Screenshot

5.3 Solver Visualization

The green path shows the algorithm's progress.

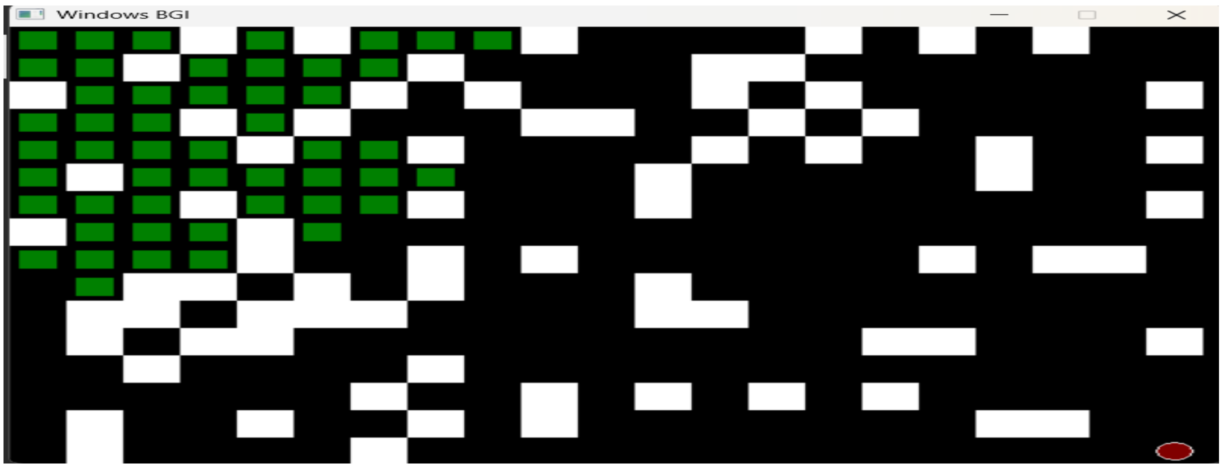


Fig: BFS Traversal

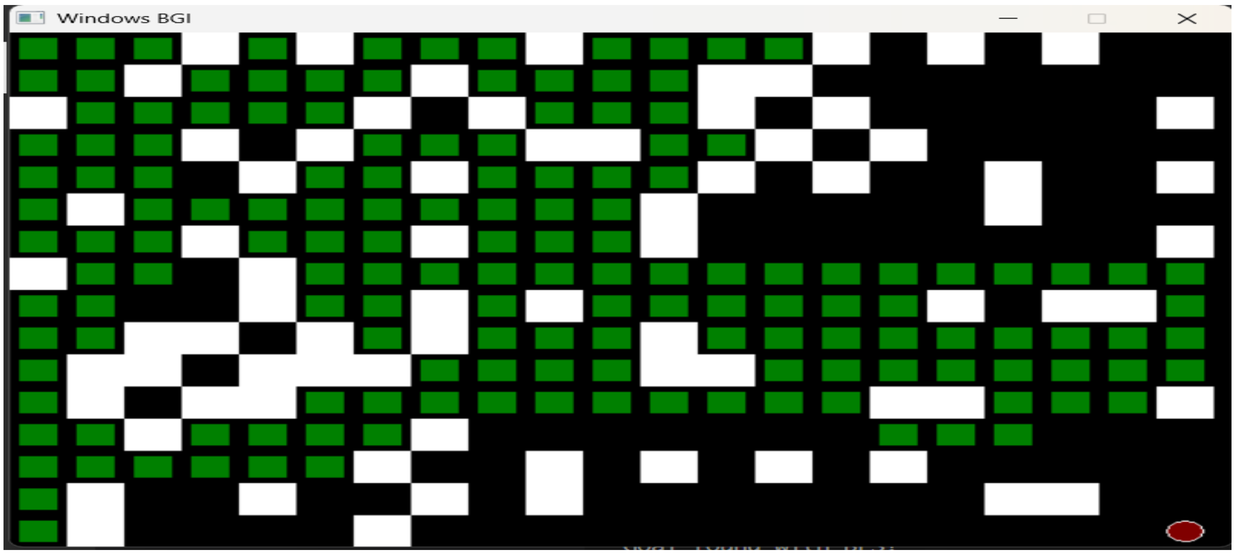


Fig: DFA Traversal

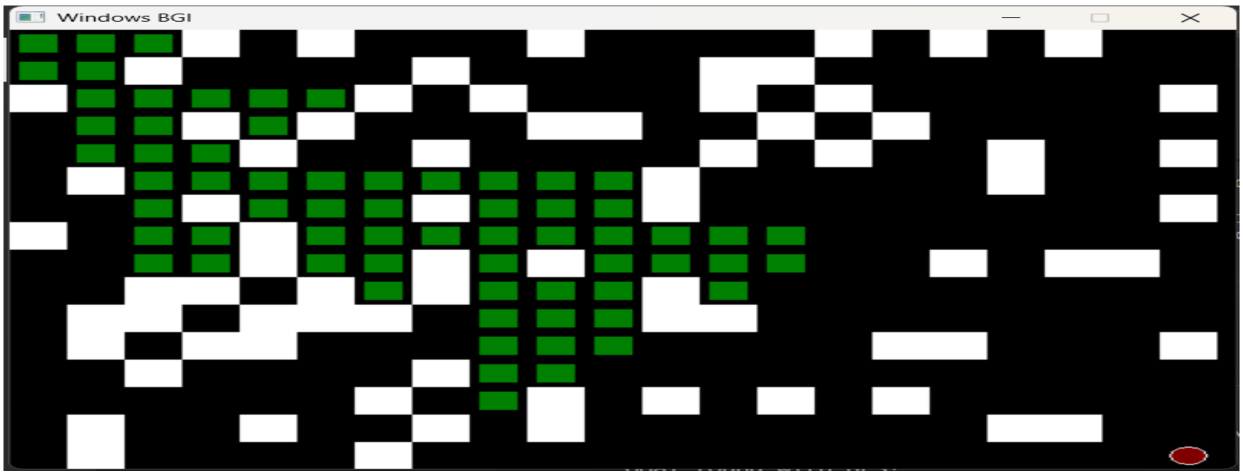


Fig: A* Traversal

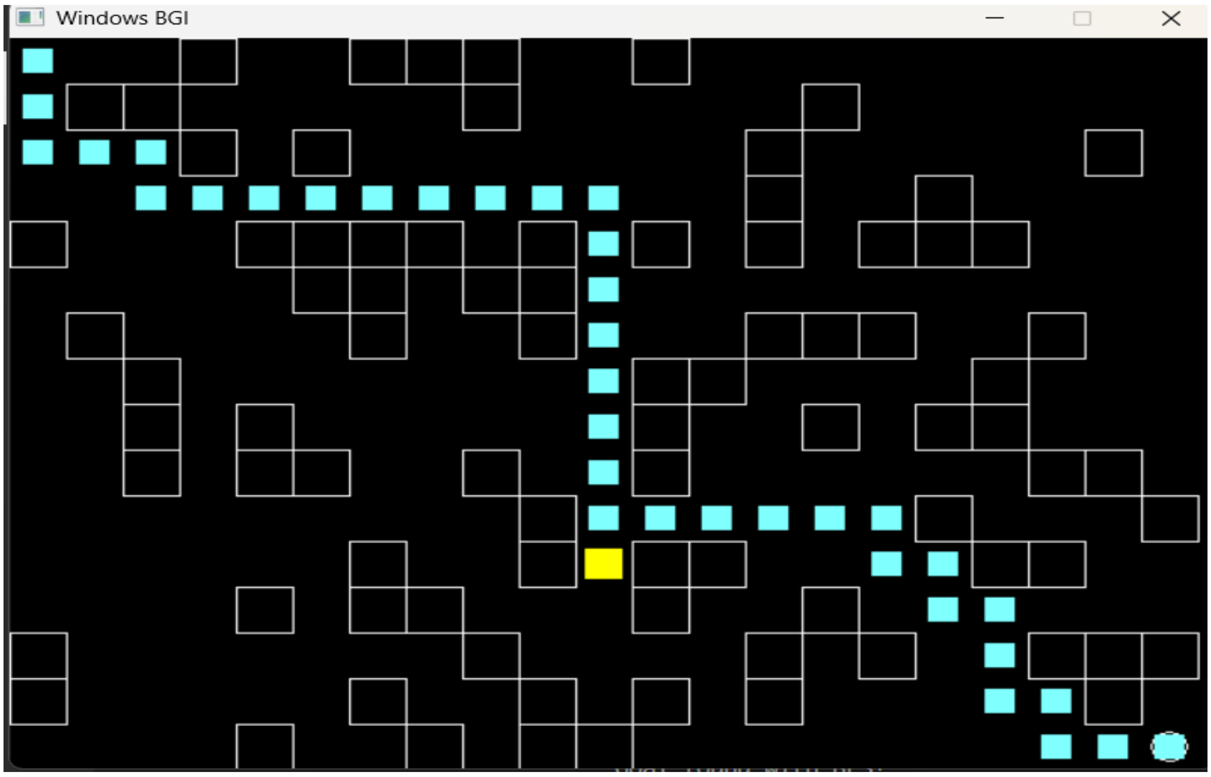


Figure 5: Solution Path Screenshot

6. Challenges Faced

6.1 Challenge 1: Random Maze Solvability

- **Issue**: During development, a significant hurdle emerged with randomly generated mazes, as the initial algorithm occasionally produced layouts that were unsolvable—lacking a viable path from the starting point to the goal. This issue undermined the project's core functionality, as both game and solver modes required a solvable maze to ensure a meaningful user experience.
- **Solution**: To address this, a path-finding check was integrated into the maze generation process. Specifically, after creating a random layout, a preliminary BFS algorithm runs to verify the existence of a path between the start (top-left corner) and goal (bottom-right corner). If no path is found, the maze is regenerated until solvability is confirmed, guaranteeing a consistent and playable environment for all users.

6.2 Challenge 2: Graphics Library Compatibility

- **Issue**: The use of the Borland Graphics Interface (BGI) library introduced compatibility challenges, as its setup varied across different systems and development environments. Inconsistent paths to the library file (e.g., libbgi.a) caused compilation errors on some machines, complicating deployment and testing, especially in a lab setting with diverse configurations.
- **Solution**: To resolve this, the project standardized the BGI library path to a specific location (e.g., "C:\MinGW\lib\libbgi.a") compatible with the MinGW compiler used in development. Additionally, detailed setup instructions were documented and included in the appendix, providing step-by-step guidance for installing the compiler and configuring the library, ensuring portability and ease of use across systems.

6.3 Challenge 3: Real-Time Visualization

- **Issue**: Real-time visualization of the solver algorithms (BFS, DFS, A*) posed a performance challenge, as the initial rendering process was excessively slow. This lag disrupted the step-by-step display of path-finding progress, making it difficult for users to follow the algorithms' operations, particularly on less powerful hardware, thus diminishing the educational impact.
- **Solution**: To mitigate this, custom delay functions (e.g., customDelay()) were introduced to regulate the pace of rendering, allowing users to observe each step clearly without overwhelming the system. Furthermore, drawing routines were optimized by minimizing redundant graphical updates—such as reusing existing screen elements instead of redrawing the entire maze—enhancing performance and ensuring a smooth, effective visualization experience.

7. Conclusion

7.1 Learning Outcomes

This project enhanced my understanding of algorithms, GUI programming, and software testing. I gained practical experience in C/C++ and graphics programming.

7.2 Future Extensions

- Add difficulty levels
- Implement sound effects
- Support larger maze sizes

8. References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C., *Introduction to Algorithms*.
2. Mazes for Programmers Code Your Own Twisty Little Passages (Jamis Buck)
3. BGI Graphics Library Tutorial from youtube.

9. Appendix

9.1 Installation Instructions

1. Install MinGW compiler.
2. Set BGI library path to "C:\MinGW\lib\libbgi.a".
3. Compile using g++ maze_solver.cpp -lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32.

9.2 Full Source Code

Github Link: https://github.com/tjadnan1520