
**Tribhuvan University**

**Faculty of Humanities and Social Sciences**

**A PROJECT REPORT ON**

# Maze Solver: Maze Game using Dijkstra & A* Algorithm

**Submitted to:**

**Department of Computer Application**

**Prime College**

*In partial fulfillment of the requirements for the Bachelors in Computer Application (BCA) 6th semester*

**Submitted by**

**Sagan Krishna Tamrakar (6-2-410-136-2021)**

**July 2025 A.D**

**Under the supervision of**

**Mr. Niraj Khadka**


**Tribhuvan University**

**Faculty of Humanities and Social Sciences**

**A PROJECT REPORT ON**

# Maze Solver: Maze Game using Dijkstra & A* Algorithm

**Submitted to:**

**Department of Computer Application**

**Prime College**

*In partial fulfillment of the requirements for the Bachelors in Computer Application (BCA) 6th semester*

**Submitted by**

**Sagan Krishna Tamrakar (6-2-410-136-2021)**

**July 2025 A.D**

**Under the supervision of**

**Mr. Niraj Khadka**


**Tribhuvan University**

**Faculty of Humanities and Social Science**

**Prime College**

## SUPERVISOR'S RECOMMENDATION

I hereby recommend that this project prepared under my supervision by Sagan Krishna Tamrakar entitled "**Maze Solver: Maze Game using Dijkstra & A* Algorithm**" in partial fulfillment of the requirements for the degree of Bachelor of Computer Application is recommended for the final evaluation.

……

**Mr. Niraj Khadka**

**SUPERVISOR**

**Department of Computer Science & IT**

**Prime College**


**Tribhuvan University**

**Faculty of Humanities and Social Science**

**Prime College**

## LETTER OF APPROVAL

This is to certify that this project report prepared by Bishan Poudel on "Diabetes Prediction System with Random Forest, SVM and Gradient Boosting Algorithm" in partial fulfillment of the requirements for the degree of Bachelor in Computer Application has been evaluated. In our opinion it is satisfactory in the scope and quality as a project for the required degree.

| | |
|---|---|
| Mr. Niraj Khadka<br>Supervisor<br>Prime College<br>Khusibu, Nayabajar, Kathmandu | Mrs. Rolisha Sthapit<br>Program Co-Ordinator<br>BCA Department<br>Prime College<br>Khusibun, Nayabazar, Kathmandu |
| | Mr. Kumar Prasun<br>External Examiner |

# ABSTRACT

The "Maze Solver: Maze Game using Dijkstra & A* Algorithm", is an Android Application aimed at visualizing and comparing classic pathfinding algorithms. The application allows users to generate mazes using recursive backtracking, manually solve them, or utilize automated solving through Dijkstra's Algorithm and A* (A-Star) algorithm. The application offers dual-mode experience, users can either play the maze game by navigating through swipe gestures or observe how the algorithms traverse the maze and determine the shortest path. The app also supports features like animated algorithm visualization, multiple path solutions, maze saving/loading functionality, and scoring based on the similarity between the player's path and the optimal path derived by the algorithm. By integrating gameplay with educational algorithm visualization, this project serves as both a learning tool and an engaging puzzle game. It demonstrates the practical applications of pathfinding algorithms in a visual and user-friendly manner, making it suitable for students, educators, and developers interested in game development.

*Keywords: Dijkstra's Algorithm, A* Algorithm, Algorithm Visualization, Game Development, Recursive Backtracking*

# ACKNOWLEDGEMENT

In the fulfillment of the project for this semester, I am deeply grateful to all those who have directly and indirectly contributed to the development of this "Maze Solver: Maze Game using Dijkstra & A* Algorithm" Project.

Firstly, I would like to thank my project supervisor, Niraj Khadka, for his continuous support, encouragement, and valuable feedback throughout the development process. His guidance helped shape the project into a meaningful and practical application.

I would also like to acknowledge our institution, Prime College, for providing the necessary resources and environment conducive to learning and innovation.

Furthermore, I wish to express my gratitude to the seniors of Prime College for their guidance, which proved instrumental in shaping the direction of this project. Their willingness to share their experiences and knowledge has been incredibly valuable.

Finally, I extend my sincere thanks to Tribhuvan University for providing the platform to explore and understand the development of the project through the course of Computer Application. This opportunity has been important in helping me improve skills and learn more.

-Sagan Krishna Tamrakar

# TABLE OF CONTENTS

# LIST OF ABBREVATIONS

| | |
|---|---|
| **UI** | User Interface |
| **UX** | User Experience |
| **API** | Application Programming Interface |
| **IDE** | Integrated Development Environment |
| **XML** | eXtensible Markup Language |
| **JVM** | Java Virtual Machine |
| **SDK** | Software Development Kit |
| **OOP** | Object-Oriented Programming |
| **A*** | A-Star Algorithm |
| **BFS** | Breadth-First Search |
| **DFS** | Depth-First Search |
| **UX** | User Experience |
| **OS** | Operating System |

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1:
# INTRODUCTION

## 1.1 Introduction

Maze-solving is a classical problem in computer science and mathematics, often used to demonstrate and test pathfinding algorithms and problem-solving logic. With the development of mobile computing and interactive application development, maze-solving games have become an excellent platform for integrating algorithmic knowledge with user experience and interactivity.

This project, “Maze Solver: Maze Game using Dijkstra & A Algorithm,” is an interactive Android application that allows users to explore and solve mazes manually or watch them be solved automatically using two algorithms: Dijkstra’s Algorithm and A* (A-Star) Algorithm. The application not only serves as a game but also as a visualization tool for understanding how these algorithms go across a grid and determine optimal paths.

The maze is generated dynamically using the recursive backtracking algorithm, ensuring a unique and challenging layout for every session. The application further supports animations for algorithm steps, score comparison between user paths and optimal paths, and saving/loading mazes for replay or testing.

This project aims to bridge the gap between theory and practice by turning algorithmic learning into an engaging and hands-on experience.

## 1.2 Problem Statement

While pathfinding algorithms like Dijkstra's and A* are fundamental topics in computer science, they are often taught in abstract forms with limited opportunities for practical and interactive exploration. Traditional methods may not provide an intuitive understanding of how these algorithms work in real-time or how they differ in behavior and performance. Additionally, most maze-solving applications either focus solely on gameplay without educational insight or lack features that allow users to interactively compare algorithmic efficiency and pathfinding logic.

With the need for an interactive, educational, and visually rich platform that enables Real-time visualization of Dijkstra's and A* algorithms. Along with User Interaction to manually solve mazes and compare results and the ability to generate, save, and load unique mazes for testing and learning. This project addresses these issues by developing a mobile-based maze-solving game that combines gameplay with educational utility.

## 1.3 Objectives

- To implement Dijkstra's and A* algorithms for automated maze solving and recursive backtracking algorithm for generating mazes.
- To allow users to manually solve the maze and receive a score based on optimal path comparison.
- To provide functionality to save, load, and delete multiple mazes.

## 1.4 Scope and Limitation

**Scope:**

- It enables users to interactively solve or visualize solutions to randomly generated mazes.
- Both Dijkstra's and A* algorithms are used for demonstrating and comparing pathfinding efficiency.
- Users can save multiple mazes, load them for later use.

**Limitations:**

- Maze grid size is fixed and not dynamically resizable at runtime.
- The application currently supports square mazes only.
- Scoring depends on strict sequential comparison with the optimal path and may not reflect alternate efficient paths.

## 1.5 Development Methodology

A software development methodology is a structured approach or set of practices used to plan, design, develop, test and implement.

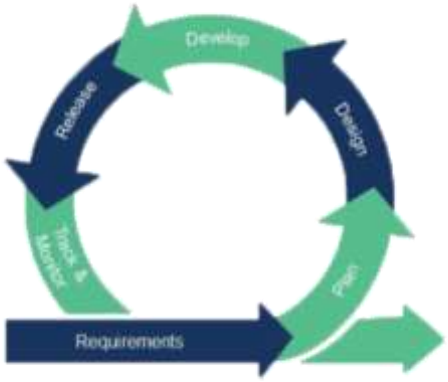



**Figure 1.1: Agile Methodology**

In this project the development methodology used was Agile Methodology, the development was divided into 3 iterative cycle or sprints, each focusing on delivering a functional part of the system.

**1. Planning and Algorithm Selection**

- Defined the core objective and identified essential features
- Selected Recursive backtracking algorithm for maze generation and Dijkstra and A* algorithm for maze solving
- Designed the conceptual diagram for the project

**2. Development and Algorithm Integration**

- Developed the Frontend of the project using Android Studio, XML.
- Implemented Algorithm for maze generation and solving.
- Implemented user solving and modified recursive backtracking algorithm for multiple path maze generation.
- Refined solved algorithm process with visualization of the algorithm paths.

**3. Implementation of Other features**

- Implemented the Ability to save and load mazes.
- Improved the visualization of the solved solve by highlighting the shortest path.
- Implemented Deletion of Saved maze.
- Conducted minor improvements for better user experience.

## 1.6 Report Organization

This project report is organized into five main chapters, each detailing specific aspects of the system development process. The structure of the report is as follows:

Chapter 1: This chapter provides an overview of the project, including the background, problem statement, objectives, scope and limitations, and the development methodology adopted.

Chapter 2: This chapter covers the theoretical background related to maze-solving algorithms and mobile game development. It also includes a review of related works and technologies used.

Chapter 3: This chapter describes the system requirements, feasibility study, and detailed system design using objects, dynamic, and process modeling. It also explains the architecture and algorithms used (Dijkstra & A*).

Chapter 4: This chapter focuses on the implementation environment, tools used, code structure, and detailed testing procedures. It also presents the result analysis of the implemented features.

Chapter 5: This chapter summarizes the outcomes of the project, lessons learned and provides suggestions for further enhancement of the application.

# CHAPTER 2:
# BACKGROUND STUDY AND LITERATURE REVIEW

## 2.1 Background Study

Maze solving is a classical problem in computer science, often used to demonstrate pathfinding algorithms, recursion, graph traversal, and algorithm efficiency. The foundation of this project is based on transforming theoretical maze-solving concepts into a visual and interactive Android application that allows both automatic and manual solving of mazes.

Among the many algorithms used in pathfinding, Dijkstra's Algorithm and A (A-Star) Algorithm* are widely recognized for their accuracy and performance. Dijkstra's algorithm finds the shortest path in a graph without considering heuristics, making it suitable for uniform-cost paths. A*, on the other hand, uses heuristics (such as Manhattan distance) to guide its search, offering a more efficient solution in many scenarios.

The implementation of this maze game required not just understanding of the pathfinding techniques, but also the development of robust UI, gesture controls, save/load mechanisms, score evaluation, and visual feedback (like animations and path highlighting) for enhanced user interaction.

## 2.2 Literature Review

"Pathfinding Algorithms for Maze Solving: A Comparative Study" [1] examines various pathfinding algorithms, comparing their efficiency in solving mazes. The research highlights the advantages of A* and Dijkstra's algorithms, emphasizing their accuracy, computational efficiency, and applicability in real-time problem-solving.

"Mobile Applications for Interactive Learning: Enhancing User Engagement through Gamification" [2] explore the role of interactive learning through mobile applications. The study emphasizes how gamification and user-friendly interfaces contribute to better user engagement and knowledge retention, supporting the application's goal of making maze-solving an enjoyable and educational experience.

"Customizable Digital Mazes: Enhancing User Creativity in Algorithmic Problem Solving" [3] analyzes the impact of customizable maze applications on user creativity

and engagement. The findings suggest that allowing users to create and modify mazes enhances their problem-solving skills and overall learning experience.

"Artificial Intelligence in Puzzle Games: The Role of Algorithmic Complexity" [4] investigates how artificial intelligence, and advanced algorithms contribute to the effectiveness of puzzle-solving applications. The research supports the implementation of A* and Dijkstra's algorithms, as they provide optimal pathfinding solutions while maintaining computational efficiency. "Algorithmic Approaches to Maze Solving" (Smith et al., 2020): Discusses various pathfinding algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and A*.

Comparison of Pathfinding Algorithms for Maze Solving: Dijkstra vs. A* [5] states that while Dijkstra guarantees the shortest path by exploring all possible routes, it can be slow in larger mazes. A*, introduced by Hart et al. (1968), improves efficiency by using a heuristic (like Manhattan Distance) to prioritize promising paths, offering faster performance with similar accuracy. Research by Khan et al. (2021) in mobile gaming also supports A* as more practical for real-time applications, making it ideal for maze-solving games like this project.

# CHAPTER 3:
# SYSTEM ANALYSIS AND DESIGN

## 3.1 System Analysis

### 3.1.1 Requirement Analysis

**Functional Requirements:**

The Maze Solver game application is designed for Android devices which allow users to generate random mazes. The application also allows the users to solve the mazes manually or solve them automatically through the use of Dijkstra algorithm and A* Algorithm. The system also allows the user to save, load and delete multiple maze files.

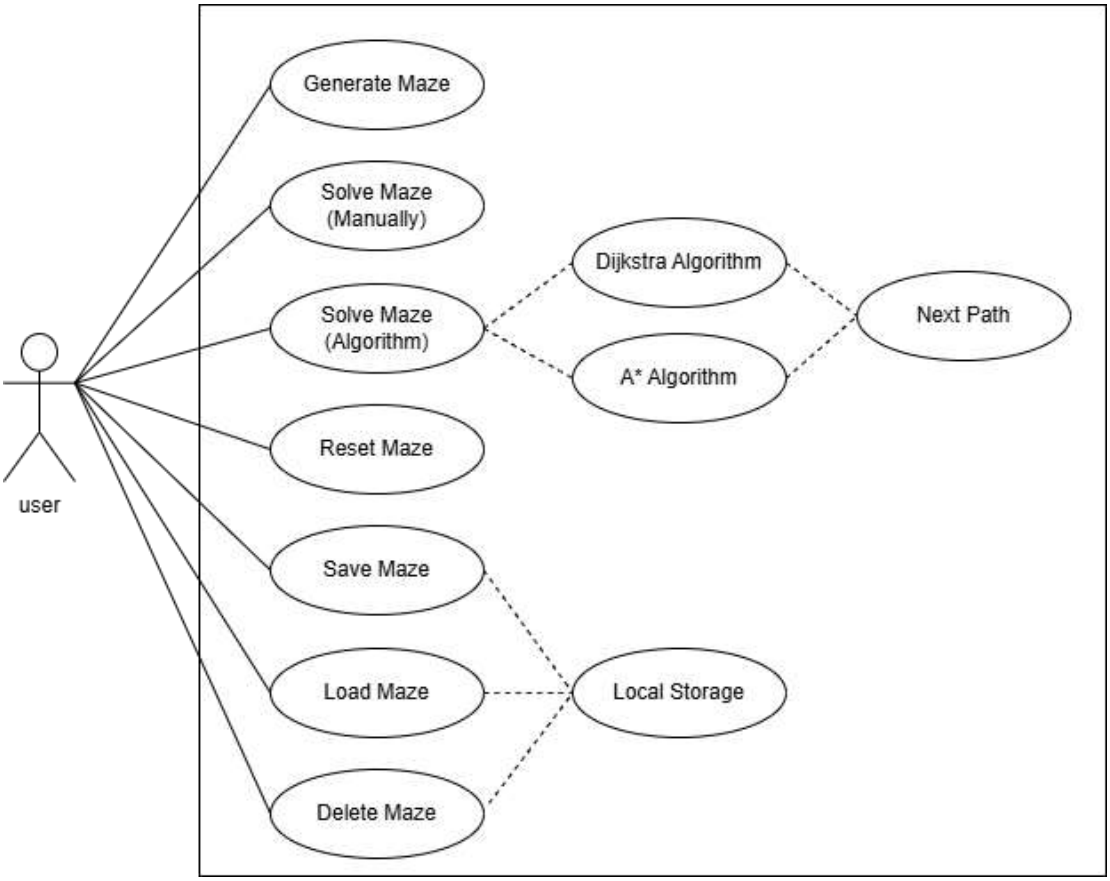



**Figure 3.1: Use Case Diagram**

**Non-Functional Requirements:**

1. **User Friendly:** The system should be user friendly and should have a simple interface for easy usability by the user.
2. **Device Compatibility:** The application should run smoothly on mid-range Android devices.
3. **System Feedback:** The system should provide feedback via animation and scoring.
4. **Responsive Interaction:** User interaction (gesture control) should be responsive and intuitive.
5. **Maintainability:** The codebase should be modular and easy to update, allowing future improvements or bug fixes with minimal effort.

### 3.1.2 Feasibility Analysis

**i. Technical Feasibility:**

- **Algorithm Integration:** This project uses 2 pathfinding algorithms- Dijkstra and A* algorithms to solve the maze and uses Recursive Backtracking algorithm to generate the maze.
- **Computational Resources:** The computational requirements for running these algorithms are manageable with standard hardware configurations, such as Android 6.0 or more compatible devices.
- **App Development:** The application is developed using Java in Android Studio, which is suitable for mobile development.

**ii. Economic Feasibility:**

- **Cost of Development:** No hardware or paid libraries were used to develop this project, all the tools and resources used were open-sourced.
- **Resource Allocation:** The project's computational and infrastructure requirements are within the available resources, making the project economically viable.

### iii. Operational Feasibility:

- **User Experience:** The system is designed with a focus on ease of use for everyone. The application is simple and requires no learning curves.
- **Scalability:** The system architecture supports future scalability, allowing for the addition of more features like premade levels and scalable grid-size of the maze.

### iv. Schedule Feasibility:

To ensure the timely completion of the project, the time has been divided into different schedules

**Table 3.1: Schedule Feasibility**

| Sprint & Phase | Start Date | End Date | Duration (Days) |
|---|---|---|---|
| **Sprint 1: Planning & Algorithm Selection** | | | |
| Planning | 15-Mar | 18-Mar | 3 days |
| Designing | 19-Mar | 24-Mar | 6 days |
| Development | 25-Mar | 1-Apr | 8 days |
| Testing | 2-Apr | 4-Apr | 3 days |
| Review | 5-Apr | 8-Apr | 4 days |
| **Sprint 1 Total Duration** | 15-Mar | 8-Apr | 24 days |
| | | | |
| **Sprint 2: Development & Algorithm Integration** | | | |
| Planning | 9-Apr | 11-Apr | 3 days |
| Designing | 12-Apr | 14-Apr | 3 days |
| Development | 15-Apr | 28-Apr | 14 days |
| Testing | 29-Apr | 2-May | 4 days |
| Review | 3-May | 5-May | 3 days |
| **Sprint 2 Total Duration** | 9-Apr | 5-May | 27 days |
| | | | |
| **Sprint 3: Implementation of Other Features** | | | |
| Planning | 6-May | 8-May | 3 days |
| Designing | 9-May | 12-May | 4 days |
| Development | 13-May | 26-May | 14 days |
| Testing | 27-May | 30-May | 4 days |
| Review | 31-May | 2-Jun | 3 days |
| **Sprint 3 Total Duration** | 6-May | 2-Jun | 28 days |


**Figure3.2: Gantt Chart**

### 3.1.3 Object Modeling: Object and Class Diagram

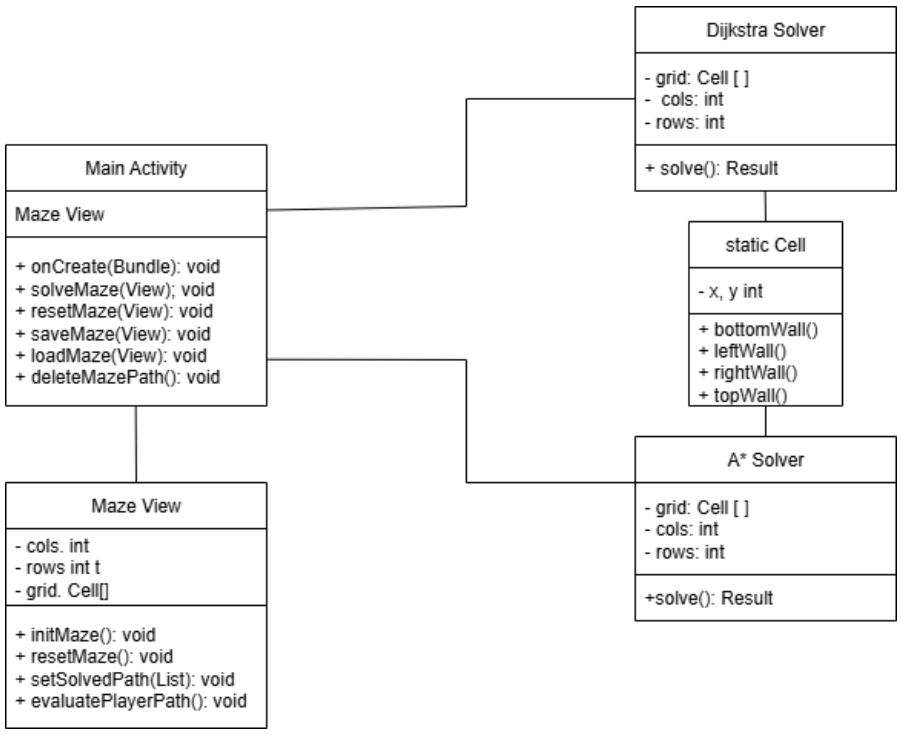



**Figure 3.3: Class Diagram**

In the above figure there are multiple classes each with their own functions. The separation of classes helps maintain the balance of the project and ensures the project's functionality. The maze view class is responsible for the generation of mazes. Each algorithm class is responsible for executing its specific logic, while the main activity coordinated these processes to deliver the final completed solution.

### 3.1.4 Dynamic Modeling: State & Sequence Modeling

The state diagram shows how the maze and player interact across different states in the game. The system has the following major states:

- **Maze Initialization** – The maze is created using recursive backtracking with optional extra paths.
- **Idle State** – In this state the application is waiting for the human inputs to continue.
- **Solving State** – In this state the user can either choose to solve the maze using the two algorithms (A* or Dijkstra's algorithm) or solve the maze manually.
- **Solved State** – During this state the program is completed, and the final path is revealed, and user reaches the goal.

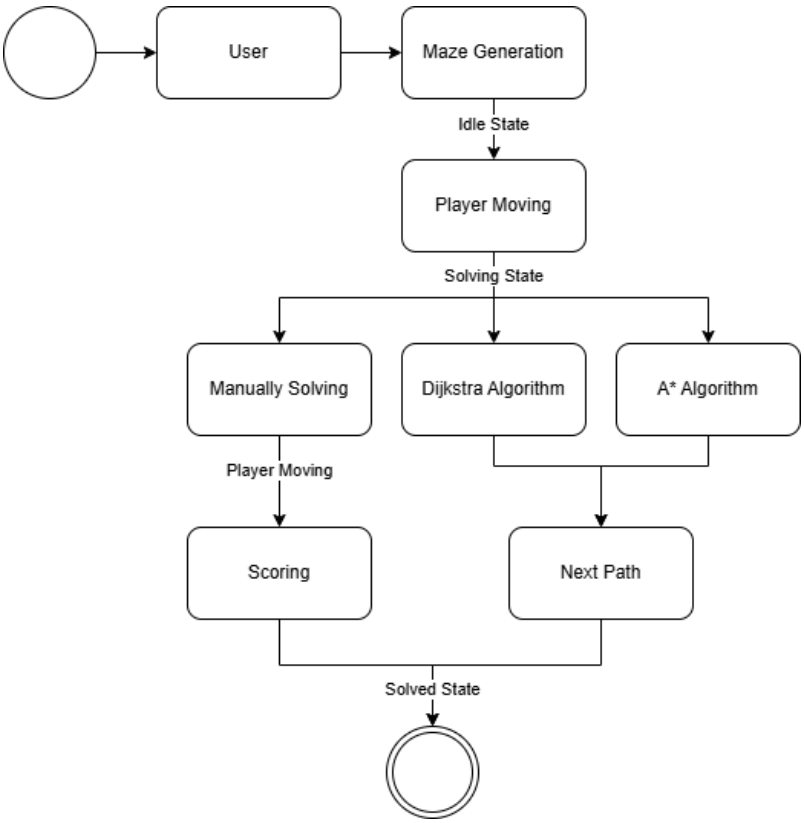



**Figure 3.4: State Diagram**

The sequence diagram for the "Maze Solver: Maze Game using Dijkstra & A* Algorithm" project demonstrates the interaction between three key components: the User, the System, and the Algorithm module. It begins with the user initiating a maze-solving request by pressing a button in the application interface. This triggers the system to handle the user input and determine which algorithm has been selected through the spinner component.

Once the selection is made, the system communicates with the corresponding algorithm class, passing the maze structure for processing. The algorithm executes its pathfinding logic and computes the optimal path from the start to the end of the maze. The computed path, along with any visited nodes for animation, is then sent back to the system.

The system then updates the custom view component (MazeView) to visualize the solution. This includes animating the traversal process and drawing the optimal path. If the user has manually attempted the maze, the system also compares the user's path with the algorithm's solution to calculate a score. The final result is displayed to the user as visual feedback, combining both educational and interactive elements.

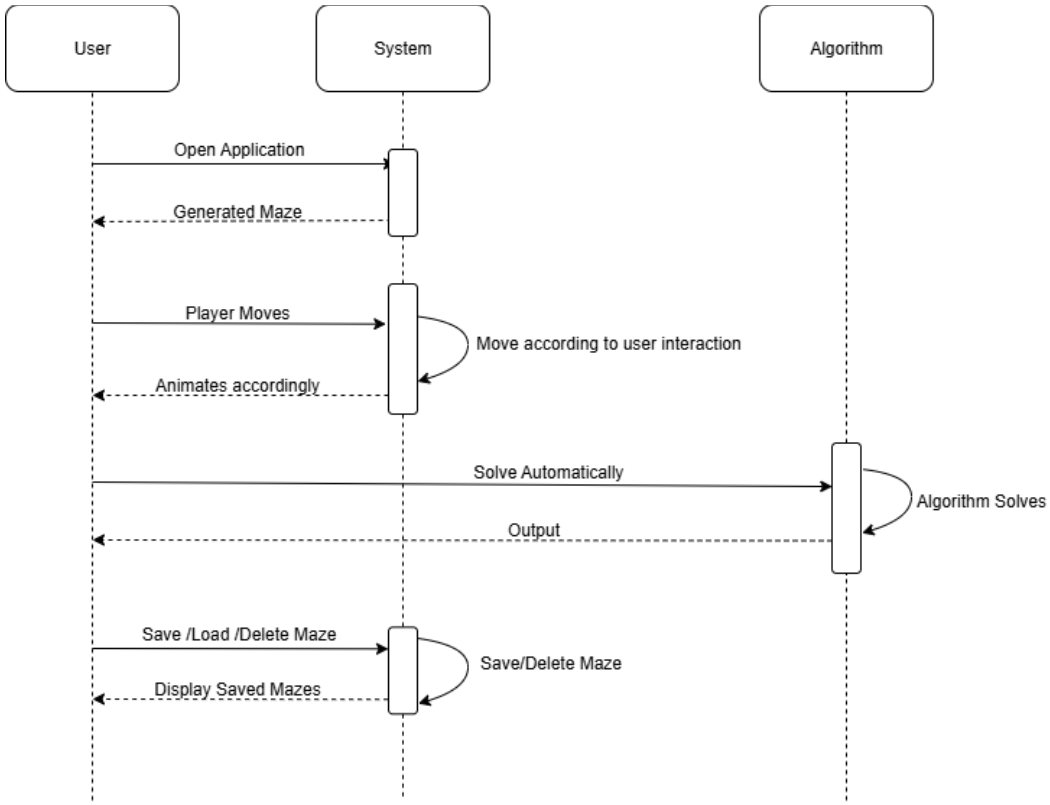



**Figure 3.5: Sequence Diagram**

### 3.1.5 Process Modeling: Activity Diagram

The activity diagram represents the flow of control from one activity to another, highlighting the dynamic aspects of the Maze Solver application. It showcases how the user interacts with the system—from launching the app to solving the maze using algorithms or manual navigation.

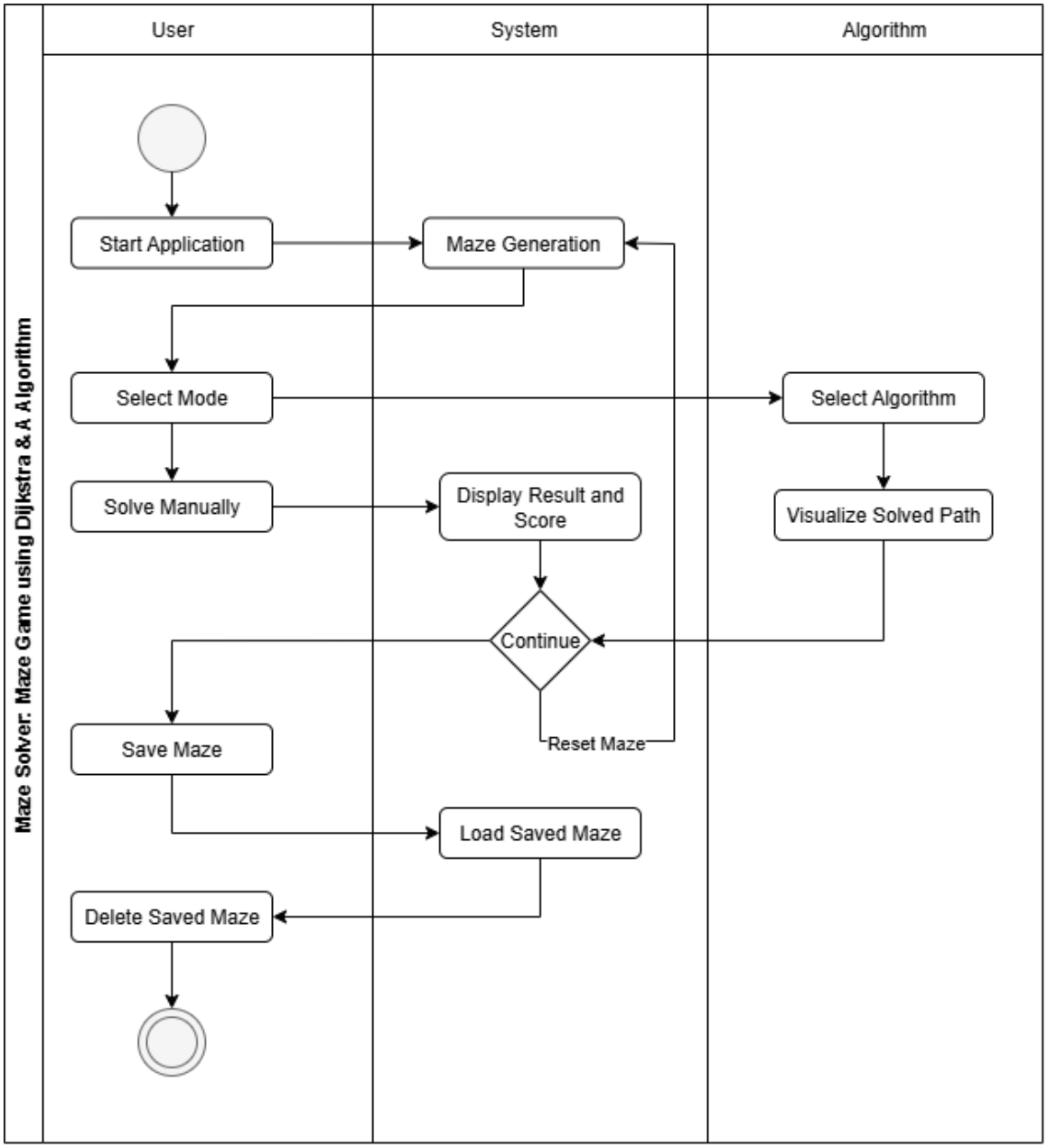



**Figure 3.6: Activity Diagram**

## 3.2 System Design

This section explains how the Maze Solver application is structured at a higher level. It includes the architectural layout of components, how they interact, and how the application is deployed and executed on Android devices.

### 3.2.1 Architectural Design

The Maze Solver is designed using a **modular architecture** that separates core functionalities into reusable components, making the system scalable and maintainable. The primary modules include:

- **UI Layer (MainActivity & XML Layout)**: This layer is responsible for handling user interactions and UI rendering. It provides the visual representation of the project.
- **Maze Logic (MazeView.java)**: This layer manages maze generation, player movement, and drawing of the mazes. It is responsible for generating random mazes through the use of recursive backtracking algorithm.
- **Algorithm Module (DijkstraSolver.java, AStarSolver.java)**: This layer is responsible for solving the mazes through implementation of maze-solving algorithms.
- **Storage Utility**: This layer is responsible for saving, loading, and deleting mazes in the local devices.

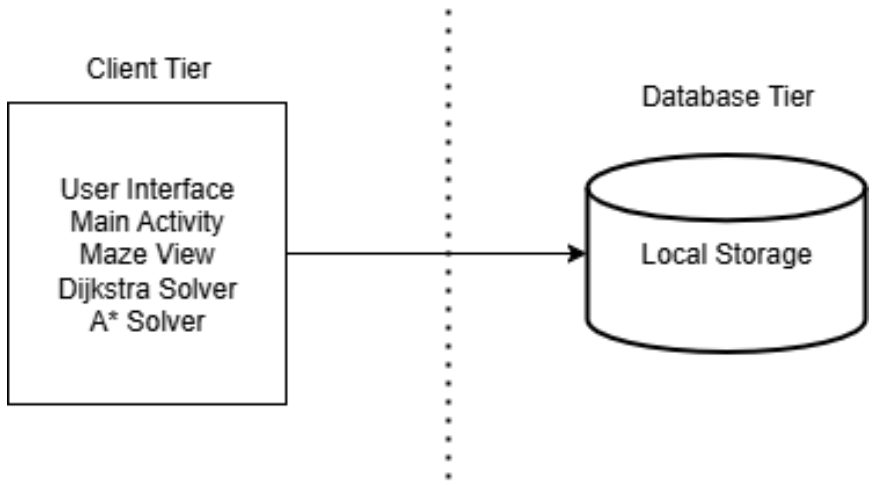



**Figure 3.7: System Architecture of Maze Solver: Maze Game using Dijkstra & A Algorithm**

### 3.2.2 Component Diagram

The Component Diagram shows how each major part of the application interacts. It helps visualize dependencies between components in the app.

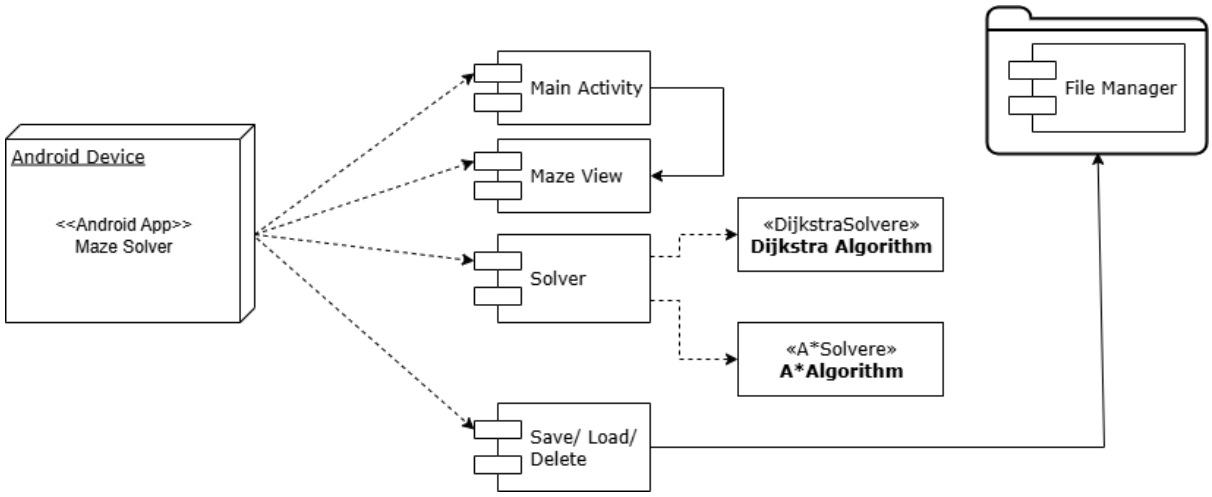



**Figure 3.8: Component Diagram**

### 3.2.3 Deployment Diagram

The **Deployment Diagram** shows the physical layout of software artifacts in the system. For this Android application, the deployment is simple and limited to a mobile device.

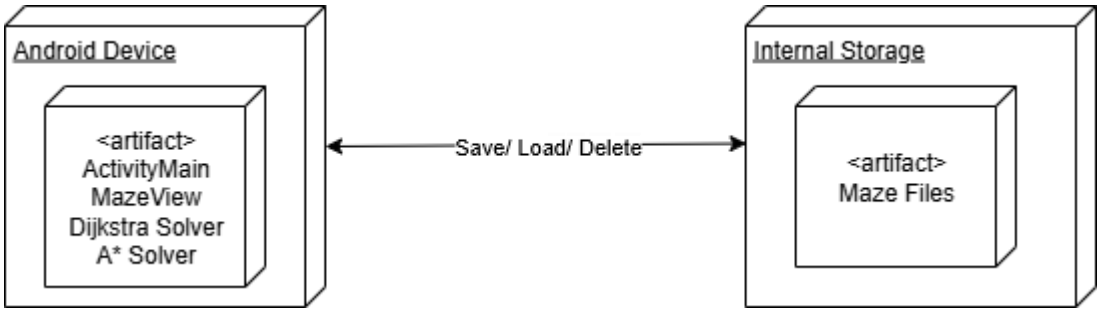



**Figure 3.9: Deployment Diagram**

## 3.3 Algorithm Details

The **Maze Solver: Maze Game using Dijkstra & A*** implements two of the most popular pathfinding algorithms to visualize and solve mazes: **Dijkstra's Algorithm** and **A* (A-Star) Algorithm**. These algorithms are integrated into the app to help users understand how different pathfinding strategies work and to allow comparison between them.

### 1. Dijkstra's Algorithm

Dijkstra's Algorithm is a widely used pathfinding technique that computes the shortest path between a starting point and all other nodes in a graph with non-negative weights. In this project the main purpose of this algorithm is to find the shortest path between two points in a graph with non-negative weights.

**How It Works**:

The algorithm starts by assigning a tentative distance value to every node: zero for the starting node and infinity for all others. It maintains a priority queue (usually a min-heap) that always picks the node with the smallest tentative distance for exploration. From the current node, it checks all unvisited neighbors and calculates the total distance from the start. If this new distance is shorter than the previously known distance, it updates it and records the current node as the predecessor. This process repeats until the goal node is reached or all reachable nodes are visited.

Although Dijkstra Algorithm guarantees the shortest path in any graph with non-negative edge weights, it explores all possible paths with equal priority, so it can be slower in larger or more complex mazes compared to other pathfinding algorithms.

Dijkstra Algorithm Uses the following formula to calculate its next node

$$distance[v]=min(distance[v], distance[u]+w(u,v))$$

Where:

- distance[v] is the current known shortest distance to node v
- u is the current node being visited
- w (u, v) is the weight (cost) of the edge between node u and node v
- If distance[u] + w (u, v) is less than the current distance[v], update it

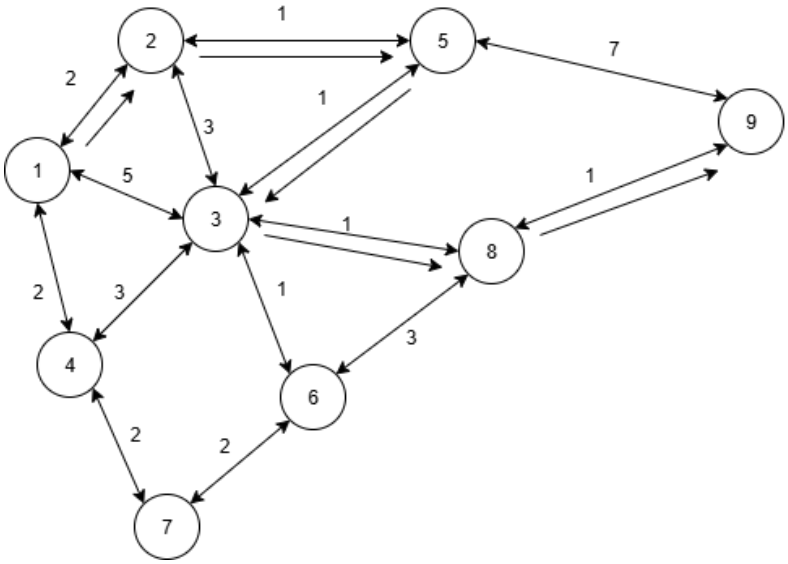



**Figure 3.10: Dijkstra Algorithm Visualization**

**2. A* (A-Star) Algorithm**

A* algorithm is an advanced pathfinding algorithm widely used in games, robotics, and AI. It is a modified version of Dijkstra's algorithm that allows it to find the shortest path faster. Like Dijkstra, it explores nodes in order of increasing cost, but A* tries to **guess** the direction of the goal and prioritizes paths that are likely to reach the goal sooner.

**How It Works**:

The algorithm works by maintaining a priority queue of nodes to explore, ranked by a cost function defined.

$$f(n) = g(n) + h(n).$$

Where, g(n) is the exact cost from the start node to the current node n, while h(n) is the heuristic function estimating the cost from n to the goal. The total cost f(n) allows A* to balance between nodes that are cheap to reach and nodes that appear closer to the target.

The most important feature of A* is the heuristic function h(n), which guides the search. In your maze game (grid-based with 4 directions), the Manhattan Distance is commonly used:

$$h(n)=|x_{goal}-x_n|+|y_{goal}-y_n|$$

This estimates how many moves it would take to reach the goal if there were no walls, by adding horizontal and vertical distances. It's fast to compute and works well in grid-based environments like mazes.

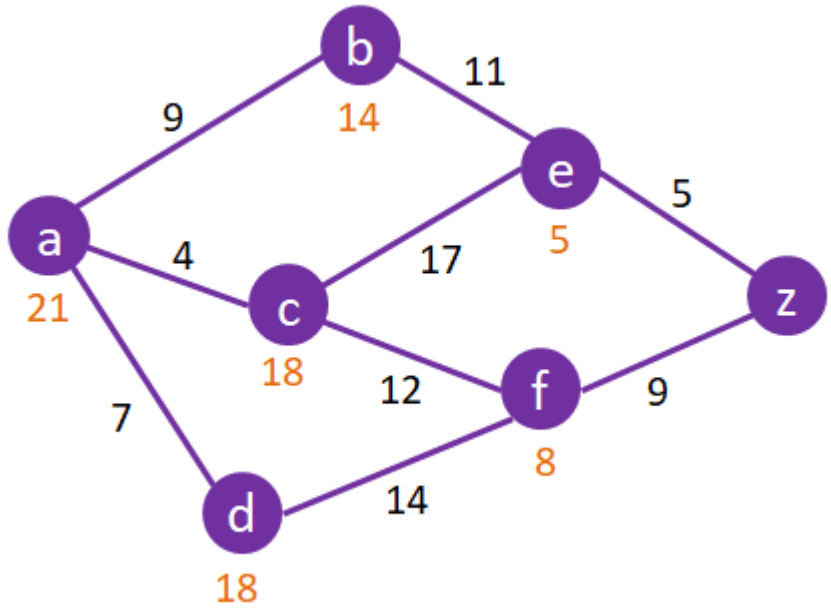



**Figure 3.11: A* Algorithm Visualization**

**Application Integration**

Both algorithms are implemented in separate solver classes (DijkstraSolver.java and AStarSolver.java). Users can:

- Select which algorithm to visualize using a spinner.
- Watch the animation of the solving process.
- Compare efficiency by observing the number of visited cells.

# CHAPTER 4: IMPLEMENTATION AND TESTING

## 4.1 Implementation

### 4.1.1 Tools Used

**Java**: Java was used as the core programming language to build the application's logic. It was used for handling user input, implementing pathfinding algorithms (Dijkstra and A*), managing file operations for saving/loading mazes, and controlling game mechanics like scoring and player movement.

**Android Studio:** Android Studio was used in this project to write code, design the UI, debugging, and testing. It provided a robust environment with features like real-time previews, emulator support, Gradle-based build tools, and version control integration.

**XML (Extensible Markup Language)**: XML was used to define the layout of the user interface. Buttons, views, spinners, and custom elements were organized using ConstraintLayout and LinearLayout. It allowed for precise control over UI positioning and responsiveness across different screen sizes.

**Android SDK (Software Development Kit)**: The SDK provides the necessary libraries, emulator images, and tools needed for Android development. It includes APIs for file handling, gesture detection, UI components, and system interaction.

**Java Serialization & File I/O**: ObjectOutputStream and ObjectInputStream were used to serialize and deserialize maze objects (2D Cell arrays), enabling saving and loading custom mazes with full structure preservation. Each maze is stored as a .maze file in internal storage.

**Emulator / Android Devices:** The app was tested on both emulators provided by Android Studio and physical Android devices to ensure real-world compatibility, responsiveness, and touch interaction accuracy.

### 4.1.2 Implementation Details of Modules

#### 1. Maze Generation Module

This module is responsible for the generation of the maze through the use of Recursive Backtracking Algorithm. This algorithm ensures there is always one unique path, and additional passages can be added for complexity.

```
private void generateMazeRecursiveBacktracking(int x, int y) {
        Cell current = grid[x][y];
        current.visited = true;

        List<int[]> directions = Arrays.asList(
                new int[]{0, -1}, new int[]{-1, 0},
                new int[]{0, 1}, new int[]{1, 0}
        );
        Collections.shuffle(directions);

        for (int[] dir : directions) {
            int nx = x + dir[0];
            int ny = y + dir[1];

            if (nx >= 0 && ny >= 0 && nx < cols && ny < rows &&
            !grid[nx][ny].visited) {
                if (dir[0] == 1) {
                    current.rightWall = false;
                    grid[nx][ny].leftWall = false;
                } else if (dir[0] == -1) {
                    current.leftWall = false;
                    grid[nx][ny].rightWall = false;
                } else if (dir[1] == 1) {
                    current.bottomWall = false;
                    grid[nx][ny].topWall = false;
                } else if (dir[1] == -1) {
                    current.topWall = false;
                    grid[nx][ny].bottomWall = false;
                }
                generateMazeRecursiveBacktracking(nx, ny);
            }
        }
    }
```

**Figure 4.1: Recursive Backtracking Algorithm**

## 2. Algorithm Modules

- **DijkstraSolver.java**: The DijkstraSolver is used to implement Dijkstra's algorithm which returns the shortest path and shows all the visited nodes.

```java
public Result solve() {
        Map<MazeView.Cell, MazeView.Cell> parentMap = new HashMap<>();
        Queue<MazeView.Cell> queue = new LinkedList<>();
        Set<MazeView.Cell> visited = new HashSet<>();

        MazeView.Cell start = grid[0][0];
        MazeView.Cell goal = grid[cols - 1][rows - 1];

        queue.add(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            MazeView.Cell current = queue.poll();

            if (current.equals(goal)) break;

            for (MazeView.Cell neighbor : getNeighbors(current)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    parentMap.put(neighbor, current);
                    queue.add(neighbor);
                }
            }
        }

        // Reconstruct path
        List<MazeView.Cell> path = new ArrayList<>();
        MazeView.Cell step = goal;
        while (step != null && parentMap.containsKey(step)) {
            path.add(step);
            step = parentMap.get(step);
        }

        path.add(start);
        Collections.reverse(path);

        return new Result(path, visited);
    }
```

**Figure 4.2: Dijkstra Algorithm**

**AStarSolver.java**: The AStarSolver is used to implement A* algorithm with Manhattan Distance as heuristic, which is the updated or modified version of the Dijkstra Algorithm

```
public Result solve() {
        Set<MazeView.Cell> visited = new HashSet<>();
        Map<MazeView.Cell, MazeView.Cell> cameFrom = new HashMap<>();
        Map<MazeView.Cell, Integer> gScore = new HashMap<>();
        PriorityQueue<MazeView.Cell> openSet = new PriorityQueue<>
        (Comparator.comparingInt(c -> gScore.getOrDefault(c, Integer.MAX_VALUE) +
        heuristic(c)));
        MazeView.Cell start = grid[0][0];
        MazeView.Cell goal = grid[cols - 1][rows - 1];
        gScore.put(start, 0);
        openSet.add(start);
        while (!openSet.isEmpty()) {
            MazeView.Cell current = openSet.poll();
            visited.add(current);
            if (current.equals(goal)) break;
            for (MazeView.Cell neighbor : getNeighbors(current)) {
                int tentativeGScore = gScore.getOrDefault(current, Integer.MAX_VALUE) + 1;
                if (tentativeGScore < gScore.getOrDefault(neighbor, Integer.MAX_VALUE)) {
                    cameFrom.put(neighbor, current);
                    gScore.put(neighbor, tentativeGScore);
                    if (!openSet.contains(neighbor)) {
                        openSet.add(neighbor);
                    }
                }
            }
        }
        List<MazeView.Cell> path = reconstructPath(cameFrom, goal);
        return new Result(path, visited);
    }
```

**Figure 4.3: A* Algorithm**

### 3. Maze Save/Load/Delete Module

This module is responsible for saving the maze file in the local storage when the save button is clicked, loading the saved file when the load button is clicked and deleting the maze when the delete button is clicked.

Below is the code for saving the file in the local storage, retrieving them from the saved location and deleting them.

```
saveButton.setOnClickListener(v -> {
            EditText input = new EditText(this);
            new AlertDialog.Builder(this)
                    .setTitle("Save Maze As")
                    .setMessage("Enter a name for this maze:")
                    .setView(input)
                    .setPositiveButton("Save", (dialog, which) -> {
                        String name = input.getText().toString().trim();
                        if (!name.isEmpty()) {
                            mazeView.saveMaze(this, name);
                        }
                    })
                    .setNegativeButton("Cancel", null)
                    .show();
        });
```

**Figure 4.4: Save Maze**

```
loadButton.setOnClickListener(v -> {
            Set<String> saved = MazeView.getSavedMazeNames(this);
            if (saved.isEmpty()) {
                Toast.makeText(this, "No saved mazes.", Toast.LENGTH_SHORT).show();
                return;
            }
            String[] mazeNames = saved.toArray(new String[0]);
            new AlertDialog.Builder(this)
                    .setTitle("Load Maze")
                    .setItems(mazeNames, (dialog, which) -> mazeView.loadMaze(this, mazeNames[which]))
                    .show();
        });
```

**Figure 4.5: Load Maze**

```
deleteButton.setOnClickListener(v -> {
            Set<String> saved = MazeView.getSavedMazeNames(this);
            if (saved.isEmpty()) {
                Toast.makeText(this, "No saved mazes to delete.", Toast.LENGTH_SHORT).show();
                return;
            }
            String[] mazeNames = saved.toArray(new String[0]);
            new AlertDialog.Builder(this)
                    .setTitle("Delete Maze")
                    .setItems(mazeNames, (dialog, which) -> {
                        MazeView.deleteMaze(this, mazeNames[which]);
                        Toast.makeText(this, "Deleted: " + mazeNames[which], Toast.LENGTH_SHORT).show();
                    })
                    .show();
        });
```

**Figure 4.6: Delete Maze**

## 4.2 Testing

Testing is the process of detecting errors. It performs a very crucial role for quality assurance and for ensuring the reliability of the software. The results of testing are used later on during maintenance also. Testing requires a lot of time and labor.

### 4.2.1 Test Cases for Unit Testing

**Table 4.1: Test Case for Recursive Backtracking Algorithm**

| Test Case ID: TC-001 | | | | | |
|---|---|---|---|---|---|
| Module Name: Maze Generation | | | | | |
| Test Title: Maze Generation with at least one solution | | | | | |
| Description: This test verifies that the maze generated has at least one solution from starting point to end point. | | | | | |
| Test Steps:<br>1. Open the Application<br>2. Reset the current maze | | | | | |
| **Test Case** | **Test Scenario** | **Input** | **Expected Outcome** | **Actual Outcome** | **Status** |
| 1 | Generate maze on app start | App Launch | A fully connected maze with a path from start to end | Maze generated with solution | Pass |
| 2 | Generate maze after clicking Reset | "Reset Maze" button clicked | New maze different from previous one | New maze generated | Passed |
| 3 | Ensure maze has at least one solution | - | At least one solution exists | Solution exists | Passed |

| | | | | | |
|---|---|---|---|---|---|
| 4 | Save and reload generated maze | Save and load buttons used | Reloaded maze structure matches previously saved one | Successfully loaded saved maze | Passed |

**Table 4.2: Test Case for Dijkstra Algorithm**

| Test Case: TC-002 | | | | | |
|---|---|---|---|---|---|
| Module Name: Dijkstra Solver | | | | | |
| Test Title: Shortest Path Calculation using Dijkstra | | | | | |
| Description: This test ensures that Dijkstra's algorithm finds the shortest path from the start cell (0,0) to the end cell (cols-1, rows-1) and properly visualizes visited nodes. | | | | | |
| Test Steps:<br>1. Open the application<br>2. Reset the current maze<br>3. Select Dijkstra from the algorithm selector<br>4. Click “Solve” button | | | | | |
| **Test Case** | **Test Scenarios** | **input** | **Expected Outcome** | **Actual Outcome** | **Status** |
| 1 | Test Shortest path is drawn correctly in the maze | Solve button Clicked | Path traced from start to end using Dijkstra’s logic | Path correctly drawn | Passed |
| 2 | Verify all reachable nodes are visited during solving | - | All reachable nodes visited; shortest path selected | All nodes visited appropriately | Passed |
| 3 | Validate scoring when player's path is compared with Dijkstra's path | Compare to player path | Evaluates similarity and returns score (0–10) | Score displayed accordingly | Passed |

**Table: 4.3: Test Case for A*(A Star) Algorithm**

| Test Case ID: TC-003 | | | | | |
|---|---|---|---|---|---|
| Module Name: AStarSolver | | | | | |
| Test Title: Heuristic-based Shortest Path Finding using A* | | | | | |
| Description: This test verifies that the A* algorithm calculates the shortest path using Manhattan distance as the heuristic and animates exploration efficiently. | | | | | |
| Test Steps:<br>1. Open the Application<br>2. Select "A*" from the algorithm selector<br>3. Click the "Solve" button | | | | | |
| **Test Case** | **Test Scenario** | **Input** | **Expected Outcome** | **Actual Outcome** | **Status** |
| 1 | Check if A* finds the shortest path in default maze | Solve Clicked | A* finds path from start to end using heuristic + cost | Shortest path drawn | Passed |
| 2 | Verify accuracy maze where shortest path is obvious | Simple maze | Minimal steps with clear visualization | Correct path shown | Passed |
| 3 | Compare results with Dijkstra to verify optimality and speed | Compare to Dijkstra | Similar or equal path length; fewer nodes possibly visited | Algorithm performs efficiently | Passed |

### 4.2.2 Test Case for System Testing

**Table 4.4: System Testing Table**

| S. N | Test Case Description | Steps | Expected Outcome | Actual Outcome | Result |
|---|---|---|---|---|---|
| 1 | Solving a Maze | Launch app → Tap "Solve Maze" | Shows visited cells and solution path | Shows the Solution | Passed |
| 2 | Reset Function | Tap "Reset Maze" | Maze regenerates and clears paths | Generates new Maze | Passed |

| | | | | | |
|---|---|---|---|---|---|
| 3 | Save Maze | Tap "Save" → Give the name for the maze | Toast appears "Maze Saved" | Mase Saved | Passed |
| 4 | Load Maze | Load → Select Saved Maze | Same maze structure is loaded | Saved Maze Loaded | Passed |
| 4 | Score Evaluation | Play and reach goal | Score reflects path similarity | Score Appeared | Passed |
| 5 | Delete Maze | Tap "Delete Maze" → Select file | Maze file is removed and UI updates | Maze Deleted | Passed |
| 6 | Multi-device Testing | Run on emulator and phone (e.g., 6", 10") | Layout adjusts, controls responsive | Controls responsive | Passed |
| 7 | UI Responsiveness | Swipe gestures → Path draw → Button taps | Smooth performance, no lag | Smoothly operated | Passed |

## 4.3 Result Analysis

The implementation and testing phases of the *Maze Solver: Maze Game using Dijkstra & A** Algorithm* project demonstrate that the system meets its core functional requirements effectively. The application was evaluated based on user experience, algorithm accuracy, animation performance, and feature completeness.

**1. Algorithm Efficiency**

- **Dijkstra's Algorithm** consistently produced the shortest path without using heuristics, ideal for uniform-cost mazes.
- **A*** Algorithm performed slightly faster due to its heuristic (Manhattan distance).
- Both algorithms successfully returned the optimal path from the start cell to the end cell under all test conditions.

**2. User Interaction & Game Play**

- The swipe-based manual solving was intuitive and responsive, allowing players to explore the maze naturally.
- The **scoring mechanism** effectively motivated users to find the shortest possible path by comparing their route with the optimal one.
- The application correctly identified the goal state and triggered a celebration message with an accurate score.

**3. Visualization and Animation**

- The pathfinding animation provided a visual understanding of the algorithm's decision-making process.
- Visited nodes were marked in **yellow**, and the final path was drawn in **cyan**, making the difference visually clear.
- The animation was consistent across different Android devices tested.

**4. Maze Management**

- Users could **save**, **load**, and **delete** multiple mazes using friendly dialog interactions.
- The internal storage system worked reliably to persist and retrieve maze data by name.
- This functionality enhanced replayability and personalization of the maze-solving experience.

**5. Overall System Stability**

- The app maintained high responsiveness and avoided crashes even under stress conditions (e.g., rapid swipes, loading multiple mazes).
- UI elements adapted well to screen sizes and densities.
- No memory leaks or performance bottlenecks were observed during testing.

# CHAPTER 5: CONCLUSION AND FUTURE RECOMMENDATION

## 5.1 Conclusion

The project titled **"Maze Solver: Maze Game using Dijkstra & A* Algorithm"** successfully meets its objectives of providing an interactive and educational Android application where users can manually solve or automatically solve a generated maze using popular pathfinding algorithms.

Through the implementation of both **Dijkstra's Algorithm** and the **A*** Algorithm, the application demonstrates the difference between uninformed and informed search methods. The project helps users visualize how algorithms traverse paths and how optimal solutions are computed, thus promoting learning through interaction.

In addition, the game-like interface with gesture-based navigation, animation of explored paths, a scoring system based on path comparison, and the ability to save/load/delete custom mazes enriches the user experience and highlights practical use of algorithms in real applications.

Overall, this project combines logic, visualization, interactivity, and design to create a meaningful tool that can serve both as a learning platform and a fun game.

## 5.2 Future Recommendation

While the current version of the Maze Solver application is fully functional and feature-rich, the following improvements are recommended for future development:

1. **Custom Maze Editor:**
    - Allow users to draw or design their own maze layout manually using touch input.
2. **Maze Difficulty Levels:**
    - Add support for different maze sizes and difficulty levels to offer more variety and challenge.
3. **Leaderboard Integration:**

- Implement a scoring system with leaderboard support to compare scores globally or locally among users.

4. **Multiplayer Maze Challenge:**
    - Add a feature where two players can race to solve the same maze in real-time, comparing strategies and paths.
5. **Dark Mode Optimization:**
    - Improve visual contrast and design specifically for AMOLED displays in dark mode.

# REFERENCES

[1] A. Roy and S. Ghosh, *"Pathfinding Algorithms for Maze Solving: A Comparative Study,"* International Journal of Computer Applications, vol. 180, no. 35, pp. 25–30, Dec. 2022. DOI: 10.5120/ijca2022923773

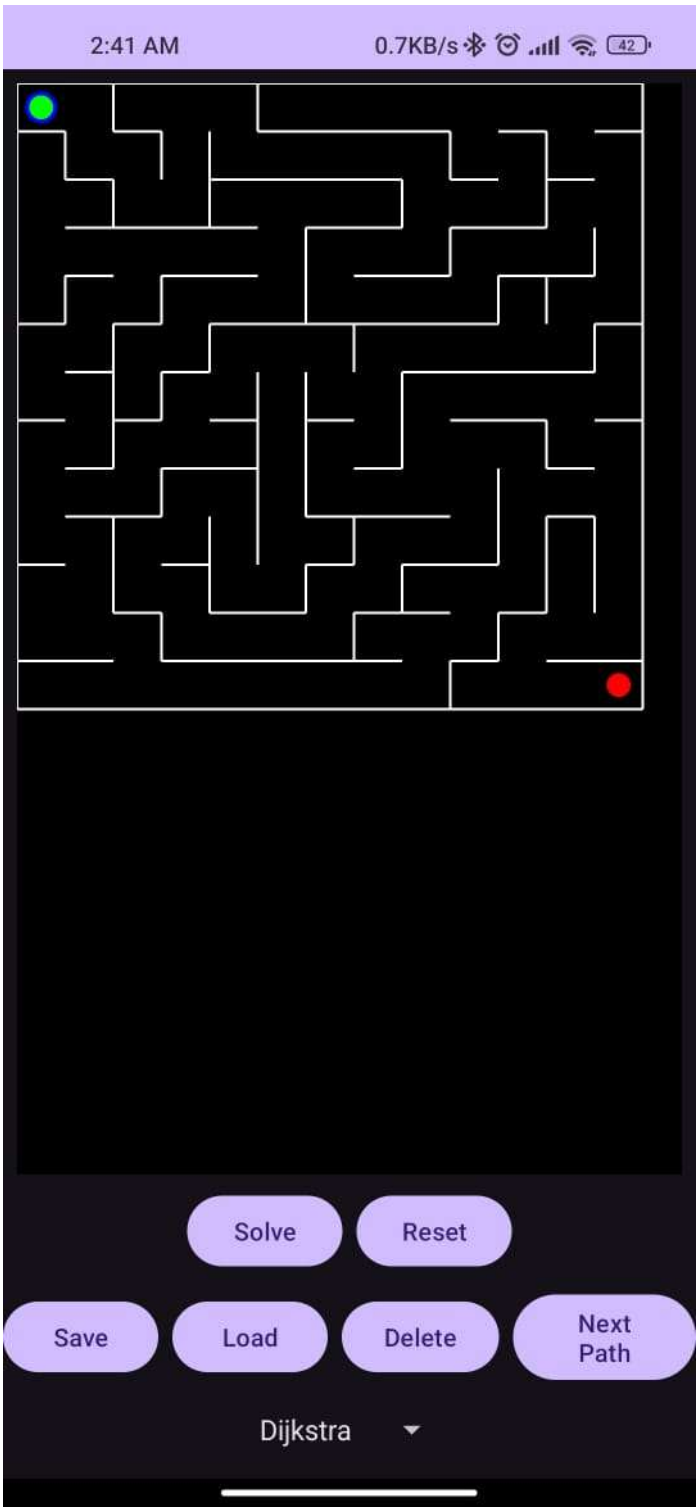
[2] M. Khan and L. Zheng, *"Mobile Applications for Interactive Learning: Enhancing User Engagement through Gamification,"* Journal of Educational Computing Research, vol. 58, no. 4, pp. 902–919, Apr. 2021. DOI: 10.1177/0735633120906317

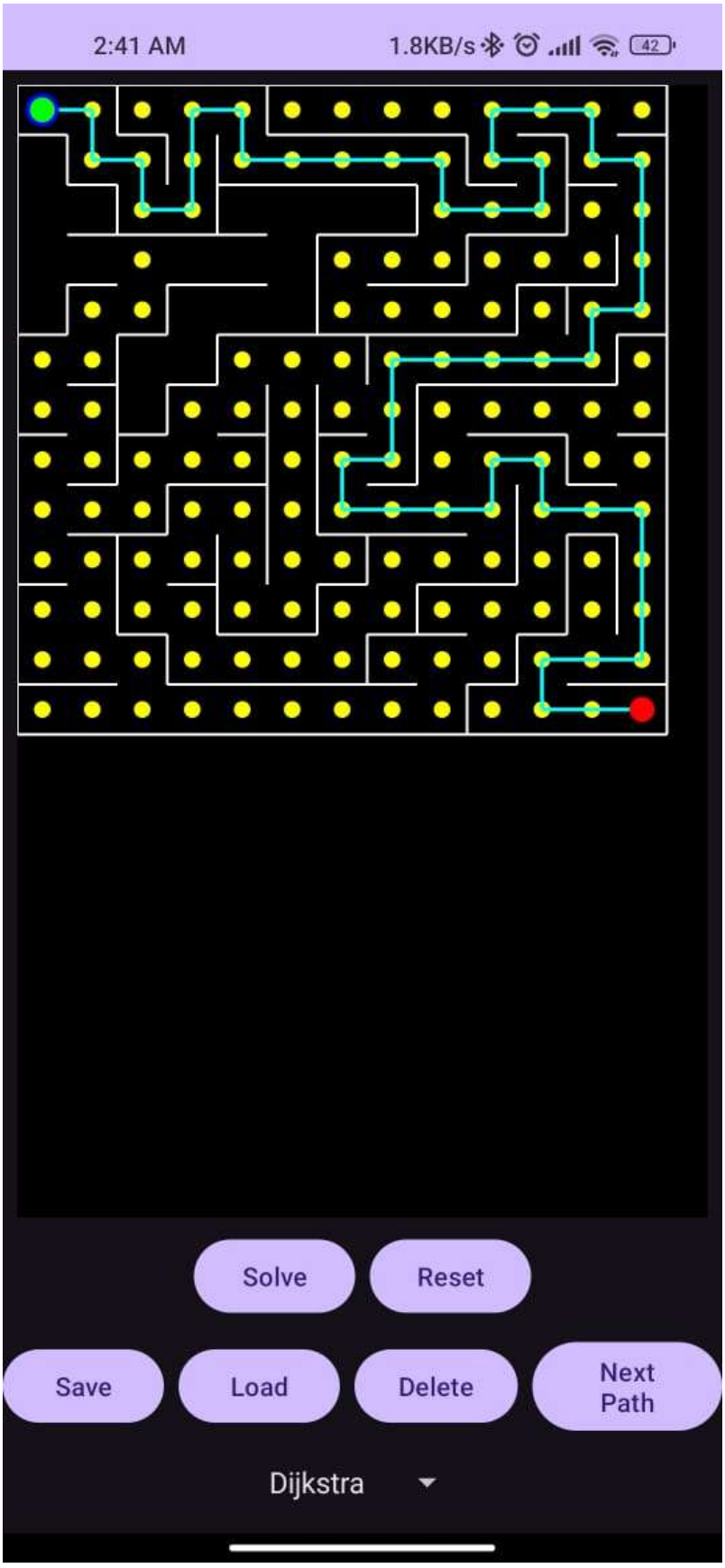
[3] J. Patel and A. Sharma, *"Customizable Digital Mazes: Enhancing User Creativity in Algorithmic Problem Solving,"* Journal of Interactive Learning Environments, vol. 29, no. 2, pp. 145–158, Jan. 2021. DOI: 10.1080/10494820.2020.1719957

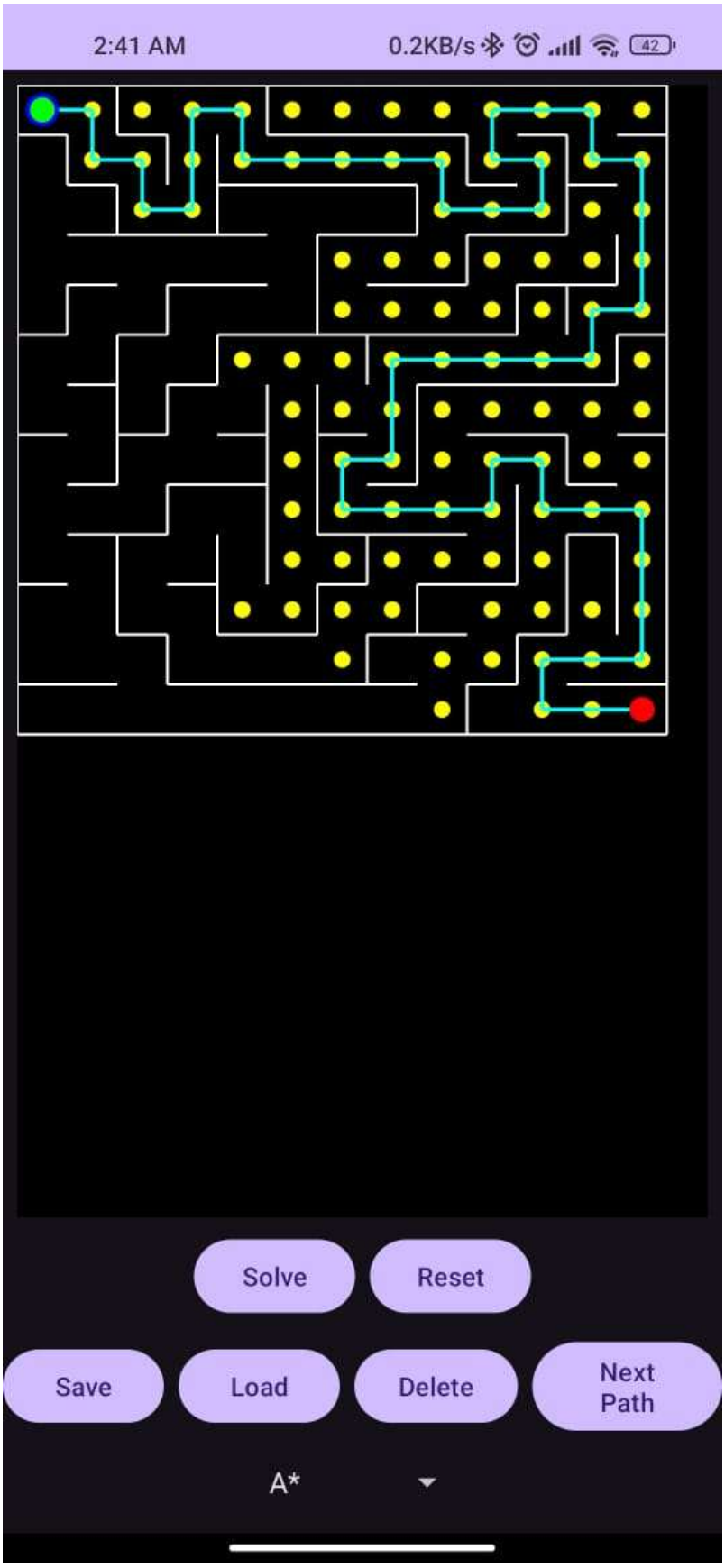
[4] L. Zhang and B. Wang, *"Artificial Intelligence in Puzzle Games: The Role of Algorithmic Complexity,"* ACM Transactions on Multimedia Computing, Communications, and Applications, vol. 17, no. 3, pp. 1–18, July 2021. DOI: 10.1145/3457163

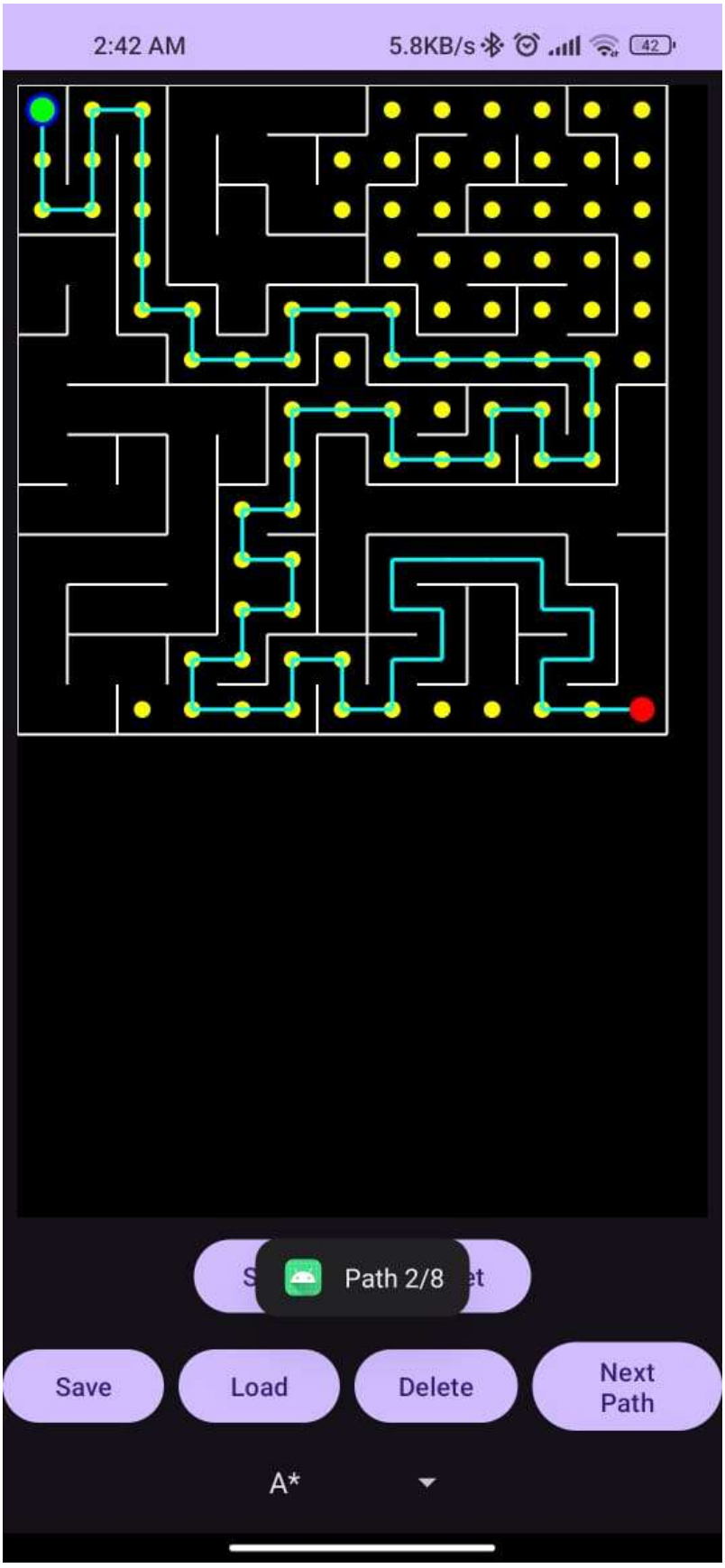
[5] S. Smith, R. Taylor, and M. Johnson, *"Comparison of Pathfinding Algorithms for Maze Solving: Dijkstra vs. A*,"* IEEE Transactions on Games, vol. 12, no. 2, pp. 150–158, Mar. 2020. DOI: 10.1109/TG.2019.2958764

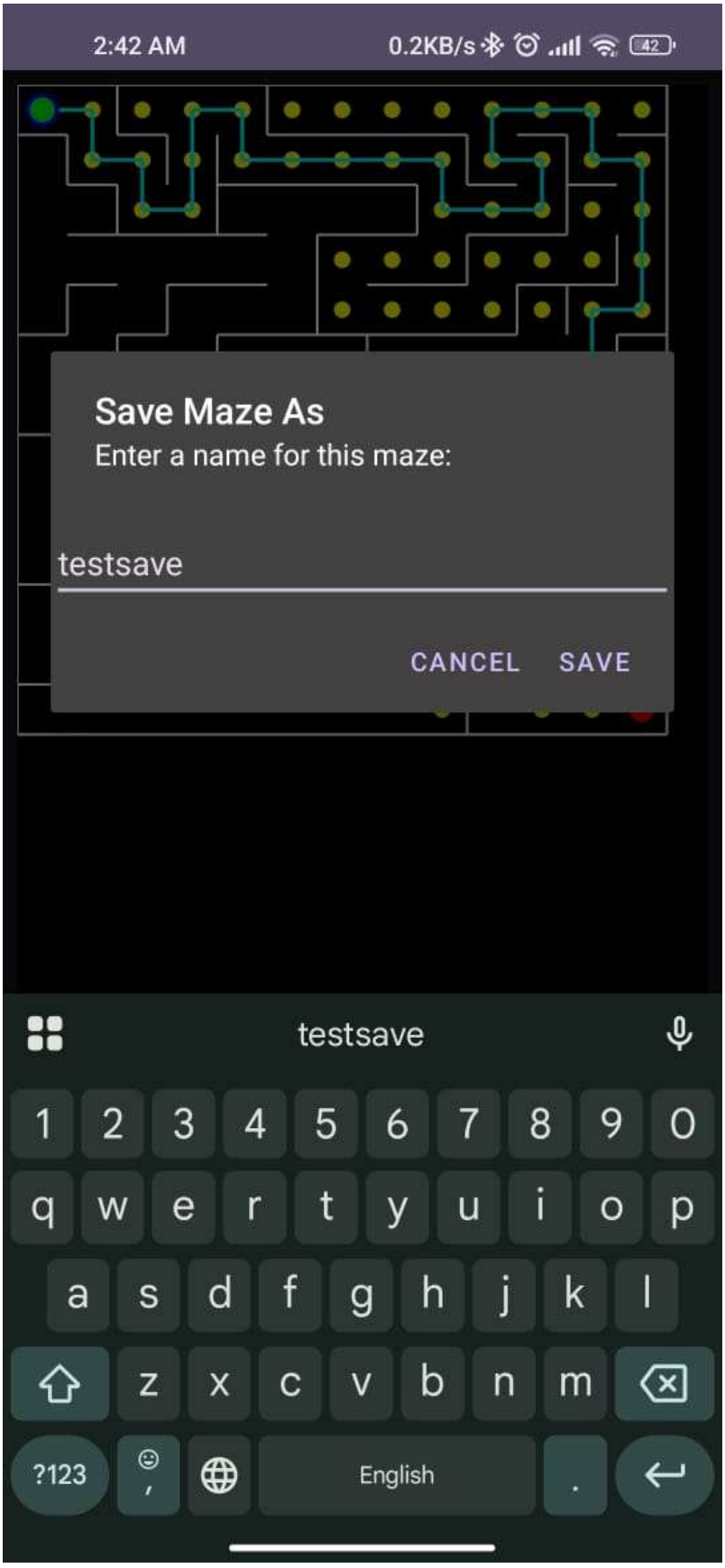
## Appendix

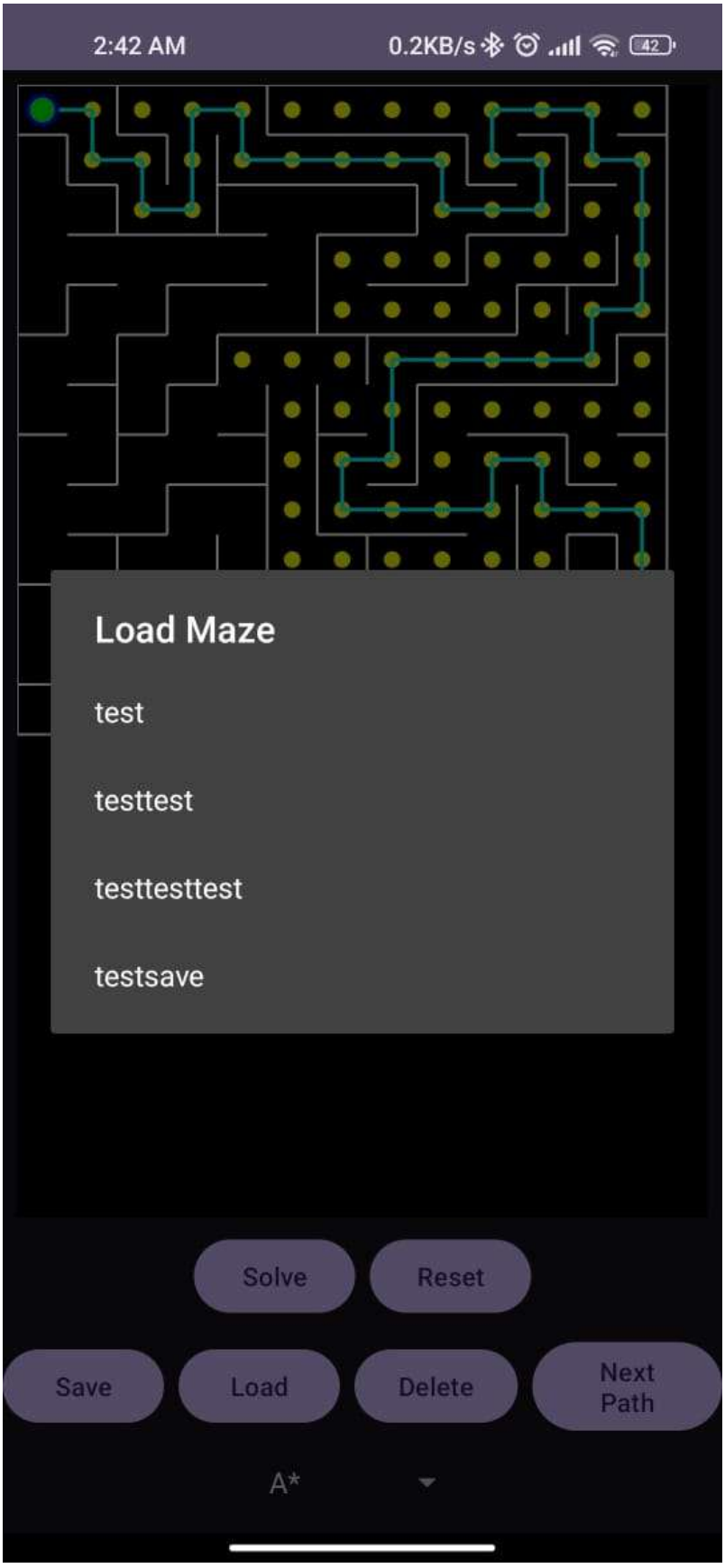

2:41 AM
0.7KB/s
Solve
Reset
Save
Load
Delete
Next Path
Dijkstra

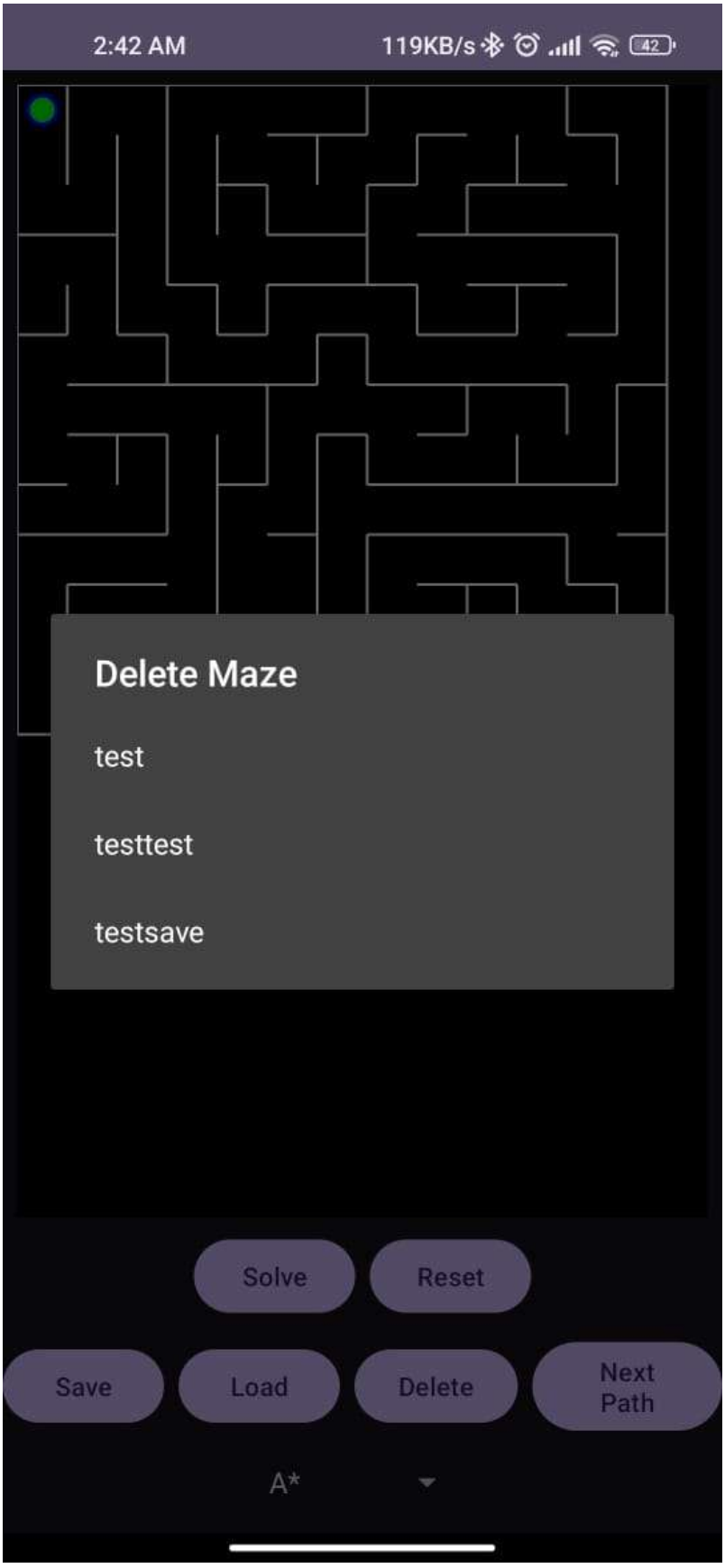

2:41 AM
1.8KB/s
42
Solve
Reset
Save
Load
Delete
Next Path
Dijkstra

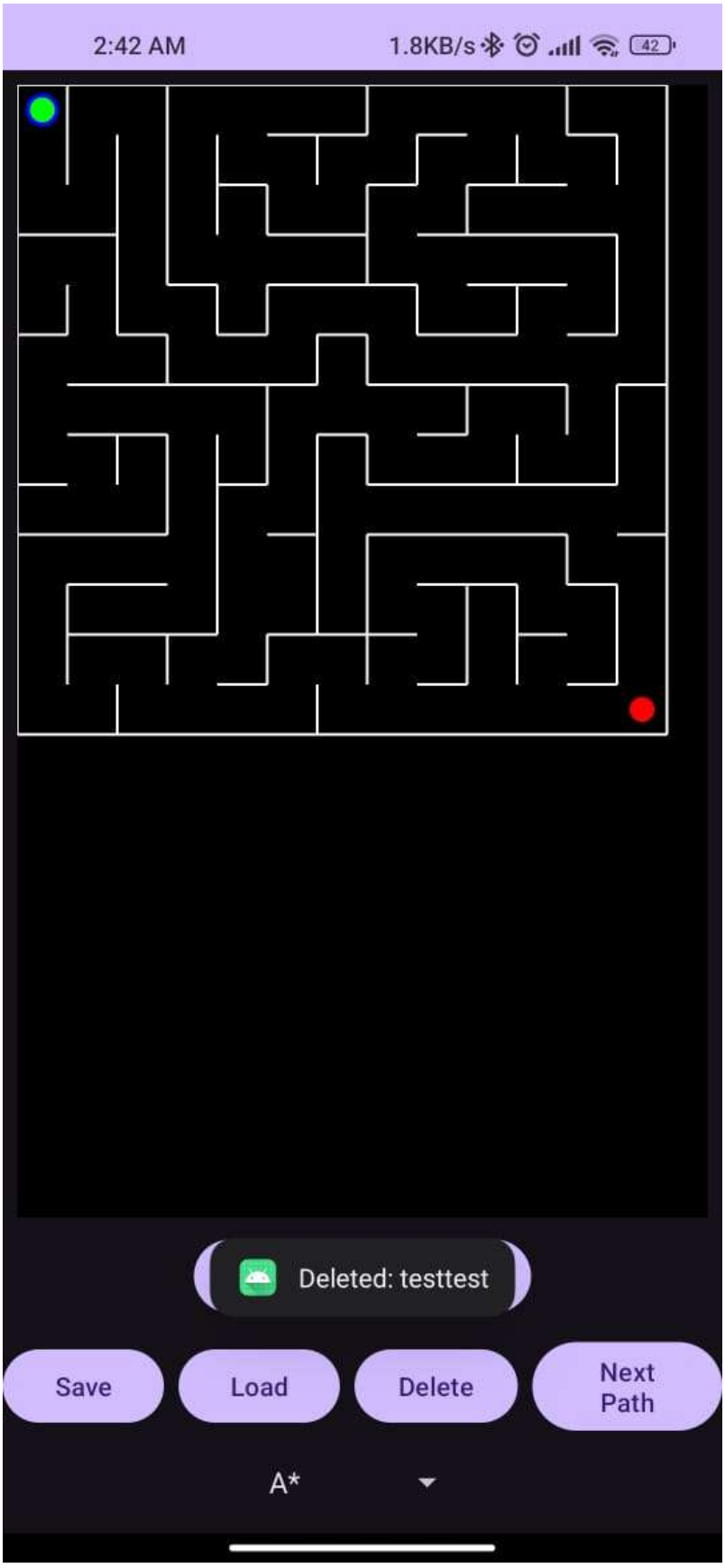

2:41 AM
0.2KB/s
42
Solve
Reset
Save
Load
Delete
Next Path
A*


2:42 AM
5.8KB/s
Path 2/8
Save
Load
Delete
Next Path
A*


2:42 AM
0.2KB/s
Save Maze As
Enter a name for this maze:
testsave
CANCEL
SAVE
testsave
1 2 3 4 5 6 7 8 9 0
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
English


2:42 AM
0.2KB/s
Load Maze
test
testtest
testtesttest
testsave
Solve
Reset
Save
Load
Delete
Next Path
A*


2:42 AM
119KB/s
Delete Maze
test
testtest
testsave
Solve
Reset
Save
Load
Delete
Next Path
A*


2:42 AM
1.8KB/s
42
Deleted: testtest
Save
Load
Delete
Next Path
A*