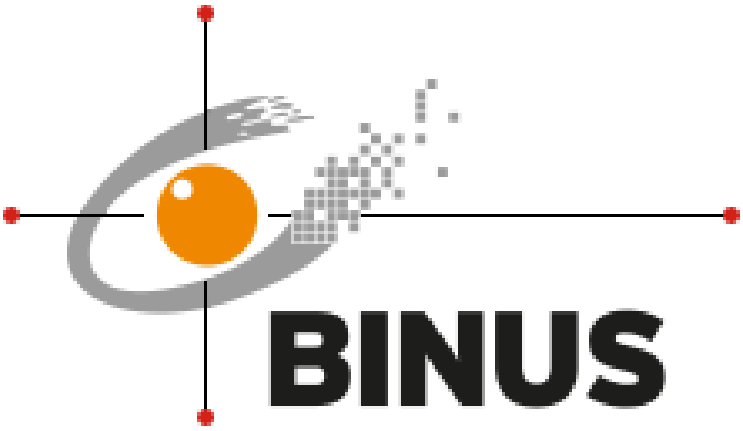


BINUS
UNIVERSITY
INTERNATIONAL

MAZE MAKER + SOLVER
PROJECT REPORT

Harris Ekaputra Suryadi - 2802400502

Michael Arianno Chandrarieta - 2802499711

Muhammad Ryan Ismail Putra - 2802522733

Binus University International Program

Computer Science

17 March 2025

Table of Contents

Table of Contents 2
Background 4
Problem Definition 5
Solution 7
- **OOP Part:** 7
  - **Inheritance** 7
  - **Interfaces** 7
  - **Polymorphism** 7
  - **What Our Program Does** 8
  - **How We Used OOP** 8
    - **Main** 8
    - **Algorithm** 8
    - **Node** 9
  - **How Our Program Works** 9
  - **Instruction Manual** 10
    - **1. Introduction** 10
    - **2. System Requirements** 10
    - **3. Installation and Setup** 10
    - **4. Application Usage** 11
      - **4.1 Launching the Application** 11
      - **4.2 Interacting with the Maze** 11
      - **4.3 Running Algorithms** 11
      - **4.4 Saving and Loading Mazes** 11
    - **5. Algorithms Overview** 12
      - **Algorithm.java (Default)** 12
      - **Algorithm2.java (Alternative Implementation)** 12
      - **To use Algorithm2:** 12
    - **6. Source Code Structure** 12
    - **7. Maze File Format** 13
    - **8. Output Files** 13
    - **9. Notes** 13
- **DS Part:** 14

Results 18
- **Analysis:** 20
- **Growth Graph:** 22
- **Best for our case:** 24

Documentation..... 25
Class Diagram:..... 25
Evidence of Working Program:..... 27
References..... 29
Appendix..... 30

Background

A maze is a path or a collection of paths, typically from an entrance to a goal. The word refers to both to branching tour puzzles through which the solver must find a route and to simpler non-branching (“unicursal”) patterns that lead unambiguously through a convoluted layout to a goal. The term “labyrinth” is generally synonymous with “maze”, but can also suggest a unicursal pattern. The pathways and walls in a maze are typically fixed, but puzzles in which the walls and paths can change during the game are also categorized as mazes or tour puzzles.

Mazes have been used throughout history for various purposes, from entertainment and meditation to religious and cultural symbolism. Ancient civilizations, such as the Greeks, incorporated mazes into mythology, with the most famous example being the Labyrinth of Crete, which housed the mythical Minotaur. In medieval times, mazes were often found on cathedral floors, designed as walking paths for spiritual reflection. Today, mazes remain popular in amusement parks, escape rooms, and video games, where they challenge players’ problem-solving skills and sense of direction.

Modern mazes come in forms, including hedge mazes and digital mazes, each offering a unique challenge. Some mazes are designed to test intelligence and patience, while others serve as artistic expressions or social attractions. With advancements in technology, interactive and dynamic mazes have emerged, allowing for changing pathways and new levels of difficulty. Whether used for fun, education, or symbolic representation, mazes continue to captivate and challenge people of all ages across different cultures. Mazes also play a significant role in psychology and cognitive science, often used in experiments to study learning, memory, and problem-solving abilities. The famous rat maze experiments have been instrumental in understanding spatial navigation and the effects of reinforcement and conditioning.

Similarly, human maze tests, such as the Morris water maze, help researchers analyze brain function and disorders like Alzheimer's disease. The challenge of navigating a maze mirrors real-life decision-making processes, making it a valuable tool in both research and education. The application of ‘finding and solving’ a maze is very well sought after and useful…

Problem Definition

A maze-maker + solver are designed to create their own maze, but they also need to connect and find the most efficient path, from the starting point to the goal of solving it. Solving a maze efficiently is a common computational problem that requires a balance between the speed and the accuracy of the algorithm. The maze solver needs to avoid the path that leads to dead ends while minimizing the time and length of the path to get to the goal. This essay will explain the problems that could affect the efficiency of the algorithm.

First of all, a maze-maker and solver will need the maze map itself, which will also be generated by the algorithm. The generated maze will affect the algorithm depending on its complexity. The structure of the maze itself can significantly impact the algorithm's ability to solve the maze. Indeed, the more complex the maze is, the more time it takes for the algorithm to solve it.

The efficiency of the algorithm itself also affects the time and memory needed to solve the maze, which refers to the correctness and the accuracy of the algorithm. Many types of algorithms can be used to solve a maze, and each of them has a different, varying performance in terms of speed and memory usage. Some of the algorithms could spend less time than others, but they require a bigger memory. On the other hand, some algorithms will need more time, but they use less memory to solve the maze. The algorithm must consistently find a valid and optimal path, if one exists, while handling edge cases like isolated sections or multiple goal points.

Another crucial thing that can also be a problem for the algorithm is the real-time execution. Find the best path to solve the maze, it will require real-time results, so we will know which one is the most efficient path that has been generated by the algorithm. If we don't produce the results in real time, then the accuracy might not be accurate. Thus, optimizing and ensuring the runtime of the chosen algorithm is really crucial.

Last but not least is scalability, which refers to the ability of the algorithm to efficiently and effectively handle increasingly complex and larger mazes. Some of the algorithms perform well on small mazes but struggle with larger and more complex mazes. This is why, on this

project, we need to test the algorithm at different types of complexity levels. It will ensure the performance of the algorithm in different types of mazes, without a significant performance degradation.

In conclusion, there are many problems that we might face in creating this algorithm. Starting from the complexity of the maze, the more complex the maze, the more time and usage that might be required for the algorithm to solve the maze. Another problem is the algorithm's efficiency and effectiveness in solving the maze. There are many types and scales of the algorithm, making it hard to choose which one is the best without testing it. This is why testing the algorithm in many conditions is crucial.

Solution

OOP Part:

Inheritance

- **Main extends Canvas**
  This means Main inherits all the methods and properties of the Canvas class (from java.awt), allowing it to be used as a drawing surface.

Interfaces

- **Main implements Runnable**
  This allows Main to be run in a separate thread (the run() method).
- **Main implements MouseListener**
  This requires Main to provide implementations for mouse event methods (mousePressed, mouseClicked, etc.).

Polymorphism

- **Method Overriding:**
  Main overrides methods from MouseListener (e.g., mousePressed, mouseClicked, etc.) and from Runnable (run()).

- **Interface Polymorphism:**
  An instance of Main can be referenced as a Runnable or MouseListener, allowing it to be passed to APIs expecting those interfaces.

Summary:

Inheritance: Main extends Canvas
Interfaces: Main implements Runnable, MouseListener
Polymorphism: Achieved via method overriding and interface implementation

What Our Program Does

We built a Java application that lets users draw a maze, set start and end points, and then watch different pathfinding algorithms (DFS, BFS, A*) solve the maze visually. Users can also save and load mazes.

How We Used OOP

- **Classes:**
  I split my code into three main classes:

  ### Main

  Controls the entire application: sets up the window, handles user input, manages the maze grid, and connects everything together.

  Key Features:

  - Extends Canvas for custom drawing.
  - Implements Runnable (for the render thread) and MouseListener (for mouse input).
  - Holds a 2D array of Node objects (nodeList) representing the maze.
  - Has static references to the current Main instance, the Algorithm object, and the start/end Node.
  - Handles menu actions (save/load, clear, run algorithms).
  - Handles mouse clicks to set walls, start, and end points.
  - Draws the maze and updates the display in a loop.

  ### Algorithm

  Contains the logic for solving the maze using different pathfinding algorithms (DFS, BFS, A*) and visualizes the search process.

  Key Features:

  - Methods for each algorithm: dfs, bfs, and Astar.
  - Uses color changes and delays to show the search progress and the final path.
  - Can adjust the speed of visualization (searchtime).
  - Works with the Node grid, updating node states as it searches.

Node

Represents a single cell in the maze grid.

Key Features:

- Stores its position, color (state), and references to neighbor nodes (left, right, up, down).
- Methods to render itself, change state (wall, start, end, path), and reset.
- Provides information to algorithms (e.g., if it's walkable, its neighbors).
- Encapsulates all logic for what a cell can be and how it behaves.

- **Inheritance & Interfaces:**
  My Main class extends Canvas so I can draw graphics, and implements Runnable (for threading) and MouseListener (for mouse input).

- **Encapsulation:**
  Each class keeps its own data private and exposes only what's needed through public methods. For example, Node has methods like isWall(), setColor(), and getNeighbours().

- **Composition:**
  Main holds a 2D array of Node objects to represent the maze. Each Node knows its neighbors (left, right, up, down), and Main also has an Algorithm object to run the searches.

- **Polymorphism:**
  I override methods from MouseListener and Runnable in Main, so my class can be used wherever those interfaces are needed.

How Our Program Works

- **Startup:**
  The program creates a window, sets up the menu, and initializes the maze grid.

- **Drawing & Input:**
  The grid is drawn using the render method. Mouse clicks let users toggle walls or set the start/end points.

- **Menu Actions:**
  The menu lets users save/load mazes, clear the board, or run a pathfinding algorithm.

- **Pathfinding:**

When an algorithm is chosen, the Algorithm class runs the search and updates node colors to show progress and the final path.

- **Saving/Loading:**
  Mazes can be saved to or loaded from a file, so users can reuse their designs.

Instruction Manual

1. Introduction

The Maze Solver Java Application is a GUI-based program developed using the Eclipse IDE. It allows users to design custom mazes and apply different search algorithms to find the shortest path from a user-defined start node to an end node. This application also provides runtime comparisons between different implementations of the same algorithm.

2. System Requirements

- Java Development Kit (JDK)
- Any Java IDE
- Operating System: Windows, macOS, or Linux

3. Installation and Setup

1. **Download the Project Files:**
    - Ensure the folder contains the following directories and files:
        - src – Source code files
        - bin – Compiled .class files
        - sample – Sample maze files
        - .classpath and .project – Eclipse project configuration files
2. **Import into Any IDE (Eclipse IDE preferably):**
    - Open Any IDE.

- Import the files (Either Download ZIP File or Git Clone)
- Open main.java
- Run Program

4. Application Usage

4.1 Launching the Application

- Run Main.java.
- The GUI window will launch, displaying an empty maze grid.

4.2 Interacting with the Maze

- **Left Click** – Set **Wall Node**
- **Right Click** – Set **End Node**
- **Middle Click** – Set **Start Node**

4.3 Running Algorithms

- After setting the start and end nodes:
  - Use the menu to choose one of the available algorithms:
    - **Breadth-First Search (BFS)**
    - **Depth-First Search (DFS)**
    - **A Star Search***
  - The algorithm will compute and display the shortest path along with its execution time.

4.4 Saving and Loading Mazes

- From the menu, users can:

- Save current maze configuration as a .maze file.
- Load pre-configured maze files from the sample/ directory (e.g., sample1.maze).

5. Algorithms Overview

Algorithm.java (Default)

Implements maze-solving using the following:

- **DFS** – Stack-based approach.
- **BFS** – Queue-based traversal with path reconstruction.
- **A* Search** – Uses an ArrayList for the open set and applies heuristics.

Algorithm2.java (Alternative Implementation)

For comparative study:

- **DFS** – Recursive implementation.
- **BFS** – Uses an ArrayDeque.
- **A*** – Uses a LinkedList for managing the open set.

To use Algorithm2:

1. Replace Algorithm.java with Algorithm2.java in the src/ folder.
2. Rename the class and file to Algorithm.java (the main program expects this name).

6. Source Code Structure

File	Description
Main.java	Initializes GUI, handles interactions, algorithm execution, and file I/O.
Node.java	Represents maze nodes with their type, color, and neighbors.
Algorithm.java	Primary implementation of DFS, BFS, and A* algorithms.
Algorithm2.java	Alternative implementation using different data structures.

7. Maze File Format

Maze files are text-based with the following representations:

Symbol	Description
1	Wall Node
0	Path Node
2	Start Node
3	End Node

Example:

1 1 1 1 1
1 2 0 3 1
1 1 1 1 1

8. Output Files

- **bin/**: Stores compiled .class files.
- **sample/**: Contains sample maze files for testing.
- **.classpath & .project**: Eclipse configuration files defining project structure.

9. Notes

- Ensure only one Algorithm.java file exists in the src/ folder when compiling.
- GUI operations are optimized for mouse input.
- Runtime is displayed after algorithm execution for performance analysis.

DS Part:

The Algorithms we chose to use in this program are listed below:

DFS (Depth-First Search): Uses a stack to explore nodes.
BFS (Breadth-First Search): Uses a queue to explore nodes and keeps track of the previous nodes to reconstruct the shortest path.
A* Star Search: Uses an open Array List implementation for a heuristic to prioritize nodes closer to the target.

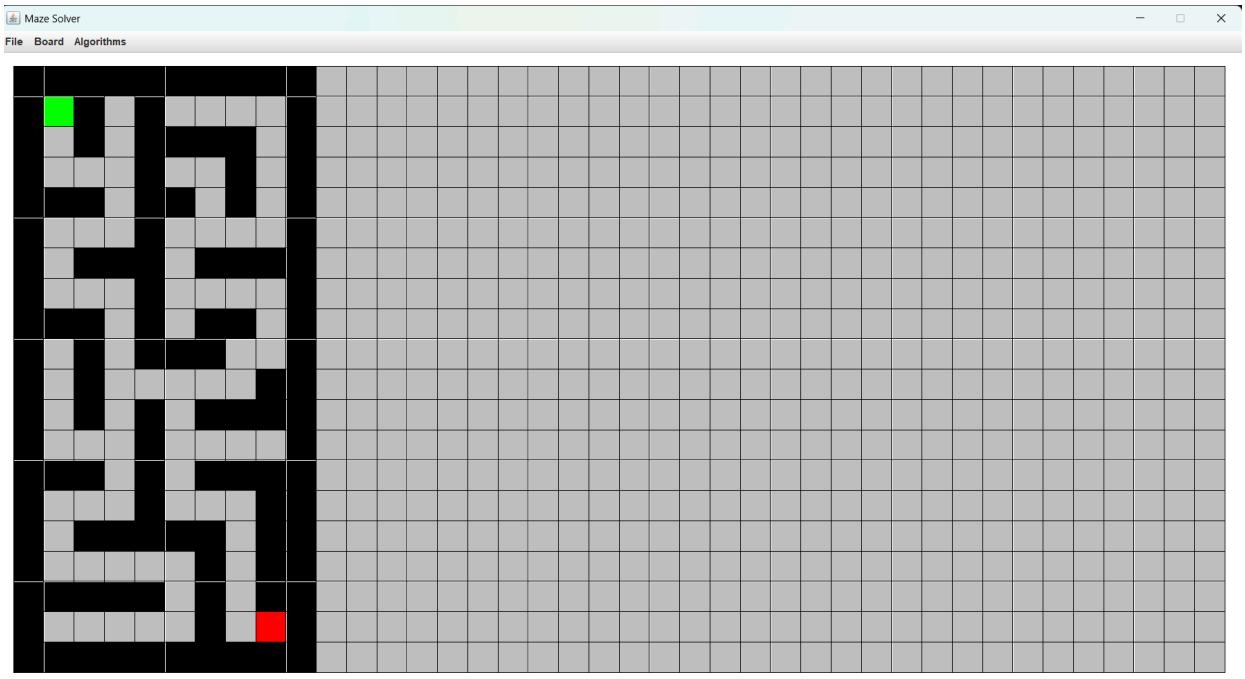
However, the second algorithm uses these:

DFS (Depth-First Search): Uses recursion to explore nodes.
BFS (Breadth-First Search): Uses an Array dequeue to explore nodes and keeps track of the previous nodes to reconstruct the shortest path.
A* Star Search: Uses an open linked List implementation for a heuristic to prioritize nodes closer to the target.

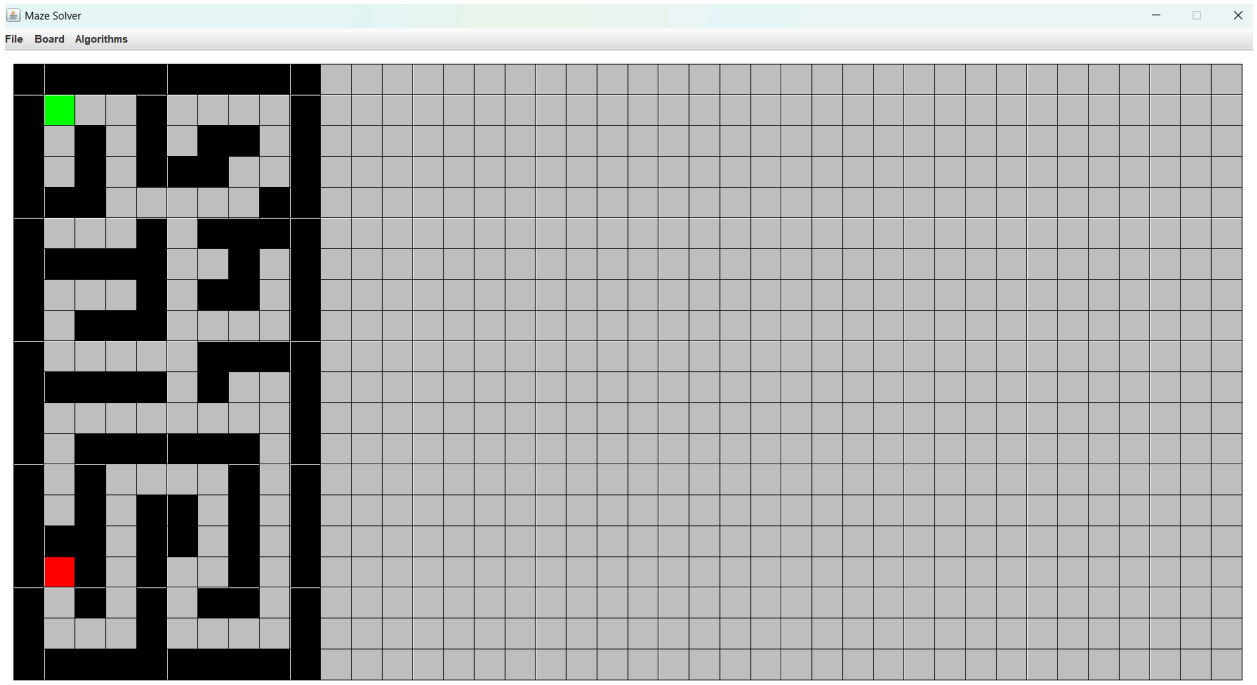
So, we will be running all 3 of these algorithms using 2 different data structures to find out which of them would run faster in 6 of the mazes we created.

Here are the mazes:

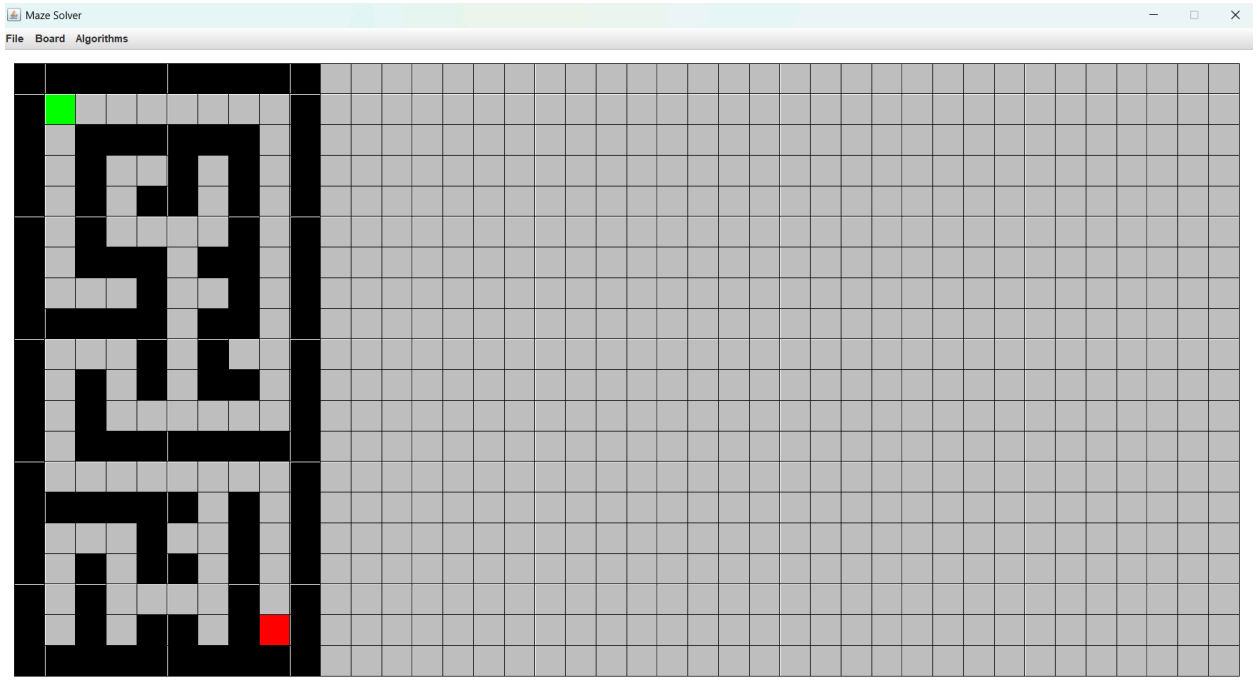
- Small maze 1.maze



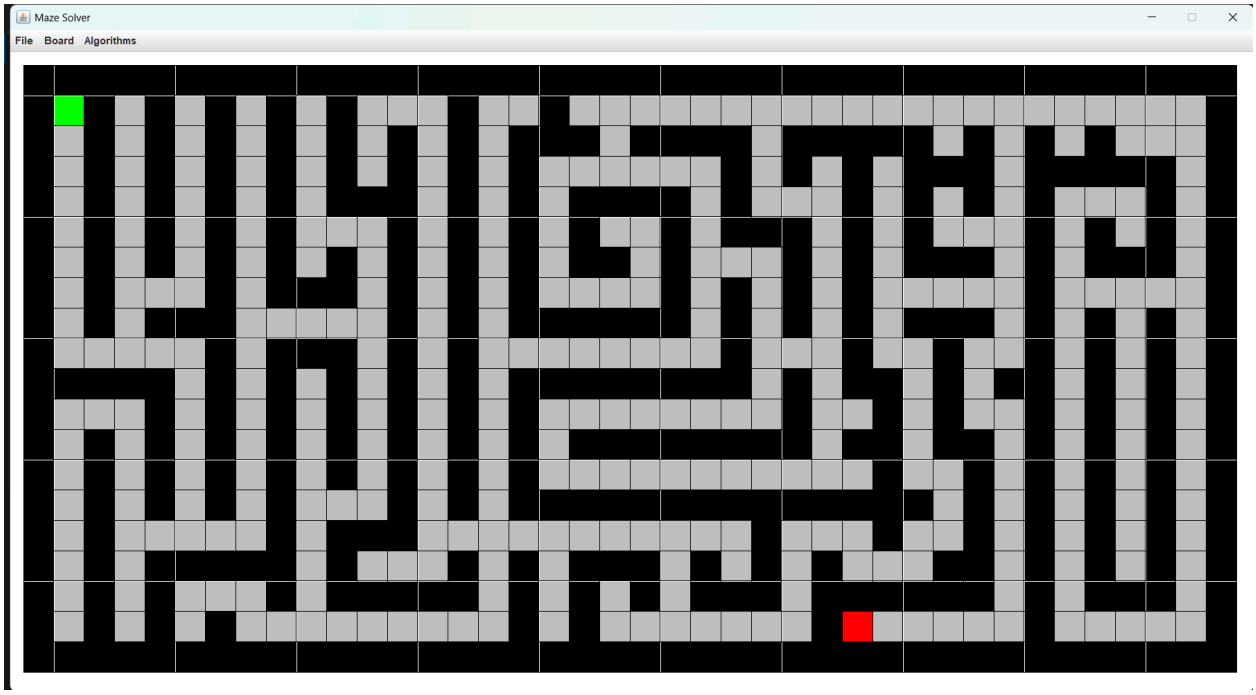
- Small maze 2.maze



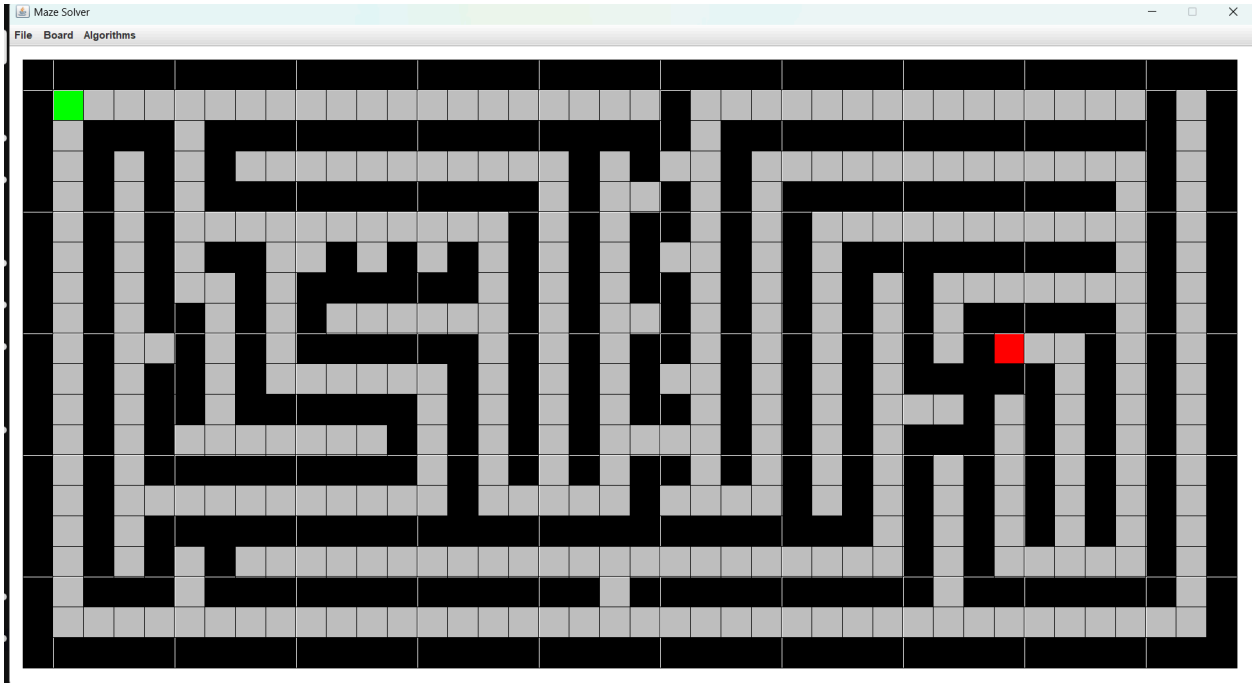
- Small maze 3.maze



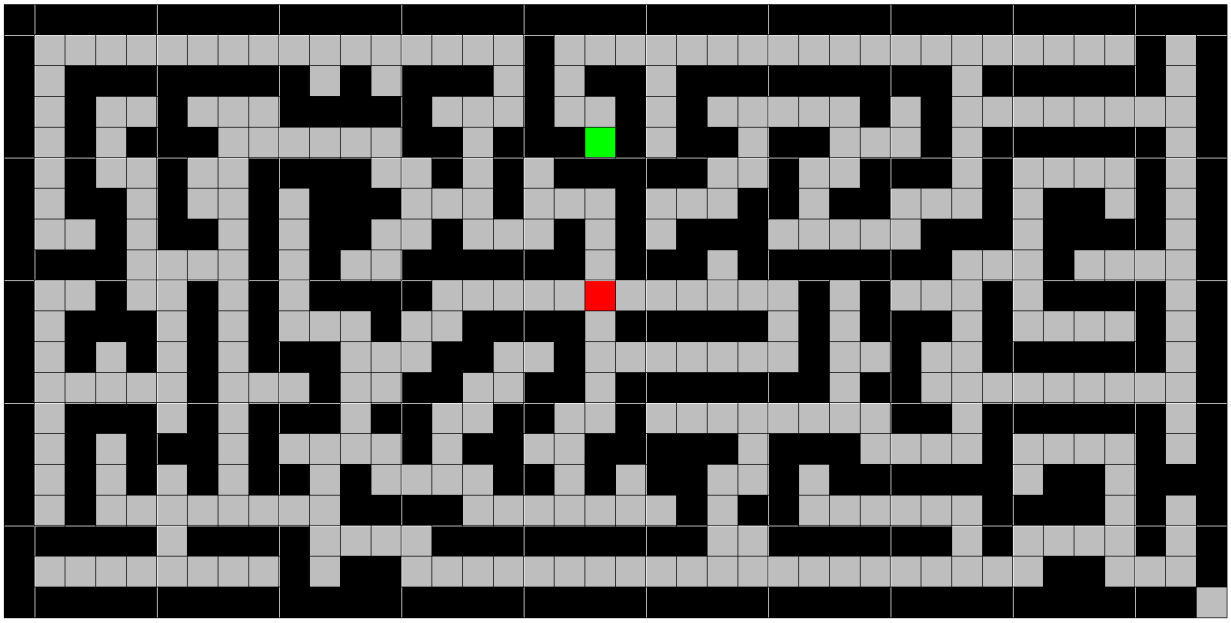
- Big maze 1.maze



- Big maze 2.maze



- Big maze 3.maze



Results

DFS —

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
DFS	Stack	Small Maze 1	6245 ms	6239 ms	6259 ms
DFS	Stack	Small Maze 2	6773 ms	6802 ms	6810 ms
DFS	Stack	Small Maze 3	6130 ms	6127 ms	6098 ms
DFS	Stack	Big Maze 1	23529 ms	23395 ms	23517 ms
DFS	Stack	Big Maze 2	17474 ms	17448 ms	17466 ms
DFS	Stack	Big Maze 3	17997 ms	17967 ms	17985 ms

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
DFS	Recursion	Small Maze 1	6206 ms	6281 ms	6264 ms
DFS	Recursion	Small Maze 2	6816 ms	6791 ms	6742 ms
DFS	Recursion	Small Maze 3	6134 ms	6120 ms	6064 ms
DFS	Recursion	Big Maze 1	23522 ms	23512 ms	23473 ms
DFS	Recursion	Big Maze 2	17414 ms	17442 ms	17599 ms
DFS	Recursion	Big Maze 3	18023 ms	18015 ms	18177 ms

BFS —

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
BFS	Queue	Small Maze 1	6690 ms	6664 ms	6665 ms
BFS	Queue	Small Maze 2	8948 ms	8873 ms	8901 ms
BFS	Queue	Small Maze 3	8059 ms	8104 ms	8075 ms
BFS	Queue	Big Maze 1	36872 ms	37061 ms	36960 ms
BFS	Queue	Big Maze 2	39525 ms	39706 ms	39736 ms
BFS	Queue	Big Maze 3	30027 ms	30264 ms	30415 ms

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
BFS	Array Dequeue	Small Maze 1	6652 ms	6688 ms	6653 ms
BFS	Array Dequeue	Small Maze 2	8882 ms	8975 ms	8930 ms
BFS	Array Dequeue	Small Maze 3	8031 ms	8105 ms	8087 ms
BFS	Array Dequeue	Big Maze 1	36809 ms	36902 ms	36788 ms
BFS	Array Dequeue	Big Maze 2	39614 ms	39821 ms	39716 ms
BFS	Array Dequeue	Big Maze 3	30047 ms	29895 ms	29898 ms

A Star —

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
A Star	Array List	Small Maze 1	3765 ms	3800 ms	3786 ms
A Star	Array List	Small Maze 2	8905 ms	8905 ms	8906 ms
A Star	Array List	Small Maze 3	7549 ms	7541 ms	7554 ms
A Star	Array List	Big Maze 1	32226 ms	32401 ms	32301 ms
A Star	Array List	Big Maze 2	28444 ms	28298 ms	28715 ms
A Star	Array List	Big Maze 3	16886 ms	16926 ms	16923 ms

Operation type	Data Structure	Maze	Runtime 1	Runtime 2	Runtime 3
A Star	Linked List	Small Maze 1	3771 ms	3751 ms	3770 ms
A Star	Linked List	Small Maze 2	8930 ms	8901 ms	8957 ms
A Star	Linked List	Small Maze 3	7594 ms	7601 ms	7587 ms
A Star	Linked List	Big Maze 1	32268 ms	32285 ms	32235 ms
A Star	Linked List	Big Maze 2	28435 ms	28248 ms	28305 ms
A Star	Linked List	Big Maze 3	17025 ms	16916 ms	16947 ms

Analysis:

The test is run on the Asus TUF F15 (Not charged)

- Intel(R) Core(TM) i7-10870H CPU
- NVIDIA GeForce GTX 1660 Ti
- RAM 8 GB
- Graphics Card 6 GB

As you can see from the results above, we can calculate the average of the data.

DFS Stack :

- DFS - Stack (Small maze 1): 6247.67 ms
- DFS - Stack (Small maze 2): 6795 ms
- DFS - Stack (Small maze 3): 6118.33 ms
- DFS - Stack (Big maze 1): 23480.33 ms
- DFS - Stack (Big maze 2): 17462.67 ms
- DFS - Stack (Big maze 3): 17983 ms

For **DFS** using a **Stack**, the fastest on the small maze is on the third maze (6118.33 ms) compared to the other small mazes. However, for the big maze, the fastest is on the second maze (17462.67 ms).

DFS Recursion :

- DFS - Recursion (Small maze 1): 6250.33 ms
- DFS - Recursion (Small maze 2): 6783 ms
- DFS - Recursion (Small maze 3): 6106 ms
- DFS - Recursion (Big maze 1): 23502.33 ms
- DFS - Recursion (Big maze 2): 17485 ms
- DFS - Recursion (Big maze 3): 18071.67 ms

For **DFS** using **Recursion**, the fastest on the small maze is on the third maze (6106 ms) compared to the other small mazes. On the other hand, for the big maze, the fastest is on the second maze too (17485 ms).

BFS Queue :

- BFS - Queue (Small maze 1): 6673 ms
- BFS - Queue (Small maze 2): 8907.33 ms
- BFS - Queue (Small maze 3): 8079.33 ms
- BFS - Queue (Big maze 1): 36964.33 ms
- BFS - Queue (Big maze 2): 39655.67 ms
- BFS - Queue (Big maze 3): 30235.33 ms

For **BFS** using a **Queue**, the fastest runtime on the small maze is on the first maze (6673 ms) compared to the other small mazes. However, for the big maze, the fastest is on the third maze (30235.33 ms).

BFS Array Dequeue :

- BFS - Array Dequeue (Small maze 1): 6664.33 ms
- BFS - Array Dequeue (Small maze 2): 8929 ms
- BFS - Array Dequeue (Small maze 3): 8074.33 ms
- BFS - Array Dequeue (Big maze 1): 36833 ms
- BFS - Array Dequeue (Big maze 2): 37717 ms
- BFS - Array Dequeue (Big maze 3): 29946.67 ms

For **BFS** using **Array Dequeue**, the fastest runtime on the small maze is on the first maze too (6664.33 ms) compared to the other small mazes. On the other hand, the fastest on the big maze is on the third maze too (29946.67 ms).

A Star Array List :

- A Star - Array List (Small maze 1): 3783.67 ms
- A Star - Array List (Small maze 2): 8905.33 ms
- A Star - Array List (Small maze 3): 7548 ms
- A Star - Array List (Big maze 1): 32309.33 ms
- A Star - Array List (Big maze 2): 28485.67 ms
- A Star - Array List (Big maze 3): 16911.67 ms

For **A Star** using an **ArrayList**, the fastest runtime on the small mazes is on the first maze (3783.67 ms) compared to the other small mazes. However, the fastest runtime on the big maze is the third maze (16911.67 ms).

A Star Linked List :

- A Star - Linked List (Small maze 1): 3764 ms
- A Star - Linked List (Small maze 2): 8929.33 ms
- A Star - Linked List (Small maze 3): 7594 ms
- A Star - Linked List (Big maze 1): 32262.67 ms
- A Star - Linked List (Big maze 2): 28329.33 ms
- A Star - Linked List (Big maze 3): 16962.67 ms

For **A Star** using a **Linked List**, the fastest runtime on the small mazes is on the first maze too (3764 ms), compared to the other small mazes. On the other hand, the fastest runtime on the big maze is on the third maze too (16962.67 ms).

The data that we could get from the fastest of each operation type with 2 data structures are :

- DFS - Stack (Small maze): 6118.33 ms
  DFS - Stack (Big maze): 17462.67 ms
- DFS - Recursion (Small maze): 6106 ms
  DFS - Recursion (Big maze): 17485 ms

DFS using a Stack is slightly better in bigger mazes than DFS using Recursion, so also on the other hand, DFS using Recursion is slightly better in smaller mazes than DFS using a Stack.

- BFS - Queue (Small maze): 6673 ms
  BFS - Queue (Big maze): 30235,33 ms
- BFS - Array Dequeue (Small maze): 6664.33 ms
  BFS - Array Dequeue (Big maze): 29946.67 ms

BFS using an Array Dequeue is slightly better in both small and big mazes than BFS using a Queue.

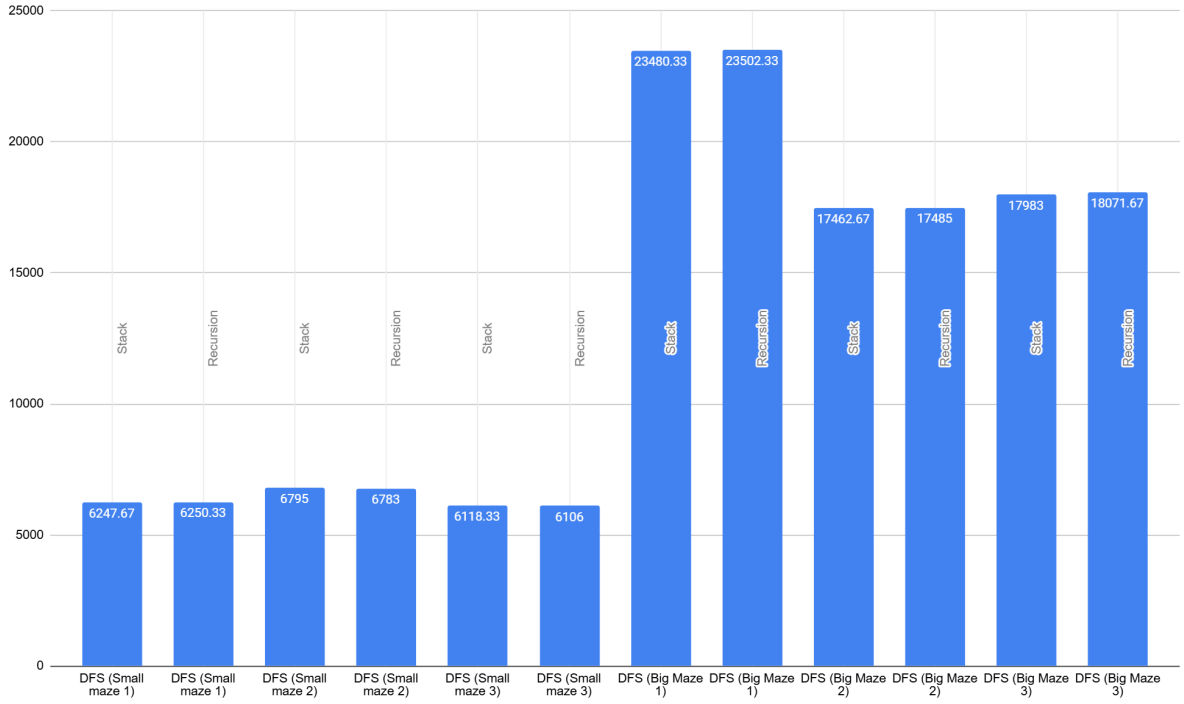
- A Star - Array List (Small maze): 3783.67 ms
  A Star - Array List (Big maze): 16911.67 ms
- A Star - Linked List (Small maze): 3764 ms
  A Star - Linked List (Big maze): 16962.67 ms

A Star using an Array List is also slightly better in big mazes than A Star using a Linked List, so also on the other hand, A Star using a Linked List is slightly better in small mazes than A Star using an Array List.

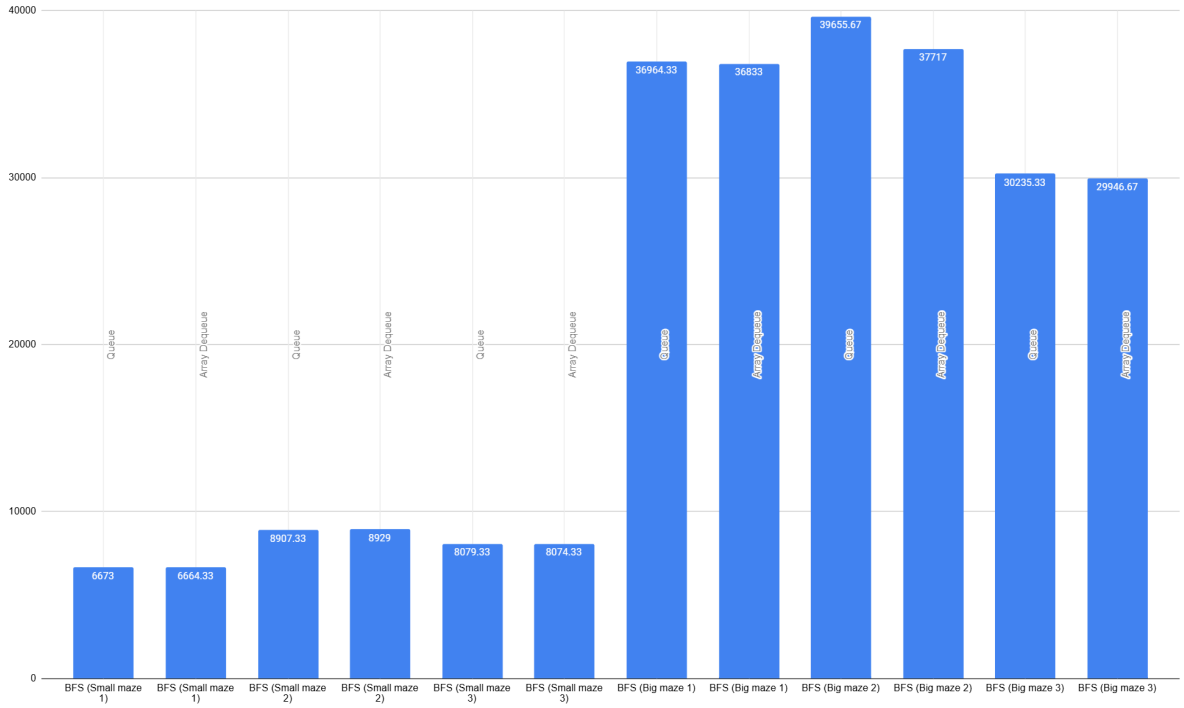
As we can see from all of the comparison results, we can conclude that the best algorithm for solving a maze is A Star, whether it uses an ArrayList or a LinkedList; both of them are better in different sizes of mazes. At the second comes DFS, same as A star, it also includes all of the data structures, because they both are better in different sizes of mazes. Lastly, BFS took longer runtime to solve the maze, and the best data structure for solving a maze is using an Array Dequeue.

Growth Graph:

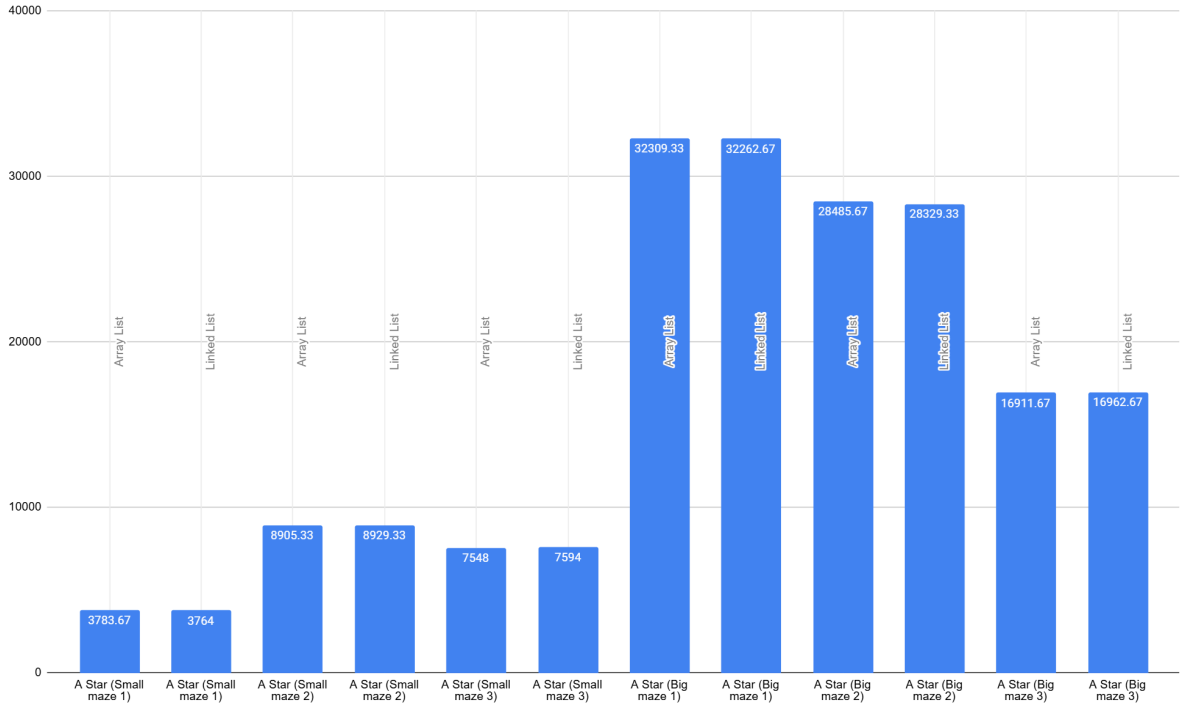
DFS - Stack and Recursion Growth Graph :



BFS - Queue and Array Dequeue Growth Graph :



A Star - Array List and Linked List Growth Graph :



Best for our case:

The best Data Structure for our 3 chosen algorithms is as follows:

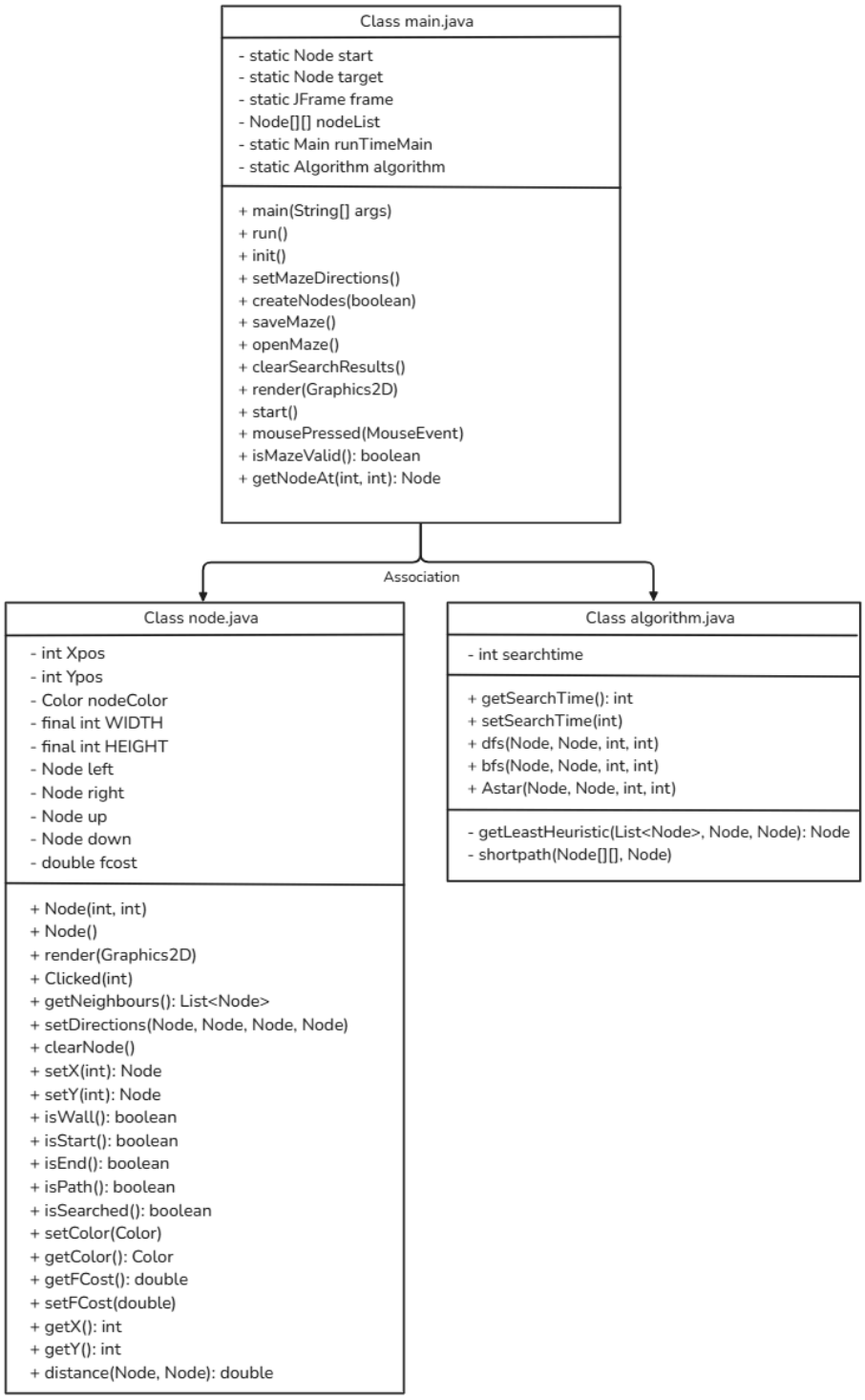
DFS (Depth-First Search) best data structure to implement is both of them, on different scenarios. While stacks perform better than recursion in big mazes, recursion is slightly better in smaller ones. Therefore, if we made separate algorithms for both scenarios, both stack and recursion could be used.

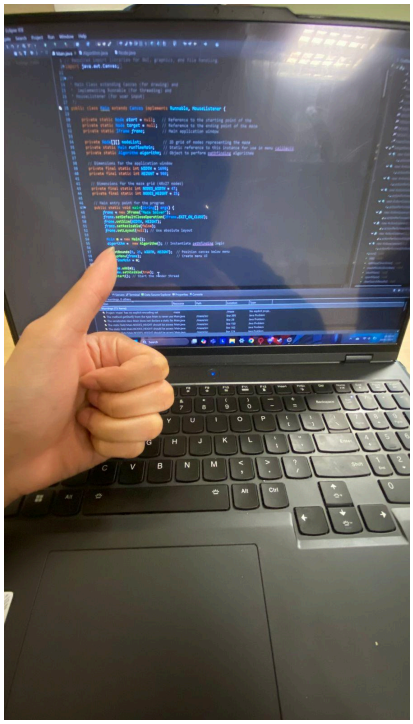
BFS (Breadth-First Search) best data structure to implement is an Array Dequeue because it's better in both small and big mazes.

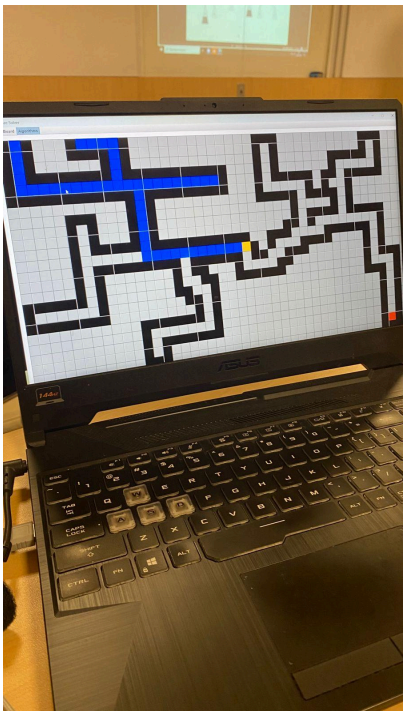
A* (A Star) best data structure to implement is both of them, on different scenarios. While ArrayList performs better than LinkedList in big mazes, LinkedList is slightly better in smaller mazes. Therefore, both of them should be implemented on separate occasions.

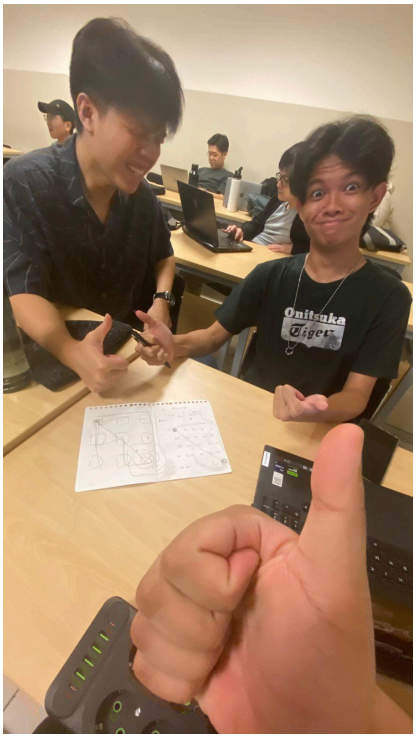
Documentation

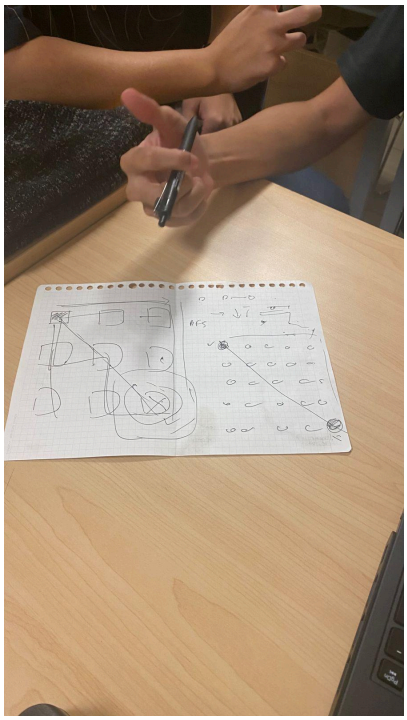
Class Diagram:

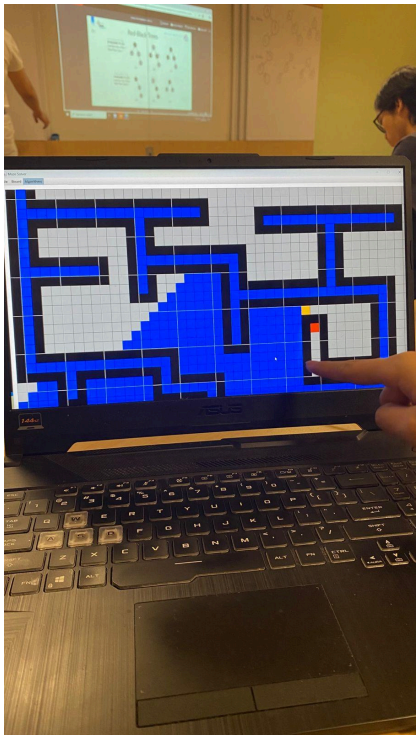


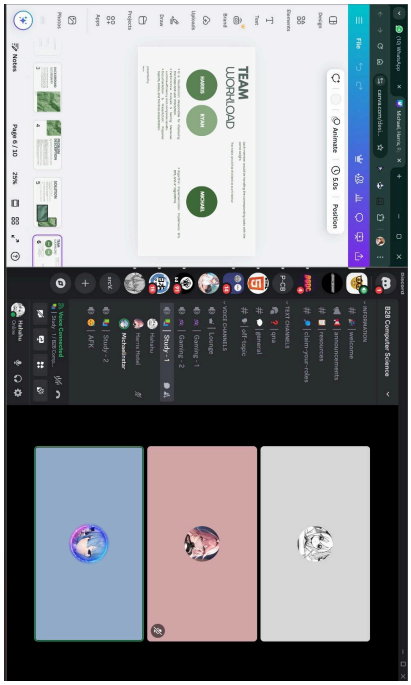





Onitsuka
Tiger

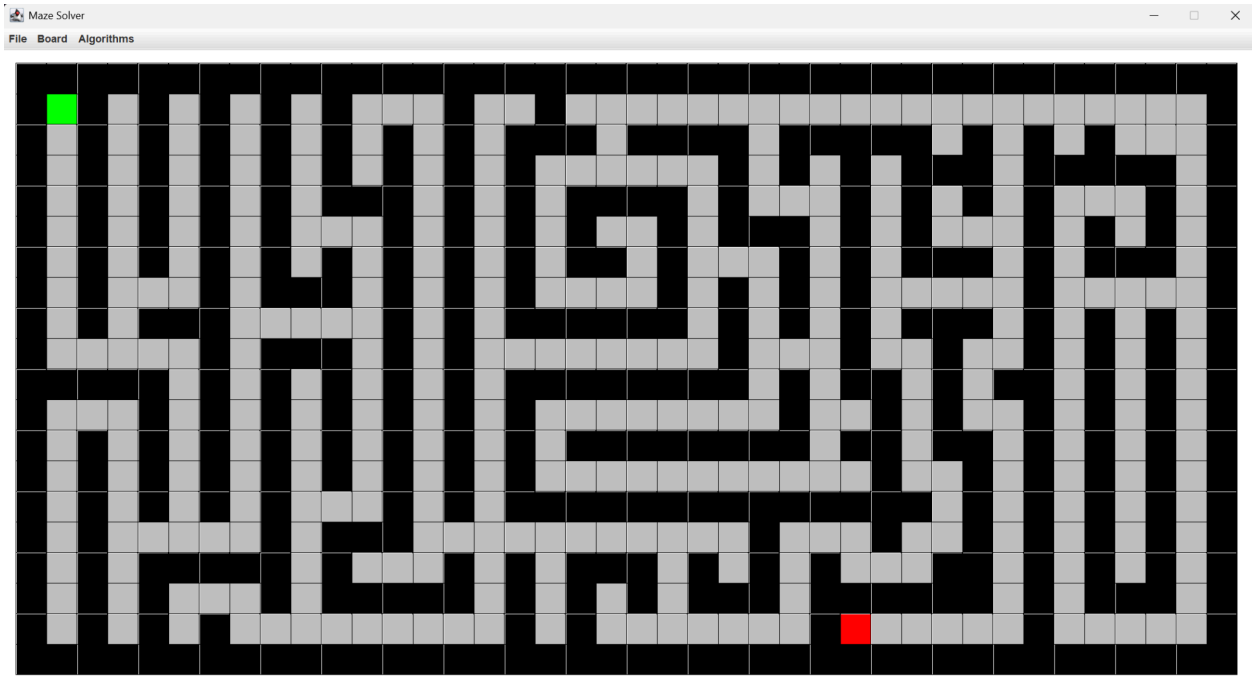





TEAM
WORKLOAD

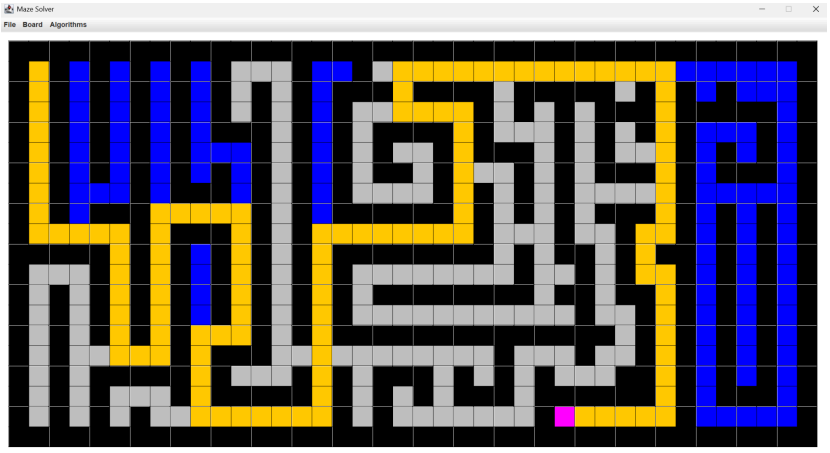
Evidence of Working Program:

Maze 1 (Unsolved)

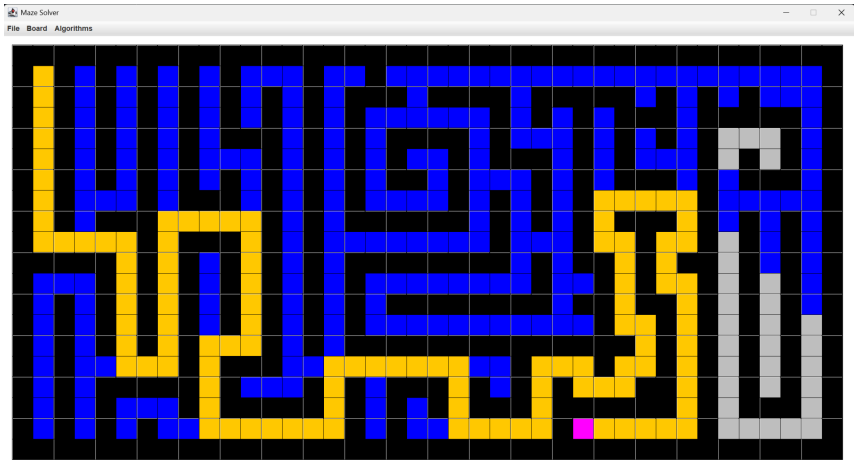


Maze 1 (Solved)

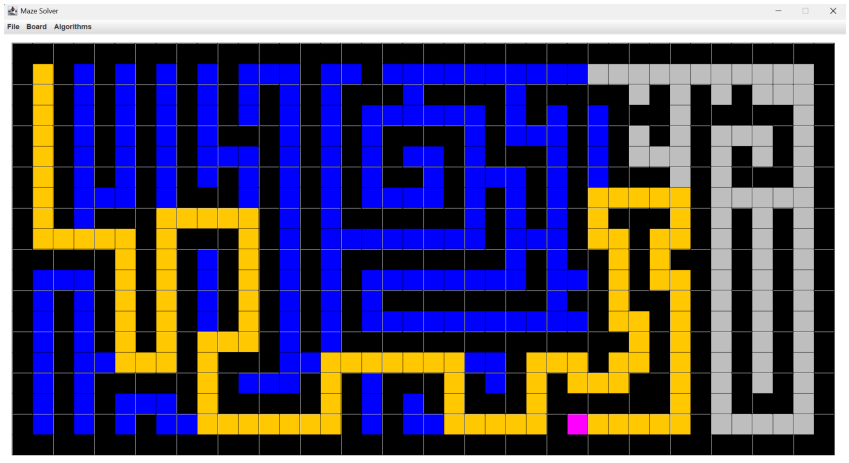
DFS:



BFS:


Maze Solver
File Board Algorithms

A Star:


Maze Solver
File Board Algorithms

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

Lafore, R. (2017). Data structures and algorithms in Java (2nd ed.). Sams Publishing.

Patel, A. (2014). A pathfinding for beginners. Stanford CS Theory. https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison

GeeksforGeeks. (2021). Breadth-first search (BFS) for a graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

GeeksforGeeks. (2021). Depth-first search (DFS) for a graph. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Buck, J. (2011). Maze generation: Recursive backtracking. https://www.jamisbuck.org/mazes/

Oracle. (2023). JavaFX graphics tutorial. Oracle Docs. https://docs.oracle.com/javase/8/javafx/api/toc.htm

Baeldung. (2022). Graphs in Java. https://www.baeldung.com/java-graphs

Stack Overflow. (2010). When is it practical to use DFS vs BFS? https://stackoverflow.com/questions/3332947/when-is-it-practical-to-use-dfs-vs-bfs

Github References:

https://github.com/mtajammulzia/AStarSearch-MazeGenerator
https://github.com/NDCHIRO/Algorithms-Visualizer.git

Appendix

Git Web: https://github.com/MichaelFirstAC/Maze-Maker-Solver.git

Presentation file:
https://www.canva.com/design/DAGhSSPDMNs/TLa5GtqPYZOOsObG9k0MPg/edit?utm_content=DAGhSSPDMNs&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Poster:
https://www.canva.com/design/DAGn3j-vWsA/wKHQhsZIXmDXxUbeZqZ4aQ/edit?utm_content=DAGn3j-vWsA&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Growth Graph:

https://docs.google.com/spreadsheets/d/1zX9xIS4gcgxbXJYEaws-EI-AlaR7J6QnRZQO43l4q10/edit?usp=sharing