Ojaswi Tyagi - 100809169 - Second year Data Science

1. General Description

The project Lost In Maze is a game in which the user controls a rat which is stuck in the centre of a maze. The player uses their arrow keys (up, down, left, right) to guide the rat out of the maze. The user can create a new maze or play an older maze from storage (created earlier). After the user selects the maze, the rat will be deployed after which the user will be able to move the rat around. The number of moves taken to solve the maze is computed, and is used to determine the score of the player. The rat can now move freely in the empty spaces between the walls, but not through the walls. However, there will be "bridges" in the maze which will allow the rat to go over walls and move to a different part of the maze. There is only one exit out of the maze and the user will try to find the way out by moving the rat around the maze. There is an opponent in the Maze which is trying to eliminate the player Rat before it can solve the maze. The win condition is to lead the rat outside the maze by finding the exit before it gets eliminated. If the user wins, they are congratulated. If the user beats the maze with fewer moves than before, the maze object's high score will be updated with the user's name and the new high score. If the user decides to give up, the programme will highlight the path from the user's current position to the exit. The user can keep playing the game with the highlighted path, but they won't be able to win anymore.

There are a few changes from the initial plan:

1. **Added opponent** to make the game more challenging.
2. Timer is replaced with the **number of moves**. This is done because adding the Opponent can sometimes mean that the user has to wait for the opponent to leave in order to continue. Number of moves is a better metric to judge performance.

2. User interface

Running the program

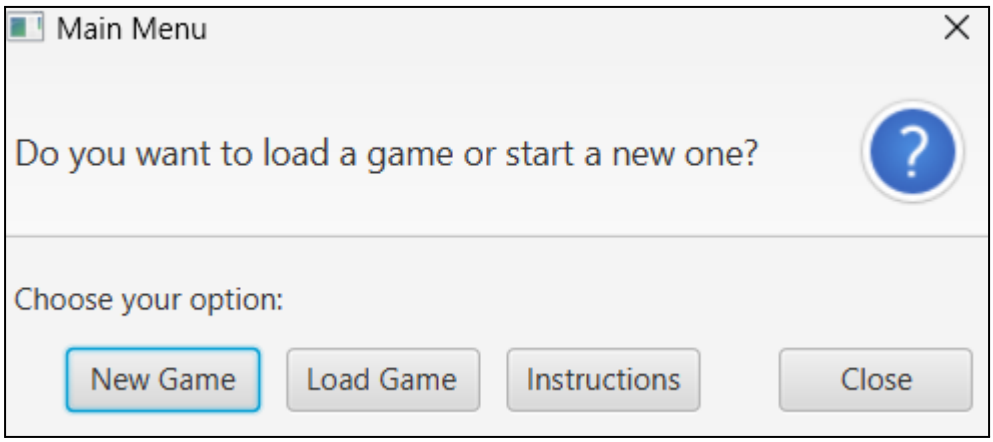
In IntelliJ sbt shell:

1. **run** to build and run the program.
2. **test** to run the program.

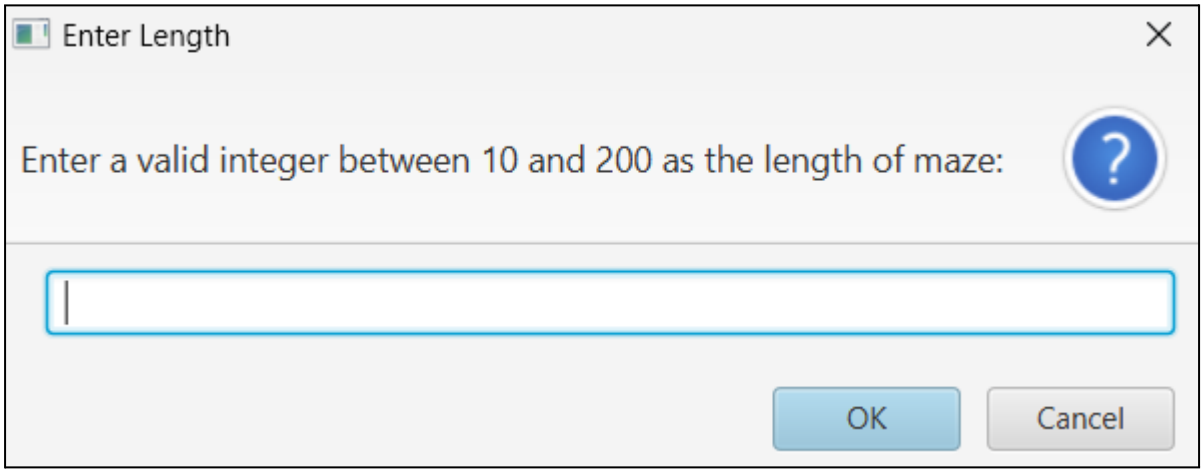
You can also use the lost_in_maze.jar file to run the game.

Menus in the Game

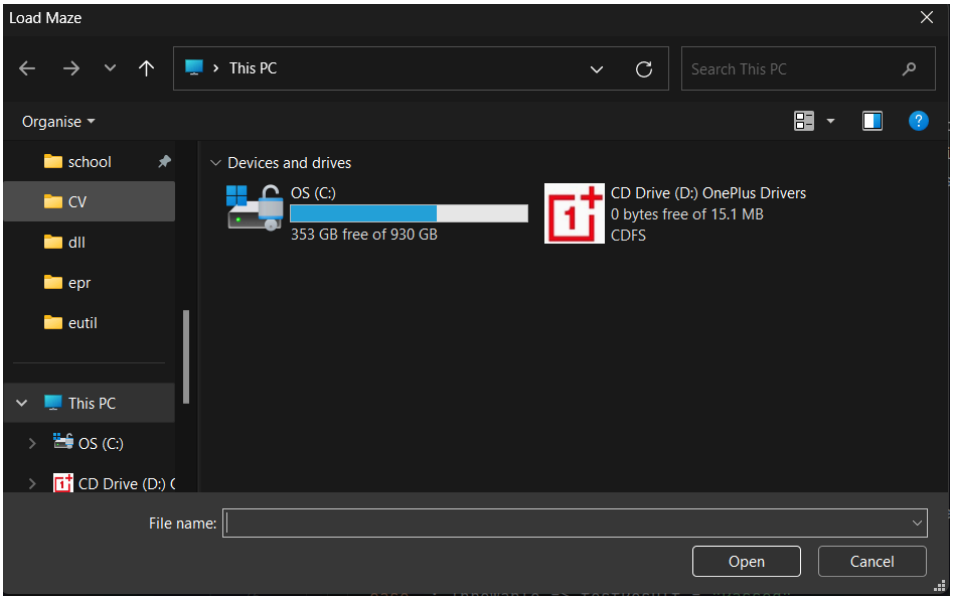
1. Main menu View: Upon running the game, the player will see the main menu. Here the player can create a new game, or load an existing game. Users can also view the instructions or close the programme.



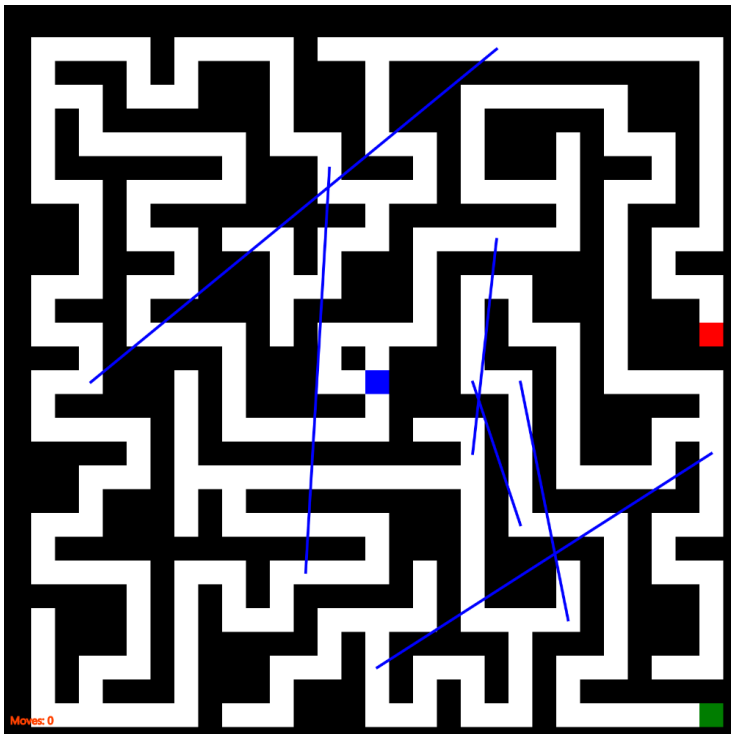
2. New Game view: If the user selects the new game option in the main menu, the user is asked to enter length and breadth. The dimensions must be valid integers between 10 and 200 (inclusive). If the user enters faulty dimensions, an error alert is shown and the game is gracefully exited.



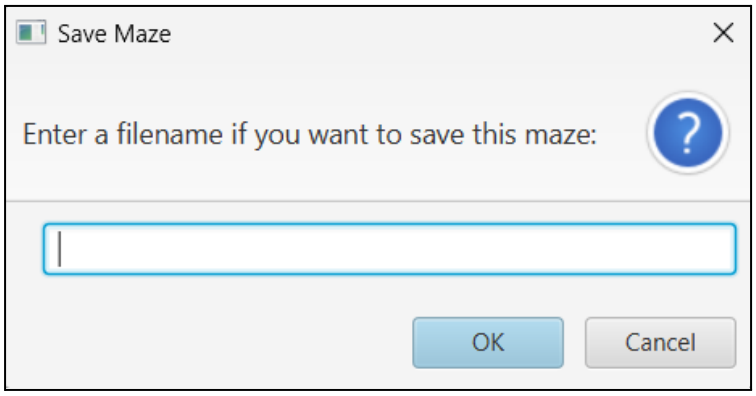
3. Load Game View: If the user selects the load game option in the main menu, a file explorer window is opened which allows the user to select a maze file. If a corrupter/wrong file is chosen, an error alert is thrown, and the same window appears until a valid file is chosen.



4. The Game view: After the game is successfully loaded or a new maze is created, the game window appears. The blue square represents the player Rat, and the red square is the opponent. The blue lines are bridges, if the user gets to one end of the bridge, the rat can transverse to the other side of the bridge. The green square is the exit to the maze. The game ends if the user touches the opponent or reaches the exit.



5. Saving file: After the game is over, the user can save that maze.



3. Programme structure

class **Game**

- Manages the overall game flow, including maze construction, rat deployment and interactions with other game components.
- Relationships: Contains instances of Rat and Storage.
- Methods:
  i. newMaze: Generates a new maze based on user-defined dimensions.
  ii. startGame: Initialises the game, resets and deploys the rat
  iii. endGame: Checks whether game has ended

class **Maze**

- Represents an individual maze.
- Responsible for solving mazes and tracking user high scores.
- Contains information about walls, passages, bridges as arrays and start and exit coordinates as Cells.
- Relationships: Contains multiple subtypes of Cells and bridges.
- Methods:
  i. solveMaze: Implements maze solving algorithms to find the exit path using breadth first search.
  ii. possiblePassages: Finds possible passages from current passage including bridges
  iii. updateHighscore(player: str, timeTaken: int): Updates the high score record with the player's name and time taken to solve the maze.

class **Rat:**

- Represents a rat.
- Stores the current position.
- Relationships: Belongs to a Game instance.
- Methods:
    i. move(direction): Moves the rat in the maze according to the user's input direction (up, down, etc)
    ii. moveToOtherEnd: Moves the rat to the other end of the bridge.

class **Cell:**

- Represents a position within the maze (x, y).
- Relationships: Inherited by wall and Passage. Class Cell class is used to extend Wall and Passage to ensure similar code does not have to be repeated.
- Methods:
    i. neighbors: calculates the neighboring passages of a cell.

class **Wall** extends **Cell:**

- Represents a position within the maze (x, y) for a wall.
- Relationships : Subclass of Cell, contained as an Array in Maze.

class **Passage** extends **Cell:**

- Represents a position within the maze (x, y) for a passage.
- Relationships : Subclass of Cell, contained as an Array in Maze.

class **Bridge:**

- Represents the bridges in the maze.
- Connects 2 specific **Passages**, allowing the rat to move over walls.
- Relationships: Contains two Passages.

class **Storage:**

- Handles reading and writing maze data.
- Saves high scores, maze layouts, and other relevant information.
- Relationships: Used by Game
- Methods:
    i. readMazeData(): Reads maze data from storage.
    ii. writeMazeData(data): Writes maze data to storage.

object **Mazegenerator:**

- Helps with maze generation.
- Methods:
    i. generateMaze: Uses recursive backtracking to create a maze.

object **MazeDraw:**

- Helps with maze drawing on canvas.
- Methods:
    i. drawMaze: draws the maze, rat, bridges, etc.

object **DisplayMessages:**

- Helps with alert messages and dialogue boxes.

UML CAN BE FOUND IN APPENDIX AND IN GITLAB

4. Use case description :

- **Game Initialization:** Upon launching the game, a Game object is instantiated with a rat object, a storage object, and an empty maze object.
- **Main Menu Interaction**: The user selects "create new maze" from the main menu and specifies the maze dimensions. The created maze will contain information about the Walls, Passages, and the Bridges.
- **Maze Generation and Loading:** The Game object generates a new maze file in storage based on the specified dimensions. The maze file is loaded as a maze object, and it is displayed on the screen along with the Rat.
- **Gameplay:** The user navigates the Rat through the maze, encountering walls, passages, bridges, and the opponent that determine their movement. The game ends when the exit is reached or the rat is eliminated by the opponent.
- **Game End:** The Game object receives the signal to end the game, stopping the gameplay.
- **High Score and maze saving**: With the assistance of the storage class, a new maze file can be saved or the new high score can be written onto an existing maze file for future reference.

5. Algorithms:

Maze creation:

Since I wanted to create a “perfect maze” , I had to ensure that every maze created by my programme would have exactly one solution. I narrowed my choices between Kruskal’s algorithm and Recursive Backtracking algorithm. I think I will go with the **recursive backtracking algorithm (1)** due to its

guarantee of exactly one solution, along with hard to solve, unpredictable mazes. There are also no isolated areas, thus every passage can be accessed by the rat.

Recursive backtracking algorithm: The maze starts as a large grid of cells, each cell starting with four walls around it. Starting from the middle cell we mark it as visited and choose a neighbouring cell that hasn't been visited yet. Then we remove the wall between the current cell and the chosen neighbour. We mark the new cell as visited and add it to a stack (check data structures section). Then we continue this process, until we get to a cell that has no unvisited neighbours, thus reaching a dead-end. When at a dead-end, we go back through the path until we reach a cell with an unvisited neighbour. We keep doing this process until eventually, we backtrack all the way to the beginning cell.

Maze Solving:

Maze Solving: I chose to utilise Breadth First Search (BFS) for maze solving because it effectively finds the shortest path while also accommodating bridges within the algorithm seamlessly. Here's how BFS was used in my game. It begins by exploring all possible paths (nodes) at the current depth level before moving on to nodes at the next depth level. This systematic approach ensures that it always considers the closest nodes first, gradually moving outwards from the starting point. By exploring in this breadth-first manner, BFS guarantees that the first path it finds to the exit is the shortest possible path.

BFS was ideal for my maze, where there exists only one solution. Its exploration and ability to find the shortest path without any unnecessary backtracking made it a perfect fit for efficiently navigating through the maze.

I could have chosen something complex yet more efficient like A* algorithm, however, for this project, I decided to go with a simple and reliable algorithm which has been used countless times to solve mazes.

Opponent:

The opponent rat in the maze is programmed to move around in an unpredictable way. It remembers the last cell it visited so that it doesn't go back to the same path it just came from. It then finds all the possible passages that are visitable from the current location. It then picks one of these paths randomly, making it unpredictable. But if it only has one path left, it means that the opponent has reached a deadend. In this case, it can go back to the same way it came from. A timer is set up using the Future concurrent object from scala.concurrent to run the opponentRatMoveTask repeatedly at intervals (in this case one move every 350 milliseconds). If the location of the opponent is the same as the rat, the game is ended and an alert is shown.

6. Data Structures

I will use Vectors to store the maze properties. There will be three vectors for storing Walls, Passages, and Bridges, respectively.

For the Maze generation, I will be using a mutable dynamic stack **(4)** from the scala collection library. This is because stacks follow LIFO (last in, first out) principle **(3)**, which is particularly important for Recursive backtracking. Since the size of the stack can change, it must be dynamic length. You can read more about how the stack will function in the Recursive backtracking section. To keep track of the visited cells, I will use a dynamic structure such as a list or a set from the scala library. . As cells are visited or explored, they can be added to this collection to prevent revisiting and ensuring efficiency in maze solving.

For the maze solving algorithm, I will be using a mutable dynamic queue from the Scala collection library. This choice aligns perfectly with BFS's nature, which explores nodes in a breadth-first manner, making a mutable queue ideal due to its ability to handle the FIFO (first in, first out) principle crucial for BFS. The dynamic length of the queue allows for efficient exploration of adjacent passages.

To keep track of visited passages and ensure efficiency in maze solving, I will use a dynamic structure such as a mutable set from the Scala library. As passages are explored and marked as visited, they will be added to this collection to prevent revisiting. This approach ensures that the algorithm progresses methodically, considering all possible paths before reaching the solution.

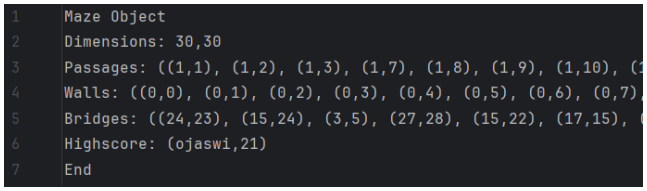
A data structure that supports key-value pairs, such as a map or dictionary, is appropriate for storing high scores. The player's name can be associated with their corresponding high score, allowing for easy retrieval and updating.

There are multiple other collection type data structures used in the project, however there is no concrete reason to choose these data structures over other structures, and thus I have not justified the use of such data structures.

I will be using no custom data structures in this project.

7. Files

I will be using a custom text file for saving the Labyrinth's structure and relevant information in a readable and parseable manner. The file will start with the header "Maze Object". This can help identify and validate the file format and version during parsing. A section in the file will represent the maze structure. This section will include information about walls, passages, bridges, starting coordinates, and exit coordinates. Furthermore, a section to store the high score achieved for the maze and the name of the player who achieved it will also be added. The file ends with the keyword "End" to signal stop reading.



As it can be seen from the image, for each property, the beginning of the property starts with the name

of the property followed by ":". Thus, when reading from a file, each property is read by using a regex pattern corresponding to that property. For bridges, pairs of passages represent one bridge. This approach ensures that each property is read correctly from the file, and the content within each property is processed based on the file format specifications.

The file will only be read between keywords "Maze Object" and "End", meaning that the programme will not read the lines that are before or after those keywords. Any comments or notes can be written after the "End" keyword.

When maze is to be updated, the programme will overwrite the high score and the player name properties according to user input.

Sample files can be found on GITLAB.

8. Testing plan

System testing:

The following manual tests were run and passed:

1. Check maze generation, maze solving, move count functionality, high score tracking, and user input handling.
2. Verify that the game flows correctly from start to finish, including maze generation, rat movement, and victory conditions.
3. Verify that user inputs are properly captured and processed (rat shouldn't move through walls, etc) with a variety of maze dimensions (rectangular, odd-even etc).
4. Try error handling by creating an impossible maze and check how the programme responds to that (negative numbers, too small, too large). Ensuring graceful handling of these errors and appropriate feedback is important.
5. Manually verify that the file format is properly saved and read. This will unfortunately only be possible for smaller mazes.
6. Manually corrupt a maze file and try loading that. Files of wrong format should not be loaded.
7. Test case with a maze with no solution or multiple solutions.
8. Asking friends and family to try the programme on their device to ensure it works on all devices.
9. After maze solving was implemented, I found pre-solved mazes from the internet and solved them using the programme. The expected output matched the output provided from the maze solution.

Unit Testing:

1. Core functionalities are Maze generation, Maze solving, and file loading.

2. Print Maze: This test prints a maze based on the given length, width, end coordinate, and start coordinate. Then the maze is printed and the game can be run for the same game. The printed maze can be used to verify that the correct maze properties are loaded. The game can also be compared to the printed maze. This test was used for multiple inputs and passed every time.
3. I focused most on maze saving and loading as these are the aspects which are the easiest to have an error in.
4. Test “written maze is equal to read maze” initialises one maze and saves it using the storage class. Then the written maze is read using the storage class, and another maze is initialised based on the read data. The test then checks whether all properties of both mazes are equal.
5. Test “all possible combinations of random mazes” creates all possible mazes constrained by length and width in the range 10 to 200. Then test number 4 is conducted for all these mazes. This test ensured all the possible mazes can be saved and read properly.
6. Test “load non-existent maze” tries to load a maze from a file which does not exist. This test passes if an error is thrown.

9. Known Bugs and missing features

1. One known bug is that when wrong parameters are given for length or breadth when creating a new game, an error will be shown. This is expected behaviour. However, when correct parameters are provided after this error, the opponent's movement becomes extremely unpredictable and does not follow the way it was programmed. It starts to move more than one passage at a time and sometimes does not move at all. My suspicion is that the opponent continues to operate with the faulty maze, causing its movement to be so erratic. I tried to update the maze object in a variety of ways, but this bug could not be resolved. Thus, I was left with no other choice than exiting the game when wrong parameters are entered. Although users can still see the error alert, they cannot continue playing and must restart the game. While this is not ideal, it is a known issue, and I plan to address it as a future improvement when working on the project for my own enjoyment. One possible way to fix this issue would be to create a new game object every time a new game is started.
2. One missing feature is the ability to pause and resume the game. For longer mazes, completing them in one session can be challenging. Implementing a pause feature would enhance the gameplay experience, especially for extended play sessions. Again, I intend to add this functionality at a later stage.

10. 3 weaknesses and 3 best features

Weakness:

1. Code organisation: I think some parts of my code are a bit disorganised. For instance, the mazeGUI (main file) could be more polished and be shorter to make it easier for others to read and understand

my code. Perhaps I could have made a different class for the opponent. This would make the main file less cluttered and divide the code into more sections for better abstraction. As mentioned earlier, sometimes the choice of data structures for smaller tasks was not very well thought-out. This did not have a big significance due to the scale of the project. However, if I was working with significantly more data, there could be better data structure usage for some sub-sections of the code.

2. Multi-threaded nature and associated bugs: I found a bug in the programme which was later solved. When the rat in my game would collide with the opponent, the game would end and the opponent would stop moving, but the alert saying "game over" was not being displayed until the user would press a button. To fix this bug, I deployed a robot from the Java AWT Robot class. This robot would press a button right after the collision to ensure that the game over alert is displayed immediately after collision. Thus the bug was solved. However, I am not completely satisfied with this method and I think it's a weakness which I would like to work on in the future.
3. Opponent erratic movement: As mentioned earlier in the bugs section, the opponent would act unexpectedly after incorrect inputs were entered. This is somewhat related to the multi-threaded nature point I talked about above. I managed to work around this bug, but ideally there should be a more concrete bug resolution.

Best features:

1. Maze solving and generation: The maze solving and generation algorithms were well planned and careful consideration was employed for the choice of data structures used. This resulted in excellent time complexity for both tasks. Both maze generation and solving are almost instantaneous for the parameters expected to be used for this game.
2. Zooming and Panning: The addition of zooming and panning into the maze is an additional feature which was not required to be added. However, I think these features make the gameplay polished by allowing the user to understand the maze and thus enhance user interaction with the GUI.
3. Scoring and highscore: These features were not mandatory either, but I feel like they add a competitive nature to the game, making it more enjoyable. The user can try to beat their own highscore, or they can try to beat someone else's highscore. This competitiveness makes the game more engaging.

11. Deviation from the plan

There was a significant deviation from the plan I presented during the project plan. That is, I was super excited to implement new features and make the gameplay polished. This resulted in making faster progress than expected by me during the planning phase. I got somewhat attached to this project and I was thinking about the project even in my free time.

This taught me a lot about project selection. I am extremely happy I chose a project which I was passionate about. Writing code for this project did not seem like tedious homework to me, instead, I genuinely enjoyed every minute I worked on this project. I found joy in doing this project and seeing the game come to life, which was a rewarding and motivating experience for me. The revised project progress looked like this:

step	hours
1. Project initialising	2
2. Implement Game, Rat and Cell class	10
3. Implement Maze and Maze generation	12
4. Implement Basic GUI	12
5. Implement Maze solving	10
6. File reading and writing	15
7. Testing	10
8. Implementing advanced GUI (zooming and panning)	10
9. Implementing opponent	5
10. Final testing and bugs	15
11. Writing Final document and submission	5

12. Final evaluation

Overall, I am quite content with my project. Lost in Maze offers a solid gameplay for a maze solving labyrinth game. It lets users decide the difficulty based on the inputs given and supports saving and loading of a particular maze. Potential improvements are discussed in the bugs and weaknesses sections.

GUI implementation: I am very happy with how the GUI looks and the choice of Java/Scalafx was a good idea. JavaFX had a lot of examples online, and it was not difficult to use Javafx in scala.

The program structure: The programme structure was understandable and clear. In some parts there could have been some improvement. This is discussed in the weakness section in greater detail.

Save and load: The save file is in a human readable form, and the loading and saving works as expected. Due to the human readable format, it is quite easy to write your own maze files and run them with the game.

Data structures and algorithms: The data structures and algorithms used are mostly well planned and enable fast and scalable performance. In some places, the choice of data structures was not given much importance due to the small scale of the programme.

Version control: Gitlab worked very well as a version control for this project. It was easy to rollback changes to previous states and I found myself using this functionality a lot. I made a total of **118 Commits** for this project, amounting to an average of 2.5 commits each day.

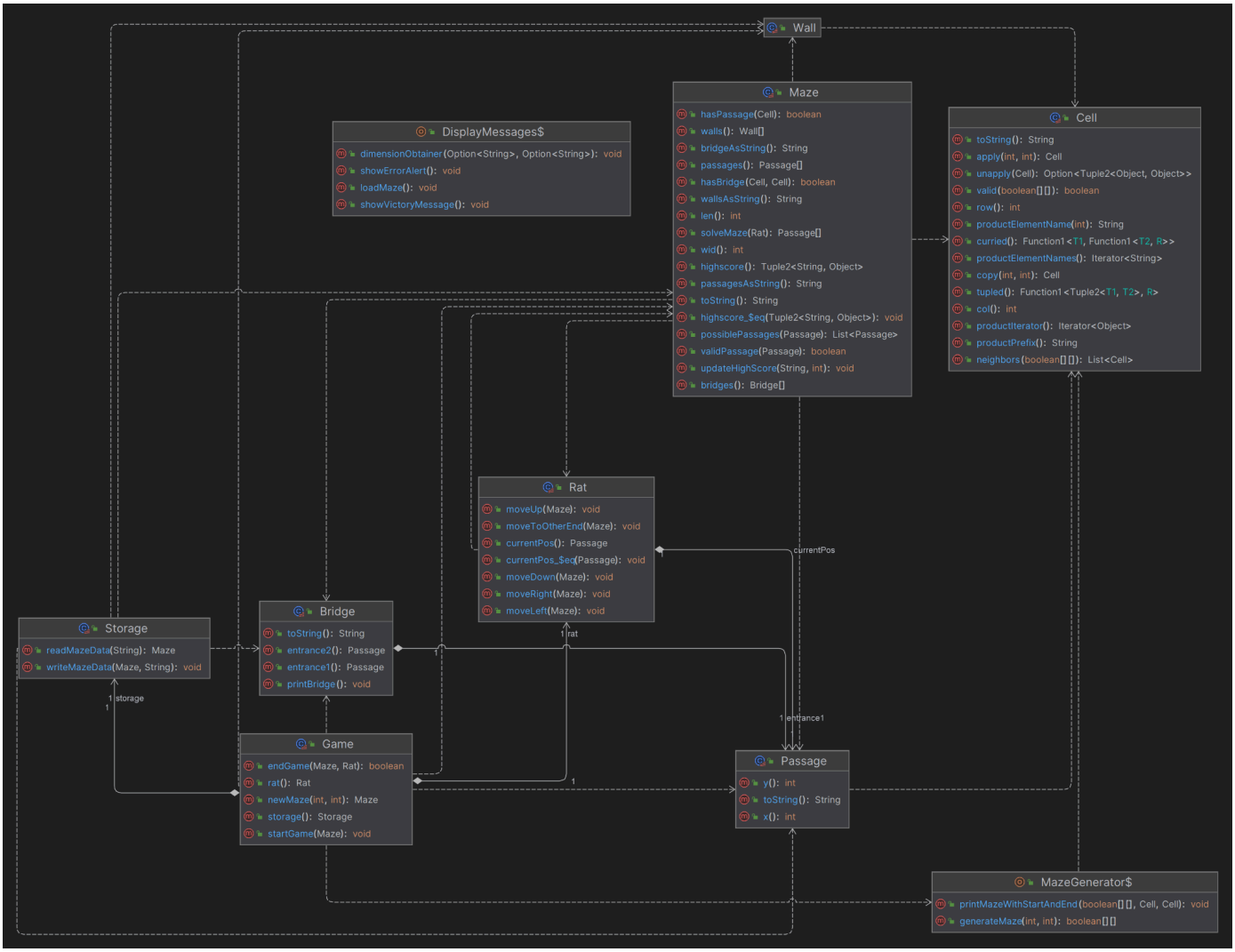
Here I would also like to express my heartfelt gratitude to **Quan Hoang** for his invaluable support and guidance throughout this project. This project wouldn't be nearly as successful without his constant assistance. Thank you Quan, it was a great pleasure to work on this project under your guidance.

13. References

1. Maze Generation: Recursive Backtracking available at Jamisbuck.org
2. Depth First search: DFS | Brilliant Math & Science Wiki
3. Last in First out: LIFO approach in Programming - GeeksforGeeks
4. Scala stack: Stack (scala-lang.org)

14. Appendix

You can find the code and .JAR file in the .zip file. You can also find the code here: Tyagi Ojaswi / lost_in_maze · GitLab (aalto.fi).



UML DIAGRAM