JIMP2 Project Report in Java

Stanisław Dutkiewicz, Filip Kobus

May 2024

Contents

1 Introduction

The aim of the project is to develop a program that allows finding a path in a loaded maze, for example, the shortest path from the entrance to the exit. The maze is provided in the form of a text or binary file, and the program, created in Java, is responsible for loading the maze, finding the path, and saving the maze and the path to a file.

1.1 Problem Statement

The main challenge of the project is to develop a clear graphical interface that allows the user to seamlessly navigate the application's functionalities and create an effective animation illustrating the operation of the algorithms.

An additional requirement is the efficient management of system resources while processing large mazes, despite the fact that there is no direct limit on memory usage in Java applications. However, it is important that the program runs smoothly and quickly, even for mazes with a maximum size of 1024x1024. This challenge requires the implementation of effective search algorithms and the optimal design of data structures used in the program.

1.2 Solution Approach

Two search methods will be used to solve this problem: BFS (Breadth-First Search) and DFS (Depth-First Search), from which the user can choose one. These algorithms will be used to explore the maze to find a solution.

DFS is an algorithm that explores the maze by moving according to predetermined movement priorities (avoiding walls), and when it encounters a dead end, it backtracks to the last encountered node to try a different path. The algorithm is implemented iteratively, which has an advantage over the recursive form due to significantly lower resource usage.

BFS is an algorithm that searches the maze breadthwise, recording possible cells to visit in a previously created queue, and then successively visiting each of them, which allows finding the shortest path to the exit. Its advantage over the DFS algorithm is the certainty of finding the shortest path, which in DFS depends on the initial settings and the configuration of the maze walls.

The implementation of these algorithms in Java required particular attention to effective memory management. To meet this challenge, it was crucial not only to design the algorithms themselves but also to efficiently store data structures, such as stacks and queues, used to remember visited cells and paths in the maze.

1.3 Types of Input Files

Text File: Contains symbols representing maze elements, such as the maze entrance (P), the maze exit (K), walls (X), and spaces that can be moved through (space).

Binary File: Consists of a file header, informing about: maze dimensions, entrance and exit coordinates, and binary representations of maze elements (the same as in the text version). The maze itself is then provided in a form similar to the text version with a slight difference, i.e., before the information about the type of cell (wall or free space), the number of its occurrences in a given place is placed, which allowed for significant reduction of the input file size. The last element of the file is the solution section where the maze author can provide a solution to the maze, although it is not required.

2 Implementation

The project implementation, carried out in Java, focused on solving the navigation problem through the maze using depth-first search (DFS) and breadth-first search (BFS) algorithms, while simultaneously limiting memory usage. A key design decision was the modular system architecture, which facilitated code management and memory optimization. Below are the key modules of the project that together form a cohesive system capable of effectively solving mazes.

2.1 Modules

AlgorithmBfs Module: The `AlgorithmBfs` module implements the breadth-first search (BFS) algorithm to find a path in the maze.

- `queue`: queue storing points to visit.
- `runAlgorithm()`: method running the BFS algorithm.
- `addPossibleMovesToQueue(Point point)`: method adding possible moves to the queue.

AlgorithmDfs Module: The `AlgorithmDfs` module implements the depth-first search (DFS) algorithm to find a path in the maze.

- `stack`: stack storing points to visit.
- `makeMove()`: method performing one step of the DFS algorithm.
- `getMove()`: method returning the last performed move.

Binary Module: The `Binary` module is responsible for converting the maze from binary format to text format.

- `readUnsignedShortLittleEndian(DataInputStream dis)`: method reading 2 bytes from a binary file in little-endian order.
- `convertBinaryToText(String binaryFilePath, String textFilePath)`: method converting a binary file to text format.
- `generateMazeFromEncoding(DataInputStream binaryFile, BufferedWriter textFile, long wordCount, int separator, int wall, int path, int cols, int rows, int entryX, int entryY, int exitX, int exitY)`: method generating a maze based on binary encoding.

DataArray Module: The `DataArray` module represents the data structure storing the maze.

- `array`: two-dimensional array storing the state of each maze cell.
- `putPointIntoArray(Point point)`: method placing a point in the array.
- `setAsVisited(Point point)`: method marking a point as visited.

Point Module: The `Point` module represents a point in the maze, storing its coordinates, type, and parent.

- `x`: x-coordinate of the point.
- `getX()`: method returning the x-coordinate of the point.
- `movePoint(int xDiff, int yDiff)`: method moving the point by the given differences.

FileIO Module: The `FileIO` module handles loading and saving mazes in text and binary formats.

- `readMazeFromFile(File file)`: method loading a maze from a file.
- `writeMazeToFile(BufferedImage image, File file)`: method saving a maze to a file.
- `saveMazeAsText(BufferedImage mazeImage, JFrame window)`: method saving a maze as a text file.

MainGuiPanel Module: The `MainGuiPanel` module is responsible for the user graphical interface and managing the main application panel.

- `mazeRenderer`: object of the `MazeRenderer` class for rendering the maze.
- `run()`: method running the main GUI panel.
- `openMaze()`: method opening a maze file.

MazeRenderer Module: The `MazeRenderer` module is responsible for rendering the maze in the graphical interface.

- `mazeImage`: image of the maze.
- `createMazePanel()`: method creating the maze panel.
- `solveMazeWithBfs(DataArray dataArray)`: method solving the maze using the BFS algorithm.

MazeUtilities Module: The `MazeUtilities` module contains helper functions for handling the maze, such as checking entrance and exit points.

- `setSelectedState(state:  int)`: method setting the selected state.

- `handleMazeCellSelection(MazeRenderer mazeRenderer, int imageX, int imageY, JFrame window)`: method handling maze cell selection.
- `resetEntranceAndExit(MazeRenderer mazeRenderer, JFrame window)`: method resetting the entrance and exit points in the maze.

Main Module: The `Main` module is the main class running the application.

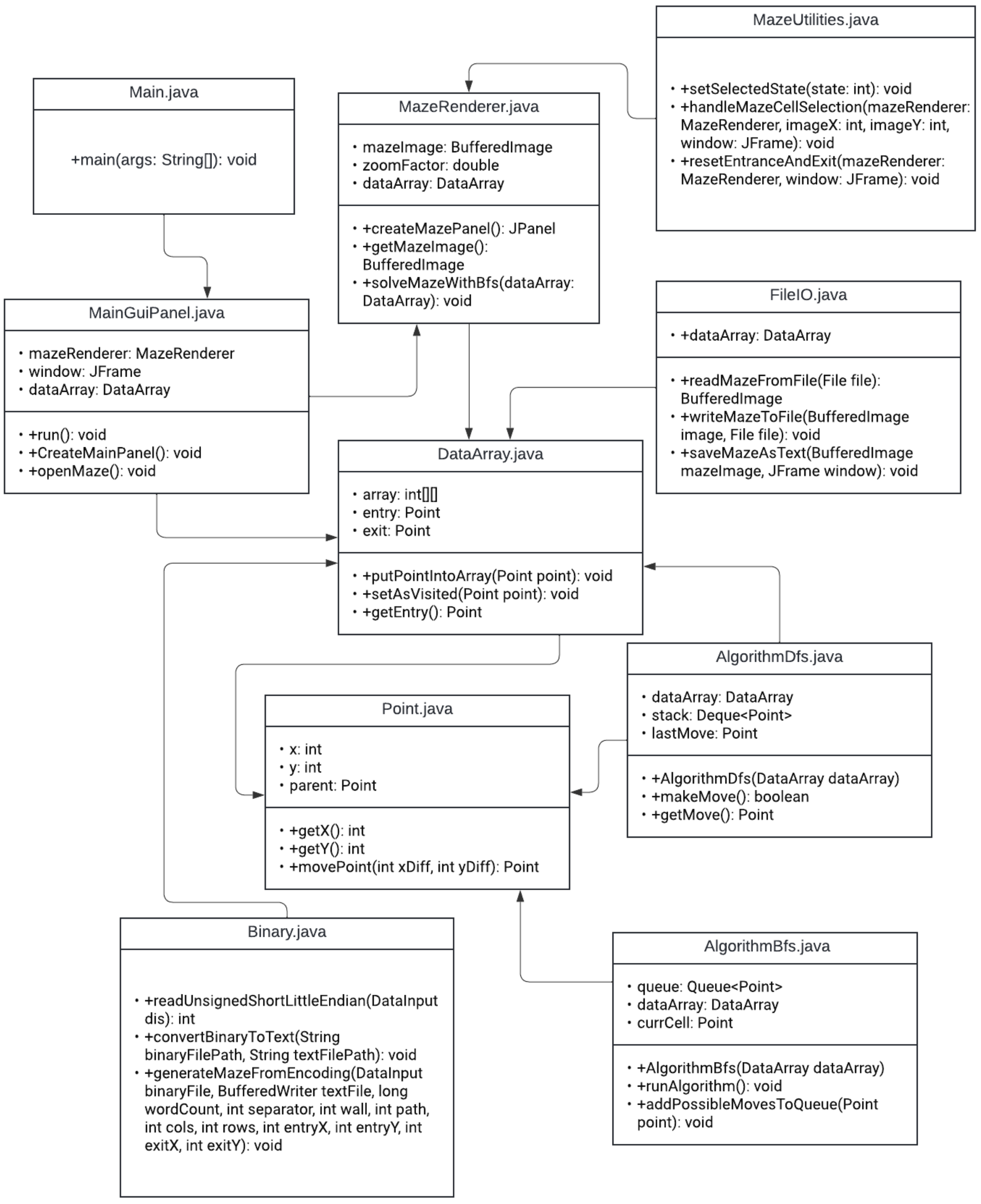
- `main(String[] args)`: main method running the application.

2.2 Design Patterns

- `Singleton`: `FileIO` Module: The `FileIO` class has a private constructor, suggesting the use of the Singleton pattern to ensure that only one instance of this class is used in the program.
- `Strategy`: Algorithms `AlgorithmBfs` and `AlgorithmDfs`: These classes implement different search algorithms (BFS and DFS). They can be treated as different strategies for solving the maze navigation problem, which can be easily switched depending on the needs.
- `Composite`: The `DataArray` class stores the maze structure as an array of points (`Point`). The Composite pattern is used in the way individual maze cells are treated as components of a larger structure.

2.3 Class Diagram

Below is the class diagram created in Lucidchart:



MazeUtilities.java
+setSelectedState(state: int): void
+handleMazeCellSelection(mazeRenderer: MazeRenderer, imageX: int, imageY: int, window: JFrame): void
+resetEntranceAndExit(mazeRenderer: MazeRenderer, window: JFrame): void
Main.java
+main(args: String[]): void
MazeRenderer.java
mazeImage: BufferedImage
zoomFactor: double
dataArray: DataArray
+createMazePanel(): JPanel
+getMazeImage(): BufferedImage
+solveMazeWithBfs(dataArray: DataArray): void
FileIO.java
+dataArray: DataArray
+readMazeFromFile(File file): BufferedImage
+writeMazeToFile(BufferedImage image, File file): void
+saveMazeAsText(BufferedImage mazeImage, JFrame window): void
MainGuiPanel.java
mazeRenderer: MazeRenderer
window: JFrame
dataArray: DataArray
+run(): void
+CreateMainPanel(): void
+openMaze(): void
DataArray.java
array: int[][]
entry: Point
exit: Point
+putPointIntoArray(Point point): void
+setAsVisited(Point point): void
+getEntry(): Point
AlgorithmDfs.java
dataArray: DataArray
stack: Deque<Point>
lastMove: Point
+AlgorithmDfs(DataArray dataArray)
+makeMove(): boolean
+getMove(): Point
Point.java
x: int
y: int
parent: Point
+getX(): int
+getY(): int
+movePoint(int xDiff, int yDiff): Point
Binary.java
+readUnsignedShortLittleEndian(DataInput dis): int
+convertBinaryToText(String binaryFilePath, String textFilePath): void
+generateMazeFromEncoding(DataInput binaryFile, BufferedWriter textFile, long wordCount, int separator, int wall, int path, int cols, int rows, int entryX, int entryY, int exitX, int exitY): void
AlgorithmBfs.java
queue: Queue<Point>
dataArray: DataArray
currCell: Point
+AlgorithmBfs(DataArray dataArray)
+runAlgorithm(): void
+addPossibleMovesToQueue(Point point): void

Fig.1 Class Diagram of the project

3 User Guide

The user guide describes the steps to use the Java maze-solving program. This program allows loading a maze from a file, selecting a solution algorithm (BFS or DFS), and then running the algorithm to find a path from the entrance to the exit. Below are detailed instructions for using the application.

3.1 Environmental Requirements

To run the program, you need to have:

- Java Development Kit (JDK) version 11 or newer,
- Operating system: Windows, macOS, or Linux.

3.2 Installation

1. Download the project files from the repository or another source.
2. Unpack the downloaded archive to a chosen directory.
3. Open a terminal or command prompt and navigate to the project directory.

3.3 Running the Program

1. Configure the runtime environment:
   - Ensure that the JAVA_HOME and PATH environment variables are correctly set.
2. Open the directory containing the Main.java file:

   ```
   cd src\main\java
   ```

3. Compile the program code by typing in the terminal:

   ```
   javac Main.java
   ```

4. Run the application by typing in the terminal:

   ```
   java Main
   ```

3.4 Using the Application

1. After starting the program, the main application window with a menu and a panel for displaying the maze will appear.
2. Select the **File -> Open Maze** option to load a maze from a text or binary file.
3. After loading the maze, you can adjust its view using the zoom in and zoom out buttons.
4. Set the entrance and exit points by clicking on the appropriate maze cells:
   - Select the **Options -> Set Entrance and Exit** option.
   - Click on the selected cell on the edge of the maze to set the entrance point (marked as P).
   - Then click on another cell on the edge of the maze to set the exit point (marked as K).
5. Choose the algorithm you want to use to solve the maze:
   - **Options -> Find Shortest Path (BFS)** to use the breadth-first search algorithm.
   - **Options -> Visualize Path Search (DFS)** to use the depth-first search algorithm.
6. After finding the path, the result will be displayed on the maze panel.
7. You can save the resulting maze to a file by selecting the appropriate option from the **File** menu:
   - **Save Maze (txt)** to save the maze in text format.
   - **Save Maze Image** to save the maze as an image (PNG, JPEG, BMP). </enditemize

3.5 Resetting and Stopping Visualization

- To reset the set paths, select **Options -> Reset Paths**.
- To interrupt the ongoing visualization, also select **Options -> Reset Paths**.

3.6 Closing the Program

To close the program, click the window close button.

4 Summary

The project, aimed at developing a Java program capable of finding paths in mazes, was successfully implemented. By using DFS and BFS algorithms, the program can effectively navigate mazes of various sizes and complexities. The modular system architecture and efficient memory management allowed achieving high performance and scalability of the solution.

During the project work, we gained valuable experience in implementing advanced navigation algorithms, memory management in Java, and software testing. This project provides a solid foundation for further work on more advanced navigation and optimization systems.