Labyrinth Game

Aleksi Kääriäinen

728971

Computer science

1st Year

28/04/2021

---

1. General description

The goal of the project is to program a game that creates and shows a randomly generated labyrinth. The labyrinth must be solvable. The program sets a pawn in the middle of the maze in a free square. The player must control the pawn out of the maze. If the player is not able to find the solution to the maze, they can use a command to show the correct solution to the maze. The maze is not really 2D, as it contains so called bridges which make the paths overlap, allowing traversal over or under another path. Additionally, the maze can be saved from the program to a file. The project will be done by the intermediate requirements.

2. User interface

The program is started by running GUI.scala. Upon starting the game, a window pops up asking for an input number for the games size. The input number can be any number ranging from 10 to 80. Pressing submit creates a game which size is input times input squares.

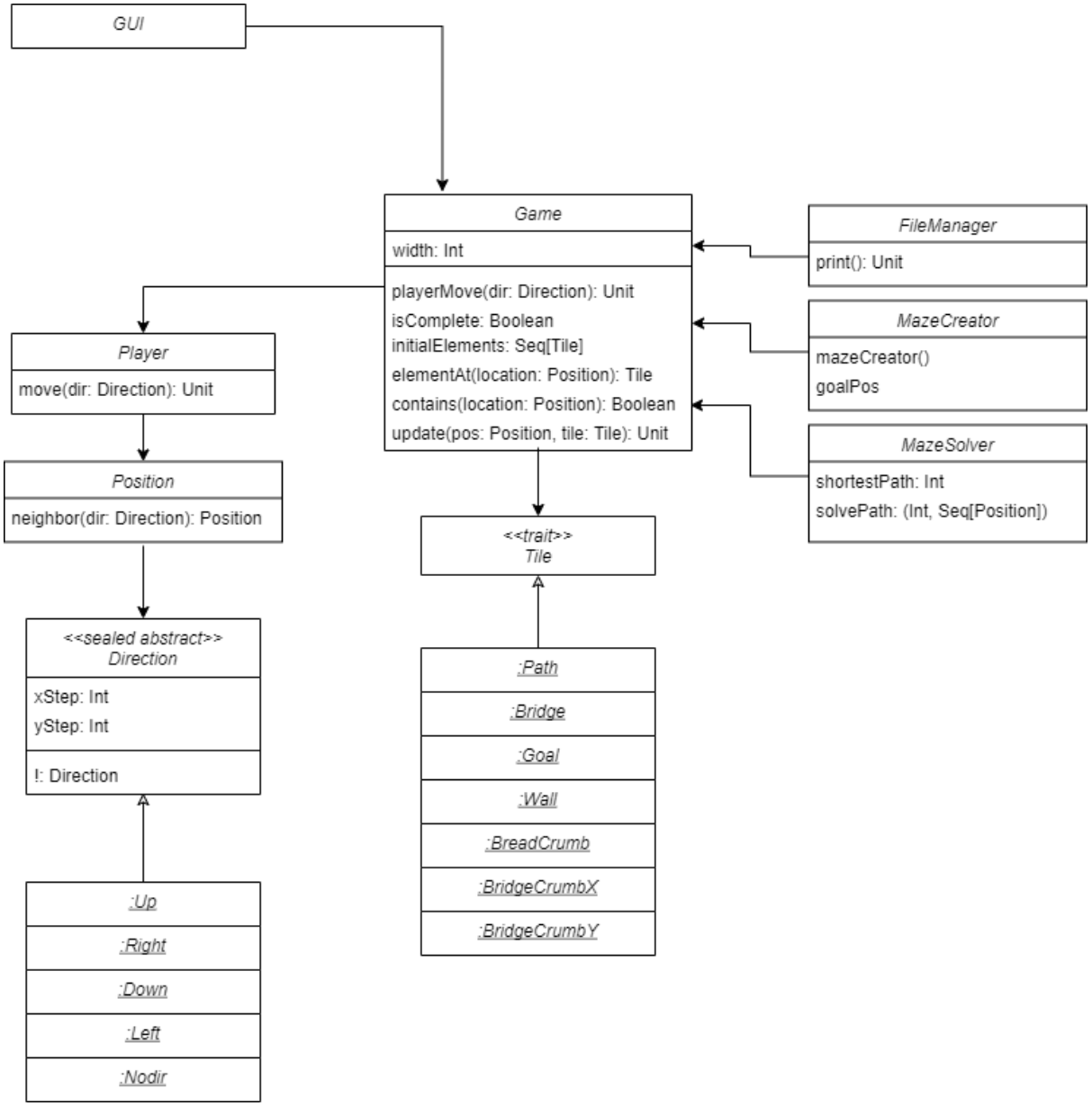
Then, the game view is made visible. The player controls the black arrow that is initially located in the middle of the maze and their goal is to move the arrow on the yellow square that is located somewhere near the edge of the maze.

The player controls the arrow with WASD-keys. Other controls are:

- CTRL + O: prints a line to console which tells the player how many steps are needed to reach the goal through the shortest path.

- CTRL + P: draws the solution to the maze on the screen.
- CTRL + S: Saves the game. Writes a text file ‘LabyrinthSave.txt’ representing the current game state.
- ESC: Closes the program.

3. **Program structure**



The main classes and traits used to describe the game world are Game, Tile, Position and Direction. Game is a grid that consists of Positions. Positions contain Tiles and Directions are used to traverse the Grid. Tiles are used to implement game logic, for example the player is not allowed to walk inside walls.

I tried to make the class structure simple and easily understandable. In earlier stages of the program, I had an abstract class called Grid, which Game inherited. I decided to implement all the methods in Grid to Game and remove Grid completely, because I noticed that Game was Grids only inheritor and that there were no advantages to having the abstract class Grid. I could have included the methods of FileManager, MazeCreator and MazeSolver in Game and eliminated those three classes, but decided against it, because then Game would have been too cluttered and I wanted that a single class depicts a domain of the program

Key methods:

- Game:

- initialElements returns an immutable Seq[Tile], that is filled with game.width times game.width Wall objects. Method is used for initializing the game.
- elementAt returns the Tile at the given position.
- contains determines if a position is in the game. For example Position(5,5) is not in a Game of size 4. Method returns false.
- update changes the Tile in the given position to the given Tile.
- playerMove checks if the player can move in the given direction, and moves the player if the check is successful.
- isComplete checks if the player is standing on the goal.

- Others:

- FileManager.print saves the game by printing the current game state in a text file.
- MazeCreator.mazeCreator is called at the start of the game. It is responsible for randomly generating the maze.
- MazeCreator.goalPos picks a random Position in the Game, then checks if the Position can be the location for Goal. Goal must be on the edge of the grid, must connect to the player and it must be on a former Wall.
- MazeSolver.shortestPath is a recursive branching algorithm that calculates the shortest path from players location to the goal.
- MazeSolver.solvePath is the same algorithm as shortestPath, but it carries over a Seq[Position] that contains the solution to the maze.
- Position.neighbor returns another Position that is next to this Position in the given Direction
- Direction.! returns the direction that is opposite of the Direction it was called on.

4. **Algorithms**

Algorithms used in my program handle the creation and solving of the maze. Algorithm used in maze creation is a recursive algorithm that randomly picks the direction for the maze to continue to. The idea is that the game starts as a grid full of walls, and we start randomly carving the path into the grid full of wall. Result is a 2.5D maze that has overlapping paths.

Pseudocode:

start in the middle of the grid
pick a random direction
continue until all directions are tested {
take 4 elements in selected direction
check if the 4th element is inside the game {

- if 2nd Tile is Path and others are Wall make 2nd a bridge and the others path and continue carving from 4th Tile
- else if 2nd Tile is Bridge make others a Path

} check if 2nd element is inside the grid {

- if 1st and 2nd are Wall make them Path and continue carving from 2nd element

} select next direction and start over

If 4th element is inside the grid, the second check does not do anything.

The program also selects a position for the goal by randomly picking a position, then checking that it is on the edge of the grid and there is a solution from the player to the goal. If the selected position does not fulfill the mentioned conditions, pick another random position until a proper position is found.

The algorithm used for solving the maze is a recursive branching algorithm that returns the shortest path to the goal.

Pseudocode:

def shortestPath(x, y, goalX, goalY) {
start from position (x, y)
mark that position (x, y) is already checked
call the method again but for all 4 directions from the current position
(x + 1, y), (x – 1, y), (x, y + 1), (x, y – 1)
the algorithm is done when (x, y) == (goalX, goalY)
pick shortest(smallest returned Integer) of all branches that end in (goalX, goalY)
}

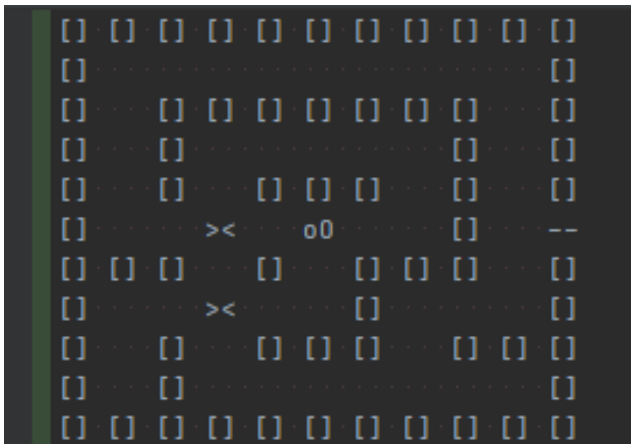
If the algorithm is called on a position that contains a wall or a bridge and an attempt is made to move in a direction that the bridge does not allow, algorithm returns a very large integer so that the algorithm will never pick such a path as solution.

5. **Data structures**

The main use of collections in my program is to store the state of the game. I decided to use mutable arrays to store the game state. In fact, Game class variable content is an Array[Array[Tile]] matrix. This way, accessing information stored in the variable is easy when it is given a Position, because content(Position.x)(Position.y) returns the Tile in Position.

6. **Files and Internet access**

The program loads sprites from image files. The sprites are used in the graphical user interface to represent different Tiles and to show the solution path. The program is also able to save the current game state by writing it into a text file. Different characters represent different game elements. The save is not perfect, because it may lose information about the game. If the player is standing on a Bridge when the game is saved, then the information about the Bridge is lost.



“[]”: Wall

“ “: Path

“><”: Bridge

“—“: Goal

“oO”: Player

a saved game of size 11

7. **Testing**

The testing process was roughly the same as planned. Testing was mainly done via the graphical user interface starting from getting the correct sprites to show up on the screen

when given a fixed game object to testing the game's logic. Maze creation was also tested by running the program and seeing how the maze turned out. Visually analyzing the generated maze helped me realize the algorithms shortcomings and develop it further. Unit testing was used when implementing the shortest path algorithm. The algorithm was given a specific maze with a known shortest path and checking if the algorithm returns the shortest path. When algorithm returned the correct path for the specific maze, I started testing the algorithm with randomly generated mazes.

8. **Known bugs and missing features**

There are a couple of bugs/defects in my program. The maze creation algorithm does not necessarily create a perfect maze, i.e. every part of the maze is not connected to eachother. The maze creation algorithm also checks that the maze is solvable. I noticed that may become a problem when creating a maze of large dimensions(+100 wide). The game can run into a run time error if there is no position on the edge of grid that is eligible for a goal. In such situations, the algorithm ends up in a loop, which results in a stack overflow error. The problem would be fixed if the maze creation algorithm created a perfect maze, because then the maze solver could not get stuck in execution. There is also another bug in the solving algorithm. I have encountered this bug also only on a big maze(circa 100 wide and bigger). It is possible that the game starts as intended, but when calling the shortest path algorithm, it returns 1000000000, which it never should. Solve algorithm returning 1000000000 means that the maze has no solution. Calling the method for drawing the solution results in a run time error. This bug would also be fixed if the maze creation algorithm would always create a perfect maze.

9. **Best sides and weaknesses**

The best sides of my program are in my opinion the solve algorithms. They are fast and effective. While testing the algorithm, some solutions were over 1000 steps long, and even the algorithm which produces the correct path to goal was done in a few seconds.

The weakest part of my program is the GUI. I knew in the beginning of the project that I would not sink countless hours into making the game visually attractive but having no prior experience on scalafx and studying too little on the subject before starting to code the GUI

led to the code being a mess. The GUI does what I want it to do and the visuals are okay in my opinion but the code is hard to read and not structured properly.

10. **Deviations from the plan, realized process and schedule**

I started the project by implementing the classes as I had planned. With the classes done, I started implementing the GUI so I could test the games logic. After two weeks I had done a rough GUI and implemented the core classes. During the next two weeks I implemented the game's logic into the GUI and completed the maze creation algorithm. During the following two weeks I worked on the maze solving algorithm, game's sprites and saving the game. All in all, the order of progress went by the plan, but the time estimates were off by a large amount. For example, getting the graphical user interface to work took me a lot more time than I had thought, and the algorithms and game saving took considerably less time than I had thought they would take. During the course of the project, the class structure went through an overhaul, but the core classes that were in the plan are still there.

11. **Final evaluation**

All in all, the program does what it is required to and despite the few shortcomings, I think it works fine. The maze creation algorithm produces a 2.5-dimensional maze, but the created maze is not necessarily perfect. The solve algorithms are fast and produce the correct solution, but do not necessarily work with bigger mazes due to the defect in the maze creation. In hindsight the class structure could be simplified even more without making the code much harder to read. Because the trait Tile is only inherited by immutable objects, the trait could be disposed of. Instead of using the trait Tile and its inheritors Path, Wall etc. the maze could be depicted as a matrix of integers, where one integer represents one element of the maze.

Extending the program to create for example 3 dimensional space mazes or mazes where the cells are hexagons is pretty much impossible, because the classes and the methods were designed for square-based mazes from the very start.

If I started the project again from the beginning, I would study scalafx more extensively. Also I would the game elements unimplemented and use integers to depict the game elements.

12. **References**

Walter D. Pullen, 'Maze Classification', *Think Labyrinth*, astrolog.org, 2021, Document, http://www.astrolog.org/labyrnth/algrithm.htm, (accessed 28.04.2021)

Unknown author, 'ScalaFX', *scalafx,* 2021, http://www.scalafx.org/docs/home/, http://www.scalafx.org/docs/quickstart/, http://www.scalafx.org/docs/properties/, http://www.scalafx.org/docs/dialogs_and_alerts/, (accessed 28.04.2021)

Unknown author, 'JavaFX Tutorial', *tutorialspoint,* 2021, website, https://www.tutorialspoint.com/javafx/index.htm, (accessed 28.04.2021)

Unknown author, 'Maze generation algorithm', *Wikipedia,* 2021, website, https://en.wikipedia.org/wiki/Maze_generation_algorithm, (accessed 28.04.2021)

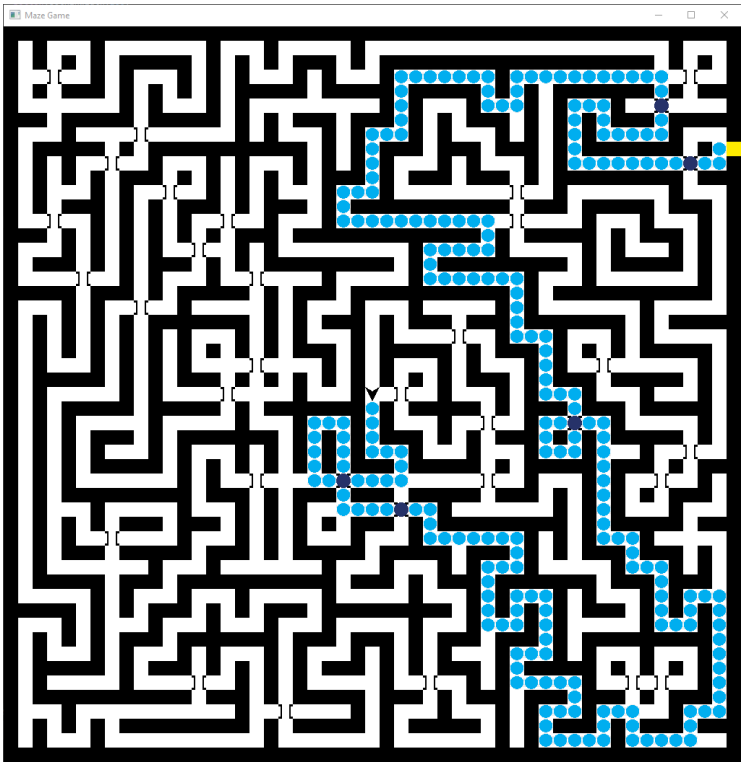
'Keyboard and Mouse Input 1 (in ScalaFX)'[Video], *Youtube*, Mark Lewis, 2015, Video, https://www.youtube.com/watch?v=Zqmo0ba58Mo, (accessed 28.04.2021)

'Recursive Maze Solution (in Scala)', *Youtube*, Mark Lewis, 2015, Video, https://www.youtube.com/watch?v=jmztV7nYg_k, (accessed 28.04.2021)

13. **Appendixes**

First view when the program is run.

A maze of size 51



Same maze with the solution shown. Note that the light blue circle indicates that under it is a normal path and a dark blue circle that under it is a bridge.