Final Year Project Report

Full Unit - Final Report

Solving "2048" using AI

Diego Toledano

A report submitted in part fulfilment of the degree of

BSc Computer Science with Management

Supervisor: Khuong Nguyen

Department of Computer Science
Royal Holloway, University of London

April 22, 2020

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 14536

Student Name: Diego Toledano

Date of Submission: 22.04.2020

Signature: Diego Toledano

Table of Contents

Abstract

2048 is a great game! It is a puzzle where you are fighting randomness. How to win all the time? Is it possible? Can AI help?

Maybe as we can never stop randomness, we can threat this issue with using large number of trials and averages. This where AI comes in the pictures.

There are many approaches or algorithms. One of them is the most appropriate and shows great results.

To establish the best strategy, we need to follow certain steps: Make simple and fancy GUI, the puzzle should be displayed on CMD while calculations are done by the RAM for testing purpose, use the best AI algorithm to solve it and to it faster.

Of course, the AI should solve the real puzzle game like the remake version, used for test.

I have found multiple Algorithm that can solve the puzzle in five minutes, and even achieve bigger number on the grid than 2048!

But since randomness is the biggest risk I faced, and cannot annihilate it, I had to treat this risk. By doing so my solutions are either to use a special algorithm that tries to predict randomness by doing a task a multiple time, the Monte-Carlo tree search and/or to use a special algorithm that that tries to minimize the maximum loose which is in that case depending on heuristics values, Minimax.

After all of this, is it working?

For most of the attempts, the AI is outputting the expectations.

But in fact, as a result of this experiment, we will see that it cannot conventionally solve this puzzle game on a 100% basis, since randomness is accounting too much in this puzzle. This is not like conventional puzzles like Sudoku or conventional games like Chess where you can solve them easily, using Backtracking, Brut Force algorithms, or by knowing all the possible states of the game before starting resolution.

For MCTS, results are 80% chances of achieving 2048 with 100 runs, 90% with 500 runs and 99% with 1000 runs. The more runs you do, the longer it takes... For Minimax, results are equivalents, the main point is that we must consider the ratio result/time...

Project Specification

My project is solving the puzzle game “2048” by using an AI. “2048” is a puzzle game made by Gabriele Cirulli, a Dutch product designer and developer. The game was made in 2014 and the development of it took two days.

As seen on Figure 1 bellow, the puzzle is a 4 by 4 grid with two numbers (either 2 or 4) placed randomly on the grid. You can do only four actions which are moving tiles in four directions (Up, Left, Down and Right). Each time you move tiles, a new tile with either a 2 or a 4 spawn randomly on a blank space on the grid. When two tiles with the same value touch, they merge and the sum of those two digits are added to the total score. The goal of the puzzle game is to achieve “2048” by merging tiles.

As said before, my project is solving the puzzle game “2048” by using an AI. This means that my project a software that solve the real puzzle game made by Gabriele Cirulli. Hence, my project is about making an AI that can solve interact with the puzzle game thinks by itself and solve the puzzle. In order to do so, the AI should scan the puzzle live, thinks with an algorithm (Here, Monte-Carlo tree search or Minimax) and solve the puzzle using either keyboard input or by controlling the webpage or the mobile app.

In addition to that, the software should have a fancy and user-friendly GUI that allow the user to start the software with an action button. Once you have clicked this button an AI should solve the puzzle. Finally, the puzzle game should be able to be reset without restarting the software and/or refreshing the web page and keep track of the score.

The design of the puzzle game and of the GUI, should be the same as the original puzzle made by Gabriele Cirulli and follow Nielsen and Molich’s User Interface Design Guidelines.

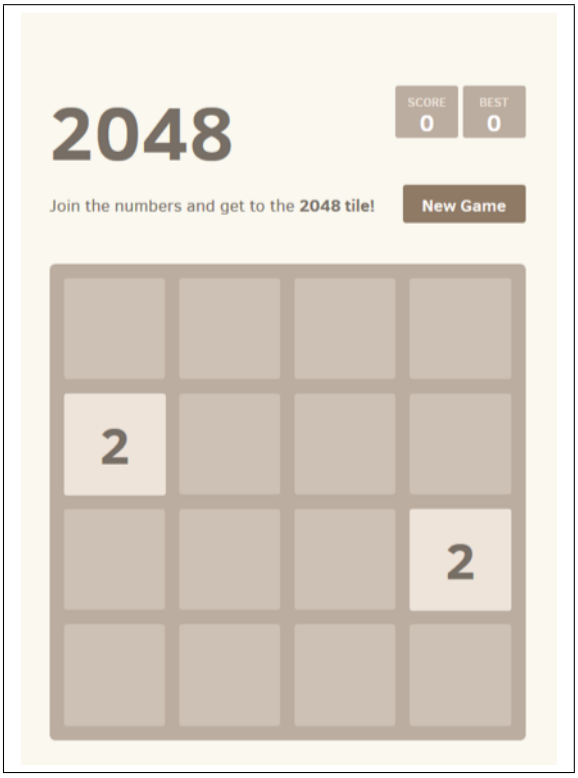

Figure 1: Design of the original puzzle.

Chapter 1: **Introduction**

This chapter is the Introduction one where you will find the Introduction where I explain my motivations for this project. You will then understand why I have done this project. Moreover, here you will learn about the background of this project. This means that you will learn more about the puzzle, who made it, how and why. You will also know the basic theory for solving puzzles my aims of this project and a section about the structure of this project.

1.1 Motivations

This project was made because randomness makes solving puzzles more complex and interesting. I had the idea to use AI to help us and choose 2048 as a good example for that. Consequently, I needed to make a working AI that can solve the 2048 puzzle. By the end of the year, I want to remake 2048 and to solve it using different Algorithm and AI technics. So, I want to learn how to make AIs and how to make fancy GUIs. Another aim that I want to achieve is to learn more about puzzles games in general but especially about the puzzle game 2048. In addition to that, by doing this project, I want to learn more about randomness and especially in puzzle games.

Playing games is one of the biggest passions I have. Among the big variety of game that exists, I do play a lot of puzzle games like sudoku, karoku puzzle, 2048 and connect 4. This is since nowadays those games are quite easy to access whether it is online or on our smart phones. We can have access to all those puzzle games on our smartphones. On the other hand, I have always considered working as a game developer or just as making mobile games. I know I might not do a mobile application, but puzzles games are the type of games that sell best. In addition to that, since I play a lot of puzzle games, I am happy to use AI to solve those puzzles for me. This is a good opportunity for me to learn how to deal with big project like this and to learn how to make an AI and/or make "fancy" GUI so I really think that this will help me in my career.

1.2 Background

1.2.1 Background

Nowadays, among the great variety of games that we have, we can find puzzles games.

Between all those puzzle games, the one I am solving with the help of an AI is "2048" which, as said before, was created by Gabriele Cirulli, in a weekend "just for fun". This game is heavily inspired by two games, a game called 1024! And a game called Threes which are roughly identical with 2048. 2048 was published as an open-source software of GitHub. This is where the puzzle game increase in popularity. One week after the release people got jealous of the success of this puzzle game and attracted a lot of controversies. 2048 was even plagiarized.

Those puzzles like 2048 are mainly solo games that are quite easy to solve once you know the steps to solve it, the algorithm. This algorithm can be found and used by an AI. The AI

needs to find the best algorithm to solve those puzzles games. This algorithm is said to be the best if it is the fastest one or the one that can go the further than the other algorithm. This algorithm is the “Key” to solve those puzzles, once you have it, you can technically solve any state of the game.

Since we are dealing with randomness, there are few Algorithms that can solve puzzles with randomness. The three most famous ones are the Monte-Carlo tree search, Minimax and Expectiminimax. Those three algorithms have the same goal which is to find the best move at a given state of the puzzle and has the same way of dealing with the randomness which is to do an something that leads to reducing the impact of the randomness on the decision making. As said before, Monte-Carlo tree search does multiple runs of the puzzle in the background and try to match the most fruitful while Minimax tries to minimise the maximum loose which is, in that case, depending on heuristics values. Finally, Expectiminimax that tries to predict the average of the minimise loose and of the maximum loose which is, in that case, is also depending on heuristics values.

1.2.2 Literature Survey

To solve this puzzle, there are two things you need to look for. The first one is to have a look at the puzzle itself, to play it and learn about the mechanics of this puzzle. To do so, you would have to check at the puzzle game’s creator website [9] that can make you learn more about the creator and the game. On the creator’s website, you will also find his GitHub [10] where you can have a look at how the game was made and how to “copy” it. The second thing you need to have a look at are some algorithms that allow you to solve the puzzle game. The three most famous ones as previously discuss, can be find here [14] and [28], here [26] and here [30]

In my case, I am using the Monte Carlo Tree Search algorithm which you can find a copy of it easier for 2048 at a website made by Matt Overlan and Ronen Zilberman [12] which are two developers who manage to find a viable solution to solve the puzzle game. My AI is heavily inspired by their AI. I am also using Minimax. For this one my implementation is just the pseudo-code translated to java, however, I have used weighted matrices to force the good positioning of values on the board [21] as heuristic values.

However, if you want to become an expert in solving puzzles with AI I recommend you start by solving less complex puzzles that do not involve randomness issues like 2048. Consequently, I recommend you to have a look at how to solve Sudoku [8] and thus on how backtracking works [15]. On these two websites, you should learn basics technics that will be useful to solve pretty much every non-random puzzle. If you want to have a look to randomness and try to predict it, you should have look at the journal [3]

Finally, the best way I recommend to become an expert in solving puzzles is just to practice. What I mean by that is that the best way to learn is to play and try by yourself. Since nowadays we have access to all those puzzles on our smartphone it is easy for us to play and practice.

1.3 Research questions and research objectives

At the end of this project, I would like to:

- Have a working puzzle game with some “special features” like loading and saving the state of the puzzle.
- Have a working and efficient AI that can solve the puzzle game 2048. I want to have something I can be proud of.
- Learn more skills on how to deal with big projects.
- Learn how to make efficient and beautiful GUI.
- How to make an AI and/or how to teach to an AI.
- Learn about this puzzle game that I like.
- Improve my skills and knowledge about this game. - As a comparison, this is like chess, by doing a project like this and studying what the AI is doing, I will know more strategies and be able to learn new skills.
- Be able for the AI to deal with randomness. - Because of this, the AI cannot predict the next move and be like “chess” AIs that think more than 10 moves ahead.
- Learn how random works and how to predict it.

By the end of this project, I would like to know more about the implication and the consequences of dealing with and making an AI. Since my AI is making small decisions using an Algorithm, it can be easily transposed to a bigger decision-making process or to take more important decisions. If we push it to the extreme, the AI can then take decisions in which life are considering. In that case, by the end of this project, I should be able to make sure that my AI cannot lead to issues like this.

1.4 Structure of this report

This project is divided into six parts/ chapters that are:

Firstly, you will find the Introduction where I explain my motivations. The background of this project, the aims, and a section about the structure of this it.

Then, there is the Timeline where you will understand what I have done with a timeline section where I give the key dates and deadline of my work. In this chapter, you will also find my project dairy and the key issues I have faced during my project and research.

The third chapter is about how I solved the puzzle. Over there you will find proofs of concepts I have made during the first and second term and an explanation of the algorithm I have used for my AI.

The fourth chapter is about Software engineering and especially how I have used a Revision Control System and how I have done tests. In addition to that, over there, you will find a UML diagram that explains how my AI solver works.

The fifth chapter looks at the professional issues I have encountered and think about the ethical implication of them. In this chapter, you will also find how AI leads to many implications, and sometimes, you have not to think of them earlier.

Finally, the last chapter is the conclusion of this report where you will find a summary of my work and my futures intentions.

Chapter 2: **Timeline**

This chapter is the Timeline one where you will understand what I have done with a timeline section where I give the key dates and deadline of my work. In addition to that, in this section you can understand my working process, the mistakes I have made, and the corrections have done. In this chapter, you will also find my project dairy and the key issues I have faced during my project and research.

2.1 Timeline

Comparing to the timeline I have done for the project plan, the actual timeline I had been following during this first term is different. The timeline is as follows.

For the first term, I had to submit the project plan on the fourth of October. Then, on the fifth, I started the project. I spent the following week trying to re-adapt myself with using JavaFX and the drag and drop sample controller so I will be able to make a GUI for the puzzle. On the twentieth of October, I have made a simple GUI using an active button and a colourful background. Later, around the twenty-fourth of October, I have made a non-binary trees program thinking it could be useful to see every path to solve the puzzle game. I ended up not using it because we cannot predict the future with randomness in it. After that, I started working on the actual AI. On the first of November, I have made a playable 2048 in CMD. Since I needed to focus on the AI, I have had to make this playable CMD puzzle so I can test my AI. Finally, on the fourteenth of November, I have made a working AI that can solve the puzzle under 1h30. One week later, I have managed to reduce this time to roughly one minute.

Comparing to the timeline I have done above, the actual timeline I had been following during this second term is different. The timeline is as follows.

At the end of the first term, I was planning on making either an AI that can go beyond 2048 and achieve 4096 (it turned out that this was easily achievable by the first Algorithm) and a mobile version of the puzzle. In addition to that, I was planning on making my own version of the puzzle game. As said before the first Algorithm can easily achieve 4096, we just need to wait longer for him to think and do the actions. The mobile app idea was too ambitious, too complex and unrelated to this project so I had to cancel this idea while for the idea of making my own version of the puzzle had to be cancelled since the puzzle game is using randomness.

Indeed, my implementation of that randomness might not be the same as Gabriele Cirulli's one. In that case, every analyse of the randomness or randomness anticipation to do better moves in fake since it might not be applicable on the real game.

During the second term of this project, I have been focusing on three new features I wanted to add to my 2048 AI solver. They are all very important, however, I will say that the first feature was necessary for the development of this project. Indeed, the first feature was changing from an AI that solves the 2048 puzzle game in the command line to an AI that solves the real 2048 puzzle game made by Gabriele Cirulli. This has been made around the first week of February. Later, from the sixth of February to the twentieth of February I have been working on making a fancy GUI reusing the template I have done during the first term (cf. Proof of Concept 1) using JavaFX and the drag and drop sample controller. However, this time I have used some CSS to make it look like Gabriele Cirulli's puzzle game and more

user friendly than the first proof of concept. Finally, from the twenty-first of February to the end of March, I have been working on a second Algorithm to solve the 2048 puzzle game so I can have an element of comparison and discuss which one is faster and go further.

Finally, during the Easter break, I will focus on the demo presentation of this project.

2.2 Project Dairy

3:52 pm on October 11, 2019: “Work on the proof of concept number 1. I have readapted myself with JavaFX and made a Simple GUI that changes colours”.

4:07 pm on October 23, 2019: “Finish the report on Design pattern (might have to come back to it later in the project), I have merged the Proof of Concept number 1. I have to clean up the report and merge it into master.”

5:58 pm on October 24, 2019: “Finish Proof of Concept number 2 on Non-Binary Trees (ready to merge) and started Proof of Concept number 3 on about a playable 2048. “CMD GUI” is working”

10:53 am on October 27, 2019: “Almost finish Proof of Concept number 3 that is making 2048 in the CMD in java (need to add score, game over and win)”

2:52 pm on November 1, 2019: “Finish the CMD playable game.”

2:58 pm on November 4, 2019: “Fix some issues on the Proof of concepts 3 which is the CMD playable 2048 puzzle and merged the branch”

3:40 pm on November 5, 2019: “Create a Facade for the playable CMD 2048 puzzle so it’s easier for “the AI” to solve it”

5:01 pm on November 6, 2019: “Finnish the Facade and created an AI solver that just need to add the algorithm. Also solved some issues”

5:17 pm on November 7, 2019: “Keep trying on solving the puzzle, I have to try predictions of the next move but it’s difficult because there is some randomness in this puzzle. Right now, the max score is 256.”

5:00 pm on November 8, 2019: “Still Trying to solve the puzzle, my new strategy is to play multiple games and figured out the best move each time.”

3:17 pm on November 29, 2019: “Improve the algorithm so the Ai can solve the puzzle in roughly one minute.”

4:58 pm on February 5, 2020: “Improve the AI so it can now solve the “real” puzzle game (the online one) available at https://play2048.co/”

6:25 pm on February 20, 2020: “Made a working GUI for the user, so he can easily scan the puzzle and let the AI play with it.”

6:25 pm on March 1, 2020: “Have done some test and research on how to use another method to solve the puzzle”

6:26 pm on March 4, 2020: “Starting minimax by creating a TreeNode of every state of the game depth of 3”

4:20 pm on March 9, 2020: “Minmax is running on the test puzzle. still have to do some debugging and make sure it is working on the real puzzle”

11:28 pm on March 11, 2020: “Done some debugging on the tree node class so minimax can work on it”

7:28 pm on March 14, 2020: “Used a weight matrix for the heuristic, seems to be working, however, I have some bugs with my implementation of minimax”

2.3 Issues and Risks

While I was working on this project, I have only faced some issues and risks.

The first issues I have faced was, finding the algorithm to solve the puzzle game. I have spent most of my time dealing with this issue. The issue was that every algorithm/solution that worked for human player could not work for my AI. As an example, according to almost everyone (including me), to achieve 2048, you should make sure to keep the highest number in one corner. This strategy does not work with an AI because some time you are forced to move the highest value out of the corner due to randomness and the AI does not know how to save the situation. Out of all those conventional strategies, I have tried to play with heuristics values and tried to prioritise score over longevity and all the way around. I have also tried to maximise the number of empty/black square on the grid. Unfortunately, none of those technics manage to achieve the 2048 scare on the grid. In addition to that, the other heuristics value I have tried that manage to achieve a good score was weighted matrices see Figure 2.1 bellow. Those matrices are called weighted matrices because it gives values (weight) to scares on the grid and says which scare is more important. Hence those important scares should contain the biggest values of the grid. This help figuring out how good a board is which the positions of the squares. As a heuristics value for the Monte-Carlo tree search algorithm, it did not change to achieve 2048. However, as a heuristics value for the Minimax and the Expectiminimax algorithm, it manages to achieve 2048 and promising results.

Figure 2.1: Example of a weighted matrix.

As said before, randomness can ruin the situation. Hence, the second risk I have faced was while my AI is solving the puzzle game is Randomness. Since every time someone “makes a move”, a new value is randomly added to the board, we cannot do like chess and predict/think multiple moves ahead. In order to treat this risk, I have used a special algorithm that I will explain later. However, this involves doing a task many times. The more the AI is doing this task the more the AI will be able to determine which move is the best.

Finally, the third issue that I had encountered was making sure my AI respect some ethical behaviour. As mentioned earlier, one of my aims for this project was making sure I did not have ethical issues with my project. Since my project is an AI I have had to make sure nothing can be reused to make complexes dissections or for other purposes. In other to do so, one of my solutions was to make a very specific AI that can only interact with the real 2048. In order to do so, I have been scanning the pixels of the puzzle so my AI can only recognize and interact with the real puzzle.

Chapter 3: **Solving the 2048 puzzle game**

This chapter is about how I solved the puzzle. Over there you will find the proofs of concepts I have made during the first term and the second term. You will then find an explanation of the fives proof of concepts programs, how they work, why I have made them and how they are related to each other. You will also understand the algorithms I have used for my AI.

3.1 Proofs of concepts

While I was working on this project, I had to make several proofs of concept programs. I have made four relevant proofs of concept programs and one non-relevant one that turn out during the second term of this project to be relevant. All those programs are: ”Simple colourful GUI with an active button”, ”Relevant data structures”, ”Playable CMD 2048 puzzle game”, ”AI that can solve 2048 using the Monte Carlo tree search algorithm” and ”AI that can solve 2048 using the Minimax algorithm”

So, as said before, the first proof of concept program I have done was the simple GUI one as seen on Figure 3.1 bellow. This means that I have made using JavaFX and the drag and drop scene builder. This program consists of a simple window and a big button saying ”Click me to change colour” which obviously, when you click it, randomly change the colour of the background using random integers between 1 and 255 so we can change the RGB values of the background.

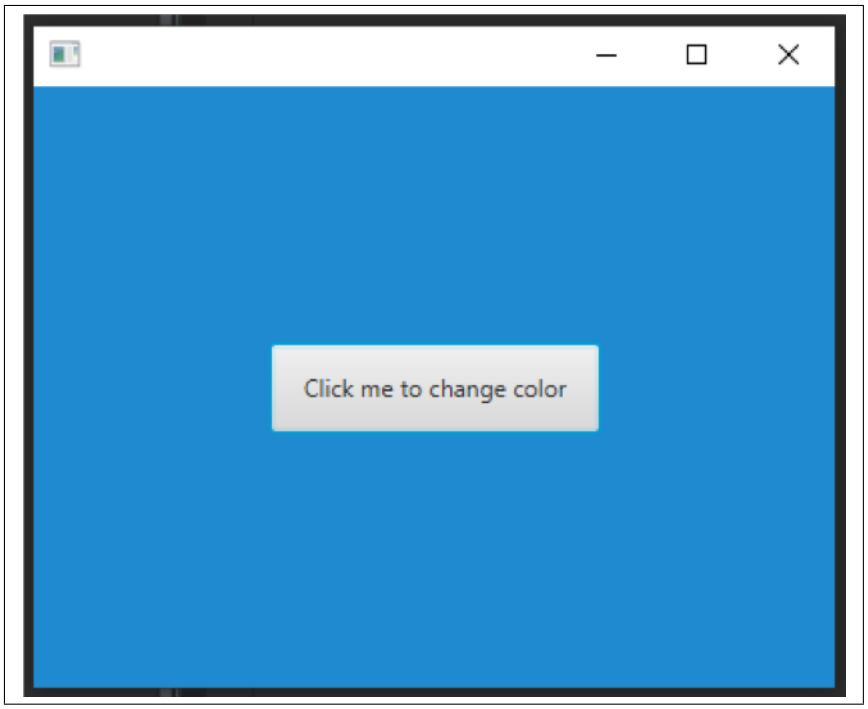

Figure 3.1: Proof of Concept 1

Then, the second proof of concepts program, I have done is the non-relevant one called ”Relevant data structures. The data structures I have decided to make is Non-Binary trees. This proof of concept program is said to be non-relevant because I have not used Non-Binary trees to solve the puzzle game using Monte Carlo tree search approach. I have made this program because, by the time I was coding it, I was thinking that my AI will be able to

predict every future move and therefore to store every action into a tree. I was thinking it was like chess when you have every state of the game stored in a tree and that all I had to do was to apply a search on that tree to find the best solution. However, since I have to take into account randomness, it is impossible for me and for my AI to predict future states of the game. However, this proof of concepts became useful when I started using Minimax to solve the puzzle game. De facto, my implementation of Minimax is going through a Non-Binary tree node to find the next move of a given state of the puzzle while even though it is called a tree search, my implementation of the Monte Carlo tree search is not using a Non-Binary tree to search for the next move.

After that, I have started to make the third proof of concepts program which is making a playable 2048 puzzle game in CMD I have done this for test purpose and so I can easily focus on my algorithm later. To do that I had to understand how the game exactly works. I have spent a few days doing some research on that. Then I have started coding the game. The board is a 4x4 grid, so I have used a 4x4 two-dimensional array with 0s as blank/empty positions on the grid. I am using a scanner to detect user inputs so the user can press ‘W’ to slide up, ‘A’ to slide left, ‘S’ to slide down and ‘D’ for right. Each time the user slide in one direction, the program randomly adds a new digit to an empty position of the board. If the user input is invalid, the program asks the user “Press the correct key”. In addition to that, the program keeps a track of the score and displays it as seen on Figure 3.2 bellow.

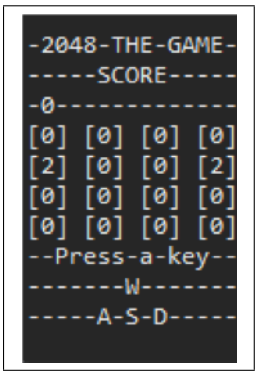

Figure 3.2: Proof of Concept 3

Then, my fourth proof of concept program is an AI that solves the puzzle using the Monte Carlo tree search algorithm. The first thing I have done was to make an Interface design pattern, so it is easier for my AI to play with the puzzle. Since my puzzle is working in the CMD, I could not use buttons, therefore, in order to turn the AI “on”, you have to use the scanner and to type “solve” instead of the classic WASD. Then, I have tried multiple methods to solve this puzzle. However, every conventional strategy did not work due to randomness. Consequently, I had to come out with a new approach. The approach of this proof of concept is called Monte Carlo tree search.

Finally, the last proof of concept program I have made is an AI that solves the puzzle game using the Minimax algorithms. As I was coding and programming it, I was testing and running it using the CMD puzzle. Hence in other to turn the AI “on” and see if the AI can achieve 2048 you have to use the scanner and to type “solve” instead of the classic WASD. The approach of this proof of concept is called Minimax.

Since the fourth and fifth proof of concepts were running in CMD, I have made a GUI as seen on Figure 3.3 bellow, that allows the user to control the AI with two actions which are to scan and to solve the puzzle.

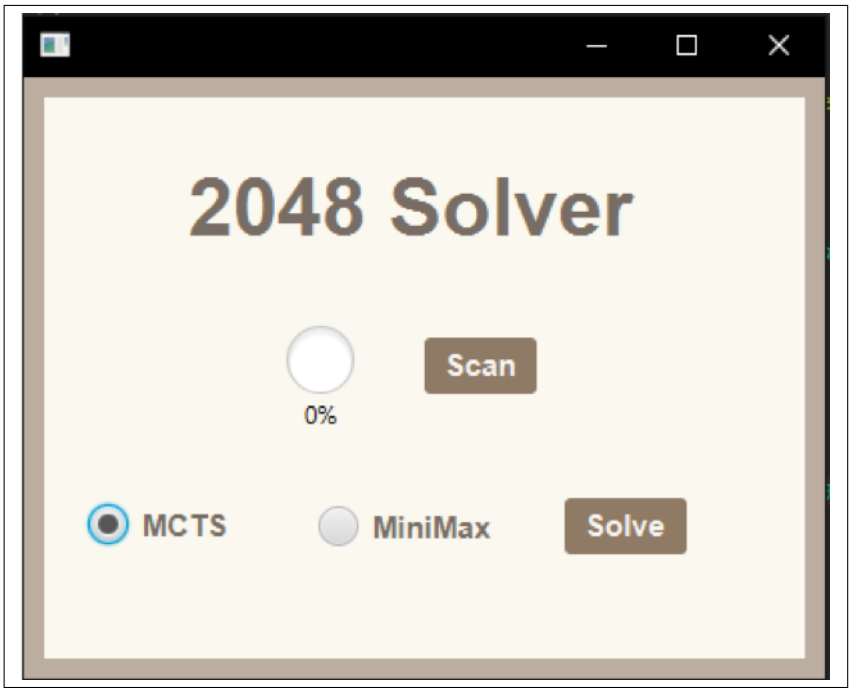

Figure 3.3: GUI of the AI

3.2 The Algorithms

The Algorithms my AI are using are Algorithms that treat the randomness issue. To solve the puzzle, the AI need to know what to do for each state of the puzzle. Therefore, for every states of the puzzle game, the AI needs to figure out in which direction it needs to slide. To do so I am using two different algorithms. As mentioned earlier, I am using the Monte Carlo tree search algorithm and the Minimax algorithm.

3.2.1 Monte Carlo tree search

One of the two algorithm I have used for my AI is the Monte Carlo tree search algorithm which is an algorithm that take place using four steps, selection, expansion, simulation, and backpropagation.

The selection step is all about setting up the tree, it is about selecting the game state as the root note and the children of this root are all the possible states according to the moves the AI can do without a simulation. In the case of 2048, the root node is the state of the puzzle game while the children are the four states after the four moves. Hence, the children are the next state after sliding left, the next state after sliding right, the next state after sliding up and the next state after sliding down.

The expansion step is about increasing the tree size by reaching every states of the puzzle game. This is done by creating a child each time the leaf is not an end node (either a win of a loss) That mean that is you can keep playing, you add a node to the tree. In the case of the puzzle game, the if the current state of the board is not a game over or a win, the AI keeps exploring the current board state’s children.

The simulation step is the step where the AI do multiple random playout throughout the tree. This means that since we have a built-in tree from the previous state, we can explore random paths of that tree. In the case of the puzzle game, it means that the AI must plays

multiples random games and keep their data.

The backpropagation step is the thinking step. In words, it is the step in which the AI use the previous stored data and think on what the best next move is according to those data. On the puzzle game point of view this is where the AI decided to slide and go to the next state.

Finally, the AI needs to restart this whole process for each states of the puzzle game until the puzzle game is solved or until the AI achieve a game over so the AI can solve the game. In case of a game over, the AI should either increase the number of random games done during the simulation step or just try again. Since luck is involve due to randomness and that the AI does not know how to do some “RNG manipulation” we are not a hundred percent time sure that the puzzle can be solve at that given state. However, the AI is trying to reduce this risk.

Despite the name of the algorithm, my implementation of this algorithm has nothing to do with a tree. De facto, since the Monte Carlo tree search algorithm is mainly used for strategy games like poker or the go game and not really for puzzle games that involve randomness, I have had to readapt the algorithm. Therefore, my implementation of this algorithm comes as follow:

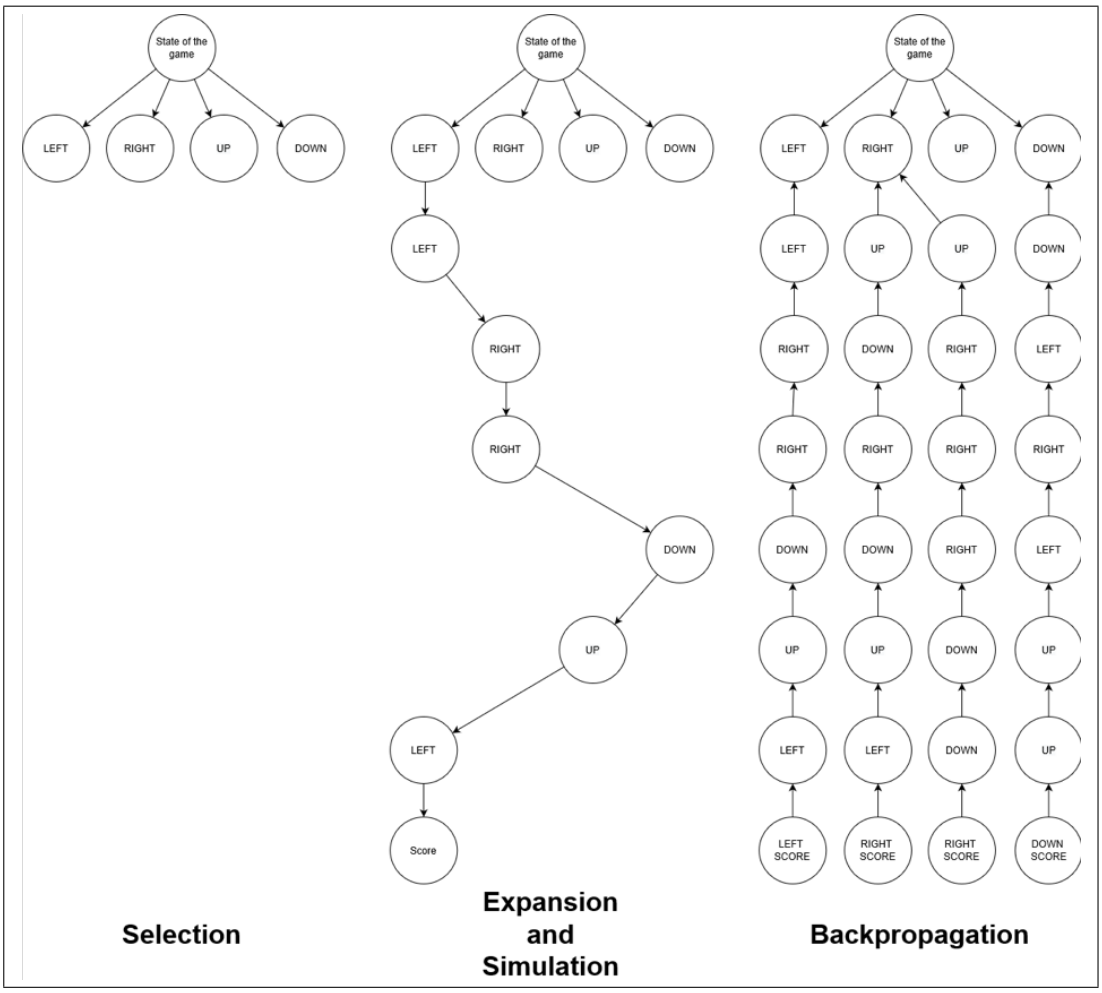

Figure 3.4: GUI of the AI

According to the steps above, my implementation is only using the selection step, the simulation step, and the backpropagation step see Figure 3.4. To be more precise, the simulation and the expansion step are combined in one step. Seeing that we cannot predict next moves due to randomness, the expansion must be done while the simulation is done. Consequently,

for each states of the puzzle games, the AI setup the four different “paths” that could be starting with either by sliding left, by sliding right, by sliding up or by sliding down. After that, the AI do some simulation and play randomly n times a game in the background and keep track for each of those games of the first move they have done and the score after m number of moves. Here n is the number of times the AI do some simulations while m is the “depth” of those simulations. The bigger n is the better the Algorithm will be. Same thing for m. Consequently, the bigger those values are, the longer the Algorithm will take. Hence, I have found that a good ratio between time and result is n = 100 and m = 10. In that case the Algorithm take an average of one minute in my CMD testing environment and an average of five minutes in the real puzzle game environment. There is such a big difference between those two environments since the CMD one is not using the sliding animation that the real one is using. In addition to that, with those values, there is an 80% chance of solving the puzzle and achieving 2048. After doing the simulations it calculates the average score by starting moves. Then the next move is the starting move that has the maximum average score. As an example, the AI perform a hundred games as a simulation for the given state and figured out that going left has an average score than right up and down. Hence, it should lead to a best score than sliding right, up or down, consequently, it the next move is left. In the same way, the AI a hundred games in the background (the RAM), then it figured out that by starting right, it is more likely for the puzzle to go further than the other three directions. Consequently, the AI swipe right. Here is the formula, the AI is using to make its decision:

$$dir = max(\ \bar{s} \cdot sim \cdot n \cdot \forall m), m \notin \emptyset, n \geq 2 \wedge n \in N$$
$$dir = 0 \Rightarrow dec = LEFT$$
$$dir = 1 \Rightarrow dec = RIGHT$$
$$dir = 2 \Rightarrow dec = UP$$
$$dir = 3 \Rightarrow dec = DOWN$$

Here, *dir.* is the outputted direction where *dir.* could only be 0, 1, 2 or 3. According to those values, the AI knows which decision to make and where to go, with s as the final score after the simulations, m the four possible moves, n the number of simulations, sim the simulation and `max()`, the maximum function.

You will find bellow the methods I used for my implementation of the Monte Carlo tree search algorithm.

The first method I have used is the main method I am using to solve the puzzle game. To solve the puzzle with the Monte Carlo tree search algorithm this is the only method that you have to call. This method will then call two other methods to help with some repetitive tasks.

```
Solve ()
  tempBoard <- int[4][4]

  left <- 1
  right <- 1
  up <- 1
  down <- 1

  leftScore <- 0
  rightScore <- 0
  upScore <- 0
  downScore <- 0
```

```
avrgLeftScore <- 0
avrgRightScore <- 0
avrgUpScore <- 0
avrgDownScore <- 0

tempScore <- score of the current state of the puzzle

WHILE the simulation is not over DO
  FOR i <- 0 TO number of simulations DO

    tempScore <- score of the current state of the puzzle
    tempBoard <- board of the current state of the puzzle

    randomPlay <- random(tempBoard)

    IF randomPlay[1] is 0 THEN
      left <- left + 1
      leftScore <- leftScore + randomPlay[0]
      ELSE IF randomPlay[1] is 1 THEN
        right <- right + 1
        rightScore <- rightScore + randomPlay[0]
        ELSE IF randomPlay[1] is 2 THEN
          up <- up + 1
          upScore <- upScore + randomPlay[0]
          ELSE IF randomPlay[1] is 3 THEN
            down <- down + 1
            downScore <- downScore + randomPlay[0]
          ENDIF
        ENDIF
      ENDIF
    ENDIF

 set the score of the current state of the puzzle to tempScore

  avrgLeftScore <- leftScore / left
  avrgRightScore <- rightScore / right
  avrgUpScore <- upScore / up
  avrgDownScore <- downScore /down

  ENDFOR

leftScore <- 0
rightScore <- 0
upScore <- 0
downScore <- 0

left <- 0
right <- 0
up <- 0
down  0

IF max(avrgLeftScore, avrgRightScore,
avrgUpScore, avrgDownScore) is "LEFT"
  slide left
```

```
    ELSEIF max(avrgLeftScore, avrgRightScore,
    avrgUpScore,avrgDownScore) is "RIGHT"
      slide right
      ELSEIF max(avrgLeftScore, avrgRightScore,
      avrgUpScore, avrgDownScore) is "UP"
        slide up
        ELSEIF max(avrgLeftScore, avrgRightScore,
        avrgUpScore, avrgDownScore) is "DOWN"
            slide down
        ENDIF
      ENDIF
    ENDIF
  ENDIF

ENDWHILE
```

Let me explain in detail what this method does. The first lines until the while loop are to setup and reset all the variables I am using for this method. Then, the while loop is here to repeat this whole process until the puzzle is solve or Game Over. The nested for loop is here because we want to repeat the whole process until the AI has done enough simulations. Then, `tempScore` and `tempBoard` are used to store the current values of the score and of the board so I easily reset everything. After that, the AI call and store the values of the method `Random(board[][])` that return an array with two values stored in it, the first value `randomPlay[0]` is the first moves done by the simulations while `randomPlay[1]`is the score at the end of the simulation. Afterward the AI use a bunch of if statements to count the number of moves that has been made to the directions and ad the score to the total score of that given direction. Then the AI reset the score of the simulation using `tempScore` after that, the AI update the average score according to the starting move. That is all for the nested for loop. After that for loop, the AI resets the values used during the for loop so it can be reused for the next state of the game. Finally, it finds the maximum average score according to the starting move amongst the four average scores using the `max(a, b, c, d)`method and slide to the direction that could leads to the highest score. Then the AI start again until the puzzle is solved and end the while loop if it’s the case.

As mentioned earlier, the `solve()` method used before is using two methods to perform some tasks. One of them is the `Random(board[][])` which do some simulations. This method plays random games in the “background” (the RAM of the computer) and return the first move of the simulation as well as the score of the simulation.

```
Random(board[][])
  moves <- {"LEFT", "RIGHT", "UP", "DOWN"}

  random <- random value between 0 and 3
  move <- moves[random]
  firstmove <- random

  IF move is "LEFT" THEN
    boad[][] swipe left
    ELSE IF move is "RIGHT" THEN
      boad[][] swipe right
      ELSE IF move is "UP" THEN
        boad[][] swipe up
        ELSE IF move is "DOWN" THEN
          boad[][] swipe down
```

```
        ENDIF
      ENDIF
    ENDIF
  ENDIF

  FOR 1 to number of moves the simulation has to do DO
    random <- random value between 0 and 3
    move <- moves[random]

    IF move is "LEFT" THEN
      boad[][] swipe left
      ELSE IF move is "RIGHT" THEN
        boad[][] swipe right
        ELSE IF move is "UP" THEN
          boad[][] swipe up
          ELSE IF move is "DOWN" THEN
            boad[][] swipe down
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDFOR

  scoreMove[0] <- score of the board
  scoreMove[1] <- firstmove

RETURN scoreMove
```

Let me explain line by line what this method does. The first line is an array that contains all the possible directions the simulation can take. Then, the next three lines are about making the first move randomly and keeping track of it. After that, there are a bunch of if statements that swipe to the correct direction using the first move used before. After dealing with the first move, AI has to do multiple moves. Consequently, the method is using a for loop that repeats itself for every moves the simulation has to do. Inside this for loop, we can find, two parts. The first one is the selection part with the two first lines of the loop that select which random moves needed to be done. Then, the second part is the same bunch of if statements I have been using before that swipe to the correct direction according to the random move selected the lines above. Then the AI start again until all the wanted move of the simulation are done and end the for loop if it’s the case. Finally, this method returns an array with two values in it, the first one is the final score of the simulation and the first movie done by it later.

As mentioned before, the `solve()`method used before is using two methods to perform some tasks. The second one is `max(a, b, c, d)` that output the direction of the maximum average given scores. This method comes has follow:

```
max(a, b, c, d)

max <- maximum value between a, b, c and d

  IF max is a
    RETURN "LEFT"
    ELSEIF max is b
      RETURN "RIGHT"
```

```
      ELSEIF max is c
        RETURN "UP"
        ELSEIF max is d
        RETURN "DOWN"
        ENDIF
      ENDIF
    ENDIF
  ENDIF
```

Let me explain line by line what this is doing. The first line finds the maximum value between the four given one. Then the AI is using nested if statements to return the corresponding direction according to the maximum average score.

3.2.2 Minimax

The second algorithm I have been using for my AI is the Minimax algorithm which is an algorithm that tries to minimise the maximum loose. This is done by looking ahead at possible states before deciding which decision should be taken. This algorithm assumes that the user is playing against the computer and that the user needs to do “the best move” while the computer is doing “the worst movie” This algorithm uses a tree that hosts every state of the puzzle without taking care of the probabilities. The size of this tree is limited since we do not want it to be massive and host all the states of the puzzle of the game we are solving. In the case of Gabriele Cirulli’s puzzle, the tree is taking the current state of the game as the root. Then the four different moves of this puzzle are the children of the root node. Finally, since we are not taking probability into account, each of those four nodes has two children for each blank square on the grid. One for the value 2 and one for the value 4.

Then, we need to perform a static evaluation using heuristics values to evaluate which state is the best and which state is the worse and assign to each leaf of the tree a value. The heuristic value of non-leaves nodes is determined by looking at the value of its children. If those children are not leaves nodes, the heuristic value of those nodes is determined by looking at the value of its children and so on until we reach leaves see Figure 3.5. The best move for the user is then the move that minimises the loose while the best move for the computer is the move that maximises the loose of the user.

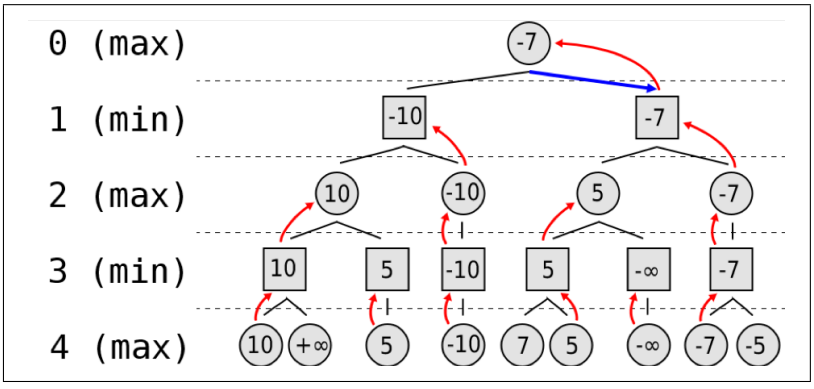

Figure 3.5: Example of a tree using the Minimax algorithm.

Here is the formulas the minimax algorithm using:

$$minimax(p) = staticEvaluation(p) \text{ if } p \text{ is a leaf}$$
$$minimax(p) = max(minimax(O1), \ldots, minimax(On)) \text{ if } p \text{ is not a leaf and it is not the Ais turn}$$

minimax(p) = min(minimax(O1), . . . , minimax(On)) if p is not a leaf and it is the AIs turn

with Onth child of p

The static evaluation is the core of this algorithm. The result of this algorithm depends on the static evaluation and the heuristic value given to the leaves of the tree. Hence, if the static evaluation gives a non-relevant heuristic value by doing an incorrect evaluation, the puzzle or the game could not be solved or result in a win. Some examples of static evaluation are that we can say that a state is better if this state has more blank space on the grid. Another one is that a state is better if the sum of all the element on the grid is higher. In addition to that, a state is better if the squares on the grid are well organised.

To make sure the grid is well organised, we use what we call weighted matrices. Those matrices are matrices that stated which square is more important on the grid by giving them value (weight). Consequently, the highest value of the grid should be on the scare with the highest weight. Among all the weighted matrices you can find about 2048 the two most famous weighted matrices that worked with Minimax are the snake-shaped matrix see Figure 3.6 and the corner shaped matrix see Figure 3.7. The snake-shaped matrix is a matrix that focuses on the fact that merging numbers will be easier since the two digits that you want to merge are next to each other. In addition to that, they should be next to the next digit they want to be merged to while the corner shaped matrix is a matrix that focuses on having the highest number in the corner so the board is not a big mess. Hence the smallest value should be in the opposite corner. One on the advantage of this matrix is that we can easily control where the new digit is going to “spawn”. It should be in the opposite corner where all the small values are.

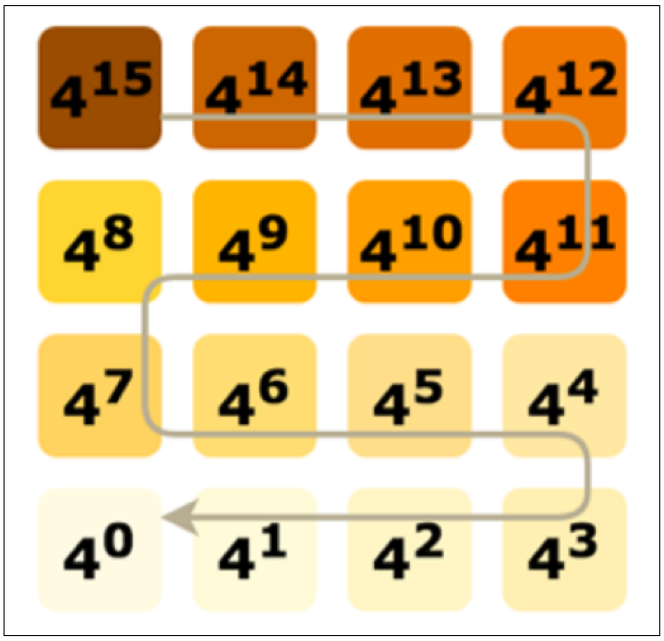

Figure 3.6: Snake-shaped weighted matrix.

Figure 3.7: Corner-shaped weighted matrix.

My implementation of minimax is then using the corner shaped matrix as a heuristic. Hence, my implementation comes in two classes. The first class is the class that made the Non-Binary tree the algorithm is going to use. In order to do so, I am using a custom object node for the nodes of the tree and array-lists to store the children of the nodes. The second class is the class that solves the puzzle and use the minimax algorithm as long as three other methods that help to do simple tasks. The first method I have used is the main method I am using to solve the puzzle game.

In order to solve the puzzle with the Minimax algorithm, this is the only method that you have to call. This method will then call one other method which calls some other methods to help with some repetitive tasks. This method comes as follow:

```
Solve()

  WHILE the puzzle is not solve DO

    tempBoard <- int[4][4]
    tempBoard <- board of the current state of the puzzle

    root <- int[4][4]
    root <- tempBoard

    rootNode <- node with root as data

    childLeft <- int[4][4]
    swipe left
    childLeft <- board of the current state of the puzzle
    childLeftNode <- node with childLeft as data
    set childLeftNode as a child of rootNode

    FOR i <- 0 TO length of the current board DO
      FOR j <- 0 TO length of the current board DO

        IF position (i, j) of the current board is 0 THEN
```

```
      position (i, j) of the current board <- 2
      set a new node with the current board as a child of childLeftNode
      position (i, j) of the current board <- 4
      set a new node with the current board as a child of childLeftNode
      position (i, j) of the current board <- 0
    ENDIF
  ENDFOR
ENDFOR

board of the current state of the puzzle  tempBoard

childRight <- int[4][4]
swipe right
childRight <- board of the current state of the puzzle
childRightNode <- node with childRight as data
set childRightNode as a child of rootNode

FOR i <- 0 TO length of the current board DO
  FOR j <- 0 TO length of the current board DO

    IF position (i, j) of the current board is 0 THEN
      position (i, j) of the current board <- 2
      set a new node with the current board as a child of childRightNode
      position (i, j) of the current board <- 4
      set a new node with the current board as a child of childRightNode
      position (i, j) of the current board <- 0
    ENDIF

  ENDFOR
ENDFOR

board of the current state of the puzzle <- tempBoard


childUp <- int[4][4]
swipe up
childUp <- board of the current state of the puzzle
childUpNode <- node with childUp as data
set childUpNode as a child of rootNode

FOR i <- 0 TO length of the current board DO
  FOR j <- 0 TO length of the current board DO

    IF position (i, j) of the current board is 0 THEN
      position (i, j) of the current board <- 2
      set a new node with the current board as a child of childUpNode
      position (i, j) of the current board <- 4
      set a new node with the current board as a child of childUpNode
      position (i, j) of the current board <- 0
    ENDIF

  ENDFOR
ENDFOR
```

```
    board of the current state of the puzzle <- tempBoard


    childDown <- int[4][4]
    swipe down
    childDown <- board of the current state of the puzzle
    childDownNode <- node with childDown as data
    set childDownNode as a child of rootNode

    FOR i <- 0 TO length of the current board DO
      FOR j <- 0 TO length of the current board DO

        IF position (i, j) of the current board is 0 THEN
          position (i, j) of the current board <- 2
          set a new node with the current board as a child of childDownNode
          position (i, j) of the current board <- 4
          set a new node with the current board as a child of childDownNode
          position (i, j) of the current board <- 0
        ENDIF

      ENDFOR
    ENDFOR

    board of the current state of the puzzle <- tempBoard

    moves <- {"LEFT", "RIGHT", "UP", "DOWN"}

    IF moves[readMinimax(rootNode)] is "LEFT"
      swipe left
      spawn a new digit
      ELSEIF moves[readMinimax(rootNode)] is "RIGHT"
        swipe right
        spawn a new digit
        ELSEIF moves[readMinimax(rootNode)] is "UP"
          swipe up
          spawn a new digit
          ELSEIF moves[readMinimax(rootNode)] is "DOWN"
            swipe down
            spawn a new digit
          ENDIF
        ENDIF
      ENDIF
    ENDIF

ENDWHILE
```

Let me explain line by line what the following method does. This whole method is surrounded by a while loop. This loop is here to ensure the action (move to solve the puzzle) is repeated until the puzzle is solved. After that, the two following lines are here to set up a temp board that will be used to reset the board later. Then the three next lines setup the root node of the tree which a node with the current state of the puzzle as the data of the tree.

The following part comes in four parts. One part for each direction. These four parts are all the same except that the sliding direction and the names changes. I will take the left

direction in my explanation. The first few lines are here to create and set up the nodes. The first line creates the board where we are going to store the data of the left child. Then, the second line slides/swipes left. The next line fills the board where we are going to store the data of the left child whit the data of the slid board. After that, the two next lines setup and create the node of the tree. It makes sure it has been linked to the root node. Then, we will find two nested for loops that allow the AI to iterate through every square of the puzzle board. These for loop contains an if statement that checks for every blank square of the grid. If it is the case, the AI then change the value of that square to a 2, create a node with that board, change again the value of that square to a 4, create a node with that board then to reset it, change again the value of that square to a 0. (0s are blanks squares on the grid). After that, it reset the board using the `tempBoard` value and do the same thing with the other three directions.

Finally, the next part is about using minimax and selecting which move is the best. Hence, we are using an array that store every direction the AI can possibly choose. After that, we are using a bunch of nested if statements and use `moves[readMinimax(rootNode)]`that output a string with the direction of where the AI should slide. Hence, the AI swipe to that direction and start this whole process again until the puzzle is solved.

As mentioned before, the `solve()` method used before is using a method called `readMinimax(node)`that read the minimax method. It comes as follow:

```
readMinimax(node])

  minimax <- minimax(node, 3, true)
  FOR every children IN node DO
    IF data of children is minimax THEN
      RETURN index of this children
    ENDIF
  ENDFOR
```

Let me explain what this method does. The first line calls the minimax method and stores the output. Then, for every child of the given state of the tree, it checks it the data correspond to what minimax is returning. Consequently, it returns the index of the corresponding child so the AI can easily know what the next move is going to be.

Hence, the minimax method comes as follow:

```
minimax(position, depth, maximizingPlayer)

  IF depth is 0 or position is a leaf THEN
    RETURN staticEvaluation(position)
  ENDIF

  IF maximizingPlayer is true THEN
    maxEval <- -Infinity

    FOR i <- 0 TO number of children of position DO
      child <- child of postion at the ith position
      eval <- minimax(chid, depth - 1, false)
      maxEval <- max(maxEval, eval)
    ENDFOR
```

```
    RETURN maxEval
  ENDIF

  IF maximizingPlayer is false THEN
    minEval <- +Infinity

    FOR i <- 0 TO number of children of position DO
      child <- child of postion at the ith position
      eval <- minimax(chid, depth - 1, true)
      minEval <- min(minEval, eval)
    ENDFOR

    RETURN minEval
  ENDIF
```

Let me explain what this algorithm does in detail. `minimax(position, depth, maximizingPlayer)` is using a tree to work. This algorithm takes three values as inputs, the current position of the tree, the depth of the tree and a Boolean representing which player has to play. This method is made with three if statements.

The first if statement states that if the depth of the tree is 0 or if the current position is a leaf, the AI should return the static evaluation of that position.

The second if statement states that if it is the user's turn (inputted Boolean is true) then, we set a variable to the lowest value and use a for loop that iterates through every child of the starting position. Hence, for every child of that position, the Algorithm returns the maximum value between the variable that we set earlier and the recursive call `minimax(chid, depth - 1, false)`.

The last if statement states that if it is not the user's turn (inputted Boolean is false) then, we set a variable to the highest value and use a for loop that iterates through every child of the starting position. Hence, for every child of that position, the Algorithm returns the minimum value between the variable that we set earlier and the recursive call `minimax(chid, depth - 1, false)`.

While finding the best possible move at a given state, `minimax(position, depth, maximizingPlayer)` is, calling a method called `staticEvaluation()` that output the heuristic value of a given state. It comes as follow:

```
staticEvaluation(position)
  staticEval <- 0
  wMatrix <- {{7, 6, 5, 4}, {6, 5, 4, 3}, {5, 4, 3, 2}, {4, 3, 2, 1}}
  sum[][] <- int[4][4]

  FOR i <- 0 TO size of wMatrix DO
    FOR j <- 0 TO size of wMatrix DO
      sum[i][j] <- wMatrix[i][j] * position.data[i][j]
    ENDFOR
  ENDFOR

  FOR i <- 0 TO size of wMatrix DO
    FOR j <- 0 TO size of wMatrix DO
      staticEval <- staticEval + sum[i][j]
    ENDFOR
  ENDFOR
```

```
  set the data of position to staticEval
  RETURN staticEval
```

This method returns the evaluation of a state of the game. Let me explain how this is done by using weighted matrices. The first line set a variable to 0. Then the second line creates the weighted matrix. Here the AI is using a corner shaped matrix. The third line is then, creating a matrix in which we are going to store values. After that, we have two nested for loops that go through every value of the matrices. For the first nested loop, every value of the sum matrix is the value of the weighted matrix at that same position times the value of the given state of the puzzle at that same position. While for the second nested loop, we store into the variable that we have set before, the sum of every value of the sum matrix. Finally, the AI return that sum of all the values in the sum matrix.

Finally, minimax is also using two methods `max(a, b)` and `min(a, b)` that respectively return the maximum value between a and b and the minimum value between a and b.

```
max(a, b)
  IF a is bigger than b THEN
    RETURN a
  ELSE
    RETURN board
  ENDIF

min(a, b)
  IF a is smaller than b THEN
    RETURN a
  ELSE
    RETURN b
  ENDIF
```

3.2.3 Comparison

As a part of my project, my AI needs to decide which algorithm is the best. Hence, this subcategory will threat this issue and find the best algorithm for my AI. The two algorithms I am using are the Monte Carlo Tree Search algorithm and the Minimax algorithm those two algorithms are good in their own manners, let me describe the positives and negative aspects of those two algorithms.

Firstly, one of the main advantages of the Monte Carlo Tree Search algorithm over the Minimax one is that even though those two algorithms are dealing with trees, the Monte Carlo Tree Search algorithm deals with a simpler and smaller tree. Indeed, the tree here is said to be an asymmetrical tree since it only keeps track of the best states of the puzzle and thus, do not care about all the other unnecessary states of the puzzles. In addition to that, this algorithm can be stopped at any time/states. This is not like the Minimax algorithm in which you need to specify the depth of the tree before creating it and/or explore it. Even though the Monte Carlo Tree Search algorithm have some advantages over the Minimax algorithm, it has some major drawbacks. All of them are due to the main risk of this puzzle which is randomness. Hence, due to randomness, we cannot ensure that the AI solves the puzzle at every attempt. As said before, a way to threat that risk is to increase the number of simulations and the depth of those simulations. Consequently, the negative aspect here is that for "better" result the AI need more times. Another drawback I have figured out within all the simulations I have done with this AI is that sometime the AI overthink too much and repeat itself. This can be seen right before a game over. The AI avoid doing the last move

because somehow it knows it will be a game over. It also happened at random states.

Secondly, the main advantages of the Minimax algorithm over the Monte Carlo Tree Search one is that it does not need to be “set up” by specifying the number of simulations like the Monte Carlo Tree Search algorithm before it starts. Hence, the time for each solves will always be the same. In addition to that, one main positive aspect of using minimax in context of the 2048 puzzle game is that it depends on the static evaluation of the given states. Consequently, this can be a double-edge sword since if the evaluation is amazing, the solving process can be done very fast however, if the evaluation is not performing well, the whole process will fail. As we have just seen the minimax algorithm have some negative aspects, most of them are related with time. Another one of them is that by using this algorithm, the AI need to visit every child of the tree twice, once to find the children and another one to output his heuristic value. This is due to that fact that with the Minimax algorithm you have to build the tree first then explore it while with my implementation of the Monte Carlo Tree Search algorithm, the AI build the tree as it goes. In addition to that, one disadvantages of the Minimax algorithm is that the deepest we explore/build the tree, the bigger the tree will be. As an example, with a small depth of 3 the tree has almost a hundred leaves at the start of the resolve and with a depth of 5, the tree has more than eight thousand leaves at the start of the resolve. Therefore, since the tree is bigger, the process is slower.

After seeing the advantages and the drawbacks of those two algorithms, we are going to see how we can evaluate them and figured out which one is the best. An algorithm is said to be better than another one if it is performing the task (here solving the puzzle) faster than it later. In addition to that another way to figured out which algorithm perform better is to see how “far” they can go. This means that, an algorithm performs better if his average score is higher than the other algorithm. To figure this out, I will use benchmarks and the time complexity of those two algorithms.

Consequently, the benchmark over ten runs of the Monte Carlo Tree Search algorithm that considers the averages scores that leads to deciding the best move comes as follow, see Figure 3.8:

	100 runs		500 runs		1000 runs	
#	final_score	top_tile	final_score	top_tile	final_score	top_tile
1	16556	1024	37128	2048	114768	8192
2	36892	2048	37132	2048	32968	2048
3	36836	2048	37208	2048	37088	2048
4	61504	4096	76268	4096	61560	4096
5	27732	1024	39656	2048	61496	4096
6	37064	2048	37248	2048	79440	4096
7	36604	2048	36944	2048	81496	4096
8	36812	2048	28180	2048	36984	2048
9	24040	2048	81588	4096	61656	4096
10	36876	2048	61640	4096	37068	2048
$\bar{x}$	**35091.6**	**2048.0**	**47299.2**	**2662.4**	**60452.4**	**3686.4**

Figure 3.8: Benchmark over ten runs of the Monte Carlo Tree Search algorithm.

Here, we can see that the more runs the AI is performing during simulations, the bigger the score and the tile with the highest value will be, as we can see with the average score and the and the average highest score at the bottom of the figure above. We can have a better look at this by looking at some graphs see Figure 3.9 and Figure 3.10 that come as follow:

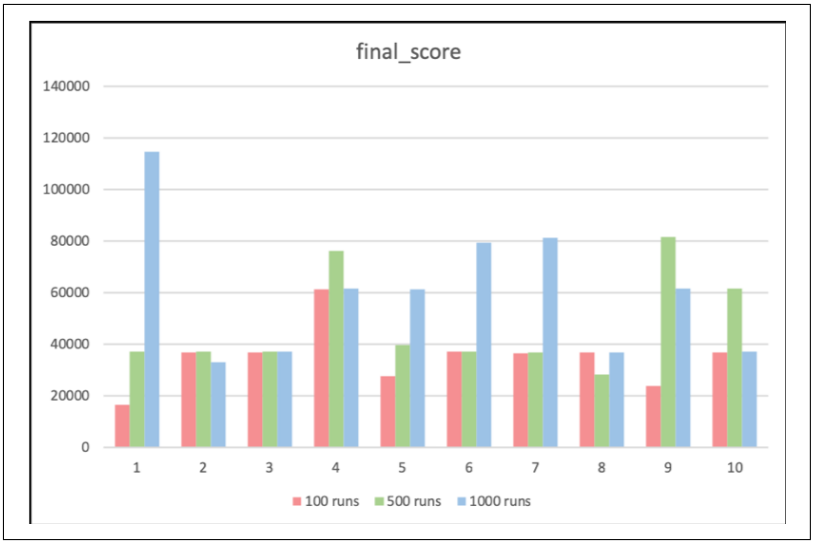

Figure 3.9: Graph representing final scores of different numbers of simulations.

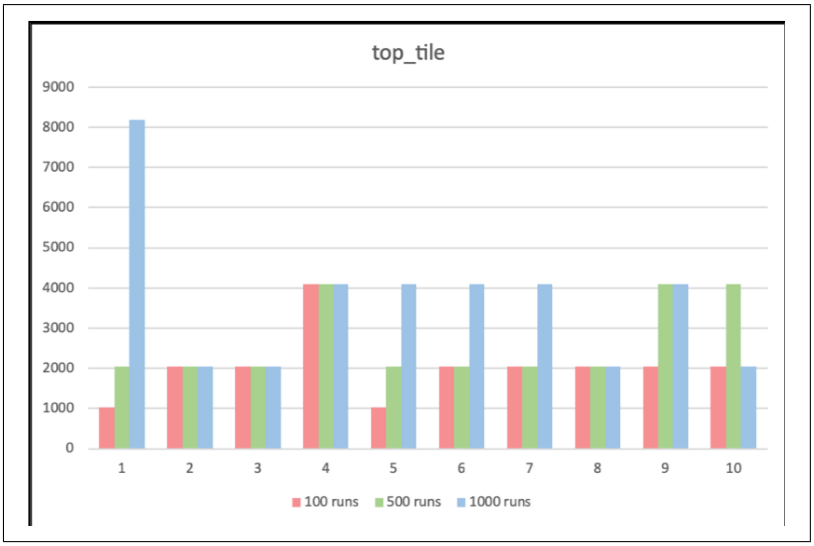

Figure 3.10: Graph representing maximum tiles of different number of simulations.

As we can see with the it is clear that the more simulations the AI is performing, the better the results are going to be. Therefore, we can see from the figures above that the maximum value achieved with the Monte Carlo Tree Search algorithm is 8192 with more than a thousand simulations. However, if we have a look at the benchmark with a hundred simulation, we can see that there is 80% chance of achieving 2048 and solving the puzzle.

By doing the benchmarks of the Monte Carlo Tree Search algorithm over the puzzle game, we have found the possible scores and the maximum tiles we can achieve with a number of simulations given. In order to complete the evaluation of this algorithm we need to have a look at its time complexity which comes as follow. The time complexity of the Monte Carlo Tree Search Algorithm is taking four factors into account since this algorithm is divided into four parts, the selection, the expansion, the simulation, and the back propagation. The algorithm repeats those steps n times, where n is the number of move we need to do before achieving the tile 2048 and solve the puzzle game. (For our calculations, we are going to assume that n is the same for the Monte Carlo Tree Search algorithm and the Minimax one) Hence, we have:

$$O(n*(selection + expansion + simulation + backpropagation))$$

However, my implementation of this algorithm is doing the expansion step at the same time as the simulation one. Hence, we have:

$$O(n*(selection + simulation + backpropagation))$$

Then, since the selection step is where the AI setup the tree and get ready for the expansion and the simulation step by building a tree, the time complexity becomes:

$$O(n*(b*d + simulation + backpropagation))$$

Where b is the branching factor of the starting tree and d is its depth. In other words, b is the number of leaves while d is the number of levels of the tree.

The backpropagation step is proportional to the depth of the tree since in order to do a backpropagation and find which move is the best, the AI need to have a look at the tree. Hence, we have:

$$O(n*(b*d + simulation + d))$$

Finally, the simulation part is separated in two parts, the number of times we are doing the iterations and the it takes to do a simulation. Since a simulation is in our case, the repletion of random moves, we have:

$$\begin{gathered} O(n*(b*d + m*n + d)) \\ O((b*d*n + m*n + d*n)) \\ max\ O(b*d*n),\ O(m*n),\ O(d*n) \\ O(n) \end{gathered}$$

Since the time complexity of an algorithm is the time complexity of its dominant term, its time complexity and thus, its big O notation is O(n).

Now that we have the benchmarks and the time complexity of the Monte Carlo Tree Search algorithm over the 2048 puzzle game, we are going to have a look at the same factor for the Minimax algorithm with the final scores as heuristic values. Its benchmarks see Figure 3.11 come as follow:

	Minimax		
Evaluation	score		
Depth	2	3	4
High Score	25592	34968	58732
Average Score	9346	17421	27250
Pruned	No	Yes	Yes
256	99%	100%	100%
512	87%	99%	100%
1024	46%	85%	98%
2048	2%	27%	71%
4096	-	-	4%

Figure 3.11: Benchmarks of the Minimax algorithm using the final score as a heuristic value.

As we can see here, the deeper the tree is, the better the results are going to be. However, as said before, the deeper, the tree is, the longer it will take for the AI to solve the puzzle. Let us now have a look at the benchmark with weighted matrices as heuristic values. The benchmark see Figure 3.12 comes as follow:

	Minimax		
Evaluation	Weighted matrix		
Depth	3	6	8
High Score	-	-	177352
Average Score	-	-	≈ 40000
Pruned	No	No	No
2048	76%	90%	100%
4096	-	-	80%

Figure 3.12: Benchmarks of the Minimax algorithm using a weighted matrix as a heuristic value.

As we can see here, the use of a different static evaluation mater since the weighted matrix as a heuristic give a better result than the score as a heuristic. In addition to that, we can confirm that the deeper the tree is, the better the results are going to be.

By doing the benchmarks of the Minimax algorithm over the puzzle game, we have found the possible scores and the maximum tiles we can achieve with a depth given. In order to complete the evaluation of this algorithm we need to have a look at its time complexity which comes as follow:

The time complexity of the Minimax algorithm is taking two factors into account since this algorithm is divided into two parts, the generation of the tree and the algorithm itself. The algorithm repeats those steps n times. Hence, we have:

$$O(n*(\textit{Tree generation} + \textit{Minimax}))$$

As seen earlier, the time complexity of the tree generation is the branching factor b, times the depths of the tree d. Hence, we have:

$$O(n*(b*d + Minimax))$$

The time complexity of Minimax is the use of two recursive calls that each deduce one to the depth and a constant. This is due to the fact that the algorithm is separated in three parts, two recursive calls and a base case. Therefore, we have:

$$O(n*(b*d + (2(d-1) + C)))$$
$$O(n*(b*d + 2(d-1)))$$
$$O(n*(b*d + 2*d-2))$$
$$O((b*d*n + 2*d*n - 2*n))$$
$$max\ O(b*d*n),\ O(2*d*n),\ O(-2*n)$$
$$O(n)$$

Since the time complexity of an algorithm is the time complexity of its dominant term, its time complexity and thus, its big O notation is O(n).

After seeing those results, we can not state witch algorithm is better since those two algorithms are better “in their own way”. Indeed, as we can see with the benchmarks, the Monte Carlo Tree Search algorithm gives in average best and more promising result while as we can see with the time complexity the Minimax algorithm is in an average faster since a quadratic speed is faster than a linear speed.

Chapter 4: **Software Engineering**

The fourth chapter is about Software engineering and especially how I have used the Revision Control System called GitHub and how I have done tests for my proofs of concept programs. In addition to that, over there, you will find a UML diagram that explains how my AI solver works and which design pattern I have used.

4.1 Revision Control System

The revision control system I have used for this project is GitHub. I have decided to use version control so I can keep track of all the changes I have made. In addition to that, by using version control, I was able to recall previous versions. For example, while I was working on the AI, I had an algorithm that give the AI the ability to achieve the value 256 while the next one could not get something higher. Therefore, I had to come back to the previous version and start working from there again.

The way I have worked with GitHub is that for every progress regarding proof of concepts programs, features or reports I have made, I have created a new branch to work on it. Once I was done with the work I was doing on this branch, I merged it back into master. By doing this I was sure to never directly commit into master. For every commit I was made sure that my comment was relevant so I can know what I was working on last time. In addition to that, GitHub was very useful for every time I was changing coding environment. Whether it was my personal laptop, the library computers or just a random laptop, by using GitHub, I was able to continue working on my project by cloning the branch I was working on. Consequently, if I needed to show the advancement of my project. All I have had to do was to clone the master branch.

4.2 Testing

The way I was testing my code depends on what I was working on. Testing a GUI is not the same as testing a non-GUI based program. For almost all my programs, I have used White Box testing.

For the first proof of concept program I have written, since it is a GUI, I have used a non-conventional way of testing it. The way I have tested it was that, for every feature I wanted to have if that specific feature did not show up on the screen and/or was not working properly, it failed the test and I have to fix it.

The second proof of concept program I have written was a data structure, the one about Non-Binary Tree. To test it, I have used Test Driven Development (TDD), Therefore, for this program, I have used Black Box Testing. This means that I have written the tests before the code.

Then, for the third proof of concept program, the way I have tested it is very similar to the way I have tested the first proof of concept program. The third proof of concept program is the playable 2048 in CMD. In the same way, I have made the GUI for the first program, I have made the GUI for this program. Then for every feature, since it is a puzzle game, I have it passed tests if that specific feature was working. As an example, once I have coded

the new digits system that displays randomly two values on the board if it did not have two values randomly placed on the 4x4 grid the test failed.

After that, for the fourth proof of concept program I have done, there was one goal. This goal was: solving the puzzle game by achieving 2048. So basically, the test failed if and only if the AI achieve game over before having 2048 on the 4x4 grid.

Finally, the last proof of concept program I have done was solving the puzzle using the minimax algorithm. Since this proof of concept program was using a different implementation than the proof of concept which was the second one about Non-Binary Tree. Consequently, I have had to do two different types of testing this last proof of concept. For the Non-Binary Tree, I have changed for implementation with arrays to implementation with array lists. This required some testing. Here the testing process was using Test Driven Development (TDD), Therefore,, for this program, I have used Black Box Testing. This means that I have written the tests before the code. All I have had to do is make sure the AI can create at Non-Binary tree. Once the Non-Binary tree proof of concept was working, I could start focussing on the actual proof of concept. Since there was one goal which was: solving the puzzle game by achieving 2048, the test failed if and only if the AI achieve game over before having 2048 on the 4x4 grid. All I have had to do is make sure the AI was using the minimax Algorithm.

After the proof of concepts programs, all I had to do testing for was the GUI and the classes that connect and interact with the actual puzzle.

For the GUI, I have used a non-conventional way of testing it. The way I have tested it was that, for every feature I wanted to have if that specific feature did not show up on the screen and/or was not working properly, it failed the test and I have to fix it while for the interaction classes I have used White Box testing. Indeed, if the output of the “screen scanner” wasn’t as expected, I had to change to fix something.

4.3 UML

The UML diagram of my final proof of concept program comes as follow:

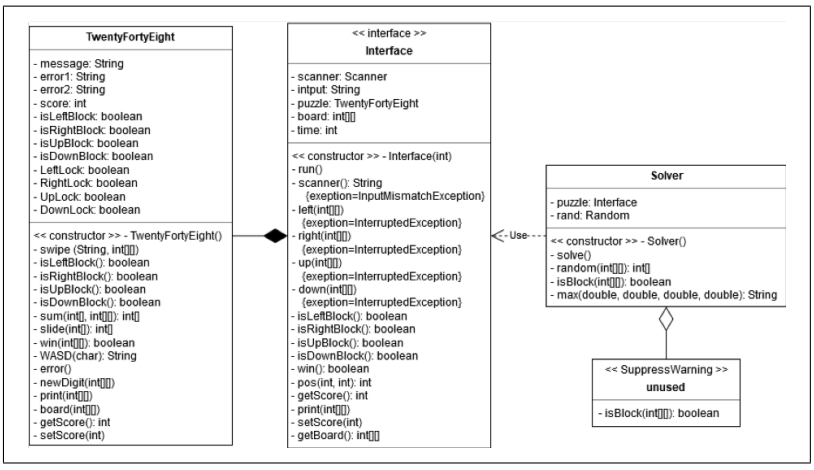

Figure 4.1: UML diagram of the proof of concept 4

This UML diagram see Figure 4.1 above is a simpler version of the next UML diagram however it is a good way of understanding the thoughts and the processes behind my project. Here we have a class called TwentyFortyEight that is basically a copy of the real 2048 puzzle game. Then we have a class called Interface which is just an interface design pattern for

the TwentyFortyEight class. Finally, there is a class called solver which is the core of this diagram since it is where you can find the algorithm the AI is using. In addition to that, since that class need to do some simulation, it can create some “background” puzzles and interact with them using the interface.

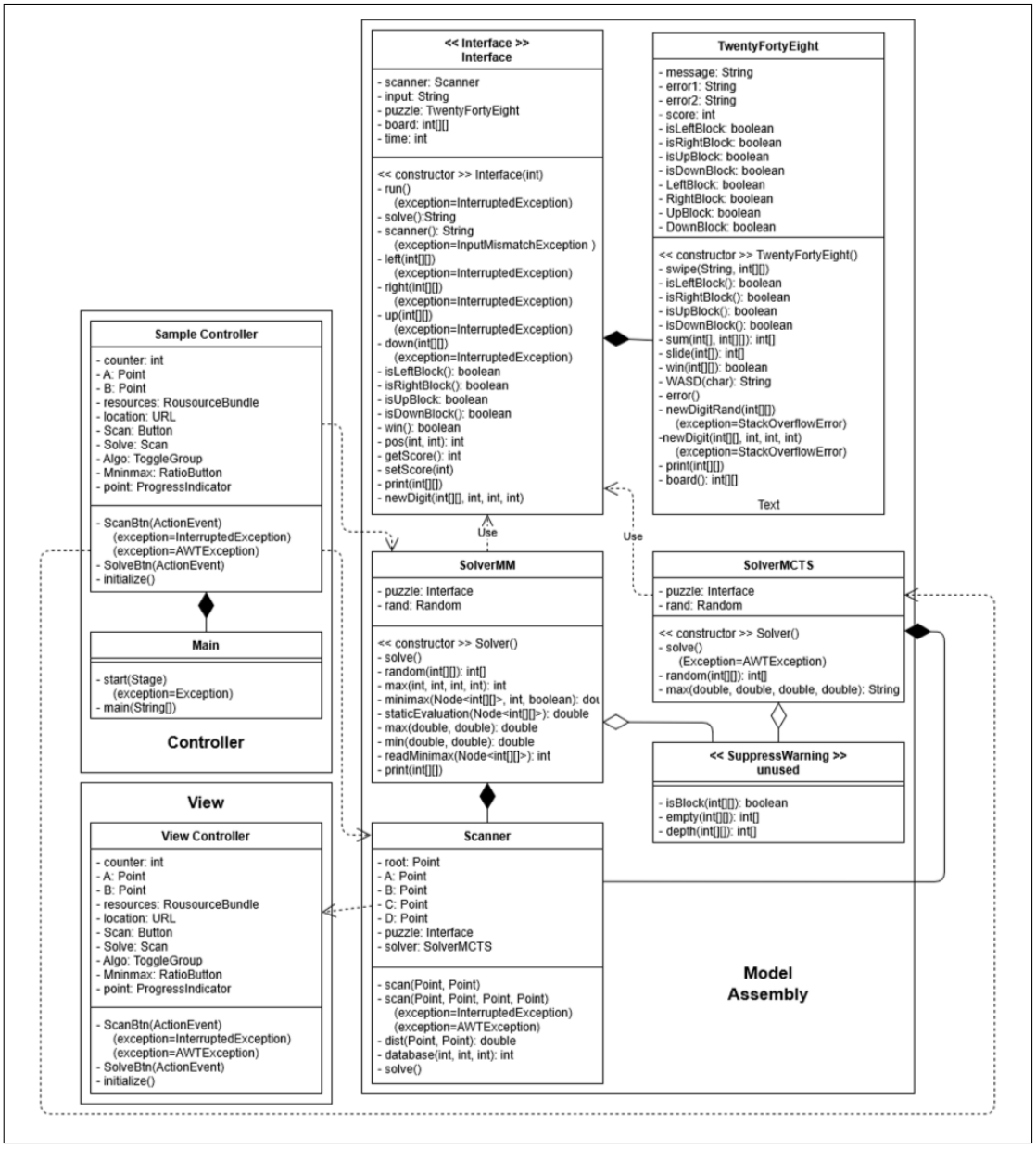

Figure 4.2: UML diagram of the whole project

This UML diagram see Figure 4.2 above represents how everything classes of my project are related to each other. This diagram is separated into three parts: the View, the Controller, and the Model Assembly.

Firstly, the view is the GUI where the user can interact with the AI that solves the 2048 puzzle game.

Secondly, the Controller is the “bridge” between the Model Assembly part and the View part of this project. Hence, the Controller part is made of two classes, the first one is the Main class that is the first thing the user will run to start the AI. Then the main class call the Sample Controller class that basically control everything. Indeed, this class take every input form the GUI and decide what to do with them. Here those inputs are which algorithm the

user wants the AI to use to solve the puzzle game and if the User is ready to scan the Puzzle grid, so it is easier for the AI to read the puzzle game. In addition to that, this class output some indication of how the scanning of the puzzle is performed.

Thirdly, the Model Assembly is where all the “magic” happened. In other words, all the complex algorithms the AI is using are in the classes of this part. This part is made of five classes. The first class is the class called TwentyFortyEight that is just a copy of the game that you can play in CMD. After that, I have made the Interface class as an Interface design pattern so it will be easier for the solver classes to interact with the game. Consequently, the two solver classes are SolveMCTS and SolveMM which are respectively the classes that solve the puzzle using the Monte Carlo Tree Search algorithm and the Minimax algorithm. Those two classes use the interface to create and interact with the puzzle game so they can do some simulations. Finally, the last class of this part is the scanner that is used to scan the screen to find the real puzzle game and interact with it and control it.

In addition to that, I have some suppress warning: unused that are basically all the tests methods I have made that did not help by the time but that could be handy later.

Chapter 5: **Professional Issues**

The fifth chapter is about all the professional issues I have faced during the process of making my AI solver. In addition to that this chapter is about the ethical implications of the professional issue of replacing Humans with technologies.

5.1 Encountered issue

While doing my project, a professional issue arose. This issue is a usability threat. It is a usability threat because my AI solver is replacing the user. Since I have made an AI that solves the 2048 puzzle game it can be easily transposed to solve firstly other puzzles, then bigger and more complexes problems. Hence, it could also be transposed to make important decisions.

I would not say that my AI is worse or better than a Human however I have figured out that, while I was working on this project, many people wanted to play the game. It means that, while seeing my AI performing, people wanted to challenge it and play better than it. Overall, some people were intrigued and wanted to beat it while others were afraid of it.

5.2 Ethical Importance

Even though replacing Humans with technologies such as AI or Robots have some benefit which is an increase in precision, completing tasks faster, the ability to work without interruption since AI and Robots do not need to sleep and go on vacations and AI and Robots do fewer mistakes, it leads to some ethical issues.

The first ethical issue is the fact that people are losing their job. This is only affecting the most automatable activities which are what we call predictable and/or repetitive work. Those works are in general, works that do not require either skills or specific knowledge. As an example, we have cashiers who are getting replaced by those manual automatic robot cashier. This is an ethical issue since someone is losing his job and have to either find a new one in better cases or to transferring carrier to another field of study.

Another more recent example that involves AI is the Tesla self-driving cars. The AI is replacing Taxis and Chauffeur Driven Vehicle. Here, despite the fact of people losing their job, the ethical issue is that if the AI makes some mistake (which could happen), the error will cost lives. In fact, sometimes, the car has to make some immoral decisions. As an example, there is the study called moral machine made by MIT students where you have a self-driving car that have only two options, either to choose to crash the car and kill the people in the car or to kill the people crossing the roads. In this study, the car has to choose which life is "more important" and save it. [16] Consequently, as said before, my AI have to make small decisions on what actions to do in order to solve the puzzle game 2048, however it can be transposed to one of those moral issues with the self-driving car in which the AI have to make a decisions on what actions to do in order to save lives.

The second ethical issue is the fact that replacing Humans with technologies such as AI or Robots leads to some inequality. Since some company are replacing some employees by AI or Robots, the employers will get the benefit of not having to pay the wage for the AIs.

Thus, the employers will get all the money and benefit of having AIs while the remaining employees won't have any advantages. Consequently, this will increase inequality since the wealth gap between employees and employers increase. Consequently, this is only beneficent for the riches and increase social differences.

The third ethical issue is the fact that replacing Humans with technologies such as AI or Robots leads to AI punishment. In fact, if the AI is doing something wrong like killing people in the previous example of the self- driving car or another illegal action like purchasing illegal drugs. Who should we blame and punish? Who should we send to jail? Should we punish the AI/Robot itself for killing people or should we punish the creator or programmer of it?

Some says that we should blame and punish the AI because well-made AI got the ability to think, to analyse situations and to learn from mistakes.

Some says that we cannot punish the AI because the AI is not human. In that case who should we blame? The obvious answer should be its creator since it is him that creates and teaches the AI/machine how to think and act. However, the since the creator and or programmer did not teach intentionally teach the AI how to kill or steal, we cannot blame him. In fact, the Geneva Conventions argue that “a commander is accountable for the crimes of a subordinate if the commander knowingly or negligently allowed the crime to occur.” In that case we cannot blame the creator if he did not wanted the AI to commit a crime.

Overall, I would say that in order to choose who we should blame, we should do it on a case-by-case basis.

Chapter 6: **Conclusion**

My project was about solving a puzzle using AI and especially the 2048 puzzle. Since 2048 is a puzzle that involves randomness, I had to use a non-conventional algorithm that tries to avoid the randomness issue. As a result of this, I have found that due to randomness there will never be a one hundred per cent chance for the AI to solve the puzzle. However, there is always a way to treat this risk and to increase the probability of success. In order to do so, I have used two different algorithms, the Monte Carlo Tree Search algorithm, and the Minimax algorithm.

In my opinion, solving the puzzle game with the help of an AI is doable. However, with conventional techniques and the fact that this puzzle involves randomness, achieving 2048 is not an easy task. After all, I have found that the main issue for AI is not randomness, it is hardware limitation. Obviously, the more simulations we are doing, or the bigger trees are, the longer it will be to do a move and thus to change the state of the game. Therefore, with better computers, I believe that this should be faster and solve the puzzle faster.

Overall, after doing all the experiment I have learn that doing a project like this is not simple and that you will always face some unexpected issues like randomness or simply coding issues. I have learned to manage my time, to setup goals and to deal with big projects. On a personal experienced point of view, this experiment helps me getting a foothold in the world of AIs and in solving complex puzzles. I have then improved my technics and my way of thinking regarding puzzle games.

The next step will be to try to transpose my work to the mobile app since I am currently solving the web version of this puzzle. Moreover, I would like to contact the creator of the puzzle game and figured out how exactly he made it. It means that I want to learn h ow the game is working, I mean in depts. Therefore, even thought, making an AI was very interesting for me, I especially want to learn how the randomness works in this game and in general. If I can know which method the creator of this game used to make the random part of this puzzle, I might be able to be better at predicting values in this puzzle and thus, to come out with my own algorithm.

If this puzzle game did not have randomness, it would have been easier for the AI to achieve 2048 or even go beyond that and be faster.

Bibliography

[1] MdRafiAkhtar (n.d.). Search Algorithms in AI - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/search-algorithms-in-ai/ [Accessed 5 Dec. 2019].

[2] Van Der Bok, K., Taouil, M., Afratis, P. and Sourdis, I., 2009, December. The tu delft sudoku solver on fpga. In 2009 International Conference on Field-Programmable Technology (pp. 526-529). IEEE.

[3] Schilling, M.F., 1994. Long run predictions. Math Horizons, 1(2), pp.10-12.

[4] Hoffman, E.J., Loessi, J.C. and Moore, R.C., 1969. Constructions for the solution of the m queens problem. Mathematics Magazine, 42(2), pp.66-72.

[5] Barták, R., 1999, June. Constraint programming: In pursuit of the holy grail. In Proceedings of the Week of Doctoral Students (WDS99) (Vol. 4, pp. 555-564). Prague: MatFyzPress.

[6] Stanford Junior Univercity (2008). YouTube. [online] Youtube.com. Available at: https://www.youtube.com/watch?v=p-gpaIGRCQI [Accessed 5 Dec. 2019].

[7] Mayam (1994). ExpectimaxSearch.

[8] Robin (2009). Brute Force Search. [online] Intelligence.worldofcomputing.net. Available at: http://intelligence.worldofcomputing.net/ai-search/brute-forcesearch.html. [Accessed 5 Dec. 2019].

[9] Cirulli, G. (n.d.). Gabriele Cirulli. [online] Gabrielecirulli.com. Available at: https://gabrielecirulli.com/ [Accessed 5 Dec. 2019].

[10] GitHub. (n.d.). gabrielecirulli/2048. [online] Available at: https://github.com/gabrielecirulli/2048 [Accessed 5 Dec. 2019].

[11] Caseyrule.com. (n.d.). Casey Rule. [online] Available at: http://www.caseyrule.com/projects/2048-ai [Accessed 5 Dec. 2019].

[12] 2048 game AI. (n.d.). 2048 game AI. [online] Available at: https://ronzil.github.io/2048-AI/ [Accessed 5 Dec. 2019].

[13] Rodgers, P. and Levine, J., 2014, August. An investigation into 2048 AI strategies. In 2014 IEEE Conference on Computational Intelligence and Games (pp. 1-2). IEEE.

[14] Neller, T.W., 2015. Pedagogical possibilities for the 2048 puzzle game. Journal of Computing Sciences in Colleges, 30(3), pp.38-46

[15] Delahaye, J.P., 2006. The science behind Sudoku. Scientific American, 294(6), pp.80-87.

[16] Awad, E., Dsouza, S. and Chang, P. (2020). Moral Machine. [online] Moral Machine. Available at: http://moralmachine.mit.edu/ [Accessed 28 Feb. 2020].

[17] Vlasic, B. and Boudette, N.E., 2016. 'Self-Driving Tesla Was Involved in Fatal Crash,'US Says. New York Times, 302016.

[18] Heilweil, R. (2020). Tesla needs to fix its deadly Autopilot problem. [online] Vox. Available at: https://www.vox.com/recode/2020/2/26/21154502/tesla-autopilot-fatal-crashes [Accessed 28 Feb. 2020].

[19] Bossmann, J. (2016). Top 9 ethical issues in artificial intelligence. [online] World Economic Forum. Available at: https://www.weforum.org/agenda/2016/10/top-10-ethical-issues-in-artificial-intelligence/ [Accessed 28 Feb. 2020].

[20] Sarch, A. and Abbott, R., 2019. Punishing Artificial Intelligence: Legal Fiction or Science Fiction.

[21] Nie, Y., Hou, W. and An, Y., 2016. AI Plays 2048. [online] stanford. Available at: http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf [Accessed 3 April 2020].

[22] Stack Overflow, 2016. What is the optimal algorithm for the game 2048?. Available at: https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22498940 [Accessed 3 April 2020].

[23] Surma, G., 2017. 2048 - Solving 2048 With AI. [online] Medium. Available at: https://towardsdatascience.com/2048-solving-2048-with-monte-carlo-tree-search-ai-2dbe76894bab [Accessed 3 April 2020].

[24] Stack Overflow, 2017. Complexity of the min-max algorithm. Available at: https://stackoverflow.com/questions/45658879/complexity-of-the-min-max-algorithm [Accessed 3 April 2020].

[25] StackExchange, 2016. What's the time complexity of Monte Carlo Tree Search?. Available at: https://cs.stackexchange.com/questions/51726/whats-the-time-complexity-of-monte-carlo-tree-search [Accessed 3 April 2020].

[26] Willem, M., 1997. Minimax Theorems. Boston: Birkhäuser, vol. 24.

[27] Noa Yarasca, E. and Nguyen, K., 2018. Comparison of Expectimax and Monte Carlo algorithms in Solving the online 2048 game. Pesquimat, 21(1), p.1.

[28] Coulom, R., 2006. Efficient Selectivity And Backup Operators In Monte-Carlo Tree Search. Springer, pp.72-86.

[29] D. Scheffer, B. Dierickx and G. Meynants, ”Random addressable 2048/spl times/2048 active pixel image sensor,” in IEEE Transactions on Electron Devices, vol. 44, no. 10, pp. 1716-1720, Oct. 1997.

[30] Donald, M., 1966. Game-Playing And Game-Learning Automata. Elsevier, pp.183-200.