

Final Year Project - Final Report

Full Unit – Final Report

Using AI to solve 2048

Rayan Miah

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Yunkuen Cheung



Department of Computer Science
Royal Holloway, University of London

March 23, 2023

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 13271

Student Name: Rayan Miah

Date of Submission: 23/03/2023

Signature:

A handwritten signature in black ink, appearing to be 'Rayan Miah', written in a cursive style.

Table of Contents

Chapter 1: Rationale	3
1.1 Report Structure.....	3
1.2 Aims and objectives	4
Chapter 2: Literature Review & Background Reading	5
2.1 Literature survey	5
2.2 Background Reading	5
Chapter 3: Professional Issues.....	6
Chapter 4: Planning and time-scale	8
Chapter 5: Summary of work & practical/theoretical aspects of the code	9
5.1 Summary of completed work	9
5.2 Practical aspects of the code.....	9
5.3 Theoretical aspects of the code.....	18
5.4 Technical decisions	22
5.5 Performance of the AI including data	24
Chapter 6: Diary	26
Chapter 7: Running the program	30
7.1 Submission directory structure	30
7.2 Package installation.....	30
7.3 Starting the program	31
7.4 Code deployment.....	31
Bibliography.....	36

Chapter 1: Rationale

1.1 Report Structure

This document aims to talk about my final year project and what I have done for it so far. As well as this, it will have several other sections delving deeper into the project as a whole. These sections include:

- A section motivating the project and the original project aims, including a description of how the work involved in my project will help me in my future career.
- Description and critical analysis of relevant background material from books, research papers or the web and an analysis of existing systems that solve similar tasks.
- A section discussing professional issues in computer science with respect to my project.
- My original draft and time-scale for what I was planning to do with the time I had for the project.
- Summary of completed work with a self-evaluation for how the project went.
- Section discussing my actual achievements during the project and how successful it was. Will include things that I may have tried but had difficulty with or deemed unnecessary.
- Section going over technical decisions I made and why I went with them/against them.
- Section going over the overall performance of the AI with data examples I gathered.
- Section explaining the relevant theory for my project.
- My diary that shows what I did on what days in detail.
- Section explaining how to run the software and what packages you need to install.
- Section explaining the submission directory structure.
- Screenshots of my program being run and a video showcasing it.
- Bibliography and citations.

1.2 Aims and objectives

In my project I had several aims and objectives. To begin with, I wanted the user to be able to play a game of 2048 themselves without the assistance of AI. This would be a simple mode with a high score system so the user can play the game when they want to. The second and more focused mode would be the 'AI plays' mode. This mode would allow the AI to play the game and the user would be able to watch as the AI attempts to get the highest score possible. The AI would achieve this by using certain AI algorithms such as Expectimax. This would allow the AI to look 4-8 moves ahead while calculating the chance for each move to occur and selecting the best move based on a scoring system (more details in the theory and code sections). I also wanted to make a way to automatically gather data on the AI, and how well it performed, to use in my final report for concrete evidence on any claims I made.

I would have also liked to include ways to make my project more unique from the others who have done it before me. This may include ways to change the game such as having a different grid size that is not four by four or having a different shaped grid such as a hexagonal 2048. These were a couple of the ideas I originally had to make the project more unique, but I unfortunately did not manage to implement these into my project. I go into more detail of the work I did complete and why I did not complete others in the summary of completed work section.

In terms of more general accomplishments I wished to achieve, I aimed to use this project as a means to learn how to manage my time better, as I can procrastinate often. I also wanted to use this project as a steppingstone into learning specific AI techniques and how to use them in a game environment, as this is something that I have a personal interest in. This project would also help my general understanding of AI, and how it links to the other AI related courses I took part in such as AI agents and machine learning. I also received advice related to this, saying that it may have been possible to use machine learning algorithms to better optimise my heuristic weightings and find the best combination for them. This is something I would have liked to add but was only if I had a lot of spare time on my hands, as it would take quite a bit of research to learn how to implement it into my AI.

This project will help me in many ways in the future of my career, one of the main reasons is that it will give me insight into the AI field and how to use different AI algorithms to improve certain projects. It also has given me a lot of experience on managing a big project as a whole, learning to manage my time and ideas, as to not go overboard with the resources I have.

Chapter 2: Literature Review & Background Reading

2.1 Literature survey

In my literature survey I will be writing about how different people have tackled the problem of solving 2048 and what they have found to work the best. Many methods have been found to work on the game including alpha-beta search, Expectimax or the usual method of searching when it comes to single player stochastic games. In the paper from reference [9], there is talk of another paper published by Szubert and Jaskowski (see reference [12]) and their idea of playing the game using temporal difference learning with n-tuple networks. This method proved to be a massive success reaching a win rate of 97%, an average of 100,178 and a maximum tile of 261,526. These scores are massive compared to scores I have seen in other papers. For instance, let us take the paper published by Yarasca and Nguyen which is about the comparison of the Expectimax and Monte Carlo methods of solving 2048 (see reference [7]). As seen in the paper Expectimax does perform pretty well, reaching the 2048 tile even at a depth search of two. The average score of the Monte Carlo method however, reached max tiles of 4096 almost exclusively, this method however takes a lot more time on average making it not a very user friendly watching experience. As seen neither of these methods can compare to the scores reached by the multi-stage temporal difference learning however there is no mention of the time it takes for this method to work. This method learns and as such needs training games for the learning to take place. It is stated in the paper that 5 million training games were run for each stage and a sample of the scores were taken every 1000 games. If you want the user to be able to watch the game play itself at a fast speed Expectimax seems to be the solution with its low move times. It can also be seen to perform well the deeper the searches are, and this can be increased to numbers such as six and eight if optimisation methods are included. The optimisation methods I have seen the most are the use of transposition tables with implementations such as Z-hashing, you could also fork the processes and use multi cores to speed up the process at higher depth searches.

2.2 Background Reading

For my project, the vast majority of the background reading I did was looking for different ways to improve the AI's score with heuristics and optimisations. For example, when looking at other people's implementations of the 2048 AI, they had versions that could look at depths of 7+. After further digging on their codes, I realised that they were able to do this because of several things. One of the main things that a lot of others had success with was the speed of the AI's search, this was mainly due to the fact that they had used languages that used compilers compared to Python's interpreter. This results in their AI's being more than 10x faster than python's version and thus lets them search much deeper. However, there may have been a way around this even on python. One person's code (see reference 13) encoded the entire board as a single 64-bit integer where each of the tiles were 4-bit chunks. This allowed them to pass the entire board state around on a single machine register (assuming the machine is 64-bit). The problem of speed was something I should have been prepared for since after some reading I knew that Expectimax could not really be optimised or pruned as well as other AI algorithms.

Chapter 3: Professional Issues

In this section I will be going over certain professional issues that are relevant to my 2048 AI project. I am going to go over plagiarism (specifically using other people's code) and will also touch on AI that can be used to produce work that is not your own (ChatGPT).

Firstly, I am going to go over ChatGPT. ChatGPT is an AI that has risen in popularity in the past few months, it is an AI that is meant to generate human-like responses to a wide range of queries and prompts. The reason ChatGPT is an issue in the professional world is because it can generate answers to a very wide range of problems that you may come across during work or education. Of course, when students use this tool they will be learning from it, but the problem arises when the student asks the AI for specific answers or asks it to write paragraphs for them. This is an obvious problem because the work produced is not from the student themselves. Another glaring issue that arises with the AI is that because the responses are very human-like and not taken word for word from its database, your typical plagiarism checker would not be able to recognise it as an act of plagiarism. However, this problem is slowly being worked around as big websites such as Turnitin are slowly implementing tools into their checkers which can detect if an AI has been used on the work. The AI is also capable of writing code, although not perfect, which can greatly aid computer science students with their work, but again, if they use it excessively they may run into problems of plagiarism. This has concerned me during my project because ChatGPT is a tool that I use occasionally, so I have to make sure to be very careful with it, especially if I am using it with relation to university work. A general rule that I have come up with for the tool, is that a person should use it similarly to google, where you can ask it general questions about topics since it will give answers back in a much more understandable way than google ever could. If you stick to this rule, you can be free of worry of not producing your own work and simply use the tool as a way to learn new things. In terms of the ethical problems relating to using code from an AI, I think AI once again is quite controversial. The AI is not an individual person so you are not really hurting its livelihood by stealing its work, but you can also argue that the AI comes to the conclusions that it makes because it pulls data from hundreds of thousands of people. The question is that when the AI learns from all these different people and gives its own answers, are you really stealing from all of the original people who gave the AI these ideas?

Next, I am going to go over plagiarism, specifically in relation to using other people's code with acknowledgement. Using other people's code is a very slippery slope, especially in the environment of this project, where it is allowed but you obviously need the majority of your code to be written by you. Were this issue not properly addressed, then workers and students would be able to use other people's code at leisure with no worry about consequences, this could lead to many people who claim to be the original authors of certain code, when in reality they had no part in its production. In the work/education environment, this can eventually lead to people getting accepted, when they are actually underqualified for the job or course. This in turn could have massive consequences depending on what sort of job they went for. Once again, this is a very tempting act to undergo, since the internet is filled with examples of what you are trying to do but done much better by someone else. When researching on this project, I too came across many others who had already undertaken this particular project and had perfected their AIs to reach massive scores. I made sure that any code I did use was credited to the original author and was also very clear in the source code where it was used. An idea I had with using other people's code was to identify how important what you are researching is to the core concept of your project, and if it is something that is not integral to the idea behind your code, then feel free to use other people's code while making sure to acknowledge them of course. For example, in my project I used someone else's code for the menu system GUI, as this is something that I did not deem necessary to waste hours on when there was already a very efficient way of making that menu. Of course, when I did this I made sure to document that I used that other person's code in this report, made it clear in the code and also gave the original link of the code in the bibliography. In terms of the ethical problems, when it comes to using other people's code, I think it is quite clear that it is very

unethical, especially if it a private piece of software as it could hurt the original authors livelihood. For instance, if the code was of a tutorial or maybe a blog post where people gave answers, I like to believe this is acceptable as they put the code out for others to use and study. However, if you took someone else's code for something they made privately (like a game or a project) and they had no intention of sharing, I believe that this is quite akin to an act such as stealing someone else's artwork and passing it off as your own. Going back to my own project, this was something I was slightly concerned about. This was because when I was in the testing stage for my AI and I saw that I needed improvements in certain areas, I would often look to other people's code to get ideas. If the code was a small heuristic for example, this was not something I could easily modify, as it was just a few lines, to justify calling it my own code and therefore the author had to be credited in which I worried that maybe I was using too much code that was not my own.

Chapter 4: Planning and time-scale

Since the project is of large scale, careful consideration and planning had to be taken before starting. In my project plan I had made a timeline of how I should be spending my time on a week by week basis and what I should be doing during those weeks. Unfortunately, certain circumstances prevented me from following this plan word for word. This however did not affect the project overall as I was still able to complete what I had planned for myself in a shorter time frame. This shows that I had given myself more time than was needed in my plan and I could have prepared the plan in a stricter fashion albeit the circumstances having me fall behind. Instead, I have created a new table showing what my time-scale looked like in reality (see figure 2) compared to my old one shown in the project plan (see figure 1).

Week 1:	Work on Menu GUI for the proof of concept program.
Week 2-3:	Work on the base game of 2048 for the proof of concept.
Week 4:	Study AI algorithms and their implementations into the game.
Week 5-6:	Work on the proof of concept program where the AI plays a pre-determined version of the game.
Week 7:	Study optimisation techniques and apply them in the proof of concept.
Week 8:	Write report on different AI algorithms and their uses.
Week 9:	Attempt to make the hexagonal 2048 proof of concept program.
Week 10-11:	Prepare for interim report and presentation.

Figure 1. Old timeline as shown in project plan.

As seen above in figure 1, I did not actually follow this plan perfectly. For example, in the weeks of 5 and 6, I was meant to make a program that plays a pre-determined version of the game. This was scrapped as I ended up completing the AI in a shorter time than expected and the AI that I had made worked on the original game itself, therefore removing the need to make a test version on a simplified game.

Figure 2. New table showing how time was actually spent.

Week 1:	Worked on Menu GUI for the proof of concept program.
Week 2:	Worked on the base game of 2048 for the proof of concept.
Week 3-4:	Very busy couple of weeks, did not get much work done other than a few fixes to the code.
Week 5:	Studied AI algorithms such as Expectimax and how to implement it into the code.
Week 6:	Implemented the Expectimax algorithm and changed the game to work in OOP as it originally did not do so.
Week 7:	Worked on small Expectimax changes and a GUI for the game.
Week 8:	Report writing and final code adjustments.

Chapter 5: Summary of work & practical/theoretical aspects of the code

5.1 Summary of completed work

In terms of what I have actually achieved for this project, I have done many of the things I set out to do. This includes the main concerns and goals I planned for: a version of the game where the player can play themselves. This mode lets the user play the original 2048 game themselves, although this was not the main focus of my project I wanted to include this as it added a few options to the project as a whole. My version of this however, is very bare bones and does not allow for high scores to be collected or include different difficulties or renditions of the game. It does however come with a feature that allows the user to play themselves while the AI version runs in the background. Another main and most important goal of the project was the AI version of the game. This version allows the user to watch the AI play the game and achieve scores at different speeds. This was not a feature I originally planned to add but with the differing speeds of the depths that the AI could go, I thought this would be a nice feature if the user would like to see the AI go much faster at a lower depth. The main issue I had with the AI was figuring out how to speed up the search and thus allow the AI to look further into the future. I came up with a few different ideas to help with this. The main technique that I implemented and kept in the code was multithreading. This allowed the AI to search deeper than 4 and keep a speed of at least one move per second. This however did not help as much as I wanted since going deeper than 5 would result in moves taking several seconds, which I did not want. The multithreading definitely helped however because without it a depth of 6 would take 20+ seconds. For more details on other techniques I tried and why I kept/did not keep them check the technical decision making section. Related to the last point, because the speed of AI was a big issue and I struggled to fix that, the heuristics was my main concern in making the AI perform better. However, even with my focus on the heuristics I never managed to get the AI to the stage where it could beat the game 100% of the time. This is something that I would love to improve upon in the future. I will go over the AI in detail in the coming sections. The last thing I planned to include was a menu system that allowed the user to easily switch between these modes. I completed this goal however, I would have liked to make the GUI look nicer in general and give it more flair, this is something I can improve upon in the future but did not deem necessary when I was under a time limit. One more goal I had was to make the project more unique by including a version of the game on a hexagonal grid instead. This was something I planned to do if I had extra time, but I did not manage my time well enough and spent a lot of time trying to improve the AI's score. This is something I definitely would like to try in the future if I manage to perfect the AI's score and make it beat the game 100% of the time. Speaking of which, I will also go over the scores of my AI and how well it performed with data examples in the coming sections. I gathered the data for the AI automatically after every run by writing to a text file called 'data.txt'.

5.2 Practical aspects of the code

5.2.1 'FinalGame.py' (Menu)

This section will go over the important parts of the code in my final game.

Firstly, the menu of the program that has the purpose of giving the user a main menu screen with buttons that send them to another screen. When they click these buttons, the window that they are currently on will change to the desired window seamlessly. No new windows will open but the current one will change. This is done by creating a class with a single frame.

```
class Windows(tk.Tk):
    def __init__(self, *args, **kwargs): #Allows for extra arguments.
        tk.Tk.__init__(self, *args, **kwargs)
        self.wm_title("AI 2048")
```

```
#Create a frame and assign it to a container
container = tk.Frame(self, height=400, width=600)
#The region where the frame is packed in root
container.pack(side="top", fill="both", expand=True)
```

There is then a function that is made that changes the current frame with a different one creating the seamless experience of changing windows.

```
def show_frame(self, cont):
    frame = self.frames[cont]
    #Raises current frame to the top
    frame.tkraise()
```

The buttons on each page simply have a ‘command=lambda:’ call to this function to call each of the different pages.

```
switch_window_playPage_btn = tk.Button(self, text="Play Game",
command=lambda: controller.show_frame(PlayPage), activebackground="red", bg=
"#e88504", width=40)
switch_window_playPage_btn.pack(pady=100)
```

I followed a tutorial for the menu program since I did not deem it something I should be trying myself and just wanted the best possible way of doing it (See reference [1] for the page). I did however change the GUI and added elements to make it look more like the original game of 2048.

5.2.2 ‘FinalGame.py’ (AI Page)

This file also includes the code for the AI and user GUIs as well as the code to actually run the AI game.

On the page there are several buttons and an empty grid to start out with. Once you click the run button the game will start, and the user will have the option to change between the different speeds that the AI provides. The page also includes a score and highest tile label that updates in real time for the user’s convenience. For an image of this page check (figure 15) in the code deployment section. When the run button is clicked the game enters a loop that infinitely finds the best moves until the number of spaces left equals zero.

```
if (self.game.grid.move(self.game.ai.get_best_move(self.game.grid,
self.game.depth)) == False):
    for i in self.game.moves:
        if (self.game.grid.move(i) != False):
            break
    self.update_grid_cells()
```

When this happens a hidden game over label becomes visible again and shows the user that the AI has failed. The Optimal button changes the depth of the game according to how many spaces are left on the board.

```
if (self.optimalToggle):
    if (len(self.game.grid.get_empty_spots()) < 7):
        self.game.depth = 5
```

```
else:
    self.game.depth = 4
```

This allows for the game to still move quickly but make smarter moves when needed.

5.2.3 'FinalGame.py' (Player plays page)

This page is very similar to layout as the AI page excluding the buttons that give the option to change the speed of the AI, because here the AI is not playing but the user is. There is a simple label telling the user the buttons to play the game and as soon as the run button is clicked the program waits for key presses and acts accordingly.

```
# Waits for key presses.
self.scoreLabel.bind_all("<Key>", self.move_game)
```

The code above is binding an event reader to the label to make sure that whenever a key is pressed, the next function is run with the key that was pressed as a parameter.

```
def move_game(self, event):
    if (event.keysym == "w" or event.keysym == "Up"):
        self.game.grid.move("up")
    if (event.keysym == "a" or event.keysym == "Left"):
        self.game.grid.move("left")
    if (event.keysym == "s" or event.keysym == "Down"):
        self.game.grid.move("down")
    if (event.keysym == "d" or event.keysym == "Right"):
        self.game.grid.move("right")
```

I did not include a detailed section for how the GUI works for these pages as it is mainly a lot of labels being changed and there is not any complicated code that I have not already mentioned in the menu section. One feature worth mentioning that I added, which is both a part of the play page and the AI page, is a real-time counter for the score and the current highest tile. I did this by having a label for both of them and then a function that updates each one with their corresponding return methods (for example the return_score method), both of these functions are called every move when the grid cells are updated.

5.2.4 'Grid.py' (2048 logic)

This file is meant to be a recreation of the base game of 2048. Originally I had followed a tutorial to make a console only version of the game (see reference [2]). This version of the game however did not use OOP and so I eventually had to recode all of it to make sure it would work with my later implementations of the AI. The original version also had many problems, such as the game adding tiles when making moves that did not affect the grid in any way and the grid size not even being four by four. I fixed these problems before I moved to recode the game.

The game represents the grid by creating a simple matrix, this is done in the initialisation of the class. If I was to improve this program in the future, the part where '[0]' is being multiplied by four can be changed to a variable of the grid size to allow for different sized grids. As for different shaped grids, I am not sure if using a matrix would work, so I would have to look into another approach.

```
def __init__(self) -> None:
    self.matrix = [] # Declaring an empty list.
    for i in range(4): # Giving the matrix the shape it should have.
        self.matrix.append([0] * 4)
```

Next is a function for adding a random two or four tile into the grid. This is done by first getting a random number to determine whether a two or a four should be added. A two is added if less than 0.9 and a four if greater, this is to emulate the chances of the tiles, two being 90% and four being 10%. The number is stored in the 'number' variable.

```
def add_random_tile(self) -> None:
    # Getting a 4 has a 10% chance
    if random.random() < 0.9:
        number = 2
    else:
        number = 4
```

It then finds two random numbers between 0 and 3 (both 0 and 3 included) corresponding to the indexes for the matrix. If the value of that spot is zero (an empty spot) it will continue, otherwise it will keep finding random spots. It then adds the new tile into the matrix using the indexes and the 'number' variable.

```
# choosing a random index for row and column.
r = random.randint(0, 3)
c = random.randint(0, 3)

# while loop will break as the random cell chosen will be empty (or
contains zero)
while(self.matrix[r][c] != 0):
    r = random.randint(0, 3)
    c = random.randint(0, 3)

# we will place a 2 or 4 at that empty random cell.
self.matrix[r][c] = number
```

Next is the moving of the tiles and the merging. This is done in multiple functions, namely the compress, merge, reverse and transpose functions. The first two functions are then called by the move_left function. The right and up move functions call move_left with the reverse and transpose functions correspondingly and the down function calls the right with the transpose. This allows for moving all tiles in the grid in specific directions.

Firstly, the compress function compresses the grid after every step before and after merging the cells. It shifts the tiles to their extreme left, row by row. It does this by checking if the cell is non-empty and if it is, shifts the tile to the previous empty cell in the row shown by the 'pos' variable.

```
def compress(self) -> bool:
    changed = False
    new_mat = []
    for i in range(4):
        new_mat.append([0] * 4)
    # here we will shift entries of each cell to it's extreme left row by row
    for i in range(4):
        pos = 0
        # loop to traverse each column in respective row
        for j in range(4):
            if(self.matrix[i][j] != 0):
    # if cell is non empty then we will shift it's number to previous empty cell
    in that row denoted by pos variable
```

```

        new_mat[i][pos] = self.matrix[i][j]
        if(j != pos):
            changed = True
        pos += 1
    self.matrix = new_mat
    return changed

```

Next, the merge function combines the cells after the grid has been compressed. It does this by checking if the current cell has the same value as the next in the row and they are both non-empty.

```

def merge(self) -> bool:
    changed = False
    for i in range(4):
        for j in range(3):
            # if current cell has same value as next cell in the row and they are non-
            # empty then
            if(self.matrix[i][j] == self.matrix[i][j + 1] and
self.matrix[i][j] != 0):
                # double current cell value and empty the next cell
                self.matrix[i][j] = self.matrix[i][j] * 2
                self.matrix[i][j + 1] = 0
            # make bool variable True indicating the new grid after merging is
            # different.
            changed = True
    return changed

```

These are then called in the move_left function in the order compress, merge and compress. The other move functions call reverse and transpose. Reverse reverses the content of each row.

```

def reverse(self) -> None:
    new_mat = []
    for i in range(4):
        new_mat.append([])
        for j in range(4):
            new_mat[i].append(self.matrix[i][3 - j])
    self.matrix = new_mat

```

Finally, transpose interchanges the columns and the rows of the matrix.

```

def transpose(self) -> None:
    new_mat = []
    for i in range(4):
        new_mat.append([])
        for j in range(4):
            new_mat[i].append(self.matrix[j][i])
    self.matrix = new_mat

```

Finally, the last function I made was a general move function that would call the other move functions. This was needed because when the other move functions were called I needed a way to use the 'changed' variable that indicates whether the grid had actually made the move. This allowed me to check if the move was actually made and if it was, the function would automatically add in the next tile.

```

def move(self, direction) -> bool:

```

```

        changed = False
        if (direction == 'up'):
            changed = self.move_up()
        if (direction == 'right'):
            changed = self.move_right()
        if (direction == 'down'):
            changed = self.move_down()
        if (direction == 'left'):
            changed = self.move_left()
        if (changed == True): # Checks if move was able to be made.
            self.add_random_tile()
            return changed
        return changed

```

5.2.5 'AI2048.py'

For the game to actually run I made another file which creates a Game class. This Game class initialises the previous Grid class, initialises the AI class, adds the first tile to the grid and some other things. This classed is used in the GUI file to create an object that holds the grid and AI objects within it. However, the file itself was mainly used to test the AI in an environment where the game could loop endlessly and gather data in the data file.

```

def __init__(self) -> None:
    self.grid = Grid()
    self.ai = AI()
    self.gameOver = False
    self.depth = 5
    self.moves = ['up', 'right', 'down', 'left']
    self.grid.add_first_tile()

```

I wrote the results of the game into a text file and also had the code be capable of showing the time per move and the average time per move overall.

```

moveTimes.append(time.time() - start)
print("avg: ", sum(moveTimes)/len(moveTimes))
f = open("data.txt", "a")
row_item = [str(self.return_score()), str(self.return_highest_tile())]
data = "{}{:>20}".format(row_item[0], row_item[1])
f.write("\n"+data)
f.close()

```

The looping simply made the object and ran the function to play again. This allowed me to leave the program running on the slower versions in the background, which helped me a lot when I was gathering data for the AI and testing which heuristics performed better overall.

```

if __name__ == '__main__':
    # For the AI plays version of the game.
    for i in range(10):
        game = Game()
        game.run_AI_game()

```

5.2.6 AI-related code

For my AI, different parts of the code had blocks that had to be added/modified to let the AI work in the grand scheme. The main Expectimax algorithm is in a new file called AI.py, this file includes two functions. One is to calculate the best move and the other is the Expectimax, the best move function calls the Expectimax function. The grid class also had additions to help with the AI.

Firstly, the functions that were added to the grid class to help with the AI. The simplest but particularly important one is the clone function. It simply makes a copy of the current grid object; its main use however is to copy the matrix and still have access to the grid functions.

```
def clone(self) -> object:
    copy = Grid()
    copy.matrix = np.copy(self.matrix)
    return copy
```

Next we have a simple function to make a list of all the empty spots in the grid. This is used frequently throughout the code in combination with len as it allowed me to just get the number of empty spots, not the specific cells.

```
def get_empty_spots(self) -> list:
    cells = []

    for i in range(4):
        for n in range(4):
            if (self.matrix[i][n] == 0):
                cells.append([i, n])
    return cells
```

Also, another simple function to insert a specific tile into any position on the grid. Both of these functions are used in the computer's turn in the Expectimax algorithm.

```
def insert_tile(self, position, value) -> None:
    self.matrix[position[0]][position[1]] = value
```

The last function that was added to the grid class is a method to calculate the AI score of a particular game state. This is an important function as it determines how the AI picks what moves are actually the best moves to make. I finished with multiple heuristics, however not all of them are included in the final code. This may be because I found them to make little difference or if they were to make more of a difference I would need to do a lot more testing on different weightings. I will go over the heuristics included in detail in the theory section, and for more details on the heuristics I may not have included, those will be included in the technical decisions section.

The heuristics included currently are weight matrix, empty tiles, highest tile in right spot and smoothness. In order their codes are:

```
self.weights = [[2048, 1024, 512, 256],
                [16, 32, 64, 128],
                [8, 4, 2, -2],
                [-32, -16, -8, -4]] # Weighted matrix
for i in range(4):
    for n in range(4):
        score += (self.weights[i][n] * self.matrix[i][n]) *
matrixWeight
```



```
# Empty squares
score += len(self.get_empty_spots()) * math.log(self.get_score()) *
emptySquaresWeight
```

```
# Highest tile in right spot
if (self.get_highest_tile() == self.matrix[0][0]):
    score += self.get_highest_tile() * highestTileWeight
else:
    penalty += self.get_highest_tile() * highestTileWeight
```

```
# Smoothness
directions = [[1, 0], [-1, 0], [0, 1], [0, -1]]
for y in range(4):
    for x in range(4):
        if (self.matrix[y][x] != 0):
            for i in range(4):
                position = [y + directions[i][0], x +
directions[i][1]]
                if (self.is_in_bounds(position)):
                    neighbour =
self.matrix[position[0]][position[1]]
                    if (neighbour != 0):
                        penalty += abs(neighbour -
self.matrix[y][x]) * smoothnessWeight
```

When it comes to the AI.py file where the Expectimax algorithm is run there are two functions. I did a lot of my research on how to actually implement Expectimax by looking at examples of other people's implementations of it (see references [3], [4] and [5]).

Firstly, we have the `get_best_move` function. This function has two modes that determine how it will decide the best move. If the depth is set to four or less, it creates a clone of the current grid and makes a move on that clone. It does this for each of the four moves that can be made. If the move does not result in a different grid, then that move is omitted from being one of the best moves. If the move can be made, the Expectimax function is called with the clone, `depth - 1` and "computer" as parameters. The Expectimax function returns a score and if this score is greater than the past score, the score is updated, and the best move is changed to the move that resulted in this score.

```
def get_best_move(self, matrix, depth) -> str:
    score = 0
    bestMove = None
    self.matrix = matrix
    self.depth = depth
    if (self.depth <= 4):
        for i in self.moves:
            newMatrix = matrix.clone()

            if (newMatrix.move(i) == False):
                continue
            newScore = self.expectimax(newMatrix, depth - 1, "computer")
```

```

        if (newScore > score):
            bestMove = i
            score = newScore

    return bestMove

```

If the depth is greater than four, it does the same thing but with multiprocessing instead. For the multiprocessing to work I had to create a function that starts the Expectimax, since the map function needs a function and a list of the parameters it will run on that function in parallel.

```

else:
    with Pool(6) as pool:
        newScore = pool.map(self.startExpectimax, self.moves)
    for i in range(len(newScore)):
        if (newScore[i][0] > score):
            bestMove = newScore[i][1]
            score = newScore[i][0]
    return bestMove

```

The reason I have this and not just multiprocessing all of the time is because, when searching with a depth of four or less, using multiprocessing is actually slower than searching normally. I believe this may be because the overhead of creating the threads for every move is much larger than searching through normally the small number of game states that comes with a depth of four (4^4).

Finally, the Expectimax function is a recursive function that is split into two parts: a player turn and a computer turn. Each part calls the other until the depth reaches zero, signalling the end of the recursion.

```

def expectimax(self, matrix, depth, agent) -> float:
    if (depth == 0):
        return matrix.get_AI_score()

```

The player part is pretty much identical to the get_best_move function however it only returns a score and not the best move.

```

elif(agent == "player"):
    score = 0
    for i in self.moves:
        newMatrix = matrix.clone()
        nextLevel = newMatrix.move(i) # Try each possible move.
        if (nextLevel == False):
            continue
        newScore = self.expectimax(newMatrix, depth - 1, "computer")
        if (newScore > score):
            score = newScore
    return score

```

The computer part is different in that it's simulating the turn when the computer spawns a random two or four tile. It tries placing a two and a four in every empty spot and then calls the players turn in the Expectimax function. The scores returned by the part that places a four in every spot are multiplied by 0.1 to reflect the chance of that game state happening (10%) and 0.9 for the two's placement (90%).

```

elif (agent == "computer"):
    score = 0

```

```

cells = matrix.get_empty_spots()
totalCells = len(cells)

for i in range(totalCells):
    newMatrix = matrix.clone()
    newMatrix.insert_tile(cells[i], 4)
    newScore = self.expectimax(newMatrix, depth - 1, "player")
    if (newScore == 0):
        score += 0
    else:
        score += (0.1 * newScore)
    newMatrix = matrix.clone()
    newMatrix.insert_tile(cells[i], 2)
    newScore = self.expectimax(newMatrix, depth - 1, "player")
    if (newScore == 0):
        score += 0
    else:
        score += (0.9 * newScore)
if (totalCells != 0):
    score /= totalCells
else:
    score = 0
return score

```

After the whole recursion is done, the move that returned the highest score is the best move the AI can take, weighed against the chances of tile spawns.

5.3 Theoretical aspects of the code

5.3.1 Using SE tools

During the development of my project, I used the version control system provided to us by the university, namely GitLab. The use of my GitLab (see reference [6]) was quite simple in the first term. Since I was not working on the final and main code, I did not use techniques such as branching the repository into working and complete branches. Since I only worked on my proof of concept programs, I simply updated any work I did and pushed it to the repository. This helped make sure any work I had done could not be lost either. In the next term I did however use branching methods. I mainly used this technique during when I was testing multiprocessing to make my AI faster. I did this because I had a working version of the code that I did not want to edit until I was sure my implementation of the multiprocessing was working as intended. Finally, when my code was in a finished state I released the complete version to a tag branch to show that this was the first finished version of the code.

5.3.2 Expectimax

Expectimax is the algorithm that powers the whole AI and how it thinks to make the best moves. In this section I will explain how Expectimax works in the theoretical sense as I have already gone over the implementation of it.

Expectimax is an algorithm that is a variation of minimax, a similar but different version of the algorithm that assumes that two humans play against each other. This however could not be used for my project as 2048 has only one person playing. Instead after every turn of the player the computer spawns in a random tile. This is why Expectimax is the ideal approach for making an AI

of this game, as the outcome of the algorithm depends on a combination of the player's skill and chance elements (the random spawning). In the Expectimax game tree there are chance nodes. These nodes calculate the average of all utilities under them and allows the computer to pick the path with highest utility (the most ideal move). See figure 3 for an example.

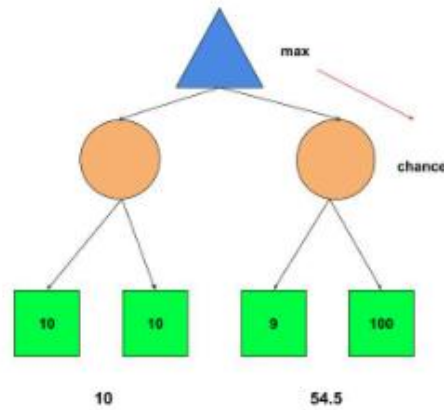


Figure 3. Example of an Expectimax game tree.

In this example, the left tree has an average utility of 10 as opposed to the right tree with an average utility of 54.5. The maximiser node chooses the right tree as it is much more likely to give a better score when taking chance into account. This is different to the usual minimax approach as that would take the left tree because the minimum is higher there. For research on Expectimax see references [7], [8] and [9].

Using Expectimax on 2048 uses these same principles, where the AI looks ahead for each possible move and calculates how beneficial of a game state it is. It then spawns a new tile in every spot and calculates a final score for the game state, this score is weighted by the chance of the specific tile spawning, for instance a 4 tile spawning has a chance of 10% while 2 has 90%. For example, the AI may see that moving down is a very good move that includes a 4 tile spawning in a very convenient space after the move, but this move will most likely be outweighed by a much safer move that considers a 2 spawning instead.

There are not many ways to reduce the total search tree of the Expectimax search algorithm (for example, pruning in minimax). One way is to remove highly unlikely game states, but this method does not help a lot with optimisation when it comes to 2048. Therefore, as said by nneonneo, “the algorithm used is a carefully optimized brute force search” (see reference [10]).

5.3.3 UML

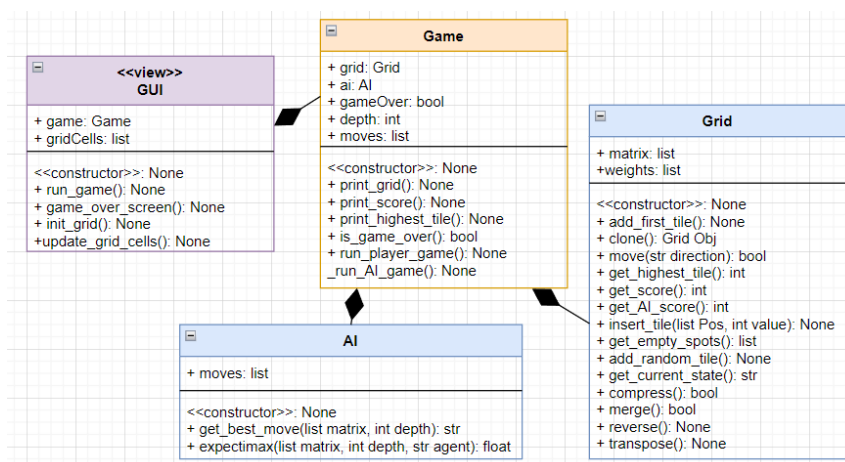


Figure 4. UML diagram of current code structure.

Above (figure 4) is a UML diagram of the structure of the AI code including the GUI. This picture shows the general layout of my code and how each class uses each other. As seen in the picture, the lowest level classes are those of the Grid class and the AI class. The Game class then instantiates both of these into one object and adds support for a few new functions for testing purposes. The Game class is then used in the GUI class to make a fully functioning game and AI supported within the GUI.

5.3.4 Heuristics

This section will go over the heuristics included in my final code, for examples of ones I tried out but decided against check the technical decisions section. For the code check section 5.2.6.

Heuristics can be considered the most important aspect of the AI. They allow the AI to ‘know’ which game states are good and will help with its longevity within the game. In the code, I include several heuristics which give a score back to the AI, the AI then adds all of the scores up and compares them to the other scores retrieved from other game states, this allows the AI to pick the move with the highest score signalling that this move is the ‘best move’. The heuristics also have a weight applied to them in the code, and this weight lets the AI know how important each heuristic is relative to each other. It also allowed me to test around with different weights to see which resulted in better average scores.

Weight matrix:

The first and one of the most important heuristics is the weight matrix heuristic. This makes sure that the AI tries to keep the board in an orderly fashion. There were two different layouts I could have gone for, one of them is a monotonic board and the other is a snake board. I went with the snake board for reasons explained in the next section. This board sets the highest tile in the corner and then snakes its way down the board with the second highest and so on. This was done by giving each cell a weight. An example of a perfectly snaked board looks like this (figure 5).

2048	1024	512	256
16	32	64	128
8	4	2	2

Figure 5. Perfectly snaked grid.

Empty squares:

The second heuristic is the empty squares heuristic. This gives the game state a higher score the more empty squares there are on the board. This heuristic is also special because the longer the game goes on, the more important it becomes. This important feature was implemented of course, I did this by taking the logarithm of the current score of the board and multiplying it with the original empty squares score.

Smoothness:

The third heuristic is the smoothness heuristic. This makes sure to try and make game states where neighbouring tiles are equal to each other. This is important because this helps the AI set up game states where tiles can be easily merged together, resulting in higher scores. I did this by checking each of the neighbouring tiles for every tile currently on the board. It first checks whether all neighbours of the tile are in bounds of the board (for example, the top left tile does not have any neighbours above or to the left of it). It then checks if the neighbours are not empty tiles, and if so, the absolute difference of the neighbours is added to the penalty. This makes it so when two neighbours are equal there is no penalty but if they are not equal there is a penalty depending on how large the difference is. Here is an example of a perfectly smooth grid (figure 6).

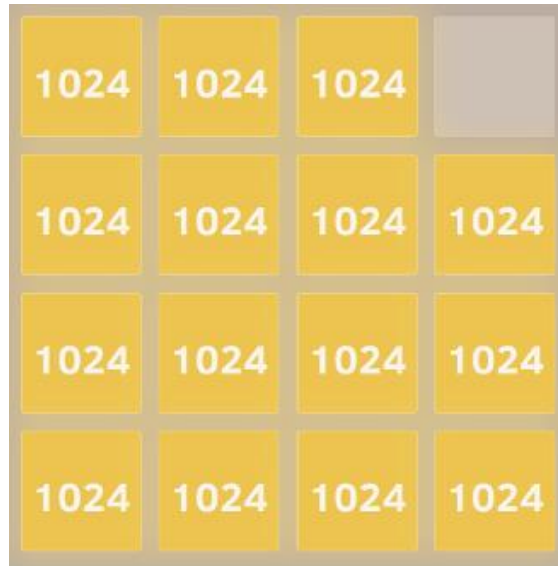


Figure 6. Perfectly smooth grid.

Highest tile in correct spot:

This is the last heuristic I added. I implemented this one after I had done some initial testing on the AI, and realised that a lot of the time, the AI would lose the game very early because it would make the mistake of moving the highest tile from its spot in the top left corner. This would result in the board ending up in a very awkward formation that the AI could not save. To put it simply all this heuristic does is check that the highest tile is in the top left, if it is then that state would get a boost to its score, and if it were not there, it would receive a penalty instead.

I did make a modified version of this heuristic that is not included in the final code, which was the same thing but instead checked that the top four tiles were in their corresponding spots and gave the same boost/penalty for each of them if they were correct. I will go over why this is not in the final code in the next section.

Again, section 5.4 will include the heuristics I tried but decided to leave out of the final code.

5.3.5 Multiprocessing

Multiprocessing is a feature I included in my final code. It speeds up the AI considerably when searching deeper and further. It does this by allowing the AI to run each of the first four moves it looks into the future on separate processes, meaning that they all run in parallel. This speeds the AI up by a large amount. Take the example of a depth of six, normally the AI would have to search through 4^6 (4096) game states, this can obviously take a while but when the first four moves are on different processes the search tree is split in four. This means that each process does its own search tree of 1024 game states instead, effectively speeding up the search by four times.

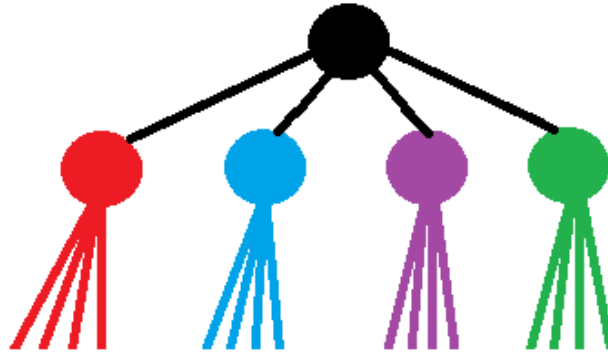


Figure 7. Illustration of the four moves being split into different processes.

I included this feature for whenever the AI searched at a depth of more than four, since anything below that ended up making it slower overall because the overhead created from making the new processes out-weighed the work needed to be done itself (the workload of smaller depths is not large enough to justify using multiprocessing).

5.4 Technical decisions

This section will include decisions I made regarding my project and why I made them.

Firstly, I will go over certain parts of my code that I decided to leave out for several reasons. There were many techniques I tried on my AI when I was testing it in an attempt to bring the overall quality of the AI up.

5.4.1 Caching

One of the techniques I implemented was caching. Caching is when you store certain data for later use, I tried this, storing a large amount of game states and their corresponding best moves, eventually removing the oldest ones when the cache became full. This was implemented because I thought that it would speed up my AI, because some moves could skip the search entirely if they were already in the cache, which would then result in me being able to search deeper. This however was not the case, when testing I found that identical game states would rarely appear, when they did the move would be much faster, but it happened so little that the overall time saved was negligible. Therefore, I opted to remove this since it made little to no difference on the product and was practically dead code. The code is still included in the project ('AI.py', reference [6]) but commented out.

5.4.2 Excluded heuristics

This section will go over the heuristics that I implemented and tried but decided to leave out of the final code for various reasons.

The first that I excluded was a simple heuristic that made moves that merge the highest tile happen more often, I did this by simply adding the highest tile to the score of that particular state, meaning that if a move had merged the highest tiles it would be worth more. The reason I did not include this however, was because it did not add to the functionality of the AI and did not raise the average score by as much as I thought it would. It also was an unnecessary heuristic since the score it overlapped with the original weight matrix heuristic, for instance, the weight matrix already would encourage merging the highest tile since the largest weight would become much more if the highest tile was larger itself. Therefore, the score added by this heuristic was simply unnecessary.

Another heuristic I added was one that would penalise the AI when the incorrect tiles got stuck in top right corner. This spot was reserved for the fourth highest tile as per the snake formation. The

AI would frequently get smaller tiles stuck there, so I implemented this to try and get rid of that problem. I did not include this however because it did not seem to work with the weightings I gave it. I could have made this work if I had more time to mess around with the weighting but since this was something I added near the end of the project I did not have the time. Therefore, I excluded it because it was dead code in the project.

One more that I did not include for the same reasons of not being able to optimise the weightings to make it work correctly with AI was the modified highest tile heuristic. This was the one I mentioned that took the top four tiles and boosted/penalised the score depending on whether they were each in their correct spots.

The last heuristic that I did not add was the other formation for the weight matrix. This formation would be one that followed a monotonic style and was the formation I used for the large majority of the project as I was working on it. This style was one, that in my research, found that a lot of others who attempted this project used, and originally I was the same. A monotonic style is one where the grid favours having large tiles in the corner and makes sure that the numbers are decreasing when going right and down from the top left. Here is an example of a perfect monotonic grid (figure 8).

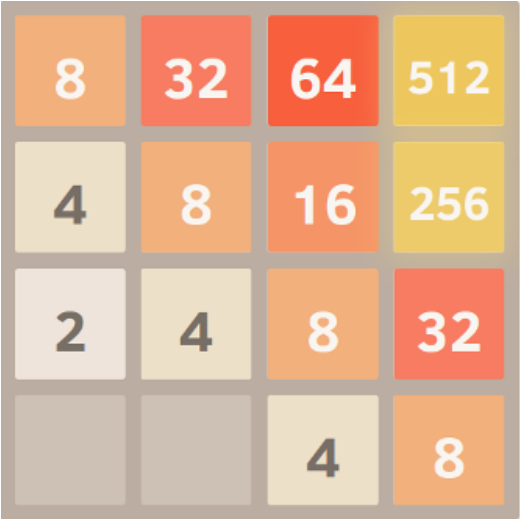


Figure 8. Perfectly monotonic grid.

Monotonic (fast)					Snake (fast)				
Score	Highest Tile				Score	Highest Tile			
802	512				2714	2048			
946	512				2136	1024			
580	256				2930	2048			
1382	1024				2888	2048			
1608	1024				506	256			
1718	1024				1032	512			
1256	1024				2104	1024			
550	256				1424	512			
572	256				1118	512			
1114	512				2050	1024			
986	512				1186	512			
2012	1024				2144	1024			
1166	512				1194	5121			
1536	1024				576	1024			
1292	1024				1578	1024			
(avg)	(mode)	(high score)	(low score)	(highest tile)	(avg)	(mode)	(high score)	(low score)	(highest tile)
1168	1024	2012	550	1024	1772	1024	2930	506	2048

Figure 9. Tests comparing monotonic and snake formations.

I did not include this heuristic because the tests I ran on the AI showed that this formation performed significantly worse than its snake counterpart. I ran 15 tests for each formation with no changes other than the formation itself, and the results were clear (figure 9).

5.4.3 Java

After the first term of the project, where all I had done were the proof of concepts programs, I realised that my AI suffered from the slow speed of python. Everywhere I researched it seemed as though people were able to get their AIs to run at incredibly high speeds. This of course could have been because of a multitude of factors, but I knew for a fact that if I were to swap to a language that used a compiler rather than an interpreter, my AI could see increases in speed by at least tenfold.

This was something I really wanted in my code, so I naively decided to try and recode all of my proof of concepts into java. This of course did not go as smoothly as planned. I like to think it was very possible to do this, but not in the time frame that I had, with only one term left I did not have the time to attempt this as I had greater concerns about the project to focus on. Eventually, I decided that this was not worth the time and made do with python, trying other methods to optimise the AI.

5.5 Performance of the AI including data

Overall, the performance of the AI is mediocre at best, I would have liked for the AI to at least be able to get a 2048 tile 100% of the time, as this is something that I saw a lot when researching others who had undertaken the same project. The AI does not seem to get much higher scores on average when searching at a depth of 5 compared to 4. This most likely would not have been the case if I was searching at depths of 6 or more, however with my current AI, a depth of 6 or more is unreasonable as the moves take far too long for comfortable watching. This is unfortunate as one of the main reasons I implemented multiprocessing was to allow for higher depths but all it really allowed was a depth of 5, since that still had moves that took under a second.

For how my AI actually performed, I gathered data on different points of the AI when I was messing around with the different weightings. The highest tile that my AI consistently gets is the 1024 tile, with an average score of around 1860. One of the most important stats collected and the one most would be interested in, is how often does the AI get the winning 2048 tile. For my AI this happens on average, every four to five games (A win percentage of 20-25%). This is unfortunate as I hoped for a much larger win percentage. This may have been possible if I was able to find the right weightings for the heuristics and managed to add in others that prevented the AI from making mistakes that lead to early losses because right now it seems very luck-based for if the AI manages to get the right sequences to win the game. Below I have included the data that lead me to these conclusions (figure 10).

Snake (fast)				
Score	Highest Tile			
4018	2048			
1170	512			
2224	1024			
2954	2048			
3138	2048			
1614	1024			
1154	512			
1954	1024			
1302	512			
1062	512			
1596	1024			
1238	1024			
1138	512			
1906	1024			
1438	1024			
(avg)	(mode)	(high score)	(low score)	(highest tile)
1860.4	1024	4018	1138	2048

Figure 10. Data on AI performance.

(final) fast				
Score	Highest Tile			
3062	2048			
1288	1024			
910	512			
1986	1024			
2166	1024			
1604	1024			
2112	1024			
1126	512			
3092	2048			
698	256			
(avg)	(mode)	(high score)	(low score)	(highest tile)
1807	1024	3092	698	2048

Figure 11. More data on AI performance.

Chapter 6: Diary

Below I have included the work diary I wrote in during the development of my project. For the markdown version, preview the 'Diary.md' file from my GitLab (see reference [6]).

13/10/2022:

Today I worked on a proof of concept program where I make a menu GUI that allows for the user to go to different pages and back to the main menu. There were a lot of examples and ways to do this online, so I just had to find an OOP approach on the problem. The code works by creating a single window with lots of different frames, these frames are then brought up to the top when needed. The frames are just the different menu pages which are created in their own classes. The menu is very bare bones at the moment in terms of aesthetics, I will make it look nicer next time.

20/10/2022:

Today I got an implementation of 2048 using a matrix and matrix functions for the moves, I took this example from the internet, but I plan to try my own implementation for the game in the time frame that I have to complete the game. The implementation that I try myself will include the GUI for the game. There were a few problems with the code I took from the internet that I fixed, for example there was a problem where the code would get stuck in an infinite loop.

09/11/2022:

I have been very busy for a couple weeks and had to put the project back. I'm back to being able to work on it again now though. Today I realised that I had a problem with my game where if you made a move that would not change the grid at all, the game would still add in an extra tile, this shouldn't be the case. I spent a lot of my time today revising for another assignment but I managed to fix this problem quite quickly. I compared the matrix before the move and after, if they were not the same the code would continue and if they were it would loop back to the move selection.

13/11/2022:

I have started reading up on and researching different AI algorithm methods for solving the game such as minimax and more preferably Expectimax.

17/11/2022:

Carried on researching more into Expectimax as this will be a focus of my interim review report. Also looked more into the practical side of things like how to implement it into the game itself.

20/11/2022:

Started implementing the Expectimax algorithm, I think I've got the base theory code down for it to work but I am running into problems with it actually running. I think it may be because the implementation of the 2048 game that I have doesn't actually use OOP. If I can't fix it tomorrow I will probably re-write the major parts of the game to work with OOP instead. I don't really want to have to do this but if I can't fix it I will have to, besides the code will be better off in OOP.

21/11/2022:**Part 1:**

Carried on working on my Expectimax algorithm. I ended up having to re-write a lot of my code to change it to OOP. This change however was something I should have started with. It made my code a lot easier to work with, and the problems I was having yesterday are not present anymore. I stubbornly tried to implement the algorithm without it because my game didn't use it either. Right now, I only have one heuristic implemented which is the weight matrix and this alone isn't actually enough for the AI to reach the 2048 tile, the highest it got in the few runs I did was 1024. I plan to implement more heuristics to improve the performance of the AI. I will add in a simple heuristic of getting bonuses for empty squares as that would be relatively easy to implement. Also, the performance of the AI speed wise isn't the greatest at the moment. If the depth of the search is less than 6 it runs decently fast and if lower than 5, it is incredibly fast. However, I would like my AI to run at a depth search of 8 when close to losing and 6 default. I will need to find a way to optimise it if I want that.

Part 2:

I added the open squares heuristic, and I wasn't too sure how large of a bonus to add to it at the start. I started by multiplying the score by 1.5 for each empty square, but this turned out to be too much as the AI wouldn't keep to the weighted matrix as much which is essential for high scores. I ended up just making it add a flat score of 10 for each empty square, this however is still not enough for the AI to even win the game. I have either got the scoring wrong for the heuristics or I simply need to add more.

Part 3:

I added another heuristic which was just a score add from the value of the highest tile, this makes it so moves where the highest tile gets merged are good moves for the AI. I also changed the bonus for empty squares to just add to the score instead of penalising. I added weights to all of the heuristics so I can change how much they should be scoring in relation to each other as well. The AI managed to hit 2048 once in my most recent run but it still takes a long time for it to get there. I need to add some sort of optimisation into the code as this will also allow me to up the depth of the search from 4 and 7. I was thinking of figuring out a way to implement a transposition table, so if needed the search terminates if getting to a game state it has already seen.

22/11/2022:

Implemented the GUI for the game from a source online. This is just a temporary GUI since I wanted one in there at least so I didn't have to keep looking at the console representation of the game when I watched the AI play. Will replace this GUI with my own version and implement it into the menu system. Eventually I want the menu to have options of whether you want to play yourself or if you want the AI to play the game. Also need to make the GUI work with arrow keys when on the player option. Would also like to have options in the AI menu for changing the depth search etc. I will probably work on this after the interim review as it's not the most important thing right now. I need to update the heuristics of my AI and its optimisation to make reaching 2048 much more likely.

13/01/2023:

Started looking into ways to improve the optimisation of the algorithm including multithreading and did some small updates on the readme. Will start implementation of the optimisation and work on improving the heuristics so the AI can beat the game more often, eventually I would like the AI to be able to beat the game at 100% efficiency.

22/01/2023:

Haven't been working on the project as I've been busy getting into a rhythm with the new modules this term. The first thing I plan to do, is combine all of my proof of concept programs into one program and make it work that way. This will require some changing around of certain things, for example, I will need to update the 'player plays' version of the game so that it works with the GUI and uses events to wait for key presses rather than having to press enter. Incorporating both the GUIs into the menu program shouldn't be too difficult but I may run into problems with how my menu code is structured. After this, I will get to testing more heuristics and trying to add multithreading.

27/01/2023:

Spent a while joining up all my proof of concepts into one final program. This took longer than expected as I had to mess around with my menu program to make it work with the AI GUI. There were problems with the buttons not working because the game loops until it loses and widgets appearing all over each other. These are all fixed now, I have yet to complete the GUI for the 'player plays' version of the game but this shouldn't take long as it is nearly identical to the AI version with just a few changes.

04/02/2023:

Started implementing multithreading with the hopes of speeding up the game by a large magnitude, this would then let me increase the search depth of the game resulting in higher scores. Had some trouble implementing it but I managed to get a version that doesn't output any errors. However, the multithreading that I have implemented is much slower than the non-MT version. Clearly something has gone wrong, and I must've made a mistake somewhere. I will continue working on this and try to fix the problems as it is my main priority for the project right now.

12/02/2023:

Carried on working on the multithreading. After further testing it seems that my implementation has actually worked when using searches deeper than 5. On the normal version a depth of 6 takes around 30 seconds per move, while with the multithreading it takes around 7 seconds. This is a big improvement and shows that the multithreading is working but I would like the each move to be under a second. I'm not sure if this is possible with multithreading anymore since I am using all 6 cores I have available. However, I still believe there is improvement to be made on my implementation. Another option is to reduce the amount of game states that are searched through, for example if the program is searching through duplicate game trees, these can be removed. I'm not sure if this is possible with expectimax but I will look into it. Also changed up the weight matrix to create a more snake like pattern that the AI should aim for, this seems to have improved the general performance of the AI.

18/02/2023:

Cleaned up the final code and merged my multithreading branch back into the main branch. I changed up the GUI for the 'AI plays' game to make it so the user has an option to change between normal and fast mode. The normal mode takes advantage of the multithreading and searches at a depth of 5. The fast mode does not use multithreading and searches at a depth of 4, the reason it does not use multithreading for this depth is because it is faster without it then with it, I think this may be because of the overhead that comes with the multithreading so there isn't much point to it at lower depths. Both of the modes have a corresponding button and can be changed during the game, there is no need to wait until the game is over. I also completed the 'player plays' version of the game and implemented it into the GUI. This was a little bit tricky since I had trouble figuring out how to use events that wait for key presses inside of the class frame system I had used for my GUI. I eventually got it though and now the game is fully functioning for the player. Both these modes

can be played from the program now and it is even possible to switch to the 'player plays' version while the AI version runs in the background. The final thing I feel like I need in my program is just to improve my heuristics now as the AI still makes dumb moves that gets itself to lose every now and then. If I can iron these problems out the AI should succeed at getting a winning score 9/10 times.

10/03/2023:

Cleaned up the GUI even more and added an extra option for the user where they can select an 'optimal' mode, this mode changes the depth on the fly to speed through the easier parts and then slow down when the game gets difficult for the AI. I also added in an extra heuristic that penalises the AI for game states that aren't very smooth (neighbouring tiles aren't equal to each other). Also added in a very simple data file where the score and highest tile are written to when the game ends, I did this so I have some data that I can use to show the performance of different modes and put them in my final report.

It is later in the day, and I have gathered more data on the AI. I tried 15 runs on the fast(depth of 4) setting for both the monotonic weight grid and the snake weight grid. The dataset is small but it seems pretty evident that the snake method seems to work better in my case with the AI beating the game 20% of time as opposed to the 0% with the other method. Later, I am going to try the same experiment but on both the optimal(depth changes accordingly) and the normal(depth of 5) modes. Hopefully, both modes should show better results for each grid type and the monotonic one may end up being better when the game tree search is 4 times as large.

13/03/2023:

Tried out some caching in hopes of speeding up the AI, it doesn't seem like it makes much difference though since identical game states don't occur very often. Also messed around with the heuristic weightings hoping to get the scores that each provide in a much closer range. I still need to experiment more but I feel this will help make the other heuristics have more impact because currently it seems only the weight matrix is being considered.

17/03/2023:

Messed around with the heuristic weighting to try and get better scores. Added a new heuristic that gives the AI either a bonus or a penalty for whenever the top tile is in the correct spot. Made a version of this for the top 4 tiles and their corresponding spots but it is commented out since I felt that it wasn't performing very well, maybe with some tweaks it may do better. Commented out the caching for now since I feel that it doesn't really make any impact on the AI since it doesn't occur very often and when it does it seems to be for the worse not the better. Changed up the empty tiles heuristic to make it more prominent as the game goes on. I may do a few more changes to the code but nothing major as the deadline is coming up and I have to focus on the report.

20/03/2023:

Fixed a small error that may have made some games end early. A lot of my game loops were made so that when there were no empty spaces left the game would end. After some thinking I realised that this is not always the case, so I swapped them around to use the original loss definition.

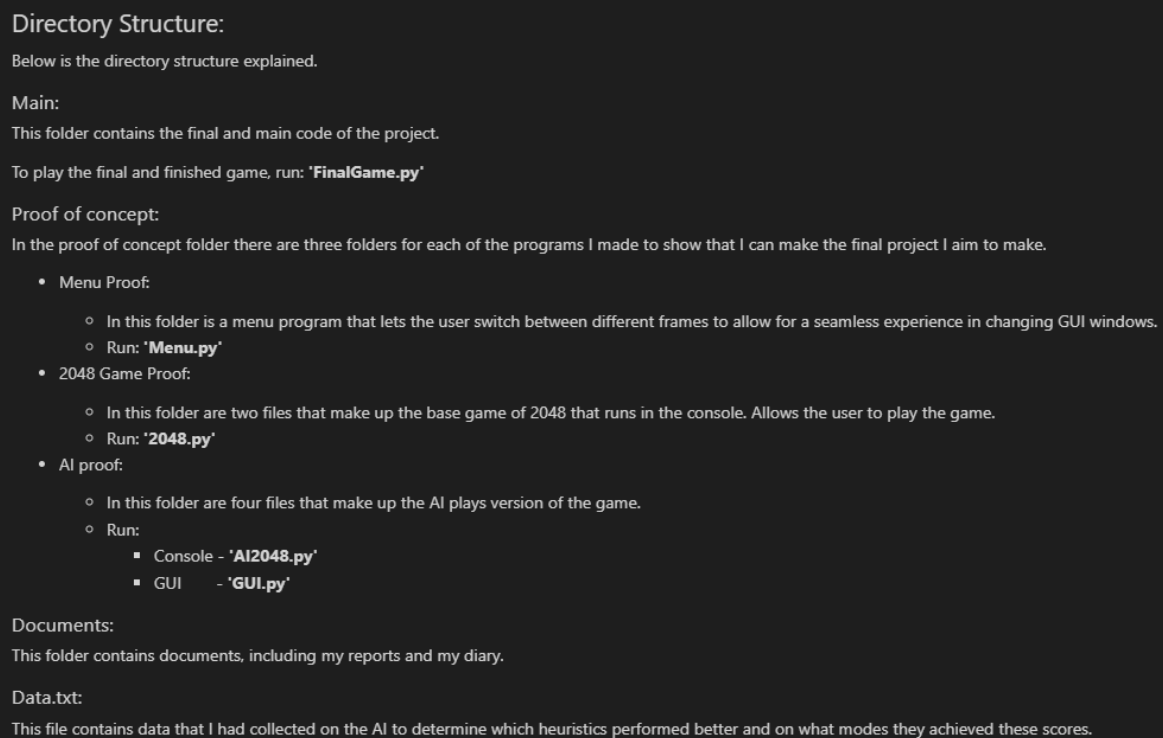
21/03/2023:

Finalised the code. I cleaned up a lot of the comments in the code. The blocks of code that were techniques I tried to use and decided against have been added into my report. Updated the readme for the new code and created a tag release for version 1.0 of my code.

Chapter 7: Running the program

7.1 Submission directory structure

The structure of this submission is as follows. The zip file will include one folder and this report. The folder contains my git directory, and the readme will provide instructions on which files to run for each of the programs I have written and where they are located. If you would like to read the .md files in their intended way, open the file through my GitLab repository (reference [6]) or through an IDE. Below is a screenshot of the directory structure section of the readme in case problems occur.



Directory Structure:

Below is the directory structure explained.

Main:

This folder contains the final and main code of the project.

To play the final and finished game, run: **'FinalGame.py'**

Proof of concept:

In the proof of concept folder there are three folders for each of the programs I made to show that I can make the final project I aim to make.

- Menu Proof:
 - In this folder is a menu program that lets the user switch between different frames to allow for a seamless experience in changing GUI windows.
 - Run: **'Menu.py'**
- 2048 Game Proof:
 - In this folder are two files that make up the base game of 2048 that runs in the console. Allows the user to play the game.
 - Run: **'2048.py'**
- AI proof:
 - In this folder are four files that make up the AI plays version of the game.
 - Run:
 - Console - **'AI2048.py'**
 - GUI - **'GUI.py'**

Documents:

This folder contains documents, including my reports and my diary.

Data.txt:

This file contains data that I had collected on the AI to determine which heuristics performed better and on what modes they achieved these scores.

Figure 12. Directory structure of the code folder.

7.2 Package installation

To run the program, I used python 3 and it is essential that the user does the same to reduce problems arising.

Packages included so far:

- Numpy
- Tkinter
- Random
- Math

These are included in the python standard library (except numpy). To install the ones not included:

Before installing the packages, the user must have PIP installed on their computer.

- Windows: <https://www.liquidweb.com/kb/install-pip-windows/>
- Linux: <https://www.tecmint.com/install-pip-in-linux/>
- Mac: <https://www.groovypost.com/howto/install-pip-on-a-mac/>

7.2.1 Numpy installation

For Numpy simply run this line in the command prompt/terminal:

```
pip install numpy
```

7.3 Starting the program

As seen in (figure 12), to run the program you must run the 'FinalGame.py' file. After this a menu will open (figure 13). To play the game yourself click on the 'Play Game' button and then click the 'Play' button to run the game, the controls are listed at the top of the page (figure 14). To run the AI version of the game, click on the 'AI Play' button and then click the 'Run' button. The speed of the AI can be changed with the corresponding buttons (figure 15). With both pages you can restart the game by clicking the same button that you did to start the game.

7.4 Code deployment

7.4.1 Video of code deployment

Below is a link to a video where my code is deployed (Turn captions on as there are subtitles).

<https://youtu.be/0NM1m2K95Zk>

7.4.2 Pictures of code deployment

Below are screenshots of my code being deployed.

Menu:

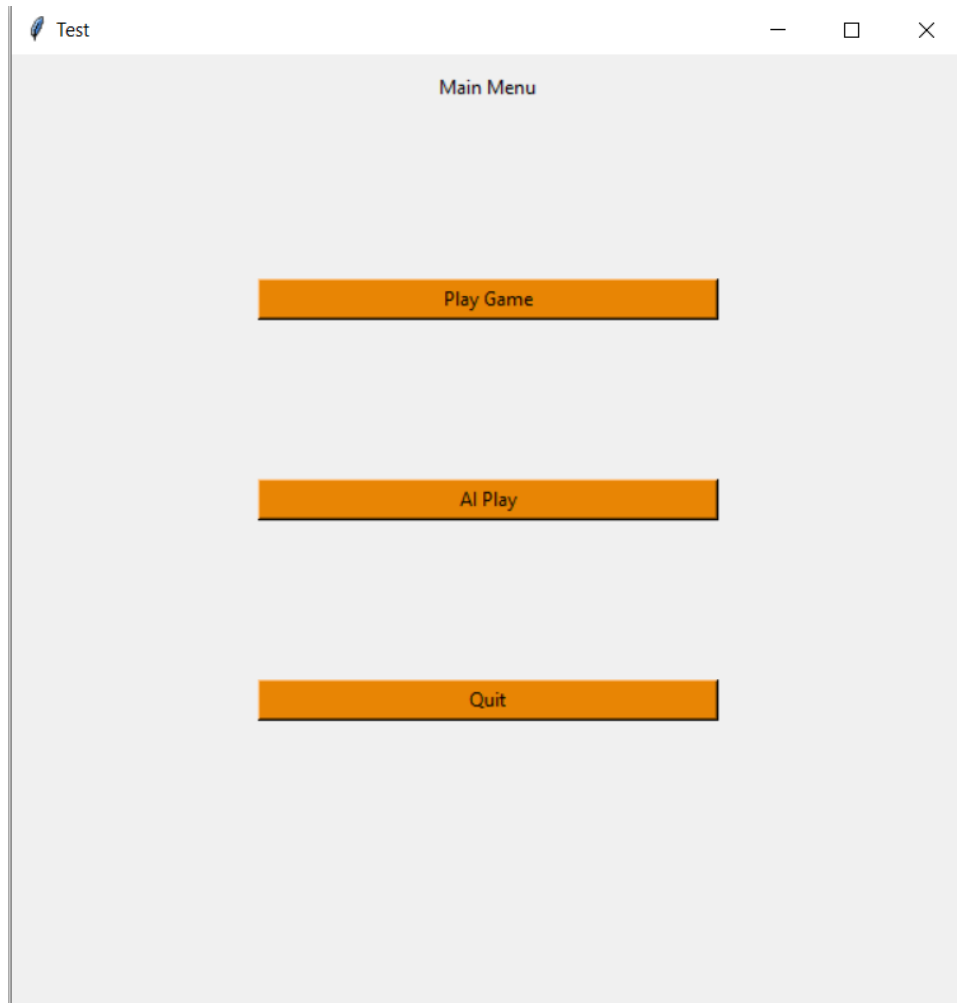


Figure 13. Main menu page with buttons for different pages.

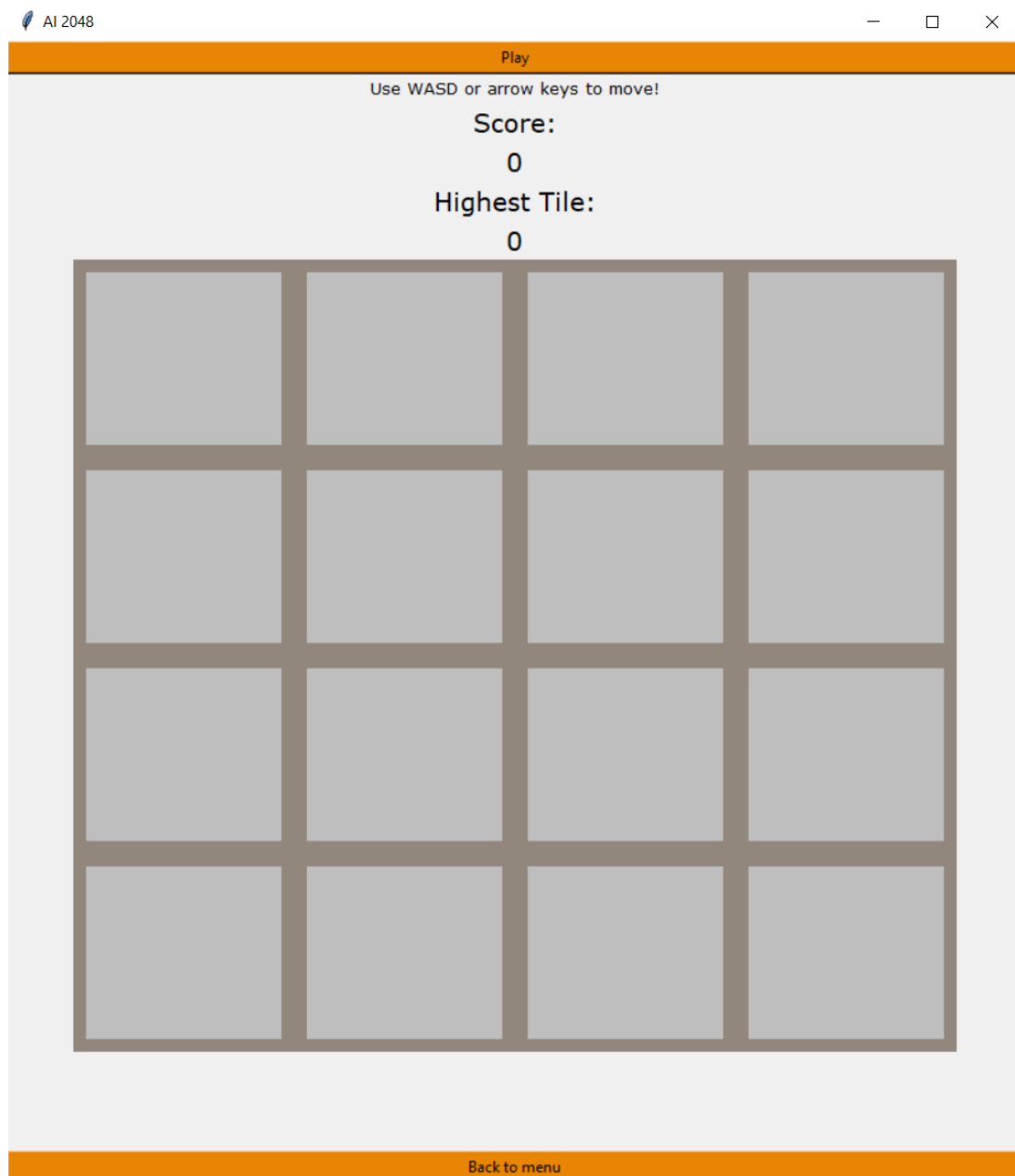
Player plays version of the game:

Figure 14. Player plays screen before game is run.

AI plays version:

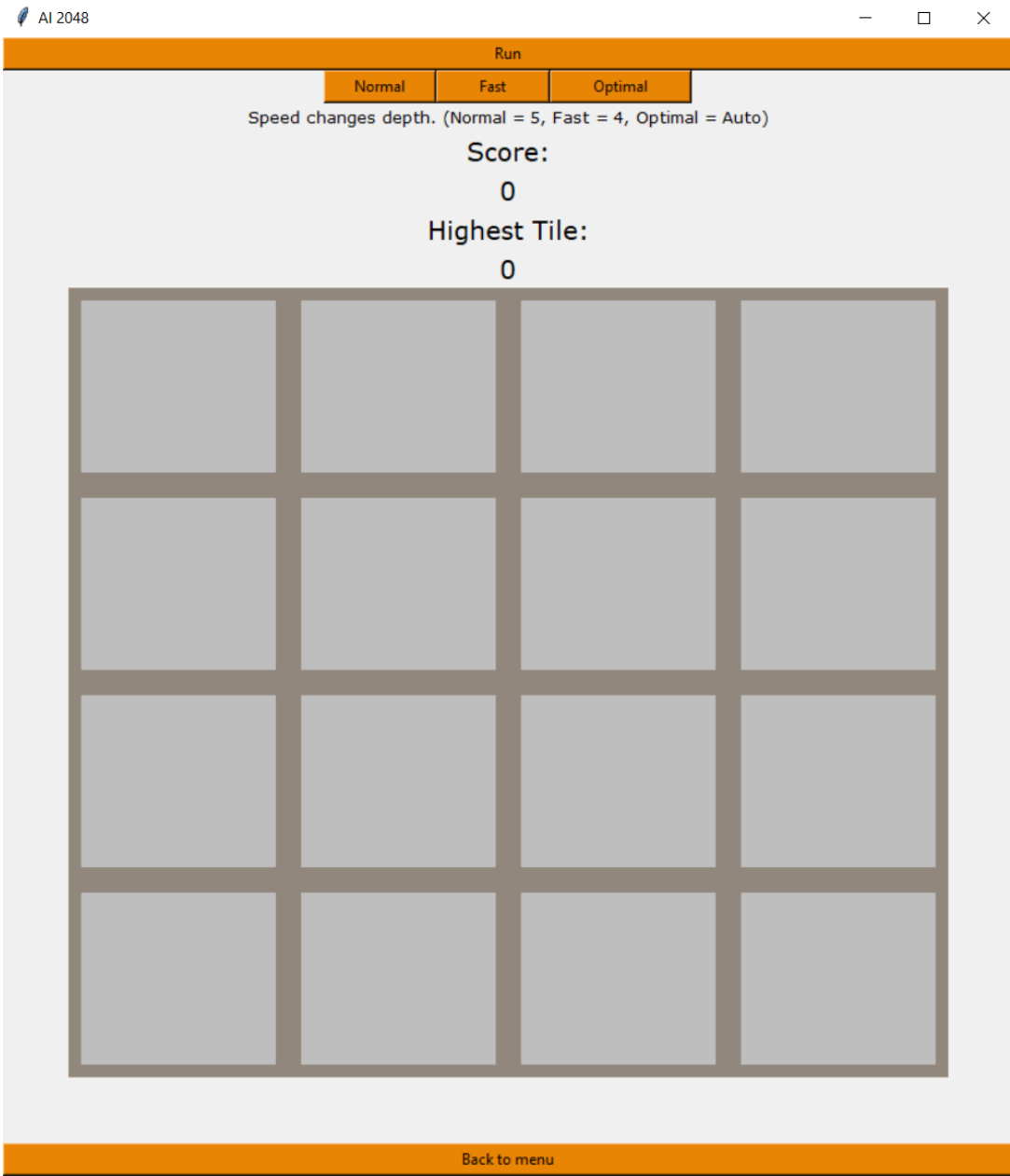


Figure 15. AI screen before the game is run.

Game Over:



Figure 16. Example of the game over screen.

Bibliography

[1] Gangwar, M. (2022) *Advanced tkinter: Working with classes*, DigitalOcean. Available at: <https://www.digitalocean.com/community/tutorials/tkinter-working-with-classes>

- I used this website for the original template for my menu proof of concept program, which I then modified and adapted for use in my final program GUI.

[2] Mangal, A. (2023) *2048 game in Python*, GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/2048-game-in-python/>

- I used this website to get a general idea for my first 2048 game implementation on the console. I had to change this one up a lot because this version of the game did not use OOP and had a lot of broken features, but it still served as a good starting point for the game.

[3] Ovolve (2015) *OVOLVE/2048-ai: A simple AI for 2048*, GitHub. Available at: <https://github.com/ovolve/2048-AI>

- Used this repository as a means to get ideas and research how to implement the expectimax algorithm.

[4] Lesaun (2018) *Lesaun/2048-expectimax-ai: 2048 game solved with Expectimax*, GitHub. Available at: <https://github.com/Lesaun/2048-expectimax-ai>

- Used this repository as a means to get ideas and research how to implement the expectimax algorithm.

[5] Wahab16 (2018) *Wahab16/2048-game-using-expectimax: 2048 game using Expectimax*, GitHub. Available at: <https://github.com/Wahab16/2048-Game-Using-Expectimax>

- Used this repository as a means to get ideas and research how to implement the expectimax algorithm.

[6] Miah, R. (2022) *My 2048 AI University Project*, GitLab. Available at: <https://gitlab.cim.rhul.ac.uk/zjac382/PROJECT>

- My own repository for the project.

[7] Yarasca E. , Nguyen K. (2018), *Comparison of Expectimax and Monte Carlo algorithms in solving the online 2048 game*.

- Research on expectimax and used in my literature review.

[8] AftaabZia (2021) *Expectimax algorithm in game theory*, GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/expectimax-algorithm-in-game-theory/>

- Used in general research on expectimax and learning how it worked, I did not use it for 2048 specifically but more on understanding expectimax itself.

[9] Wu C. , Yeh K. , Liang C. *et al.* (2014), *Technologies and Applications of Artificial Intelligence*, pp. 366-378.

- Used in my literature review.

[10] nneonneo. *et al.* (2014) *What is the optimal algorithm for the game 2048?*, *Stack Overflow*. Available at: <https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048>

- Used a lot in the early stages of this project to learn and get ideas of what others did and how they achieved their AI's.

[11] Veness J. , Blair A. (2007), *Effective Use of Transposition Tables in Stochastic Game Tree Search*.

- Used in my research when I was thinking of using the techniques in this paper for my own project. It is basically a cache, which I eventually decided against using in my project.

[12] Szubert M. , Jaskowski W. (2014) *Temporal Difference Learning of N-tuple Networks for the Game 2048*.

- Used in my literature review.

[13] Nneonneo (2021) *NNEONNEO/2048-ai: AI for the 2048 game*, *GitHub*. Available at: <https://github.com/nneonneo/2048-ai>

- Used when I was doing background research for my project, as this person had a different approach to the AI which encoded the whole board as 64-bit integers.