Final Year Project Interim Report

Full Unit – Interim 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

December 01, 2022

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 5959

Student Name: Rayan Miah

Date of Submission: 01/12/2022

Signature:

Table of Contents

Abstract

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 going deeper into the project as a whole. These sections include:

- Aims, objectives and literature survey
- Planning and time-scale
- Summary of completed work
- Section describing the practical aspects of the code
- Section explaining the theory behind the AI
- Bibliography and citations
- Diary
- 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

Chapter 1: **Aims, objectives and literature survey**

1.1 Aims and objectives

In my project I have several aims and objectives. To begin with, I want 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 allows 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 allows 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). If possible, I would also like 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.

1.2 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 2. 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.

Chapter 2: **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 3: **Summary of work and practical/theoretical aspects of the code**

3.1 Summary of completed work

In this term I have completed all of the work I had planned to, excluding the attempt at a hexagonal 2048 proof of concept. This means I completed the menu proof of concept which allows the user to move between different windows that have distinct functions. I also completed the base game that allows the user to play 2048, although originally I had a version of the game that did not use OOP, and this had caused me problems later down the line. Then I completed the Expectimax implementation, which required me to recode the base game to use OOP as it made it much easier for the AI to be implemented. I worked on improving the heuristics to try to raise the score of the AI and managed to make it better, although the AI has only reached the 2048 tile once so far. I plan to spend more time calibrating the scoring of the AI and adding more heuristics in term two to make sure my AI can get high scores. I would also like to spend my time in term two making my version of the 2048 AI unique in different ways, namely, the hexagonal version. If I cannot do this I will find other ways of making the code different such as changing the grid size or making it so the AI can give hints to the user when they are playing the game themselves.

3.2 Practical aspects of the code

3.2.1 'Menu' proof of concept program

The menu proof of concept is a 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)
        #Title
        self.wm_title("Test")
        self.geometry("600x600")

        #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))
```

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 also have yet to implement this menu program into the rest of the codes GUI, this is something I plan to do in term two.

3.2.2 'Base game of 2048' proof of concept program

This proof of concept program 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 anyway 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. The part where [0] is being multiplied by four can be changed to a variable of the grid size in the future to allow for different sized grids. As for different shaped grids, I am not sure if using a matrix will work as well so I will have to look into another approach.

```
def __init__(self) -> None:
            self.matrix = [] # Declaring an empty list.
            # Giving the matrix the shape it should have.
            for i in range(4):
                  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 not 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. This is all in the above function.

```
            # 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:
            # Bool variable to determine any change happened or not
            changed = False
            new_mat = []
            for i in range(4):
                  new_mat.append([0] * 4)

            for i in range(4):
                  pos = 0
                  for j in range(4):
                        if(self.matrix[i][j] != 0):
                              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(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
# Change bool 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
```

These functions are called as said above.

For the game to actually run I made another file which creates a Game class. This Game class initialises the previous Grid class, sets a Boolean gameOver variable and adds the first tile to the grid.

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

Every move, the user's input is taken and if the move results in a different grid a new random tile is added. At the end of the loop, the game checks if the game is over, and if so it ends the game, printing the score and highest tile achieved.

```
def run_player_game(self) -> None:
        print("Player plays 2048.\n")
        print("""Controls:
        w: up
        a: left
        s: down
        d: right""")

        while (self.gameOver != True):
            print("\n")
            self.print_score()
            self.print_grid()
            print("\n")
            playerInput = input("Enter a move: ")
            if (playerInput == 'w'):
                self.grid.move('up')
            elif (playerInput == 'a'):
                self.grid.move('left')
            elif (playerInput == 's'):
                self.grid.move('down')
            elif (playerInput == 'd'):
                self.grid.move('right')
            else:
                print("Invalid key press.")
            self.is_game_over()
        print("\nGAME OVER.")
        self.print_score()
        self.print_highest_tile()
```

Next term, I would like to incorporate the player version of the game into the GUI and make it wait for arrow presses for the moves. This would be much more user friendly as it would not require them to press enter after every move and the game would not be displayed in the console.

3.2.3 'AI plays 2048' proof of concept program

For the 'AI plays' version of the game, the code is mostly the same with a few method additions in the Grid.py file to help with the AI. 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.

Firstly, the functions that were added to the grid class to help with the AI. The simplest but most 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.

```
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. At the moment, I have three heuristics but these need to be improved on and more need to be added. I also need to research the proper way to score the heuristics as this would make my AI much smarter.

The monotonicity heuristic makes sure that the grid favours having the larger tiles in the corners and that the grid is decreasing when going right and down from the top left. This was done by making a weight matrix in the initialisation and multiplying the tile value by the weight they are currently occupying.

```
self.weights = [[7, 5, 3, 1], # Weighted matrix used for the AI scoring.
                [5, 3, 1, 0],
                [3, 1, 0,-1],
                [1, 0,-1,-2]]

def get_AI_score(self) -> int:
            score = 0
            penalty = 0
            monotonicityWeight = 1.5
            emptySqauresWeight = 2.5
            highestTileWeight = 1

# Monotonicity heuristic. Each cell is weighted to ensure the AI makes
# the best moves.
            for i in range(4):
                  for n in range(4):
                        score += (self.weights[i][n] * self.matrix[i][n])
* monotonicityWeight

            # Penalty for having too little squares available.
            score +=  len(self.get_empty_spots()) * emptySqauresWeight

            # Makes moves that merge the highest tile happen more often.
            score += self.get_highest_tile() * highestTileWeight

            return score - penalty
```

As you can see in the bottom half, the other two heuristics include a penalty for having too little squares available and bonus to the score for having the highest tile possible, encouraging merges.

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 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 __init__(self) -> None:
        self.moves = ['up', 'right', 'down', 'left']

# Returns the best move the AI can take.
def get_best_move(self, matrix, depth):
        score = 0
        bestMove = None

        for i in self.moves:
            newMatrix = matrix.clone()

            # If no move made return to top.
            if (newMatrix.move(i) == False):
                continue

            newScore = self.expectimax(newMatrix, depth - 1, "computer")

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

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):
      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 the move isn't possible, skip the 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()
                  # Testing each empty cell with a value of 4.
                  newMatrix.insert_tile(cells[i], 4)
                  newScore  =  self.expectimax(newMatrix,  depth  -  1,
"player")
                  if (newScore == 0):
                        score += 0
                  else:
# If the score is good, it's added but multiplied by 0.1 as that is the
# chance for a 4 to occur.
                        score += (0.1 * newScore)
                  newMatrix = matrix.clone()
# Testing each empty cell with a value of 2.
                  newMatrix.insert_tile(cells[i], 2)
                  newScore  =  self.expectimax(newMatrix,  depth  -  1,
"player")
                  if (newScore == 0):
                        score += 0
                  else:
# If the score is good, it's added but multiplied by 0.9 as that is the
# chance for a 2 to occur.
                        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.

3.2.4 GUI for the ‘AI plays 2048’ program

The GUI program for the ‘AI plays’ program, does what the name suggests, it gives the game colour and displays it in a Tkinter window. I included this because I wanted to have a way to display the grid in a nice format instead of in the console. The current GUI is not my code (See reference [4] in the GUI file for code) and I have only made small changes to it. I included it now to give me ideas for my future GUI which will have new features not included in the current one. These will include being able to swap between the AI and player versions of the game, include buttons for resetting the game and more.

3.3 Theoretical aspects of the code

3.3.1 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 this 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. Next term, I do plan to split my repository into branches as I will be working on the final product.

3.3.2 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 random spawning of tiles is the reason why Expectimax is an ideal approach when it comes to this game.

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.

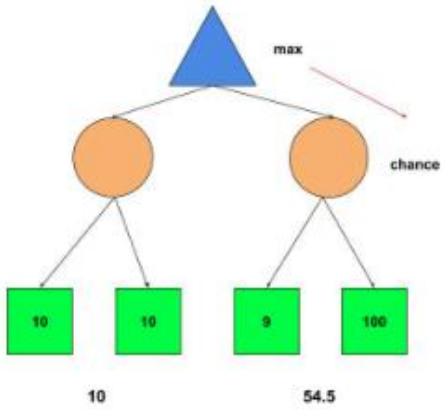


Figure 3. Example of an Expectimax game tree.

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].

3.3.3 UML

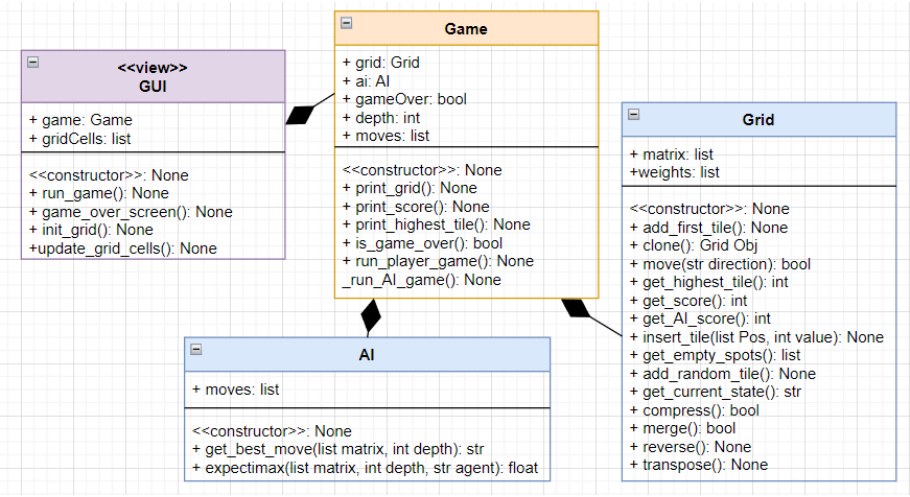


Figure 4. UML diagram of current code structure.

Above is a UML diagram of the structure of the AI code including the GUI (see figure 4).

Chapter 4: **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.

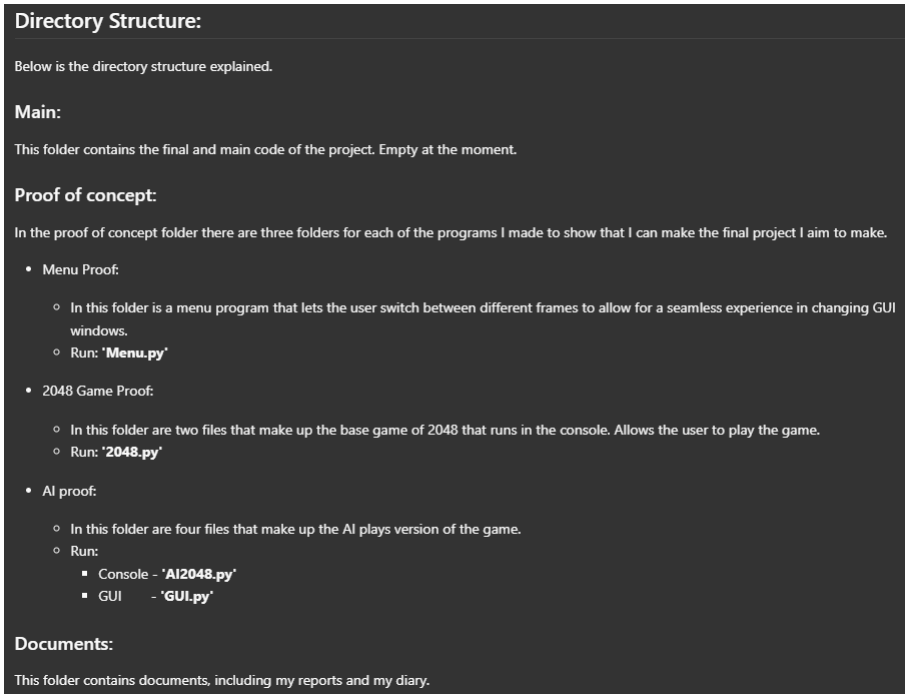
22/11/2022:

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.

Chapter 5: **Running the program**

5.1 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. Empty at the moment.

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.

Figure 5. Directory structure of the code folder.

5.2 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

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/

5.2.1 Numpy installation

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

pip install numpy

5.2.2 Tkinter installation

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

pip install tk

5.3 Code deployment

5.3.1 Screenshots of code deployment

Below are screenshots of my code being deployed.

Menu:

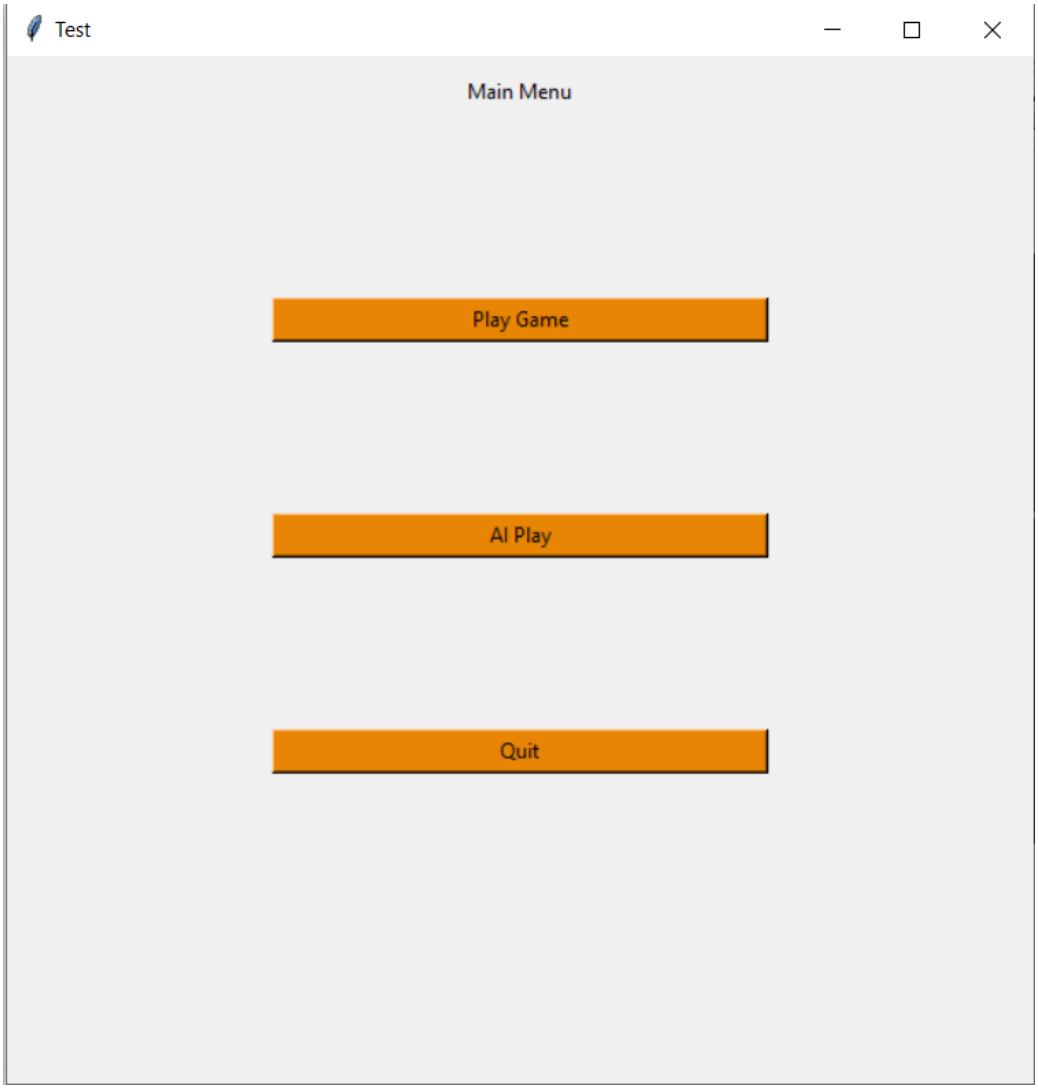


Figure 6. Main menu page with buttons for different pages.

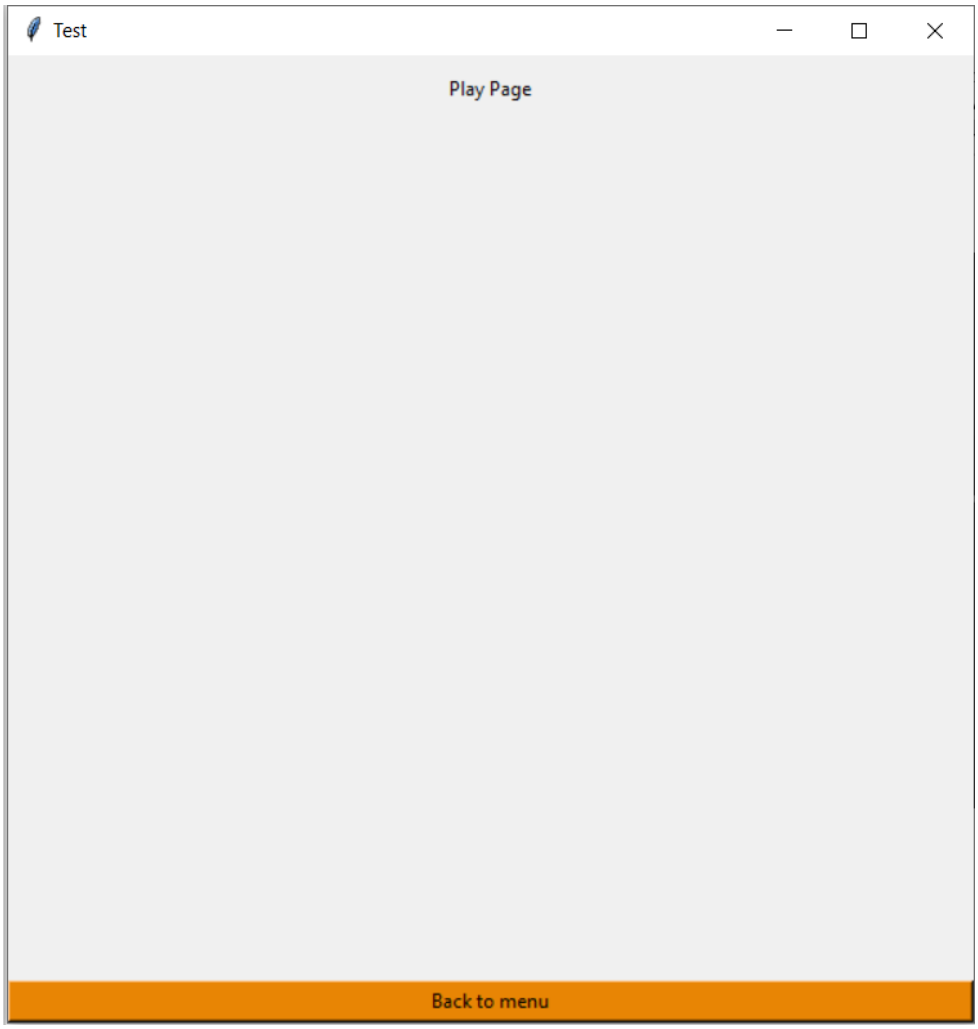


Figure 7. The play page after clicking it on the menu.

Empty but shows the functionality of a menu system.

Player plays version of the game:

```
Player plays 2048.

Controls:
        w: up
        a: left
        s: down
        d: right


Score:  2
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 2, 0]


Enter a move:
```

Figure 8. Player plays version of the game (start of game).

```
Enter a move: w

Score:  4
[0, 0, 2, 0]
[0, 0, 0, 0]
[0, 2, 0, 0]
[0, 0, 0, 0]

Enter a move: a

Score:  8
[2, 0, 0, 0]
[4, 0, 0, 0]
[2, 0, 0, 0]
[0, 0, 0, 0]

Enter a move: d

Score:  10
[2, 0, 0, 2]
[0, 0, 0, 4]
[0, 0, 0, 2]
[0, 0, 0, 0]
```

Figure 9. Player plays version cont.

```
Score:  30
[8, 4, 8, 2]
[0, 0, 0, 4]
[2, 0, 0, 2]
[0, 0, 0, 0]

Enter a move: a

Score:  34
[8, 4, 8, 2]
[4, 4, 0, 0]
[4, 0, 0, 0]
[0, 0, 0, 0]

Enter a move: w

Score:  36
[8, 8, 8, 2]
[8, 0, 0, 0]
[0, 0, 0, 2]
[0, 0, 0, 0]

Enter a move: a

Score:  40
[16, 8, 2, 0]
[8, 4, 0, 0]
[2, 0, 0, 0]
[0, 0, 0, 0]

Enter a move:
```

Figure 10. Player plays version (further in game).

AI plays version:

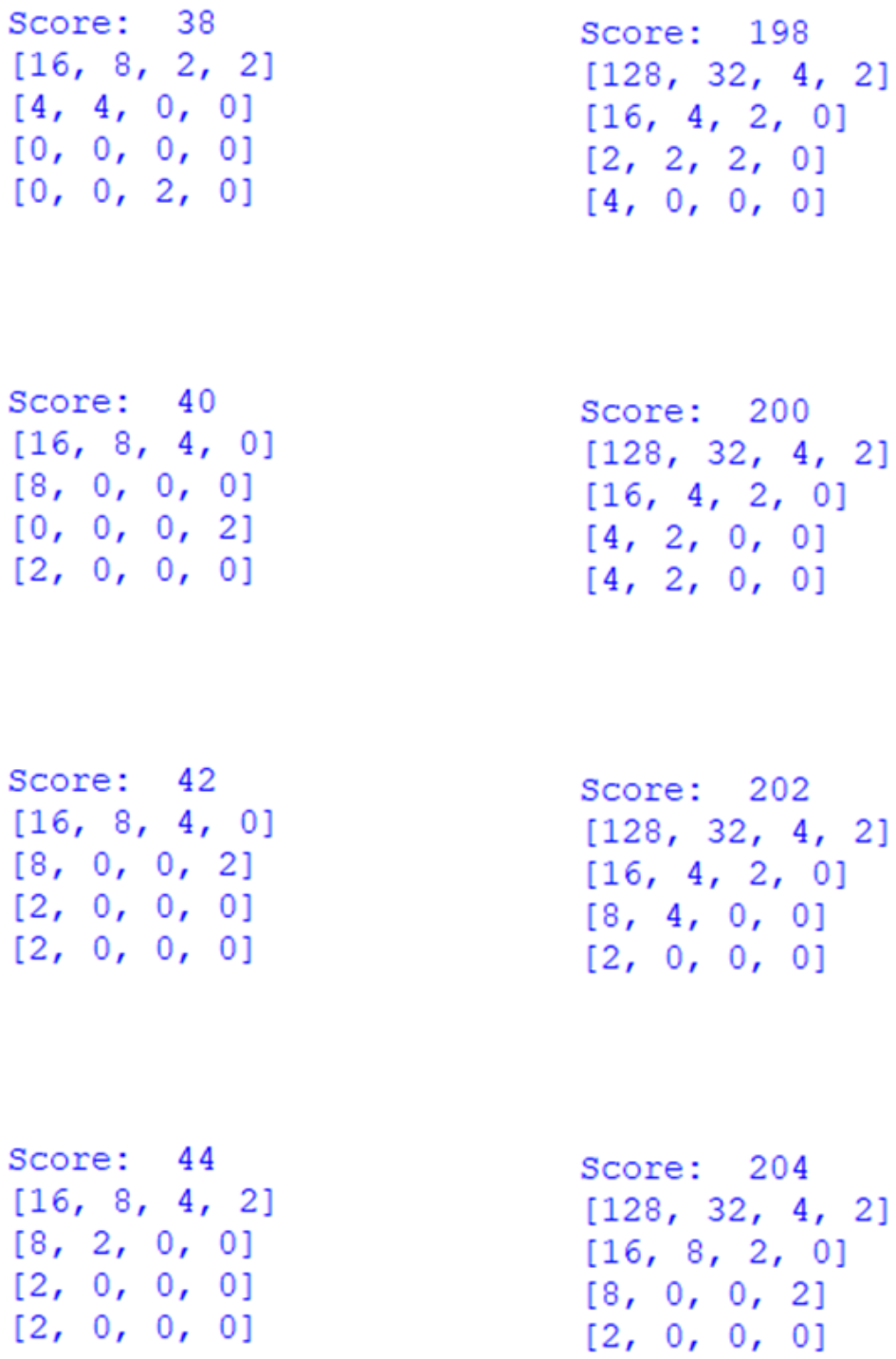


Figure 11. AI plays version of game making moves on its own.

GUI:

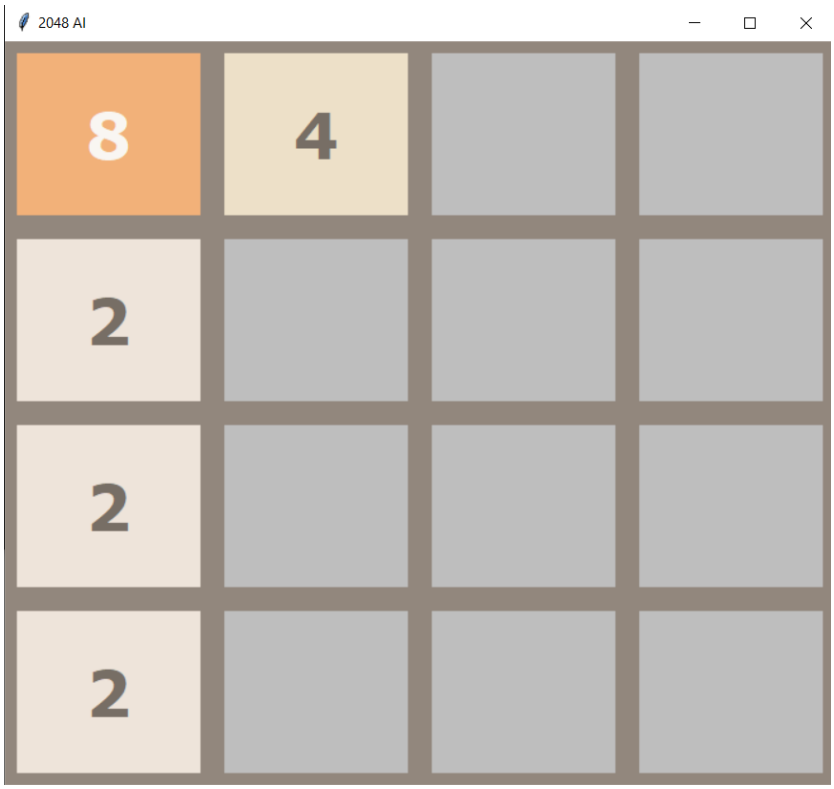


Figure 12. GUI grid showing gameplay.

Figure 13. GUI showing gameplay further in.

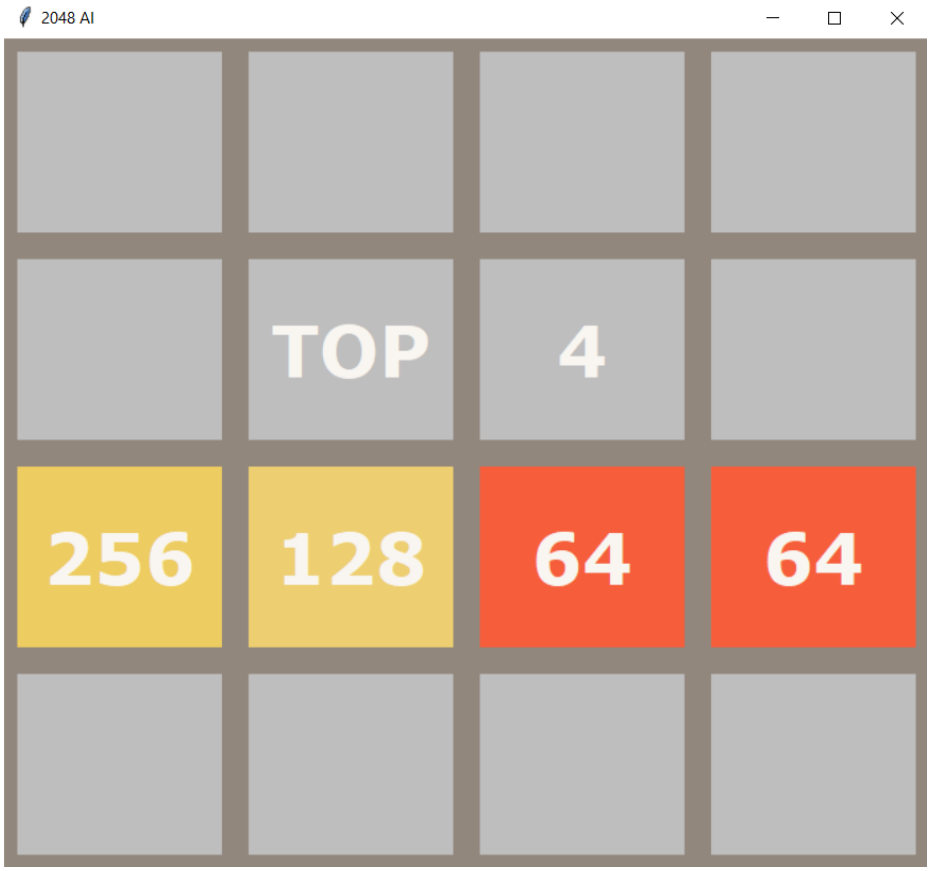


Figure 14. Game over the screen, showing top tiles.

5.3.2 Video of code deployment

Below is a link to a video of the programs running.

https://www.youtube.com/watch?v=tA-tAtZSsB0

Bibliography

[1] https://www.digitalocean.com/community/tutorials/tkinter-working-with-classes

[2] https://www.geeksforgeeks.org/2048-game-in-python/

[3] https://github.com/ovolve/2048-AI

[4] https://github.com/Lesaun/2048-expectimax-ai

[5] https://github.com/Wahab16/2048-Game-Using-Expectimax

[6] https://gitlab.cim.rhul.ac.uk/zjac382/PROJECT

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

[8] https://www.geeksforgeeks.org/expectimax-algorithm-in-game-theory/

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

[10] https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

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

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