BEST MINIMAX ALGORITHM TO MAXIMIZE SCORE IN 2048

CSCI 4511W Final Project Report

AlYaqdhan Al Maawali & Ahmed Al Raisi

December 2022

Abstract

2048 is a puzzle game where it presents a series of tiles on a 4x4 grid, and it allows the player to combine the tiles non diagonally only if the tiles have the same number. This presents a lot of decisions that the player can take that could either contribute to either prolonging the game or getting closer to finishing it. This report explores how high of a score an AI agent can achieve with a certain algorithm, in this case the minimax algorithm, while comparing the score with different heuristics and depth-cutoff within the algorithm. Therefore, technically, if an algorithm is written that addresses all the possible strategies then the agent should achieve really high scores, which would also correlate to a high win percentage of reaching the 2048 tile. This report explores how far a certain heuristic can go when evaluating the decisions of the agent into the game, and possibly compare it with other heuristics.

Introduction to problem

In order to be able to program an agent to solve the 2048, a specific algorithm (which in this case is the Minimax algorithm) is explored with different heuristics while keeping in mind how to optimally take into account strategies that help get a high score in the game. Therefore with regards to the algorithm, this report focuses on implementing the system through a minimax algorithm. This algorithm was chosen because it is usually used on turn-based games with 2 players and that would help us in choosing the next best move in 2048. With regards to the actual game itself, we will need to keep in mind certain strategies that allow us to achieve a high score. Some of the strategies that we may use in our heuristic evaluation functions are:

1. Prioritize having as many empty tiles as possible to delay the game from ending.
2. It would be most optimal to keep the highest tiles in the corners to allow for a greater chance of merging.
3. Maximizing the score of the bordering tiles will allow for a greater score to be achieved.
4. Combining different heuristics with weights depending on their importance to the game.

In terms of the visibility of the game. The environment of the game is a 4 by 4 matrix in which the tiles randomly spawn in empty spaces. Making a move means moving all the tiles in a certain direction, adding up two elements if they have the same value and combining them into one element, so using a 2d array would make sense in implementing this algorithm.

The implementation will take place in python. We will evaluate our solution by comparing the score the AI got with an average score of a simple heuristic that only took into account the score of the tiles and the game. If the AI works, the score of the agent with the heuristics should be similar to our score or much higher. If the AI got a really low score on average, we would know that the heuristic is not optimal or the strategy should not be prioritized.

The report will also test out and compare the code using different heuristics, this would ideally help us achieve higher scores. In addition we will look at the strategies presented to succeed in the game and determine whether prioritizing a certain strategy will be better than others.

Background or Related Work

2048, invented in 2014 by Gabriele Cirulli [2], is an addictive puzzle game available on different platforms and devices. The idea of the game is to slide tiles on a four-by-four grid in four possible directions: up, down, left, and right. The puzzle, however, is different from traditional puzzles as there is no clear optimal path to the goal when looking at the board [7]. The performance of 2048 is measured by a high score system. To score, you need to merge the tiles with the same numbers together so they add up. Two with two to make up four, four with four to make up eight, and so on. The game is won when the tile with the value 2048 appears on the board. However, the player can continue to play and the game terminates when there are no possible moves left i.e. the board is filled with no adjacent tiles of the same value [1]. Over the years, research was conducted on finding the optimal path to solving this game using various search algorithms. This paper will go through some of the possible search algorithms to find the optimal path.

One of the algorithms used to solve this puzzle is the minimax algorithm. According to Pot et al., "The minimax algorithm is a decision rule for minimizing the possible loss for the worst case scenario, or in other words for minimizing the maximum loss" [6]. Zadrozny describes it as "A recursive algorithm which is used to choose an optimal move for a player assuming that the adversary is also playing optimally" [8]. This is used for a two-player zero-sum game, meaning the gain of a player is exactly the same as the loss of the other player, and vice versa [9]. Max, the player that maximizes the score will be the player, and Min, the player that minimizes it will be the game itself. Therefore, we can think of it as the Min player trying to place tiles in the worst possible way for Max, the player [4]. According to Rodgers and Levine [7], depth-limited minimax search was the first published AI for the game 2048, appearing on the internet in March 2014. The Minimax algorithm is expensive in terms of complexity since it requires expanding all of the nodes of the tree. This can be improved by alpha-beta pruning, which is cutting the number of nodes in the search tree by stopping evaluating a move when at least one possibility shows the move is worse than the one examined previously [9]. According to Zaky, the heuristic function used for the algorithm is based on a) empty spaces b) smoothness, which is calculated by the difference of the value of adjacent tiles and c) big tiles in the border [9].

The second search algorithm that research has been done on to solve 2048 is the expectimax algorithm. The expectimax algorithm is similar to the minimax in that it is a recursive, depth-limited tree search algorithm. However, the player doesn't maximize the minimums from the opponent but the maximum of the expected scores of the next states [7]. In addition, "The expected score of a state is the sum of the Expectimax value of each of the next states, multiplied by the probability of that state occurring" [7]. When comparing the minimax algorithm result to the expectimax algorithm, Pot et al. got good performance on both algorithms. Minimax with depth 8 has a success rate of 70% whereas expectimax with depth 3 has a success rate of 80% and a probability of 40% to reach tile 4096 [6]. Zaky's minimax algorithm, on the other hand, with depth limit 8 has a success rate of 75%, while expectimax with depth limit 3 has around 80% winning rate and 40% chance of getting a 4096 tile, similar to Pot et al. research [9]. Both of these papers show the expectimax algorithm beating the minimax algorithm in terms of win rate.

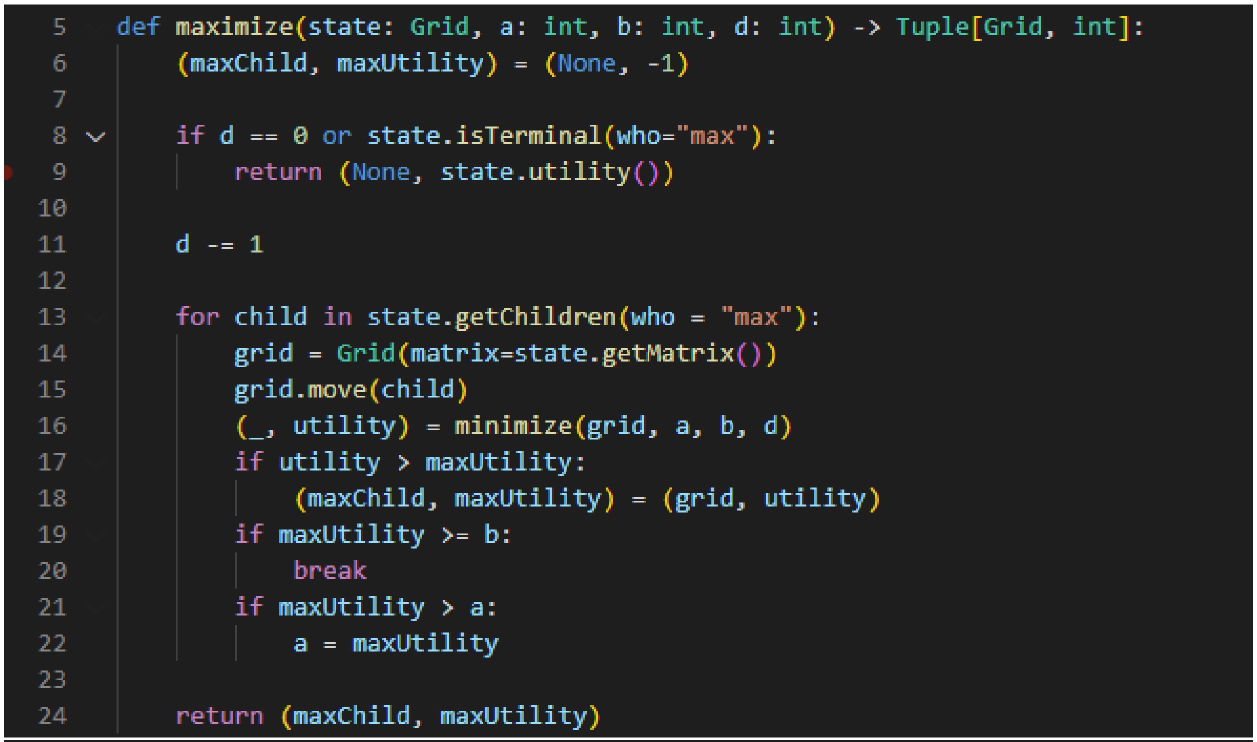
The third way to create an agent for 2048 was creating an N tuple deep conventional neural network trained by reinforcement training. In this study done by Kondo and Matsuzaki [3], they looked into testing out different numbers of layers with the same weight to see how that affects the score achieved by the agent. They limited the layers in the range from 2 to 9 and tried to see the correlation between having a higher amount of layers. Looking at the 5 and 7 layers marks, our best average score was approximately 93,000 for the 5 layers, and 400,000 for the 7 layers mark. The drastic difference in these two layers foreshadows that the hypothesis is true, where training a 2048 agent with a deep conventional neural network will yield the best results at least when compared to other conventional neural networks.

Building on using conventional neural networks, it would also make sense to look at backward temporal coherence learning. Matsuzaki [5] thought testing out this algorithm would make sense since 2048 is a long sequence game, therefore working backward should ideally find a solution to the best sequence of moves to win the game. Through the paper, they try to conduct the experiments both with and without multi-staging and restarting. The results with the greedy approach vs expect max approach really differed in the sense that the greedy approach was more consistent with the score with and without multi-staging and restart, whereas the expectimax algorithm performed worse without the restart than with it. When looking at our project we can look into implementing the restart idea, as it applies to any 2048 game, as the game becomes difficult towards the end, therefore we will look at the position halfway from the end.

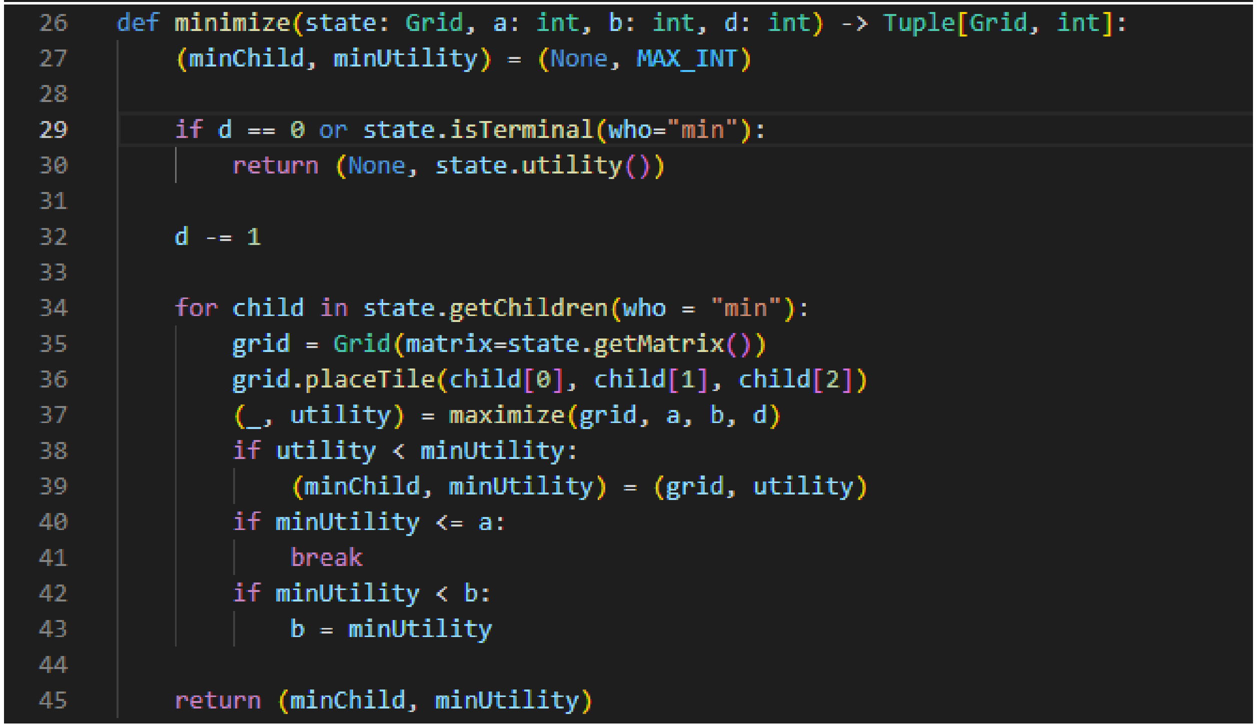
In conclusion, we looked at 4 different algorithms that we can use to implement an agent that solves the 2048 game. We looked into the minimax algorithm and talked about how we can apply this 2 player algorithm to this one-player game. We also looked into alternative algorithms that researchers have used. These included the expectimax algorithm, Monte Carlo search, deep conventional neural networks, and backward learning algorithms. These algorithms differed in complexity, some being out of the scope of this class, but we can see some advantages and disadvantages with each of the algorithms, some of which we can apply to our implementation such as learning with a restart. Overall, this is an interesting field in which a lot of paths have been explored.

Methods and Approach

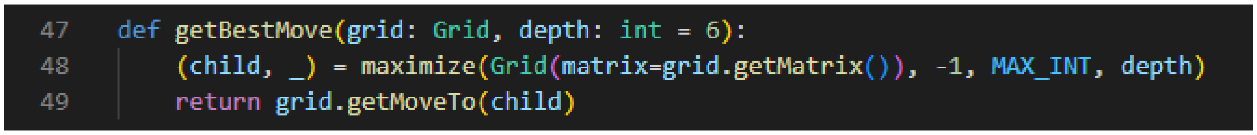
The approach to finding the best way of maximizing the score of the 2048 game is to investigate a minimax algorithm and experiment using different heuristic functions and depth cutoff. The particular minimax algorithm investigated in this research paper is created by Dorian Lazar [4], found at https://github.com/lazuxd/playing-2048-with-minimax, which is a program written in python that uses the selenium library to automate the chrome browser to solve the game.



```python
def maximize(state: Grid, a: int, b: int, d: int) -> Tuple[Grid, int]:
    (maxChild, maxUtility) = (None, -1)

    if d == 0 or state.isTerminal(who="max"):
        return (None, state.utility())

    d -= 1

    for child in state.getChildren(who = "max"):
        grid = Grid(matrix=state.getMatrix())
        grid.move(child)
        (_, utility) = minimize(grid, a, b, d)
        if utility > maxUtility:
            (maxChild, maxUtility) = (grid, utility)
        if maxUtility >= b:
            break
        if maxUtility > a:
            a = maxUtility

    return (maxChild, maxUtility)
```

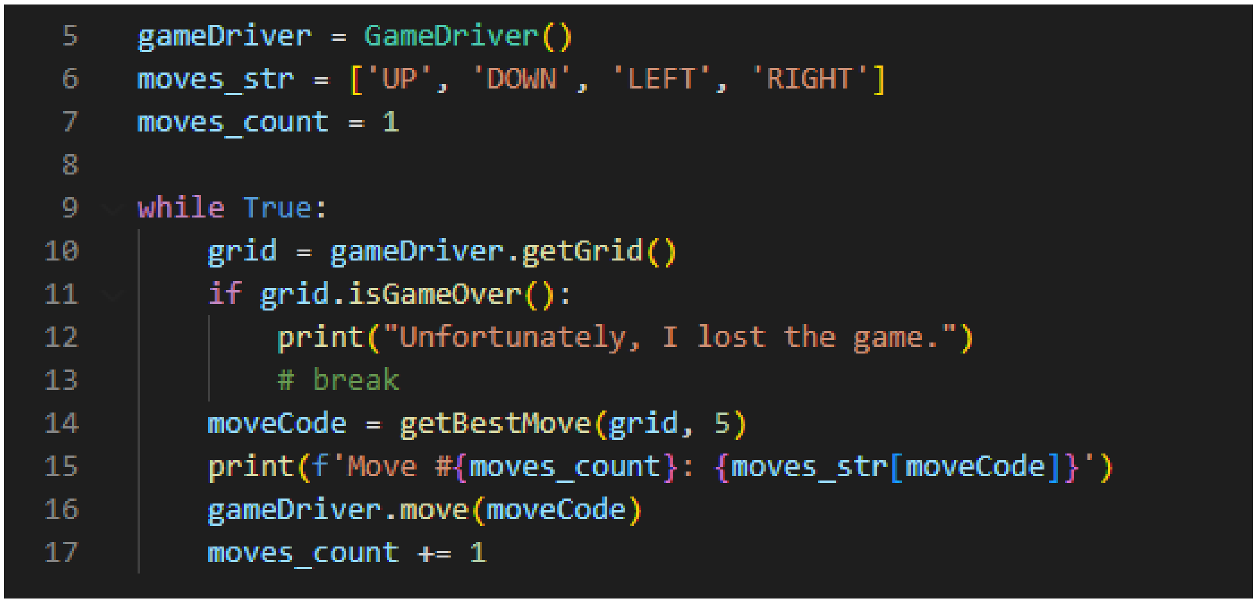


```python
def minimize(state: Grid, a: int, b: int, d: int) -> Tuple[Grid, int]:
    (minChild, minUtility) = (None, MAX_INT)

    if d == 0 or state.isTerminal(who="min"):
        return (None, state.utility())

    d -= 1

    for child in state.getChildren(who = "min"):
        grid = Grid(matrix=state.getMatrix())
        grid.placeTile(child[0], child[1], child[2])
        (_, utility) = maximize(grid, a, b, d)
        if utility < minUtility:
            (minChild, minUtility) = (grid, utility)
        if minUtility <= a:
            break
        if minUtility < b:
            b = minUtility

    return (minChild, minUtility)
```



```python
def getBestMove(grid: Grid, depth: int = 6):
    (child, _) = maximize(Grid(matrix=grid.getMatrix()), -1, MAX_INT, depth)
    return grid.getMoveTo(child)
```

The getBestMove function takes the grid of the game and the depth cutoff and calls the maximize function. The maximize function terminates if the depth cutoff or terminal state is reached, otherwise, it moves to the max child and calls minimize, which does the same thing as maximize except it moves to the min child and calls maximize. This minimax function also uses the benefit of alpha-beta pruning by keeping track of the alpha and beta values to prune nodes that don't need to be investigated. Finally, the getBestMove returns the best move determined by the minimax function, which the game loop uses to solve 2048.



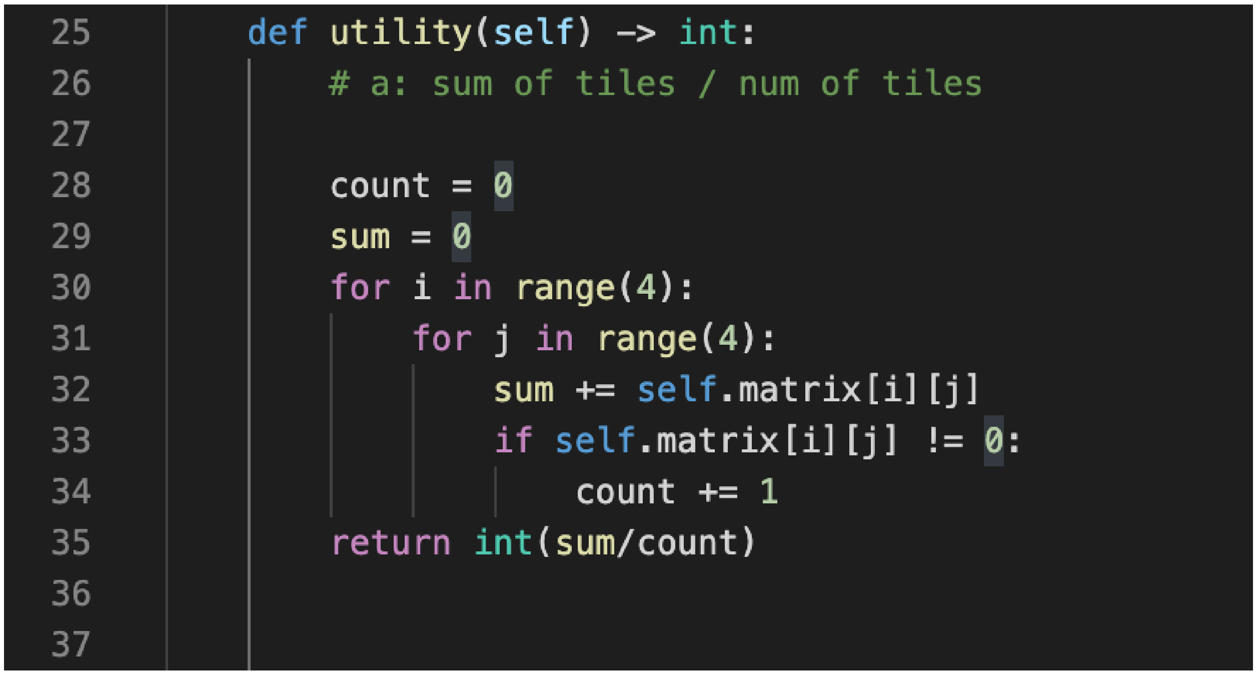
```python
gameDriver = GameDriver()
moves_str = ['UP', 'DOWN', 'LEFT', 'RIGHT']
moves_count = 1

while True:
    grid = gameDriver.getGrid()
    if grid.isGameOver():
        print("Unfortunately, I lost the game.")
        # break
    moveCode = getBestMove(grid, 5)
    print(f'Move #{moves_count}: {moves_str[moveCode]}')
    gameDriver.move(moveCode)
    moves_count += 1
```

The game loop python file runs a loop where it creates the grid, calls the minimax function while passing the grid and the preferred depth-cutoff value to get the best move, then moves and runs the loop again until the game is over where it breaks the loop.

Coming to the heuristic, the method tests the program with 5 different heuristics, each with 3 different depth values. The heuristics are as follows:

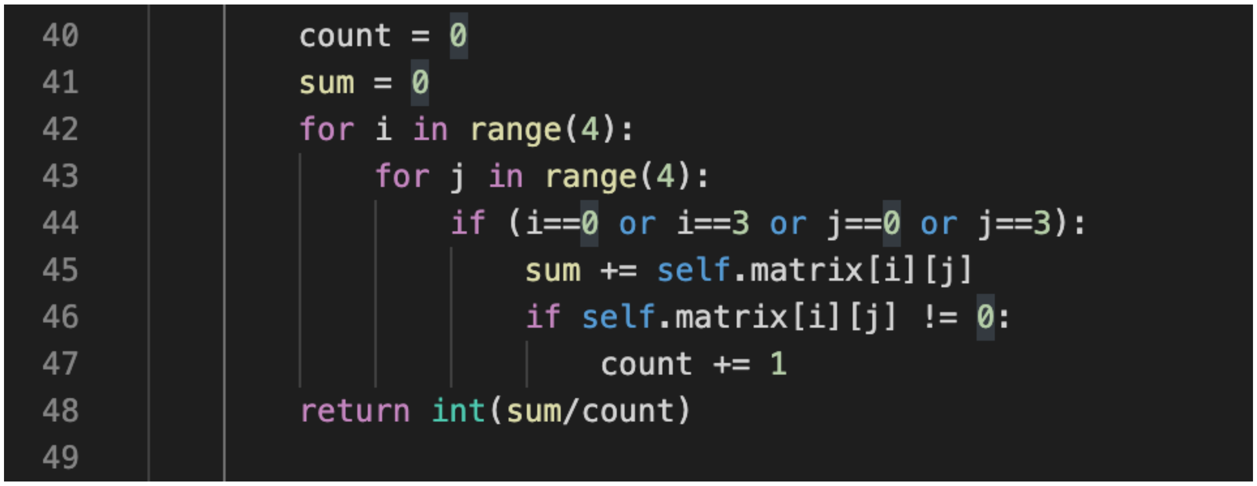
- a. The sum of the tiles divided by the number of non-empty tiles
- b. The sum of tiles in the borders divided by the number of non-empty tiles
- c. The sum of empty tiles multiplied by some constant
- d. The sum of corner tiles divided by the number of non-empty tiles
- e. Heuristics a,b, and d combined together with some weight

The first heuristic, which is the default one written by Lazar [4], loops through the matrix and takes the sum of all the tiles divided by the number of non-empty tiles. The idea is to get the average value of the non-empty tiles. The higher the value, the better the score hence the better the move.



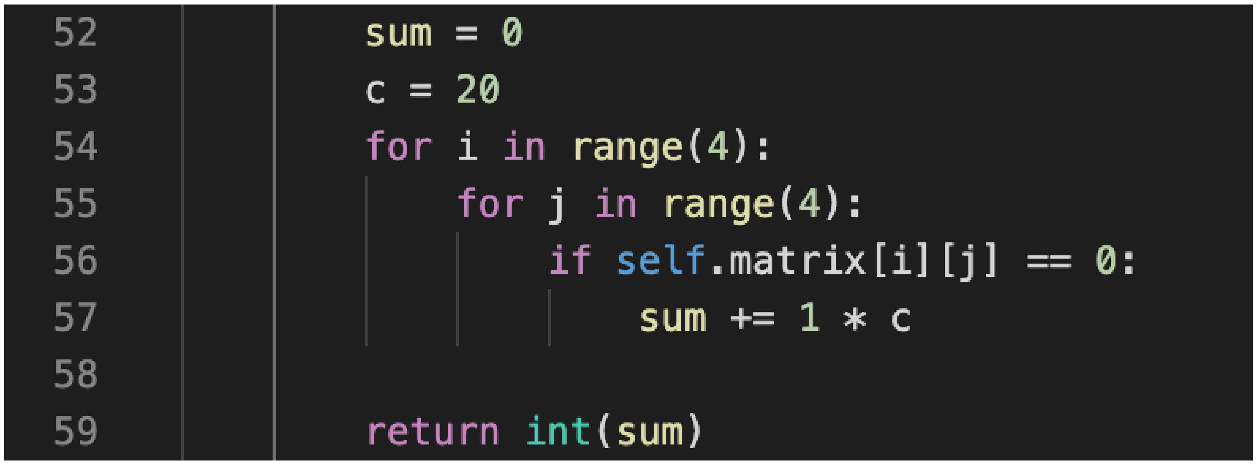
```python
    def utility(self) -> int:
        # a: sum of tiles / num of tiles

        count = 0
        sum = 0
        for i in range(4):
            for j in range(4):
                sum += self.matrix[i][j]
                if self.matrix[i][j] != 0:
                    count += 1
        return int(sum/count)


```

The second heuristic, b, takes the sum of tiles in the borders only and divides them by the number of non-empty tiles. It is similar to heuristic ‘a’ in which you take the average of non-empty tiles but it only does so for the border of the matrix. Since in the game a good strategy is to keep large valued tiles in the border, this heuristic labels that as a good move.



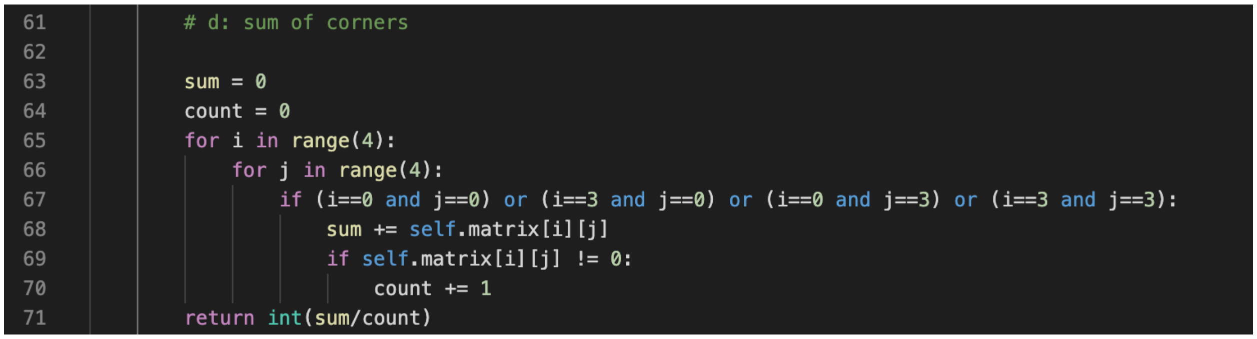
```python
        count = 0
        sum = 0
        for i in range(4):
            for j in range(4):
                if (i==0 or i==3 or j==0 or j==3):
                    sum += self.matrix[i][j]
                    if self.matrix[i][j] != 0:
                        count += 1
        return int(sum/count)

```

Heuristic c looks at empty tiles and sums them together multiplying them by some constant. The idea here is very basic; the more empty tiles you have, the further you are from filling the board and losing the game.



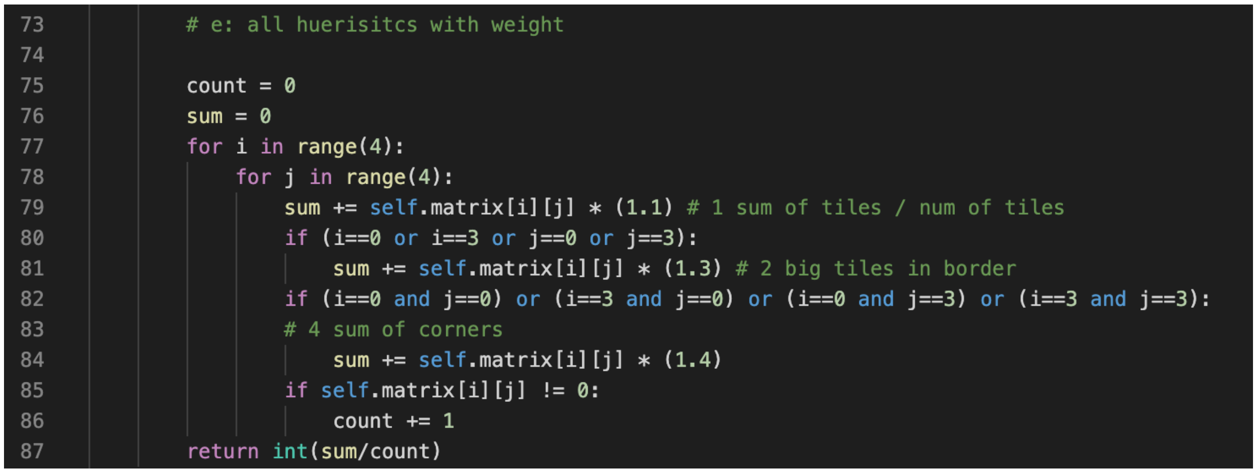
```
        sum = 0
        c = 20
        for i in range(4):
            for j in range(4):
                if self.matrix[i][j] == 0:
                    sum += 1 * c

        return int(sum)
```

Heuristic d has a similar idea to heuristic c but it only takes account of tiles in the corner of the matrix. It loops through the matrix, takes the sum of corner tiles, and divides them by the number of non-empty tiles



```
        # d: sum of corners

        sum = 0
        count = 0
        for i in range(4):
            for j in range(4):
                if (i==0 and j==0) or (i==3 and j==0) or (i==0 and j==3) or (i==3 and j==3):
                    sum += self.matrix[i][j]
                    if self.matrix[i][j] != 0:
                        count += 1
        return int(sum/count)
```

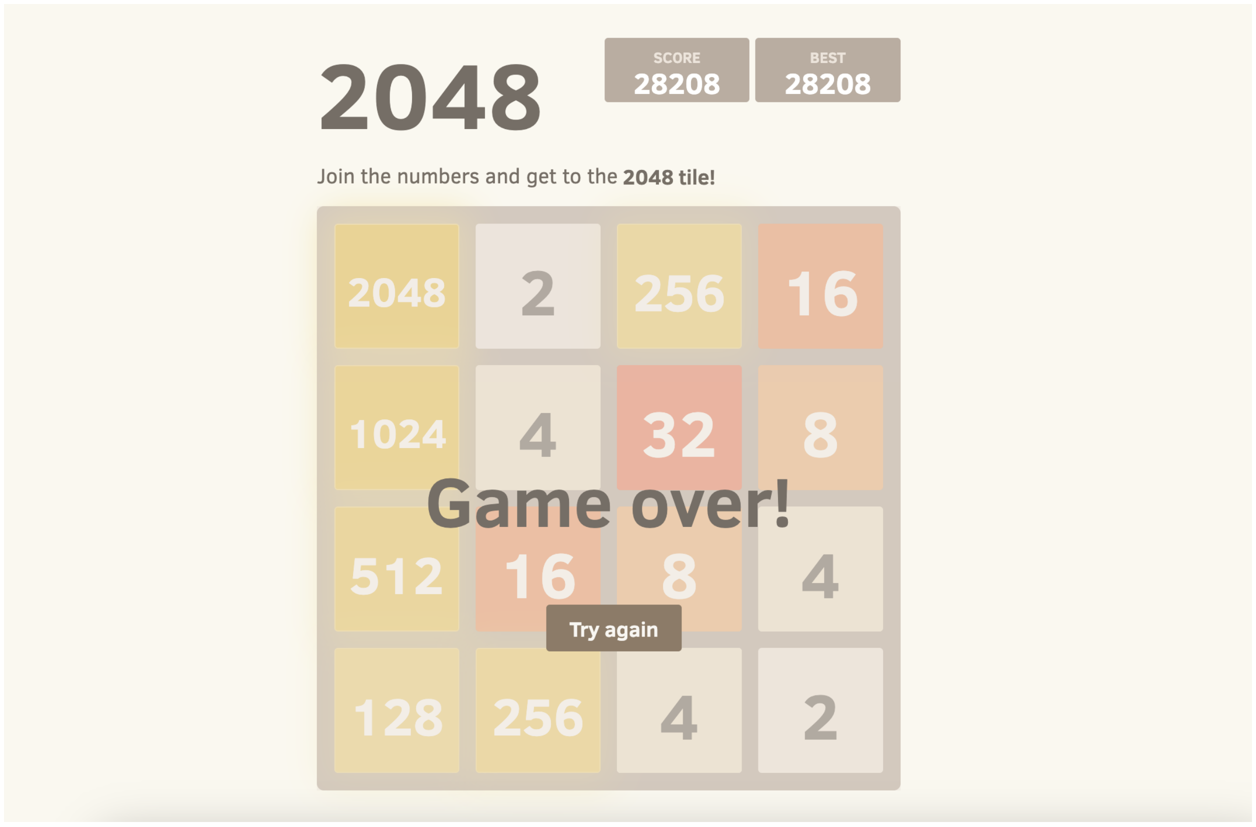
The last heuristic, labeling it as the "ultimate heuristic", combines what heuristics a, b, and d do in one heuristic multiplying each result by some weight. The idea is to consider most strategies of the game and give a larger weight to heuristics with a better strategy, while still including other heuristics with a mediocre strategy and giving them a smaller weight. Heuristic 'a' is multiplied by 1.1, 'b' is multiplied by 1.3, and 'd' is multiplied by 1.4.



```
# e: all huerisitcs with weight

count = 0
sum = 0
for i in range(4):
    for j in range(4):
        sum += self.matrix[i][j] * (1.1) # 1 sum of tiles / num of tiles
        if (i==0 or i==3 or j==0 or j==3):
            sum += self.matrix[i][j] * (1.3) # 2 big tiles in border
        if (i==0 and j==0) or (i==3 and j==0) or (i==0 and j==3) or (i==3 and j==3):
        # 4 sum of corners
            sum += self.matrix[i][j] * (1.4)
        if self.matrix[i][j] != 0:
            count += 1
return int(sum/count)
```

Experiment and Results

The experiment is to run the program with each of the five heuristics using 3 different depth cutoff values. Depth 3, 4, and 5. For each individual heuristic and depth, the program is executed 3 times and the scores are recorded in a table. The average score of each heuristic is also recorded in a table.



The results of the experiment show that for heuristic ‘a’, the worst score is 5,740 which occurred at depth 3 and the best score is 63,004 which occurred at depth 5. For heuristic ‘b’, the worst score is 9,236 which occurred at depth 4 and the best score is 72,820 which occurred at depth 5. For heuristic ‘c’, the worst score is 10,560 which occurred at depth 4 and the best score is 34,896 which occurred at depth 5. For heuristic ‘d’, the worst score is 1,336 which occurred at depth 3 and the best score is 14,088 which occurred at depth 5. Lastly, for heuristic ‘e’, the worst score is 31,080 which occurred at depth 3 and the best score is 68,268 which occurred at depth 5.

Scores of each heuristic with different depths						
		Heuristic				
		a	b	c	d	e
Depth	3	30,496	16,872	22,384	12,846	33,612
		26,908	55,140	14,568	6,492	45,984
		5,740	48,032	20,836	1,336	31,080
	4	34,636	39,804	30,092	3,964	38,540
		54,712	9,236	10,560	2,328	62,120
		22,880	57,952	32,554	13,432	56,924
	5	44,200	53,136	15,375	2,316	68,268
		63,004	70,972	34,896	6,360	42,812
		11,060	72,820	24,798	14,088	63,924

On average, the scores were the best with depth 5 for all heuristics and the worse with depth 3 for all heuristics except heuristics ‘b’ and ‘d’ which performed worse with depth 4 with a very slight difference for heuristic ‘d’. For all the depths, the average score for heuristics a, b, c, d, and e is 32,626, 47,107, 22,896, 7,018, and 49,252 respectively. The order for the best average score for all depths to worse is as follows, with heuristic ‘e’ being best and heuristic ‘d’ being worse:

1. A: The sum of the tiles divided by the number of non-empty tiles
2. B: The sum of tiles in the borders divided by the number of non-empty tiles
3. C: The sum of empty tiles multiplied by some constant
4. D: The sum of corner tiles divided by the number of non-empty tiles
5. E: Heuristics a,b, and d combined together with some weight

Average scores of each heuristic with different depths						
		Heuristic				
		a	b	c	d	e
Depth	3	21,048	40,015	19,263	6,891	36,892
	4	37,409	35,664	24,402	6,575	52,528
	5	39,421	65,643	25,023	7,588	58,335
Average of all depths		32,626	47,107	22,896	7,018	49,252

Analysis

Looking at the results initially, The heuristics and the depth did indeed have a big effect on the outcome of the decisions and score of the minimax agent during the game. Starting with the most simple heuristic, a which takes into account the sum of the tiles and divides it the amount of non empty tiles, This led to an average score of 32,000 which compared to the other heuristics was in the middle. Therefore this heuristic would be a great point to compare other heuristics with in order to see if they are "good" or "bad" relative to one another.

These heuristics were mainly based on different tactics that are often used to maximize the score of the 2048 game normally, and based on their score we can define how effective the tactic is, not only for the agent, but also for humans, as in the game there are often times where you would need to favor one tactic over another when making a decision.

Starting with heuristic b, on average it would score around 15,000 points higher than the heuristic a which tells us that the tactic is indeed more effective if prioritized over having a higher number of tiles in the board. Recalling the idea behind heuristic b which was to sum up the tiles in the borders, assuming that having a the higher valued tiles in the border will lead to a higher score. Which would make sense as it allows for the smaller tiles to have more freedom to merge with the newer spawning tiles where the bigger tiles remain in the border and wait for similar valued tiles to be merged with them. This is clearly and effective technique which led to very high scores, even glitching the game sometimes and reaching scores within the millions. We didn't take those scores into consideration as they were outliers and just a glitch in the game.

Heuristic c on the other hand scores almost 10,000 points lower on average than heuristic a, which is surprising. The heuristic focuses only on the number of empty tiles in the board, and tries to minimise them by taking the decisions that would lead to that. The hypothesis behind this heuristic is that in order not to lose in the game 2048 you have to not let all the tiles get filled up, and as a result the understanding was that the more empty tiles you have the better chance the player has of winning. On paper, this is understandable, but through the scores that we have achieved we can notice that the game does not only boil down to this, and that other tactics are more effective and should sometimes be prioritized.

Heuristic d was the worst performing heuristic, as it scored almost 15,000 less on average than heuristic a. The heuristic focuses on prioritizing the biggest tiles in the corner of the grid. This was suggested by multiple sources as a highly effective tactic in order to be able to achieve the highest scores within 2048. From reviewing the game play and comparing it to the high tiles in the border agent, we can see that prioritizing the corners is not an effective method, as the small tiles sometimes get to the borders, and that is something that fills up tiles, and makes them be away from newly spawning tiles, this as a result caused the game to not even win (i.e. reach the 2048 tile), as this was happening early in the game. Therefore we can conclude that prioritizing the corners with high tiles only is not an effective strategy, as the whole borders should be taken into consideration.

On the other hand, heuristic 'e' was the highest achieving heuristic which was expected.

The idea behind this strategy was to mix the heuristics that we focused on, and we add a weight to each heuristic based on how important we think the heuristic is. The first heuristic used was was the sum of tiles divided by the number of non empty tiles and as it was a basic heuristic we weighted it with the constant 1.1, on the other hand, we bordering and corner heuristic were also taken into account with even higher weights as the tiles being in the border leads to higher scores, and the highest tiles in the corner supporting that. Surprisingly, when these two heuristics were combines together, we can see that the corner tiles were holding the largest values with large values in the bordering tiles, this led to a very high average of 49,252, which showed us that when the heuristics are working together instead of independently they can lead to very high score, and more often than not, breaking the game and reaching scores within the millions.

As expected, the depths of the minimax search did indeed affect the scores. As a refresher the depth is how many plys the heuristic considers when calculating the expected value. Throughout all the heuristics we can see a trend where the the larger the depth is the higher the score the agent achieved. This was predicted as the game would be more precise of the decisions that it made, and that it ensures it would make the most optimal decisions to gain the highest scores possible.

Conclusion

In conclusion, 2048 is an interesting game which has an environment of a 4x4 grid, there are tiles that spawn randomly in empty spaces and each tile with the same value could be combined to double the value into a single tile. The objective of the game is to reach the 2048 tile. Through this project, an agent was explored with different heuristics that targets different optimal play style techniques to see which strategies could be more useful than others. The heuristics we explored were:

- A: The sum of the tiles divided by the number of non-empty tiles
- B: The sum of tiles in the borders divided by the number of non-empty tiles
- C: The sum of empty tiles multiplied by some constant
- D: The sum of corner tiles divided by the number of non-empty tiles
- E: Heuristics a,b, and d combined together with some weight

We found that these heuristics do make a difference in the agents play style, as some when prioritized have a better effect on the score, whereas other had a more negative effect. When combining the tactics together into one heuristic, we found that that yielded the best score. Overall, the experiment can be improved by taking more scores into account for the averages, and testing out multiple ways to implement the same tactics to see if that makes a difference. Limitations should also be considered where the higher the score of the game, the harder the game becomes, but also that the score exponentially increases as the tile values double in size every time. Other search algorithms could also be tested with the same heuristics to see how differently they react to each of them.

Contributions of each member

- Ahmed Al Raisi: Analysis and Conclusion
- AlYaqdhan Al Maawali: Methods and Approach, and Experiment and Results
- Shared: Abstract, Introduction, Background, and Source Code

References

[1] Gayas Chowdhury and Vignesh Dhamodaran. 2048 using expectimax.

[2] Iris Kohler, Theresa Migler, and Foaad Khosmood. Composition of basic heuristics for the game 2048. *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019.

[3] Naoki Kondo and Kiminori Matsuzaki. Playing game 2048 with deep convolutional neural networks trained by supervised learning. *Journal of Information Processing*, 27:340–347, 2019.

[4] Dorian Lazar. How to apply minimax to 2048, May 2021.

[5] Kiminori Matsuzaki. Developing a 2048 player with backward temporal coherence learning and restart. *Lecture Notes in Computer Science*, page 176–187, 2017.

[6] Miklos Pot, Milovan Milivojevic, and Srdan Obradovic.

[7] Philip Rodgers and John Levine. An investigation into 2048 ai strategies. *2014 IEEE Conference on Computational Intelligence and Games*, 2014.

[8] Bartosz Zadrozny. Beginner's guide to ai and writing your own bot for the 2048 game, Nov 2018.

[9] Ahmad Zaky. Minimax and expectimax algorithm to solve 2048. 2022.