CSC111 Project Report: Designing an AI and Strategy Feedback System for Othello

Chun Yin Yan, Gabriel Pais

April 17, 2021

Problem Description and Research Question

Project Goal: How can we create an AI and a strategy feedback system for Othello?

Othello is a board game played with 32 dark pieces and 32 light pieces on a board consisting of 64 squares in a 8×8 grid, like a chess board. The goal of Othello is to fill the board and capture the pieces of the opponent such that when the game ends, the one with the most pieces on the board wins.

Othello begins by having two dark pieces and white pieces placed diagonally in the middle of the board as default. Dark pieces always go first. To place a piece legally, the new piece must sandwich the opponent piece(s) between your piece(s), and the opponent piece(s) will be flipped such that they become your existing pieces (same color as yours). Note that you can sandwich opponent pieces in different rows, columns and diagonals with one single piece in one legal move. After a piece is placed, the turn is over and the next player plays. If the current player has no legal moves, the next player plays instead.

The game ends when the board is fully filled or both players run out of moves. The winner is declared as the player with the most pieces of his color on the board. If the number of dark and white pieces are the same, then the game is a draw.

Othello is the modern version of Reversi, which was invented in the late nineteenth century[1]. It is estimated that there are at most 10^{28} legal positions in Othello[2]. Othello is not as popular as other board games, such as chess and checkers, so there are not many Othello game engines. We are motivated to design an AI and strategy feedback system for Othello since we played this game very often when we were in our childhood, and it is also a very challenging game like chess: it requires creativity and a lot of thinking to win this game. Moreover, we haven't come across any feedback system for Othello online, unlike other board games like Chess and Checkers, which have many advanced game engines that can provide feedback to the players. These are the reasons that motivated us to begin this project.

Our goal is to create AIs with varying difficulties so human players with different experience can play comfortably with these AIs. Moreover, We also want to create a strategy feedback system which can provide feedback on the human player on which moves made in the game are considered excellent or poor. We hope that this feedback system can help player learn from their mistakes and let them become better Othello players.

[1]CYNINGSTAN (Ed.). (n.d.). Reversi. Retrieved March 06, 2021, from http://www.cyningstan.com/game/73/reversi
[2]Allis, L. V. (1994). Searching for solutions in games and artificial intelligence. Wageningen: Ponsen amp; Looijen.

Computational Overview

Major data types:

To create an Othello game, we created several custom dataclasses. These include GameTree, Othello, GameEngine, and Player.

GameTree is a dataclass which the computer (a Player dataclass) uses to determine which move it should make during playing Othello. It also serves as a suggestion tree for players to use during playing the game. The GameTree contains a root, which contains the previous move; a list of subtrees, which are all GameTree objects and each contains a valid move given the current game state; and a score which corresponds to how advantageous the move is to the player choosing.

Othello is a dataclass which controls how the game runs. It is the game board for Othello, and administrate different players. Using Othello, player can get valid moves, make moves and get the current scores of the game. Othello is important to restrict illegal moves and ensure no illegal moves.

GameEngine is a dataclass which allows quick simulation of Othello matches between two computer players of different difficulties. Users of this function can first create two computer players, then call functions in GameEngine to simulate many matches between them. The win-loss-draw ratio are calculated for each computer player. This dataclass is useful because it allows us to calibrate the difficulties of the computer players.

Player is a dataclass which represents a computer player. Each computer player has its own GameTree, which the player uses to make smart decisions; different depths, which controls how big its GameTree should be to balance efficiency and accuracy; and a value which is the chance the player makes random moves during a game. All computer players are RandomPlayer or SmartPlayer, which are child classes of Player.

The dataclasses are the major data types in this project and they work together to create the Othello game engine, computer Othello players and the Othello strategy feedback system.

Major computations:

Major Computation 1: Building GameTrees

There are two types of GameTree. One type of GameTree is generated using the method of the minimax algorithm. This algorithm generates the full-sized GameTree containing every possible paths up to a certain depth. It first calculates all the possible paths of moves using recursion, and build the tree. Then it calculates the score of all the roots, and calculate the maximum or minimum score depending which color it plays. A larger positive score favours white, while a larger negative score favours black. This is the minimax algorithm: if the current player is white, then it will calculate the maximum score among its subtree and the root will be assigned this maximum score. Similar calculations are done for black. Thus, for each node in the tree, we can know the advantage of making that move of the node. An example of such gametree can be found in the folder where all the files are located, after running main.py, in a .txt file named fulltree_depth_5.txt.

The other type of GameTree is generated using the method of minimax algorithm and alpha-beta pruning. This is very similar with the first type of GameTree, except the alpha-beta pruning algorithm eliminates unfavourable subtrees during the building of the whole GameTree. The variable alpha stores the score of the best move so far for white (most positive score), and the variable beta stores the score of the best move so far for black (most negative score). When we calculate that the subpaths are proven to be disadvantageous for the other player, this means that it is very likely they won't choose those subpaths. Thus, the subpaths are not calculated and not included in the GameTree. The alpha and beta variables serve as cutoffs to determine if furthur paths are needed in the GameTree. Using the alpha-beta pruning algorithm can greatly improve the efficiency of GameTree building because paths that are unlikely to be played are not evaluated or included in the GameTree. An example of such gametree can be found in the folder where all the files are located, after running main.py, in a .txt file named prunedtree_depth_5.txt.

Major Computation 2: Computer Players

Computer players are all Player classes. They have their own GameTree and can find the best move from the GameTree according to the current game state. They can also update their own GameTree and make educated guesses about the opponent's move. There are several attributes that affect the decision making of the computer players: color, normal_depth, cutoff, cutoff_depth and rnd. color is the color that the player is playing, which affects which score (maximum or minimum) it wants and hence what moves it picks; normal_depth is the depth of the GameTree in the first part of the game. The second part of the game, as indicated by the value of cutoff (the total number of pieces on the board), the GameTree will have a depth of the value of cutoff_depth. Since the end of the game is much more important than the beginning, the value of cutoff_depth is higher than normal_depth, so the computer player will generate a GameTree of greater depth when cutoff is reached. This allows the computer player to find all possible paths until the end of the game, and make the best possible move. Finally, rnd is the value between 0 and 1 which indicates the chance of it making a random move during its turn. This creates unpredictability and makes Othello more exciting.

Major Computation 3: Visualizing Othello

To visualize Othello, we created the Othello class which contains all functions related to the game. For this part we simply integrated all the functions into our UI, this include functions such as draw_game_state which draws the current state of the game board, get_valid_moves which returns all valid moves for the current player, make_move function that makes the move for the current colour and returns the positions of pieces captured, possible_moves_white that returns all possible moves for the white player from valid_moves, possible_move_black which does the same as the function described above, and also as a last the flip_white/black pieces that flips the pieces on the board. All these functions have been integrated into our pygame interface.

Major Computation 4: Strategy feedback system

The strategy feedback system can only be accessed when the user of this program play against a computer player in the pygame window. The strategy feedback system comes with a HINT button, which allows the user to have a sneak peek of the current best move of the game state. The strategy feedback system records the score of the game at every turn and the accuracy of each move. The score is just the number of white pieces and black pieces on the board.

The accuracy is calculated by the following method:
First, get the moves from the GameTree of the cpu player. Next, from those moves, get the move with the highest score, H, and the move with the lowest score, L, according to the game state. Then, when a player or the computer makes a move, that move will have a score (which is based on the GameTree), and we compare how good this score is. For white players, we want the score of the move, M, to be as high as possible. Thus, the equation for accuracy if the current player is white is

$$\frac{|M-L|}{|H-L|},$$

which is basically the ratio of the distance of the move score from the lowest possible score to the maximum range of score possible; for black players, we want the score of the move, M to be as low as possible. the equation for accuracy if the current player is black is

$$\frac{|M-H|}{|H-L|},$$

which is basically the ratio of the distance of the move score from the highest possible score to the maximum range of score possible.

If the move is not found in the GameTree of the cpu player, then the move will have a score of 0.0.

Major Computation 5: Game Engine

Game Engine is a console-based simulation. It takes in two different computer players (Player) and simulates many matches between those players. It returns the win-loss ratio of each player so that we can know how well they play. This function is used to make and calibrate several computer players of difficulty for the Interactive application.

Libraries and functions used:

The external library used is pygame.

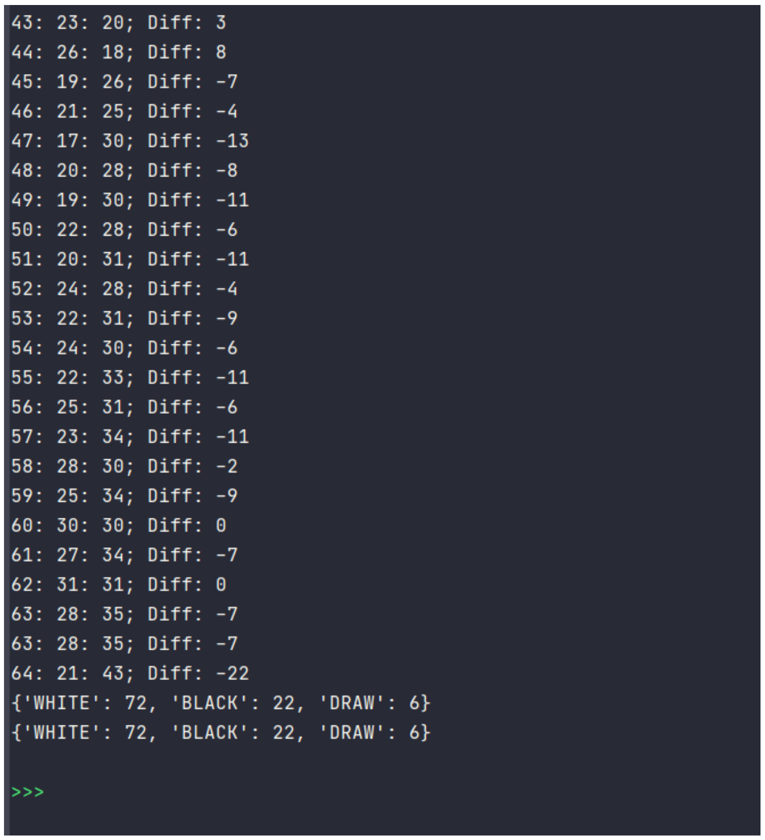
There following are the main functions of this project:

Useful functions	Location	Purpose
Dataclass: Player		
cpu_make_move	GameEngine_AI.py	Returns the best move from its GameTree
play	GameEngine_AI.py	Simulates an Othello game between two computer
Dataclass: GameTree		
find_subtree_by_move	GameTree_Game.py	Returns the subtree which has the move searched
calculate_score	GameTree_Game.py	Calculates the score of the move
generate_moves_full	GameTree_Game.py	Calls minimax to generate a complete GameTree up to a depth
generate_moves_quick	GameTree_Game.py	Calls minimax to generate a pruned GameTree
fulltree_to_text	GameTree_Game.py	Creates a .txt file which contains a full GameTree
quicktree_to_text	GameTree_Game.py	Creates a .txt file which contains a pruned GameTree
Dataclass: Othello		
draw_game_state	Othello_Game.py	Draws the current game board state using pygame
get_valid_moves_now	Othello_Game.py	Returns a list of valid moves
make_move	Othello_Game.py	Makes move on the game board
score	Othello_Game.py	Returns a tuple which shows the current score
get_winner	Othello_Game.py	Returns the winner of the game
Visualization		
main_menu	Visualization.py	Starts the Othello game app
Button (dataclass)	Visualization.py	Creates a button

How the computations are reported via an interactive application:

The computations can be found in the Interactive application. From there, users can play Othello against computer players of different difficulties, as well as try out the hint button and match strategy feedback system.

To access how well the computer play against each other, users can call the ai_test in main.py and run simulations of Othello matches between two computer players, our function runs 100 simulation between 2 chosen AI Players. Different players have different difficulties, so their win-loss ratio against one another will be different: the computer players with higher difficulties will win more than the ones with lower difficulties. For instance in the example below (Figure2), by running ai_test(Professional_W, Beginner_B) we get 100 simulations run in python console between a Professional AI Player and a beginner (note that to run all 100 simulation we would have to wait around minutes), and we get a result of 72 wins for White, 22 for Black and 6 draws. So overall the Professional Player has a higher win rate as expected. In Figure 2, the first numbers represent the move number in that particular game (i.e 59, 60...). The second set of numbers(25:34, 30:30...) represents the number of white and black pieces on the board which most recent move changed, and so the first number shows the number of white pieces and the second one shows the number of black pieces. And as a last, diff which stands for difference tells us the difference between the white pieces and the black ones. If the score is positive that means white has more pieces on the board than black, and if the diff is negative that means black is winning and has more pieces on the board than white.



```
43: 23: 20; Diff: 3
44: 26: 18; Diff: 8
45: 19: 26; Diff: -7
46: 21: 25; Diff: -4
47: 17: 30; Diff: -13
48: 20: 28; Diff: -8
49: 19: 30; Diff: -11
50: 22: 28; Diff: -6
51: 20: 31; Diff: -11
52: 24: 28; Diff: -4
53: 22: 31; Diff: -9
54: 24: 30; Diff: -6
55: 22: 33; Diff: -11
56: 25: 31; Diff: -6
57: 23: 34; Diff: -11
58: 28: 30; Diff: -2
59: 25: 34; Diff: -9
60: 30: 30; Diff: 0
61: 27: 34; Diff: -7
62: 31: 31; Diff: 0
63: 28: 35; Diff: -7
63: 28: 35; Diff: -7
64: 21: 43; Diff: -22
{'WHITE': 72, 'BLACK': 22, 'DRAW': 6}
{'WHITE': 72, 'BLACK': 22, 'DRAW': 6}

>>>
```

Figure 1: Console Output for Professional vs Beginner

Features of Othello Game Application

- Play Othello!
- Compete against AIs of different difficulties! Can you beat the Impossible AI?
- Improve your gameplay using the Strategy Feedback System!
- Stuck on a move? Try using the hint button to find the best move!

Instructions

1. Download all the required files (.py files).
2. Install Python 3.9 and install all Python libraries listed under a requirements.txt file.
3. Make sure you have installed pygame. (You should have done this in step 2.)
   a. if you haven't, go to the terminal in PyCharm and enter the following, "pip install pygame".
4. Move all the files into one folder. Or used the folder downloaded directly.
5. Run the file main.py.
6. Two new files should be created in the same folder where main.py is located: fulltree_depth_5.txt and prunedtree_depth_5.txt. Open these files and compare the difference.

   - Note that the tree in prunedtree_depth_5.txt is much simpler than the one in fulltree_depth_5.txt, although the score of each subtrees of the root are the same in each tree.

- This means that the pruned tree is more efficient at evaluating future moves.

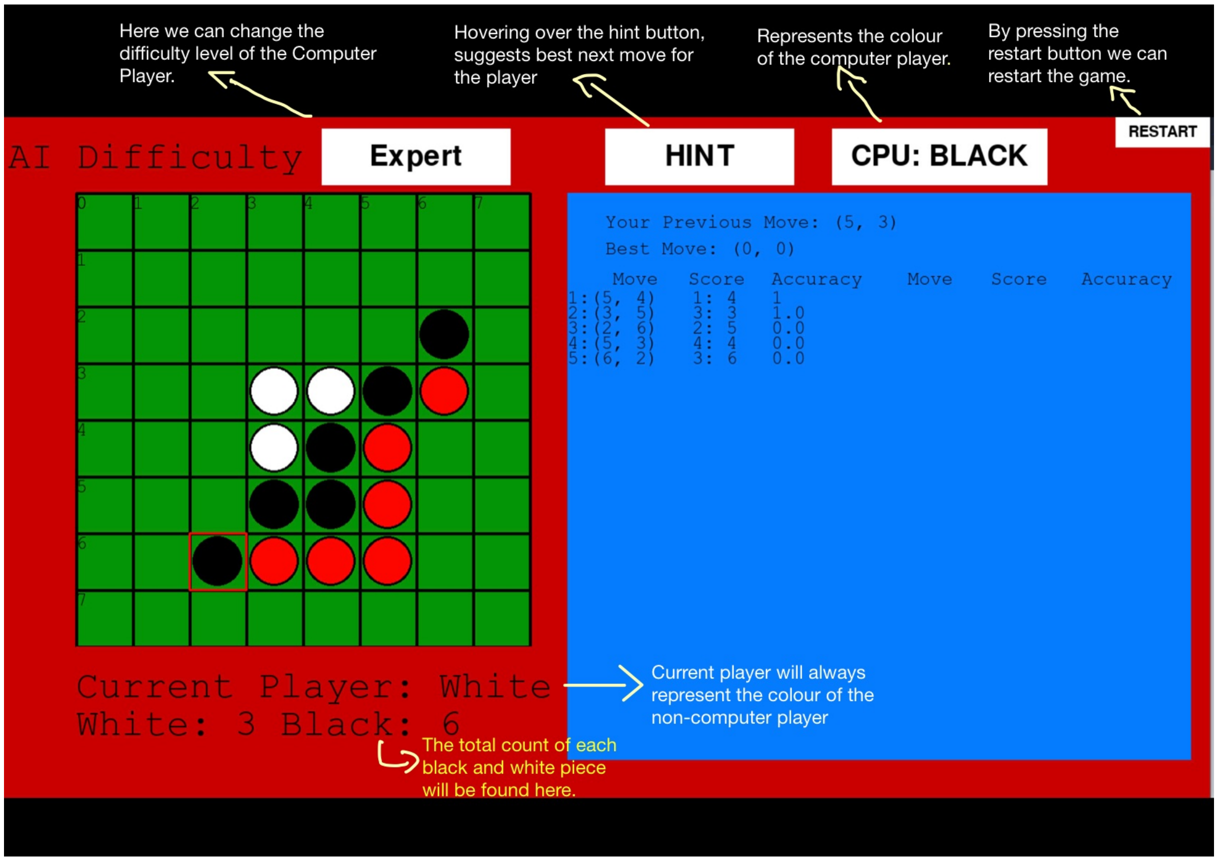
- You can test this function more by creating a GameTree object, make several moves, and call each of the functions above to create two GameTrees of the current game state. Note that these GameTrees satisfy what the point above describe.

7. Next, open the Interactive pygame application window.
8. Choose the color of the computer player by clicking on the CPU: White button. Click RESTART button to apply. (See Figure 2)
9. Choose the difficulty of AI by clicking on the AI Difficulty button. (See Figure 2)

   - You may only change the difficulty of the AI before you make your first move.
10. Hover the cursor on the HINT button to get hint. Note that there will be no hints for playing against beginner players. (See Figure 2)
11. Refer to the strategy feedback system (the blue area) on the right of the screen. (See Figure 2)

    - It shows that past scores and moves made by both players, as well as the accuracy score of each player. The accuracy score ranges from 0 to 1; the higher the score, the better is the move made.

    - This system can tell the user how well they (and the computer) moved and adjust their strategy accordingly in the remaining of the game.
12. Play Othello by clicking the red circles in the gameboard. The red circles indicate the current valid moves.
13. When a winner is declared or there are no more valid moves for either player, click RESTART to play a new game.


Here we can change the difficulty level of the Computer Player.
Hovering over the hint button, suggests best next move for the player
Represents the colour of the computer player.
By pressing the restart button we can restart the game.
RESTART
AI Difficulty
Expert
HINT
CPU: BLACK
0
1
2
3
4
5
6
7
Current player will always represent the colour of the non-computer player
The total count of each black and white piece will be found here.

```
  Your Previous Move: (5, 3)
  Best Move: (0, 0)

   Move    Score   Accuracy    Move    Score   Accuracy
1:(5, 4)   1: 4    1
2:(3, 5)   3: 3    1.0
3:(2, 6)   2: 5    0.0
4:(5, 3)   4: 4    0.0
5:(6, 2)   3: 6    0.0
```

```
Current Player: White
White: 3 Black: 6
```

Figure 2: Othello Board Game

Changes after Proposal

We used the minimax and alpha-beta pruning algorithms to generate the GameTree for computer players, as suggested by the TA who marked our proposal, instead of just generating the whole GameTree, which would be extremely inefficient and take a lot of memory. We also adjusted how the AI make a move: instead of getting the best move from its subtree (of different depths), each AI of different difficulties now makes a combination of random and calculated moves during the game. This can increase the unpredictability of the game and makes Othello more interesting. This idea was also suggested by the same TA mentioned above.

Discussion

Findings:

Othello is a very interesting game because in a few moves a lot of things may change, unlike other board games like Chess, which is much harder to come back to a win in a disadvantageous situation. Beginners might be fooled by the amount of pieces they have on the board in the middle of the game, only to be beaten in the last few turns. This inspired us to start this project and compute a balanced computer player and a strategy feedback system to let players to reflect on how well they do.

After running many simulations between different computer players, we find that it is not as important to generate a large GameTree in the first half of the game than the second half. Although the player may find himself down some pieces in the mid-game, a lot of things can be changed at the endgame. Thus, what we did with the decision making of the computer players was that the computer will focus less in the midgame (generating GameTree with a small depth). Instead, after the gameboard has certain amount of pieces, the computer player will generate the full GameTree which shows game paths until the end of the game. This can always allow the computer to know the best move. Noticing that the end-game matters much more than the first half of the game is crucial, as this allows us to make the computer players generate their GameTrees faster.

Another finding is that we can use alpha-beta pruning algorithm to improve the efficiency of the GameTree. We realized that many of the times that several moves were obviously bad, and competent players would not choose those moves. Thus, using the alpha-beta pruning algorithm, as we are generating the GameTree, we will cut down some evaluations of subtrees and streamline building GameTrees. If the opponent chooses a path which is not on our GameTree, then it will be advantageous for us, since those paths are the 'bad' paths for the opponent. In this case, we will just expand the GameTree and continue the game.

The common alpha-beta pruning algorithm is a searching algorithm, meaning that it searches for the best move from an existing GameTree. Since it is extremely inefficient to generate a whole GameTree and apply this searching algorithm, we came up with an optimized custom version of the alpha-beta pruning algorithm, which allows us to build the GameTree while we are evaluating it. Our implementation of this algorithm is inspiring, because it combines the searching function of alpha-beta pruning and the building function of minimax algorithm together in the same function. Throughout designing this function, it took us many tries to successfully maximize the efficiency of this function by calculating as few paths as possible which getting the best possible moves.

Last but not least, designing the strategy feedback system is also a main part of the project. We are inspired by different Chess game engines which tell us how well we play, what the best move is, and what we should have done when we are playing. In this project, we did a simplified version of strategy feedback system, which lets us know the best possible move in our turn (using the HINT button) and how well we played throughout the game using the feedback system. It also shows how well the computer player plays, so we can compare how well we are playing against the computer. This strategy feedback system can let beginner players to test how well they play and allow them to choose the computer player with similar level as them to play against.

Limitations:

There are several limitations when we were doing this project.

First, there was a lack of datasets for Othello matches. On the Internet, there are several datasets for Reversi (the old version of Othello), which has different game instructions than Othello. The datasets which are Othello games have only games played in a tournament, which don't contain as many matches as we like. Thus, we decided to create our own game tree by simulating the computer players against each other (or itself) to generate its own GameTree.

Second, generating GameTrees take a lot of time and memory. During the game, the GameTrees are generated in real time, meaning that as the computer player plays the game, it will update and expand its GameTree every move. This is essential in order to keep track of the game while calculating the best move for the current game state. Thus, it is important to make the efficiency of generating GameTree high. The cost of this is that the GameTree generated with have a short height (height of 5 to 7) and only strategically-viable paths are included using alpha-beta pruning. This limitation may be lifted if we have a more powerful machine and save the generated tree.

Third, different computer players generate their GameTrees using similar methods, so to make different players to have different difficulties, we added a parameter which dictates how often the AI makes a random move. Since all the AIs may make random move, it is possible that sometimes the stronger computer players be beaten by weaker computer players. This limitation can be reduced by increasing the difference of this parameter among the computer players, so stronger players have much lower chance of playing a random move instead of making a sophisticated move.

Next Steps:

There are several next steps we can do to make our Othello application better.

First, we can improve how well the computer plays by giving extra instructions to the computer, such as taking the corner of the gameboard all the time if possible. The current versions of computer players consider the positioning of the move and the game score when making a move. However, the corner pieces can never be captured and hence is a good idea to always place pieces there. It might be a good idea to re-calibrate how the computer players make decisions.

Second, we can improve the strategy feedback system. One flaw of the system is that the accuracy score of the move may not reflect how well the player makes the move. This is because if the move is not found in the GameTree of the cpu player, then the move will have a score of 0.0. However, this may not mean that the move is not favourable. One reason is that the GameTree of the computer player has been pruned by the alpha-beta algorithm, so moves which are as advantageous as the moves before may be pruned (according to our algorithm), even though they are not worse. This means that some decent moves might be omitted in the GameTree and thus the accuracy score might be 0, which might not reflect the real accuracy. The accuracy score can be redesigned such that all valid moves are considered when we are calculating accuracy, though this might take up extra memory and need extra running time to finish the computations. Moreover, we want to implement an UNDO function, which allows the player to return to the previous game state and undo the bad moves that he/she did. This will be beneficial for improving the decision-making of the players.

References

- Allis, L. V. (1994). Searching for solutions in games and artificial intelligence. Wageningen: Ponsen amp; Looijen.
- Cercopithecan. (2018, April 20). Algorithms Explained – minimax and alpha-beta pruning. YouTube. https://www.youtube.com/watch?v=l-hh51ncgDI.
- CYNINGSTAN (Ed.). (n.d.). Reversi. Retrieved March 06, 2021, from http://www.cyningstan.com/game/73/reversi
- Zhang, Z., amp; Chen, Y. (2014). Searching Algorithms in Playing Othello. Min H. Kao Department of Electrical Engineering and Computer Science. http://web.eecs.utk.edu/ zzhang61/docs/reports/2014.04%20-%20Searching%20Algorithms%20in%20Playing%20Othello.pdf.