Implementing and Evaluating Artificial Intelligence on the Othello Game

Student Name: Yizhuo Liang

Supervisor Name: Tom Friedetzky

MSc Dissertation submitted as part of the degree of M.Sc. MISCADA to the

Board of Examiners in the Department of Computer Sciences, Durham University

June 2022

***Abstract* —** Artificial intelligence is becoming more and more common in our lives. This project will implement and explore the strength of several different AI systems in playing the Othello board game. The implementation will include a complete Othello game environment and an interactive interface that humans can use to interact with each other or the AIs. All AI agents include multiple basic strategic agents as well as several traditional algorithms, including Minimax, Alpha-Beta, Monte-Carlo tree search, and deep reinforcement learning algorithms. Each algorithm was tested with various parameters to ensure that the maximum strength represented by the algorithm was achieved in the final test. In the end, the most suitable AI algorithm for Othello was Alpha-Beta, which outperformed the other algorithms regarding win rate and time spent. Additionally, the deep reinforcement learning algorithm was found to be unsuitable for use in board games such as Othello.

***Keywords* —** Othello; Artificial intelligence; AI; Minimax; Alpha-Beta; Monte-Carlo tree search; Reinforcement learning; Deep learning; Python

I INTRODUCTION

Artificial intelligence (AI) is a system whose behaviour makes it impossible to distinguish from humans; the definition was first proposed in 1950 and has evolved over the years to become a part of people's lives (Haenlein & Kaplan 2019). Since humans began to study AI in depth in 1990, gaming has been one of the areas where people can test the capabilities of AI by using it to play against humans, especially in board games. One of the most famous matchups is Alpha-Go versus Lee Sedol (Wang et al. 2016), in which the AI defeated the go master who ranked second in international titles in a best-of-five match of the GO game in 2016. However, this is not the first time; in 1997, the AI deep blue also defeated a human (Campbell et al. 2002) who is a Russian chess grandmaster; former World Chess Champion, in the less complex game, Chess.

This project focuses on AI players in Othello, a less complex board game than chess, and will implement a platform that allows the AI to play games against other AI or humans to test its capabilities. For Othello, there are plenty of established algorithms; this article will implement the various versions of Minimax, Monte Carlo tree search, deep reinforcement learning, then assess the capabilities of each AI.

A Aim

This project aims to implement several machine learning AI agents and evaluate their ability and performance when playing against each other. How well they perform against some basic Othello strategies, such as random, evaluation, and min/max scores, will be evaluated, and the most suitable AI algorithm for Othello will be found. Indeed, there is an interesting argument that artificial neural networks do not achieve the desired results regarding board games (Barber et al. n.d.), and are, on the whole, inferior to several traditional intelligent algorithms (Barber et al. n.d.). This argument will also be tested in this project; the artificial neural network used for testing is an algorithm called Double Deep Q-network learning, details of which will be explained later.

During an actual Othello match, each human player is given a total time limit for one game, typically 5 to 30 minutes(*Othello Game Rules* n.d.). A time limit will also be set for the AI by allowing a 30 seconds time limit for each move. Theoretically, if the AI has unlimited time to think about each move, it will be surely win, since it can Brute-force search through all possibilities to find a move that will ensure it wins the game.

The program also has a graphical user interface that allows humans to join the game to see how well they can play against humans. However, as no professional players could be found to play the game with AI, this article does not prove that these AIs can beat humans consistently.

B About Othello

For those who do not know, Othello — also called Reversi — is a PSPACE-complete game (Iwata & Kasai 1994) regarding computational complexity. It is a perfect information game with a zero-sum property where two players compete on an 8×8 board with a total of 64 black and white pieces. Each round begins with 2 black and 2 white pieces in the centre of the board, as shows in Figure 1.

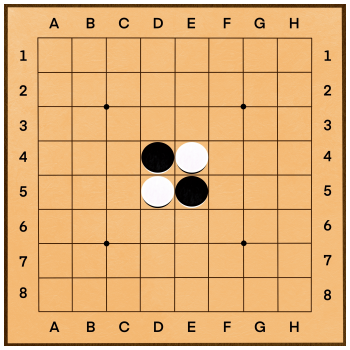
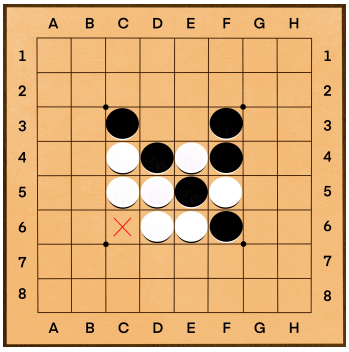
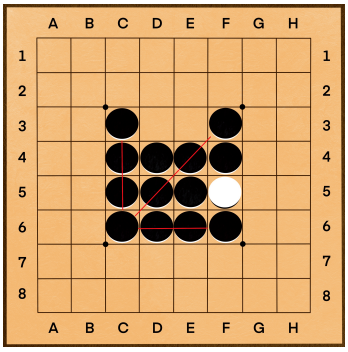


Figure 1: Initial game board

The two players will take turns placing pieces of their colour on the board; black will always be the first. The game's objective is to flip pieces of their opponent's colour to their colour. At the end of game, the player with more of their colour pieces on the board is victorious. The rule for flipping their opponent's pieces, swapping the colour, is to have a piece of their colour at each end of the opponent's linked pieces, either horizontally, vertically or diagonally. The player must be able to flip one or more of their opponent's pieces at every move; otherwise the player skips the turn. Figure 2 shows the three situations in one turn where the black pieces can flip an opponent's pieces. At the turn on the left board in Figure 2(a), the 6 white pieces will be converted if black plays on the red cross. All pieces under the red line on the right board in Figure 2(b) can be flipped because they have a black piece at each end.



(a): Before black move



(b): After black move

Figure 2: The way a player can flip the pieces

Finally, there are four scenarios to complete a game. In the first case, the entire board is filled with pieces. Another possibility is that only one colour remains on the board. In the third case, neither player can make a move. The last case occurs when a player has run out of time. Regardless of the number of pieces on the board, the player has simply lost; however, since we do not time the entire game, but give the AI a time limit at each move, the last case is not considered in this report.

C Main Basic strategic

When playing Othello, both beginner and professional players should consider the following two basic strategies to win the game.

C.1 Stability of pieces

In Othello, the stability of a piece means whether the opponent's pieces can easily flip it. For example, the pieces in the four corners of the board can be called stable pieces, because the opponent cannot flip them once they have been put on the board, as they cannot be bracketed. Likewise, any pieces of the same colour connected to a corner piece can be called stable pieces. Thus, to win the game, a player should try to increase their number of stable pieces and minimise their number of unstable pieces whenever possible.

C.2 Mobility of player

Mobility is the number of possible places a player can move in a turn. The rules mention that if a player cannot flip any of their opponent's pieces, the turn is skipped, and the opponent can move on. The players must therefore consider the mobility of both players when playing. To win the game, a player should consider making their opponent's mobility as small as possible so that his opponent has fewer options and is more likely to choose poor move. The best possibility is when the opponent's mobility is 0, so that they can make consecutive moves. At the same time, players need to make their mobility as large as possible to have more options when making moves.

II RELATED WORK

For computer agent to play Othello, it must first be able to read the current state of the board and then use some algorithm to decide where the next move should be. Finally, the result must be output to the board. The intermediate algorithmic steps will be the focus of this work.

With today's rapidly developing artificial intelligence, many sophisticated algorithms can be used by computers to decide where to play. However, this project's analysis focuses on the algorithms: Minimax, Minimax with Alpha-Beta pruning, Monte-Carlo tree search algorithm, Double Deep Q-Networks (Reinforcement and Deep Learning).

Before discussing more details about the implementation and analysis, here are some brief descriptions of the algorithms used

A Basic agents

Basic agents include agents that use elementary, straightforward algorithms, which generally follow a single strategy. They typically only consider the present state of the council and possible positions; hence they do not consider the situation a few movements later.

A.1 Random agent

Random agents are the most straightforward agent; they do not involve any thought process but simply reads the state of the board, finds all the available squares, and chooses one at random. Any logical agent or human player with background knowledge is good enough to beat this agent. This agent is used to judge whether other agents are being adequately implemented.

A.2 Maximise the score

Often the first thought of a player new to Othello is to always keep more pieces of their colour on the board. This agent uses this strategy. It finds all the possible positions at the current stage, calculates the number of pieces of its colour on the board if that move is used, and then chooses the move that will give it more pieces. However, if a player has many pieces on the board, it causes the opponent's mobility to get bigger, which is not a good strategy.

A.3 Minimise the score

This agent has the same structure as Maximise the score, but at the end, the agent chooses the move that leaves the fewest pieces of its colour on the board. The existence of this agent is based on the basic strategy Mobility of player. If fewer pieces of the player's colour are on the board, the opponent has fewer chances to flip any pieces, resulting in less mobility for the opponent.

A.4 Roxanne

Roxanne uses an 8×8 evaluation matrix to choose how to play the game. The scores in the matrix correspond to the positions of the board, as depicted in Figure 3 below.

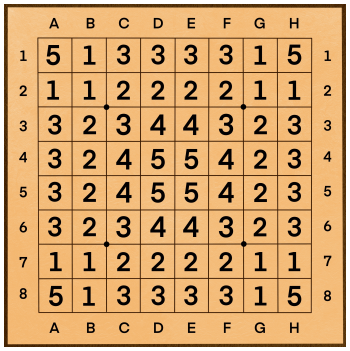


Figure 3: Example of evaluating matrix

The scores in this matrix are a combination of many different basic Othello strategies. For example, the stability of the pieces mentioned earlier, which is reflected by the highest scores being in the four corners, ensuring that the agent will always play there if possible. Additionally, the mobility of players must be considered. When the pieces are concentrated in the middle of the board, the opponent is less likely to flip them. Therefore, their opponent's mobility is lower, resulting in high scores in the centre of the board. The scoring matrix continues taking more basic Othello strategies into account until the matrix is filled.

At each turn, the agent compares all possible moves with this scoring matrix, selecting the highest scoring move, or choosing randomly between the highest tied actions. Since this agent combines several basic strategies, it should be able to outperform the previous agents. At this stage, a novice human cannot beat this agent. Winning requires that the human has studied Othello for quite some time.

B Complex agents

Unlike the previous simple and straightforward agents, the following agents have more sophisticated algorithms that allow them to consider not only the current state of the board, but the state of the board after various possible moves is also analysed and studied, or even considered the state of the board simulated within the algorithm after several rounds of moves by both players.

Based on this reasoning, it is theoretically possible that the following agents are much more powerful than the previous simple agents. As for humans, they need to have extensive Othello experience to be able to compete with them.

B.1 Minimax algorithm

MiniMax is the famous depth-first search algorithm. It is a robust algorithm for finding the most appropriate move position among all possible options (Cherry 2011), and this algorithm is often used in board games AI for two-sided, zero-sum games. In game theory, a zero-sum game is where if one player has an edge, the other player must have a downside. Two players have conflicting game objectives, and their interests are summed to zero or a constant (Dahmani et al. 2020).

Specifically, the MiniMax search builds a game tree on each turn (Sannidhanam & Annamalai 2015), which theoretically contain all possibilities for the game from now on. Then the algorithm evaluates each path in the tree, which represents all possible scenarios after the current state of the board, to find and select the one that allows it to win the game. However, since Othello is PSPACE-complete in complexity, it has a large number of game possibilities (Iwata & Kasai 1994). Due to computer hardware performance limitations, it is unlikely that the tree will have many layers in practice.

A portion of the 3-level depth tree is drawn in the Figure 4, in which the squares represent the turns of the player from their point of view, the circles are the turns of the opponent, and each line between them is a possible move for a given turn. Note that all scores in the figure are arbitrary for illustrative purposes; the intricated scoring system will be explained later in the implementation phase.

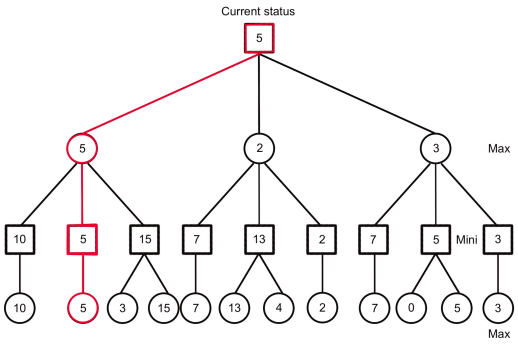


Figure 4: The 3-level depth tree

The MiniMax algorithm will first explore all possible child nodes in the tree by adding all possible actions to the child nodes separately. Then, score the game board state represented by the child nodes at the last level (Cherry 2011). In this 3-level tree, the score at the last level represents the score of the board after the agent's last move, so the agent chooses the node with the higher score to return to its parent node. The score in the third level is the score of the board after the opponent has played. Since Othello is a zero-sum game, from the agent's point of view, the lowest score is the most advantageous position for the opponent. The agent will assume that the opponent will think like himself and choose the smallest score to return to the parent node. At the second level, it is back to the agent's turn again, so the agent chooses the largest score. Thus, a depth-3 MiniMax search is completed, with the agent choosing the red path, which is the most rewarding path for the agent, under the assumption that the opponent will also think similarly (Cherry 2011).

However, one of the big problems here is that the opponent will not necessarily think the same way as the agent, so they may not play the same way as the agent thinks. This scenario results in the agent recreating the tree each turn and searching for all possibilities again. It is conceivable that as the game progresses, the breadth of the tree will become so large that it cannot use deeper trees to do the exploration. This problem leads to the following variant of the MiniMax algorithm.

B.2 Minimax with Alpha-Beta pruning

The Alpha-Beta agent has the same core algorithm as the previous Minimax agent. The primary enhancement in this version is that the agent ignores some unimportant nodes to make the exploration process faster. When exploring deeper levels in the tree, there are some sub-nodes that do not affect our final result, but the normal Minimax still explores these useless nodes, resulting in a meaningless time increase in exploration. The details of how to achieve this by cutting out unimportant nodes are shown in Figure 5.

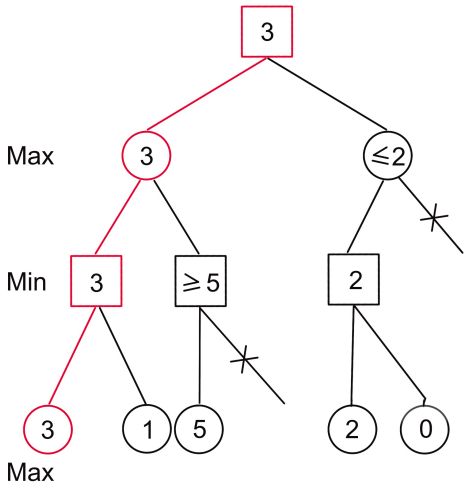


Figure 5: Alpha-Beta pruning process

This figure is also a three-level deep exploration tree. When the agent starts exploring the deepest level from left to right, we can see when the third node results in a 5. By the logic of returning the maximum value, the second node in the penultimate level can only have a number greater than or equal to five. However, the agent has the smaller number 3 in the first node, so whatever the number on this side, the agent will not be considering this line, and the remaining nodes in the line can then be pruned. The same thing happens on the right. When the agent gets to 2 and 0, it can deduce that the largest result in the second level of the tree on the right is 2. Since there is a better result 3 on the left, all the other nodes on the right can be pruned. If the agent explores the tree this way, the time spent will be significantly reduced, even allowing the agent to use deeper layers of the tree than it would otherwise have time to explore fully. In this case, the deeper trees would lead to more accurate results and a higher win rate.

B.3 Alpha-beta with hash tables

This agent adds only a minimal functionality to the previous agent. However, with some training, it will theoretically reduce the time spent significantly, making it possible for the Alpha-Beta agent to use a deeper exploration tree.

Each time the agent wins a game, it stores all the choices made during the game and the state of the board at that time in a hash table. In future games, the hash table is searched to see if the current board state is in the hash table before creating the exploration tree. If it does, the choices made at that time are extracted and applied directly, so that the exploration step is skipped, and the time spent on the game is significantly reduced.

B.4 Monte-Carlo tree search algorithm

This agent uses the famous algorithm, Monte Carlo tree search(MCTS) (Browne et al. 2012). In general this MCTS agent has a very similar way of thinking to the human mind. When a human is playing Othello, it is not possible to list all the possibilities in the mind. The player can only rely on previous knowledge and experience to develop a few possible moves that are most likely to give an advantage and then try to think of the opponent's possible responses to those moves. After careful consideration, the player will choose one of these possibilities to implement. This process is also the core logic of the MCTS agent.

The MCTS algorithm takes a total of four steps in a loop: selection, expansion, simulation and backtracking. A single game turn consists of the agent repeating these four steps thousands of times. One iteration of these four general steps is illustrated in Figure 6:

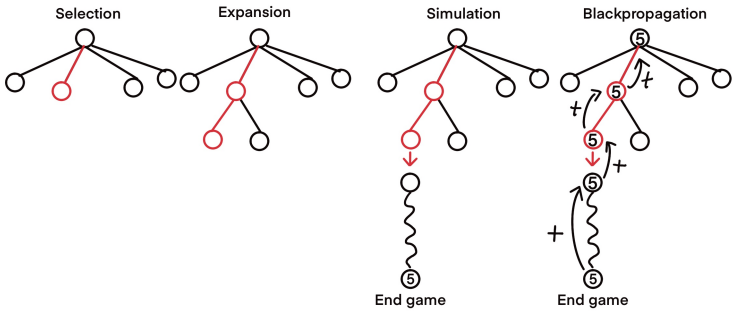


Figure 6: The 4 steps of the Monte Carlo tree search algorithm

The MCTS agent will also create a search tree with the current state of the board as the root node. In the first step, selection, the agent finds all possible moves, turns them into child nodes, scores all of them using the evaluating function, and then chooses the one with the highest overall score. This process is like humans using the lessons learned to pick a few of the most likely response options. Details of how the scoring is carried out will be explained later in the implementation phase. Expansion is the second step, at which point it is the opponent's turn to play, represented by the second level of the tree. The agent finds all of the opponent's possible moves and adds them to the child nodes. In the third step, simulation, the agent randomly chooses a move from all of the opponent's possible moves as a starting point, and quickly simulates the entire game. Generally, it is assumed that both players play randomly until a winner is reached, with the score recorded. In this example, if the agent wins, the score is plus 5; if it loses, the score is minus 5, again all scores are arbitrary for illustrative purposes. Furthermore, here it is like a human being deep in thought about those few possible options. The final step, blackpropagation, returns the score to the parent node, layer by layer, until it reaches the root node.

During a turn, the agent repeats the above loop thousands of times, after which the move that has been selected the most times in the selection step is the best move for the turn. This choice results from the selection step, which evaluates the initial score of each move along with all previous simulations. The higher the initial score of a move, the more likely it is to be selected for a simulation, and the more times it wins in a simulation the more likely it is to be selected next time. In contrast, losing all the time in a simulation will reduce the chances of being selected. This way, the move with the highest number of visits has an excellent initial score and a higher chance of winning the game in many numbers of simulations.

B.5 Double Deep Q-network learning

The algorithm used by this agent, Double Deep Q-network Learning (DDQN), is part of deep reinforcement learning, which combines the two main machine learning methods - deep learning and reinforcement learning (Fan et al. 2020). The algorithm takes the feature extraction ability of deep learning and the targeting ability of reinforcement learning, allowing the whole algorithm to face more diverse environments (Kumar et al. 2019).

Q learning is originally a basic value-based reinforcement learning algorithm (Watkins & Dayan 1992). The algorithm selects the action with the largest expected reward based on comparing the expected reward of all actions that can be taken at a given state at a given time (Watkins & Dayan 1992). The algorithm constructs a Q-table of all states and their corresponding actions with expected rewards, and updates the entire Q-table in real time after each action based on the difference between the actual and expected reward (Watkins & Dayan 1992). The formula for updating the Q-table is given as equation (1) (Abed-alguni 2017):

$$\mathrm{Q}(s,a) \leftarrow \mathrm{Q}(s,a) + \alpha(r + \gamma * \mathrm{Max}(\mathrm{Q}(s',a') - \mathrm{Q}(s,a)) \tag{1}$$

where s is the current state, s' is the next state, a is the current action taken, a' is the next action taken, r is the real reward given by the environment, α is the learning efficiency, which determines how much of this error is to be learned, and γ is the decay factor which, the closer it is to 1, the more sensitive the algorithm is to future rewards(Abed-alguni 2017).

Adding deep learning and double-network techniques to the above Q-learning is the algorithm used by the DDQN agent.

III IMPLEMENT OF OTHELLO GAME AND COMPLEX AGENTS

The game body and all the AI agents are implemented using the Python language, which is an easy-to-use and very efficient language, especially in the field of machine learning (Arora 2021). It is a modular programming language, there are many free packages available that can be used directly in any programs (Stančin & Jović 2019). For example, in some cases the functionality needed in implement may have already been written and wrapped in a package by other people, so others can use it directly. It is worth mentioning that during the process of writing the graphical user interface(GUI), a small bug was found in the **pygame_menu** package used, which caused the text and images to be misaligned when some items were moved around. Fortunately, the author of the package was able to locate and fix the bug within a few days after emailing him and giving him feedback on the bug.

A Othello game

In the main file, the functions for the basic rules of the Othello game are written using the most basic packages, such as **numpy** and **random**. It contains the most essential and frequently used function of the game, **flip_pawn**. This function is crucial because it returns which pieces on the board will be flipped after a move is inputted. This is the core of Othello and is called for every possible legal move, every move, and every flip after a move.

The game's graphical interface is written using the **pygame_menu** package. The whole program can be used for human versus human, human versus AI, and, most importantly, AI versus AI. The program itself also includes data collection and documentation. When the data collection mode is switched on, the two chosen AIs play 1000 consecutive games and exchange black and white positions after each game, such that each side has 500 times to take black pieces. This procedure ensures that the final data is not affected by the natural differences between black and white in Othello. After 1000 games, the program saves all the data collected as CSV files in a folder corresponding to the two AI names.

B Complex agents

The implementation of the basic agents is straightforward; therefore, only the complex agents are described here.

B.1 Minimax

The Minimax agent and several other variants of the agent are identical in terms of the core algorithm. It is divided into two main functions: evaluation and move. In the move function, the entire search tree is created recursively, with each level of recursion searching and adding all possible moves to the children. This process is repeated until the last level, when the evaluation function is used to score the children at the last level and then return to the parent node one level at a time, following the method described earlier in II.B.1, to get the best move. The pseudo-code for the move is as follows in algorithm 1:

Algorithm 1 Move in Minimax

Input: $board_state$, $depth$, $player\_colour$, $move = [None, None]$, $end = True$.
Output: $action$.

$tem\_board \leftarrow board_state$
if $move \neq [None, None]$ **then**
    $play\ [x, y]\ action\ on\ tem\_board\ with\ corresponding\ colours$
end if
if $depth == 0$ **or** $gameover$ **then**
    return $evaluation$
end if
$possible\_moves \leftarrow get\ all\ possible\ moves$
while depth > 0 **do**
    $depth \leftarrow depth - 1$

```
    if player_colour == 'black' then
        score ← −99999
        for move in possible_moves do
            tem_score ← Move(tem_board, depth, 'white', move = move, end = False)
            if tem_score > score then
                score ← tem_score
                action ← move
            end if
            if end then
                return action
            else
                return score
            end if
        end for
    else
        score ← 99999
        Same process frome line 13 to 24 just changes the player colour
    end if
end while
```

The evaluate function is the same in all versions of the Minimax algorithm. In particular, it evaluates the current state of the board in 4 ways. First, the current board is scored by multiplying the score matrix from the previous Figure 3, where black on the board is +1 and white is -1. Second, the number of pieces each player has is then scored, based on the previous II.A.2, where the fewer pieces a player has at the beginning, the better, and the more pieces at the end, the better. Thirdly, the opponent's Mobility, as mentioned in I.C.2. Finally, it depends on whether any of the pieces are in or near a corner, with extra points for pieces in the corner and minus points for pieces one move away as this makes it easier for the opponent to take the corner.

B.2 Monte-Carlo tree search

The MCTS algorithm is a bit more complex than the previous Minimax, so we first create a class **Node**, which allows us to create the required search tree more easily. All objects of the **Node** class contain the following attributes: **move** - the move represented by the current node; **board** - stores the state of the board if the move of the current node is executed; **player** - the current colour; **init_value** - the initial score of the current move; **result_value** - the reward received after the simulation; **visit** - the number of times the current node has been explored; **children** - list of children of the current node; and **parent** - parent of the current node.

The Node class also contains several essential functions: **fully_expanded** - checks if the node has been fully explored; **add_child** - adds children to this node; **update_score** - back-propagates the results to all nodes after MCTS has gone through the simulation step; **best_child** - selects the node with the highest overall score from the children of the current node. The most important step is **best_child**, as it determines which node will be explored and simulated, indirectly affecting the agent's choice in this round. The formula for how to combine the initial and simulated scores of the nodes is as follows:

$$\textbf{score} = mean(\textbf{result\_value}) + \alpha * \frac{\textbf{init\_value}}{1 + \textbf{visit}} \tag{2}$$

where α is a hyper-parameter and is chosen as 0.4 in this algorithm. Many numbers were tested in the interval from 0 to 5, and 0.4 is the one with the best results. Details of the specific test will be explained subsequently.

The main function Move of the MCTS algorithm is as follows in algorithm 2:

Algorithm 2 Move in MCTS

Input: $board_state$, $player\_colour$, $hype\_parameter$, $max\_iter = 1000$.
Output: $action$.

1: $root \leftarrow Node(board\_state,\ player\_colour,\ hype\_parameter)$
2: **for** i in $range(max\_iter)$ **do**
3: **if not** $root.fully\_exppanded$ **then**
4: $possible\_moves \leftarrow Find\ all\ possible\ moves$
5: **for** $move$ in $possible\_moves$ **do**
6: $add\ child\ to\ root$
7: **end for**
8: **for** $child$ in $root.children$ **do**
9: $play\ the\ move\ as\ child.move\ in\ child.board$
10: $child.init\_value \leftarrow evaluate\ the\ child.board$
11: **end for**
12: **end if**
13: $best\_child \leftarrow root.best\_child\ with\ player\_colour\ switched$
14: **if** not $game\ over\ with\ best\_child$ and $best\_child\ has\ any\ legal\ moves$ **then**
15: $repeat\ line\ 3\ to\ 12\ with\ root\ replace\ by\ best\_child$
16: **end if**
17: $select\_child \leftarrow random.choices(best\_child.children)$
18: $reward \leftarrow simulation(select\_child)$
19: $select\_child.update\_score(reward)$
20: **end for**
21: $best\_action \leftarrow find\ who\ has\ biggest\ visit\ in\ root.children$
22: **return** best_action.move

The evaluation function mentioned in line 10 of algorithm 2 is the same as the evaluation function in the previous Minimax algorithm. It is also worth mentioning that in the simulation step on line 18, instead of using the random method mentioned in II.B.4 to play until the end of the game, the Roxanne method is called. Roxanne has a significantly higher win rate than the random method; therefore, MCTS will theoretically perform better with this substitution.

B.3 Double Deep Q-network learning

This algorithm consists of two sections, the training part and the agent who actually plays the game.

In the training section, there are two essential parts. One is the construction of the neural network, where the **torch** package is utilised to help create the network layers. The NET class is defined whose properties include a neural network with 10 layers, not counting the output layer, of which 8 are convolution and 2 are connection layers. The reason for choosing this 10 layers here is that in some studies it has been shown that this 10 hidden layers have the best overall performance for Othello (Liskowski et al. 2018). Additionally, the forward function to pass parameters from layer to layer which is a relatively simple function that just passes the output of the previous layer to the next layer as input, without any computation in between.

The other part is the class DQN for Q-learning. In the original Q-learning there was a Q-table with all the states and corresponding actions, but in Othello, it is impossible to create such a table; there are too many possible states and corresponding actions. Therefore, the neural network created above will be used to dynamically output the Q values of the corresponding states. Specifically, on each turn, the current board state is input into the neural network, which then outputs a Q value corresponding to the current state and finds the move with the maximum Q value by matching the current possible moves to the output Q value.

Furthermore, to achieve a higher precision level, the DDQN uses two neural networks with the same structure but different parameters, one for the black and one for the white case. They are stored separately during training, and the corresponding neural network is loaded during the agent's move, depending on the colour taken. The following pseudo-code algorithm 3 shows their training process:

Algorithm 3 Training in DDQN

1: $offensive \leftarrow DQN(1)$ {load the black network}
2: $defensive \leftarrow DQN(-1)$ {load the white network}

```
episode ← number of round wants to train
for i in range(episode) do
    game_state ← setup the game environment
    while True do
        {Black turn}
        offensive play move base on its network
        Storage experience pool for black
        if game over then
            offensive.learn{using the experience pool saved before to update the network}
        end if
        {White turn}
        defensive play move base on its network
        Storage experience pool for white
        if game over then
            defensive.learn{using the experience pool saved before to update the network}
        end if
    end while
    if episode + 1 == 0 then
        save the parameters of offensive network
        save the parameters of defensive network
    end if
end for
```

As in the other section, the agent that actually plays Othello was quite simple to implement. By loading the corresponding neural network and entering the current board state, the best move of the agent choice can be obtained extremely quickly.

Compared to the previous two algorithms, the DDQN is surprisingly simple to implement since the complex computations and attempts take place inside the neural network, which cannot be seen or controlled. Therefore, the difficulty with DDQN lies more in understanding the whole architecture of the neural network, the logic of Q-learning updates, and the long time needed to train the neural network itself.

IV RESULTS AND EVALUATION

After all implementations were completed, a series of tests were carried out on all agents to analyse their strengths. The hardware used for all tests was a PC with an Intel(R) Core(TM) i9-9900KF @3.60GHZ CPU, NVIDIA Geforce RTX 2080TI (11GB) GPU and 32GB of RAM. Except for the test between basic agents where 100 games of data were used, the tests were conducted with 1000 games between the two AIs to ensure relatively accurate data was collected. All tests were conducted by exchanging the colour of the pieces held by each side after each game, to ensure that the final data was not affected by the natural advantage of different colours. This is because when testing the basic agents, it was found that they performed differently when given different coloured pieces.

A Black and White

Before conducting any tests when the program was still in the implementation phase, the same agent behaved slightly differently when in the position of black and white pieces respectively. After completing the implementation, the basic agents first played a test of 1000 games each to determine whether black or white was advantaged, the results of which are shown in the Figure 7:

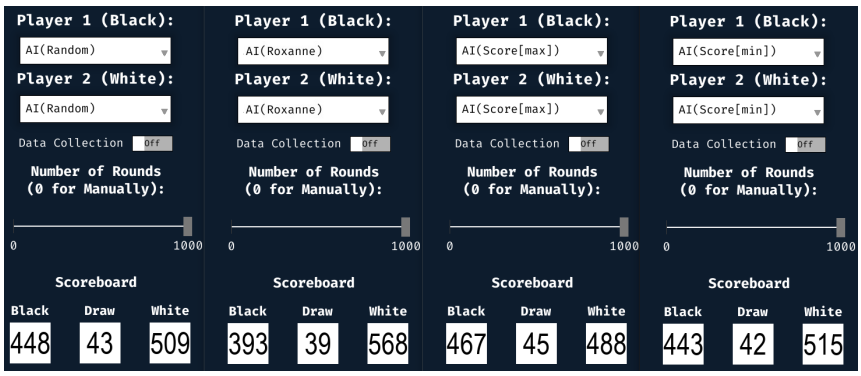


Figure 7: 1000 games with all basic agents play to itself

All 4 basic agents have some advantage in terms of winning percentage against themselves on the white side. Therefore, the white side appears to have a natural advantage in Othello. In all subsequent tests, the AI will switch colours after each game to be as fair as possible.

B Basic agents

Due to the natural difference in algorithmic complexity, it is essentially impossible for a basic agent to beat a complex agent. However, in subsequent tests, the baseline agent was sometimes needed, and all the basic agents were much faster than the complex agents, with their average single-move time being less than 0.01 seconds. So The strongest basic agent was selected to be the baseline agent in the subsequent tests. Additionally, the data from 100 games was used in this section only. As these basic agents essentially follow a single strategy, although some random selections are added, their overall strength does not fluctuate very much. After performing the test, the results as in the following Figure 8

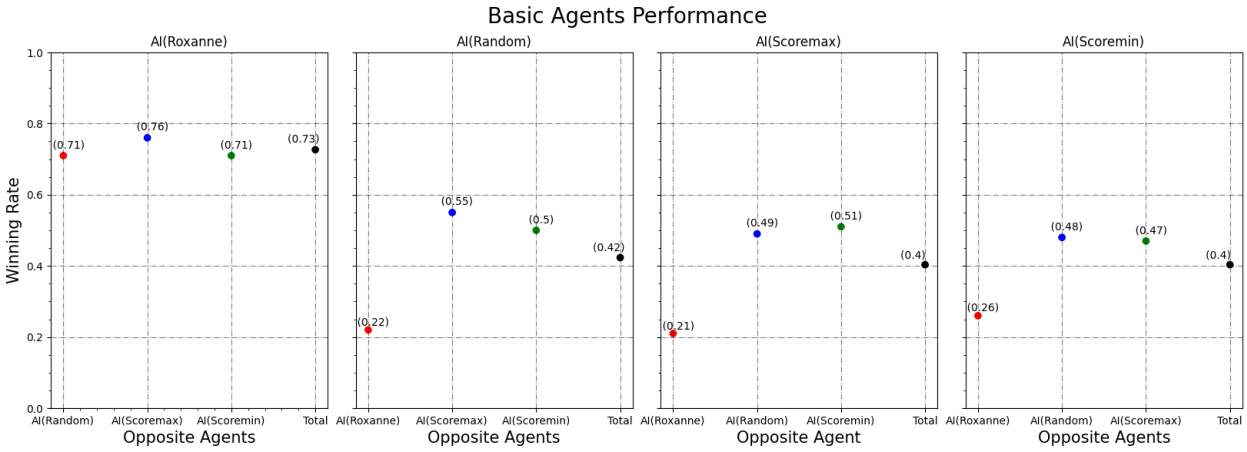


Figure 8: The 4 basic agents play to each other

It is clear from the figure that Roxanne is the strongest of the four agents and will be the baseline agent after this. It has an average win rate of 73% against the other three agents, while the other three have an average win rate of around 40%. This result is also in line with expectations, as the evaluation matrix used by Roxanne is a combination of multiple strategies, which allows it to take a more integrated view of the situation at hand. The other agents follow only one strategy, which leaves them with many limitations.

C Complex agents with different parameters

It is necessary to play different AIs against each other to assess the level of strength between different AI agents. However, each AI has specific parameters that will affect its final performance. Therefore, the most robust version of each AI agent must be found before evaluating the different AI agents.

C.1 Minimax agents and Alpha-Beta pruning

Since Minimax and Alpha-Beta use the same core algorithm, this means that the factors affecting the strength of both agents are the same, namely the depth of the exploration tree. In theory, with a deeper exploration tree, they should perform better. To test this, experiment with 5 depths from 1 to 5 will be used for testing. However, as the following Figure 9 shows the depth-5 on Minimax cannot used in this project as it exceed the time limit.

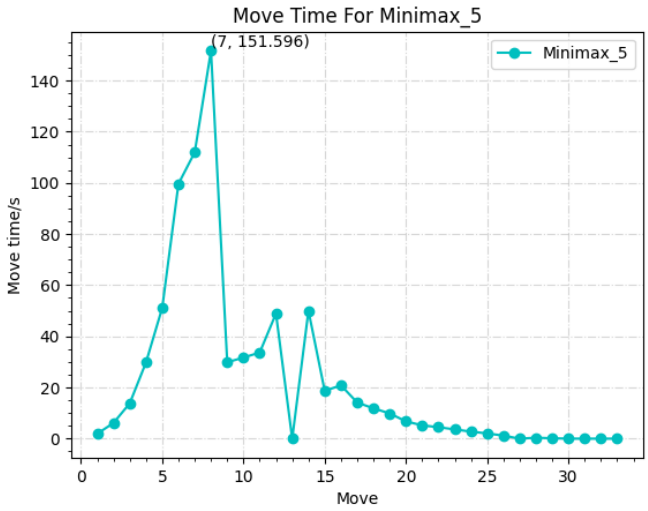


Figure 9: The time taken for each moves of Minimax_5 during a game

The Figure 9 shows the time taken by the Minimax_5 agent for each move against the Roxanne agent. It is clear from the figure that in many moves, the agent takes much longer than the limit of 30 seconds; in some cases, it takes up to 150 seconds. Therefore the performance of Minimax agents will be compared when the depth is 4 or less.

In the test, each Minimax agent plays 1000 games against the other three agents, recording their winning and losing situations. Their win rates for the 3,000 games were combined, and the results are shown in Figure 10:

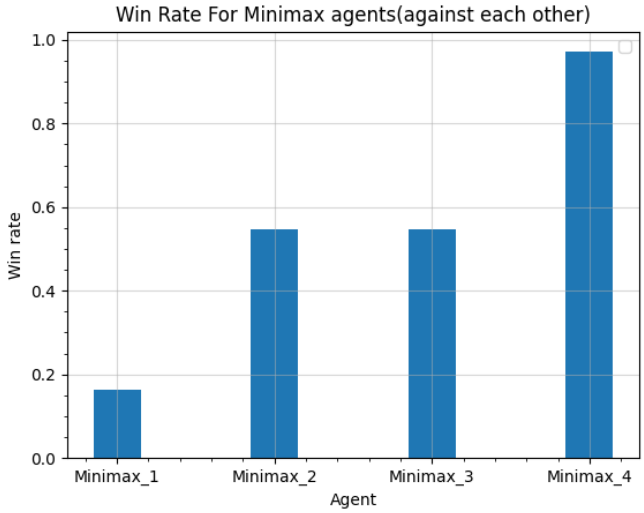


Figure 10: 4 Minimax agents of different depths play against each other

As one can clearly see from the figure. Unsurprisingly, the Minimax_4 agent with the deepest exploration tree has a 97% win rate, which is well ahead of the other three agents. In the subsequent analysis, The Minimax_4 is used for the further tests, and any Minimax agent that does not indicate depth represents the Minimax_4 agent. However, it is interesting to note that the Minimax_2 and Minimax_3 agents have very similar winning rates, at 54.6% and 54.5%, respectively, which are almost identical. After carefully checking the data for any errors, one possible reason for this phenomenon is that when using different depths of the exploration tree, the

last layer of the tree, the layer where the agents scores the game board, is caused by a different players. In simple terms, when the depth is odd, the last layer represents a situation caused by the agent, and when the depth is even, the situation is created by the opponent. Therefore, the likely conjecture is that the only way to improve Minimax is to analyse the situations caused by the deeper opponent; this finding means that each time the depth is increased to an even number, the agent's strength will increase significantly, and the odd-numbered depth will remain the same as the previous depth.

Alpha-beta pruning is literally a time-reduced Minimax algorithm. Theoretically, both agents should have the same win rate against the same opponent at the given depth, except that Alpha-Beta pruning will take much less time for each move. The tests were performed on Minimax and Alpha-Beta pruning to prove this hypothesis. A comparison of the time taken by the two agents at each move for the same depth of 5 is shown in the Figure 11,

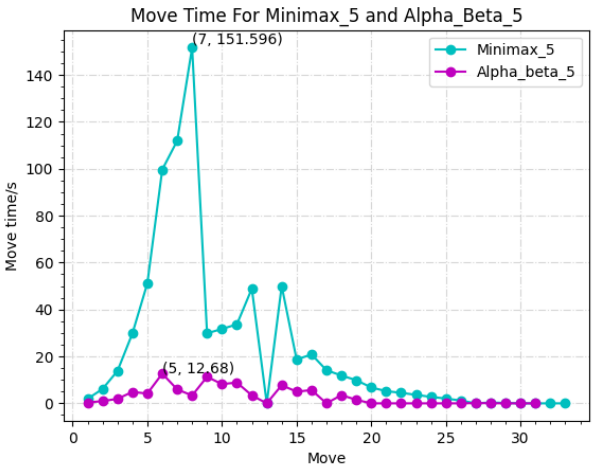


Figure 11: Minimax_5 and Alpha_Beta_5 against baseline agent

It is clear from the plots that Alpha-Beta pruning is a considerable improvement in terms of time spent, at most saving over 85% compared to Minimax. This process allows Alpha-Beta_5 to remain within the set upper limit of 30 seconds, and for all further tests on Alpha-Beta pruning were able to use depths 1 through 5. As for depth 6, since it is a tree search, the number of child nodes increases almost exponentially with each additional level, and even with Alpha-Beta pruning, it still takes more than 30 seconds. Thus depths 6 or more are not considered here.

After this, the 4 sets of experiments were conducted at 4 different depths, 1 to 4, of Alpha_Beta and Minimax against the baseline agent. The result helped determine whether the two algorithms differ only in time per move. Depths 1 to 4 were chosen because Minimax at depth 5 would take too long to run through 1000 games, so using depths 1 to 4 allowed the experiments to finish more quickly. The test results are shown below in Figure 12.

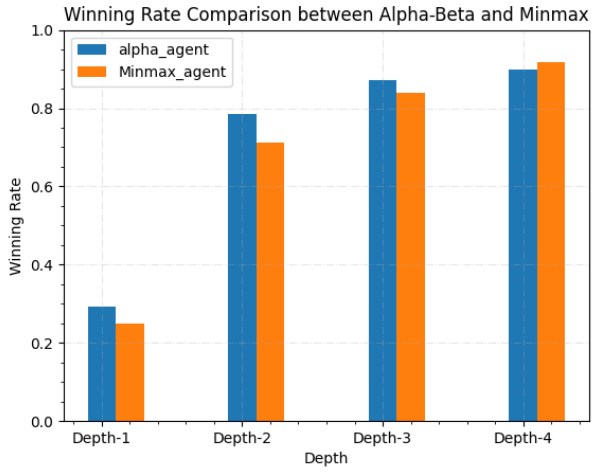
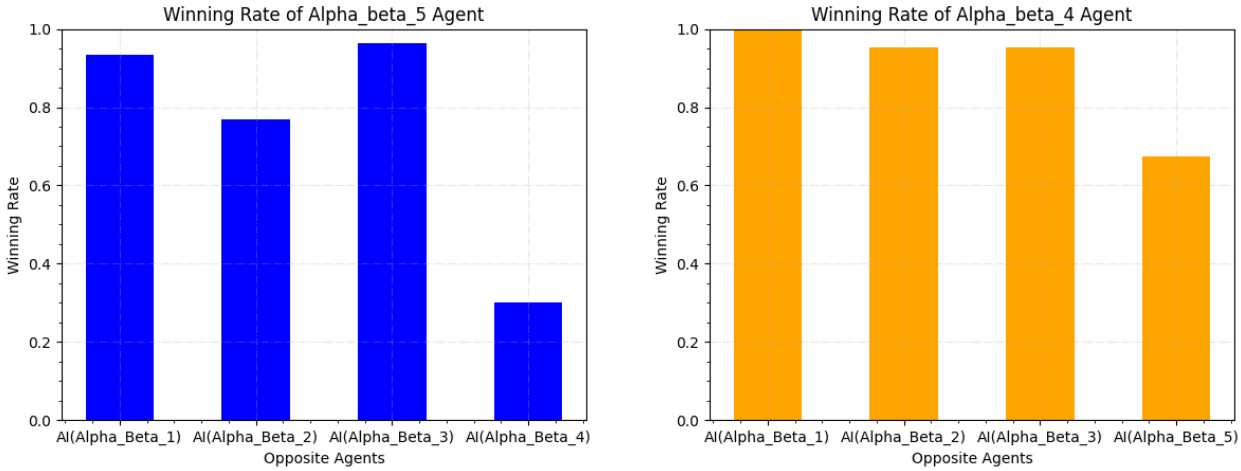


Figure 12: Minimax_5 and Alpha_Beta_5 against baseline agent

As the reader can see from the graph, the difference in win rate between two agents of the same depth in each group is minimal, which is basically in line with the expectation. The difference between the two agents is only a time difference per move. The slight difference in win rate between them, which is within 10%, is due to the fact that there is a certain amount of randomness in the behaviour of all agents.

If presented with precisely the same situation, the original agents must have come up with exactly the same actions. This repetition leads to many games having the same process, which is inconsistent with the independence of each game. The solution of this is in the selection of scores, as by the nature of the Python language, if two moves have the same score when selecting the highest scoring move, then the final outcome must be the move in the first order. Therefore, when moves with the same score are encountered, a move is randomly selected to execute, giving more variety to the game without affecting the strength of each agent.

After proving that Alpha_Beta has the same performance as Minimax, the best Alpha_beta agent will likely be Alpha_beta_5 at the highest depth. Alpha_beta_5 agent was used directly against the other 4 depth agents to verify the hypothesis. However, the results are shown in Figure 13 below.



(a): Winning rate of Alpha_Beta_5 agent

(b): 4 Winning rate of Alpha_Beta_4 agent

Figure 13: Alpha_Beta_4 and Alpha_Beta_5 agents play with other Alpha_Beta agents

The graph on the left (a) shows the winning rate of the Alpha_Beta_5 agent against agents of other depths. It can be seen that the win rate of the agent with depth 4 is only 30% in 1000 games, which is far below the expected value and indicates that his strength is lower than the Alpha_Beta_4 agent. Therefore, with the results of the test (b) on the right, a second test of the Alpha_Beta_4 agent was conducted in the same way. The results show that the Alpha_Beta_4 agent does perform better than the Alpha_Beta_5 agent over the combined 4 sets of 1000 games. Therefore, the Alpha_Beta_4 agent will be used for subsequent tests.

This result also raises the question of why agents with a deeper exploration tree do not perform as well as those with a shallower one. The possible reason for this is that there is some minor unnoticed bug in the implementation, which affects the performance of the Alpha_Beta_5 agent for unknown reasons. However, since the results from the first four agents are as expected, and the difference between them is simply a substitution of the depth parameter in the input, this reason is not particularly likely. Or it could be due to overfitting of algorithm. Further testing and discussion of this issue may be required, but we will not address it in this project.

C.2 Monte-Carlo tree search

In the Monte-Carlo tree search algorithm, there are three factors that affect the strength of the agents at different stages mentioned in III.B.2, the first being the method used in the third stage of the simulation. The second is the hyper-parameter in the equation used to select the child nodes in the first stage, and the third is is how many times the complete 4 steps will be looped. These three factors are tested in order to select the best MCTS agent.

In the third simulation stage, the agent needs to simulate the whole game in an extremely short time, so any complex algorithm is not suitable here. Typically, the whole simulation would be done directly here using random selection, but in IV.B, Roxanne was seen to have a much higher win rate with the same swift moves. In

theory, Roxanne will perform better if used in the simulation stage. The 1000 games were played between two MCTS agents with exactly the same algorithm except for the third simulation step to test this hypothesis, and the results are shown in Figure 14,

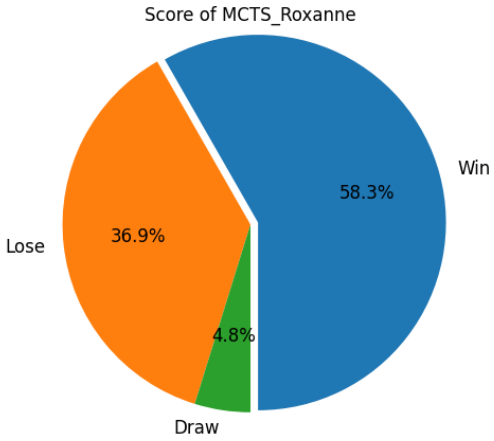
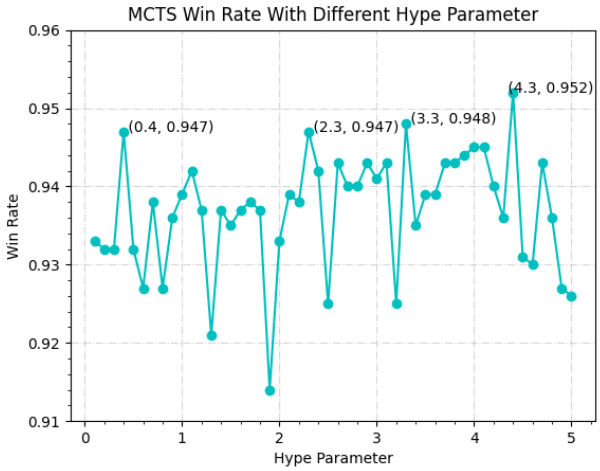


Figure 14: Win/Loss Ratio for MCTS_Roxanne versus MCTS_Random

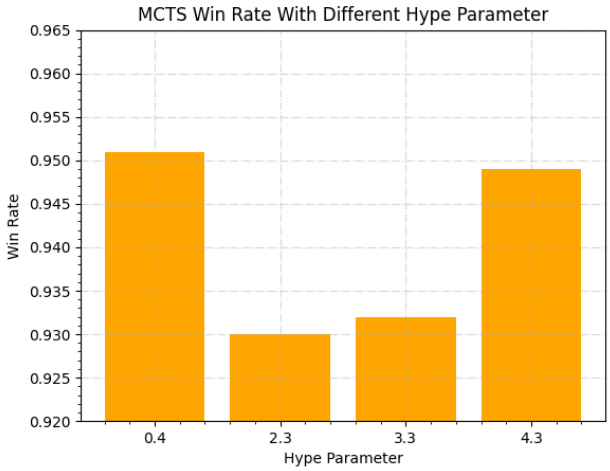
You one can see that although the improvement is not particularly large, the 58.3% win rate is already a considerable advantage, so Roxanne is indeed better algorithm here.

In the previous section on implementation III.B.2, we mentioned the hyper-parameter in the MCTS algorithm, which is an essential factor that affects the strength of the MCTS agent. There is no fixed range of values for this hyper-parameter, so it could theoretically be any number. However, as the evaluation function's scoring system uses scores around 5, it was decided to test all possibilities between 0 and 5 with an interval of 0.1 to not oversize this part of the scale.

The test is divided into two parts. First, the MCTS agent plays 100 games against the baseline agent using all numbers between 0 and 5, separated by 0.1 as hyper-parameters. Then, the hyper-parameter that has an excellent win rate is selected. The reason for using 100 games is that there are 50 agents in the test, and it would take too long to play 1000 games. After an initial screening of 100 games, the selected hyper-parameters were again played against the baseline agent for 1000 games to select the most suitable hyper-parameters. The results of the two parts are shown in Figures 15(a) and 15(b) below,



(a): hyper-parameters with 0 to 5



(b): 4 hyper-parameters with outstanding results

Figure 15: MCTS agents with different hyper-parameters versus baseline agents

As the reader can see in the graph on the left, 4 hyper-parameters in the initial 100-game round had significantly higher winning rates than the others, at 0.4, 2.3, 3.3, and 4.4. Therefore, in the second 1000-game round, these 4 hyper-parameters were tested again. As one can see on the right, the agent with a hyper-parameter of 0.4 had a slightly higher win rate than the other 3 agents, and the MCTS hyper-parameter was determined to be 0.4 in all the further tests.

The last one is the total number of loops: theoretically, the more loops, the more simulations, the better the final move, but the longer it takes. The 7 levels of loops are used here: 50, 100, 200, 300, 400, 500, and 1000. Since there are not very many possible moves in a single board state, 1000 loops are more than enough to cover all the good moves multiple times, and if loops continue to be added, it will make the test take too long. The exciting findings are shown in Figure 16,

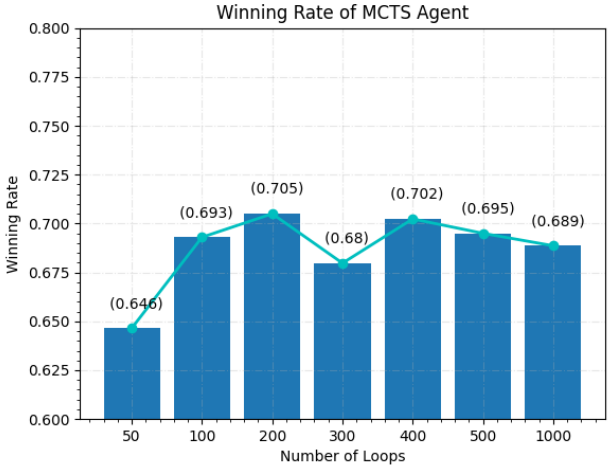


Figure 16: Winning Ratio for MCTS_Roxanne with different loops

The graph shows that the agent with the highest win rate is not MCTS_1000 with the maximum number of loops; instead, there is a significant drop after 200 loops making MCTS_200 the agent with the highest win rate. The reason for this could be that after a certain number of loops, when using the formula mentioned in III.B.2 Equation 2, the second term is almost equal to zero because it is the size of the visit of the best move. When running many simulations, the best move will unavoidably have failed simulations and pull down the average. This occurrence gives some of the lesser moves a chance to catch up.

Finally, the MCTS agent we use for the final test will be MCTS_Roxanne_200 with hyper-parameter 0.4.

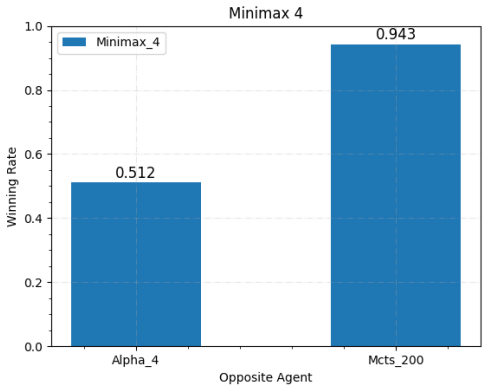
C.3 Double Deep Q-network learning

There are three main factors that affect the performance of a DDJN agent: one is the number of hidden layers, the second is the number of nodes in each hidden layer, and the last is the amount of time spent on training. However, all three aspects require a lot of time and resources to experiment with and get correct. For example, each adjustment to the structure of a neural network, both in terms of the number of hidden layers and the number of nodes, requires complete and extensive training before the results of that adjustment can be tested. Unfortunately there is not enough time and resources to test different configurations of DDQN from the ground up. Therefore, as mentioned in III.B.3, the DDQN agent is built based on an architecture of 10 hidden layers, with 8 convolutional layers and 2 connection layers, as described in other papers which shows that this construct has the best overall performance (Liskowski et al. 2018).

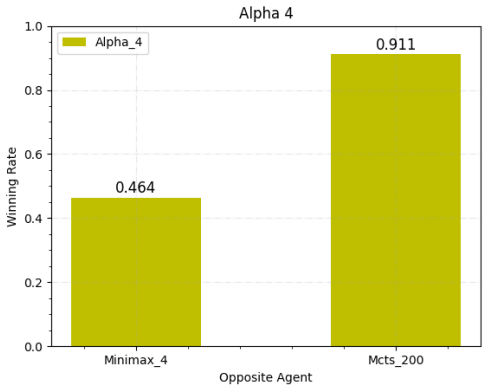
D Strongest agent

Eventually, after each algorithm has selected the best agent, these agents can perform a final test for the strength of Othello.

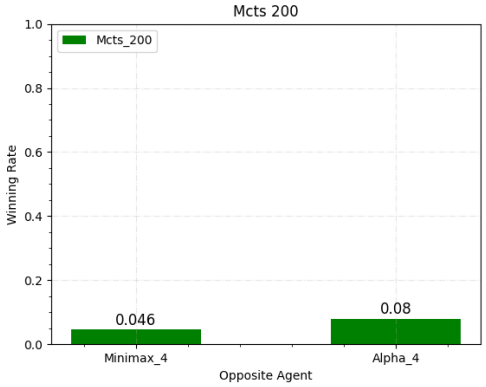
The three traditional agents, Minimax, Alpha_Beta and MCTS, were used to play 1000 games against each other to find what the best traditional algorithm for Othello, in terms of win rate and time. Firstly, the results for the winning rate are shown in Figure 17,



(a): Minimax versus Alpha_Beta and MCTS



(b): 4 Alpha_Beta versus Minimax and MCTS



(c): MCTS versus Minimax and Alpha_Beta

Figure 17: 3 agents play to each other

As mentioned in IV.C.1, Figures 17(a) and (b) proves that Alpha_Beta and Minimax have the same selection of moves, and their win rates against the other two agents are very similar, with a variance of only about 5%. However, an astonishing result is presented in 17(c), where the MCTS agent has a very low win rate against both other agents, totally defeated by the other two. This defeat led to concern that if there was something wrong with the implementation. However, as before with IV.C.2, when testing the MCTS agents for differentiation of the loops, the MCTS agent does beat the selected baseline agent perfectly, which means that the whole code works without problems.

The time spent by the 3 agents in each step of the game was extracted, and averaged to see how they performed throughout the game. These results are in Figure 18

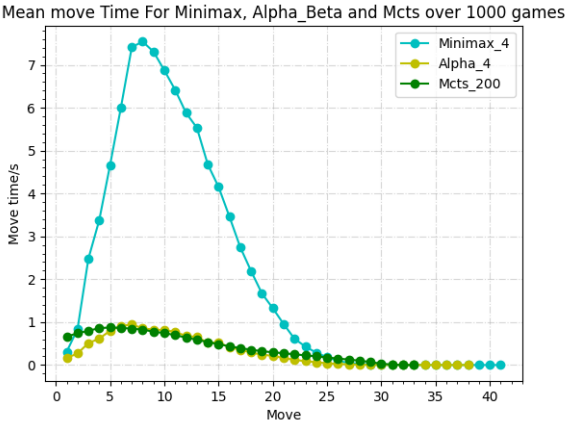
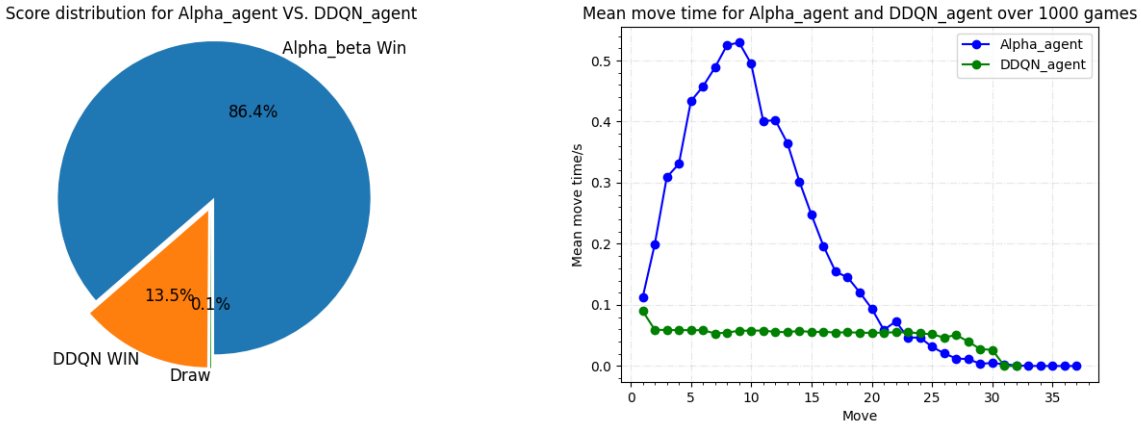


Figure 18: Mean single move time of 3 agents

Due to the previous winning ratio comparison, we are now focusing on Minimax and Alpha_Beta agents. Therefore, with similar overall winning rates, the Alpha_Beta agent has an absolute advantage in terms of time spent, reducing the average time of the Minimax agent from a maximum of 7.6 seconds to 0.9 seconds over 1000 games. In the end, the best traditional algorithm for Othello, considering both win rate and time spent, is Alpha_Beta with deep 4.

E Traditional algorithm versus Deep reinforcement learning at Othello

A final 1000-game set was played between the best selected traditional algorithmic agent, Alpha-Beta, and the DDQN agent to confirm the suitability of Deep reinforcement learning for the Othello game. The following Figure 19 shows the win rates and time usage of these 1000 games.



(a): Score distribution for Alpha_agent versus DDQN_agent

(b): Mean time on each moves

Figure 19: The Alpha_Beta agent versus DDQN_agent

The final results on 19 (a) confirm the view expressed in other studies that deep reinforcement learning is not suitable for board games. Although one can see from 19 (b) that the DDQN agent is applied directly to the current board state using the results of numerous training sessions. Therefore, each moves takes very little time and can gain a significant advantage over the fastest traditional algorithm, Alpha_Beta. However, for the same reason, DDQN makes choices based on the existing environment, i.e. The current state of the board. For any board game, including Othello, the current result of a single move does not determine how good or bad that move is. The same move may end up with the completely opposite result if several subsequent moves are chosen differently. So to play the board game well, the agent must be able to simulate and analyse the results after multiple moves, much like a deep exploration tree in a traditional algorithm. Therefore, more advanced deep reinforcement learning algorithms fail to outperform traditional algorithms in board games.

V CONCLUSION AND FURTHER STUDY

A Conclusion

This project successfully implemented a complete Othello game environment, including a graphical interface, as well as several basic strategy agents used include Random, Maximise the score, Minimise the score, and Roxanne, several traditional AI algorithms, Minimax, Alpha_Beta, Monte-Carlo tree search, a deep reinforcement learning algorithm, and Double Deep Q-network learning. The various AI agents were allowed to play against each other in a game environment with a limit of 30 seconds per step to find the most appropriate agent for the Othello game,Alpha_Beta with the depth 4.

Among them, the best basic strategy agent was Roxanne, which was well ahead of the other basic strategy agents in terms of win rate using an evaluation matrix that combined many strategies. The Minimax agent also showed a powerful game strength, even on a similar level to Alpha_Beta's win rate. However, it was not as fast as the Alpha_Beta agent in terms of time spent on each move. As for the MCTS agent, after we replaced his

simulation step with a strategy using Roxanne instead of random, the overall performance was in fact not as impressive as the other two AI agents, but it was still strong enough to crush all the basic strategy agents. The champion agent, Alpha_Beta, is the winner in terms of strength of the gameplay and time spent on each move. It combines the strength of Minimax in terms of win rate with the method of pruning the tree to significantly reduce the time spent exploring the whole tree.

After many number of training sessions, the final DDQN agent has a speed of play that is as fast as the basic strategy agent. However, it cannot match the strength of the traditional AI algorithm because it has the same drawback as the basic strategy, such as the ability to see the game situation after multiple moves, which is very important in board games.

B Thinking and further improvement

There were several question encountered throughout the project. First, when testing the Minimax agents, it was found that they showed very similar strengths at depths of 2 and 3 respectively. This finding does not follow the general logic that deeper tree exploration, leads to a more robust algorithm. After carefully reviewing the code implementation, no bugs were found in the Minimax code; therefore, the problem may lie in implementing the algorithm itself or maybe due to overfitting. The hypothesis that the different depths represent different player identities at odd and even depths was also mentioned in the context of the project, but more experiments will be needed to verify this.

The second question is that when testing Alpha_Beta agents of different depths, the depth-5 agent was not as strong as the depth-4 agent. Again, this does not follow the logic of strength versus depth. After reviewing the implementation, no possible bugs in the code were found. The reason for this part needs to be found later by testing more and deeper depths of the Alpha_Beta agent.

The last question concerns the strength of the MCTS agent, which, given the natural complexity of the algorithm, is not supposed to be so different from the other two agents in terms of strength. A possible reason for this is that the other two algorithms are slightly less complex than MCTS, making it more likely that they can achieve the extremes that the algorithm is meant to perform. In contrast, the MCTS algorithm has more parameters to tune and a more comprehensive range of options for these parameters. The actual optimal parameters may be outside this project's scope, resulting in the test results not being representative of the maximum strength of the MCTS algorithm itself. It is also possible that the formula used in the selection step of MCTS may significantly impact the overall performance level. If more advanced selection formulas could be found, the overall strength of MCTS could probably be improved.

If more accurate results were to be obtained, the whole project could be improved in several ways. For example, the whole project was carried out on a single hardware environment, and sometimes different algorithms may behave differently on different hardware. Additionally, if more time was available, a greater number of games could be used in the tests to reduce the impact of random elements on the overall strength of the algorithm. Finally, it is also possible to connect the application to the API interface of some major online platforms, allowing the agent to play against other high level agents to obtain the agent's strength level crossing the world.

References

Abed-alguni, B. H. (2017), 'Bat q-learning algorithm', *Jordanian Journal of Computers and Information Technology (JJCIT)* **3**(1), 56–77. 7

Arora, M. (2021), 'Advantages of python programming language in the world of big data: poster poster abstract', *Journal of Computing Sciences in Colleges* **37**(3), 170–170. 8

Barber, A., Pretorius, W., Qureshi, U. & Kumar, D. (n.d.), 'An analysis of othello ai strategies'. 1

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. & Colton, S. (2012), 'A survey of monte carlo tree search methods', *IEEE Transactions on Computational Intelligence and AI in games* **4**(1), 1–43. 6

Campbell, M., Hoane Jr, A. J. & Hsu, F.-h. (2002), 'Deep blue', *Artificial intelligence* **134**(1-2), 57–83. 1

Cherry, K. A. (2011), 'An intelligent othello player combining machine learning and game specific heuristics'. 4, 5

Dahmani, D., Cheref, M. & Larabi, S. (2020), 'Zero-sum game theory model for segmenting skin regions', *Image and Vision Computing* **99**, 103925. 4

Fan, J., Wang, Z., Xie, Y. & Yang, Z. (2020), A theoretical analysis of deep q-learning, *in* 'Learning for Dynamics and Control', PMLR, pp. 486–489. 7

Haenlein, M. & Kaplan, A. (2019), 'A brief history of artificial intelligence: On the past, present, and future of artificial intelligence', *California management review* **61**(4), 5–14. 1

Iwata, S. & Kasai, T. (1994), 'The othello game on an n× n board is pspace-complete', *Theoretical Computer Science* **123**(2), 329–340. 2, 5

Kumar, N., Kumar, P., Sandeep, C., Thirupathi, V. & Shwetha, S. (2019), 'A study on deep q-learning and single stream q-network architecture', *Int J Adv Sci Technol* **28**(20), 586–592. 7

Liskowski, P., Jaśkowski, W. & Krawiec, K. (2018), 'Learning to play othello with deep neural networks', *IEEE Transactions on Games* **10**(4), 354–364. 10, 17

Othello Game Rules (n.d.), [OL]. `https://www.ultraboardgames.com/othello/game-rules.php` Accessed Aug 4, 2022. 1

Sannidhanam, V. & Annamalai, M. (2015), 'An analysis of heuristics in othello'. 5

Stančin, I. & Jović, A. (2019), An overview and comparison of free python libraries for data mining and big data analysis, *in* '2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)', IEEE, pp. 977–982. 8

Wang, F.-Y., Zhang, J. J., Zheng, X., Wang, X., Yuan, Y., Dai, X., Zhang, J. & Yang, L. (2016), 'Where does alphago go: From church-turing thesis to alphago thesis and beyond', *IEEE/CAA Journal of Automatica Sinica* **3**(2), 113–120. 1

Watkins, C. J. & Dayan, P. (1992), 'Q-learning', *Machine learning* **8**(3), 279–292. 7