Creating Othello AI Players Combining Heuristic Evaluation And Reinforcement Learning

- Final Report

Author: Lida Wen

Supervisor: Bailin Deng

Moderator: Hantao Liu

Module: CM3203 – One Semester Individual Project (40 Credits)

Institution: School of Computer Science and Informatics, Cardiff University

Date of Completion: 27th of May 2022

Abstract

The aim of the project is to evaluate possible approaches to creating AI capable of playing the board game, Othello. This project explores AI algorithms that are able to choose sensible moves in the game, such as Minimax and Reinforcement learning, combining strategic heuristics and Convolutional Neural Networks(CNNs) concepts. The report also demonstrates the implementation of a GUI to investigate the game's state and experiment the intelligent agents. Four types of intelligent agents are created, which are Heuristics, Minimax, MCTS and DQN. Another three test agents have been implemented as the benchmark. Performance is evaluated in terms of win rates against test agents, computational efficacy, and variables of training opponents. In conclusion, the adversarial heuristics models had superior results in the experiments because of the particularity of certain strategies in Othello, and the performance is sensitively affected by the choice of heuristics. DQN agent would perform better if I had a higher quality training model and hardware. The purpose of this project is not to create the strongest AI but to demonstrate the implementation and comparison of multiple models in order to contribute to the further improvement on the effectiveness of Othello AI player development.

Acknowledgement

Sincere thanks for all the kindly advice from my supervisor Dr.Bailin Deng, and also for giving me the opportunity to work on this project. Thank you for the practical suggestions proposed in the meetings.

I would like to extend my sincere thanks to Dr.Hantao Liu, who is willing to spend time moderating this project, and supported me by giving feedback on the Initial plan with appropriate comments.

Table of Contents

Table of Figures

Section - 1 Introduction

Artificial intelligence as an innovative field of Computer Science is applied in many areas, such as map navigation, facial detection, recommendation algorithms, and playing board games. The board game AIs such as AlphaGo and AlphaZero, are designed and trained to play board games against masters, they play and analyse the games, evaluating the possible moves and therefore choosing the optimal solution. This approach has been used in many two-player turn-based board games such as Chess, Go and Othello.

Othello is a strategy board game with the winning condition of flipping more opponent disks at an 8×8 uncheckered board when the last empty square is taken. Players flip opponent´s disks when the disk just placed formed a straight line with another disk of the current player, and it can be horizontal, vertical or diagonal.

This project aims to explore possible ways of creating an Othello AI player that is capable to make sensible decisions in Othello games. The complexity and interactivity make this topic particularly interesting. To achieve the aim, the project starts with analysing the background, rulesets and strategies of Othello. A GUI has then been designed with Pygame to visualise the game, with the implemented game rule.

The project has separate approaches to programming AI systems, including a heuristics algorithm version of AI agent that has been implemented based on game theory. The key concepts of heuristics strategies for Othello AI involve board logic analysis, corner priority, winning move detection, blocking point detection and pattern recognition. The heuristic algorithm put these techniques together as principles to evaluate the state of the game board.

The heuristics evaluation algorithm could combine with Minimax, a backtracking algorithm that is very commonly used in two-player turn-based board games. It uses recursion to search through the tree nodes and makes decisions based on heuristics evaluation to maximise the score of moves, assuming the opponent also makes optimal moves. Also, the alpha-beta pruning algorithm has been applied to decrease the number of nodes evaluated in the search tree. In comparison with native Minimax algorithms, this optimization reduces the search time by ignoring unnecessary nodes.

Another AI agent is the DQN agent trained using PyTorch. In reinforcement learning, neural networks such as CNNs have been widely used in video games, it allows the learning agent learns from video frames and accumulates cumulative rewards. Unlike supervised learning, RL does not require sub-optimal actions to be exactly corrected, it focuses on exploring a balance between exploration and exploitation. In this project, I trained multiple DQN agents with different agents as training opponents over 15 days, by letting them play hundreds of thousands of games themselves.

After the design and implementation sections, the implemented AI players are then put into practice. They have been tested through play against each other, and against benchmark agents.

Patching the AI algorithms and GUI together makes it a deliverable outcome of the project. Users could use the programme to play against customizable AI with hierarchical difficulty, or enhance the theoretical understanding of this topic by testing AI vs AI. My hope in this project is to demonstrate the implementation in detail which could potentially be helpful for those who are designing reinforcement learning models with similar feature demands.

Section - 2 Background and Research

The purpose of this section is to introduce the basic concepts of Othello and gather the ideas of game strategies based on research. For the preparation of the project, the research started by studying the basic rulesets of the game and advanced strategies along with game theory. I made my research on the topic of popular board game AI algorithms and implement machine learning on the board games. Before moving on to the approach and design section, more relevant detail based on research provides here to better understand this topic.

2.1 Background of Othello

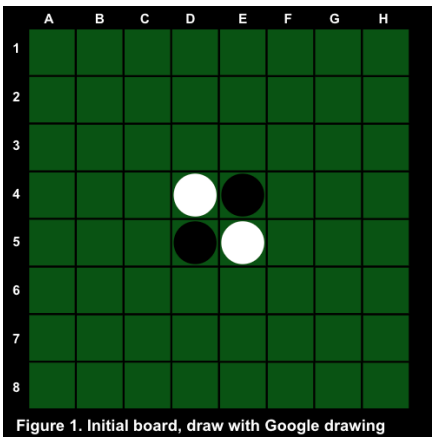


Figure 1. Initial board, draw with Google drawing

Othello, also known as Reversi, was invented during the Victorian era by the British and evolved into the modern version of rulesets in 1971, which was patented by a Japanese salesman, Goro Hasegawa[1].

The game is renamed Othello which referenced the famous tragedy written by William Shakespeare. Othello was the main character in the story who has black skin, while his wife has white skin. He killed his wife himself because he was provoked by villains and his distrust. He regretted it and then committed suicide. This is the origin of the name Othello, which implies racial disputes in society among people with different skin colours.

2.2 Basic Rulesets

As an abstract strategy board game, Othello is played by two players on an 8×8 board. The board always has a greenish background colour and 64 squares on the board, corresponding to 64 pieces. The pieces in this game are called “disks”, which are white on one side and

black on the other. Each side of the disk represents one of the players, the player put the disk in the centre of the square to make moves.

Initially, 4 disks are placed in the central squares which are D4, E4, D5 and E5 shown in Figure.1, and the positions of the initial disks are fixed. Always Black side makes the first move. Since 4 squares are taken and the square can only be refilled by flipping the disk that is already on the board, the maximum number of moves in one game is 64 - 4 = 60 moves, while the minimum number of moves to end a game is 9 moves in a regular game.

Players flip the opponents' disks to their colours by placing a disk next to an opponent's disk and forming a line, this can be done horizontally, vertically or diagonally. The goal is to have the majority of disks turned to display one's colour when the last playable empty square is filled. The game alternates by turns and players can only make a move when there are disks being flipped. If the current player cannot make a valid move, another player takes the move, and if both players cannot make a valid move, the game ends before all the squares are filled.

2.3 Advanced Strategies

To study the strategies, I purchased a downloadable electronic edition of Othello learning materials on Baidu Wenku[2], which explains in detail the tactical strategies of Othello in the beginning, middle and end of the game.

Firstly, corners are important in this game, because they cannot be flipped once it is placed. If two adjacent corners have been taken by one colour, it will own the connecting side. Therefore the strategy is to avoid making moves on squares that are adjacent to corners, especially B2, G2, B7, and G7 on the board since the inner board(Interiors) has always been filled in early games, making these might result in losing the corners.

New players usually like to make moves that flip more disks in that turn, which is namely Maximum Disks Strategy. It is not always a good strategy, because the disks can be easily re-flipped later, and the game often finishes when all squares are filled. By the same token, 4 sidelines(Frontiers) are also important as it is more stable, which makes the rows and columns adjacent to the sides less desirable.

Another situation that needs to be avoided is leaving an empty square within the surrounding ally disks. For example:

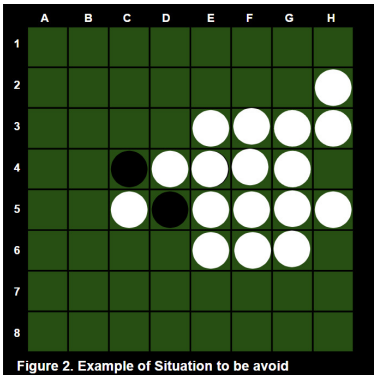


Figure 2. Example of Situation to be avoid

In this case, if the Black disk takes H4, the Black side will have an advantage in this game. The H1 corner will be taken at the next round, thus the entire H side squares. Whiteside can do nothing to stop it. This is called as Minimum Frontier Strategy.

The Degree of Divergence(DoD) in Othello means the number of empty squares around a disk. Every time we make a move there must be disks been flipped, combing the DoD of placed disk and flipped disks, we get Field of Divergence. This is a theory proposed by Murakami Ken, who has won the World Championship of Othello. A move with a total DoD of 1 is usually a good move. However, there are exceptions to this statement, which can not be regarded as inevitable.

With the innovation of technology, people have discovered the potential of applying AI to board games.

2.4 Milestones for Othello AI development

The first Othello AI program was invented back in 1977 by N.J.D.Jacobs[3], when the development of all board game AI has been boosted. “Logistello” beat the world champion, Takeshi Murakami, in 1997, when it has played over 100,000 games against itself. Also exists an interesting AI that lost 1.29 million games and has paltry 4,000 wins. And this is what the author expected, which is encouraging people to play this game by winning against AI.

The idea of applying machine learning to board games AIs has always been a popular topic since this suggestion has been made. The success of Alpha Go again proved the potential of reinforcement learning, where Convolutional Neural Networks(CNNs) allow it to become smarter at learning and decision-making over time.

2.5 Research Aim and Objective

The aim of this project is to explore possible approaches to creating and designing an AI player for playing Othello games. Considering the limitations in the hardware, the expectation of this project is not to design a new generation of the strongest AI player, but hoping to demonstrate the implementation of approaches and design , which would potentially benefit those who are designing reinforcement learning models with similar features requirements. Specific concepts of design and approaches will be explained in detail in the next section.

Section - 3 Approaches and Design

This section contains information on the theory of possible approaches to creating an Othello AI player, including designing the UI, implementing the Game logic, utilisation of Heuristics, Minimax, Alpha-beta pruning, MCTS and Reinforcement learning concepts such as Markov Decision Process(MDP), Convolutional Neural Networks and DQN.

3.1 Programming Language

Python 3.10.2 is used in this project, it is a high-level interpreted language that reads like daily English, thus potentially can make the AI development process easier and less complex. It is also one of the most popular programming languages for artificial intelligence and machine learning. It provides a sufficient choice of libraries for AI and MR projects, which saves time from coding the base-level components. Pygame is the open-source module that will be used to visualise the UI that implements the rules of the game. It specialises in creating video games and multimedia applications based on the Simple DirectMedia Layer(SDL) library. PyTorch module was used to build the Conv layer of CNNs.

3.2 UI Design

In order to visualize the progress of the game and make it easier to understand for the users of the final program, the UI to play Othello has been implemented.

3.2.1 Visual Design

The user interface was designed by the Pygame module in Python. A menu page has been implemented contains linked buttons to other pages of each AI algorithm.

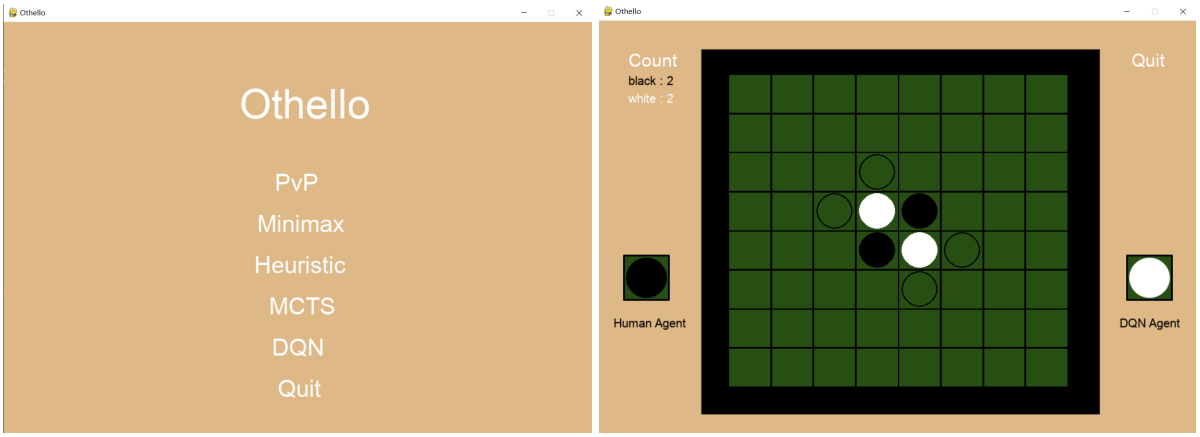


Figure.3 Demonstration of UI design

All images used in the program are drawn by Google drawing, and the font used in the program is Arial MT. The game board has a dark green background which is followed by the official regulations of Othello board standards, while the main background of the window has a light beige colour to highlight the dark themed board. The empty circles on the board represent valid moves that can be made for the current player.

3.2.2 Approach to Implementing the Game Logic

To implement the rules of Othello, firstly the board image has been imported to the blank page and then the position of each centre point of squares is located by calculating from the x-coordinate and y-coordinate of the Pygame window. Noted that the origin (0, 0) of the windows is at left-top the corner, and as it moves rightward the x value increase, moves downward the y value increases. Then an empty 8×8 2D-array was created corresponding to the coordinates found:

```
board =  [[0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0]]
```

```
board[3][4] = 1
board[4][3] = 1
board[3][3] = 2
board[4][4] = 2

where  1 represents Black,
       2 represents White
```

In this way, an initial state of the Othello board has been created. The game always starts with the Black side. For example, assuming the Black player clicks the board on UI, we use

```
event.type == pygame.MOUSEBUTTONDOWN
```

to detect clicked coordinates, and then calculate if the clicked point is on the board and to which square of the board it belongs. The clicked square display a black disk, and the corresponding element in the array turns into 1. Othello is two-player turn-based, the offensive side swaps each turn unless there is no valid move that can be made by one player or both.

The most important functions in implementing the game rules are to detect valid moves that can be made and flips the disks between placed disk and another disk with the same colour. Initially, I used the While loop to check 8 directions separately. Then later I found it is less complex to use For loop with assigning argument:

```
directions = ([[0, -1], [-1, -1], [-1, 0], [-1, 1], [0, 1], [1, 1], [1, 0], [1, -1]])
```

Only moves that flip other disks are considered legal move, loops the program traverses the board array and checks whether the placed disk is able flips other disks horizontally, vertically, or diagonally.

The detected valid moves are added to a list and displayed as an empty circle on the board UI. By traversing the board array we could also get the total number of current disks for each player, which displays on the window for users to analyse the state of the game. Functions such as game end judgement, remaining moves etc. have also been created and will be used later for developing the AI agents.

3.3 Heuristics

The heuristics algorithm as a static evaluation function has been frequently used in traversing the search space based on given conditions in a given environment. This algorithm allows problems to be solved in a relatively quick way that provides sufficient results in given time constraints.

3.3.1 Valuation of the Game Board

Combing the rulesets and advanced strategies discussed in sections 2,2 and 2.3, a basic valuation table are proposed:

```
board_valuation =   [8, 1, 7, 6, 6, 7, 1, 8],
                    [1, 0, 2, 3, 3, 2, 0, 1],
                    [7, 2, 5, 4, 4, 5, 2, 7],
                    [6, 3, 4, 0, 0, 4, 3, 6],
                    [6, 3, 4, 0, 0, 4, 3, 6],
                    [7, 2, 5, 4, 4, 5, 2, 7],
                    [1, 0, 2, 3, 3, 2, 0, 1],
                    [8, 1, 7, 6, 6, 7, 1, 8]
```

Larger in number means the position is more desirable. If there is more than one valid move in the list, the most desirable position will take precedence. Corners are most valuable, they cannot be flipped once placed. In terms of stability, the squares in the middle of the sides are the second most valuable. Accordingly, the squares adjacent to the corners and sides are less valuable, because taking them will face the risk of losing corners and sides.

In case there are multiple positions that have the same valuation in the valid move list. The move that would flip the least disks will be selected. The reason for picking ‘least move’ instead of ‘most move’ can be referred to Degree of Divergence theory. A move that eliminates Degree of Divergence would be considered more stable in most situations, as it restricted the opponent's mobility. Maximum Disks Strategy is the heuristics to be avoided.

3.3.2 Exploitation of the Game Theory

For a stronger AI player, I have proposed the following evaluation functions to be included in the program, which have higher precedence than the valuation table[4]:

Winning Move Detection - assuming a move that wins the game exists, regardless of the valuation, this move is prioritised

Blocking Move Detection - detects the move that makes the opponent has no valid move in the next turn, this move is secondly prioritised, and has higher precedence than the valuation table

Cutting move Detection - detect the move that cuts through the opponent's disks, and cannot be re-flipped once the move is executed. Professionally, the moves that have 0 Degree of Divergence, for example:

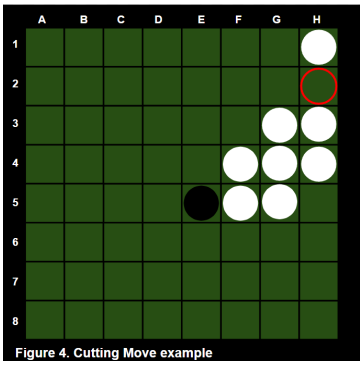


Figure 4. Cutting Move example

Assuming it is the Black side to move, there are two valid moves that can be made, which are H5 has a higher valuation of 6 and H2 has a paltry valuation of 1. Choose a cutting move such as H2 gives the Black side initiative without risks.

Side Neighbour Detection - avoid moves on the sides that have an adjacent neighbour of the opponent's disk, such as H5 in Figure.4, which are worthless moves that could be re-flipped immediately and makes the opponent's disks more stable on the side

Although the strategies have been discussed as general game theory in many Othello's research, I create my own version of heuristic functions by combining my understanding to the strategies, which makes it differ from common ones.

3.3.3 Precedence Hierarchy of the Evaluation Functions

The above evaluation functions and the valuation table are evaluated every turn before a move is made. The order of priority of the functions are

Winning Move > Blocking Move > Cutting Move > Side Neighbour > Valuation table
> Least Move (the move that flips the least disks)

If none of the above conditions is met, and there are multiple moves that have the same valuation and flip the same number of disks once placed, it will be randomly chosen.

3.4 Minimax

The Minimax algorithm is a backtracking depth-first search recursive algorithm that uses recursion to search through the tree nodes and makes decisions based on the heuristic evaluation function to maximise the chance of winning, and reduce the possibility of loss in the worst-case (maximum loss) situation, assuming the opponent also makes optimal moves, which minimise the chance of winning (of another player).

3.4.1 Approach to Implementing Minimax in Othello Game

For an easier understanding of the logic of Minimax, this is an example game tree:

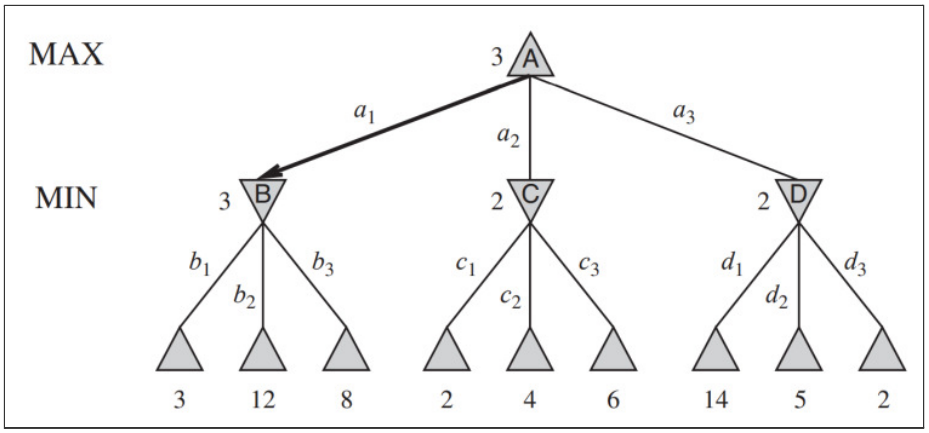


Figure.5 Example of Minimax Game Tree

In this case, we assume that node A is a root state, that branch to child B, C, and D through a_1, a_2, and a_3. Each child node also can branch to another three grandchild nodes. This algorithm shown in the example is considered as having a search depth of 2 and branching factors of 3.

Assume the numbers next to nodes represents the valuation obtained by the heuristics evaluation function. Assigning the move that maximises winning chance positive infinity(+∞), and the move that minimises winning chance negative infinity(-∞). At level 1, the second layer of the branch shown in Figure.5, the algorithm will pick, for each node, the

minimum value of child nodes and assign it to that same node. In the figure, node B picks the smallest number between 3, 12 and 8, therefore assigning the value 3 to itself. Then in level 0, which is the node A, will choose the maximum value between B, C, or D, thus assigning itself 3. Then the algorithm makes the decision of move from node A to node B which is the optimal move that minimizes the maximum possible loss.

In the case of a higher and wider game tree consisting of more branch factors and more depth of searching, the algorithm continues picking the max and min values of the child nodes, until the value of the root node has been assigned.

3.4.2 Logic for Minimax Algorithm

Pseudocode for implementing Minimax in Othello Game:

```
function minimax(node, depth, maximizingPlayer) is
        if depth = 0 or node is a terminal node then
                return the heuristic value of node
        if maximizingPlayer then
                value := −∞
                for each child of node do
                        value := max(value, minimax(child, depth − 1, FALSE))
                 return value
        else (* minimizing player *)
                value := +∞
                for each child of node do
                        value := min(value, minimax(child, depth − 1, TRUE))
                return value
```

The algorithm returns the heuristic value for leaf nodes, including terminal nodes and nodes at maximum search depth. The search depth determines the quality of the final decision, so-called a depth-first algorithm[5], with a depth-limited search procedure. The suitable search depth of the program will be tested in section 5.

Up to this point, the introduced Minimax are still insufficient in terms of efficiency. Although the search depth has been limited, there are still many branching states that need to be explored in each search. For more complicated games, such as Go (with branching factors as high as 300+ during the first 40 moves) or chess (with branching factors of around 30), the difficulty of searching the tree nodes is significantly higher, in comparison to Othello (with the average branching factors around 10).

3.4.3 $\alpha - \beta$ Pruning

Therefore, to optimise the Minimax algorithm, we need to increase the search depth while reducing the search time for a more optimal solution within a reasonable runtime. The alpha-beta pruning was implemented to cut off unnecessary leaves or branches given they can in no way affect the decision of the algorithm. Where α and β indicate:

α: the optimal choice of all MAX nodes, including the root nodes and leaf nodes. If the current optimal value of a MIN node is not greater than this value, the MAX node will not select this node, because the max node on this path has the right to choose other schemes with greater value, thus there is no need to search deeper.

β: the optimal choice of all MIN nodes. Conversely to α, if the current optimal value of a MAX node is not greater than this value, the MIN node will not select this node, so there is no need for a deeper search.

Use the game tree in Figure.5 as an example, the game tree after pruned:

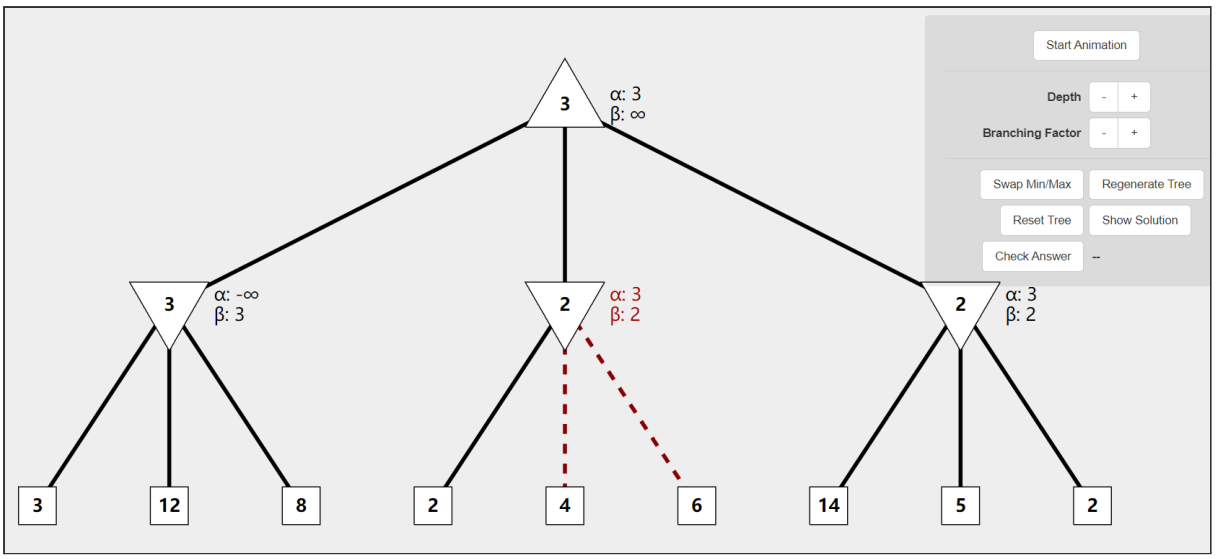


Figure.6 Applying alpha-beta pruning to the game tree(Figure.5)

Procedure:

1. Firstly, the MAX player at node A(root node) has α value as -∞ and β value as +∞, and A pass these alpha&beta values to child node B(node on the left), and B passes the same value to its child.
2. At the bottom-left node, the β value is compared with 3, 12, and 8, where min(+∞,3,12,8) = 3, 3 will be the value of β at node B.
3. Algorithm backtracking to node A, α value will change to 3 as it is a turn of MAX and max(-∞,3) = 3. Node A then have α = 3 and β = +∞
4. Then the algorithm traverse to the next child node of the root node, node C(the middle one), and the value of α = 3 and β = +∞ are passed to this node
5. Again repeat the step 2, the current value of beta will be compared to 2, which min(+∞,2) = 2, hence at node C α = 3 and β = 2, α >= β, so the rest of the child node of C will be pruned, which will not be traversed, the value of C remains 2
6. Repeat step 1 to 5 for the rest of the nodes

The move order of exploring nodes is an important factor that influences the effectiveness of the α&β pruning.

Pseudocode to implement α&β pruning:

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

[6]

Which consists of the following functions, accompanied by the corresponding definition:

S_0: initial State

Player(s): define which player currently making the move

Action(s): return the list of current valid moves

Result(s, a): define the outcome of the Action

Terminal-Test(s): examine whether is the game end or continue

Utility(s, p): define the valuation of State for Player according to the current board

Using α-β pruning reduces branching factors to be searched so as to improve the performance of the Minimax algorithm in the Othello game in terms of computational efficacy.

3.5 Monte Carlo Tree Search

MCTS is another heuristic search algorithm used in Othello AI for decision processes to solve the game tree. Unlike Minimax, MCTS only analysis the most promising moves. It searches the game tree by combining random sampling of the search space.

In a random Othello game, the MCTS algorithm “simulates” many playouts. The playouts are to predict the game state to the very end by picking the random moves for each move. Then based on the playouts, the MCTS evaluates the outcome of the results obtained from the simulations to weight the leaf nodes, therefore the better moves are more likely to be selected depending on how many simulations have been calculated. The neural networks used in MCTS include “policy network” to select the next move to play, and ”value network” to predict the winner of the game from each position.

3.5.1 Operation Logic of MCTS

Each simulation compose of 4 phases:

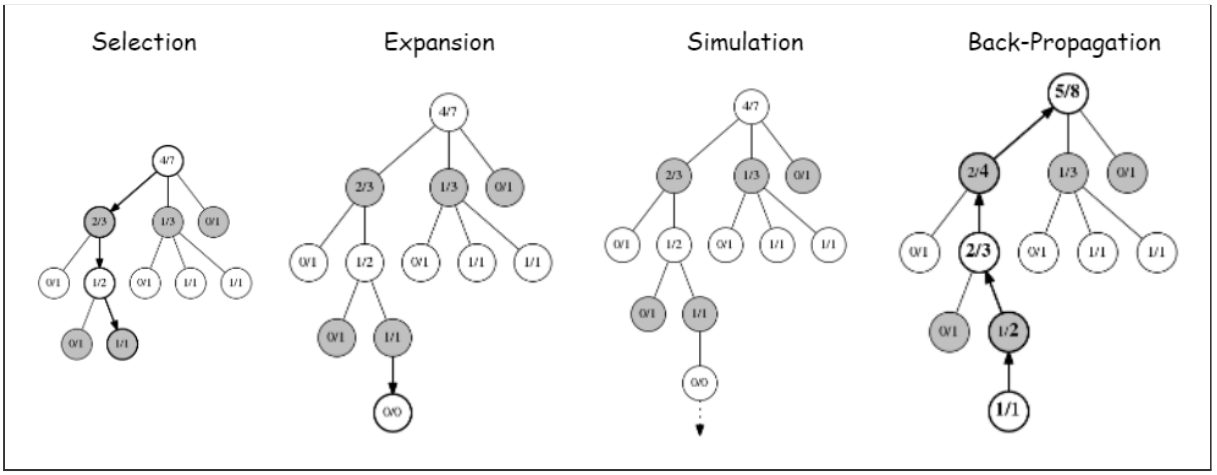


Figure.7 Monte Carlo Tree Search Example[7]

Phase 1 - Selection : start from the current game state(root node) expands in a way of biasing the choice of leaf nodes that leads to the most promising game state.

Assuming there is a "move(**a**)", the parameters used to determine the score of a move are the win rate of **a** and the valuation of **a** given by the policy networks.

Combing two parameters with the formula[8]:

$$score(a) \triangleq Q(a) + \frac{\eta}{1+N(a)} \cdot \pi(a|s;\theta)$$

- N(a) : how many times **a** has been visited, thus initially N(a) = 0. Evey time **a** has been selected, N(a) += 1
- Q(a) : the cumulative value of N(a), determined by win rate of **a** and the evaluation function. The initial value of Q(a) is 0. Every time move **a** is selected, it will update the value of Q(a)
- η : an adjustable hyperparameter
- $\pi(a|s;\theta)$: the valuation given to **a**

For example, if move **a** has not been selected yet, N(a) = 0 and Q(a) = 0. In this case, the score of move **a** are totally depends on $\pi(a|s;\theta)$, which is the valuation score from the original heuristics evaluation. If move **a** has already been selected many times, N(a) would takes higher credit than the original valuation, and the Q(a) would be the main factor that affects score(a). The purpose of coefficient η / (1+N(a)) is to encourage exploration. If there are two moves have similar score of Q(a) and $\pi(a|s;\theta)$, the move that has been selected less times will be selected. For a state(s), MCTS use this formula to calculated score(a) for all valid moves and find the one with highest score, executes this move in the simulation.

Phase 2 - Expansion : assign the selected move as a_t and execute a_t in simulation. The algorithm returns a new state(s_{t+1}) with formula $\mathbb{P}(s_{k+1} \mid s_k, a_k)$. Which predicting opponent's move randomly[9]:

$$a'_t \sim \pi(\cdot|s'_t;\theta)$$

s'_t here represents the current game state from the opponent's aspect, each time a move a_k simulated, returns a new state of s_{k+1}. The MCTS algorithm can only make predictions of the opponent's move based on its own logic, just like humans play Othello in the real world.

Phase 3 Evaluation - start from state s_{k+1} in simulation, the policy networks assume the moves for both players until game end, conclude a reward **r**, when self side wins r += 1, otherwise r -= 1.

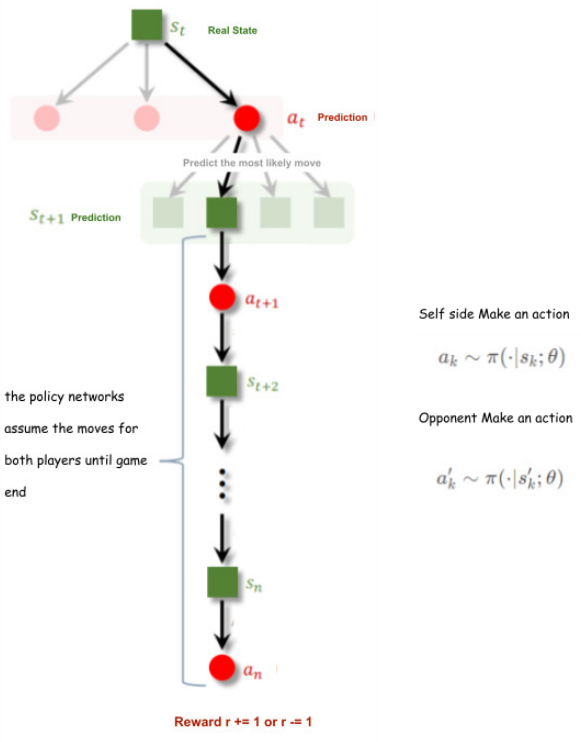


Figure.8 Policy networks play against itself[10]

As shown in the diagram, s_t is the real state, a_t is the move made in simulation, then the opponent makes a move a'_t. New state s_{t+1} has reached, and so on until state s_n, the algorithm evaluates whether it is a good state or a bad state. If good(wins), r += 1, indicates that state s_{t+1} might also be a good state. Conversely for a bad state r -=1. Therefore the reward **r**

reflects the usability of s_{t+1}. Additionally, the value networks can be used to evaluate the state s_{t+1}, the bigger the value of $v(s_{t+1};\omega)$ the better the state is.

Phase 4 Backpropagation - the valuation of next valid moves has been determined in the last phase, recorded as $V(s_{t+1})$, such values are obtained for each run of simulations. Simulations will run many times. Move a_t followed by multiple child nodes with different $V(s_{t+1})$ values, calculate the average value for each a_t as $Q(a_t)$. each time a simulation has been made, $Q(a_t)$ returns a new average value.

These four phases occur in every single simulation, and to make a final decision, MCTS do normally do hundreds, of thousand times of simulations.

3.5.1 Pseudocode for MCTS

```
def monte_carlo_tree_search(root):
        while resources_left(time, computational power):
                leaf = traverse(root) # leaf = unvisited node
                simulation_result = rollout(leaf)
                backpropagate(leaf, simulation_result)
        return best_child(root)

def traverse(node):
        while fully_expanded(node):
                node = best_uct(node)
        return pick_univisted(node.children) or node # in case no children are present / node is terminal

def rollout(node):
        while non_terminal(node):
                node = rollout_policy(node)
        return result(node)

def rollout_policy(node):
        return pick_random(node.children)

def backpropagate(node, result):
        if is_root(node) return
        node.stats = update_stats(node, result)
        backpropagate(node.parent)

def best_child(node):
        pick child with highest number of visits
```

[11]

The Monte Carlo Tree Search method has been used by AlphaGo and AlplaZero to utilise neural network predictions and select actions suitable for the current state. The approach instructed was based on AlphaGo, while AlphaZero has simplifies the procedure by combing the **Expansion** and **Evaluation**, and changed the logic of **Selection** and the function used for making the final decision. The implementation of MCTS in this project applied the ideas of AlphaZero to optimise the computational efficacy, but it still takes a relatively long running time to get a good move.

3.6 Reinforcement Learning and Deep Q-Network

Reinforcement learning(RL) as a field in machine learning that allows AI agents to become capable of learning in an environment for the sake of maximizing the notion of cumulative reward. The difference with supervised learning is that RL is more inclined to on finding a balance between exploration and exploitation using dynamic programming techniques. The environment is referred to as the Markov decision process(MDP).

3.6.1 Introduction to Markov Decision Process

In reinforcement learning, an infinite horizon represents by the interaction between an AI agent and an environment., discounted Markov Decision Process (MDP) $M(\mathcal{S}, \mathcal{A}, \mathbb{P}, r, \gamma, \mu)$. where[12]:

- $\mathcal{S}$ is the **state space**, which may be finite or infinite.
- $\mathcal{A}$ is the **action space**, which also may be discrete or infinite. $\mathcal{A}(s)$ is the action space of the state $s \in \mathcal{S}$.
- $\mathbb{P}(s_1 \mid s, a)$ is the **probability of transitioning** into state $\mathbf{s_1}$ upon taking action **a** in state **s**.
- r is the **reward function.**
- $\gamma \in (0,1)$ is the **discounted factor** and defines a horizon for the problem,
- μ is the **initial distribution** of the state $\mathbf{s_0}$.

In the most general setting, a policy specifies a decision-making strategy in which the agent selects actions adaptively based on the history of observations. For reinforcement learning, there are two types of policies[13]:

- **Stationary Policy**: The policy does not change over time and only depends on the current state. i.e $a_t \sim \pi(\cdot|s_t)$. For example, the stationary deterministic policy is a map from $\mathcal{S}$ to $\mathcal{A}$.
- **Non-stationary Policy**: The policy changes over time or depends on not only the current state but also the history. i.e $a_t \sim \pi(\cdot|s_t, t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0)$.

We now define values for (general) policies. For a fixed policy $\boldsymbol{\pi}$ and a starting state $\mathbf{s_0} = \mathbf{s}$, we define the value function $V^{\pi}: \mathcal{S} \rightarrow \mathbb{R}$ as the discounted sum of future rewards:

$$V^{\pi}(s) := \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^{t} r\left(s_{t}, a_{t}, s_{t+1}\right) | s_{0} = s, \pi\right)$$

Similarly, the action-value (or Q-value) function is defined as:

$$Q^{\pi}(s, a) := \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^{t} r\left(s_{t}, a_{t}, s_{t+1}\right) | s_{0} = s, a_{0} = a, \pi\right)$$

Let $\boldsymbol{\pi}$ be the set of all stationary policies and non-stationary policies. Given a state $s \in \mathcal{S}.$, the goal of the agent is to find a policy that maximizes the value, i.e. the optimization problem the agent seeks to solve is:

$$\max_{\pi \in \Pi} V^{\pi}(s)$$

An advantage of MDPs is that there exists a stationary and deterministic policy that simultaneously maximizes $V^{\pi}(s)$ for all $s \in \mathcal{S}.$. We first define the optimal value function and the optimal Q-factor as:

$$V^{*}(s) := \max_{\pi \in \Pi} V^{\pi}(s), \forall s \in \mathcal{S}$$
$$Q^{*}(s, a) := \max_{\pi \in \Pi} Q^{*}(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Here exists a stationary and deterministic policy π^{*} such that for all $s \in \mathcal{S}.$ and $a \in \mathcal{A}(s)$:

$$V^{\pi^{*}}(s) = V^{*}(s)$$
$$Q^{\pi^{*}}(s, a) = Q^{*}(s, a)$$

We refer to such a π^{*} as an optimal policy. The optimal policy can be given as :

$$\pi^{*}(s) \in \arg\max_{a \in \mathcal{A}(s)} Q^{*}(s, a)$$

Following equations are Bellman Optimality Equations[14] which hold for the optimal value function and Q-factor:

$$V^{*}(s) = \max_{a \in \mathcal{A}(s)} \mathbb{E}_{s_1 \sim \mathbb{P}(|s,a)}\left(r(s, a, s_1) + \gamma V^{*}(s_1)\right)$$

$$Q^{*}(s, a) = \mathbb{E}_{s_1 \sim \mathbb{P}(|s,a)}\left(r(s, a, s_1) + \gamma \max_{a' \in \mathcal{A}(s_1)} Q^{*}\left(s_1, a'\right)\right)$$

The Bellman Equations is the main idea in value-based methods.

3.6.2 Modeling DQN in Othello

A Deep Q-Network, also known as a Deep Q-Learning network are the neural networks that's can approximate the state values of the function in the Q-Learning framework. As an example, in Othello Game, it accepts as input numerous frames from the game and outputs state values for each action as an output. We first model the othello game as an infinite horizon problem. In this problem, the MDP $M(\mathcal{S}, \mathcal{A}, \mathbb{P}, r, \gamma, \mu)$ is specified as:

- **state space** $\mathcal{S}$ = {(board in the game, player that is that making a move)}
- **action space** $\mathcal{A}$: $\mathcal{A}(s)$ is valid moves can be made for the state $s \in \mathcal{S}$.
- In the othello game, the transition is deterministic. Let's us define the **transition function** as f, namely $s_{t+1} = f(s_t, a_t)$.
- **reward function** r = 1 if the current assumption of move wins the game at the end.
  r = -1 if the current assumption of move loses the game at the end, otherwise r = 0
- **discounted factor** γ = 1, the assumption of value for moves will not be discounted, the current optimal value is $V^*(s)$ for the current state ***s,*** the expected reward is the win ratio.
- **transition probability Matrix** $\mathbb{P}$ will be the state transition matrix followed by the principle of Othello rulesets, where a transition of state has occurred

The core idea of DQN theory is to obtain Q-value(Q-function) $Q(s, a, \theta)$ in a form of neural networks, where "Q" represents the function that the algorithm computes - the expected rewards for an action taken in a given state, and θ is a variable in the neural networks. We can obtain the parameter θ^* by interacting with the environment and a Gradient descent algorithm, so that the Q-value in the neural network satisfies the Bellman Optimality Equations:

$$Q\left(s,a,\theta^*\right)=\mathbb{E}\left(r\left(s,a,s'\right)+\max_{a'\in\mathcal{A}(s')}\ Q\left(s',a',\theta^*\right)\right)$$

There are three challenges encountered here: 1) How to interact with the environment 2) How to implement the gradient descent algorithm so that the Q-function can approach in the right direction 3) How to collect training data of the Q-function

1) Interact with the environment

The purpose of interaction with the environment is to provide samples for the training, which generally involves a dilemma of **Exploration** and **Exploitation**. This dilemma is that under the online setting, the Q-function must interact with the environment to obtain the information from the environment before it can be updated continuously. In order to make the Q-function have better astringency, more states need to be explored, and in order to explore more states, the suboptimal or even poor actions also need to be explored, which results in degrading the performance of the interaction process.

On the other hand, we should not only explore but also obtain higher gaining as much as possible, so Exploitation comes in here. Exploration describes as more of a long-term benefit while Exploitation exploits the agent's current evaluated value and picks the greedy approach to get the most reward. Therefore, how to balance Exploration and Exploitation is a key criterion in reinforcement learning. **Epsilon(ε) - greedy - policy**[11] can be applied here, where **ε** is the possibility for choosing a random action to explore, while there are 1 - **ε** possibility choosing the current Q-function to pick an optimal action.

2) Value Iteration Algorithm of Q-function:

After getting the samples from the interaction with the environment, it's time to think about how to iteratively solve the optimal Q-function. If we define **TD target *y*** as:

$$y := r\left(s, a, s'\right) + \max_{a' \in \mathcal{A}(s')} Q\left(s', a', \theta\right)$$

So the iteration of function is to constantly approach **TD target**. Assuming there is a batch dataset $D = \{s_i, a_i, s_i^{\backprime}, r_i\}_{i=1}^{n}$, the target function will be:

$$\mathcal{L}\left(\theta\right) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - Q\left(s_i, a_i, \theta\right)\right)^2$$

Therefore, the parameters are then updated as gradient descent:

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla \mathcal{L}\left(\theta_t\right)$$

The main drawback of the Value Iteration Algorithm is that if the state space $\mathcal{S}$ or the action space $\mathcal{A}$ are too large, then each iteration may be time-consuming, or impossible. Thus the main idea of the DQN algorithm is to approximate the Value Iteration by function approximation.

For the large state space $\mathcal{S}$, we consider introducing parametric families $\{Q\left(\cdot, \cdot, \theta\right); \theta \in \mathbb{R}^d\}$ to approximate Q^*, where W is the weights need to be trained. If the parametric families are neural nets, then we call it Deep Q learning.

In each iteration, we want to achieve this goal:

$$Q\left(s, a, \theta_{k+1}\right) \leftarrow \mathbb{E}_{s' \sim \mathbb{P}(\cdot|s,a)} \left(r\left(s, a, s'\right) + \max_{a' \in \mathcal{A}(s')} Q\left(s', a', \theta_k\right)\right), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$$

since the state space $\mathcal{S}$ or the action space $\mathcal{A}$ are too large, we only can use finite transition samples to $\left\{\left(s_0^{(i)}, a_0^{(i)}, r^{(i)}, s_1^{(i)}\right)\right\}_{i=1}^{n}$ to fit $Q(\cdot, \cdot, \theta_{k+1})$ approximately:

$$loss\left(\theta\right)=\frac{1}{n}\sum_{i=1}^{n}\left(Q\left(s_0^{(i)},a_0^{(i)},\theta\right)-\left(r\left(s_0^{(i)},a_0^{(i)},s_1^{(i)}\right)+\max_{a'\in\mathcal{A}(s')}Q\left(s_1^{(i)},a',\theta_k\right)\right)\right)^2$$

$$\theta_{k+1}\in\arg\min_{\theta\in\mathbb{R}^d}loss\left(\theta\right)$$

Sin the **argmax** can not be solved exactly, thus we can do a step gradient decent from W_k, namely:

$$\theta_{k+1}=\theta_k-\alpha\nabla_\theta loss$$

3) Experience Replay

In online setting, the sample can only be collected by interaction with the system, thus we introduce the **Experience Replay** to create **iid** samples for the above gradient descent. In reinforcement learning, a quadruples $(s,a,s^{\backprime},r)$ are called **Experience.** We collect the transitions $\left\{\left(s_t,a_t,r_t,s_{t+1}\right)\right\}_{t=1}^{T}$ to the **buffer D**. Then we sample a batch $\left\{\left(s_0^{(i)},a_0^{(i)},r^{(i)},s_1^{(i)}\right)\right\}_{i=1}^{n}\sim D$, and perform the gradient descent on the sample batch.

The training of neural networks requires independent identically distributed (iid) data, and the interaction with the environment is a trajectory. Therefore, we cannot directly use the samples on a trajectory to train the data, because the experience on the same trajectory has Markov property. Here is where Experience Replay comes in.

We store the experience of each implementation into a buffer **D**, and then extract the batch from this buffer **D** for training every time we need to train. In this way, as long as the capacity of **D** is large enough, our batch data can be regarded as data similar to **iid**.

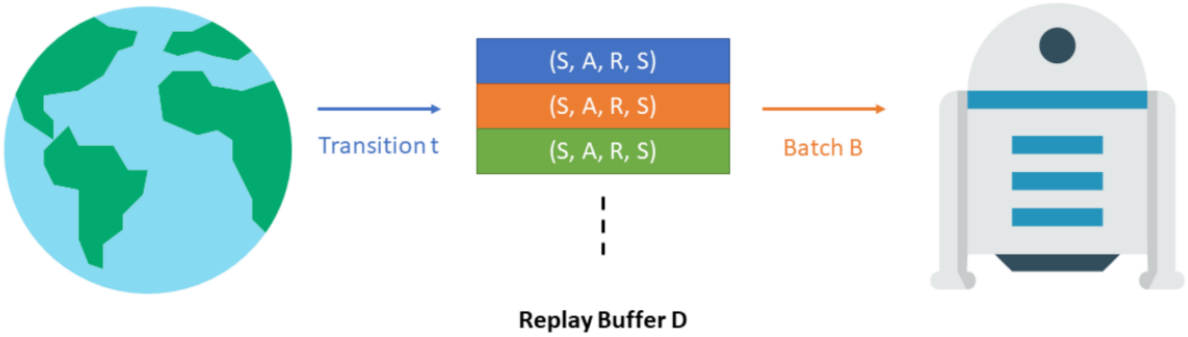


Figure.9 Explanation of Experience Replay

Logic of algorithm for DQN Othello AI agent:

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
For eposide $= 1,...,M$ do:
  For t=1,...,T do:
    With probability ε select a random action a_t
    otherwise select $a_t \sim \arg\max_a Q(s_t, a, \theta)$
    Execute action at in emulator and observe reward r_t and board s_{t+1}
    store transition (s_t, a_t, r_t, s_{t+1}) in D.
  End For
  sample a random minibatch of transitions $\left\{\left(s^{(i)}, a^{(i)}, r^{(i)}, \tilde{s}^{(i)}\right)\right\}_{i=1}^{n} \sim D.$
  set $y_i = \begin{cases} 1 & \tilde{s}^{(i)} = t_{won} \\ -1 & \tilde{s}^{(i)} = t_{loss} \\ \max_{a \in \mathcal{A}\left(\tilde{s}^{(i)}\right)} Q\left(\tilde{s}^{(i)}, a, \theta\right) & O/W \end{cases}$
  perform a step of gradient descent on $\frac{1}{n}\sum_{i=1}^{n}\left(Q\left(s^{(i)}, a^{(i)}, \theta\right) - y_i\right)^2$ with respect to thenetwork parameters θ
End For

When used in combination with Experience Replay, it is most often employed for storing episode stages in the memory for the off policies learning, in that's samples are taken randomly from the replay memory. The Q-Networks are also often tuned towards a frozen target network that's are occasionally updated with the newest weights every k steps, as opposed to a dynamic target network. The latters are improves the stabilities of training by

avoiding short terms oscillation caused by the moving targets from the occurring. The former deals with the autocorrelation that would develop as a result of on-line learning, while the latter makes the issue more resemble a supervised learning problem due to the presence of a replay memory.

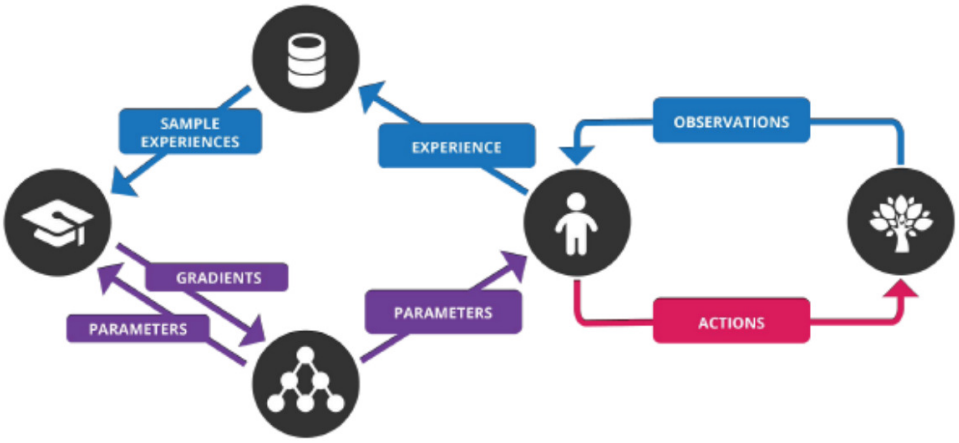


Figure.10 Explanation about algorithm logic of DQN

Neural Networks and Reinforcement Learning have never before been integrated in a scalable method, according to the Google Deep mind Team[12]. DQN has many essential elements that's are the for the first time, allowed the powers of the neural network to be merged with RL in the scalable manners (e.g., a game score). DQN's were trained from the samples selected from the pool of stored episodes and the process are the physically realized in the brain structure are called the hippocampus via the ultra fast re_activation of the recent events during the rest periods by a neurobiologically inspired method which is experience replay. DQN's success may be attributed to its use of experience replay.

Section - 4 Implementation and development

This section will explain the implementation of the approaches and design included in the last section to the code level. However, it is impractical to include pages of code in the report, so only the critical parts of code will be selected and explained. The full version of the code is attached with the report.

4.1 User Interface Development

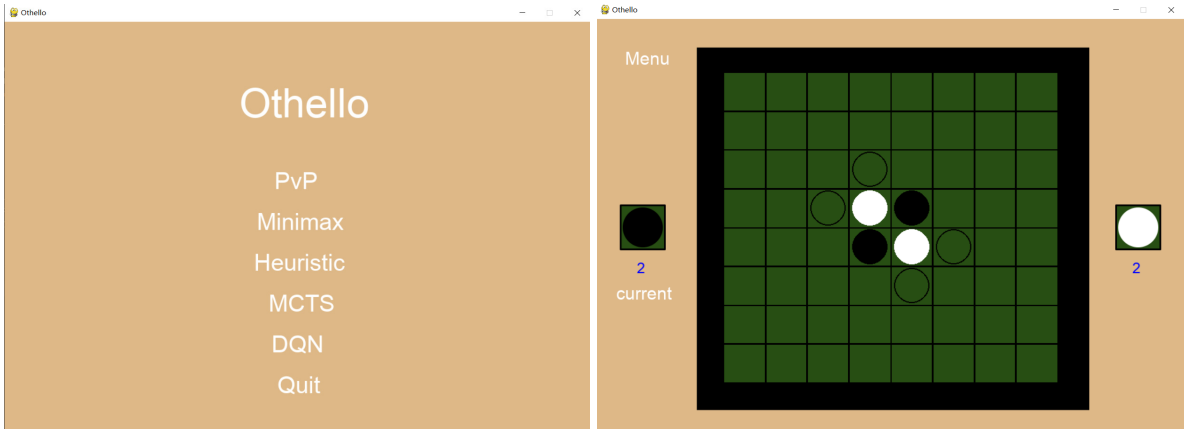


Figure.11 Final visual of the UI

As mentioned in section 3.2, the UI consists of a Menu page containing buttons that link to other pages that you can play with the AI agents, and a page for Player vs Player. The game page consists of the game board and player icons that are drawn with Google Drawing. The font used in the program is Arial MT The numbers under the icons represent the number of current disks, and “current” represents the current player that making the move.

Pygame is convenient for making 2D games with low graphical requirements. It helps users to draw images and play sounds.

In Pygames images are dynamically generated while a game is running, the program constant checks player input to update what frame is being drawn. This is called an event loop, with display logic:

1. Initialise the game window

```
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

2. Receiving player input

```
while running:
      for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                  mouse_pos = event.pos
```

the clicked position has been detected as x-y coordinates and assigned to mouse_pos

3. Use the information to place elements on the screen

use function `screen.blit()` to display text and images

```
for i in range(len(board.board)):
   for j in range(len(board.board[0])):
       if board.board[i][j] == 2:
           pygame.draw.circle(screen, BLACK, (x, y), 30)
       elif board.board[i][j] == 1:
           pygame.draw.circle(screen, WHITE, (x, y), 30)
```

draw disks on the board

4. 1 image of frames is created, while the window displays 30 - 60 times the image per second, and this is known as FPS(frames per second) in the video games

```
pygame.display.update()
```

where FPS has been set to 30 in the game

5. Return to step 2 to continue the event loop

4.2 Game Logic Development

The functions of implementing the game rules were initially put together within the same python file with UI, which become an obstacle when developing the DQN model, thus I

rewrite a new version of UI to clarify the functions. The overall process of implementation of the game rules is less complex than expected due to the flexibility of Python. The core functions of game rules consist of:

- Initialise the game board, four initial disks are placed in the centre of the board
- Find the valid moves of the current player - the move that at least flips one opponent's disk.

```
directions = np.array([[0, -1], [-1, -1], [-1, 0], [-1, 1],
                       [0, 1], [1, 1], [1, 0], [1, -1]])
      for direct in directions:
         stone_count = 0
         pwd = np.array([x, y])
         pwd += direct
         temp_flip = []
         while tmpboard[pwd[1]][pwd[0]] == 3 - player_no:
             temp_flip.append(pwd.copy())
             pwd += direct
             stone_count += 1
         if stone_count > 0 and tmpboard[pwd[1]][pwd[0]] == player_no:
             return True
      return False
```

use for loop to travaere eight direction of placed disk, append detected move to a list

- Once legal moves are made, flip the horizontal, vertical and diagonal adjacent opponent's disks if there is another self side disk on the line
- Swap player once a move is made or there is no valid move for the current player
- Count the current disk of each player
- Detect if the game is over or not

The main challenge I encountered at this stage was to find the legal move and flip the disks that need to be flipped. Initially, I used while loops separately to check the board horizontally, vertically, diagonally from top-left to bottom-right and from top-right to bottom-left, which is not efficient in terms of computational efficacy. And this would also affect the calculation time of the AI algorithms, especially when a large amount of data need to be calculated, so the final version has used fewer layers of loops to improve the calculation efficiency.

4.3 Implementing Heuristic Search Agents

4.3.1 Heuristic agent

The process of building the first simple heuristics AI agent is straightforward. Firstly assigning elements in a two-dimensional array board with a fixed score, the moves with the highest score will be considered first:

```
def get_max_score_actions(board,valid_actions,board_score):
      max_score_actions = []
      max_score = -1
      for action in valid_actions:
          row,col = board.action2square(action)
          score = board_score[row][col]
          if score > max_score:
              max_score = score
              max_score_actions = [action]
          elif score == max_score:
              max_score_actions.append(action)
      return max_score_actions,max_score
```

If the scores are similar for moves, the ones with fewer flips will be chosen. Also, these functions are created for each strategy have been listed in section 3.3.2, such as:

Winning move detection - detect move that eliminates all the opponent's, initially checks if any valid moves can achieve this condition.

Block move detection - detects the move that makes the opponent has no valid move in the next turn. The functions perform as checking if any valid move causes the opponent skips his turn.

Cutting move detection - detect the move that cuts through the opponent's disks, and cannot be re-flipped once the move is executed.

Side neighbour detection - avoid moves on the sides that have an adjacent neighbour of the opponent's disk

4.3.2 Minimax agent

Here we extend our heuristics agent by applying the Minimax concepts. Another heuristic evaluation function ***utility(state, player)*** has been used here to return the scores of valid moves, for both player 1 and player 2 in the given state. The aim of the Minimax agent is to search for the score that maximizes its own score and minimizes the opponent's score. The function max_value() has been designed to achieve this (accompany by an α-β pruning):

```
def max_value(state, depth, alpha, beta, player, original_player):
      # find valid move
      valid_points = find_valid_points(state, player, False)
      if len(valid_points) == 0 or depth == 0:
            # evaluate score for valid moves
            return utility(state, original_player), None
      v = -float("inf")
      target_point = None
      for x, y in valid_points:
            new_state = copy.deepcopy(state)
            new_state[x][y] = player
            reverse(new_state, x, y, player)
            cur_v, _ = min_value(new_state, depth-1, alpha, beta,
3-player, original_player)
            if cur_v > v:
                  v = cur_v
                  target_point = (x, y)
            if v >= beta:
                  return v, target_point
            alpha = max(alpha, v)
      return v, target_point

--------------------------------------------------------------------
--
def alpha_beta_search(state, depth, player, original_player):
       v, target_point = max_value(state, depth, -float("inf"),
float("inf"), player, original_player)
    return target_point
```

Conversely, a min_value() function is used to minimise the opponent's score, thus forming a loop to search through the nodes on the game tree.

The search depth has been set to 4, which performs a balance in efficiency and "intelligence" in the evaluation. A more detailed evaluation of the coefficient between the performance of Minimax agent and search depth will be evaluated in section 5.

4.3.3 MCTS agent

To implement MCTS firstly we need to define the tree nodes, while each node keeps track accompanied with a value Q, prior probability P, score v and how many times it has been visited which are corresponding to the parameters explained in section 3.5.1 (N(a), Q(a), $\pi(a|s;\theta)$).

```
    def __init__(self, parent, prior_p):
        self._parent : Board = parent # parent node
        self._children : Dict[int,TreeNode] = {}  # a map from
action to TreeNode
        self._n_visits = 0 # how many times it has been visited
        self._Q = 0 # average socre of nodes
        self._u = 0 # MCTS chooses node with highest Q + u
        self._P = prior_p # probability of node being selected
```

Objective of first selection and expansion phase is to select a child node to traverse along the search tree path that based on the quality score of node which obtained from early iterations.

```
  def select(self, c_puct):
     # select the child node with highest Q + u, returns a tuple of (action, next_node)
       return max(self._children.items(),
                  key=lambda act_node: act_node[1].get_value(c_puct))
```

MCTS continuously selects the child nodes of the highest Q + u until a leaf node has reached. `c_puct` is a constant that determines the level of exploration as a parameter to calculate u.

When a leaf node has reached and the game is not terminate at the state of this node, we need to expand the tree by creating new child `action_priors` which contains a list of tuples consists of actions and their prior probability according to the policy function:

```
def expand(self, action_priors):
        for action, prob in action_priors:
            if action not in self._children:
                self._children[action] = TreeNode(self,prob)
```

Update the value of the current node according to the value of the leaf evaluation. Also update the ancestors node with recursion. Then here we can calculate and return the Q+u value and move on to implement th main body of MCTS.

In another class MCTS, create an function _playout() to simulate playout from root to leaf, and propagating back the value get at the leaf.

```
def _playout(self, board : Board):
    node = self._root
    player = board.current_player
    while(1):
        if node.is_leaf():
            break
        action, node = node.select(self._c_puct)
        board.execute_move(action)
    action_probs, _ = self._policy(board)
    end, winner = board.is_end
    if not end:
        node.expand(action_probs)
        leaf_value = self.get_leaf_value(board,player)
    else:
        if winner == -1:  # tie
            leaf_value = 0.0
        else:
            leaf_value = (1.0 if winner == player else -1.0 )
    node.update_recursive(-leaf_value)
```

At this point the loop of 4 phases: Selection, Expansion, Simulation, Backpropagation have been implemented. The MCTS agent run all playouts sequentially and return the valid actions with the corresponding probabilities.

```
start_game(MCTSAgent(n_playout = 100),HumanAgent())
```

Number of playouts simulation is adjustable in the Menu class, the higher the number of playouts the decision is more optimal, however it take more time. On my laptop, 50 times is the limit to get the answer immediately(less than 3 seconds). While simulating the playout, user can check the interpreter for the process:

```
C:\Users\wenli\AppData\Local\Microsoft\WindowsApps\python3.9.exe C:/Users/wenli/Desktop/OthelloAIPlayers/Play_Othello.py
pygame 2.1.2 (SDL 2.0.18, Python 3.9.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
Monte Carlo Tree Searching: 100%|██████████| 100/100 [00:13<00:00,  7.43it/s]

Monte Carlo Tree Searching:  29%|███       | 29/100 [00:04<00:08,  7.97it/s]
```

4.6 Implementing the DQN Agent

In order to support the training of the neural networks, I used a (4 × 8 × 8) three-dimension tensor to represent the board and to feed into the neural net. For a state **s** of the board, we use following method to transform the board **s** into a (4 × 8 × 8) tensor x:

- x[0, :, :] is to indicate the disks of the player 1**,** i.e. if x[0, i, j] = 1 other wise x[0, i, j] = 1
- x[1, :, :] is to indicate the disks of the player 2**,** i.e. if x[1, i, j] = 1 other wise x[1, i, j] = 1
- x[2, :, :] is to indicate the last move of the both player. i.e, if the last hand is (i, j) , then x[2, i, j] = 1, otherwise x[2, i, j] = 0
- x[3, :, :] is to indicate which one is the current player. i.e, if the current player is 1, then x[3, i, j] = 1, otherwise x[3, i, j] = 0

To obtain the **x** that corresponds to the board state **s**, I created to_state() function in the class Board():

```
def state(self):
    n = self.size
    state = np.zeros((4,n,n),dtype=np.float32)
    state[0] = np.where(self.board==1, 1, 0)
    state[1] = np.where(self.board==2, 1, 0)
    #indicate the last move
    if self.last_action:
      last_action_square =
      self.action2square(self.last_action)
      state[2][last_action_square] = 1
    #indicate the colour to play
    if self.current_player == 1:
        state[3,:,:] = 1
    return state
```

This algorithm has the **x** that indicates state **s** as the input of the neural network. The output is the Q-value corresponding to 64 squares on the board that the player can make a move.

The network structure is divided into two main parts. The first part is the convolution neural network(CNN) of layer1-layer4, which is responsible for extracting the reflected features. The second part is the fully connected network of layer5-layer6, which is responsible for outputting the corresponding Q value The whole network structure and parameters are as follows, responsible for extracting the pattern detected by **x**. The second part is the layer 5-layer 6, the fully connected layers, which is responsible for outputting the corresponding Q value. The network structure and parameters are as follow:

layer	structure
1	Conv2d(input_channel = 4, output_channel = 32, kernel size = 3, padding =1) with Relu activation
2	Conv2d(input_channel = 32, output_channel = 96, kernel size = 3, padding =1) with Relu activation
3	Conv2d(input_channel =96, output_channel = 128, kernel size = 3, padding =1) with Relu activation
4	Conv2d(input_channel =128, output_channel = 4, kernel size = 3, padding =1) with Relu activation
5	Linear Transformation of shape (4 * 64, 256) with Relu activation
6	Linear Transformation of shape (256, 64) with Tanh activation

Figure.12 Layers of CNN

The convolutional neural network with an output of 4*64 neurons is fed directly into the dense layers, with the activation function of ReLu and Tanh. This output is an array of length 64. Where each element represents a position on the board.

The algorithm used Pyorch to train and construct the network. The implementation of the network structure of Q-fuction:

```
class Q_Network(torch.nn.Module):
    def __init__(self,board_size : int) -> None:
        super().__init__()
        self.n_square = int(board_size * board_size)
        self.conv = torch.nn.Sequential(
            torch.nn.Conv2d(4, 32, kernel_size=3, padding=1),
            torch.nn.ReLU(),
…….
Omitted the layers of CNN here for readability of report
…….
        )
        self.dense = torch.nn.Sequential(
            torch.nn.Linear(4 * self.n_square, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256,self.n_square),
            torch.nn.Tanh()
        )
        self.loss = torch.nn.MSELoss()
```

We have self.forward function here to obtain the Q-value after we have the input **x,** and self.loss as the MSE loss function:

```
    def forward(self,x) -> torch.tensor :
        conved = self.conv(x).reshape(-1,4 * self.n_square)
        predict = self.dense(conved)
        return predict
```

The execution process is step by step when the first state will be implemented and the output received then the next state will be implemented. After that, it can store the experiences of all the states and actions into the replay memory in the system.

When the all of experiences are stored in the replay memory, it can compute the all of predicted values which are collected as the output in the system by using the online network. After that for obtaining the weight, it can compute the loss activities between the predicted and target network quantities. When it done then we need to make a copy of the online network and store into the target network.

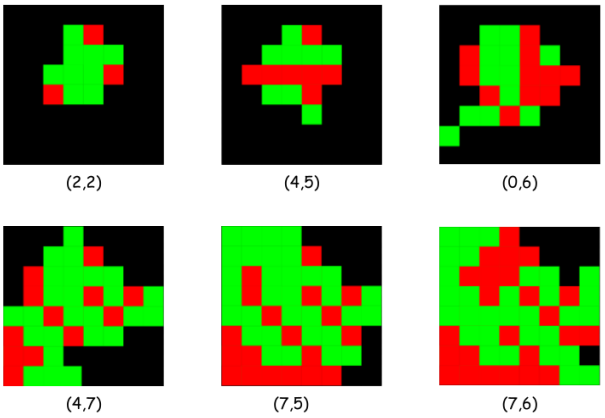


Figure.13 Example game boards denoting the board recognized by the CNN[13]

Red squares indicate the disks of the current player, Green inculcate the opponents' disks and the black space are the current empty squares. The (x, y) coordinates underneath show a predicted of move from CNN.

I created another class Memory to construct replay buffer D. Since each experience is made of a quadruple, and calculation of **TD target** involves a determination of terminal state, state, action, and reward have different data formats. Therefore the elements are stored separately. In __ init__, five variables are created:

```
def __init__(self):
    self._transitions  = None  #(batch,2,2,8,8)
    self._actions = None    #(batch,1)
    self._rewards = None    #(batch,1)
    self._terminals= None    #(batch,1)
    self.max_size = 100000
```

Where self.transitions is used to store the state in the experience, _actions , _rewards, _terminals are used for storing the corresponding action, reward and terminal state in bufferD. Also functions such as: {self.add() to add the experience, self.check_size() to check the size of D, and self.sample_batch() to extract batch from D} have been implemented.

To implement the interaction with the environment, I start with making an abstract class Agent, accompanied by a function self.select_action(), which for obtaining the action made by the agent during the interaction with the environment.

```
class Agent(ABC):
      @abstractmethod
      def select_action(self,board, action_space: List) -> int :
      pass
```

The implementation of DQN Agent is achieved by the class DQNAgent(). DQN Agent learning constantly during the interaction, so it needs to maintain a replay buffer D and a Q-value, along with interfaces for adding experience and experience replay. Therefore the parameters of self.__init__ of the agent have been set as:

```
class DQNAgent(Agent):
    def __init__(self,board_size : int) -> None:
        self._D = Memory()
        self.epsilon = 0.2
        self.mini_batch_size = 256
        self.Q = Q_Network(board_size = board_size).to(DEVICE)
          self.optimizer  =  torch.optim.Adam  (params           =
     self.Q.parameters(),  lr = DQN_lr_rate)
        self.alpha = 0.995
        self._train_mode = True
```

Where:

- self._D is the instance of replay buffer
- self.epsilon is the coefficient for exploration
- self.mini_batch_size is the size of each batch data during dqn training
- self.Q is the instance of Q-value
- self.optimizer is the optimizer of the training process, which I used the Adam optimization method
- self.alpha is the discount factor, which has been set as 0.995 instead of 1
- self._train_mode is used to judge whether it is in training mode. If not then theris is no need to calculate the gradient

During the training, every experience is to add to the replay buffer D, which is accomplished by a self.add_experience() function. And the experience replay for DQN agent implemented

by a self.experience_replay() function, in which the first half of the function compose of extracting the batch to construct TD Target with the second half to train the batch and accomplish gradient descent. Epsilon(ε) - greedy - policy has applied to interaction, thus the self.select_action() function has been implemented as:

```
  def select_action(self,board : Board, action_space: List):
        u = np.random.rand(1)[0]
        n_square = int(board.size * board.size)
        if self._train_mode and u < self.epsilon:
            return np.random.choice(action_space,1)[0]
        else:
            state = torch.unsqueeze(torch.from_numpy(board.state),dim = 0) #
(1,4,8,8)
            non_valid_action = np.delete(np.array([i for i in
range(n_square)]), np.array(action_space))
            with torch.no_grad():
                Q  : np.array = self.Q(state.to(DEVICE)).cpu().numpy()
            Q = Q.reshape(n_square)
            Q[non_valid_action] = - np.inf
            action = np.argmax(Q)
            return action
```

Where **u** is a local variable, used to determine whether random actions need to be taken.

To train the DQN agent, I used the class game to implement the logic of Othello, and a class OthelloGame to simulate the game between two agents, accompanied by the Train function:

```
def Train(agent : DQNAgent, board_size : int,Train_Epoch: int
=100000,Train_Episode : int = 20,Eval_Episode : int = 10):
```

Where:

- agent is an instance of class DQNAgent
- board_size is the board size for training, which is 8×8 for Othello
- Train_Epoch is the number of epochs used for training
- Train_Episode is the number of episodes per epoch interaction
- Eval_Episode is the number of episodes that evaluate the performance of the current Q-function after each epoch training phase

The overall logical architecture of Train function is a for loop that traverses all epochs:

```
for epoch in range(Train_Epoch):
    #Trainning
    ...
    #Evaluation
    …
```

In the training phase of each epoch, we traverse episodes for a specified number of times, add experience to the replay buffer, and call the function experience_replay to update the parameters.

In the evaluation phase of each epoch, we also traverse the episodes to see the performance of the green policy induced by the current Q-function against the random agent, and print the results.

Examples of the print output for each epoch in Pycharm :

Section - 5 Evaluating AI's Potential

This section evaluates the experiments performed to test the agents, and analysis the results found. All experiments are automatically run by the programs with identical settings. The choice of constant parameters will also be evaluated. The data collected has been summarised as tables or charts.

5.1 Test Environment

This is the details of the hardware and software that have been used for the evaluation:

Processor: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

RAM: 16.0 GB (15.4 GB available)

Graphics Card: NVidia GeForce RTX 2060

OS: Windows 10

Python version: 3.9.13 (pygame: 2.1.2, SDL: 2.0.18)

PyCharm 2021.2. , Excel, Venngage (for graph drawing),

Three extra agents have been implemented as benchmarks to test the smart agents.

1. A Random agent that makes random selections from the valid moves
2. A Greedy agent that always chooses the move that flips most disks for current turn
3. A Value Matrix agent that choose the move purely base on a value matrix assign to the board, the matrix has assigned value of:

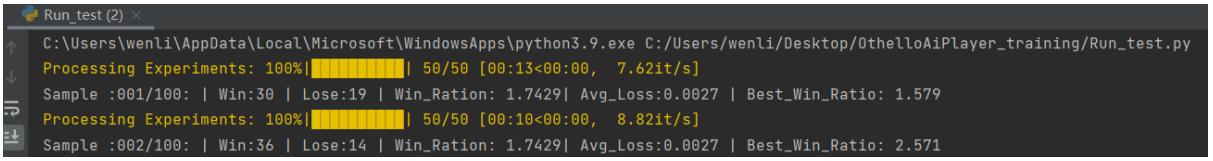
[8, 1, 7, 6, 6, 7, 1, 8],
[1, 0, 2, 3, 3, 2, 0, 1],
[7, 2, 5, 4, 4, 5, 2, 7],
[6, 3, 4, 0, 0, 4, 3, 6],
[6, 3, 4, 0, 0, 4, 3, 6],
[7, 2, 5, 4, 4, 5, 2, 7],
[1, 0, 2, 3, 3, 2, 0, 1],
[8, 1, 7, 6, 6, 7, 1, 8]

All experiments are run by Python, each experiment runs for 100 games, repeated for 20 times, thus perform a total of 2000 games between two players.

Since Othello is a two-player turn-based game, and always starts on the black side. The order of first hand and second hand may affect the winning rate, so each experiment will divide into two parts, 50 of black start and 50 of white start.

In general games, the player that makes the first move usually takes advantage, such as GO, Chess and Gomoku. However, there is a dispute on this opinion in Othello, though there is a convincing theory stating that the Whiteside(the player that secondly move) potentially has an advantage, because the last move in the game generally has a higher chance to flips more disks than the second last move.

The output of the experiments function looks like this in the Python interpreter (in Pycharm):



When all the results are obtained for one set of experiments, the sum and average(per set) of Win will also be calculated automatically and exported to Excel.

5.2 Evaluating Performance Against Test Agents

Results :

In 2000 games played	Heuristics vs Random						Heuristics vs Greedy					
	Heuritics Black			Heuritics White			Heuritics Black			Heuritics White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	887	106	7	906	85	9	762	231	7	704	287	9
Win ratio	8.368			10.659			3.299			2.453		

In 2000 games played	Heuristics vs Value Matrix						Minimax vs Random					
	Heuritics Black			Heuritics White			Minimax Black			Minimax White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	524	473	3	521	476	4	947	45	8	957	37	6
Win ratio	1.108			1.094			21.044			25.865		

In 2000 games played	Minimax vs Greedy						Minimax vs Value Matrix					
	Minimax Black			Minimax White			Minimax Black			Minimax White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	947	50	3	960	36	4	776	216	8	789	197	14
Win ratio	18.940			26.667			3.593			4.005		

In 2000 games played	MCTS vs Random						MCTS vs Greedy					
	MCTS Black			MCTS White			MCTS Black			MCTS White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	934	56	10	941	52	7	762	226	12	730	259	11
Win ratio	16.678			18.096			3.372			2.819		

In 2000 games played	MCTS vs Value Matrix						DQN vs Random					
	MCTS Black			MCTS White			DQN Black			DQN White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	661	336	3	682	318	0	754	240	6	771	217	12
Win ratio	1.967			2.145			3.129			3.553		

In 2000 games played	DQN vs Greedy						DQN vs Value Matrix					
	DQN Black			DQN White			DQN Black			DQN White		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Sum	643	352	5	590	401	9	279	715	6	287	711	2
Win ratio	1.827			1.471			0.390			0.404		

As expected, start the game as White Side(Second) does seem like have an advantage but not always. In Othello, even if a player flips more disks in one round of the game, it is easy to be flipped back in the following rounds, and this is also the most interesting concept of the game.

Second discovery is that, except for DQN, other players have a winning rate of approximately 90% against Random, which also meets expectations. The Random player does not use strategy to reason the moves, and the games it wins likely can be explained as pure luck.

Visualised results as a stacked bar chart:

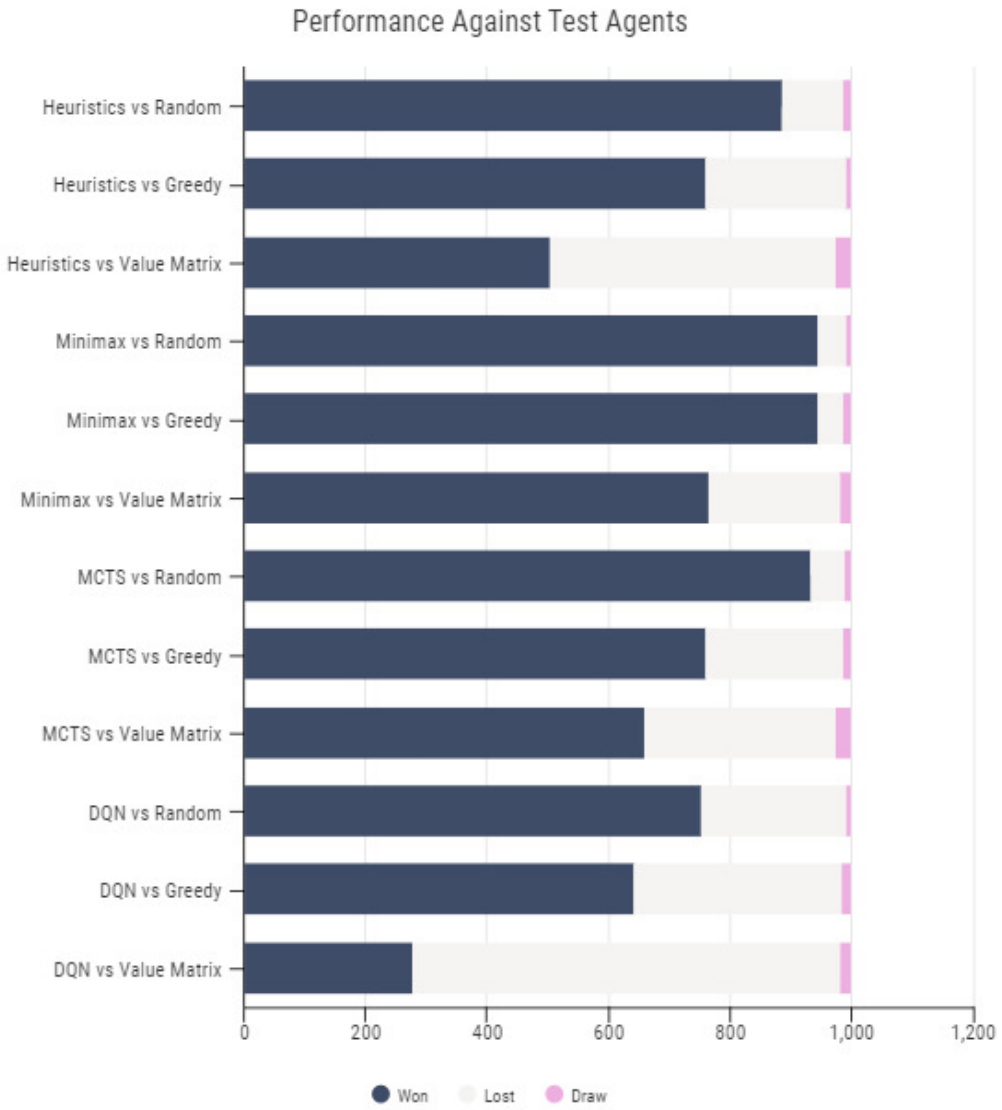


Figure.14 Stacked bar chart of AI performance

Up to this point, we can conclude the rank of overall performance in the experiment can be ordered as:

1. Minimax
2. MCTS
3. Heuristics
4. DQN

Overall, the agents all performed well against the test agents, which Minimax search performed best among the AI players, and unexpectedly, DQN has a relatively weak performance, especially when against the Value Matrix Agent.

A potential reason could be the batch size selected in the experience replay is too small, so it is hard for agents to get trained efficiently. The batch size used in the well-known DQN is very large, while in this project I used a relatively small batch size, due to the limitations of the hardware. Theoretically, the larger the batch size, the faster and more stable the training. The smaller the batch size, the longer the training time required. Also, I did not use the target network, which may lead to less stable training. The parameters used in the training may also need to be delicately adjusted.

5.3 Computational Efficacy Evaluation

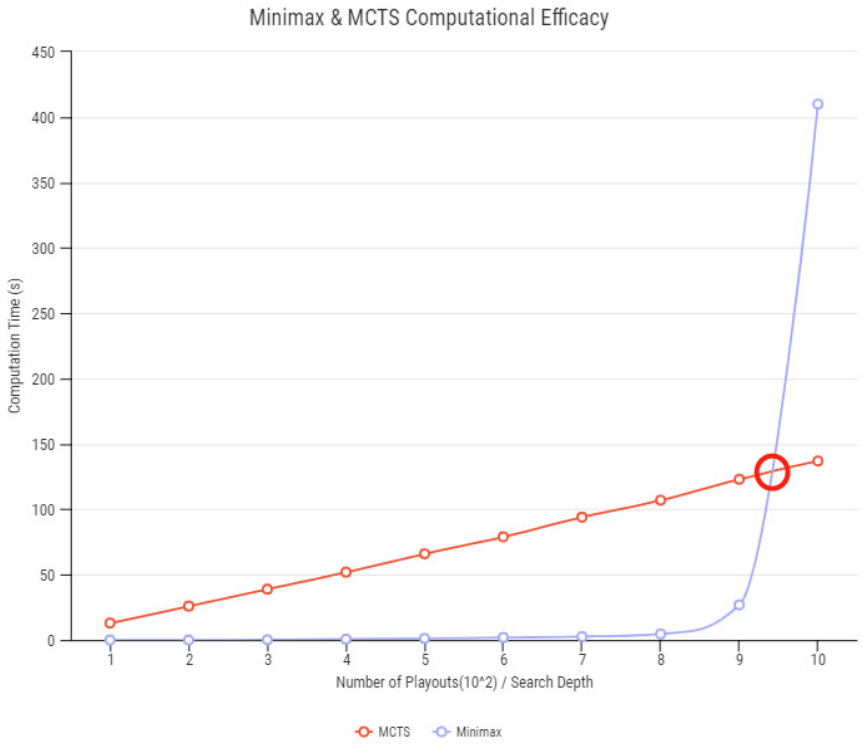


Figure.15 Computational Time of Minimax vs MCTS

This line chart shows the computational time of making the first move varies by the Search depth for Minimax and the number of playouts(10^2) for MCTS. The specific computational time is largely varied by the performance of the computer itself, thus this investigation focus on the comparison. The reason for doing the experiment on the first move is to ensure the board state is constant, different board states may differentiate valid moves, thus affecting computational time.

We can see that the MCTS curve has a constant gradient, while the computational time of Minimax has exploded during the search depth of 8 to 10 due to the increase in nodes searched.

During the experiment implementation in section 5.2, I found that some AI agents take much longer to calculate than others. Specifically, MCTS took 10 times longer time than others, because it performs a whole playout for each simulation. To make the comparison fairer, we can use the chart to find the approximate number of playouts and search depth with closet computational time, which is the interaction between the curves. This experiment composes of games that have approximately equal time allowed for each move:

In 100 games played	MCTS vs Minimax											
	MCTS Black			Minimax White			MCTS White			Minimax Black		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Results	16	33	1	33	16	1	20	28	2	28	20	2
Win ratio	0.485			2.062			0.714			1.400		

We can see that Minimax still outplayed MCTS in the experiment. This may be because of Minimax's feature of taking corners. MCTS simulates playouts without domain knowledge, while Minimax has a congenital advantage of having heuristics functions as a default set-up.

5.4 DQN Training Evaluation

In training the DQN agent, each epoch the agent is trained against a training opponent 100 times, then updates and saves the newest model. It would be interesting to investigate the performance of the agent varying the training opponents. In this experiment, 1000 epochs have trained for each model, corresponding to 10^6 training games. The chart shows the win ratio of training games per epoch with the random agent and greedy agent as opponents:

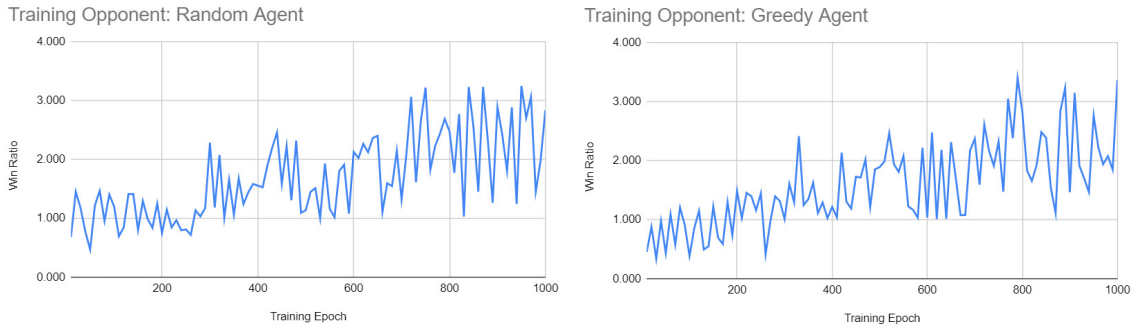


Figure.16 WinRation-Epoch with different training agent

In 100 games played	DQN(Random) vs DQN(Greedy)											
	D.R. Black			D.G. White			D.R. White			D.G. Black		
	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw	Won	Lost	Draw
Results	23	27	0	27	23	0	26	24	0	24	26	0
Win ratio	0.852			1.174			1.083			0.923		

The range of win ratio showing in both charts form an uptrend, while the one against greedy agent start will a lower initial range. The trained AI then played against each other, but there is no obvious variance in the win rate. Theoretically varying the opponents should have an influence on the learning agents because it affects the exploration and exploitation. It took 24 hours to complete 1000 epochs on my laptop. Perhaps increasing more training epochs would enhance the comparison.

5.5 Additional Evaluation Based on Investigation

Despite quantitative research has been implemented in previous sections, I have also implemented qualitative research of analysing the play style of each agent by slowly displaying the game between AI on UI.

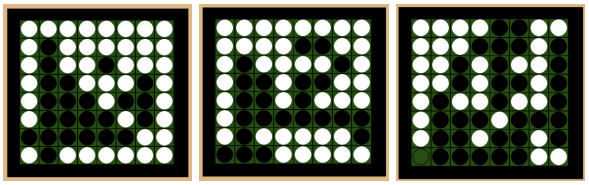


Figure.17 Examples of end game states of Minimax vs DQN

A game of Othello compose of 3 phases: beginning, mid-game and end-game. Using different strategies to cope with each phase is important. In a game between AI and AI, both players always make the optimal move base on their own logic. From qualitative investigation I observed that the determinant pattern of Minimax outplaying other AI agents is the corner oriented strategy. As we can see in Figure.15, Minimax is the White disks, it took almost all the corners, which would provide stable advantages in the middle and end game. This is a chart showing the win ratio against the random agent when games begin from a random state:

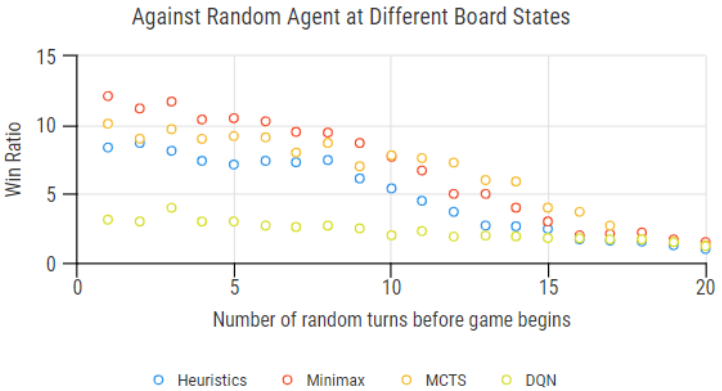


Figure.18 Add random turns before AI decisions

The win ratio of the algorithms dropped faster when taking over the game after the tenth round, in which the random agent started to reach the corners. The first few rounds seem to have little influence on the winning ratio.

Section - 6 Potential Future Improvement

Evaluated results and the problems encountered in the implementation section inspired me to some potential future improvements that can be accomplished to improve and refine the current system.

Firstly I am expecting to add more features to the heuristics evaluation function, integrating with Minimax and MCTs. Although all the AI players have been successfully implemented, they did not play a complementary role. Pattern detection and stable disk detection would be examples of new features, which recognized patterns on the board, such as by forming a diagonal line is less stable than a square. Also, it is also going to be interesting to investigate MCTS with more playout simulations, e.g. 50,000 or 100,000. This is unrealistic to be implemented on my personal laptop, probably could be achieved by running the algorithm parallelly on multiple computers.

In terms of GUI design, possible improvements include:

- add animation to the flips, so the users would better understand the transition between states, also more entertaining
- used images for boards, disks and background, so the graphics would be better
- allow other types of input such as keyboard
- redo function for user to recall a move
- a Move history that contains the moves made for the current game, or for the moves made when training the agent
- to improve the method of implementing the game logic, thus better computational efficacy for the AI agents

Note these ideas are for the aim of improving the user experience of UI, which means some of them are unpractical for improve the AI system.

Improvement for DQN involves trying out other strategies to find the balance between exploration and exploitation. Epsilon greedy policy has been used this time, which is a policy that does not always select the action with the highest Q-value. It is interesting to create a specialised policy to make a comparison. Additionally, it is interesting to investigate the performance of the DQN agent against human players in the internet gaming zone. Future work also includes investigating the influence of more differential agents as training opponents.

From the evaluation in the last section, we found that the performance of tree search agents is sensitive to its heuristic function, thus improving the heuristics function is another improvement that could be done for future work. The heuristics agent created in the project is based on a fixed valuation table, while the valuation on the board can be changed along with the change in the game state. Four additional strategy heuristics were used in the project experimentally, and advanced heuristics such as pattern detection can be applied in future work. However, it is important to note that each new feature mentioned above would potentially increase the computational burden and may not help the overall performance. It is also a meaningful attempt to apply this research to a broader level, such as in operations research and management science.

Section - 7 Conclusion

The aim of the project is to evaluate possible approaches to creating AI capable of playing the board game Othello, using methods such as Monte Carlo Tree Searching, Minimax Searching and Reinforcement Learning. To accomplish the aim, this project also involves creating a GUI accompany by the implementation of the game rules. The project has been successfully completed since the aims have been fulfilled.

The created fully-featured Othello game based on pygame module of Python performed well on HCI and implementation of the game logic, which allows AI agents to analyse the state of the game board. The GUI is designed to allow users to view the game state and play against a human player or AI agents.

The AI system implemented includes search algorithms using heuristics evaluation to make move decisions by predicting future states and DQN models that gain experience from enormous training games.

Experiments have covered the evaluation of the AI agents against the test agents, and the results met most expectations but also identified unseen outcomes. Investigation and evaluation examined the advantages and limitations of the tree search-based algorithms and the machine learning algorithm.

Overall, the main objectives of exploring AI algorithms that are able to choose sensible moves in the game have been met; the Minimax agent and MCTS agent are capable of defeating the test agents with a high level of performance constantly. The DQN show its potential to extract and incorporate field-specific knowledge based on training examples.

Personal Reflection

Designing this project gives me valuable experience which I believe would be helpful for my future work. Following the footstep of previous researchers, I enhanced my understanding of the topics of combing adversarial AI models with board games as well as the practical application of machine learning.

The report used a serif font Times New Roman with a font size of 12, and a line spacing of 2, which follows the standard of MLA style guidelines. The limitation of my English writing ability is one of the major obstacles to writing the report, while previously the longest scientific report has a maximum word count of 3,000. Since many learning materials I used to study this project were in Chinese, it was a challenge to find the correct corresponding terminology to explain the theory.

The process for the first two months was pretty smooth, I followed the time schedule planned in the Initial plan, including the research of the background, implementation of UI and game logic development. I re-edited the titles and specifies a specific plan for the machine learning part, along with deepening my understanding of this topic.

However, I encountered various difficulties when processing the test and evaluation phase. To train the DQN agent, I needed a computer to run the training model continuously for days or weeks to accumulate experience by playing against AI opponents. I attempted to run it on the desktop computer on the campus and at the library but did not succeed due to various technical issues. Therefore I have to run it on my own laptop and endure the noise generated by the fan or radiator inside the laptop when the program is running for weeks.

When it comes to the last few weeks before the submission deadline of the report, my laptop got malfunctioned. It would always make loud noises even it was not running in high demand. Said the man at the computer reparation shop that this might be some critical issues regarding GPU, it will get worsen if I keep using it. I tried to clean the fan and radiator, but it didn't work. I did not have time to send it to maintenance because the deadline is approaching. It was also a disaster that I have been sick for a week, while there were deadlines for other modules within that time period.

To get back to the point, I received suggestions for applying the AIs on the online platforms to play against human players, in order to test the ability of the AI agents, which I think is a cool idea. However, I did not find a suitable platform that allow me to accomplish this, one of the critical reasons for this is that Othello is not as popular as Chess or other board games. The platforms either be too official that I cannot get access to the interface or rarely used by people.

Anyway, I do satisfy with the outcome of this project, four playable agents have been implemented and evaluated. Initially, MCTS was not planned to be implemented, but I was able to accomplish it. I believe that the program I have implemented could be used as a tool to help the Othello beginner to practice their skills against AI agents with different difficulties. I honestly learned a lot in this project and also believe that the experience and the skills earned will be useful for my future career.

Glossary

Othello terms:

- **Disk**– Pieces used by players to play Othello. It can be of two colours: white or black
- **Board** – 8x8 totally 64 square for putting the disks
- **Flipping** – The process of converting a white counter into a black counter, or vice versa.
- **Stable** – The disk is stable if it cannot be re-flipped in any way in the current game.
- **Game State** – The situation of disks on the current game board.
- **Degree of Divergence** - the number of empty squares around a disk
- **Winning Move** - a move that wins the game
- **Blocking Move** - move that makes the opponent has no valid move in the next turn
- **Cutting move** - move that cuts through the opponent's disks, and cannot be re-flipped once the move is executed
- **Side neighbour** - moves on the sides that have an adjacent neighbour of the opponent's disk

AI terms:

- **Minimax**– a backtracking algorithm that is very commonly used in two-player turn-based board games. It uses recursion to search through the tree nodes and makes decisions based on heuristics evaluation to maximise the score of moves, assuming the opponent also makes optimal moves.
- **Heuritics** – a static evaluation function has been frequently used in traversing the search space based on given conditions in a given environment.

- **alpha-beta pruning** - to increase the search depth while reducing the search time for a more optimal solution within a reasonable runtime; to cut off unnecessary leaves or branches given they can in no way affect the decision of the algorithm.

- **Monte Carlo Tree Search** - another heuristic search algorithm used in Othello AI for decision processes to solve the game tree. Unlike Minimax, MCTS only analysis the most promising moves. It searches the game tree by combining random sampling of the search space.

- **Reinforcement learning** - a field in machine learning that allows AI agents to become capable of learning in an environment for the sake of maximizing the notion of cumulative reward.

- **Stationary Policy** - The policy does not change over time and only depends on the current state.

- **Non-stationary Policy** - The policy changes over time or depends on not only the current state but also the history

- **Deep Q-Network** - also known as a Deep Q-Learning network are the neural networks that's can approximate the state values of the function in the Q-Learning framework.

- **Epsilon(ε) greedy policy -** **ε** is the possibility for choosing a random action to explore, while there are 1 - **ε** possibility choosing the current Q-function to pick an optimal action.

Table of abbreviations

Abbreviation	Long Name
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network
GUI	Graphical User Interface
AI	Artificial Intelligence
DQN	Deep Q-learning Networks
MC	Monte Carlo
MCTS	Monte Carlo Tree Search
UI	User Interface
DoD	Degree of Divergence
MDP	Markov Decision Process

Appendix - A Ted Landau's 21 Key Strategies and Tactics [18]

1. Get More Stable Discs, Not Just More Discs – "…a player will take the move that flips the maximum number of discs. This has been clearly shown to be an inferior strategy." "The point therefore is not simply to acquire discs, but acquire discs that cannot be flipped…"

2. Not All Squares Were Created Equal – "A crucial idea that beginning players inevitably realise is the importance of the 4 corner squares on the board. A corner is important because it can never be flipped. That is, it is a stable disc." "In summary, the incorrect strategies place unwarranted value on flipping large number of discs even though they are not stable."

3. Control of the Game: Mobility Optimization and Dynamic Square Evaluation – "Your hope is to get your opponent to make a poor move that will allow you to win the game…. Your goal is to force him to make a poor move."

4. Good Moves and Bad Moves: Gaining Control – "… a good move can typically be defined in terms of the principles of gaining and maintaining control (with the ultimate aim of acquiring stable discs)." "… most expert players agree that the easiest way to get control of the game is by maintaining fewer discs that your opponent… This is referred to as evaporation strategy."

5. Good Moves and Bad Moves: Planning Ahead –"[Success] depends not merely on what is best for the current position, but what is likely to be best based on what the board will be like 2, 3, 4 or even more moves later."

6. Isolated C-square Traps – "… in this section we demonstrate a "trick" or "trap" that will force your opponent to concede a corner, no matter how many moves he might have available at the time."

7. Unbalanced Edges – "An unbalanced edge is defined as an edge occupied by five adjacent discs of the same colour immediately adjacent to a vacant corner." "Knowing whether or not to take an unbalanced edge, or to attack a currently existing one, are among the more difficult decisions in Othello."

8. Controlling the Main Diagonal: Stoner Traps and More – "Main diagonals can be so powerful, that a good player is constantly on guard for possibilities of gaining or losing control of them…"

9. Gaining and Losing Tempo – "A player is said to gain tempo when he achieves an advantage of timing by deriving one more viable move than his opponent from play within a limited area of the board and thereby forcing the opponent to initiate play elsewhere."

10. Odd and Even Regions of the Board – "… as the game develops, the board is frequently subdivided in separate regions of vacant squares. In this section we discuss a particularly noteworthy aspect of these regions… whether there are an odd or even number of vacant squares in the region." "… it is typically disadvantageous to move into an even-numbered region…"

11. Blocking Techniques: Access and Poisoned Moves – "Blocking techniques refer to ways to prevent your opponent from taking an otherwise good movie." "A poison move is [a] type of 73 blocking moves… you do not legally prevent the opponent from moving to a square; you simply make it undesirable."

12. The Semi-Forcing Move –"A semi-forced move is a move which is force not by the rules of the games, but rather by tactical considerations."

13. Mobility Reconsidered – "In some situations, there may be several safe moves that all flip about the same number of discs… How do you decide between them?"

14. The Opening: The Three Basic Variations – "… generalizations of strategy have such limited applicability in the opening, as to be practically useless."

15. The Opening: Quiet Moves –"Players frequently find themselves in the position that anything they do only make their positions worse. Their ideal move would be to pass, but they can't. So they attempt to "disturb" the board as little as possible… these types of moves are called quiet moves."

16. Midgame Strategy: Patterns –"In many cases, a global view of the midgame can assist in your decision making. This can be achieved by viewing the board position as one of several possible patterns."

17. The Endgame: The Final Count – "… clearly with only a few moves left, now is the time to go for broke and flip the most amount of discs each turn right? Well, it's almost right. The idea is not necessarily to flip the most amount of discs on each turn, but to flip the most amount of discs relative to your opponent."

18. The Endgame: Counting on More than Just Counting – "Probably the best overall advice for endgame play is "expect the unexpected" … Players need to be on guard for the hidden dangers and/or opportunities that may exist in a given position."

19. Edge Play: General Concepts and Initiating Edges –"As the end of the opening phase of Othello approaches… "… how does a player decide which edge square move is the best to initiate on?"

20. Edge Play: Developing and Resolving Edges – "Is it a good idea to resolve an edge as soon as you can, or should you leave it unresolved for a while, or even let your opponent resolve it?"

21. Decisive Moves and Prioritising of Moves – "… at some point, a move is made after which, barring any major blunders, the outcome seems decided. This is called the decisive move." "Decisive moves are really a special case of a still more general concept: how to read the position and prioritise the possible moves."

Appendix - B Initial Work Plan

END Date 13th May Tasks MUST (milestones)	February 14th	February 21st	February 28th	March 7th	March 14th	March 21st	March 28th	April 4th	April 11th	April 18th	April 25th	May 2nd	May 9th	May 13th
Weekly meeting														
Complete project with functional code														
Research on concepts of Othello and Othello AI														
Design the game UI														
Investigate existing algorithms and design the AI														
Programming														
Neural network development														
Testing and experimentation														
Run tests on system and debug														
Evaluation and write the Final report														
Review meeting														
Tasks COULD														
Not yet come up with one														

References

1. Slotnik, Daniel E. “Goro Hasegawa, Creator of Othello Board Game, Dies at 83 (Published 2016).” *The New York Times*, 24 June 2016, https://www.nytimes.com/2016/06/26/world/asia/goro-hasegawa-creator-of-othello-board-game-dies-at-83.html

2. *Othello learning materials - Baidu Wenku* https://wenku.baidu.com/view/5d741346551252d380eb6294dd88d0d233d43c9b.html

3. Computer Othello - Wikipedia. https://en.wikipedia.org/wiki/Computer_Othello

4. Cherry, Kevin Anthony. “An intelligent Othello player combining machine learning and game-specific heuristics.” *LSU Digital Commons*, https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=1766&context=gradschool_theses

5. Minimax - Wikipedia. https://en.wikipedia.org/wiki/Minimax

6. Alpha–beta pruning - Wikipedia. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

7. Monte Carlo tree search - Wikipedia.https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

8. *On-line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing*, https://dke.maastrichtuniversity.nl/m.winands/documents/CGW_2017_Online%20tuning-Proceedings.pdf.

9. “Monte Carlo Tree Search Methods” - CSDN, 10 April 2019, https://blog.csdn.net/surserrr/article/details/89176654. Accessed 27 May 2022.

10. Choudhary, Ankit. “Monte Carlo Tree Search Tutorial | DeepMind AlphaGo.” *Analytics Vidhya*, 24 January 2019, https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/.

11. Gelly, Sylvain, and David Silver. *monte carlo tree search*, https://webdocs.cs.ualberta.ca/~hayward/396/jem/mcts.html. Accessed 27 May 2022.

12. “Reinforcement learning - MDP.” *CSDN*, 29 December 2017, https://blog.csdn.net/sdgihshdv/article/details/78927589. Accessed 27 May 2022.

13. “What is the difference between a stationary and a non-stationary policy?” *Artificial Intelligence Stack Exchange*, 27 June 2019, https://ai.stackexchange.com/questions/13088/what-is-the-difference-between-a-stationary-and-a-non-stationary-policy.

14. Kaasinen, Jani, et al. “Understanding the Bellman Optimality Equation in Reinforcement Learning.” *Analytics Vidhya*, 13 February 2021, https://www.analyticsvidhya.com/blog/2021/02/understanding-the-bellman-optimality-equation-in-reinforcement-learning/.

15. “Classes of Multiagent Q-learning Dynamics with -greedy Exploration.” *Systems and Computer Engineering*, https://icml.cc/Conferences/2010/papers/191.pdf. Accessed 27 May 2022.

16. Dickson, Ben. “DeepMind scientists: Reinforcement learning is enough for general AI.” *TechTalks*, 7 June 2021,

https://bdtechtalks.com/2021/06/07/deepmind-artificial-intelligence-reward-maximization/.

17. “CNN-LRP: Understanding Convolutional Neural Networks Performance for Target Recognition in SAR Images.” *NCBI*, 1 July 2021, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8272214/.

18. Landau, Ted. 1987. Othello: Brief and Basic, revised edition. Milton Bradley Co. Also available at: http://www.tedlandau.com/files/Othello-B%26B.pdf.