MSC COMPUTER SCIENCE PROJECT DISSERTATION

Implementation of Two AI Algorithms on the Game Othello

Author: Zibo Wang 1919180

Supervisor: Dr Ian Batten

Submitted in conformity with the requirements
for the degree of MSc Computer Science
School of Computer Science
University of Birmingham

Abstract

MSc Computer Science Project Dissertation

Zibo Wang 1919180

This project began with the aim of generating a board game, Othello, which is also known as Reversi, with the implementation of two Artificial Intelligence (AI) algorithms. The game required a winning rate chart GUI to present the result of the AI Player versus Human Player or Random Player. Two main AI algorithms used here are already known that make AI players perform well on other board games. Some of them even allow the AI players to defeat the top human chess player in world, who skilled in mastering a variety of chess strategies, and could rapidly predicting opponent's tactics. These implemented algorithms used simple approach to achieve the same effect. This paper will show how an adversarial search, minimax algorithm with alpha-beta pruning, and a stochastic algorithm, namely the Monte-Carlo tree search algorithm, could be applied in a board game. It will also show how these two algorithms would be improved by implementing the domain knowledge and structural optimisation.

Key words:

- Othello
- AI algorithms
- Game development
- Monte-Carlo tree search
- Minimax with Alpha-beta pruning
- Random function

Acknowledgements

In this project, I would like to appreciate my project supervisor, Ian Batten, for his patient guidance on these weekly meetings and other lecturers on term courses for the kind help and fabulous teaching. Additionally, I would thank to my family for their long-standing support.

Table of Contents

Chapter 1

Introduction

Historically, for a long time, creating a robot that can replace humans to play games is the only way to apply artificial intelligence in games. In the early days of artificial intelligence development, most game AI researchers are trying to make a fabulous AI to defeat humans in board games. Part of the reason for this is that board games contain some basic elements of human intelligence. Most of the rules of board games are simple, but within the square grids the board, countless human geniuses have been working to figure out the best solution for seven hundred years.

Therefore, an AI that can defeat the top players is sufficient to be a grail in the AI field. The history of experts looking for chess game AI can be traced back to the era of Alan Turing and Claude Shannon. At the earliest time, Turing and Shannon tried to use the Minimax algorithm to make AI play game. The reinforcement learning and self-game was quickly applied by Arthur Samuel to Checker (1959).

In this project, Othello is selected as the algorithm implemented object, which is a classic strategic game. It uses a traditional 8 × 8 square grids played by two players in turn. The pieces are divided into black and white. Among the battle, if one player's pieces "clamps" the other side pieces trapping opponent's pieces in a continuous sequence (either horizontally, vertically or diagonally), that could be reversed and become their own pieces. One legal step at least flips one piece of the opponent's pieces (Figure 1). If one player has no more location to put piece, stop it once and allow the opponent to play again. (Engel, 2015) When both sides are in the same status, the game is over, and the player holding more pieces will be the winner. Comparing with "Tic-Tac-Toe", it uses the bigger game board, and more complicated strategies, that could be more representative of the state and performance of an algorithm. The Othello is an entertaining game, it's "simple rules and complex strategies" feature motivates us to explore the effect by using the AI algorithms.

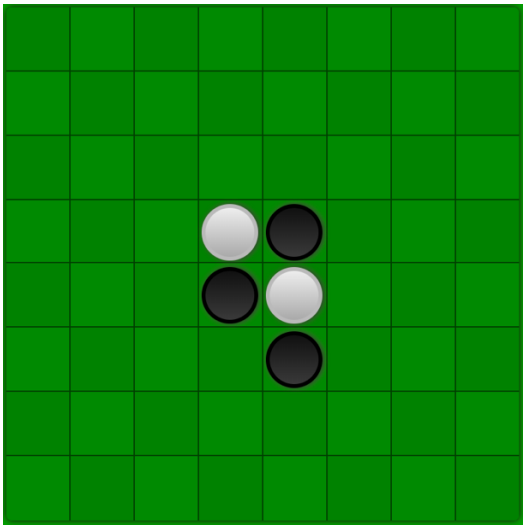


Figure 1 Display Black piece in the beginning

This project aims to develop a board game – Othello with the implementation of two Artificial Intelligence (AI) algorithms as opponents. The game will enable random players or human players to do the battle with the AI players. The primary objective of the project is going to investigate the development process of the game logic and two AI algorithms. The secondary objective is necessarily to figure out the performance difference of two different AI algorithms, in accordance of the reflection from the activity monitor diagram, and a winning rate chart, that clearly show the battle victory record between various type players. The algorithms chosen for the AI players are the Minimax algorithm with the Alpha-beta pruning and the Monte-Carlo Tree Search algorithm. Former one is an adversarial search algorithm; the latter one is a stochastic algorithm. During the process of playing Othello, the easiest way for a player to choose a certain step among several positions is to find an evaluation function that gives a valuation of each situation. If the function is sufficient to fully reflect the pros and cons of the current state, then obviously that is the best action should be taken under state S.

Equation 1

$$A(S) = \arg\max_a V(S')$$

(S' is the next state reached by action 'a' under state S)

As presented by the above function, which definitely has complex form and is unlikely to be parsed. In order to improve the ability of the program and, generally there are two solutions to make up for this deficiency, more detailed algorithm information will be indicated in next two sections:

1. Using a multi-step search, the evaluation function is recalled again in the termination state of the search process.
2. Get a better evaluation function by using machine learning.

Chapter 2

Background Material

2.1 Core Idea of AI

Theoretically, the core idea of the AI is to assume the both players will predict all future moves of the whole game. (Andrey Kurenkov's Web World, 2019) The player will always set a piece, even if the opponent selects the best reaction to that move and all moves in future, the player will still get the highest mark in the end. All of these actions could be easily implemented in an algorithm and ran by a modern computer.

2.2 AI Game History

In 1952, A.S. Douglas developed the first Tic-Tac-Toe board game. Thus, in 1992, the AI "TD-Gammon" of Backgammon based on neural network and time difference for self-contrast training achieved the top level of human beings, which is an critical step for artificial intelligence technology, and many of the used techniques were also applied in nowdays AI, such as AlphaGo. In 1994, Chinook AI defeated the world champion in the Checkers World Championship; by 2007, the optimal solution for the checkers game was also found. IBM's Deep Blue won the chess grandmaster Garry Kasparov in an eye-catching competition in 1997.

Dark Blue was based on a Minimax algorithm that has been modified by a large number of rules and the board evaluation, that was run on a supercomputer. After 19 years, AlphaGo, developed by Google's DeepMind, defeated South Korea's Lee Sedol in five games, which also marks that Go does no longer represent the unique wisdom of mankind. In second year, 2017, China's Go player Ke Jie was also defeated by the upgraded version of AlphaGo in three game rounds, and DeepMind quickly released AlphaGo Zero and Alpha-Zero, based on deep reinforcement learning, MCTS and self-game. It completely abandoned human strategies, that only used the rules of game, not only in Go, but also reached the world's strongest level in both chess and Shogi.

2.3 AI Model

There are many existing computational AI models, that is utilised to generate a learning system. They could be applied on this AI implementation project. Some suitable examples are:

2.3.1 Reinforcement Learning

The algorithm interacts with the dynamic environment, taking the feedback of environment as input, and training to select the optimal action that could achieve the goal. Principles behind the method of reinforcement learning is combining mathematical methods such as discrete mathematics and stochastic processes.

2.3.2 Decision Tree

It can be regarded as a tree prediction model, which classifies instances by arranging them from root to a child. The key of it is the choice of attributes splitting and pruning. A decision tree is a tool that uses a tree-like graph or model to represent decisions and their possible consequences, including the effects of random events, resource consumption, and usage. The decision tree is used to analyse and judge the loan intention. As shown by the Figure 2, which is a decision tree of the Tic-Tac-Toe.

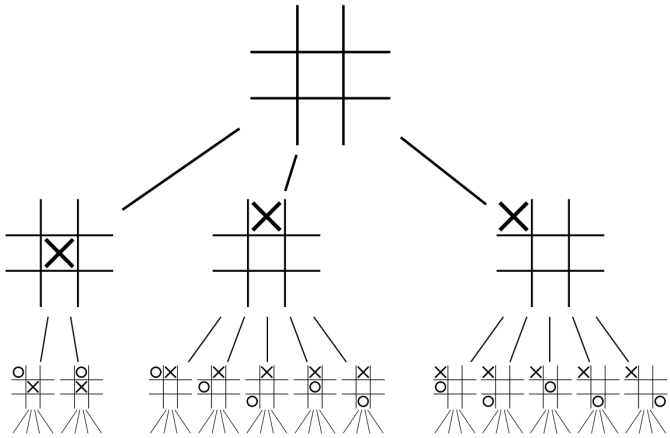


Figure 2 Tic-tac-Toe Decision Tree

2.3.2 Artificial Neural Network (ANN)

A nonlinear, adaptive information processing system composed of a large number of processing unit interconnections. It is based on the results of modern neuroscience research, trying to process information by simulating the way the brain neural network processes and memorizes information. As shown by Figure 3, ANN is a parallel distributed system. It adopts a completely different mechanism from traditional AI and information processing technology. It overcomes the shortcomings of traditional logical symbol-based AI in dealing with intuitive and unstructured information. It is adaptive and has Self-organizing and real-time learning features.

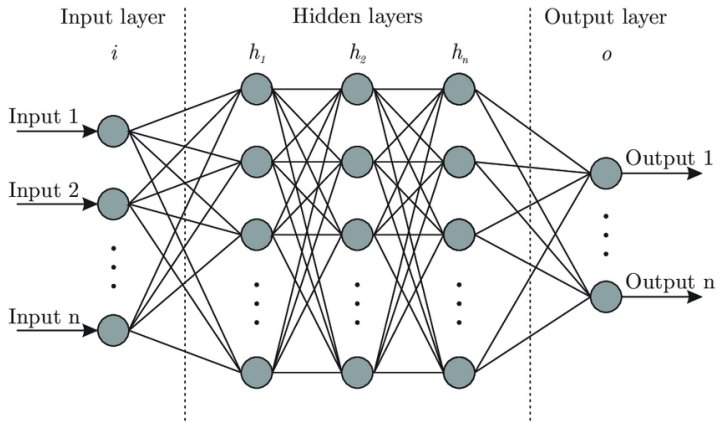


Figure 3 Artificial Neural Network Diagram

In this project, selecting a proper learning technique to be used by the AI agents is a crucial problem, which can make it work effectively. The methods chosen from above one is the decision tree, Minimax as one of its typical algorithms, that has been commonly applied in

many AI board games, it will be able to achieve a better result in accordance of a number of researches. Another one is using the Reinforcement Learning to train the AI agents, the Monte-Carlo Tree Search (MCTS) will be a significant algorithm applied at here. The problem that MCTS has to solve is that the search space is large enough to calculate the value of all subtrees. This requires a more efficient search strategy, and it must also be explored and utilized to avoid falling into local optimal solutions.

2.4 Existing Board Games AI Agent

An existing game agent that applies the reinforcement learning is the famous AI agent AlphaGo that was used to defeat the top Go player in the world, Lee Sedol (Chowdhury, 2019). It was developed by the Google DeepMind team; they implemented the Monte-Carlo Tree Search and the deep learning algorithms into the agent. The fundamental idea of the agent is going to simulate the future game multiple times and then choose the one that was selected the most among the number of simulations.

1. Based on the SL Policy Network to predict the next step in the future, until the 'L' step, reach a node. If the number of accesses to the node is greater than a threshold, then use the Tree policy to expand the next node.
2. Integrate two methods to evaluate the trend of future to the L. One is to use Value Network to evaluate and judge win. The other one is to use the Rollout Network to make further predictions until the end of the game to get the results of the simulation. The combination of the two (both weights of 0.5) evaluates the prediction of Future L steps.
3. After the evaluation, the result is taken as the Q value of the next game under the current game. The next step to be given at the beginning will be evaluated in accordance of the future direction. The larger the Q value, the more times the simulation will be selected.
4. Combine the Q value of the next move with the SL Policy Network for another simulation. If the same move occurs, taking the average of Q value of the move.
5. Repeat the above steps for n times, and then select the one that has been selected the most times as the next step.

2.5 Existing AI and Machine Learning Frameworks

2.5.1 TensorFlow

It is a commonly used and open source software framework developed by Google that is designed for deep learning or artificial neural networks. It allows people to create neural networks and computational models by using flowcharts. It is one of the best easy-maintenance and the most popular open source libraries available for deep learning. The TensorFlow framework can be used in C++ or Python. People could use TensorBoard for simple visualization and view the calculation pipeline. Its flexible architecture allows developer to easily deploy on different types of devices. The disadvantage is it has no symbol loops and does not support distributed learning. In addition, it does not support Windows system.

2.5.2 Theano

This is a Python library designed for deep learning. Developer could use this tool to define and evaluate mathematical expressions, including multi-dimensional arrays. It optimised the GPU, the tool also has NumPy integration, dynamic C code generation and symbols differentiation. However, in order to achieve a high degree of abstraction, the tool must be used with other libraries such as Keras, Lasagne and Blocks. Theano supports platforms such as Linux, MacOS X and Windows.

2.6 Game Programming Language

There were many reasonable factors as to why the Java programming language was selected to develop this project. The reasons are listed as follows: firstly, we have spent one year on the course Java Workshop to just learn this language, the proficiency of this language allows people to program and implement some functions more easily in the project. Secondly, comparing with C++ programming language, Java is fully objectified. For instance, array is an object in Java and contains a property of length; unlike an array in C++, it is a pointer, even it is faster, but it is not safe. So, accessing arrays, Java will perform boundary checking, which is safer, but at the expense of speed. At the same time, because all classes in Java inherit the base class of Object, people could associate several unrelated classes with the base class, such as in the same array. Java also has a perfect memory management mechanism, which can automatically garbage collect, which may reduce the possibility of memory overflow and improve programming efficiency.

On the other hand, especially for a board game, the multithreading and sockets supported by Java, that could occupy less memory and generates available space on the CPU, it will not ban developer while the strong processes are running in the background (Games and Bhargava, 2019). Additionally, the sockets would be able to help to develop a multiplayer game. Because Java runs on virtual machines (JVM), the game will be easier to allocate.

Chapter 3

Research

Board game is a typical dual agent, complete information, zero-sum game process (Nilsson, 1998), each player must be completely familiar with the environment, all possible moves and influences on themselves and opponents. So basically, the gamer's strategy search process can be represented by a tree structure, called a game tree. The nodes on the tree are the states in which the board may appear, and all the successor nodes derived by the parent node through one step are its children. In order to select one of the many action plans that are most beneficial to them, it is necessary to analyse the current situation and what will happen, then select the optimal walk through a search algorithm.

In this Othello game problem, there are many action plans for each pattern, so a very large game tree is generated, trying to pass the search until the end. It is impossible to get the best move, so people can only search forward for a limited number of steps. The most basic search strategy is called minimax search, that will be given detailed explanation in the following content.

3.1 Evaluation Function

An efficient evaluation function is important and will be commonly used by many AI board games. In a simplified version, mainly used the evaluations that are based on game board and piece mobility. The game board is actually a weight table for different positions on the board. The action of one side is the total number of positions where it could be displayed. If 'S' is used to indicate the state of the board (the state could be defined as the presence or absence of the piece or the piece colour of each piece), then based on the following models:

Equation 2

$$V(S) = \sum_{i=1}^{8}\sum_{j=1}^{8} \omega[i,j] \cdot s[i,j] + Mobility\ (my\ colour)$$

Equation 3

$$s[i,j] = \begin{cases} 1(colour[i,j] = my\_colour) \\ -1(colour[i,j] = opp\_colour) \\ 0 \qquad\qquad\qquad (otherwise) \end{cases}$$

As presented by the above two equations, colour [i, j] indicates the piece colour at the i row and j column, which could be abbreviated as (i, j) on the board. It could be my piece colour, the opponent's piece colour or be blank. Ω [i, j] is the weight at (i, j) and is empirical data. Mobility is the value of your own mobility.

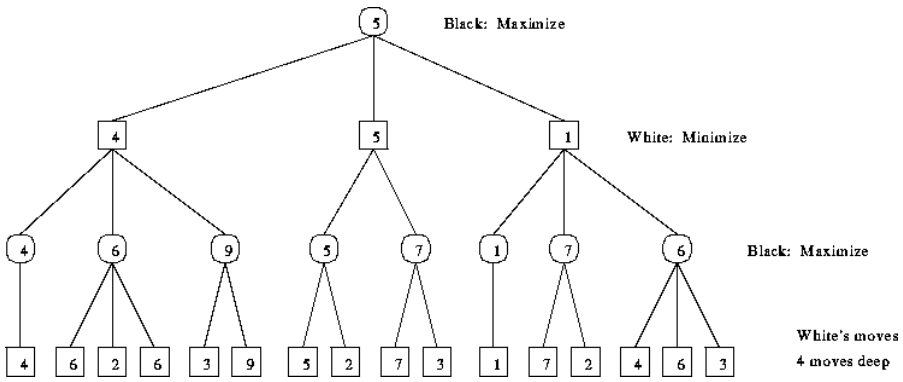
3.2 Minimax Search

Assuming the two players involved in the game are Black and White. From the view of Black, Black's action at each step always gains for its maximum benefit, called the Max process,

which the node on corresponding game tree is called as Max node; The game ability is not clear, so every step of White always minimizes its own benefit, called the Min process, corresponding to the Min node on game tree. The Othello is actually the process of alternating Max and Min.

Assuming now is Black turn, it finds the node with the largest evaluation value among all possible action-derived child nodes is S' i, then the maximum benefit is V (S'i). − V (S), the corresponding action AI is the optimal solution that can be found in a single-step search. But what if Black searches forward two steps? It may find that the minimum value of the child nodes derived from S'i is V (S'i), which is smaller than the minimum value V (S"j) of the child nodes derived from some other node. j is the largest node among the minimum values of the child nodes derived from the same layer node, then it should select the action aj corresponding to S'j as his best action, because the profit of selecting aj is not less than the V (S"j) − V (S), and this is already the best case. If the search continues to deepen, then the choice will become more and more "wise." An example is presented by the Figure 4. The implementation of the algorithm is similar to the depth-first search of the tree. The Max (Min) node takes the largest (small) value in these children nodes and returns it to the parent node as its own estimate value. The pseudo-code in Figure 5 is used to represent the algorithm.

Equation 4 Minimax Game Tree



1. The Max node is the displayed piece by yourself, and the other side will be the Min node.
2. The leaf node calls the evaluation function and propagates its value up;
    a. The Max node is assigned the maximum value of all its children nodes;
    b. The Min node is assigned the minimum value of all its children nodes;
3. The root node selects the action corresponding to the node with the largest value among its children nodes as the best action.

```
Input: board board and max search depth depth
Output: best move best_move
1 best_val ← −∞;
2 foreach move do
3     MakeMove(board, move, my_color);
4     val ← Min(board, depth − 1);
5     UnMakeMove(board, move, my_color);
6     if val > best_val then
7         best_val ← val;
8         best_move ← move;
9 return best_move;
```

Figure 4 Pseudocode of Minimax

3.3 Monte-Carlo Tree Search (MCTS)

MCTS is a process of gradually building an asymmetric search tree by randomly deducing the game. It can be regarded as a reinforcement study in a certain sense. The MCTS could be divided into four steps. Selection, Expansion, Simulation, and Backpropagation. In the beginning, the search tree has only one node, which is the status we need to make decisions. Each node in the search tree contains three basic pieces of information: represented game status, number of times visited, and the cumulative score.

3.3.1 Selection

In the selection phase, it is necessary to select a node N that is most urgently required to be expanded from the root node. The situation R is the first node to be inspected in each iteration; In terms of the inspection situation, there are three possibilities:

1) All feasible actions of this node have been expanded
2) The node has a feasible action has not been expanded
3) The game situation on this node is over

For these three possibilities:

1) If all feasible actions have been extended, then we will use the UCB formula to calculate the UCB value of all the child nodes of the node; and find the child node with the largest value to keep checking. Repeat iteration.
2) If the checked situation still has child nodes that are not expanded (e.g a node has 20 feasible actions, but only 19 child nodes are created in the search tree), then we consider this node as the target node in this iteration. And find action A that has not been extended by N. Repeating step [2].
3) If the node being checked is a node whose game has ended. Then step [4] is directly executed from the node.

The number of visited times of each checked node increases at this phase. After repeated iterations, we will find a node at the bottom of the search tree to continue next few steps.

3.3.2 Expansion

At the end of the selection phase, we found a node N that was most urgently expanded, and an action A that was not yet expanded. Create a new node Nn in the search tree as a new child of N. The Nn is the situation after node N performs action A.

3.3.3 Simulation

In order to get an initial score for Nn. We started with Nn and let the game go random until we get an ending, which will serve as the initial score for Nn. Generally, victory/failure is used as the score, only 1 or 0.

3.3.4 Backpropagation

After the simulation of Nn ends, its parent N and all nodes on the path from the root to N will add their own cumulative score based on the results of this simulation. If a game outcome is found directly in the selection, the rating is updated based on the outcome. Each iteration expands the search tree. As the number of iterations increases, the size of the search tree increases. When it reaches a certain number of iterations or ends, select the best child node under the root node as the result of this decision.

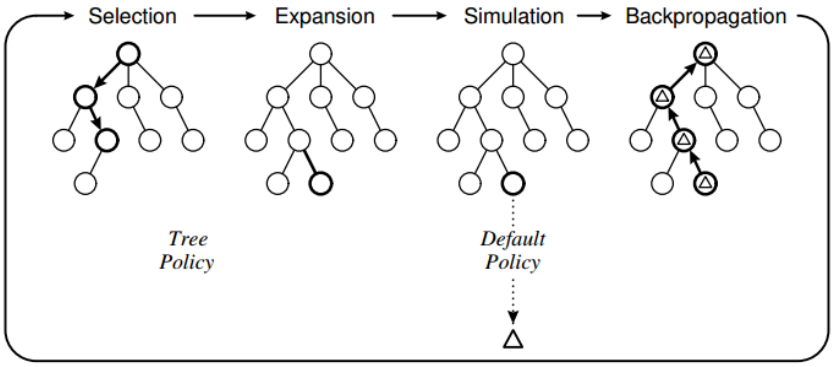


Figure 5 MCTS Statement

Chapter 4

Analysis and Specification

4.1 Requirements Analysis

The requirements for the AI Othello were splitted into two different sections: functional and non-functional requirements. The functional requirements present these detailed functions and algorithms that were truly implemented by the game among the development process. Plus, the non-functional requirements mainly elaborate the performance characteristics of the game, for instance, the AI response time (in seconds) must be less than 1s while doing the battle with other AI players or human players. These requirements of the game were all defined by using the MoSCoW method, which in terms of the four verbs “must”, “should”, “could” and “would” to prioritise how critical each requirement will be.

4.1.1 Functional Requirements

- *Othello Game Rules*

- All pieces must be put upon an eight-by-eight grid squared game board.
- At the end of the game, there must be 64 pieces have been settled on the board, it does not matter with the number of each kind piece, only require the total pieces number equals to the 64.
- The piece could only be laid adjacent to an opponents’ piece grid, or any mouse click on the other grids will not put piece.
- The game starts up with four pieces crossing with each other (two black and two white), they must be placed in the four squares in the middle of the game board, no pieces are captured and removed.
- Players’ pieces clamped by opponents’ pieces in a horizontal, vertical or diagonal row means these pieces are “captured” and turned over to match the opponents’ colour.
- If there is no suitable place to display piece, the opponent is allowed to put piece continuously.

- *Game Board (Graphical User Interface)*

- When human player clicked any grid on the board, a crossponding colour piece will be put on the specified place.
- A pull-down style menus bar must be put at the top of the border pane, it will enable player to select the game mode and do setting for the game. One is called as “Game”, the other one is called as “Setting”.
- At the top of the game window, the title “AI Othello” must be presented.
- The “Game” menu button must contain “Start Game” and “View Winning Rate Chart” functions.
- The “Mode” menu button must contain five different game modes, including “AI Ab V.S Random Player”, “AI Ab V.S AI MCTS”, “Human Player V.S AI MCTS”, “Human Player V.S AI Ab” and “Human Player V.S Random Player”.

- At the right-hand side of the game board, player will be able to view the current self-piece number and the opponents' piece number.
- At the right-hand side of the game board, a piece will be presented, that show the current players' piece colour.
- When any player wins the battle, the winning information "xx wins the Game" must be presented at the right-hand side of the game board.
- Winning Rate Representation.
- User will be able to view a winning rate chart, to present the result of two different players after a number of game rounds.
- The winning rate must be presented by a bar chart, the X axis is two players type, Y axis presents the winning rate percentage.
- The chart status cannot be changed among the game round.
- After one round finishes, if user want to review the latest result, he/she must close the old chart window then open a new one.

▪ *Game Mode*

- The game mode is divided into two sections, one is human player versus AI player mode, other one is AI player versus AI player mode.
- Human Player V.S AI MCTS: enable human player to battle with the AI player implementing the MCTS algorithms.
- Human Player V.S AI Ab: enable human player to battle with the AI player implementing the Minimax algorithm.
- Human Player V.S Random Player: enable human player to battle with the Random function.
- AI Ab V.S AI MCTS: enable AI player with Minimax to battle with the AI player implementing the MCTS algorithms.
- AI Ab V.S Random Player: enable AI player with Minimax to battle with the Random function.
- While one game mode is running, player changes the mode in setting menu will not interrupt the game. Only by clicking the "Game Start" option will change the mode and restart whole game.

▪ *Player Piece Count*

- In accordance of the players' current piece number, the number on piece count board will change as well.

▪ *Animation (Piece Colour Change)*

- Among the game round, in accordance of the recent player, the colour of the representation piece on right-hand board will change as well.
- While players' pieces are clamped by opponents' pieces, they must be changed into opponents' colour.

4.1.2 Non-functional Requirements

▪ *Usability*

- The language of the game must be English.
- The game must have a refreshing interface that is as quick as possible.
- The game must be able to give a high-level game experience for players.

- *Efficiency – Performance*

- The AI players must respond to the users' feedback less than 1 second.
- The game must strictly follow the game rules as the user requires.
- After the human player clicks on the "Game Start" option button, she/he must be presented a new game board in no more than 2 seconds.
- After the human player clicks on the "Winning Rate Chart", he/she must be presented by the chart in no more 1 second.

- *Efficiency – Space*

- Adding extra AI algorithm into the game must require minimum changes to the whole game code.

- *Dependability – Reliability*

- The game must not crash more than one time after doing 20 game rounds.

4.2 Game Testing

Among the process of the computer game development, "play testing" is one critical type of the user testing, that will enable the users from various levels to play the same computer game and evaluate it. At least ten rounds of play testing between the human player versus AI player, and the different AI players battle with each other will be carried out to ascertain which AI player with the best performance compared with others. In accordance of the feedback from winning rate charts and the algorithms code implementation to figure out what factors making it be the best AI algorithms in this project. There will be only one limited human player to attend each user testing, the purpose is to create an invariant to maintain the accuracy of the experimental results and remove the interference factors of different level players. Additionally, the more time available, the more times of the battle rounds will be carried out to make results be more precise.

4.3 Development Tools

In this project, the Java programming language was selected as the main development tool for the game, the Eclipse was utilised as an Integrated Development Environment (IDE). One standard GUI library, JavaFX, that was included in this project for the purpose of drawing the user interface of Othello Game Board and the Winning Rate Chart. To make the GUI programming process be easier, an external visual layout tool, Scene Builder, was used to draw the GUI, without coding. It only requires programming a controller class in Java, then connecting functions with these pre-defined control units in JavaFX files.

4.4 Game Concept Design

4.4.1 Unified Modelling Language Graphs (UML)

- *Class Diagram*

As indicated in the Appendix 3.3, a detailed class diagram is presented to explain the relationship among these implemented functions, multiplicities and these classes.

4.4.2 Graphical User Interface Conceptual Design

- *Game Board Design*

  As shown by the following Figure 4 and Figure 5, these are simple hand sketch design of the Othello Game Board. Because it is an AI project, there is no much need on the decoration of the GUI, it just needs to meet most basic functional requirements. In implementation process, javaFX was used as the main tool to design these graphs and Scene Builder was applied to make whole process be easier.

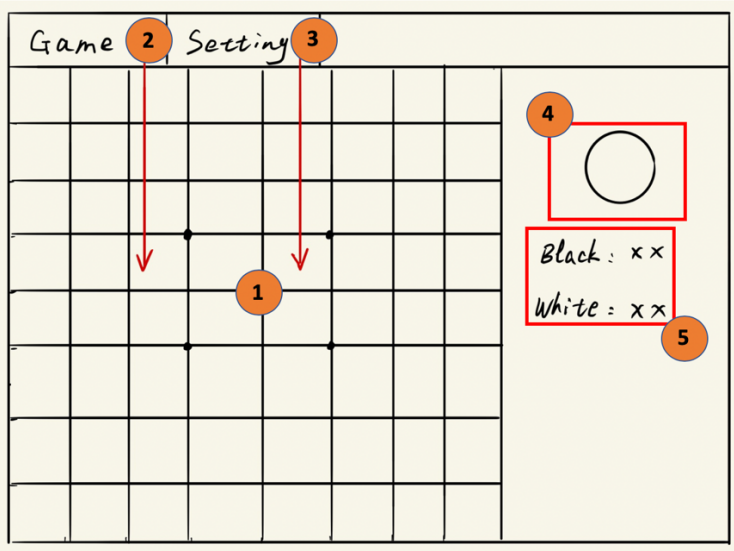


Figure 6 Hand Sketch 1

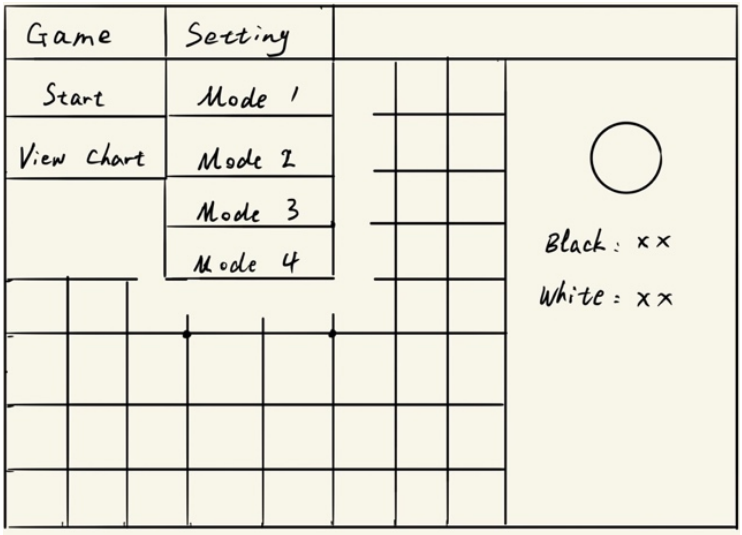


Figure 7 Hand Sketch 2

- Stamp 1: an eight-by-eight square gird game board, all these pieces will be put upon it.
- Stamp 2: this is a pull-down style menu button, named as “Start”. While player clicks on it, a menu will be extended. As presented by the Figure 5, it includes two options, one is used to start the game, it is also used as a reset button. The other one is used to view the Winning Rate Chart.
- Stamp 3: just like stamp 2, it is a pull-down style menu button, named as “Setting”. While player clicks on it, a menu will be extended. As presented by the Figure 5, it includes many different game modes.
- Stamp 4: this is a current piece indicator, in accordance of the current piece player, it will change to the corresponding colour.
- Stamp 5: this is a piece number count board, according to the situation of the two players on the game, it will present piece number of each player. When the game finished, there will be one more line to be added, showing which side is the winner.

- *Winning Rate Chart Design*

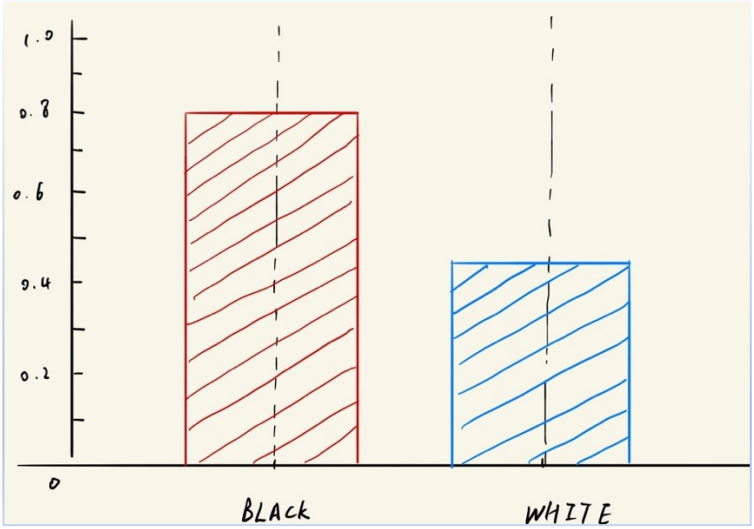


Figure 8 Hand Sketch 3

As shown by the Figure 6, that is the hand sketch design of the Winning Rate Graph within the bar chart style. After each game round, player can click the “Review Chart” option under the “Game” menu button, it will clearly show the wining rate percentage for each player. The X axis presents the two game players, the black piece and white piece. The Y axis indicates the winning rate in one percentage.

Chapter 5

Implementation

5.1 Game Logic Functions

- *Piece Display Function*

  This function is used to display the piece on these valid positions while the player clicks the specified grid. As presented by the Figure 18 in Appendix, a nested for loop was applied to traversal all the piece icons on the game board. The javaFX package was imported, a Circle object was created for the piece GUI creation. Radius was set for 20, and colour was also defined as a constant value which would be convenient for colour change in later phase. To make sure the piece would be located on the middle of each grid, the coordination centre X and Y should be equalled to the half-length of grid's width and width. Thus, a "If" conditional statement was used to judge if a specified position was empty or not. Because two pieces types were pre-defined in the pieceType class, they were two constant values with the enumerated type, "BLACK" and "WHITE". If the current step was black piece, the colour of the piece circle would be change into black, vice versa for the white piece.

- *Piece Set Rules and Reverse Function*

  This function is used to present the piece display rules, which could also be considered as Othello rules logic. Comparing with the chess, Othello follows an extremely simple rule. Rule one is the piece can only be displayed on these positions, that is adjacent to another piece, and there cannot exist any gap between two continuous piece steps. When one piece was already located on, game logic should be able to judge locations in eight directions, to see if any of these are available for next piece step to be located upon. As presented by the Figure 19, regarding the current piece coordination as (0,0), X presents the row, Y presents column. So, coordination in eight directions will be (0,1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1). To unify the pieces location in eight directions will greatly reduce the code redundancy on the location judgement logic. In the below code, a conditional operator was used to decide the type of next displayed piece. So, the coordination of the new displayed piece could be easily calculated by the following function, the existing piece coordination add the move grid number from the both column and the row, then using the calculated value as the new piece coordination.

  Rule two is about these captured pieces, only while they were clamped by two opponent's pieces in a row, these pieces could be flipped. They also should be reversed and change into opposite colour. Here using the same method as before to generate out the coordination of the displayed piece, in terms of the pieceType object in conditional operator to define the piece colour.

- *Validate Positions Function*

This function is used to find these validate positions which will enable player to locate the piece upon. As presented by the Figure 20, the first “if” condition was used to judge if the specified grid was occupied or not. If the square was not empty, the judgement finished. Same as previous two functions, the logic code for eight directions judgement was overwritten again.

- *Player Change Function*

  This function is used to alternate the player turn, which could be easily implemented. As presented by the Figure 21, setting two constant values, 1 and 0 for two types players. Among the game, while the current player is a black piece holder, he could be set as “player [0]”. Using the “if” condition to judge, if “currentPlayer == player[0]” is the current situation, change the next player into “currentPlayer == player[1]”. Additionally, change the piece colour into the white.

- *Button Click Function*

  This function is used to activate these new windows while user mouse clicked on the specified button. As shown by the Figure 22 in Appendix, the winning rate chart was generated into a FXML file. To make the chart click function within the ActionEvent, so when user clicked the optional menu, using the fxml loader to load the page that you want to present.

- *Score Count Function*

  This function is applied by the piece number count board, which will correctly count each player’s piece number on the game board. The implementation code was presented in the Figure 23 in Appendix. At the beginning, set the score value as zero. While the piece type was defined. For example, now is in the Black turn, using the double for loop to traversal whole game board. For each traversal, the value of score would increase for one. The GUI section called this function to show the number.

5.2 AI Algorithms

- *Minimax with Alpha Beta Pruning*

  Comparing with the commonly used Minimax algorithm, the algorithm used here is more efficient. It applied the alpha beta pruning on the original code, which greatly decreased the tree search range. As presented by the Figure 24, the algorithm implements two values for each node: α and β, respectively representing the lower and upper limits allowed by the node's estimate. The two form an interval, called as the α-β window, and the length of the window is defined as $\beta - \alpha - 1$. Initially, the root node's $\alpha = -\infty$, $\beta = \infty$, the other nodes' α, β values are inherited from its parent node during the depth-first search. For the following statement (Figure 9), after searching for B, we know that A's final estimate is definitely not less than the value of B (because A is the Max node), which is the alpha value of A, $\alpha = 6$. Continue the depth-first search. When E is searched, the upper limit of C is determined (C is the Min node). At this time, the β value of C is 4, and it is noted that the α of C is also 6 (this is inherited from A).

It is known from the above discussion that C is pruned at this time, in fact, when β ≤ α When it happens, pruning will occur.

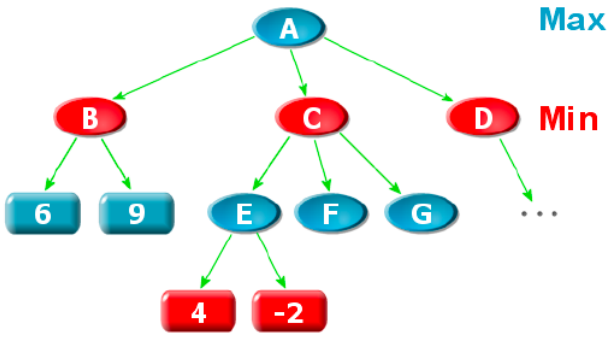


Figure 9 Simple Algorithm Statement

Basically, both α and β were updated during the depth-first search. For the Max node, if the return value of the child is greater than α, α is updated to the return value. For the Min node, the process is just the opposite, and β is updated to the return value of the child node smaller than it. The α value of the Max node does not decrease, and the β value of the Min node does not increase. When β ≤ α of a node, pruning will occur. The puncture of the Max node is called β pruning, and the Min node is called α pruning. According to the relationship between the return value of the child node and α, β, the child nodes can be divided into three categories (taking the Max process as an example, S represents the corresponding Max node, and S' represents a child node):

1. **Very poor node:** V (S') ≤ α, S' will definitely not be selected, need to continue to search for the next child node;
2. **Very good node:** V (S') ≥ β, β pruning occurred;
3. **Main step node:** α < V (S') < β, pruning cannot occur, but it may be selected, and the search needs to be continued.

- *Monte Carlo Method (Random Player)*

  In this game, a random player could be regarded as a fresh player, without any knowledge on the game strategies, like avoid walls, and keep control of the diagonals. The only thing it could do is randomly put pieces on these valid positions, which means next step is the chance of positive step is 50%. As shown by the Figure 26, just need to call the Random() method, then create an Random object in java, and make it randomly display piece on these validate positions in each game round.

- *Monte Carlo Tree Search (MCTS)*

  This is another crucial AI algorithm used in the project, the main object of generating the MCTS is going to compare with the above Alpha beta search algorithm, to see which of them perform better under the same situation. The demonstration on how to implement the algorithm was already explained in the previous Search section 3.3. The algorithm code is also presented in the Appendix 2, so here will do not make detailed statement.

5.3 Graphical User Interface Creation

- *Othello Game Board*

  As shown by the following, Figure 10 presents the initial state of the Othello while player just start to run the project in IDE. If the “Game Start” option under the Game menu was activated by mouse click, the game board status was presented by the Figure 11. Four pieces were displayed at the middle, and the count board where under the indicator piece started to count the piece number. All of the design elements on the GUI meet these requirements demonstrated in previous design specification section. Except the yellow game board and the piece indicator, all rest control units were created by Scene Builder, they were displayed on a border pane with the size of 579 by 440, the file form is fxml. Then connecting them with the functions in controller class. The board main body was created by java coding, that was presented by the Figure in Appendix. Just using a double for loop to create 64 square grids on the board, named as boardIcon in controller class. Setting the grid size as 50 and give the mouse click event on each of them, that allowed to activate new event by mouse click.

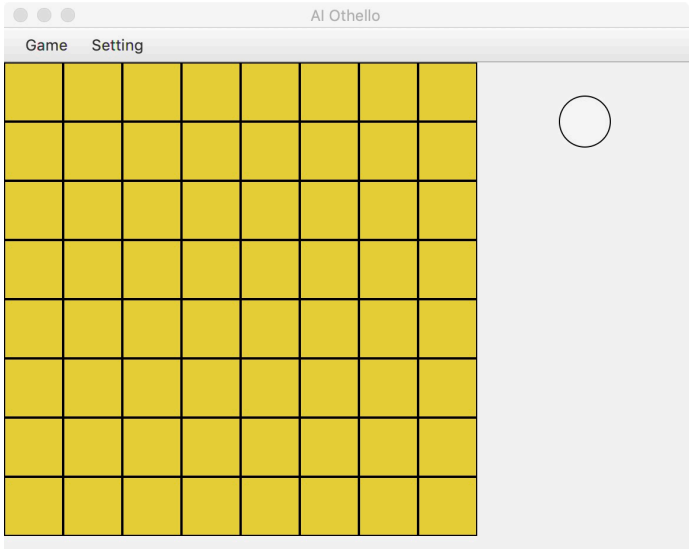


Figure 10 Game Board 1

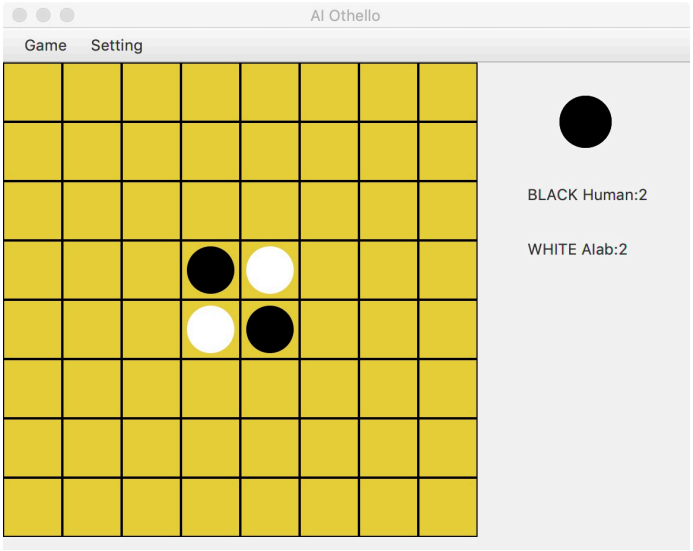


Figure 11 Game Board 2

- *Optional Menu*

  Same as the game board, Game option menu and Setting option menu presented by the Figure 12 and Figure 13 were created by using the Scene Builder with the fxml file format. These used control units include the menu button and menu items. Writing an action event type function in the controller class, then connecting it with these control units. While player is clicking on the menu button, a pull-down style menu will be extended. Any options could be checked by mouse click on.

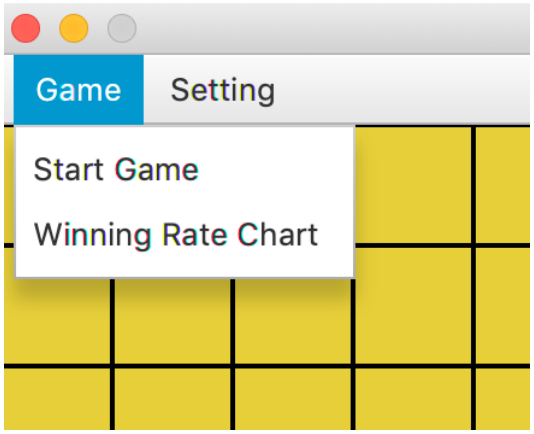


Figure 12 Option Menu 1

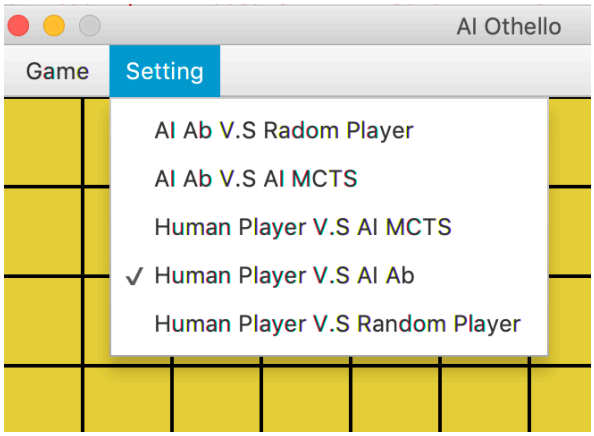


Figure 13 Option Menu 2

- *Winning Rate Chart*

  The following bar chart was designed by the Scene Builder in fxml file format, using the bar chart control unit. The bars were paint with orange colour to make them be visible. In the chart controller class, the setData function was used to set the data property of the X and Y axis. As the game mode changed, the data property on X axis would also be changed.

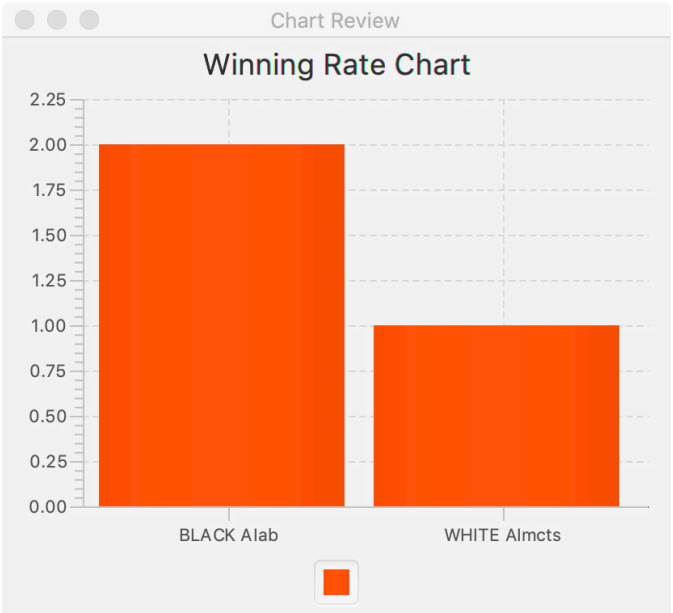


Figure 14 Bar Chart

Chapter 6

Project Management

The whole project has been well managed in the last three months, which exactly followed the process from the pre-defined Gantt Chart in the proposal. The detailed information is presented as below Table 1:

Table 1 Completed Case

Time Duration	Completed Case
10.06.19 – 19.06.19	Decide the project topic and do related work search
20.06.19 – 24.06.19	Determine the methodology and finish project proposal
25.06.19 – 16.07.19	Programming the Othello game logic function
17.07.19 – 18.07.19	Testing the game logic to achieve two human players battle
19.07.19 – 25.07.19	Go over related papers collected and extract useful AI methods
26.07.19 – 07.08.19	Learn AI algorithms and implement it on the existing Othello game logic
08.08.19 – 10.08.19	Testing the Othello to achieve human-AI game battle
10.08.19 – 12.08.19	Try to improve the existing AI algorithms to increase the winning rate
12.08.19 – 13.08.19	Program debug
13.08.19 – 14.08.19	Wrap up the work
16.08.19 – 23.08.19	Preparing for the project demonstration
23.08.19 – 09.09.19	Finish the report

Chapter 7

Results

7.1.1 Battle Results under Different Game Modes

Owning to the core object of the project is to observe the difference on performance between two different AI algorithms. The testing mainly focuses on two game modes, one is the “AI Ab V.S Random Player”, other one is the “AI Ab V.S AI MCTS”. Each game mode was running for twenty game rounds to figure out while under the same game rounds times, which AI algorithms performed better on this the Othello. The final game round screen shots and winning rate chart of each mode were presented as below. The detailed analysis was demonstrated in the conclusion (Section 9).

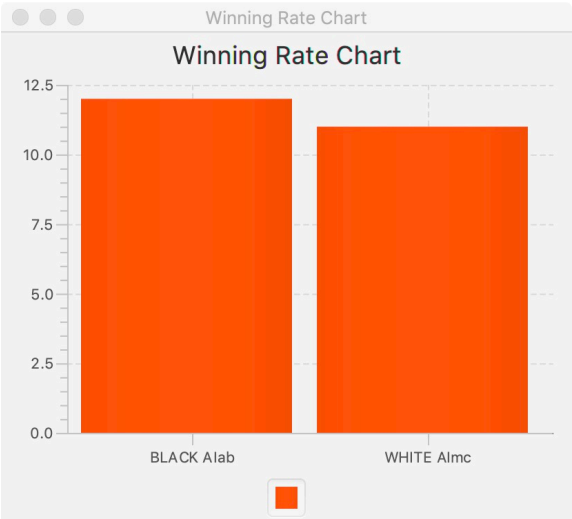


Figure 15 AI Ab VS Random Player

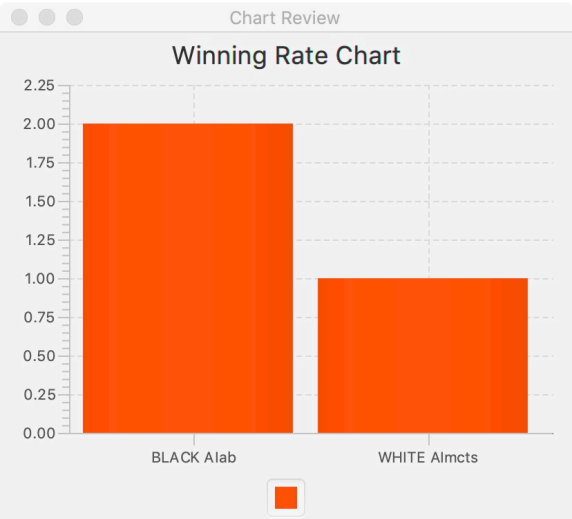


Figure 16 AI Ab VS AI MCTS

7.1.2 Testing Result

Software testing is a critical section among a significant software project development, in this implemented process the object software must be evaluated by detecting the differences between the given input and expected output. It is a method of quality assessment for the product (Zafar, 2019). At here, two mainly used testing are the Functional Testing and the Graphical User Interface Testing. The former ignores the internal code logic, it only focuses on the output results to check if the program performance met these pre-defined requirements, that is a type of the Black-Box testing cooperated with the functional requirements of the software. The later one is used to test the GUI functions and layout, to see if they match with the conceptual designs in the detailed design document. Totally, fifteen test cases were generated out, and the corresponding expected outcomes and results are presented by the following Table 2.

Table 2 Testing Results

Test Case	Expected Outcome	Result

When press “Start Game” button, four pieces (2 black, 2 white) in cross position will be put in the middle of game board	Pressing “Start Game” button, four pieces (2 black, 2 white) were displayed in the middle of game board within cross position	Pass
Press “Winning Rate Chart” option will be able to view the bar chart	Pressing the “Winning Rate Chart” button, the bar chart was presented	Pass
Click the “Game” menu button, a pull-down option menu will be presented	Clicking the “Game” menu button, an optional menu was pulled down	Pass
Click the “Setting” menu, a pull-down option menu will be presented	Clicking the “Setting” menu, an optional mode menu was pulled down	Pass
After starting game, click on the valid position grid will enable to display piece	Among the game round, clicking on these valid positions, pieces were displayed on the board	Pass
If there is a piece in the grid, click the same place will not be able to change the piece colour	Double clicking the grid that already occupied by a piece, the piece colour did not change	Pass
Without specific situation, the piece display is alternating step by step	Among the game round, the piece colour changed step by step	Pass
As a number of game rounds finish, the Winning Rate Chart will change	Under the same game mode, after five game rounds the winning rate changed comparing with after two game rounds	Pass
When game is in AI V.S AI mode, two AI players will battle following the game rules until the end	When the game in AI Ab V.S AI MCTS mode, the piece display and capture step exactly followed the Othello rule	Pass
When current player’s pieces clamp opponent’s pieces, these captured pieces will change colour	In the game, when the human player’s pieces clamped opponent’s pieces, they were changed colour	Pass
Piece count board will clearly count each player’s piece number on board	Piece count board exactly counted the number of each player’s piece in game	Pass
When one side player wins, piece count board will correctly present the information	Black side player won the game, piece count board presented the information “BLACK AI Ab wins”	Pass
The colour of the indicator piece will be changed among the game	When the piece display side was black, the piece indicator changed the colour form white to black	Pass
When two AI players are in game, if the mode is changed, game will immediately stop	When the “AI Ab V.S AI MCTS” mode was running, the game mode was changed to the “Human Player V.S Random Player”, the game process stopped immediately	Pass
Invalid positions will not be able to display on any piece	Clicking an invalid position, there was no piece displayed	Pass

Chapter 8

Evaluation and Discussion

8.1 Piece Move Decision Comparison

From the perspective of the human player, there are many tricky strategies could be used among each game round. Playing for the corners is one of the powerful strategies. As shown by the Figure 15, when Black took the corner, the white pieces along the vertical and diagonal turned over, and black was ahead from 20 to 24. Obviously, the corner taken is one of the powerful spots. Player will have the opportunity to change a number of pieces when the player got it. But the more important thing is once you displayed it, your pieces could not be surrounded or captured. This is the fun point of the Reversi, any displayed piece would be able to change the status of the whole game.

However, owning to the limits of the human brain's power, as a skilled player the maximum piece step that you will be able to predict is less than five, this is one of the problems that human player should face to. Comparing with the AI algorithms, like the Minimax, in terms of the large search depth, it will be able to find the optimal solutions among these steps which may lead to failure. This is also the key idea helping AI player to move the piece and show the biggest thinking difference from human player. The Monte Carlo Tree Search depends on its design, the search tree will automatically focus on "the changes deserve more worthwhile search". If you find a good move, MCTS will search it in depth very quickly, it can be said that it combines breadth-first search and depth-first search, similar to heuristic search.

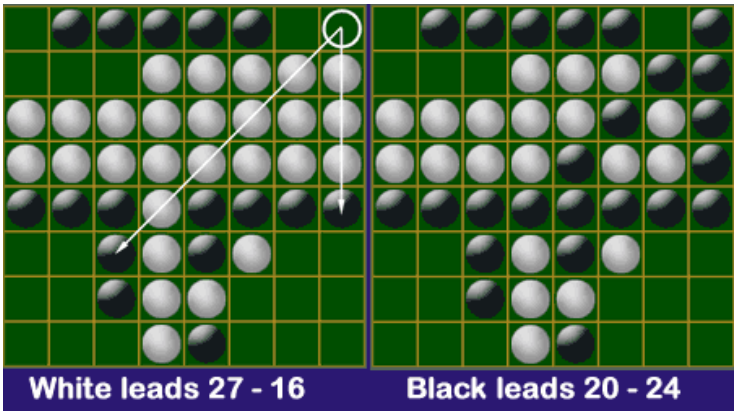


Figure 17 Corner Taken Strategy

8.2 Choice of Two AI Algorithms

First of all, these two algorithms are commonly used in the popular AI board game in recent years, and they are constantly being improved until today. This also reflects that they are fabulous algorithms. Secondly, based on the knowledge that has been learned so far, and the knowledge expansion and paper research done in the past three months, it clearly showed that the implementation of these two algorithms is a little bit challengeable and in moderate complexity. It also guaranteed the completion and success rate of this project. Additionally, it's also very interesting to allow two algorithms do battle in the game, which be able to help on observing advantages and disadvantages of the two algorithms by comparison.

8.3 Bigger Board Affect

If there is a board scale resize function has been added, player could choose to have a bigger board with the size of ten-by-ten or sixteen-by-sixteen. Based on the Minimax algorithm, under a limited board size scale, a bigger board will affect the outcome with two problems. First is the search tree will be too broad. The board is too big, and each player has a lot of choices at every step. The second problem is will be difficult to assess the winning percentage. Unless the search tree is brought to the end, it means going to more than one hundred steps. Because it is even hard for the computer to judge when will be the end of the game, so the only thing could do is keep displaying pieces to fill the board, until both sides have no place to go. Simply speaking, the search tree also requires to be particularly deep. To achieve the same performance as this project, the search depth must be increased. At this kind of situation, MCTS will be good choice. It can give a situation assessment, even it is not accurate. This partially solves the second problem. In accordance of the search property, the search tree will automatically focus on these moves that has percentage to win. If the player finds a good move, Monte Carlo Search Tree will search upon it very quickly, it integrates breadth-first search and depth-first search, which is similar to a heuristic search. This partially solved the first problem.

8.4 No Validate Location Highlight

In accordance of the results from the functional testing and the feedback form these testers, the project is not a beginner friendly game. Even the game rules are simple, while an Othello beginner entered the game for first time, they still cannot find the proper locations to put piece. Mouse clicks on these unavailable positions will not give any response, so they have to randomly click on the board to set on piece. A function that is enable these validate locations to be highlighted must be added in the future work.

8.5 Adding the Game Rounds Count

Among the testing section, sometimes the number of game rounds was easily calculated by wrong, which will make the testing results be inaccurate. It will be important to add a game rounds countability function into the game. Basically, this function could be implemented like the piece number count function; and connected with the Winning Rate Chart GUI.

8.6 Applying Evaluation Table

Basically, evaluation is a critical process composing of several individual steps, which partially effect on the main evaluation (Schestowitz, 2003). So, the function of an accumulator is used to implement all of these steps into a final value. In this project, it will be important to create an evaluation table based on the game board. A value is assigned to each piece on the board in terms of its position. These values on the board are associated with a pre-defined integer and that can be greatly tuned among each run to optimise the evaluation's faithfulness. Thus, the real improvements on the algorithms will be generated.

Chapter 9

Conclusion and Future Work

In conclusion, this is a successful project. The primary objective of the project, developing the AI Othello by implementing the Minimax Search algorithm and Monte Carlo Tree Search Algorithm was achieved. AI players were generated out, that significantly performed on the game as opponent to against the human player and random player. Additionally, the secondary goal was going to figure out the performance difference between these two AI algorithms. In accordance of the feedback on testing results (Section 7.1.1 and 7.1.2), it could be observed that the Minimax with Alpha beta Pruning performed better than the Monte Carlo Tree Search. Therefore, this project could be viewed as succeeding in accomplishing the initial aims.

Though the project successfully meets these requirements and the original goals, there are still many improvements could be applied on both the AI algorithms and the graphical user interface. As demonstrated by the contents in evaluation (Section 8.4 and 8.5), from the view of the functional requirements, Validate Position Highlight and Game Rounds Countability, these two functions should be added in the future work, which can make game be more beginner friendly. Additionally, lack of the evaluation table also lost the chance to generate the real and excellent improvements on the algorithms. Because these assigned values on each grid of the board could be greatly tuned after each iteration to optimise the evaluation's faithfulness, that can make whole algorithm be more accurate and stable, working better on the AI player performance. This issue should also be added in the later game version.

After a number of game rounds, both algorithms revealed their own shortcomings even the Minimax with the Alpha beta pruning has the better performance than MCTS. One conclusion being observed from the Minimax is while the game board has been rescaled into a bigger size. If the search depth was still kept in same value as before, the power of Minimax would be greatly wakened. The MCTS could easily defeated it. However, if the search depth increased, it took Minimax more time to search a local optimal position. At this moment, the application of the Alpha beta pruning would be very important, which could decrease the search redundancy. So, in some real problems, they generally could not construct a complete pattern tree, so you need to determine a maximum depth D, and calculate the D layer down from the current pattern at most. For the above reasons, Minimax generally seeks a local optimal solution rather than a global optimal solution. The larger the search depth, the more likely it is to find a better solution, but the computational time will exponentially expand. This is the advantage and also the disadvantage of the Minimax. It does not find the theoretical optimal solution, because the theoretical optimal solution often depends on whether the opponent is stupid enough. In Minimax, we fully take the initiative. If the decision of each step is perfect, then we can achieve the expected minimum loss pattern. If the opponent does not make the perfect decision, then we may achieve a better ending than the worst situation expected. In short, we are going to choose the best in the worst case. But when the AI Alpha beta player battled with the random player. The winning rate greatly decreased. The random player just like a beginner without any game strategies. It randomly put piece on these valid positions making the AI be unstable.

Comparing with the above issue, Monte Carlo Tree Search performed better in this situation. MCTS does not require any reasonable strategy or specific practical knowledge to make move decisions. This algorithm can work effectively without any knowledge of the game beyond the basic rules; this means that a simple MCTS implementation can be reused in many board games with only minor adjustments, it has extreme adaptability, so this also makes MCTS is a great way to play general games. It can perform an asymmetric tree to adapt the growth of the search space topology. This algorithm will visit more interesting nodes more frequently and focus on its search time in a more relevant tree part. This makes MCTS be more suitable for games with larger scale game, such as 19 by 19 Go. Such a large combination of spaces can cause problems to search methods with standard depth or width based, so the adaptability of the MCTS indicates that it can find more optimized actions and focus the search on these parts. This is also the factor why MCTS was defeated by the Minimax algorithm in this Othello game, because MCTS algorithm, according to its basic structure, does not find the best action in an affordable time in some small-scale games. This is basically due to the full size of the space in the combined step, and the critical nodes are not able to access enough times to give a reasonable estimate.

To summarise, it is believed that in this Othello game the Minimax with Alpha-beta pruning has better performance. As the game be more complex, and the development of the artificial intelligence, the machine learning could be imported to improve AI player's ability via the data training.

Bibliography

[1] Andrey Kurenkov's Web World. (2019). *A 'Brief' History of Game AI Up To AlphaGo*. [online] Available at: https://www.andreykurenkov.com/writing/ai/a-brief-history-of-game-ai/ [Accessed 31 Aug. 2019].

[2] Greenemeier, L. (2019). *20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess*. [online] Scientific American. Available at: https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess/ [Accessed 1 Sep. 2019].

[3] Chowdhury, A. (2019). *What are the games that have successfully leveraged AI so far?*. [online] Analytics India Magazine. Available at: https://www.analyticsindiamag.com/10-games-successfully-integrated-artificial-intelligence/ [Accessed 3 Sep. 2019].

[4] Games, W. and Bhargava, R. (2019). *What is the Best Programming Language for Games | Freelancer Blog*. [online] Freelancer.hk. Available at: https://www.freelancer.hk/community/articles/what-is-the-best-programming-language-for-games [Accessed 4 Sep. 2019].

[5] Mastersofgames.com. (2019). *Rules and Instructions for Reversi and Othello*. [online] Available at: https://www.mastersofgames.com/rules/reversi-othello-rules.htm [Accessed 8 Sep. 2019].

[6] Softwaretestinghelp.com. (2019). *Types of Software Testing: Different Testing Types with Details*. [online] Available at: https://www.softwaretestinghelp.com/types-of-software-testing/ [Accessed 8 Sep. 2019].

[7] Zafar, R. (2010). *What is software testing? What are the different types of testing?*. [online] Codeproject.com. Available at: https://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty [Accessed 8 Sep. 2019].

[8] Browne C B, Powley E, Whitehouse D, et al. A Survey of Monte Carlo Tree Search Methods[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 2012, 4:1(1):1-43.

[9] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” Mach. Learn., vol. 47, no. 2, pp. 235–256, 2002.

[10] Engel, K., 2015. *Learning a Reversi Board Evaluator with Minimax,* s.l.: University of Maryland, College Park.

[11] Nilsson, N. J., 1998. *Artificial Intelligence: A New Synthesis.* s.l.:Cambridge University Press.

[12] Schestowitz, R., 2003. *Final Year Project - Othello Master,* s.l.: s.n.

Appendix

1. Guidance on Software Running

- Download the whole project as a zip file from the Git project repository.
- Unzip the package.
- Importing the project into any suitable IDE (e.g Eclipse, IntelliJ).
- Finding the main method in the Main class and run it.
- An empty game board will be displayed on your screen.
- Firstly, you should click on the “Setting” menu button to choose a game board as you prefer.
- If you choose these “Human Player V.S AI Player” modes, you just need to click the “Start Game” option under the “Game” menu; and start the game round with AI player. Until game finish, the board on right hand side will tell you who is the winner.
- If you choose these “AI Player V.S AI Player” modes, you just need to click the “Start Game” option under the “Game” menu; and to view how the game round is going.

2. Git Project Repository Address

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2018/zxw882

3. Implementation Code

```
/**
 * to display the piece in the game board cell
 * bond the centre of each piece at the centre of each cell
 * @param object
 */
private void displayPiece(Object object) {
    PieceType[][] board = (PieceType[][]) object;

    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            PieceIcon pieceIcon = boardIcon[i][j];
            pieceIcon.getChildren().clear();
            Color color;

            if (board[i][j] == null) {
                continue;
            }
            if (board[i][j].equals(PieceType.BLACK)) {
                color = Color.BLACK;
            } else {
                color = Color.WHITE;
            }
            Circle circle = new Circle(20, color);
            circle.setCenterX(pieceIcon.getWidth() / 2);
            circle.setCenterY(pieceIcon.getHeight() / 2);
            pieceIcon.getChildren().add(circle);
        }
    }
}
```

Figure 18 Piece Display Function

```java
    /**
     * set the piece and reverse the piece
     * @param board
     * @param r
     * @param c
     * @param pieceType
     */
    public void setPiece (PieceType[][]board,int r,int c,PieceType pieceType) {
        int[][] dirs= {{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1},{-1,0},{-1,1}};
        //using the conditional operator to set the piece on board, and change the piece colour
        PieceType opType=(pieceType==PieceType.BLACK ? PieceType.WHITE : PieceType.BLACK);

        for (int i = 0; i < dirs.length; i++) {
            for (int j = 1; j < BOARDSIZE; j++) {
                int newr = dirs[i][0]*j+r;
                int newc = dirs[i][1]*j+c;
                if (validateRC(newr, newc)) {
                    if (board[newr][newc] == opType) {
                        continue;
                    }else {
                        if (board[newr][newc]==pieceType && j>1) {
                            for (int k = 1; k < j; k++) {//reverse the piece
                                newr = dirs[i][0]*k+r;
                                newc = dirs[i][1]*k+c;
                                board[newr][newc]=pieceType;
                            }
                            break;
                        }else {
                            break;
                        }
                    }
                }else {
                    break;
                }
            }
        }
        board[r][c]=pieceType;
    }
```

Figure 19 Piece Set Function

```java
/**
 * the method is used to prove if the position(r,c) can display piece
 * @param r
 * @param c
 * @param pieceType
 * @param board
 * @return
 */
private boolean validateLocation(int r,int c,PieceType pieceType,PieceType[][] board) {
    if (board[r][c]!=null) {
        return false;
    }
    int[][]dirs= {{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1},{-1,0},{-1,1}};//8 directions for each piece step
    PieceType opType = (pieceType == PieceType.BLACK ? PieceType.WHITE : PieceType.BLACK);
    for (int i = 0; i < dirs.length; i++) {
        for (int j = 1; j < BOARDSIZE; j++) {
            int newr = dirs[i][0]*j+r;
            int newc = dirs[i][1]*j+c;

            if (validateRC(newr, newc)) {
                if (board[newr][newc]==opType) {
                    continue;
                }else {
                    if (board[newr][newc]==pieceType&&j>1) {
                        return true;
                    }else {
                        break;
                    }
                }
            }else {
                break;
            }
        }
    }
    return false;
}
```

Figure 20 Validate Position Function

```
/**
 * the method is used to change the player
 */
private void nextPlayer() {
    if (currentPlayer==players[0]) {
        currentPlayer=players[1];
    }else {
        currentPlayer=players[0];
    }
    if (currentPlayer.getPieceType()==PieceType.BLACK) {
        circlePiece.setFill(Color.BLACK);
    }else {
        circlePiece.setFill(Color.WHITE);
    }
}
```

Figure 21 Player Change Function

```
/**
 * the method is used to start the game by clicking the game start button
 * @param actionEvent
 */
public void itemStartClick(ActionEvent actionEvent) {
    game.start();
}

/**
 * the method is used to show the bar chart when click on the chart button
 *
 * @param actionEvent
 */
public void itemChartClick(ActionEvent actionEvent) {
    Stage stage = new Stage();
    FXMLLoader fxmlLoader = new FXMLLoader();
    String viewerFxml = "ChartViewer.fxml";
    Parent page = null;
    ChartController chartController = null;
    try {
        page = fxmlLoader.load(this.getClass().getResource(viewerFxml).openStream());
        chartController = (ChartController) fxmlLoader.getController();
        chartController.setDatas(game.getPlayers());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    Scene scene = new Scene(page);
    stage.setScene(scene);
    stage.setTitle("Winning Rate Chart");
    stage.setResizable(false);
    stage.show();
}
```

Figure 22 Item Click Function

```
        /**
         * the method is used to get player's piece count number
         * @param board
         * @param pieceType
         * @return
         */
        public int getScore(PieceType[][] board,PieceType pieceType) {
            int score=0;
            for (int i = 0; i < BOARDSIZE; i++) {
                for (int j = 0; j < BOARDSIZE; j++) {
                    if (board[i][j]==pieceType) {
                        score++;
                    }

                }
            }
            return score;
        }
```

Figure 23 Score Count Function

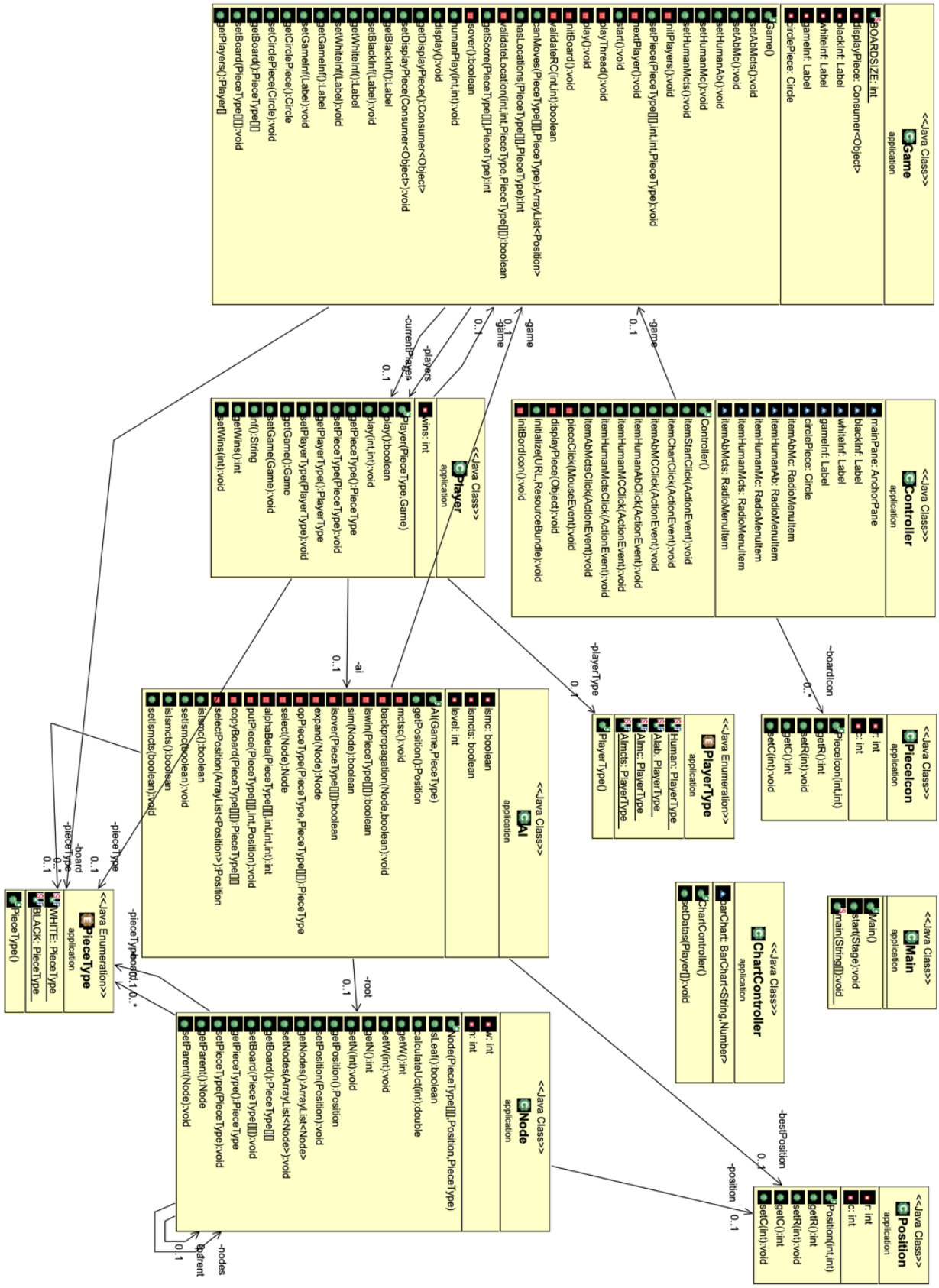
```
/**
 * the AI with alpha-beta algorithm
 * @param board
 * @param depth
 * @param alpha
 * @param beta
 * @return
 */
private int alphaBeta(PieceType[][] board, int depth, int alpha, int beta) {
    if (depth == 0) {
        if (level%2 == 1) {
            return -game.getScore(board, pieceType);
        }
        return game.getScore(board, pieceType);
    }

    ArrayList<Position> positions = game.canMoves(board, pieceType);
    if (positions.isEmpty()) {
        int dif = Math.abs(depth-level);
        if (dif % 2 == 1) {
            return -game.getScore(board, pieceType);
        }
        return game.getScore(board, pieceType);
    }

    while (!positions.isEmpty()) {
        Position position = selectPosition(positions);
        PieceType[][]copy = copyBoard(board);
        putPiece(copy, depth, position);
        int val = -alphaBeta(copy, depth-1, -beta, -alpha);
        //the value of the each will be changed after each recursion

        if (val>=beta) {
            return beta;
        }
        if (val>alpha) {
            alpha=val;
            if (depth==level) {
                bestPosition=position;
            }
        }
    }
    return alpha;
}
```

Figure 24 Alpha-Beta Search Algorithm

```java
/**
 * here is the Monte-Carlo method section, which use the monte carlo function in the Random method
 * @param positions
 * @return
 */
private Position selectPosition(ArrayList<Position> positions) {
    if (ismc) {
        Random random = new Random();//new an object of the Random method
        return positions.remove(random.nextInt(positions.size()));
    }else {
        return positions.remove(0);
    }
}
```

Figure 25 Random Player Function

4. UML Graph


<<Java Class>>
Game
application
BOARDSIZE: int
displayPiece: Consumer<Object>
blackInf: Label
whiteInf: Label
gameInf: Label
circlePiece: Circle
Game()
setAbMcts():void
setAbMc():void
setHumanAb():void
setHumanMc():void
setHumanMcts():void
initPlayers():void
setPiece(PieceType[][],int,int,PieceType):void
nextPlayer():void
start():void
playThread():void
play():void
initBoard():void
validateRC(int,int):boolean
canMoves(PieceType[][],PieceType):ArrayList<Position>
hasLocations(PieceType[][],PieceType):int
validateLocation(int,int,PieceType,PieceType[][]):boolean
getScore(PieceType[][],PieceType):int
isover():boolean
humanPlay(int,int):void
display():void
getDisplayPiece():Consumer<Object>
setDisplayPiece(Consumer<Object>):void
getBlackInf():Label
setBlackInf(Label):void
getWhiteInf():Label
setWhiteInf(Label):void
getGameInf():Label
setGameInf(Label):void
getCirclePiece():Circle
setCirclePiece(Circle):void
getBoard():PieceType[][]
setBoard(PieceType[][]):void
getPlayers():Player[]
<<Java Class>>
Controller
application
mainPane: AnchorPane
blackInf: Label
whiteInf: Label
gameInf: Label
circlePiece: Circle
itemAbMc: RadioMenuItem
itemHumanAb: RadioMenuItem
itemHumanMc: RadioMenuItem
itemHumanMcts: RadioMenuItem
itemAbMcts: RadioMenuItem
Controller()
itemStartClick(ActionEvent):void
itemChartClick(ActionEvent):void
itemAbMCClick(ActionEvent):void
itemHumanAbClick(ActionEvent):void
itemHumanMCClick(ActionEvent):void
itemHumanMctsClick(ActionEvent):void
itemAbMctsClick(ActionEvent):void
pieceClick(MouseEvent):void
displayPiece(Object):void
initialize(URL,ResourceBundle):void
initBordIcon():void
<<Java Class>>
Player
application
wins: int
Player(PieceType,Game)
play():boolean
play(int,int):void
getPieceType():PieceType
setPieceType(PieceType):void
getPlayerType():PlayerType
setPlayerType(PlayerType):void
getGame():Game
setGame(Game):void
getWins():int
setWins(int):void
<<Java Class>>
PieceIcon
application
i: int
i: int
PieceIcon(int,int)
getR():int
setR(int):void
getC():int
setC(int):void
<<Java Enumeration>>
PlayerType
application
Human: PlayerType
Alab: PlayerType
Almc: PlayerType
Almcts: PlayerType
PlayerType()
<<Java Class>>
Main
application
Main()
start(Stage):void
main(String[]):void
<<Java Class>>
ChartController
application
barChart: BarChart<String,Number>
ChartController()
setDatas(Player[]):void
<<Java Class>>
AI
application
ismc: boolean
ismcts: boolean
level: int
AI(Game,PieceType)
getPosition():Position
mcts():void
backpropagation(Node,boolean):void
iswin(PieceType[][]):boolean
sim(Node):boolean
isover(PieceType[][]):boolean
expand(Node):Node
opPieceType(PieceType,PieceType[][]):PieceType
select(Node):Node
alphaBeta(PieceType[][],int,int,int):int
putPiece(PieceType[][],int,Position):void
copyBoard(PieceType[][]):PieceType[][]
selectPosition(ArrayList<Position>):Position
ismc():boolean
setismc(boolean):void
isIsmcts():boolean
setIsmcts(boolean):void
<<Java Class>>
Node
application
w: int
n: int
Node(PieceType[][],Position,PieceType)
isLeaf():boolean
calculateUct(int):double
getW():int
setW(int):void
getN():int
setN(int):void
getPosition():Position
setPosition(Position):void
getNodes():ArrayList<Node>
setNodes(ArrayList<Node>):void
getBoard():PieceType[][]
setBoard(PieceType[][]):void
getPieceType():PieceType
setPieceType(PieceType):void
getParent():Node
setParent(Node):void
<<Java Class>>
Position
application
i: int
i: int
Position(int,int)
getR():int
setR(int):void
getC():int
setC(int):void
<<Java Enumeration>>
PieceType
application
WHITE: PieceType
BLACK: PieceType
PieceType()
-game
-players
-currentPlayer
-playerType
-boardIcon
-ai
-pieceType
-board
-root
-bestPosition
-position
-nodes
-parent