School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Exploring Artificial Intelligence within a Card and Board Game: Samurai

Yi Zhou

Submitted in accordance with the requirements for the degree of
MSc Computer Science

2018/2019

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Project Report	*PDF*	*Minerva (02/09/19)*
Project Report	*Physical Copy (x2)*	*SSO (02/09/19)*
Code	*GitLab Repository*	*Supervisor, assessor (02/09/19)*
Participant Consent Forms	*Signed forms in envelope*	*SSO (02/09/19)*

Type of Project: _____Exploratory Software_____

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

This project exploratory created an artificial player which had capable of defeating a human player in a rare researched board game called *Samurai*. A simple digital version of *Samurai* was implemented to allow different algorithms to compete and to be compared, and ultimately selecting an optimal artificial player strategy that had a satisfactory possibility to surpass a human player.

Acknowledgements

First of all, I would like to thank Dr.Brandon for his selfless guidance during my project. Secondly, I would like to thank my assessor Dr.Isolde for giving me suggestions in the midterm meeting, which pointed me a clear direction of later implementation.

Table of Contents

Chapter 1
Introduction

It can be said that game is one of the most widely applied areas of *Artificial Intelligence*(AI) , especially in the game of strategy confrontation. The confrontation between AI and players has always been the main research technology in the field of game[1]. Since professor Shannon proposed to write programs for *Chess* in 1950, Game *Artificial Intelligence* has been the forefront of *Artificial Intelligence* research technology, known as the "Drosophila" in the field of *Artificial Intelligence*[2]. Game AI promote the development of *Artificial Intelligence* technology and has a profound impact on society. In 2017, Google's *Artificial Intelligence* agent — *AlphaGo* beat human champion of Go — Lee Sedol, which has shaken the world and has led to a new round of AI fever around the world[3]. More people has started to develop and create a powerful AI to fight or even defeat humans in various types of Game.

There has been numerous research has conducted on AI programs that play adversarial games using search-intensive AI techniques. Mature and completed AI agents has been created from deeply researching for straightforward board games, such as *Chess* and *Checker.* However, there still has some “non-solved” board games which has more competitive rules and strategy to explore a effective AI agent.

Samurai is one of that “non-solved” board games which has high complexity and flexibility in adversarial games. It is very challenging and worthwhile to explore an AI agent for *Samurai* , no matter to test current AI techniques or to improve them.

1.1 The problem

The long-term exploration of AI for board game is ongoing, but most studies are more enthusiastic about classic board games[4], e.g. *Chess* and Go.There are still a large number of relatively complex board games that still need to be explored, and *Samurai* is one of them. This scope of this project is to reconstructs the 1988 strategy board game *Samurai* digitally, and develops an AI system that can mimic humans and surpass humans. Although rules of *Samurai* seem to be uncomplicated for humans, it still has more uncertainty and randomness, compared to traditional board games. These attributes make *Samurai* a particularly interesting strategy game to build an artificial player that can compete with humans and test the feasibility of various types of *Artificial Intelligence* technology.

On the other hand, although there are some digital versions of *Samurai*, the successful *Artificial Intelligence* agents in these versions are still relatively lacking. According to background research, it is difficult to find a “Solved” *Artificial Intelligence* cases and materials related to *Samurai*, so It is valuable and necessary to study the feasibility of applying some *Artificial Intelligence* technology to such a complex strategy game.

1.2 Project aim

The aim of this project is basically to construct a digital representation of the board game *Samurai* and explore a effective artificial player which can compete or even surpass the human players.

During this project, different algorithms will be tested and investigated to continually improved the performance of the agent, and attempt to apply and evaluate methods of constructing advanced artificial players in well known board games.

1.3 Objectives

Five objectives are established:

- Produce a playable programme of the board and card game *Samurai.*
- Develop an AI algorithm for playing *Samurai.*
- Compare different algorithms to find the best solution for ‘*Samurai* AI’.
- Test the programme and evaluate the performance of AI against human players.

1.4 Methodology

In this project, I shall be developing an exploratory software. It is not a business standard, it is just a research tool, so it will not do too much work on the user interface and user experience. Although the program may not be available to others as a valid program, it will fully perform the task of this course design. Due to the large unknowns and uncertainties about such projects, I choose the iterative development method during the development process, which is a method of designing and implementing a part of this product at a time and gradually completing it. Each phase of design and implementation is called an iteration.

Using an iterative approach allows development work to be initiated before the requirements are fully clarified. All methods and techniques may not be complete in the early stage of the project. Therefore, iterative method is needed to perform a new round of iteration to refine

the target and optimize the program after searching literature and getting guidance from the instructor. Under the guarantee of the original software established at the beginning, the early test data can be obtained through continuous testing and integration to make timely judgments on the progress of the current development, and to summarize the success of each stage. In this way, we will decide on the development strategy and improvement direction of the next stage.

The schedule for this project begin form the end of April up to the end of August. The Gantt chart displayed below(*Figure1.1*) illustrates four main stages are split for the whole project. The key task and aim for the first stage is to implementing a playable game for *Samurai*, which include the background research and game programming. This stage is expected to be completed in approximately 4 weeks in total.

The second stage is for implementing AI agent for the programme of *Samurai*, which also including the background research and programming tasks. This stage is the key stage for this project, hence it will take up the most of time to finish, about 5 weeks.

The third stage is aim to realize different vision of AI agents which will have different performance with various algorithms, and then the performance will be tested as well in this stage. Therefore, it may takes 4 weeks in total.

The final stage contains testing task with human players, which is a key point to prove that the project is work and significative.

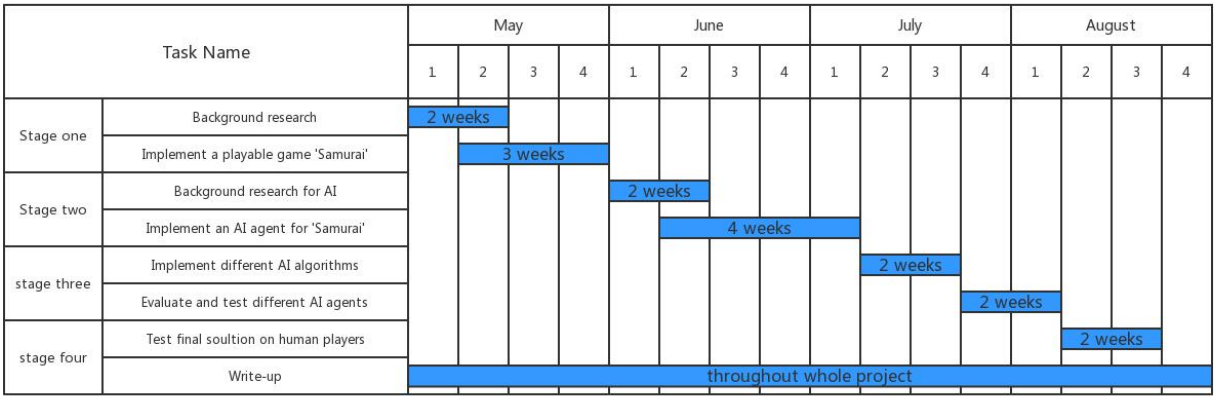

Figure 1.1 Project schedule

Chapter 2
Background Research

Before directly pushing the project, it is necessary to have a deep understanding of how to achieve the goals of the project, therefore I need to investigate and research related fields to expand my knowledge base. Although some methods and techniques may not be fully applied in the project, background research still can benefit to avoid aimless or redundant research. In addition, with the support of good background knowledge, it is of great help to accelerate development speed and enhance the understanding of the problem. Therefore, in this chapter, I will discuss some background information about *Game theory* and useful *Artificial Intelligence* techniques for games, and also will mention some understanding and discussion of the game rules of *Samurai*, which will help me achieve the goals of the project in the most sensible way.

2.1 Game theory

Game theory refers to the study of corresponding strategic decision making under specific conditions in the game where multiple individuals or teams involved consider heavily on the strategies made by other contributors or partners.It considers the predicted behavior and actual behavior of individuals and parties in games for studying optimization strategies, for example, Biologists use *Game theory* to understand and predict certain outcomes of evolution. As a branch of applied mathematics, *Game theory* is also widely being used in politics, economics, international relations, as well as in computer science and strategy games[5].

2.1.1 Terminology

So far I have not mention anything yet about the compositions of the game, and to figure these out I need to define some concepts about these components, which will also benefit to me describe them later. In addition, using defined terminology can disambiguate ambiguous terms, for example, the word "play" may represent an one-step operation of a player in everyday language, which may be confused with the word "move". Therefore it's important to define the terms of the components in the game for both the writers and the readers.

Terminology 1. Game

A *game* is described by a set of rules.

Terminology 2. Play

A *play* is a particular completed instance of a *game*, which strict follows the rules of the *game* until the *game* is end.

Terminology 3. State

A *state* within a *game* describes an temporary allocation and situation of all the existing components.

Terminology 4. Move

A *move* is a decision made according to a current *state,* considering multiple elements.

Terminology 5. Strategy

A *strategy* is a plan that lead the player to decide a *move.*

Terminology 6. Outcome

An *outcome* is the consequence which produced by a *move,* and lead to another particular *state*.

Terminology 7. payoff

A *payoff* is the reward produced by the *outcome*.

Terminology 8. Rational behavior

A *rational behavior* is a player has a *strategy* which tries to maximize his *payoff with* considerations of opponents' possible *strategies*.

2.1.2 Game Types

Game theory defines a range of methods to distinguish between different categories of games, which allows us to judge a type of game and find the appropriate strategy model to build. A reasonable model will simplify the difficulty of planning strategies and give direction and basis for choosing AI algorithms and technologies. So I will discuss some of the game categories that might be applicable to this project.

Zero-Sum and Non-Zero-Sum Games

Zero-sum game is a concept in *Game theory* means that under strict competition, the gains of one player must equal to the losses of the other players, and the sum of the gains and losses of all players in the game will always be "zero"[6]. For example, in a 2-player game, if player one gains the positive payoff with value 5, then player two will take the negative one with value -5(In the game more than 2 players, the lost players will split these negative

payoff. In such games, there is no possibility of cooperation, which means if there has one player get advantage in games while there must adversely have other players suffer from the disadvantage.

N-Player Games

The difference in players with in a game greatly affects the construction of game strategies[7]. For 1-player games, the player do not need to consider other obstacles of people and interference to them. Therefore, when the player plan a strategy, he merely need to choose the situation that is most beneficial to himself in the current *state* to maximize his own *payoff*.

However, when there is a second player to join the game, situation will become complex. First of all, players can not only consider to maximize their own *payoff*, but also may consider how to prevent opponents from harming them or consider how to hinder their opponents from easily gaining benefits. In this case, the strategy of players will become complicated and the factors to be considered will increase accordingly with difficulty of the game rules.

When the player is more than just 2 players(n>=3), the difficulty of developing a strategy will increase non-linearly according to the number of players. Meanwhile, players probably also need to consider who is the most tricky opponent in the current *state* and who will able to be a temporary ally of the players. Briefly, if there are more opponents standing on the opposite side of the player, the more struggled the player is to make a strategy, as well as to regard more factors.

Perfect and Imperfect Information Games

Perfect information games means that the players know all the current and previously occurring *state*s, although it is not possible to know what the player might to do for next *state*[8]. For example, *Chess* is a typical type of game with perfect information, in which both players can have complete awareness of every step of the occurred *state*.

In contrast, *imperfect information games* means that the players can not achieve full information about current or previous *state*. For example, *Poker* is a kind of *imperfect information game* cause the player cannot know hands of his opponents.

Cooperative and Non-Cooperative Games

Cooperative games refers to the game in which two or more players have the same interests and goals allying with each other to play against other alliances[9]. These alliances are certainly allowed by the rules of the game, such as signing contracts, and sometimes these contracts probably are enforced by the rules of the game. Members of the alliance will *play* a *game* for the same goal with different *strategies* and *payoffs*, but winners will receive the same *payoff*. Most of *cooperative games* are *played* with more than four players, but occasionally they can be *3-player games*(although this is unfair to one player). However, *cooperative games* can definitely not be in form of a one-player game cause it is pointless to give opponent a hand to win a game.

Although a player might be able to reap benefits quickly with the help of an ally, there also has unpredictable risk, such as the ally could betray at any time so that he can claim payoff alone (*payoff* will be double). However, there is no such issue in *non-cooperative games*, where each player has an individual *strategy* and goal.

Deterministic and Stochastic Games

Players can predict the outcome completely with each *state* within *Deterministic game,* cause the players can receives *perfect information*, which allows to predict any action his opponent will take with each *state*[10]. For example, In *Chess*, players can fully develop *strategies* through one *state* or a series of action from their opponents. Although, in reality humans can successfully search all the possible outcome for each *state* in naive games, such as *Tic Tac Toe*, the sequential search space complexity for games, like *Chess,* is impossible.

In comparison, *Stochastic Games* often involve random factors, which makes achieving perfect information is unimaginative for players. For example, *Poker*, which players can not have knowledge of the hand of his opponent, so he can not build a definite coping *strategy*. Hence when we dealing with these types of games, we have to include randomness into our *strategy*. However, not all the *imperfect information games* are *stochastic games*, some of them can still be judged by a *state*.

Sequential and Simultaneous Games

In *sequential games*, only one player can have a *state* or a series of actions per turn, and players turn will alternate on a sequential basis (e.g. *Monopoly*). In this form of games, the strategy of player is based on the previous *state* and the current *state* of opponent cause the player can access occurred information. Although this type of game certainly can be

imperfect information games, players can more or less judge the next *state strategy* based on what they can aware for now.

However, in *simultaneous games*, each player is able to do a *state* in the same round and these *state*s are concurrent (e.g. *Uno*). Therefore, in such games, it is difficult for players to make coping *strategies* based on the *state* of their opponents. It is obvious that such games belong to imperfect information games, so it is difficult to implement corresponding strategies when there is little information about opponents.

In addition, the representation forms of these two types of games in the digital version are also quite distinguishing. *Sequential games* are commonly represented by game trees, while *simultaneous games* are more likely constructed by *payoff* matrices.

2.1.3 Game Strategies

There are two main types of strategies: pure and *mixed strategy*[11]. A *pure strategy* is to select the pre-established strategy to act according to the achieved information, which means, in each *state*, the player will make a fixed response *state* based on the current information. More specific, a *pure strategy* is a group of strategies that have been formulated, and player will select the one of the strategies from the set that have been designed according to different situations. When the situation is suitable for a particular strategy, then only that strategy can be selected with a possible value of 1, which means that other strategy will impossible be implemented.

A *mixed strategy* has randomness. Unlike a *pure strategy*, which will use a fixed strategy to deal with one or several *states*, a *mixed strategy* will determine the probability of each strategy implementation according to the current *state*. The total probability of each strategy will be equal to 1(e.g. strategy1 =0.3, strategy2= 0.5 and strategy3 =0.2), then player can randomly pick one of these different strategies based on probability.

In addition, a *pure strategy* can be understood as a special case of a *mixed strategy*, where one strategy has a probability of 1 and the other strategies have a probability of 0(although this is the rare case).

According to the process mode of the two strategies, it is obvious *mixed strategy* is more effective and reasonable. In the case of *Rock-Paper-Scissors*, when an opponent chooses a *Rock* on the previous round, the *pure strategy* will pick the strategy to beat *Rock* from the strategy group (select the *Paper*). This makes it easy for the opponent to predict what the next *state* of the player is going to be, which would be very silly. Therefore, if a *mixed strategy* is used for improving this, though the chances of a *Paper state* will be greater than *Rock* and *Scissors*, the other two are still possible to be played, which makes it difficult for the opponent to predict next *state* of player.

2.1.4 Game Representations

As mentioned, a game is composed of different elements, such as *states* and *payoffs*, so it is important to find some reasonable and efficient models to represent these elements in a game. I will discuss some common *game representations* according to different types of games.

Normal Form

When a game is desired to be represented, a reasonable representation model normally will be chosen by the type of the game. The most common method to determine the appropriate model for a game is by judging whether it is a *simultaneous game* or a *sequential game*. If it is the former, then *Normal Form* will be the best choice. *Normal Form* will be represented in an N-dimensional form (N = the number of players), containing every *state* that each player can take and corresponding *payoffs* produced by *state*s of other players, so it will be combinations of all *state*s of players. For example, the "(1,1)" in the first upper left grid represents that when player 1 chooses the first *state* and player 2 chooses the first *state* at the same time, player 1 will get a *payoff* with value 1 and player 2 will get a *payoff with* value 1 as well(*Figure2.1*).

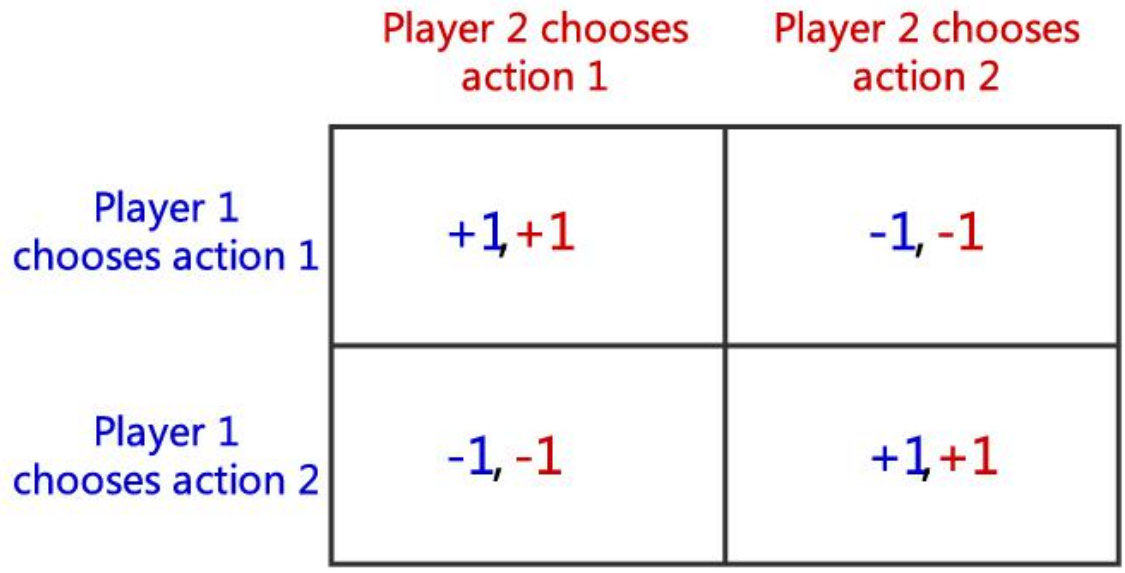

Figure 2.1 An example of Normal Form in a 2-player game

Extensive Form

For *sequential games*, a *state* or a series of *state*s is executed in succession, so *Extensive Form* is used instead to represent. A game tree is the best representative of *Extensive Form*, which uses nodes and branches on behalf of a entire game with all possible situations. In a game tree, each node denotes a specific *state* and a branch represents a possible game development trend. More specific, a game tree has an initial node, also known as the root of the tree, which not only can be the start of the game but also can be the current *state* of the ongoing game. Nodes in each layer are only connected to nodes in the upper and lower layers, and the last layer of the entire tree is usually called leaves, which also represents the end of the game or the end of the prediction. The values of such leaves often contains *payoffs* that can be achieved by a player according to the root combining with a series of *state*s among all players.

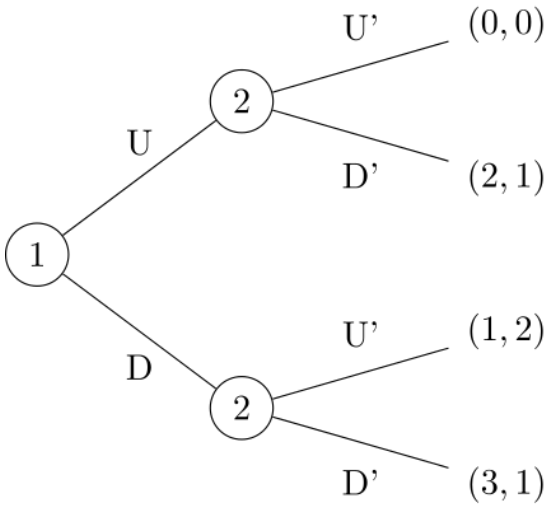

Figure 2.2 An example of extensive form

2.2 Artificial Intelligence Techniques

In order to implement an *Artificial Intelligence* for a game, searching and understanding of the existing *Artificial Intelligence* techniques is necessary. Since there are numerous AI techniques suitable for different types of areas, filtering some of the technologies and focus on the ones that are more specific used in games, is deserved and meaningful.

2.2.1 Heuristic state Evaluation

Heuristic state Evaluation are normally used to find an approximations solution of a problem that may not be optimal and perfect[12]. When some complex problems are faced, it might be extremely time wasted to search the optimal solution for one *state* by scanning all the schemes, while the heuristic algorithm can achieve a value approximating to the optimal solution in the case of time limitation. In order to obtain this approximate optimal solution, heuristic algorithm will evaluate each possible move in the current *state* by using different game elements, so that there is no need to conduct a complete search for the complete game process.

There is a good example to explain why to use *Heuristic state Evaluation* instead of full searching. In game *Go*, to obtain the optimal *state* for the current *state*, entire possible development directions of the game should be searched via Game Three until the end of the game. Although this is possible for computers dealing with numerous calculation, it is still cost massive time. In the reality rules of *Go*, the consideration time is limited, so the heuristic algorithm take its advantage and even much better when in the case of focusing more on efficiency rather than the accuracy.

Generally, the accuracy of the heuristic algorithm is associated with the *state* evaluation equation which is normally consisted by infinite figure functions. These functions denote the gains(which can be positive as well as negative) on different game elements according to each move the player can play under current *state*. The general form of heuristic algorithm is as follows:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

Eval(s) represents the heuristic values obtained by a specific move under the current *state*. Every possible move will have a individual heuristic value, and these values are defined by different figure functions(f_1, f_2, ... f_n) which will have their own weight(w_1, w_2, ... w_3). For example, in classic game *Chess,* a figure function can be designed like follows*:*

$$f_1(s) = (number\ of\ white\ queens) - (number\ of\ black\ queens)$$

Moreover, weights will determine which game elements will have a more important impact on the overall heuristic. In exploring a suitable heuristic for a game, we will not only design different elements for consideration, but also test the performance of heuristic functions by adjusting weights.

By calculating the heuristic value of each move under the current *state*, an approximate optimal solution under the current *state* can be obtained. The closer that solution is to the optimal, the more figure functions and time the heuristic algorithm will cost. Therefore, when

designing a heuristic algorithm, not only the trade-off of the weights between different figure functions need to be regarded, but also the trade-off between accuracy and time need to be took into consideration.

2.2.2 Minimax Algorithm

Regardless of the game type, most of basic board games contains similar essence, such as Go and *Chess*. So there is a common algorithm for compiling any human-computer game, and *Minimax Algorithm* is the most common and well-known one.

Minimax Algorithm is a pessimistic algorithm that assumes that each move of the opponent will choose the move that is most able to prevent the player from winning under the current *state*, while player will search for the best move under the worst *state* created by the opponent[13]. Briefly, the opponent has perfect decision-making ability, every move he made is the best move, and the algorithm will allow us to minimize our losses in this case.

Minimax Algorithm is often used for games that two players alternate make their moves (e.g. *Chess* and *Tic Tac Toe*), which also be classified as *sequential games*. Following picture is a example of *Minimax Algorithm* for *Tic Tac Toe.*

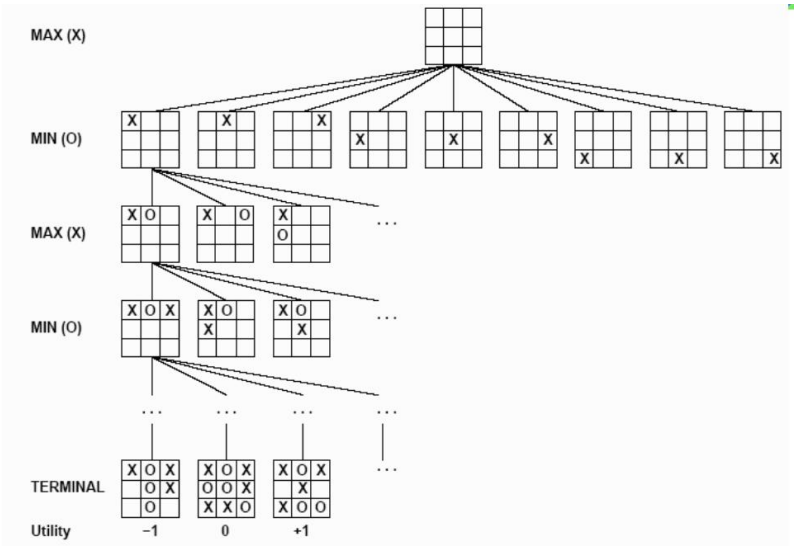

Figure 2.3 Minimax Algorithm for Tic Tac Toe

Minimax algorithm is the basis algorithm of the games which strategy based on searching the possible move. The solution of *Minimax* usually may not be the theoretical optimal solution, because the *Minimax algorithm* assumes that opponent can select the best move at his round. In fact, the opponent does not exactly behave like that in most of the case, therefore the player can always fully control the initiative. To be more specific,if the opponent

chooses the most perfect move as assumed in each *state*, the player can still achieve the predicted minimum loss outcome. However, if the opponent miss the best move, the player can achieve a better outcome than the worst-case scenario predicted. The general implementation of Minimax Algorithm is as follow:

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] — MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return  UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Figure 2.4 Pseudo-code for general Minimax Algorithm

2.2.4 Expectiminimax

In the *stochastic games*, *Minimax algorithm* is difficult to be implemented, because there are many uncertain game factors leading to the inability to predict *moves* of other opponents, such as the hands of opponents in the card game. Therefore, *Expectiminimax algorithm* can be used to add a probability to each *move*[14]. For example, if the hand of opponents are drawn at random from 20 cards, the draw probability of each card can be assumed as 1/20, and that probability will increase if there are same card in 20 cards. When an *move* of an opponent are being predicted, probabilities of each card that is being hold in the opponent hand will be taken into account. That is to say, *Expectiminimax algorithm* is roughly the same as Minimax algorithm, which alternate Min and Max approach to get the best value for current *root*, but the difference is that each node in Min and Max *state* is added a probability value to the original value corresponding to stochastic factors in the game.

2.2.5 Alph-beta pruning

For games with larger board and complex rules, a game tree using minimax algorithm will be extremely large, which means calculate all the nodes is very expensive. Therefore, Alph-

beta pruning can be used as an algorithm for decreasing branches of a game tree[13]. The time complexity of a game tree can be represented as $O(b_m)$, where b represents the average number of branches on each level, and m represents the maximum depth of the tree. This complexity can be maximum reduced to $O(b_m / 2)$ by using alpha-beta pruning [15].

2.3 Samurai

As mentioned before, some features of *Samurai* led me to choose it as the research object, such as game complexity. Before actually implementing a digital version of *Samurai*, I have to study the details of *Samurai*, understand its rules and game components.

2.3.1 Setup

The main loop of *Samurai* is the same as regular board games in which two or more players alternately drop pieces on a given board.

Board Setup

However, the board of *Samurai* adopts a hexagonal grid board(Figure2.5), which is pretty different from the grid board normally used by most board games. Meanwhile, *Samurai* holds different sizes of boards depending on the number of players(maximum 4). The smallest board can be played by two players, while the largest board can be played by a total of four players.

Each hexagon on the board is called a tile, which is the main place for players to set their pieces. Unlike the traditional *Chess*, in which every position is equal, tiles in board of *Samurai* are divided into three categories:

- Land Tiles(e.g. the blue box in *Figure2.5*).
- Sea Tiles(e.g. the green box in *Figure2.5*).
- Figure Tiles(e.g. the red box in *Figure2.5*).

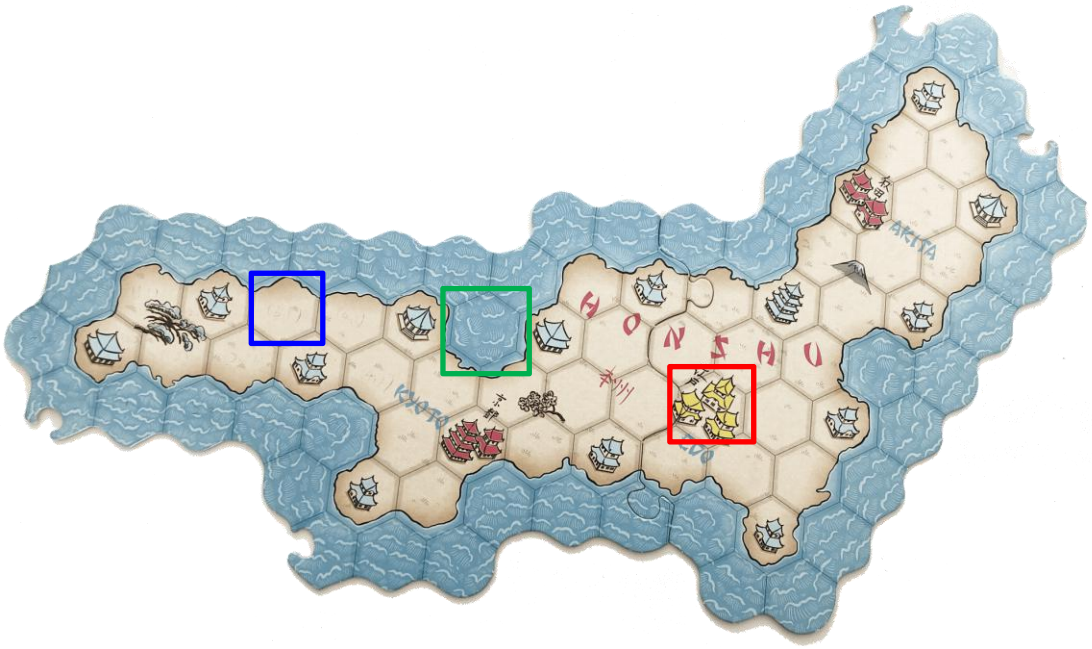

Figure 2.5 An example of 2-player board of Samurai, and the blue box in the picture shows a land tiles; the red box shows a figure tile; the green box shows a sea tile.

Among these three types,land tiles are the main territory that players fight for, which is the place where any piece can be placed except ship pieces, while sea tiles are the place specially for ship pieces. The remain one is the most special one which is the scoring point of the *Samurai*. Each figure tile has its own type and score, and can be roughly divided into three types: 1-figure type, 2-figure type and 3-figure type. The functions of these different figure tiles will be detailed in the rules of the game. So far, it is only necessary to know that they are a special scoring point and cannot be placed by any piece.

Pieces Setup

In *Samurai*, each player has a total of 20 different pieces, which make up the a individual piece deck of a player. Players are not allowed to place any piece in the deck, and are restricted to picking the pieces they want to place from the hand. The hand of each player will hold 5 pieces, which are randomly selected from the their own deck. Although each player has a separate set of pieces, the 20 pieces in their individual deck is extremely the same. There have totally 20 types of piece in *Samurai,* which are as follows:

- **1x** 2-point Buddha
- **1x** 3-point Buddha
- **1x** 4-point Buddha
- **1x** 2-point Helmet

- **1x** 3-point Helmet
- **1x** 4-point Helmet
- **1x** 2-point Rice
- **1x** 3-point Rice
- **1x** 4-point Rice
- **2x** 1-point Samurai
- **2x** 2-point Samurai
- **1x** 3-point Samurai
- **2x** 1-point Ship
- **1x** 2-point Ship
- **1x** 1-point Ronin
- **1x** Figure Exchange
- **1x** Piece Exchange

The pieces have different function to capture the specific figure tile with particular points. For example, 3-Buddha, it can capture the figure tile which contains a Buddha figure with 3 points(detail rules for capturing and detail function of each piece will discuss in 2.3.3 Scoring and Capturing).

Figure 2.6 A entire set of pieces for each player.

Figures Setup

Each figure tile in Samurai is special, because 1-3 figures are placed on each figure tile, which the number of figures is determined according to the number of buildings displayed in different figure tiles on the board. For example, in Figure 2.5, the figure tile in the red box contains three figures, and these figures are also points to be scored in the game. In Samurai, the main purpose of the player is to fight for these different figures(the method of capturing and the conditions of winning will be described later). Surely, figure also contains different types, dividing into three classifications: Rice, Helmet and Buddha, and each figure exist on the board with the same numbers at the beginning(e.g. 7 for 2-player game)

Meanwhile, *Samurai* stipulates that these figures tiles, which can be placed in multiple figures, can not contain the same kind of figures. For example, the figure tile(the red box in *Figure 2.5*) just can be placed with 1 Rice, 1 Helmet and 1 Buddha(*Figure 2.7*).

Figure 2.7 A 3-buliding figure tile with three different figures

2.3.2 Rules

At the beginning of the game, each player will randomly draw five pieces from their deck, and they will not be able to know their remaining pieces in the deck and hands and deck of their opponents . Players take turns placing their own pieces on the board, but players can only choose playable empty tiles as their decision. When a player finishes his turn, that player needs to draw some pieces from his deck to make sure he has five pieces in his hand. However, if the player does not have enough pieces to fill his hand, he will keep the current pieces.

In placing process, it is stipulated that Ship pieces can only be placed in sea tiles and other pieces can only be placed in land tiles, and figure tiles as special tiles for scoring can not be set in any pieces. In addition, there are certain rules on the type and number of pieces players can play in each turn. First of all, apart from the different functions of the 20 pieces, they can also be divided into two categories: one is called character pieces, which can be placed in any number in a round; The other is the ordinary pieces, which can merely be placed one in the same turn. Ship, Ronin, Token exchange and Piece exchange pieces are all belong to the former, while Rice, Helmet, Buddha and Samurai pieces are in the latter classification.

To sum up, all the players take turns placing pieces according to the above rules until either winning condition is met (will be described in winning *state*) or until the there is no figure to be captured.

2.3.3 Scoring and Capture

One important concept called “capture” in *Samurai* is that when the neighbor of the figure tiles is placed with the same type of piece as the figures in the figure tiles, then the figure tiles are captured by the player with a certain point. For example, when a 3-point Rice chess piece is placed on the neighbor of a figure tile containing a Rice figure, the Rice figure of the figure tile will be occupied by the player. And if there is another Buddha figure in the figure tile, the Rice piece does not have any effect on the capturing Buddha (player point in this figure tile: Rice = 3, Buddha = 0)

In *Samurai,* in order to win the game, player focus on getting as many figures as possible, but how to know a figure has been captured by a player? Remember we said that figure tiles are special scoring tiles, so when a figure tiles is “completely captured”, it means that one or more players can get “scoring”. “Completely captured” means when all the land tile neighbors of this figure tiles has been placed(Figure 2.8), and then captured points of each player on the current figure tile is calculated. The player who achieve the highest points will get the score of corresponding figures from the current figure tile, and If there are multiple figures in the figure tile, the scores will be calculated separately according to the types of each figure. However, if the players get the same point on a figure, that will be tied and no one can get score from that figure.

The score of a player can split into 3 part, one is for Rice score, one is for Helmet score, and the remain is for Buddha score. Therefore when a player get a figure , it will get the a corresponding score at one of the three, depending on the type of the captured figure.

Figure 2.8 A “Completed Captured” figure tile

2.3.4 Winning Conditions

As mentioned, in *Samurai*, each player gets one score for each captured figure. In the end, however, winner decision for *Samurai* was not to compare the total number of figures

players captured, but to compare who got the most overall scores which just has three points: one for Rice, one for Helmet and one for Buddha. For example,in a 2-player game, Rice, Helmet and Buddha figure separately have seven on the board at the start of the game. Player 1 will win a point in the overall score if he capture more Rice, helmet or Buddha figures than player 2. Finally, winner will be decided via comparing overall score of two players. However, rare cases will happen in which the overall score of two players is equivalent(e.g. Rice is won by player 1, Helmet by player 2, Buddha is not won by anyone). In this case, the total number of figures will be the additional factor to decide the winner.

2.3.5 Scope of Project

In order to ensure the successful completion of the project within the limited time and not to waste most of the time on game representation and game progress implementation, it is necessary to explain the main game mode to be implemented and simplify some complicated components, such as the number of players, the rules of game.

Samurai can have different number of players due to the different number of participants, the board will change accordingly. If all the boards are taken into consideration, the game presentation will take a unpredictable part of entire project, which will be meaningless and discursive. In addition, in the AI implementation stage, the AI agent will be confronted with multiple players, which will make the difficulty of AI implementation increasing rapidly with the people number increment. Therefore 2-player game is the best choice cause it is also as the same as other classic and fully studied board games.

Moreover, *Samurai* has different game modes. In different game modes, the initial *state* of the game and the conditions of final victory judgment will be changed. If all the game modes are taken into account, it will be difficult to complete in a limited time. On the other hand, the changes of these game modes will not be particularly big, so it is a good idea to select a popular game mode to explore. The selected game mode is called Domination mode, which has already been mainly described above.

2.3.6 Approach

During the process of implementing artificial player for *Samurai*, two stages are desired to be divided into: Using *Heuristic Algorithm* to build an artificial player; Using *Minimax Algorithm* to build an artificial player.

In the first stage, the *Heuristic state Evaluation* method will be used to calculate the best *move* that can be made in the current *state* by combining different game components. *Heuristic Algorithm* are often applied in *perfect information* games, where the previous *state*

of the game can be completed achieved. Although *Samurai* is defined as an *imperfect information* game due to the unaware hands of other players, the method to decide the best *move* by calculating the values of all possible *moves* within the current *state* can be still applicable to *Samurai*. Besides, by experimenting different heuristic *state* evaluation functions, such as modifying weight parameters and adding different game elements into the function, the performance of AI based on *Heuristic Algorithm* is worth expecting.

For the second stage, in order to further improve the performance of AI, I will probably combine *Minimax algorithm* with *Heuristic Algorithm*. Through background research, it can be known that *Minimax algorithm* will make some "predictions" of the game, which means the algorithm will conduct a n-play look-forward on the current *state.* To be specific, it will pick the best *move* at present *state* under the assumption that the opponent may make the most unfavorable *move* to player. In this case, the *Minimax algorithm* is more reasonable and logical to select instead of to use optimal heuristic algorithm with no prediction and judging by limited information.

In brief, different heuristics *state* evaluation functions will be compared in the initial stage, and then some of heuristic equations which has better performance, will be selected and applied to the *Minimax Algorithm* as well as exploring performance gaps between with and without *Minimax Algorithm*. In addition, the depth of look-forward will also be discussed by comparing the winning rate with the performance to select a appropriate depth limitation and apply it to the final AI agent solution.

2.4 Summary

Through the background research, *Samurai* can be classified as a competitive, sequential and stochastic game with imperfect information, hence *Minimax* and *Heuristic* Algorithms are generally suitable for implementing an artificial player.

- Zero-sum: Each figure is captured by a player is considered as a disadvantage to other players.
- 2-player: In this project, merely the 2-player situation will be considered.
- Competitive: Players compete to captured the limited figures on the board.
- Sequential: Players take turns setting pieces with visible previous *state.*
- Imperfect information: Hands of opponents is unaware for the other players within a particular *state*.
- Stochastic: Hands of each player is totally random,which can not be certainly predicted.

Chapter 3
Game Implementation

Before implementing an artificial player, It is necessary to initially build a digital version of *Samurai* as a testing platform for artificial players. This contains a suitable *Game design* for game components, which needs, for example, first of all, represent some game elements in the programme in a reasonable and effective method, such as the board, hands and pieces(mentioned in *section 2.3.1 Setup).* And then, integrating these elements and applying them to with the game rules and loop which mentioned in the *section 2.3.2*, *section 2.3.3* and *section 2.3.4* , which will ultimately form to a complete game process.

Besides, a suitable implementation language has to be selected to complete the programme, and the design of classes is strictly related to the picked language. Therefore, the discussion of language, function of each class and relationships between classes is included in this chapter.

Last but not the least, artificial player is desired to confront players so that interface is wanted to describe the process and *state* of the game which players can get the current information of the game. In this chapter, I will discuss some of the considerations I took into account when designing interfaces and the compromises I made when meeting some issues.

3.1 Language

Since this project is aim to construct an exploratory software, which the requirements are not as strict as commercial one, so that almost all mainstream languages can be used, such as *C*, *C++*, *Java* and so on. If the software is commercial, it is not doubt to consider improving the performance of the software as much as possible, because the devices of user are extremely different. In order to satisfy all the mainstream devices that can successful run the software, lower-level languages such as *C* and *C++* are likely to be the first choice owe to the fully programming freedom that the language gives to the programmers For example, in *C* and *C++*, programmers can manually allocate memory and garbage collection, which improves efficiency cause automatic allocation generally consumes more time and resources. It is similar that *Java* can also improve its efficiency though it does not have as many degrees of freedom as *C* and *C++*.

However, considering the limitation of time, the efficiency of software will not be the most focusing factor for this project, and the capacity of this software is not like that huge to

consider the optimization of efficiency. Therefore, *Python* will be chosen which is a simple and friendly programming language.

In addition to the simplicity of the language, *Python* has a large number of third-party packages that allow any function to be easily implemented in a short period of time when it is uncertain what functions to implement and what technologies to utilize. Meanwhile, methods of *Python* for configuring new environments and importing packages are relatively simple compared to other languages.

This project will design some interfaces though most of them is text interfaces(the reason will be explained in *section 3.3.2*). *Python* still be a perfect choice, cause it also has the support of graphical interface, such as *PyQt5*, so the possibility of further optimizing for *UI* in future research can be achieved in *Python*.

Finally, the last reason why I choose *Python* is that it is one of the most popular and mainstream languages in the field of *Artificial Intelligence*, because it is supported by a large number of *Artificial Intelligence* algorithm packages, which improve the possibility of testing different *Artificial Intelligence*(the implementations will be very efficient) in this software in the future research.

3.2 Workflow

When designing a game, the workflow of a program needs to be planned and analyzed before implementing in order to improve efficiency and clear the idea, especially when the flow is not naive.

Therefore, through the understanding of Samurai rules and main game loop, *Unified Modeling Language*(UML) is able to be used to represent the flow of the entire game(*Figure 3.1*), which is a straight forward and explicit method.

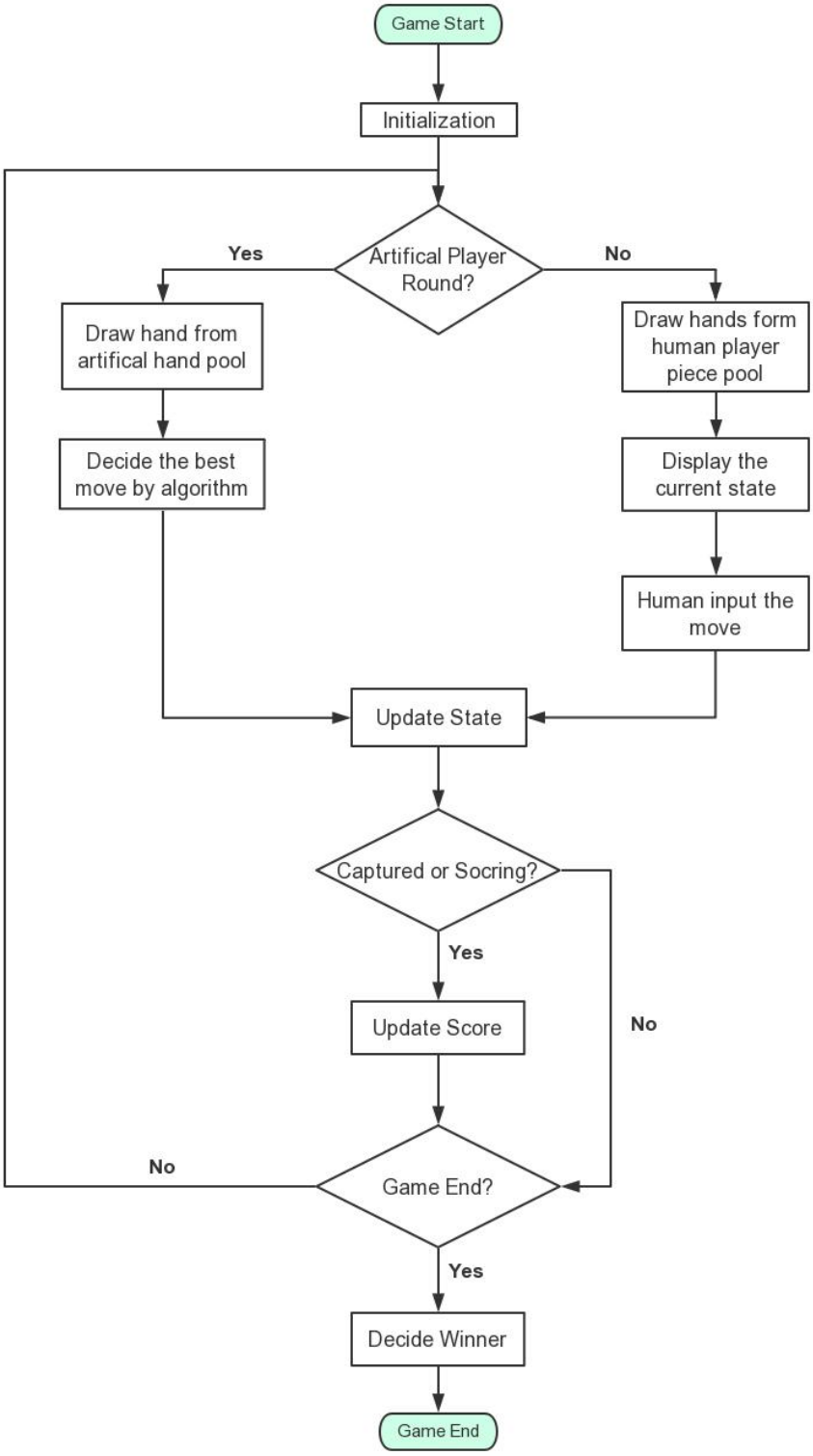

Figure 3.1 A brief workflow of *Samurai*

3.3 Game Design

This section will introduce how some important main game functions and representations are implemented during the programming of *Samurai*, such as the representation of the board and the judgment of the game end.

3.3.1 Representation of Game Components

Board Representation

The board representation of *Samurai* is pretty unique compared with other traditional board games, because in traditional board games such as *Chess*, the board is generally a rectangular grid. This means that simple two-dimensional coordinates can describe these board directly, and corresponding proximity relations can be calculated using two-dimensional coordinate systems. For example, the right neighbors of (1,1) can be simply figure out with adding 1 to x coordinate. However, the board of *Samurai* belongs to the hexagonal grid. Although two-dimensional still can be applied on the hexagonal board, the detailed representation method of hexagonal grid has slightly different with the rectangle one. Therefore the expression of x coordinate and y coordinate need to be adjusted. *Figure 3.2* compared the general rectangle coordinate expression with hexagonal coordinate expression.

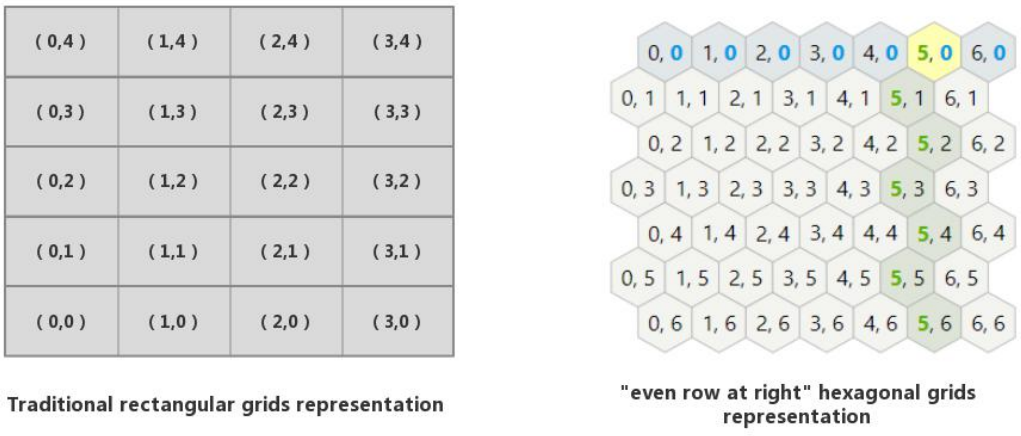

Figure 3.2 Comparison of rectangular girds and hexagonal grids representation

Besides, this layout leads to more complex neighbor calculations than the rectangle grid, because each grid has six neighbors instead of four and each row and column is misaligned Therefore, odd and even rows need to be distinguished, and they will have different neighbor calculation methods, as shown in *Figure 3.3*.

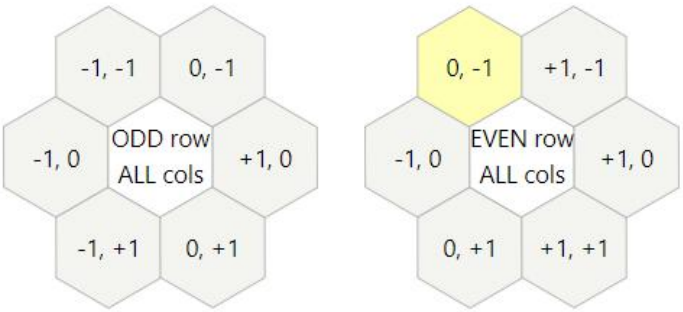

Figure 3.3 Neighbors calculation Algorithm for odd row and even row

In addition to representing the board in two-dimensional coordinates, there are other elements that need to be represented, such as the type of each tile and whether the tile is occupied by the pieces. These elements can be simply combined with the x and y coordinates of each tile to form a list, and the elements in the list are different form type to type. (*Figure 3.4*).

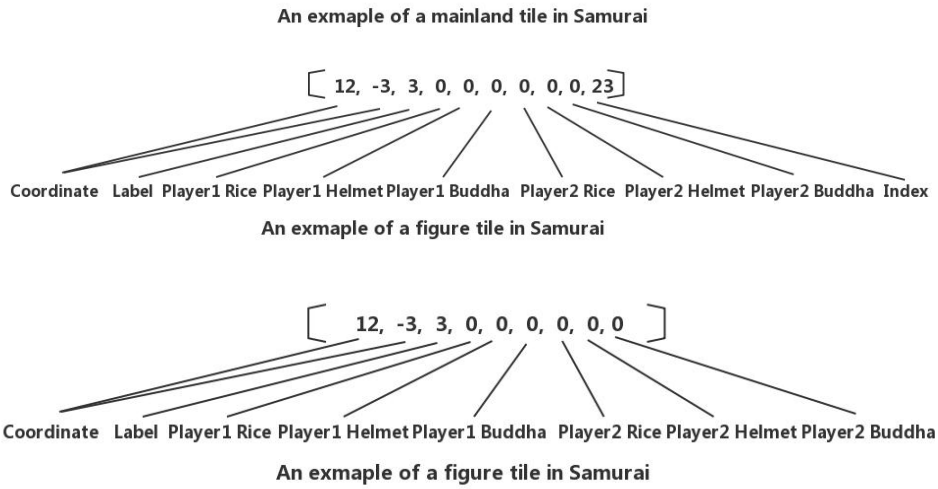

Figure 3.4 Examples of board representation for different types of tile

The meanings of number for each element in the array:

- Coordinate: Special (x,y) coordinate for each tile.
- Place *state*: No one takes current tile with value 0; Player1 takes current tile with value 1; Player2 takes current tile with value 2.
- Rice: The Rice captured points get by a player.
- Helmet: The Helmet captured points get by a player.
- Buddha: The Buddha captured points get by a player.

- Update *state*: If this tile is placed with a new piece, update *state* will be 1 to wait for updating; Value 0 means not waiting for update.
- Player1 Rice: The Rice captured points for player one on current figure tile.
- Player1 Helmet: The Helmet captured points for player one on current figure tile.
- Player1 Buddha: The Buddha captured points for player one on current figure tile.
- Player2 Rice: The Rice captured points for player one on current figure tile.
- Player2 Helmet: The Helmet captured points for player one on current figure tile.
- Player2 Buddha: The Buddha captured points for player one on current figure tile.
- Tile index: Each tile has an unique index.

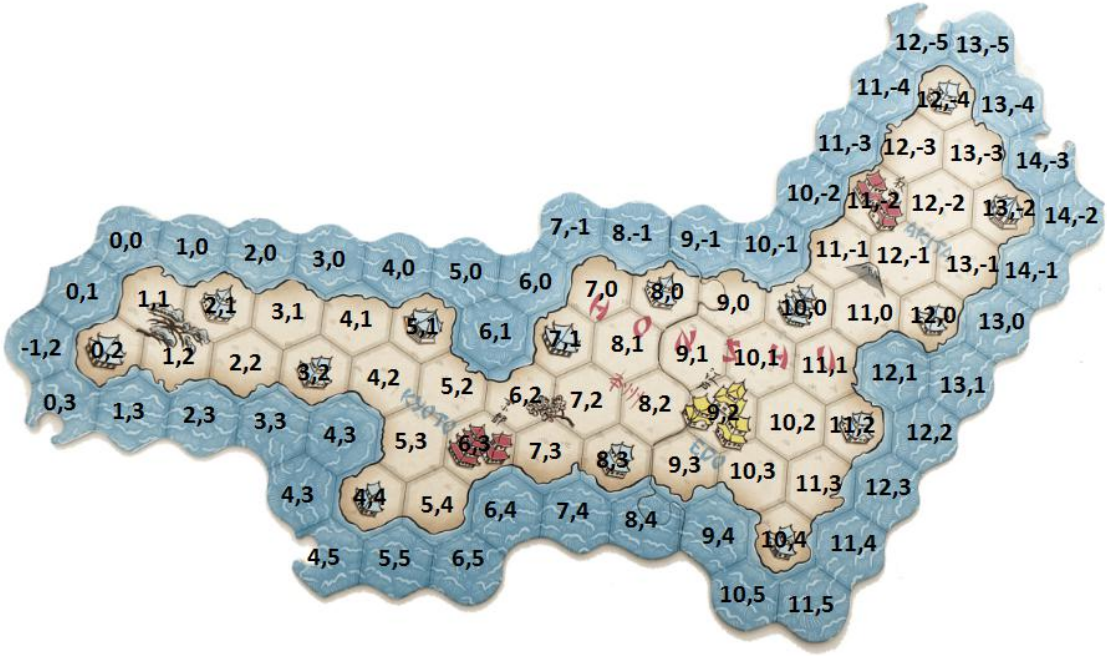

Figure 3.5 Coordinate representations of each tile

Piece Representation

The representation of the pieces in the game is similar to the board, which uses a one-dimensional array to express a particular piece, as shown in the figure.

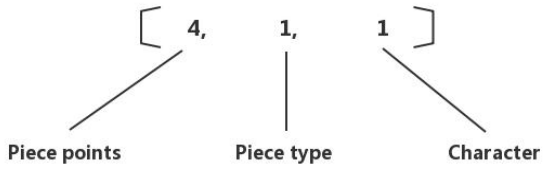

Figure 3.6 An example of a particular piece representation

The meanings of number for each element in the array:

- Piece points: Captured points current piece can add on the figure tiles.
- Piece type: Value 0 represents Rice type pieces; Value 1 represents Helmet type pieces; Value 2 represents Buddha type pieces; Value 3 represents Samurai type pieces; Value 4 represents Ship type pieces; Value 5 represents Ronin type pieces.
- Character: If current piece is a character piece: 0 means no; 1 means yes.

3.3.2 Interface

Due to time constraints, I would consider constructing an text-based interface instead of using a graphic interface. However, it is difficult for *Samurai* to use the text information to represent the current information of the board. For example, how to represent the position and relationship between the two tiles, and if a tile is placed by a piece is currently placed, how to represent the placed piece information about that tile. These issues were hardly to be displayed with merely text output. To solve this, in a limited time, I decided to use the combination of the real board and a text-based interface to display all the information of the current *state*. The representation of each tiles would be displayed in the text-based interface(*Figure 3.7*). By providing the player with a photo of a marked *Samurai* board with two-dimensional coordinates(*Figure 3.5*), and the x and y coordinate information in the representation of the board(*Figure 3.4*) displayed in the test-based interface, it was possible to know which tiles have been placed by which pieces. At the same time, the player can also input the two-dimensional coordinates of the tile that he wants to place(*Figure 3.8*). Therefore, the player can successfully compete against with a artificial player. Besides, the information of current hand would also be displayed using the representation method mentioned in the section *3.3.1(Figure 3.6)*, to show the current hand of the play(*Figure 3.9*).

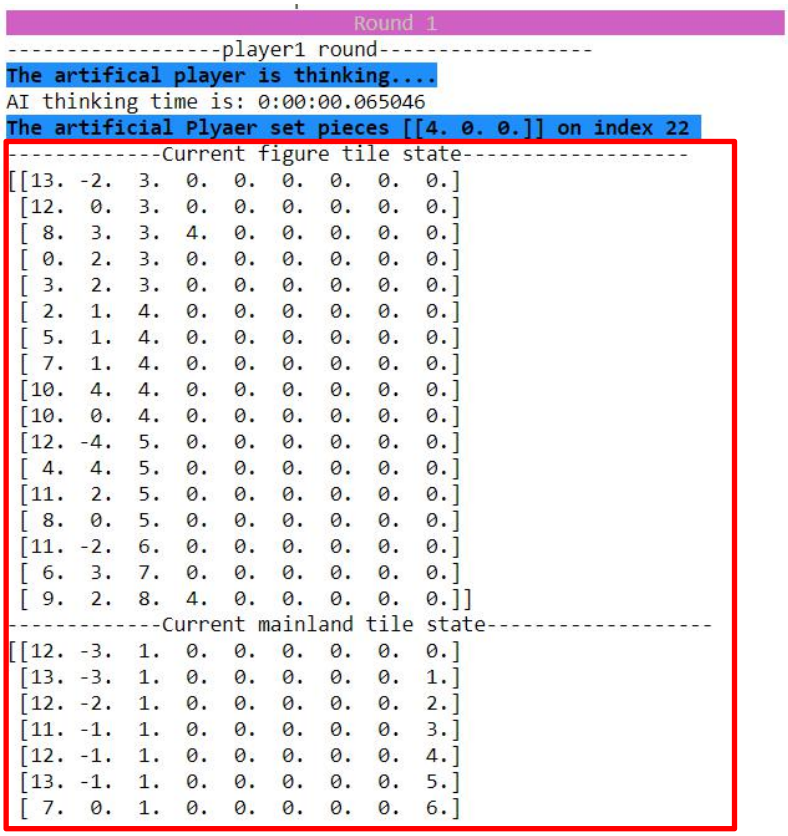

Figure 3.7 An example of an text-based interface when in the artificial player round; The text in red box shows current *state* of each tiles

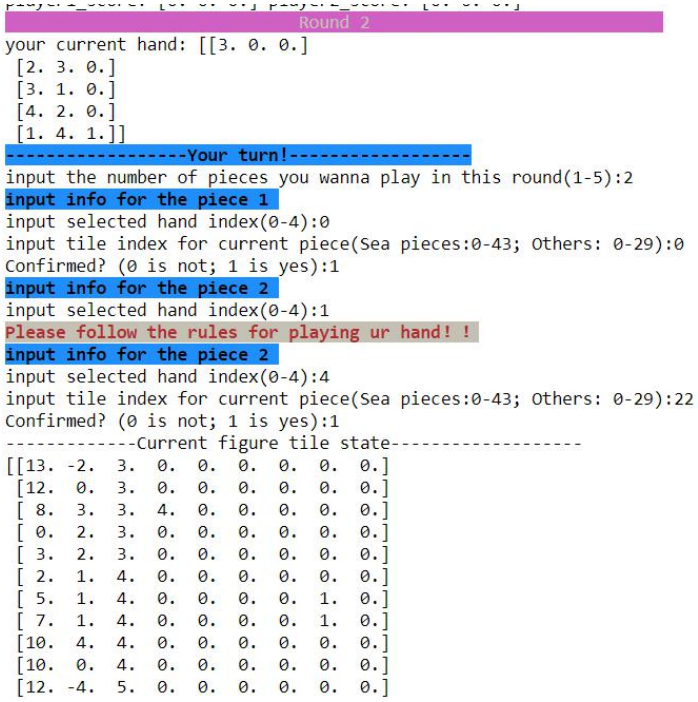

Figure 3.8 An example of an text-based interface when in the human player round

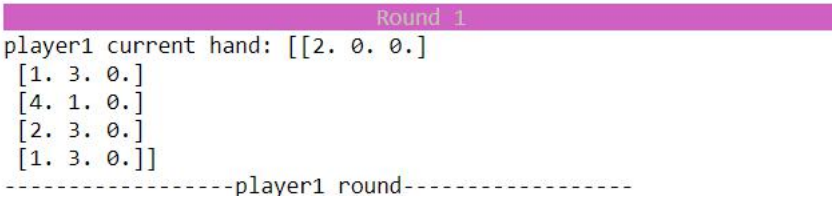

Figure 3.9 An example of an text-based interface of plays hand display

Chapter 4
Artificial Intelligence Player Implementation

After the implementation of the game, the next step is to test and analyze different AI strategies on the game platform. In this chapter, I will gradually describe the process of building AI, including the establishment of basic AI strategies, the establishment of improving AI strategies, and the adjustment of parameters. And through the comparison of different AI strategies,the best solution will be found, and finally will be used for testing.

4.1 Basic strategies

Before applying *Heuristic state Evaluation* and *Minimax Algorithm*, basic artificial players that only use naive strategies were constructed to test the workflow of the game and to be regarded as reference examples for improving artificial players. These basic strategies take no or just a little information of current *state* into account, as they were intended to imitate the behavior of in players who have no experience at all with *Samurai*.

4.1.1 Random Selection

Assuming that a human player has never played *Samurai* and even do not completely understand the rules of the game, the simplest strategy he might take is to randomly place his pieces on the board. This idea can be used as a simple strategy application on an artificial, which thinking is irrational.

When an AI takes a completely random approach, it means that it totally ignores any information in current *state*, and that the *payoff* of its next possible move is equal. As mentioned before, two pieces, “Piece exchange” and “Figure exchange”, were not considered in this project, so when we implemented this kind of AI, we simply selected a piece randomly from the hands of artificial player, and then determine whether it was a “Ship” piece or not. If so, the piece would randomly set on a playable sea tile, otherwise it would set on a random playable mainland tile.

This might seems useless, but actually it is useful as a baseline reference sample. Because, in a large number of experiments, random placement can be more general than a specific strategy, which helps test whether a strategy can compete with all types of players. In the project, I found that, supported by a large number of experiments, the random approach could still beat some advanced strategies, while some specific strategies could not.

4.1.2 Greedy Selection

Another basic strategy that can directly improve previous one is to take into account the most intuitive information on the board and immediately choose the highest *payoff* from it. For example, in *Samurai*, the most intuitive positive *payoff* is to place the piece that can achieve the most figure captured points on current *state,* regardless of how many points the other player has already taken up and whether there has already got excessive points by his own. Under this strategy, the artificial player will impatiently throw all the available pieces on the board without regard of the potential future advantage if keeping some hands. In other words, this strategy is very rude, which merely considers to maximize the immediate benefits without considering the possibility of future benefits and interference from opponents.

There are some special pieces in *Samurai*: “Ship” and “Ronin”, which can be placed in any number of hands in a turn with low scores. They tend to be played as a decisive role when needed. Therefore, when I implemented the Greedy selection strategy, two model has been built. One is the primitive greedy selection strategy, which is to consider all the pieces that can be placed at present, and place each of them on tiles that can achieve the most figure captured points. A round boundary was added in the other, which means the situation that the player “pours” all the playable hands will just happened after a certain number of turns. While , before the boundary, only one greedy piece will be played.

4.1.3 Summary

Through the above discussion, I have established three basic strategies as baseline and reference samples.

Basic Strategy 1—Random Selection

Every *move* is decided fully random.

Basic Strategy 2—Greedy Selection

Each *move* play entire playable hands on the non-placed tile with highest captured points.

Basic Strategy 3—Greedy Selection with Round Boundary

If the round is before the boundary, only the hand that can get the highest captured points can be placed; Otherwise, the *Greedy Selection* without Round Boundary decide a playable tile.

4.2 Advanced strategies

In the second stage, *Heuristic state Evaluation* and *Minimax algorithm* that mentioned above was used to establish advanced strategies. Basic strategies would used as the baseline to test performance of different improved strategies, and to determine the best strategy for *Samurai* to against a human player.

4.2.1 Heuristic state Evaluation Functions

As mentioned in *section 2.2.1,* the performance of *heuristic function* probably affected by how much useful information for previous and current *state* is used. These information can be considered as separate figure functions composing a overall *heuristic function.* Therefore, when implementing a heuristic function for an artificial player, I was tried to make it improves step by step via gradually adding figure functions.

Figure Function 1—Domain

The *heuristic function* is a formula, which is composed of multiple *figure functions* according to intention of strategy, in order to calculate the values of each *move* and pick the best *move* for current *state.* The design of these *figure functions* are closely related to the game rules and information of the previous *states*. A effective *heuristic function* normally takes maximum considerations of all the game factors affecting the trend of the game.

In *Samurai*, the information of opponent hands is not visible by another player, so when setting the *heuristic function*, the domain information that a player can achieve comes from the *state* displayed on the board. As mentioned in the *greedy decision* before, the most intuitive element that can obviously improve *payoff* of a player is to place the piece on the tile that can give maximum captured figure points on its neighbors of figure tile. In other words, the placed piece need to adjunct to as much as figure tiles that has the same type with the piece. Therefore, when I designed the first *figure function* of the *heuristic function*, the greedy decision value had been added. However, slight change had to be applied, the first figure function needs to take into account the existed captured figure points obtained by the opponent. So the first *figure function* which also be considered as a domain figure function is as follows:

```
domain =  piece_captured_points + (already_captured_points - opponent_already_cptured_points)
```

Figure Function 2—Keep in hands

Obviously, *domain function* alone is not enough. As mentioned in *section 4.1.3*, the character piece: “Ronin” and “Ship”, which are two pretty special piece in *Samurai*, lead to more uncertainty and randomness for the move. As long as the player has “Ronin” or “Ship” they can release them in any turn or just keep in their hands for waiting better chance. Therefore, it is also necessary to consider whether these character hands should be kept in the hand. This is the main idea I would focus on the second *figure function* . However, for this project, I decide to simplify the complexity of *Keep in hands* function, because the situation of whether you need to keep hand in *Samurai is* too multifarious. This can lead to the implementation of this *figure function* would occupied too much times, so I decided to just use the same idea of *greedy decision with a boundary*(*section 4.2.1*), which determine whether or not to retain according to the round. However, different from the *greedy decision with boundary* which only depends on the number of turns, the artificial players still have the opportunity to play the pieces at the early stage by combining with other figure functions when using *heuristic function*. As with the *domain function*, I would optimize the strategy used in the greedy decision with boundary, where the propensity of player to play these character pieces increases with the number of turns. Keep_in_hand figure function is as follows:

```
keep_in_hand =  k * (round - boundary)
```

Among them, value *k* will have different values according to different hands , and these *k* would be optimized in next stage.

Figure Function 3—Score *state*

If the *domain function* can be regarded as *payoff* of the current board information and the *keep_in_hands function* as *payoff* of the hands information, then the last one would need to calculate *payoff* of the last important game component: Score. In *Samurai*, experienced players will occupy a figure type that he has the most in order to get a large score, and if the opponent obtains most number of one figure type, the player will be inclined to give up this type of figure to retain the strength to fight for other types of figure. For example, if player 1 has already achieve four “Buddha” figures, then player 2 should abandon “Buddha” to compete other types of figures on the board. Since the total number of figures in each

category is only 7, player 1 gets 4 of them, then that means he has won a overall score in “Buddha” type. In addition, player 1 will drop the battle for ”Buddha” figure to prevent player 2 from getting another type of figure.On the other hand, when player 1 has already achieved three “Buddha”, he will have a strong desire to compete for an extra on.

In order to achieve this goal, artificial players and opponents will gain a significant negative value in *Score state function* when they have 4 of a figure type. And when the player holds a type of figure closer to 4, then the payoff in the hand that can compete for this figure type will also be larger. Score *state* is as follows:

$$Score\_state = fixed\_number \,/\, |number\_of\_achieved\_figure - 4|$$

The number_of_achieved_figure in the above represents a specific type of figure, of course, when there are two figure types, their values will be superimposed. In addition, fixed_number will be optimized when tuning parameters.

In brief, three advanced can be created, I would display them in the summary(see *section 4.2.3*)

4.2.2 Minimax Algorithm

As mentioned above, the *Heuristic state Evaluation Functions* have certain limitations. They only consider the information available in the current *state*, and do not predict the behavior of the opponent. Therefore, the *Minimax Algorithm* look forward a certain number of *moves*, and maximize the own *payoff form* the worst situation that opponent is assumed to create.

In *Samurai*, when an artificial player want to predict moves of opponents, he need to know their hands. However, *Samurai* is a *imperfect information game* and *stochastic game,* hence it is difficult for a artificial player to predict the next move of opponent when he can not be sure of the hands of opponents. So a conventional solution for this is to use the *Expectiminimax*(see *section 3.3.2*), that is, set a probability to all the opponent pieces in the piece pool. The number of a type piece is higher the probability that piece in opponent hands is higher. In the experiment, I found that when setting the artificial player to predict the 2-step forward which restrictively calculated the piece in current opponent hands, the *Minimax algorithm* had already consumed numerous times(normally 45 minutes). If *Expectiminimax algorithm* was used, the number of pieces to be considered increases from 5 to 20 when predicting the *moves* of opponents. This means that the permutations and combinations of all the pieces will be significantly improved, then *Expectiminimax algorithm* would consume an unpredictable time. Within the time limitation, I decided to give up the implementation of the *Expectiminimax algorithm*. On this basis, the game rules had been simplified to allow the

artificial player to know the hand of opponent, so that the *Minimax algorithm* can be executed smoothly.

Besides, the depth limitation(*Figure 4.1*) was applied for *Minimax algorithm* cause if the algorithm predict entire possible game flow the number of branches of *Minimax* will be extremely numerous. For example, if the depth limitation is 7, there will have approximately two billion nodes at the deepest depth.

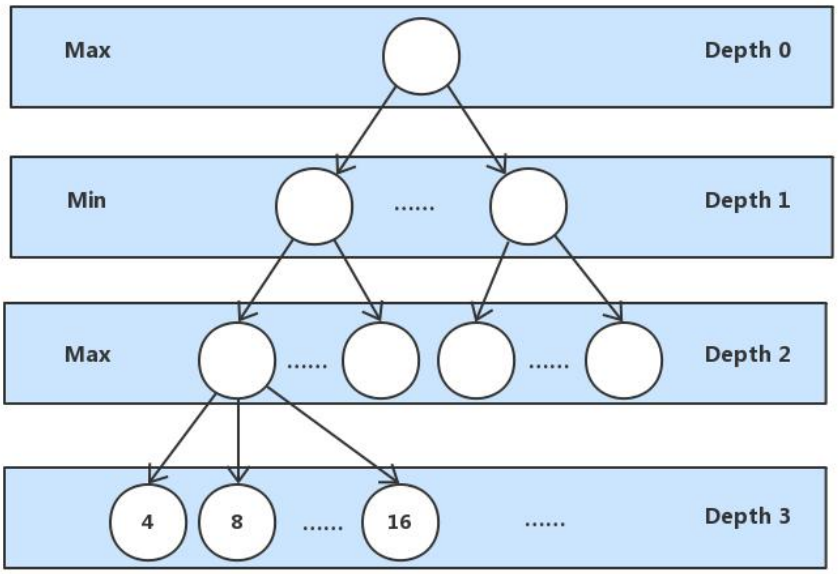

Figure 4.1 An example of a 3 depth *Minimax algorithm*

The one solution is to use *Alpha-Beta pruning*(see section 2.2.5) to eliminate some useless nodes, but I would not go to implement it in this project due to the time limitation as the *Expectiminimax algorithm.*

As mentioned in *section 2.2.3,* the values of all nodes in the deepest depth in *Minimax algorithm* can be calculated by using *heuristic algorithm*. I would choose the best solution from the heuristic functions(mentioned in *section 4.2.1*) to cooperate with *Minimax algorithm,* and then test the difference between using *Minimax algorithm* and without using it. And I would test the efficiency and performance of the minimax algorithm with different depth limits in *Chapter 5 .*

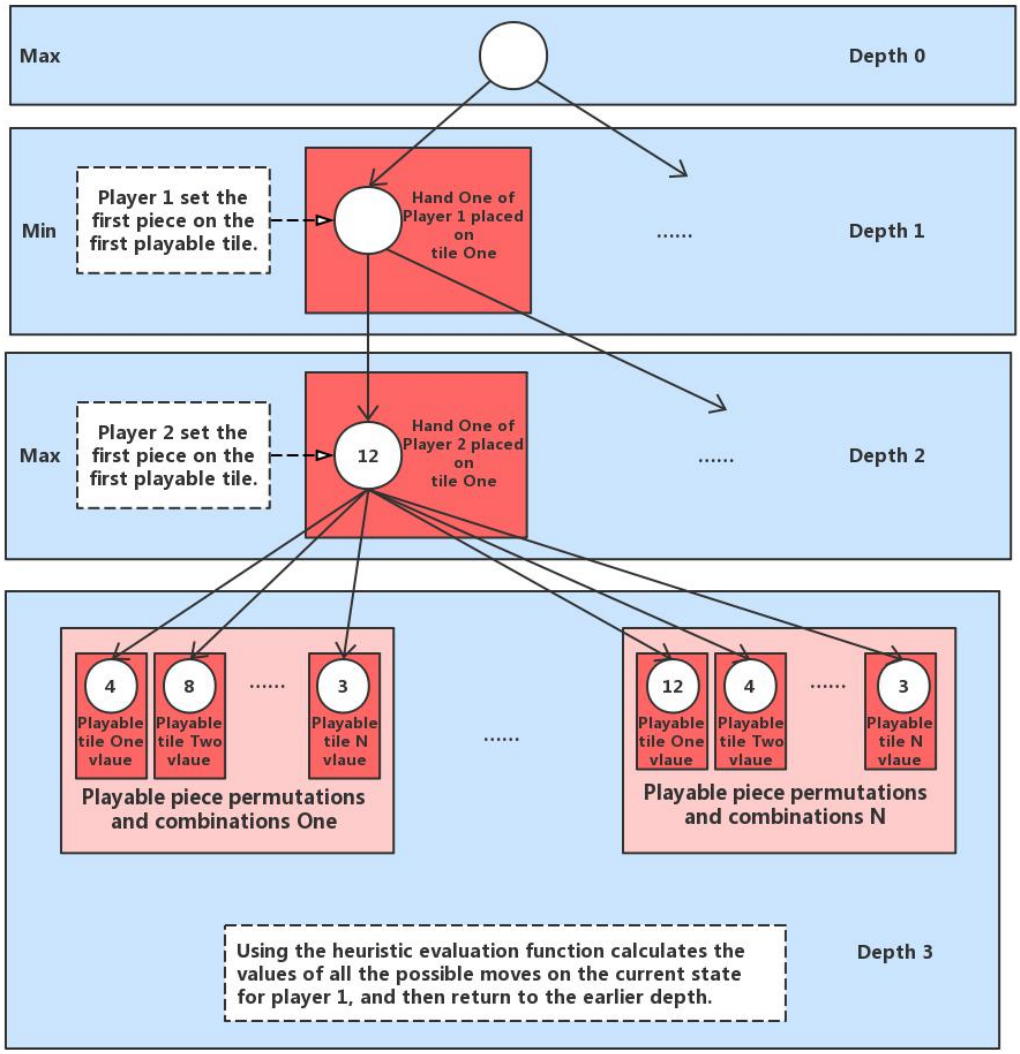

Figure 4.1 An example of a branch of a 3 depth *Minimax algorithm* for *Samurai*

4.2.3 Summary

By using the *heuristic evaluation function* and the *Minimax algorithm,* four separate *advanced strategies* were created that would also be compared to the previous *basic strategies* in *Chapter 5*. The best strategy would be picked in *Chapter 5* which would then be applied on *Minimax algorithm* meanwhile parameter adjustments that mentioned above would also be described in *Chapter 5*.

Below are the outline of the *advanced strategies* :

Advanced Strategy 1—Heuristic Function with *Domain*

Each *move* is decided by the heuristic function as follows:

```
Heuristic_Value =  w1 * Domain
```

Advanced Strategy 2—Heuristic Function with *Domain* and *Keep_in_hand*

Each *move* is decided by the heuristic function as follows:

```
Heuristic_Value =  w1 * Domain +
                   w2 * Keep_in_hand
```

Advanced Strategy 3—Heuristic Function with *Domain,Keep_in_hand* and *Score_state*

Each *move* is decided by the heuristic function as follows:

```
Heuristic_Value =  w1 * Domain +
                   w2 * Keep_in_hand +
                   w3 * Score_state
```

Advanced Strategy 4—Minimax algorithm with the best Heuristic Function

Each *move* is decided by the *Minimax algorithm* and the values of each node is calculate by the best *heuristic evaluation function* in a*dvanced strategies1, 2 and 3.*

Chapter 5
Testing

In order to determine the final artificial player solution, I would test the performance of all the strategies that mentioned in the previous chapter. Testing would allow these strategies to compete against one another by setting different baseline opponents strategies to compare the performance of different strategies. In addition, before comparing these strategies, as mentioned above, I would optimize the parameters involved in each of the a*dvanced strategies*, and then taking the best parameter solution for the final strategy comparisons. Ultimately the best strategy will compete with the human player to test the final performance.

5.1 Tuning Parameters

I will first optimize the parameters that need to be adjusted in each figure function(see *section 4.2.1*),such as parameter *k* , which can be called as internal parameters, and then optimize the weight parameters of the entire heuristic function(see section *4.2.1*), such as *w1*, *w2* and *w3*, which can be called as the external parameters.

5.1.1 Internal Parameters in Advanced Strategy 2

In the *figure function 2*(see section 4.2.1), the value of *k* should have a different value for each type of piece, then the *k* value would be divided into four separate parameters: *k1* represents the hold parameter of the normal pieces(e.g. 4-point Rice and 4-point Helmet); *k2* represents the hold parameter of the “Samurai’ pieces(e.g. 3-point Samurai); *k3* stands for “Ship” hold parameter(e.g. 2-point Ship); *k4* stands for Ronin reserved parameters (eg 1-point Ronin). I would merely explore the values of *k3* and *k4,* cause normal pieces and “Samurai” pieces should be placed if there is a suitable position on the field. So it is no sense to keep them in the hand, and they need to be placed in the early stages to gain an advantage. On the other hand, if *k1* and *k2* were also considered for optimization at the same time, it would be much more difficult to find the optimal solution, which would have to use Machine Learning or Bio-inspired Algorithms to obtain the optimal solution(see *section 6.2*). Since the time limitation, i tried to make this optimization be simper and could be roughly found a local optimal solution. Similarly, if the exploration of *Round Boundary* was introduced, the space complexity would also be increased. Therefore, merely two parameters would be adjusted to roughly find a local optimal solution for *figure function 2*, which had already achieve a satisfactory performance, normally from around 78% to 85%

winning rate(see *Figure 5.1*) The exploration of these parameters could be carried out in the future development which would be mentioned the methods of optimizing these parameters in the section 4.2.2 .

Therefore *k1* and *k2* would be given a fixed value 0, then according to some rough experiments I would limited the value of *k3 and k4* from 0 to 5. Besides, the game usually ends around 20 rounds, so a fixed value of 10 would be set for the *Round Boundary*. And *Basic Strategy 2*(*see section 4.1.3*) would be used as baseline strategy to compete with *Advanced Strategy 2.*

Test 1 Result

Step size would be 0.5 in the first test to find a brief optimal solution and for each parameter combination, the game would run 100 times. The test information would like follows:

- k1, k2 = 0
- k3 = 0~5
- k4 = 0~5
- Round Boundary = 10
- Step Size = 0.5
- Iteration = 100

The result is as follows:

Average Winning Percentages(%)(k1,k2=0)											
k3	**K4=0**	**0.5**	**1**	**1.5**	**2**	**2.5**	**3**	**3.5**	**4**	**4.5**	**5**
0	79	85	75	85	80	84	78	73	79	76	72
0.5	80	81	72	82	77	77	76	79	89	74	77
1	78	82	78	83	78	82	90	86	74	83	83
1.5	79	85	75	78	86	81	80	80	74	83	86
2	84	82	81	80	74	82	82	83	69	87	83
2.5	81	77	78	83	81	82	79	79	82	86	85
3	83	83	84	84	82	83	81	78	83	78	80
3.5	80	79	87	81	82	83	80	84	84	79	77

4	79	79	85	81	79	85	78	83	82	81	88
4.5	78	79	82	79	77	80	83	83	83	83	79
5	78	79	76	78	85	81	81	74	77	79	80

Figure 5.1 The average winning percentages of *Advanced Strategy 2* with different *k3* and *k4* combinations when competing with *Basic Strategy 2*.

Test 2 Result

On the basis of the first test I narrowed the scope and the steps of exploration to improve accuracy. The explore range is as the blue area in *Figure 5.1.* Step size would be 0.2 in the second test to find a more specific optimal solution, and for each parameter combination the game would run 300 times. The test information would like follows:

- k1, k2 = 0

- k3 = 0.7 ~ 1.5

- k4 = 2.5 ~ 3.5

- Round Boundary = 10

- Step Size = 0.2

- Iteration = 300

The result is as follows:

Average Winning Percentages(%)(k1,k2=0)						
k3	**k4 = 2.5**	**k4 = 2.7**	**k4 = 2.9**	**k4 = 3.1**	**k4 = 3.3**	**k4 = 3.5**
0.7	81.33	86.67	77.00	73.33	75.67	79.00
0.9	75.33	77.00	79.33	82.00	80.00	81.33
1.1	79.00	80.00	83.00	89.00	76.33	76.33
1.3	81.00	78.67	80.00	84.00	80.33	79.33
1.5	78.67	82.00	85.00	78.66	75.66	78.66

Figure 5.2 The average winning percentages of *Advanced Strategy 2* with different *k3* and *k4* combinations in a more specific range when competing with *Basic Strategy 2*.

I thought I need to stop the parameters exploration, because the difference between the winning percentage had became tiny. Although, through the exploration, I found that if the

exploration continue there would still have 0~2 point increase or decrease in winning rate, it is not worth with consideration of the time consumption. So the final solution for *k3* and *k4* are 1.1 and 3.3 separately.

Summary

By exploring, a partial optimal parameter solution for *figure function 2* could roughly be found , as shown below:

k1=0, k2=0, k3=1.1, k4=3.1, Boundary Round=10

5.1.2 External Parameters in Heuristic Function

Then, I would test the external parameters, that is, the weight values of the each *figure function* in the entire *Heuristic Function*. In theory, the *domain function*(see *section 4.2.1*) should have the highest weight in the entire *Heuristic Function* to guarantee the better perform cause it is the most direct *payoff* for players. That means directly getting figure captured points is more important than keeping hands and considerations of current score. Therefore I set the weight of the domain with 1, in which case to find the most appropriate values for *w2* and *w3*, and the weight range for *w2* and *w3* would be between 0-1.

Moreover, I would change the strategy of opponent from *Basic Strategy 2* to *Advanced Strategy 1(see section 4.2.1),* cause when *Advanced Strategy 2* competed with Basic Strategy 2 the winning percentages had already be high enough(normally from 87%-95%). Therefore it is pretty hard to the increasing performance of the changing parameters, *Advanced Strategy 1* as a better opponent would make the improvement more obvious.

Test 1 Result

Step size would be 0.1 in first test to find a brief optimal solution and for each parameter combination, the game would run 300 times. The test information would like follows:

- w1 = 1
- w2 = 0~1
- w2 = 0~1
- Step Size = 0.2
- Iteration = 300

The result is as follows:

Average Winning Percentages(%)(w1=0)						
w2	w3 = 0	w3 = 0.2	w3 = 0.4	w3 = 0.6	w3 = 0.8	w3 = 1
0	63.67	65.33	70.00	78.00	71.33	67.67
0.2	73.33	71.33	82.00	76.33	81.33	72.67
0.4	76.00	86.00	76.33	72.67	69.67	69.33
0.6	63.00	82.00	71.00	76.67	73.33	66.00
0.8	77.00	82.33	82.00	72.33	73.00	64.00
1	75.33	77.67	75.00	70.00	76.33	66.00

Figure 5.3 The average winning percentages of *Advanced Strategy 2* with different *w2* and *w3* combinations when competing with *Advanced Strategy 2*.

Test 2 Result

On the basis of the first test I narrowed the scope and the steps of exploration to improve accuracy. Step size would be 0.04 in the second test to find a more specific optimal solution and for each parameter combination, the game would run 300 times. The test information would like follows:

- w1 = 1
- w2 = 0.3~0.5
- w3 = 0.1~0.3
- Step Size = 0.04
- Iteration = 300

The result is as follows:

Average Winning Percentages(%)(w1=0)						
w2	**w3=0.10**	**w3 =0.14**	**w3=0.18**	**w3=0.22**	**w3 = 0.26**	**w4=0.30**
0.30	79	73	76	81	85	70
0.34	73	78	73	76	78	73
0.38	82	74	85	87	75	69
0.42	68	76	74	79	73	78
0.46	78	82	78	75	75	78
0.50	85	81	82	80	81	76

Figure 5.4 The average winning percentages of *Advanced Strategy 2* with different *w2* and *w3* combinations in a more specific range when competing with *Advanced Strategy 1*.

Summary

By exploring, a partial optimal parameter solution for Heuristic function in *Advanced Strategy 3* could roughly be found , as shown below:

w1=1, w2=0.38, w3=0.22

Besides, *w2* would also be applied in *Advanced Strategy 2.*

5.2 Strategy Comparison

After adjusting the internal and external parameters of the *Advanced Strategies*, all the strategies are ready to test their performance as an artificial player. As I mentioned earlier, I would not perform all the strategies to against a human player, because it makes quite time-consuming and unwise. Obviously, *Basic strategies* are pretty difficult to against a real human player hence, in this chapter, I mainly by setting a baseline strategy to compete with another strategy to test the performance of different strategies. The baseline strategy would be changed when the performance of the competitive strategy is better.

Besides, the optimal strategy found by experiments would be applied as the Heuristic function for *Minimax algorithm*. Finally, by comparing the performance of the best strategy with Minimax algorithm and the best strategy without Minimax algorithm, it is determined which strategy would be used in the final stage of the final confrontation with human players.

Three sets of 300 plays would be performed for two strategies comparison, the average winning percentage would be considered as the judgment criteria.

Player	Test One	Test Two	Test Three	Average Win Percentage(%)
Basic Strategy 1	21	28	14	6.99
Basic Strategy 2	279	272	286	93.01

Figure 5.5 *Basic Strategy 1* vs. *Basic Strategy 2*

Player	Test One	Test Two	Test Three	Average Win Percentage(%)
Basic Strategy 2	124	130	115	41.00
Basic Strategy 3	176	170	191	49.00

Figure 5.6 *Basic Strategy 2* vs. *Basic Strategy 3*

Player	Test One	Test Two	Test Three	Average Win Percentage(%)
Basic Strategy 3	55	44	68	18.56
Advanced Strategy 1	245	256	232	81.44

Figure 5.7 *Basic Strategy 3* vs. *Advance Strategy 1*

Player	Test One	Test Two	Test Three	Average Win Percentage(%)
Advanced Strategy 1	88	68	92	27.56
Advanced Strategy 2	212	232	208	72.44

Figure 5.8 *Advance Strategy 1* vs. *Advance Strategy 2*

Player	Test One	Test Two	Test Three	Average Win Percentage(%)
Advanced Strategy 2	138	103	112	39.22
Advanced Strategy 3	162	197	187	60.78

Figure 5.9 *Advance Strategy 2* vs. *Advance Strategy 3*

Based on the observation of the above experimental results, simply adding some profit value to the *Basic Strategy 1* could significantly improve the performance of artificial players(Figure 5.5) though this was merely compared to the randomly placed strategy. It proofed that artificial player began stimulating a experience gamer instead of an non-experienced player. In addition, the performance of the artificial player was improved slightly from *Basic Strategy 2* when it was restricted “pour” all its available hands on the board mindlessly (*Figure 5.6*).

It could also be found simply from the table that the *heuristic evaluation function* also has significant improvement on the artificial player(*Figure 5.7*), even if just adding a simple *domain figure function*. In addition, the experiment proves that, as mentioned before, if all the game information within the current *state* was considered as much as possible, the value given by the heuristic evaluation function would be a greater reference value. *Advance Strategy 3* and *Advance Strategy 2* performed better than the Basic Strategies after *Keep_in_hand figure function* and *Score_state figure function* were added respectively. This was what I expected, and it also proved that the latter two *figure functions* play an positive role in *Heuristic Function*.

5.3 Minimax and Non-Minimax Strategy Comparison

Moreover, I would apply the *Heuristic Function* in *Advance Strategy 3* on *Minimax alogrithm.* Different depth limitations would be tested to find the balanced between performance and efficiency. Merely the odd depth limitation would be tested cause I desire to stop the look-forward at the move of the artificial player, and it is easy to calculate the the value of *leaves* at that depth as well. The test result is as follows:

Depth	Plays	Minimax AI win percentage (%)	Elapsed time
1	300	56	5 mintues
3	3	100	21 hours
5	1	-	Terminated after 18 hours

Figure 5.10 Minimax vs. Non-Minimax with different depth limitations

As can be seen from the above table, *Minimax algorithm* consumes numerous time, and at depth of 5, the running time is unpredictable. This is because, in *Samurai*, when a player has more than one playable hand, the number of a possible move increases dramatically. In the depth of 3 , for example, the most *leaves* situation is that all hands of player 1 and player 2 can be placed on board in one turn, i.e., two players both have one normal piece and four character piece. In this case, there will be more than 10 million possible moves for 2-step-look-forward, which lead algorithm to run extremely slowly. Certainly, there are many ways to solve this situation, such as the *Alpha-beta pruning*(see *section 2.2.5*), which can be used to reduce the *branches* of *tree*. However, due to time constraints, I could not implement these possible solutions, but I would mention them in the future work.

However, within the limited 3 precious plays, it could be predicted that the performance of the artificial player with *Minimax* may surpass the *Non-Minimax.* Besides, as a explanation, the winning rate of depth 1 merely had 56% because there had no look-forward at all when the depth is 1, which means it just two same algorithms competed each other

5.4 Human Player Test

Due to the running time of *Minimax algorithm*, I merely let one participant fight against *Advanced strategy 4(see section 4.2.3)* for 2 plays. However, I added *Advanced strategy 3* into the experiment of human player testing, although this might make the final result not meet my expectation. In the testing of *Advanced strategy 3*, I found two participants. All the participants can be consider as a low-experienced player, cause they just play with each

other once before stating the test. I had already covered how to pit the player against the computer (see *section 3.3.3*), so I would not describe in this section. The experimental results are as follows:

Advanced Strategy 3		
Human player	**Plays**	**Win percentage (%)**
1	3	33.3
2	3	66.7

Advanced Strategy 4		
Human player	**Plays**	**Win percentage (%)**
1	2	100

Figure 5.11 Results of human testing with two strategies

Chapter 6
Conclusion

In this chapter, I would summarize the process of the entire final project and assess whether the final results meet the original aim and objectives, including discussing shortcomings, personal reflection and future extending and improving ideas.

6.1 Project Outcomes

The experimental results would be compared with the initial aim and objectives(see section) to assess the completion of the project and where it needs to be improved. Four major aim and objectives were created in the early stage of project, so I would review and evaluate them separately.

1) Produce a playable programme of the board and card game *Samurai*.

A full-process two-player *Samurai* was successfully implemented by using *Python*. When no human players participate, two artificial players with different strategies would be able to play the game automatically until the end of the game, and each round of the two players would be represented by a simple text output. On the other hand, when a human player is playing against the artificial player, a simple text interface will display the current board *state* and receive the commend of the human player. By cooperating with the picture of a real game board(see *section 1.3*), the human player can successfully compete with the artificial player.

2) Develop an AI algorithm for playing *Samurai.*

Artificial players with different strategies were successfully implemented to complete a full-process of a game, no matter in the situation of competing other artificial players or human players.

3) Compare different algorithms to find the best AI solution for *Samurai*

There were 7 different algorithms were successfully built and applied to the game. Through the competition between different algorithms, an “optimal” algorithm was finally selected. Although this “optimal” algorithm still had spaces to improve(e.g. numerous running time), its result is better than other algorithms(see *section 5.4*).

4) Test the programme and evaluate the performance of AI against human players.

The program was successfully tested on both artificial players and human players. However, I did not perform all of my strategies on human player testing, cause basic strategies are hard to really compete with human players. It would be time-consuming and unintentional to test all the strategies on human players, so that I finally decided to apply the best strategy to the human player testing. Although the final algorithm may take too much time to run, the result from only a few plays still shows that the final algorithm performs quite well.

Overall, the experimental results achieved the expected aim, although the running time of *Advanced strategy 4* was unexpected long. *Advanced strategy 4*, however, performed well in a few experiments, although it merely beat players who are not an expect for *Samurai* or even a novice. In addition*, Advanced strategy 3* also has a chance to beat these players with quick calculations, although they may have trouble to win when they encounter more experienced players. Anyway, the experimental results met the basic requirements that set at the beginning, and some methods of expansion and improvement will be discussed in the *section 6.2*.

6.2 Personal Reflection

Overall, although the completion of the entire project is challenging, I still enjoyed the whole process whether on the exploration of the unknown field or the idea communication with the supervisor. Throughout the project,I learned a variety of AI algorithms that are beyond the knowledge of the course. Although the process of exploring my own unknown field is difficult, the results of the final report are satisfactory and positive for me. So I am very honored to be able to study this topic and learn a lot from it.

However, I still found some weaknesses that need to be improved. First of all, from a technical perspective, the management and the initial design of the code are obviously insufficient. At the beginning of the project, there was no preliminary planning and design of the modules of the code so that repeated modifications were performed during the implementation of each function, which resulted in a large amount of time wasted. Besides, a lot of code duplication and too deep depth of loops have caused my code to not satisfy a *good smell* code[28]. These might indirectly lead to a slowdown computation time of final solution, and the difficulty of expanding this project will be greatly improved. This project is a non-commercial standard project so that I did not spend a lot of time on code construction

and refactory. However, in the future projects, the requirements will be greatly improved, so I still need to learn from this project.

In addition, the ability to manage time also needs improvements. Although the final progress of the report was completed in the plan, I often found out, during the experiment, that I was completing a phased goal beyond the designed time at the beginning of the report. This results in the need to compress the rest of the time to keep the overall progress unaffected, such as compressing report writing time and human player testing time. In the following projects, summing up this experience, I will add some flexible time in each stage to ensure that each part can be completed smoothly and have free time to be adjusted.

In brief, In this project, not only some areas of expertise can be learned, but also some lessons can be learned on basic skills. These will be very useful to improve in future projects.

6.3 Future Work

Alpha-beta pruning

As mentioned before, the time consuming of *Advanced strategy 4* was extremely long due to the numerous game tree created by *Minimax algorithm*(see *section 4.2.2*), *Alpha-beta pruning* could be used to reduce the capacity of the game tree and trim the useless branches to improve the efficiency of the algorithm(see *section 2.2.5*). This not only benefit to computation of the program, but also increases the efficiency of the test process.

Improving Heuristic Function

There still had room of improvement for the *Heuristic Function* in the experiment. The factors considered by the *Heuristic function* were still not complete enough. For example, when a player has a high probability of occupying a figure, he should appropriately abandon continuing set the high score piece around that figure tile. A more detailed *Heuristic function* would further enhance the performance of the artificial player.

Graphic Interface

In the later stage, a graphical interface could be developed to replace the current text interface, which not only can intuitively represent the current board *state*, but also make the user more simple to play with the artificial player.

Turning parameters by Genetic Algorithm

In s*section 5.1*, it was mentioned that when there are multiple parameters to be adjusted, some advanced algorithms are needed. *Genetic Algorithm* is an excellent choice, which combines different parameter values with a fitness function to generate mutations and crossover to find the optimal solution, and this optimal solution can be the global optimal. On the other hand, *Genetic Algorithm* also evolves to produce better final solutions by combining different heuristic functions[16].

List of References

[1] Newell, A., Shaw, J.C. and Simon, H.A., 1958. Chess-playing programs and the problem of complexity. IBM Journal of Research and Development, 2(4), pp.320-335.

[2] Shannon, C.E., 1988. *Programming a computer for playing chess.* In Computer chess compendium (pp. 2-13). Springer, New York, NY.

[3] Zobrist, A.L., 1969, May. *A model of visual organization for the game of GO*. In Proceedings of the May 14-16, 1969, spring joint computer conference (pp. 103-112). ACM.

[4] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. and Sutphen, S., 2007. *Checkers is solved*. science, 317(5844), pp.1518-1522.

[5] Myerson, R.B., 2013. *Game theory*. Harvard university press.

[6] Osborne, M.J., 2004. *An introduction to Game theory* (Vol. 3, No. 3). New York: Oxford university press.

[7] Nash, J.F., 1950. Equilibrium points in n-person games. Proceedings of the national academy of sciences, 36(1), pp.48-49.

[8] Rosenthal, R.W., 1981. Games of perfect information, predatory pricing and the chain-store paradox. Journal of Economic theory, 25(1), pp.92-100.

[9] Nash, J.F., 1951. Non-cooperative games. Annals of mathematics, pp.286-295.

[10] Jacobson, D., 1973. Optimal stochastic linear systems with exponential performance criteria and their relation to deterministic differential games. IEEE Transactions on Automatic control, 18(2), pp.124-131.

[11] Dixit, A.K. and Skeath, S., 2015. Games of Strategy: Fourth International Student Edition. WW Norton & Company.

[12] Clune, J., 2007, July. Heuristic evaluation functions for general game playing. In AAAI (Vol. 7, pp. 1134-1139).

[13] Stockman, G.C., 1979. A minimax algorithm better than alpha-beta?. Artificial Intelligence, 12(2), pp.179-196.

[14] Yen, S.J., Chou, C.W., Chen, J.C., Wu, I.C. and Kao, K.Y., 2014. Design and implementation of Chinese dark chess programs. IEEE Transactions on Computational Intelligence and AI in Games, 7(1), pp.66-74.

[15] Zettlemoyer, L. Adversarial Search. CSE 473: Artificial Intelligence Autumn 2011 [2011 10/08/17]; Available from: https://courses.cs.washington.edu/courses/cse473/11au/slides/cse473au11-adversarial-search.pdf.

[16] Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T.A.M.T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation, 6(2), pp.182-197.

[17] Chalkiadakis, G., Elkind, E. and Wooldridge, M., 2012. Cooperative game theory: Basic concepts and computational challenges. IEEE Intelligent Systems, 27(3), pp.86-90.

[18] Sukumaran, J. and Holder, M.T., 2010. DendroPy: a Python library for phylogenetic computing. Bioinformatics, 26(12), pp.1569-1571.

[19] Buckland, M., 2005. Programming game AI by example. Jones & Bartlett Learning.

[20] Tamano, T., 1979. Method of playing a board game. U.S. Patent 4,171,814.

[21] Gutschmidt, T., 2004. Game Programming with Python, Lua, and Ruby. Premier Press.

[22] Yannakakis, G.N., 2012, May. Game AI revisited. In Proceedings of the 9th conference on Computing Frontiers(pp. 285-292). ACM.

[23] Levi, M., 1997. A model, a method, and a map: Rational choice in comparative and historical analysis. Comparative politics: Rationality, culture, and structure, 28, p.78.

[24] Russell, S.J. and Norvig, P., 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,.

[25] Pearl, J., 1984. Heuristics: intelligent search strategies for computer problem solving.

[26] Tanimoto, J. and Sagara, H., 2007. Relationship between dilemma occurrence and the existence of a weakly dominant strategy in a two-player symmetric game. BioSystems, 90(1), pp.105-114.

[27] Melkó, E. and Nagy, B., 2007. Optimal strategy in games with chance nodes. Acta Cybernetica, 18(2), pp.171-192.

[28] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

Appendix A
External Materials

All of the code implemented and guidance on how to play with artificial player in this project is available on *GitLab* under the following URL:

https://gitlab.com/SASAD3/Final_Project_ml17y35z_Yi

This project made use of *Python 3* for the software implementation, and no other external resources were used.

Appendix B
Ethical Issues Addressed

In the testing stage of my project, two participants were invited to compete against the artificial player. Both volunteers were be fully explained the aim of the test and signed the *Consent Form* before testing. These forms will be submitted in a detached envelop.