UNIVERSITY OF HERTFORDSHIRE

School of Computer Science

Modular BSc Honours in Computer Science

6COM1054 - Artificial Intelligence Project

Final Report

April 2018

AI algorithm in Texas Hold'em Game

Yifan Chen

Supervised by: Daniel.Polani

Abstract

Texas Hold'em Game is a variation of the card game of poker with a long history. In modern society, Texas Hold'em Game is an important social activity in the financial community. Financial transactions and Texas Hold'em Game are all zero-sum games. They cannot produce any wealth but only transferring them. The ante in the game is similar to the handling fee in financial transactions. The player would encounter different kinds of opponents in different games, and they cannot fully understand every opponent's habits, strategies and actions. Participants would never know what will happen next time just like the market. The characteristics of the game decide that this is not only a game dealing with cards, money and luck, but also psychology, information and mathematics. Through Texas Hold'em, traders can learn more skills and lessons in each game, thereby making profits in actual transactions.

Incomplete information games and multiple player games have been the two major problems in the programming world in recent years. For incomplete information games, the situation is entirely different from the complete information game (such as go). Gamers need not only to figure out what their opponents know but also to figure out what kind of actions they will reveal their private information. The classical way to solve incomplete information game is to compute or approach the Nash equilibrium. The complexity for calculating the Nash equilibrium is high that it does not perform well in the case of multiple players.

I have constructed a game environment of Texas Hold'em with simplified rules to explore the application of reinforcement learning and probabilities in a multi-player environment. The results show that the reinforcement learning AI under plenty domain knowledge has a particular intensity, but there is still much room for improvement from the top level.

Acknowledgement

I would like to thank my supervisor, Daniel Polani who has been very helpful during my entire project development; providing accurate professional knowledge with his unique point of view and inspiring my potential throughout the process.

Table of Contents

Chapter 1 – Introduction

1.1 Project Introduction

Incomplete information game and multiple player games have been the two major problems in the programming world in recent years. Since AlphaGo came out in 2015, the enthusiasm for AI around the world is increasing rapidly. Motivated by this I decided to develop a game environment with different AI algorithms.

The project is divided into three parts. Part one is research reviewed. Reminded by my supervisor, I need to be aware of the latest technology, knowing the limitations of existing methods before following abstract concepts to build something useless. After gaining the necessary overview of this academic field, a game environment was built. Finally, the goal was to apply the AI algorithm to the existing program, looking forward to improving the performances.

1.2 Aim of Project

The main aim of the project is to construct a game environment and apply different AI algorithms. For the development process, skills of coding and debugging need to be improved. The secondary goal is to figure out the suitable algorithm for the game and find the right direction for further research; Discuss the possibility of abandon domain knowledge if possible.

1.3 Project Objectives

1. Construct a Texas Hold'em game environment with simplified rules
2. Apply complicated settlement arrangements for multiple rounds with multiple players
3. Apply different AI methods to compare their performance

1.4 Report Structure

Chapter 2 – explain the rules of Texas Hold'em Poker, gives background information and research carried out on state-of-the-art AI like Libratus and AlphaZero.

Chapter 3 – describe the structure of the program, the variation of the rules and fully explain each function.

Chapter 4 – describe the test and evaluation process that was carried out. Analyse the replay data. Identify bugs and interesting behaviour.

Chapter 5 – make an overall conclusion and further improvement.

2.1 Introduction

This chapter is to introduce the background knowledge of Texas Hold'em Poker Game, the concept of incomplete information game and multiple player games. Afterwards, I would briefly explain the main structure and the core idea of Libratus (best Texas Hold'em AI) and AlphaZero (best AI algorithm). At last, presenting an efficient compiler – pypy.

2.2 Texas Hold'em Poker

According to Wikipedia (2018):

"Texas Hold'em Poker Game is a variation of poker game. Two cards, known as **hold cards**, are dealt face down to each game player and dealt five more **community cards** face up in three stages. These stages include a series of three cards ("**flop** stage"), followed by a single extra card ("**turn** stage" or "fourth street") and the last card ("**river** stage" or "fifth street"). Each player looks for the best five card poker hand from seven cards of five community cards and their hole cards. Players have betting options**, checking, calling, raising, or folding**. The betting rounds are performed before flop stage and after each subsequent deal.

"The game usually uses small and big **blind bets** - two players are forced to bet. **Antes** (all players who make mandatory donations) can be used in addition to blinds, especially at the later stage of the competition. The **dealer button** is used to represent the player in the dealer's location; the dealer button rotates clockwise after each hand, changing the position of the dealer and the blinds. The **small blind** is posted to the left side of the dealer by the player, usually half of the amount of the big blind. The **big blind**, posted by the player on the left side of the small blind, is the smallest bet. In tournament poker, the structure of blind/bet increases periodically as the game proceeds. After the completion of a bet, the next bet will start with the small blind." (En.wikipedia.org, 2018)

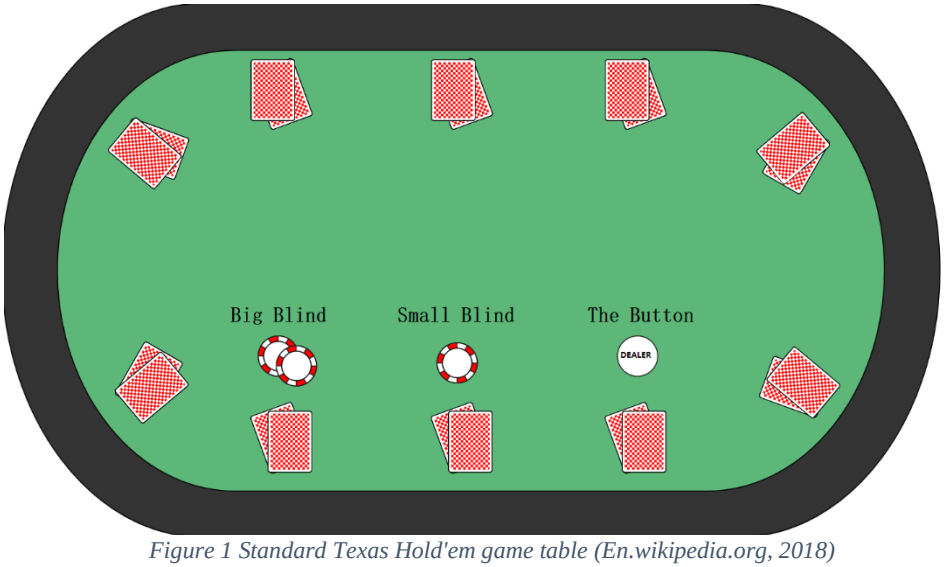

Figure 1 Standard Texas Hold'em game table (En.wikipedia.org, 2018)

2.2.1 Gaming Procedure

According to Wikipedia (2018):

"As the card shuffle, the game starts, each player has processed two cards face down, the player receives the first card in the small blind, and the player receives the last card on the button seat. (in most poker games, the deck is a standard 52 card without ghosts.) These cards are the player's hole or the card bag. These are the only cards accepted by each player individually. They will be disclosed at the time of the showdown, enabling the game to a final.

"The hand starts betting from a "**pre-flop**" stage, starting with the player on the left of the big blind (or the player on the left of the dealer, if not using the blinds) and continuing clockwise. Continue a round of betting until each player is folded, put in all chips, or matches the amount of input of all other active players. Please note that shutters are "alive" in the pre-flop round, which means that they count the number of blind players must contribute. If all players make their decision to the big blind position, the big blind can either check or raise.

"After the pre-flop round, it was assumed that at least two players were involved in the transaction. The dealer deals three community cards facing up. It is the second round of bets called "**flop**" round. This and all subsequent betting rounds start on the left side of the dealer, then proceed clockwise.

"After the end of the flop round, a single community card (called the **turn** or fourth street) is processed, followed by the third betting round. Finally, the last community card (called the **river** or fifth street) is then processed, and then fourth betting rounds and the **showdown**." (En.wikipedia.org, 2018)

2.2.2 The Showdown

According to Wikipedia (2018):

"If one player bet and all other players fold, the remaining players are granted the **pot** and do not need to display their hole cards. If two or more players stay at the end of the bet, the showdown takes place. When showing down, each player can play cards with seven cards, including their hold cards and five community cards. Players can use their hold cards, only one, or none, to form their last final hand. If more than one player shares the best hand, the pot is split evenly. Otherwise, the winner takes all." (En.wikipedia.org, 2018)

Standard hand ranking chart as shown in Figure 2 below.

Royal flush		A royal flush is an ace high straight flush. For example, A-K-Q-J-10 all of diamonds.
Straight flush		A straight flush is a five-card straight, all in the same suit. For example, 7-6-5-4-3 all of spades.
Four of a kind		Four of a kind, or quads, are four cards of equal value. For example, four jacks.
Full house		A full house contains a set (3) of cards of one value and a pair of another value. For example, Q-Q-Q-2-2.
Flush		A flush is any 5 cards, all of the same suit. For example, K-Q-9-6-3 all of diamonds.
Straight		Five cards of sequential value. Every possible straight will contain either a 5 or a 10. For example, 7-6-5-4-3 with different suits.
Three of a kind		Three cards of the same value. For example, three aces.
Two pairs		This is two cards of one value and another two cards of another value. For example, two jacks and two 8s.
Pair		One pair is two cards of the same rank. For example, two queens.
High card		The hand with the highest card(s) wins. If two or more players hold the highest card, a kicker comes into play (see below).

Figure 2 Standard hand ranking chart (Partypoker.com, 2018)

2.2.3 Kickers

In the event of a tie with four of a kind, three of a kind, two pairs, one pair, or highcards, a side card, or 'kicker', comes into play to decide who wins the pot.

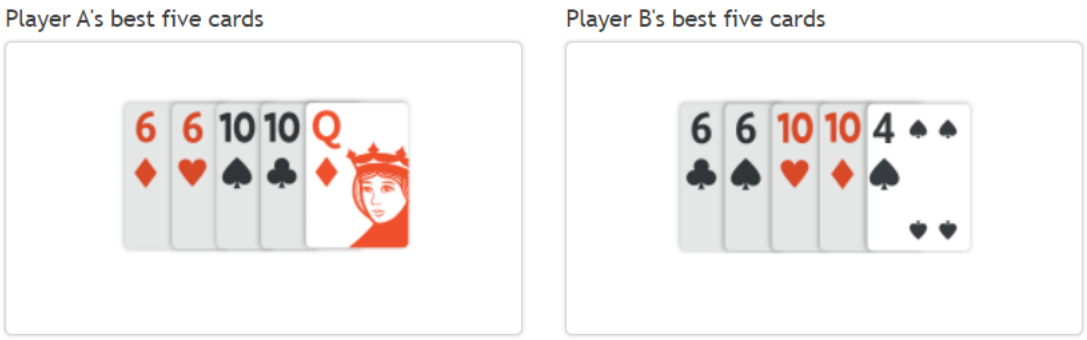

Figure 3 Kickers example. Player A wins the hand with kicker Q. If the kickers cannot decide one overall winner, those players involved are just going to have to share the pot. (Partypoker.com, 2018)

2.2.4 Technical Term

- **Pre-flop:** The first round with Hold cards before Community cards shows up.
- **Flop:** The second round with 3 Community cards
- **Turn:** The third round with 4 Community cards
- **River:** The fourth round with 5 Community cards
- **Showdown:** The settlement, each player displays their Hold cards
- **Hold cards:** Two cards face down for each player at the beginning
- **Community cards:** Several cards face up for common usage
- **Pot:** The total number of chips that everyone has already bet.
- **Stack:** Money of each player
- **Check:** If you do not need to Call, choose to give the decision to the next person.
- **Call:** Follow the previous Raise action to the same amount of money
- **Raise:** Raise an amount greater than previous Raise or Min-bet
- **Fold:** Give up your cards and wait for another game
- **All-in:** Raise all your stack
- **Button:** The player who makes the last action each round.
- **Under-the-gun(UTG):** The player who makes the first action at the beginning of a round.
- **Blinds:** A mandatory bet for players at the blind position to ensure that there is at least one number of pots.
- **Ante:** A mandatory bet for each player at the beginning
- **Min-bet:** The minimum amount of Raise
- **Kickers:** The side card to decide victory or defeat when comes to a tie.

2.2.5 Flow Chart

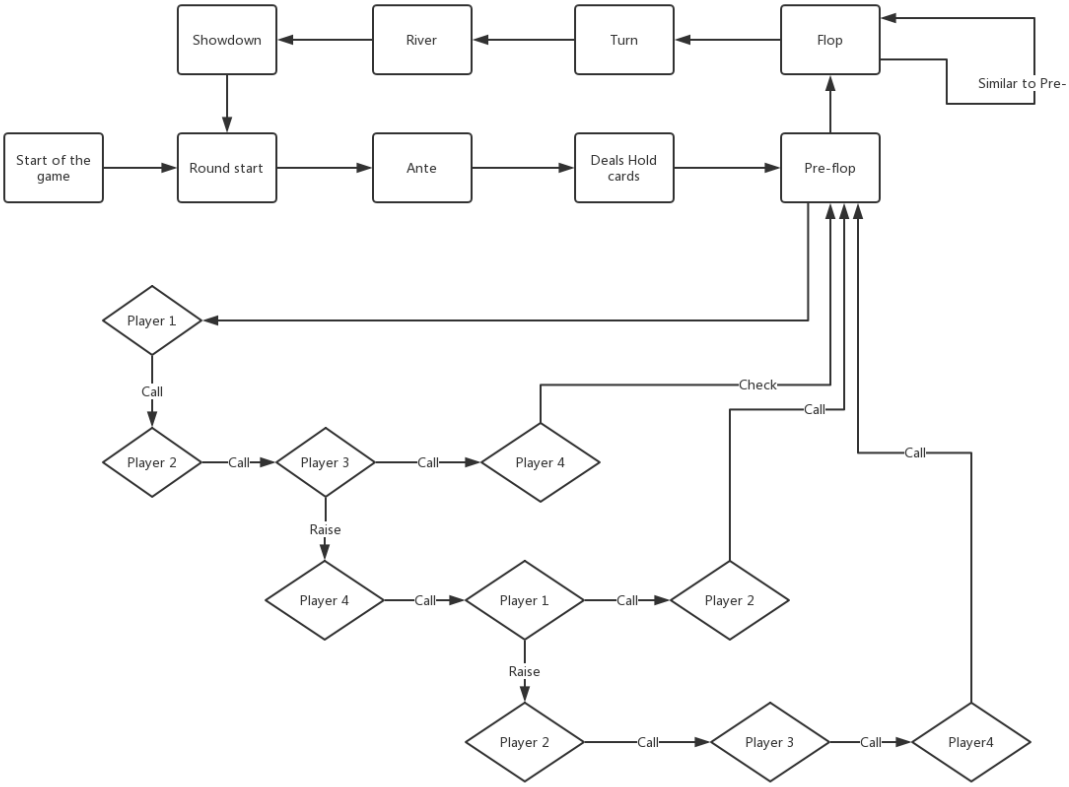

Figure 4 Flow Chart for standard Texas Hold'em Poker

2.3 Game with incomplete information

Playing games with computers is almost accompanied by the invention of computers. People have been using games as an experimental platform for Artificial Intelligence research all the time. Early work focuses on the chess game, the updated research is more complicated, such as football and poker. Research on poker and complex games have attracted increasing attention since the late 90s. (Wenkai, 2013) There are two characteristics of the early research objects. At first, all kinds of information about the opponent is known. For example, in a chess game, what is left of the opponent is clear, which we call "complete information", corresponding to "incomplete information"; Secondly, the next state of the whole game is completely determined by the current state and the current action. It does not make the next stage of the game unpredictable because of the randomness of the next card, which we call "deterministic," corresponding to "randomness". The differences between these features are as follows:

- Complete information and incomplete information: all the information in the game is visible to each player, and the information each player sees is the same. Such as chess, checkers, go and so on. In contrast, in the "incomplete information" game, there is a part of the information

that the opponent cannot see, of course, the opponent also has a part of the information that they cannot see. The representatives of these games are all types of poker and mahjong games. Take Texas Hold'em poker as an example; each player has two cards as " Hold cards", which is not seen by other rivals. This characteristic leads to the uncertainty of the player's overall state of the game, which adds to the complexity of the game.

- Deterministic and randomness: in a "deterministic" game, what is the next stage of the game is determined by the current state and the current action decision of the player. For example, in checkers, the next state is entirely determined by the current board state and the player's next plan, and the computer can fully simulate all the possible moves. But the game of randomness is not so simple. There are some random factors in such games. In Texas Hold'em, for example, what will happen is random in the next round of cards, which causes us to judge the next state of the game only based on the current state and the prediction of the next step of the player. **The existence of "randomness" causes the difficulty of modelling the whole game.**

2.4 Libratus

Libratus is the latest and most reliable AI of Texas Hold'em poker, which defeats professional players in No Limit Heads-up (one-on-one) game in 2017. Libratus consists mainly of three modules: an early game strategy called 'blueprint', training by abstract game to deals the situation of pre-flop and flop stage; a real-time subgame solving algorithm to simulate the turn and the river; and a 'learning' module to dynamically improve the early game strategy based on the previous game.

The first module calculates the abstraction of the game, it is smaller and easier to solve, and then calculates the general game strategy. This abstraction provides a detailed strategy for the early game of the game, but only an approximation in the game behind more parts of the game. "Heads-up No Limit Poker has 10^{161} decision points, so traversing the entire game tree even once is impossible. Precomputing a strategy for every decision point is infeasible for such a large game." (Brown and Sandholm, 2017).

Other modules do not affect the remaining project for some reason. Details could be found in Chapter 3.

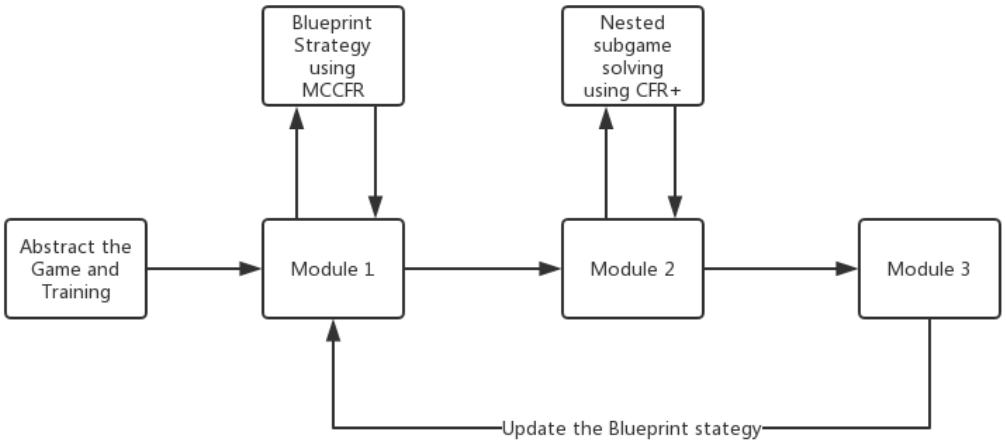

Figure 5 Basic Structure of Libratus

2.5 Pypy

According to its documentation: (Pypy Team, 2018)

"Pypy is a fast and compatible alternative implementation of Python language (2.7.13 and 3.5.3). It has several advantages and distinctive features:

"Speed: because of its immediate compiler, the Python program runs faster on Pypy. "If you want the code to run faster, you may only need to use PyPy." - Guido van Rossum (creator of Python)

"Compatibility: PyPy is highly compatible with existing Python code."(Pypy Team, 2018)

Chapter 3 – Modelling and Design

In this chapter I would explain my initial concepts for the project; the details of game abstraction and variations of rules; the structure of the program; the generations of AI and functions in code.

3.1 Program Concepts

Being motivated by the appearance of AlphaGo in 2015, I started to focus the development of game AI algorithms. Although I was not able to fully understand how it works or implement a similar program, it still attracts me deeply. Thus, I would select Texas Hold'em Poker as my final project – Poker AI and Go AI are both game algorithm, but multiplayer poker game is still a problem that has not yet been solved by Artificial Intelligence. In other words, I wanted to try something different from the previous study as a challenge for myself.

My original idea was to build a neural network for the entire game and let the computer calculate the parameters in it because I have learned a deep-learning module from Coursera by myself last year (2017). After reviewing the literature and have an understanding of the latest technology, I realise the concept is almost impossible at my present stage. According to Noam Brown, the leader of Libratus team from CMU, "Libratus does not use any knowledge from deep learning. We hope this helps people realise that AI is more important than deep learning. Deep learning itself is not enough to play poker games." (AI era, 2017). On the contrary, another AI for Texas Hold'em Poker in the early days – DeepStack – choose deep learning as the primary approach. DeepStack has never been proven to surpass the top-level AI, and Libratus beats the best Heads-up No Limit poker AI Baby Tartanian8 and winning the 2016 computer poker game. Therefore, the algorithm of DeepStack must have many precise aspects that can be learned, I still choose the Libratus as my model, and Libratus does not use deep learning.

Libratus use the variations of Counter-Factual Regret Minimization (CFR) algorithm to achieve different stage. The Blueprint stage uses the Monte Carlo CFR to train their original algorithm with an abstract model of game and CFR+ for real-time subgame solving. It also contains learning algorithm after each game, but the development team was not giving the detailed introduction. These three layers constitute the basic framework of my program.

AlphaZero is not to be ignored while mentioning the game AI algorithm, especially in the past half-century, there has been no breakthrough in gaming algorithm. In recent years, Deepmind team form Google has led the revolution of the algorithm community, especially the concept of AlphaZero has opened a field of training an algorithm without the requirement of domain knowledge (except the rules). Can I use the principle from AlphaZero to construct my poker AI? AlphaZero uses Monte Carlo Tree Search and Deep Learning as its main algorithm, but it needs 5000 TPU V1 to generate self-playing chess board. However, the most important thing is the algorithm for complete information game would not suit the situation of incomplete information

game. Whether how complex the game environment is, they would not be affected by unknown factors. The state of the game is discrete and predictable. Unlike the chess game, incomplete information game like poker, cannot be easily divided into some continues parts. The influence of multiple players also results in the exponential growth of the complexity of the whole environment. I have reasons to believe, Monte Carlo Tree Search may perform better than standard searching algorithm such as Alpha Beta pruning under the environment with complete information, Counter-Factual Regret Minimization is the state-of-the-art algorithm to deal with incomplete information game in the present. However, the methods from AlphaZero is still the most potential one in the world and might be evolved in the future to fit the broader environment.

Base on the arguments above, the implementation of domain knowledge is necessary for my project. In next part, I would discuss the simplify of rules and the underlying structure.

Two types of the program should be implemented. First is the game procedure for each round. In this program, the user can track each move made by each player and the private information for each player (Hold cards, Stack, Decision, Win rate). The second is a test environment which runs the whole game regularly and saves the result to different excel sheet for further evaluation. The test environment should be filled with several robots – bad robots who Call for all time; random robots with random decisions; standard robots with domain knowledge and fixed strategies and different generations of AI.

For the selection of programming language, Python is relatively simple among several languages I have learned, which makes it easier to focus on the logic of different algorithms instead programming itself. Since python become slow when the simulation depth increased, I decide to use Pypy to speed up the program.

No user interface is needed for this project. All the in-game process would be shown in Windows terminal. The option of adding a real player (waiting for user input at each round) was rejected because that makes the program impossible to be tested in thousands of rounds.

3.1.1 Variation of rules

In this part, I would specify the rules that changed to reduce the complexity. The original rules could be found in the background chapter before. More complicated information could be found in references and would not affect the game procedure. The method to the abstract game environment is derived from Mick West. (Mick, 2005)

1. Rounds are reduced from four to three due to the similarity of the last three stage (the Flop, the Turn and the River. The only difference from each stage is the number of the community cards).

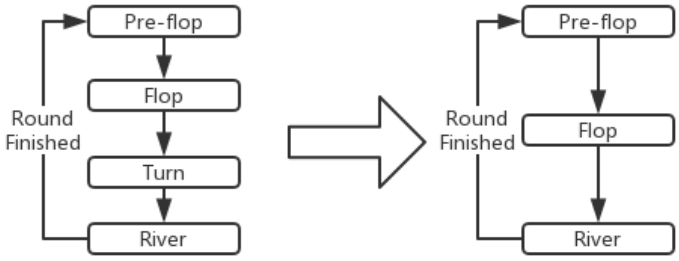

Figure 7 Reduce of Procedure

2. Suits are removed from this environment. Correspondingly the types of last cards are also reduced from ten to seven (removes Flush, Straight Flush and Royal Flush). At the beginning of modelling the game, I was trying to use a nested list "[[rank, suit], [rank, suit], [rank, suit]]" to store each hand. For instance, S for Spade, D for Diamond, C for Clubs and H for Hearts. The nested list for the chart below should be ['4h','4c','2s','Ad'] and in format of numbers: [[2,1],[2,2],[0,3],[12,4]]. During the development, nested list is inconvenient whether transforming format to string (for visualising) or to hex numbers (for computing strength, see sections below). However, that does not mean that the suits are abandoned forever, they could be implemented as an add-on function for further improvements.

Figure 6 Sample of Cards (Pokerstars.uk, 2018)

3. Typically, the maximum circle for Raise in each round is capped at three. In the real environment of my project, I found that the multiple Raise decision is not as easy as the initial concepts. It was more than just the recursive of the same function and should be treated as a new round at each. The maximum re-Raise decision is capped manually under this circumstance and still not working functionally in the end.

4. Reducing Blinds from two players to one player.

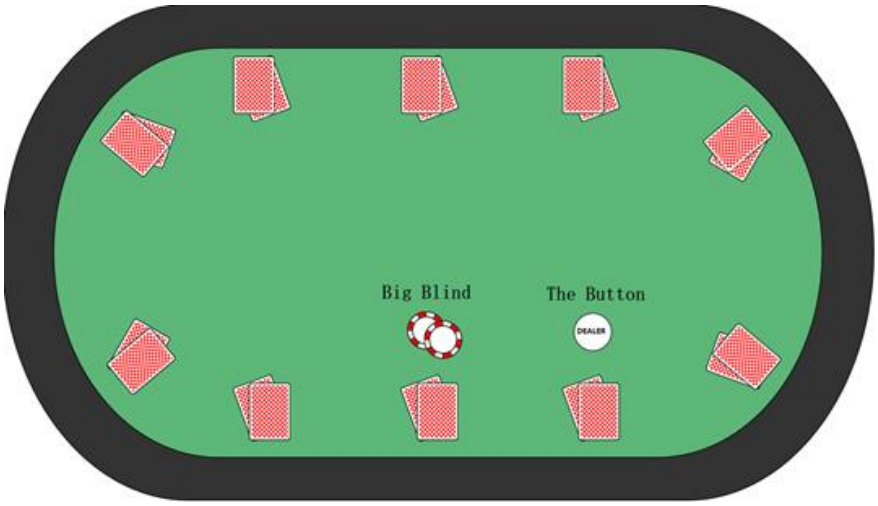

Figure 8 Simplified Game Table

5. The betting interval was to be set at some fixed levels but becoming unnecessary for this project for some reason. While attempting to implement CFR+ at the second layer, the abstract game is still much complicated for me. Thus, I change the CFR+ to another decision-making strategy based on domain knowledge. The latter method requires the specific betting amount to calculate probabilities rather than fixed interval and would be explained later.

3.1.2 Structure Design

According to the basic structure of Libratus and the experience from game environment abstraction, the new structure of my project is shown below.

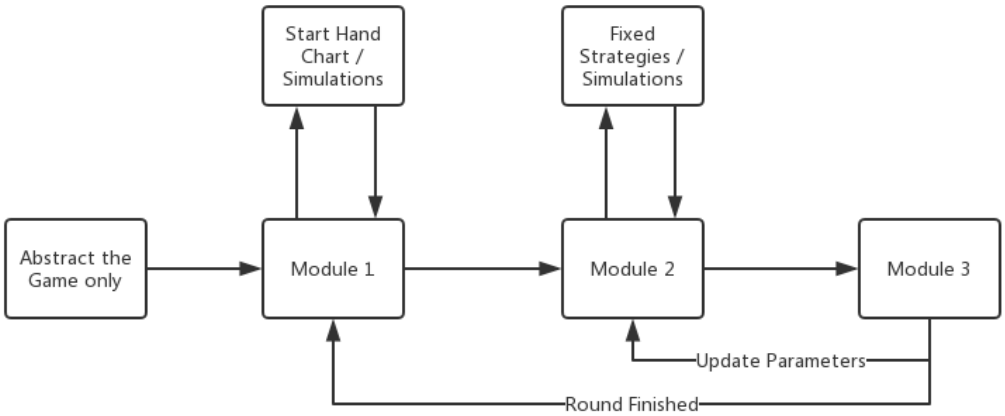

Figure 9 Structure of Standard Robot

The blueprint stage from Libratus is changed to a Start Hand Chart which enumerates the strength of one type of Hold cards and their reasonable decision correspondingly. For instance 'AA' is the best Hold cards a player can have. By the player's position (front, middle and back), and their opponents' previous movements,

the Start Hand Chart would give a relatively appropriate respond. The advantages of doing this are saving the time for the early game, because even if a game with incomplete information and explosion exponential, it still could be small at the beginning. Detailed Start Hand Chart should look like this:

Figure 10 Start Hand Chart for Beginners (PokerVIP, 2018)

The second stage was to be a CFR+ algorithm. However, the difficulties of successfully abstract the game environment are beyond my expectations, so I choose the method from Mick West as an available replacement. My supervisor suggested me to consider everything as a node for Monte Carlo Tree Search. It would reduce the total complexity in a way, but the difficulty is to combine those nodes and simulates them. You cannot predict what would happen next even if you are the banker. Moreover, this is the charm of the game with randomness.

The last stage is the learning stage with the principle of reinforcement learning. I have set up a threshold which determines two major decisions – Call and Raise. If the return-rate (will be explained below) is higher than the threshold, Raise several amounts. Otherwise, just Check or Call. If the game loses after been Raised by myself, then increase the threshold by 0.01 or reduce the contrast. Ignoring the bluff strategy, the result of this learning algorithm is out of my expectations – the AI shows different characteristics when the training interval changed. This is a windfall for the whole project. See section 3.3 and 3.4 for more details.

3.2 Data Structure

Inspired by the concept from Mick West, the types of data can be divided into four parts: (Mick, 2005)

- A ‘rank’ is an integer in the range of 0 to 12, where 0 = 2, 1 = 3, 12 = Ace.
- A ‘type’ is an integer in the range of 1 to 7, representing the type of poker hand, where 1 = highcards, 2 = one pair, 7 = quads.

- The ‘Hold Cards’ and ‘Community Cards’ are lists with different length. For instance, Hold Cards could be [9,12] and Community Cards could be [2,3,4] or [4,7,12,0,0]. Correspondingly, the ‘Best Five Hand’ is a sorted list of five best cards selected from Hold Cards and Community Cards.

- The ‘Hand Value’ is a Hex number representing the relative value or strength of any hand of cards. See the following examples:

Figure 11 Hex Number Data Structure of Hand Value (Mick, 2005)

“The hand value can conveniently be represented as a series of six nibbles, where the most significant nibble represents the Hand Type, then the next five nibbles represent the different ranks of the cards in the order of significance to the hand value.

“Example: **Ah Qd 4s Kh 8c** is a **highcard** handtype. So, the hand type nibble is set to 1. The remaining nibbles in the Hand Value are filled out with the ranks of the five cards in descending order. (**A, K, Q, 8, 4**), which translated into rank indices: **12,11,10,6,2** (or **C,B,A,6,2** in hexadecimal), and when combined with the hand type (1) in the high nibble, result in a hex number: **0x001CBA62**.” (Mick, 2005)

From the discussion above we can see that better hand always have a higher hand value, making determining the winning hand a simple comparison.

3.3 AI Specification and Generation

During the process of modelling the game environment, I realised that although every excellent AI article was focusing their algorithm of gaming process, all of them ignore the modelling part as well as agreed beforehand, which result in I underestimate the difficulty of game abstraction. I have to give up the attempts for a high-level algorithm like CFR or MCTS because of the delay of time. Instead, I choose a simple simulation to calculate the win rate considerably. In order to understand how the algorithm works, two terms should be noticed:

1. **Pot Odds:**

   “Pot odds are defined as the ratio between the size of the pot and the bet facing you. For example, if there is $4 in the pot and your opponent bets $1, you are being asked to pay one-fifth of the pot to have a chance of winning it.” (Pokerstarsschool.com, 2018)

2. **Return Rate:**

   “Rate of return is the "on average" proportion of how much you will multiply your bet by if you stay in hand.” (Mick, 2005)

Based on the fundamental knowledge, a standard Fold/Call/Raise Decision (FCR) can be implemented. For each betting rounds, the algorithm provides a very simply but useful mapping between Return Rate and FCR Decision.

```
if Return Rate < 0.8:
      95% Fold, 0% Call, 5% Bluff
elif Return Rate < 1.0:
      80% Fold, 5% Call, 15% Bluff
elif Return Rate < 1.3:
      0% Fold, 60% Call, 40% Raise
else (Return Rate >= 1.3):
      0% Fold, 30% Call, 70% Raise
```

This straightforward mapping between the Return Rate and FCR Decision form a reasonable and entertaining player. “They will tend to play strong hands; they will occasionally bluff, they will not scare easy if their hand is good, and they will abandon weak hands when raised and they will stick around on a reasonable chance of a straight draw, making for entertaining gameplay.” (Mick, 2005)

After solving the decision-making algorithm in each round, I had a concept on my own to set some of the values to a threshold, and self-improving after each game based on the result. This is motivated by the concepts of reinforcement learning. From my initial project concepts, I had mentioned that I was trying to implement Neural Network for the program and find the suitable parameters inside. My learning algorithm could be regarded as a reinforcement learning with only one parameter (could have more in further improvements). Using different training intervals (just like the gradient descent method for linear regression), I can build AI robot with different personalities.

```
...
elif Return Rate < threshold:
      0% Fold, 60% Call, 40% Raise
      israised = True
else (Return Rate >= threshold):
      0% Fold, 30% Call, 70% Raise
      israised = True
...
Round ends:
      ...
      if israised and win:
            threshold -= 0.01
      elif israised and lose:
            threshold += 0.01
      israised = False
      ...
```

How does the learning rate affect the personalities? In the situation of +0.01/-0.01, the expectations of win rate are 0.01 / (|-0.01| + |+0.01|) = 0.333, which means the

final threshold, for instance, 1.38, the AI would theoretically win one of three games under this circumstance. If I increase the learning rate to +0.01/-0.04, the expectation becomes 0.2 with 1.14 threshold value, which allows the AI player more hands with less robust cards, in other words, be more positive.

This extra function allows for directional training the AI to different strategies by reversing the training process – given reasonable expectation and derivate suitable learning rate.

3.4 Function and Code Explanation

In this section, I would thoroughly discuss each function in my program in details. The program consists two parts – main game environment and separated robot class. The game environment contains the gaming process; the function of multi-Raise behaviour and the printing and storing function. Each robot contains its unique strategies for different stage and the function to calculate hand strength.

Robot:

- **Initial and Clean:**

  Initialize the necessary variables, including two Dictionaries that are mapping the Rank and Types to numbers. Mapping would be needed when showing the outcomes.

- **Convert:**

  Convert the data structure to visualise list using mapping Dictionaries.

- **Start Hand (Standard Robot):**

```
AA,KK,QQ,AK,AQ,KQ,JJ,TT,99: Raise

88,77,66…(Cards of similar strength): Call

Other weak cards: Fold

Raise the amount of 3*minbet
```

- **MakeDecision (Standard Robot):**

  Divide the total number of players remain by three to determine the position of one player. See pseudocode below:

```
if position = back:
      if nobody raised before:
            Raise minbet
```

```
else:
      Check

calculate my hand strength

if my hands are stronger than two pairs:
      player at front: raise minbet
      player at middle: raise minbet * 2
      player at back: raise minbet * 3
```

- **RaiseDecision (standard):**

```
calculate the remaining amount that needs to be bet

calculate pot odds

if pot odds <= 0.2:
      Call
else:
      Fold
```

- **ReRaiseDecision:**

Robots would Fold when ReRaise situation happens. The reason for not using RaiseDecision function above iteratively is the gaming settlement sequence changes whenever a player decides to Raise. Even though, the game environment of this program is not fully functional when encountering repeated iteration and knock out a settlement. Results in the total money from each player is different from the beginning number but in an acceptable range (+-5%). This glitch does not affect the overall tendency, so I leave that into further studies. See evaluation chapter for more details.

- **FindMyBest:**

This is the essential function for every robot that takes a hand and returns a hand value. According to Mick West:

> "If you calculate ranks = (hearts | diamonds | clubs | spades) then the value ranks is a bit-field with a bit set for every card rank that you have at least one of. The number of bits set here is the number of **unique ranks** you have. We calculate the number of bits in each of hearts, diamonds, clubs and spades, and subtract the number of bits in the unique ranks, giving the number of **duplicated ranks**, to be used as the basis of determining what type of hand you have." (Mick, 2005)

Although there should be some existing methods to calculate the best hand, I would instead figure it out by myself as a challenge. The function may not be the optimal one but works correctly as well.

There are three essential boundaries for this function: the length of **unique ranks**; the **length** of the original ranks and the **key** cards which stores the ranks more than one. I would use a flow chart to simplify the structure:

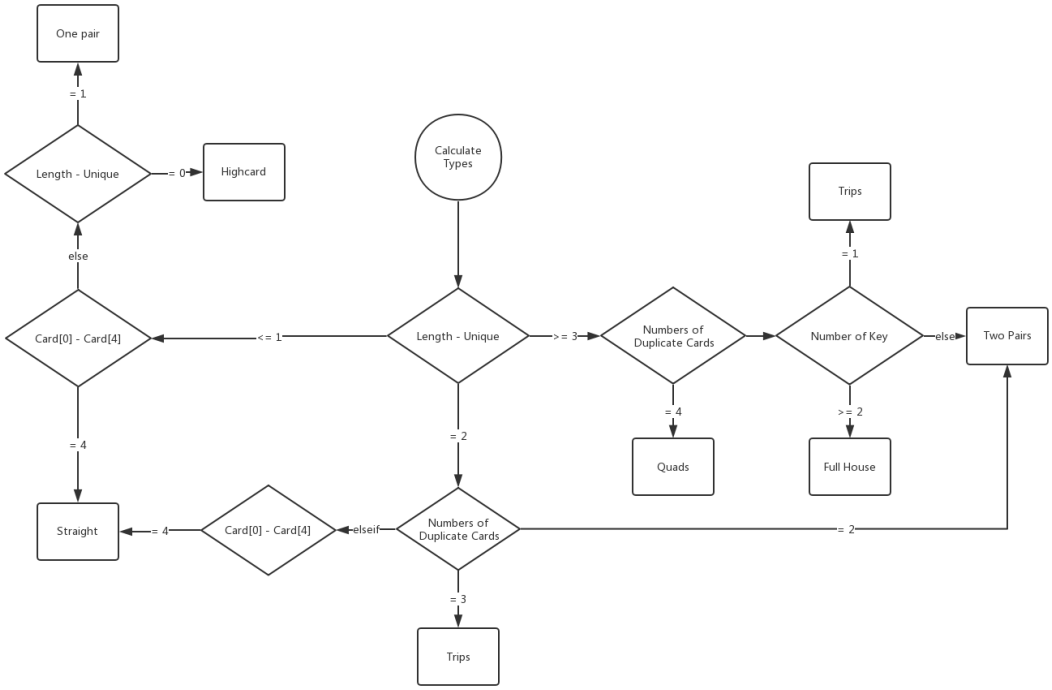

Figure 12 Types Calculation

After calculating the type of my best hands, the remaining function would pick 5 cards from the original 7 cards according to its type, then return a list and its strength as hex number.

```
if Quads:
      return [key,key,key,key,best card except key]
if full house:
      return [key1,key1,key1,key2,key2]
if straight:
      return [key,key-1,key-2,key-3,key-4]
if trips:
      return [key,key,key,best card,second best card]
if two pairs:
      return [key1,key1,key2,key2,best card]
if one pair:
      return [key,key,best card,second best card,third best card]
if highcard:
      return [best 5 cards]
```

and the strength function at last:

```
strength = hex(types * 16 ** 5 + cards[0] * 16 ** 4 + cards[1] * 16 **
3 + cards[2] * 16 **2 + cards[3] * 16 + cards[4])
```

- **SimulationOutside:**

  Given the Hold Cards, Community Cards and the number of opponents remaining, this function can simulate the win rate in this situation. The strength of a hand is between a floating-point number of 0.0 (a certain loss) and 1.0 (a certain win). For example, an HS of 0.33 means you have a 33% chance of winning.

  The easiest and most flexible way to calculate HS is to simulate the progress of the game several times and calculate the number of times you win. Suppose you simulate the game 1000 times, and in the simulation, you win 423 games, then you have a high degree of certainty, it has 423/1000 or 0.423 approximate HS.

  The process of the game simulation is very simple:

```
      Create a new deck and shuffle
      Remove the Hold Cards and Community Cards from the deck
      Deals cards to each opponent
      for range(numbers of simulation):
            if len(Community Cards) < 5:
                  deals the remaining Community Cards

            Showdown and see who has the best hand

            if win:
                  win += 1
      return win/numbers of simulation
```

  The simulation function can be applied on every decision-making stage (Pre-flop, Flop or River). To be more precise, we must allow our simulations to run if people drop off the hole cards below a certain threshold. In practice, determine whether the player stays in the simulation is a probability function of their hand strength, their table location, their stack size, the blind size, and their previous behaviour. Moreover, the result of the simulation could be counterintuitive sometimes. Considering the real game environment allows the player to fold at some stage, the real win rate is higher than the simulation. This reduces the ‘enthusiasm’ of AI player but improves the performance by some aspect, because it always considering the worst situation.

Game Environment:

Game environment program is the “User Interface” that controls the overall procedures of the game. The difficulties of this are the special cases for different AI and the various Raise settlement. Only the essential function would be discussed later.

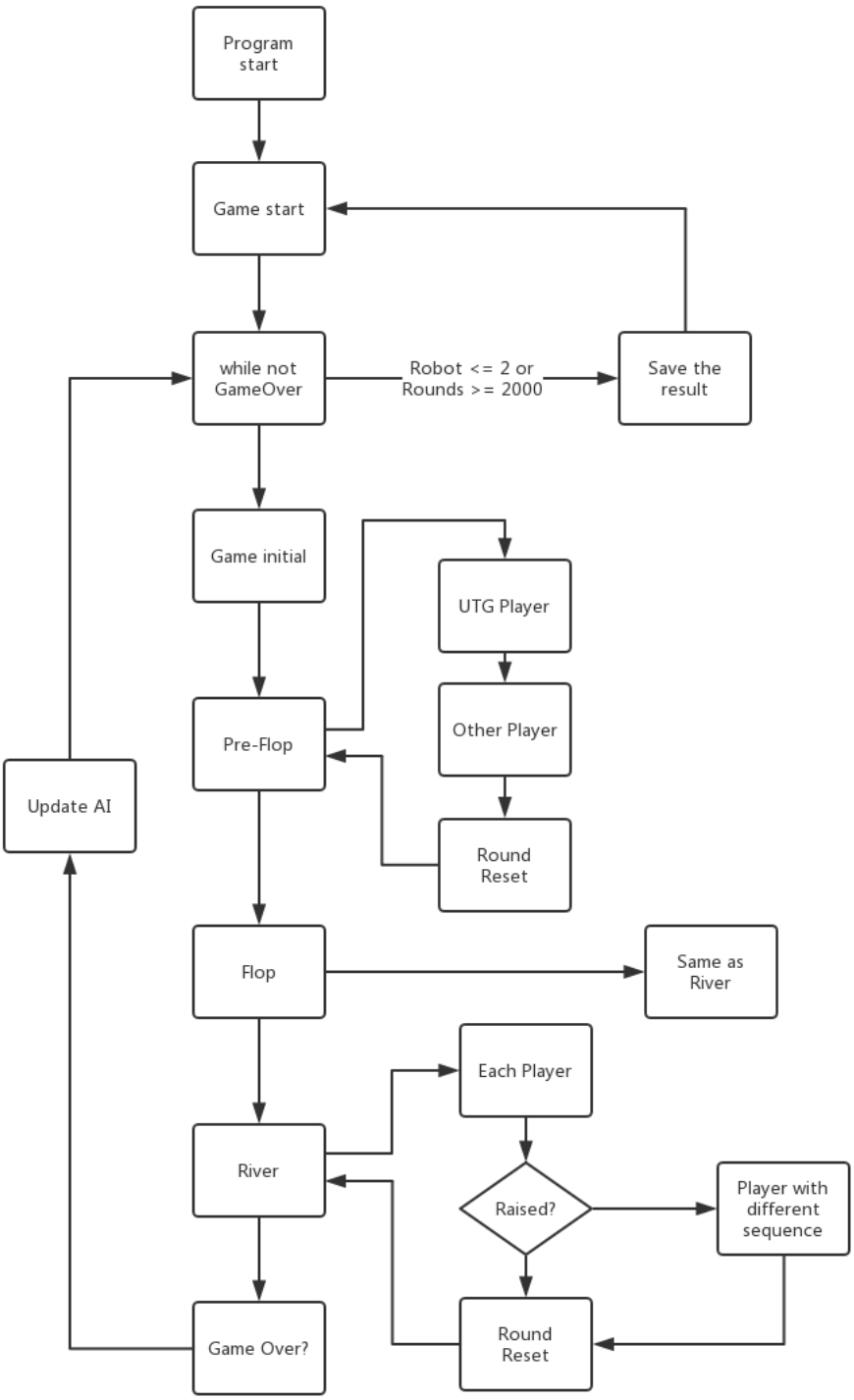

Figure 13 Total Game Procedure

- **Start:**

In Texas Hold'em, the betting sequence is not in a fixed order. There are two methods to solve this problem: One is creating a list that stores the moving sequence and updates the position each round. This kind of method is becoming complicated when multiple Raise happens because the list is hard to iterate itself. When I tried to change the list within the iterate circle, there would be several bugs saying, 'List index out of range'. The second approach is to separate the decision procedure and leave the part that can be iterated, regarding the remaining part as special cases. This would increase the complexity of the program and make it hard to modulate but works fine under presented requirements.

Figure 13 above shows the main structure of start function.

- **Test1 and Test2 (function of multiple Raise):**

```
if decision != raise:
      continue
else:
      self.israised = True

if self.israised:
      from player i+1 to player i-1: (this is another loop)
            self.test2() (Here should be test1 itself)

self.israised = False
```

Test1 function deals with the first round of Raise decision. If there was no ReRaise decision then pass, otherwise, restart the iterate from the next player to the player behind. In the Test2 function, I changed their decision manually to 'Fold' because I cannot handle another mutation of looping.

- **PrintData:**

This function belongs to the test process. I choose xlsxwriter module to store the stack tendency in excel sheet for further evaluation. I have provided two kinds of printing strategies. The first one is an overall estimate for each robot in only one game. The other is focusing one robot and drawing average trend map.

Chapter 4 - Experiment and Result

In this chapter, I would explain the test and evaluation process I have made. Starting from the test environment and different types of robots; the strength analysis for each robot; and the difficulties and unexpectable bugs I have encountered.

4.1 Test environment

In real life, a single game table of Texas Hold'em takes about 30 minutes on average. Considering the randomness in each game, it would take at least ten games to get a clear picture of the quality of an algorithm and a hundred to be sure. In this situation, automated testing is required by setting up a game environment with different types of opponents. Motivated by the supervisor, the testing environment should contain some regular robot, some bad robot (a player that always made bad moves) and some random player as well. In my early concept, eight different robots should be implemented:

- **Fold robot:** a player that fold all the time even with 'AA'.
- **Call robot:** a player that would call all the time.
- **Random robot:** a player with random decisions.
- **All-in robot:** a player that would All-in all the time.
- **Standard robot:** using Start Hand Chart and fixed functions.
- **AI V1:** using simulations without learning.
- **AI V2:**can update the parameter during the play but cannot save the parameter to next game.
- **AI V3:** the previous threshold can be passed during initialize.

When there were only a few robots left at the late game, the average stack for each player is increased, which made the influence of blinds and ante became smaller. This would make the late game environment be different from the real game. I decided to increase the number of blinds and ante dynamically, based on the number of remaining player. But the truth is the opposite of what I expected. The dynamic betting amount increases the randomness of the game and messes up the result. On the alternative, I use round counter and player counter to stop the game at a reasonable time. At the meanwhile, Random robot and All-in robot also increase the randomness of the early game. I need a bad player with stable status, so I choose Call robot as a bad player and the Standard robot as a regular player. (Full code list could be found in Appendix). Due to the shortage of time, AI V3 was failed to achieve. In the next section, I would run through the strength analysis among the remaining robot.

4.2 Strength Analysis

Test1: 1 Standard robot vs 7 Call robots:

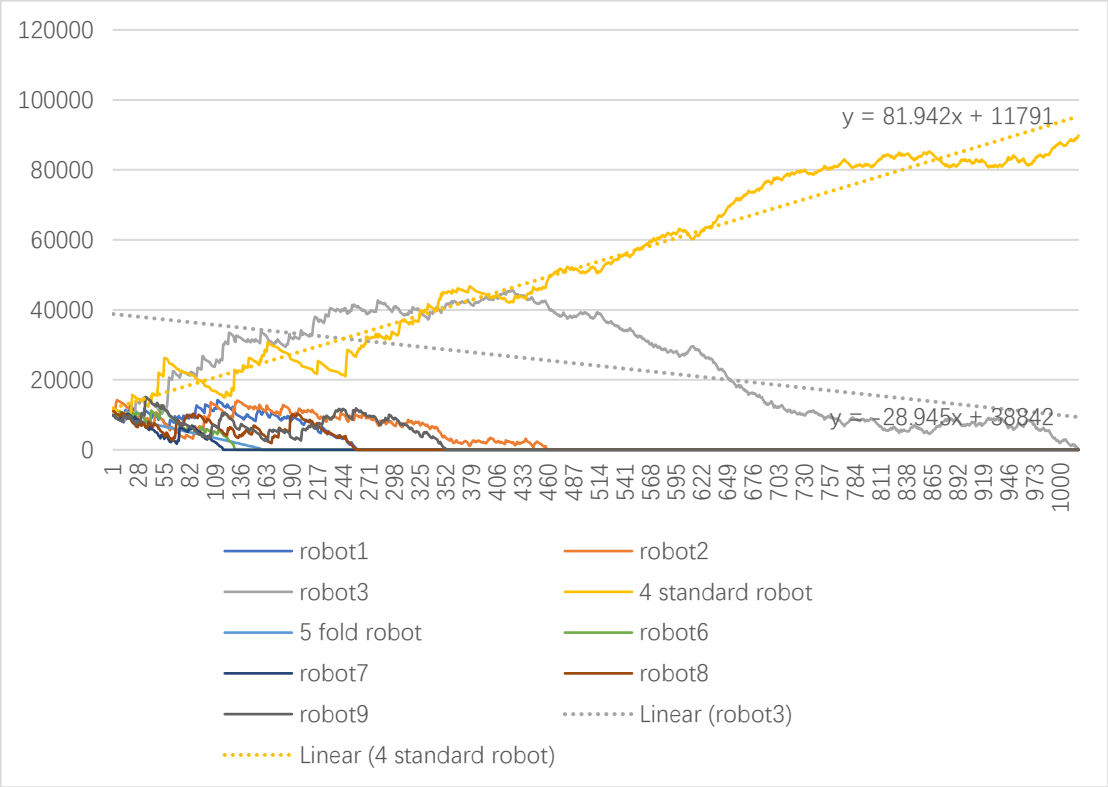

Figure 14 1 Standard robot vs 7 Call robots

Figure 14 and Figure 15 indicates the trend map of the stack for each robot in 1000 rounds of one typical game. We can clearly see that the Standard robot performs better than Call robot. With the decrease of the opponent, the Standard robot behaves better and still be competitive in multiple player environments. The gradient for the Standard robot is +81.492 when the average gradient of Call robot is -28.95.

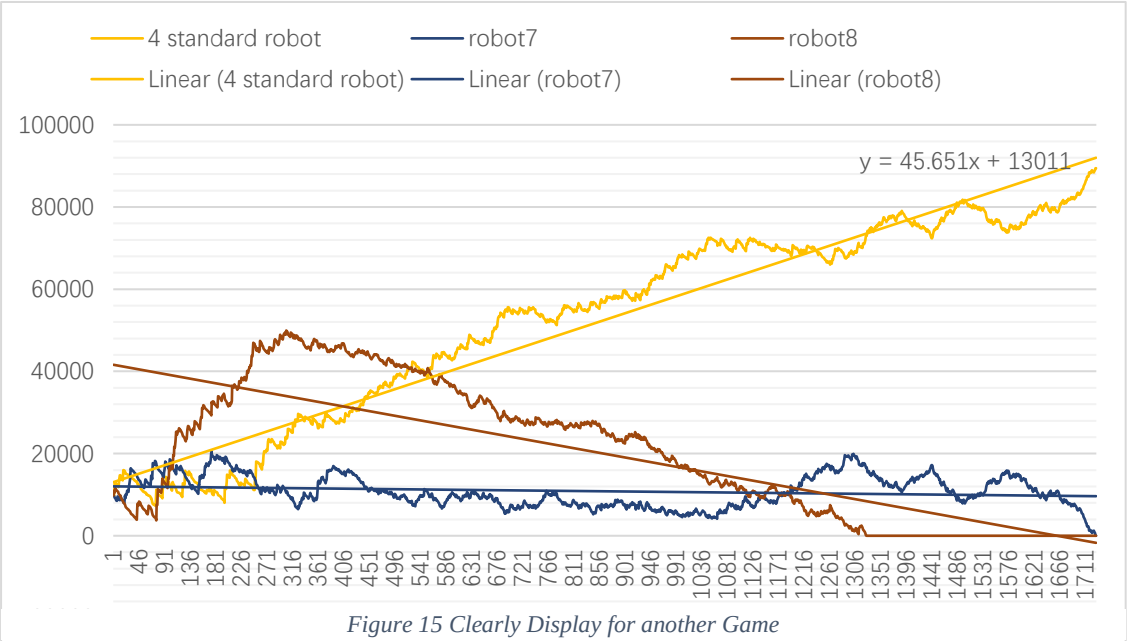

Figure 15 Clearly Display for another Game

Afterwards is the vertical comparison between Call robot and Standard robot. The average slope for the Standard robot is +70. (Figure 16) On the contrast the most of the Call robot performs worse at the beginning, although some of the Call robots wins by luck at the early game, it still cannot defeat the Standard robot just by luck if the sample size is large enough.

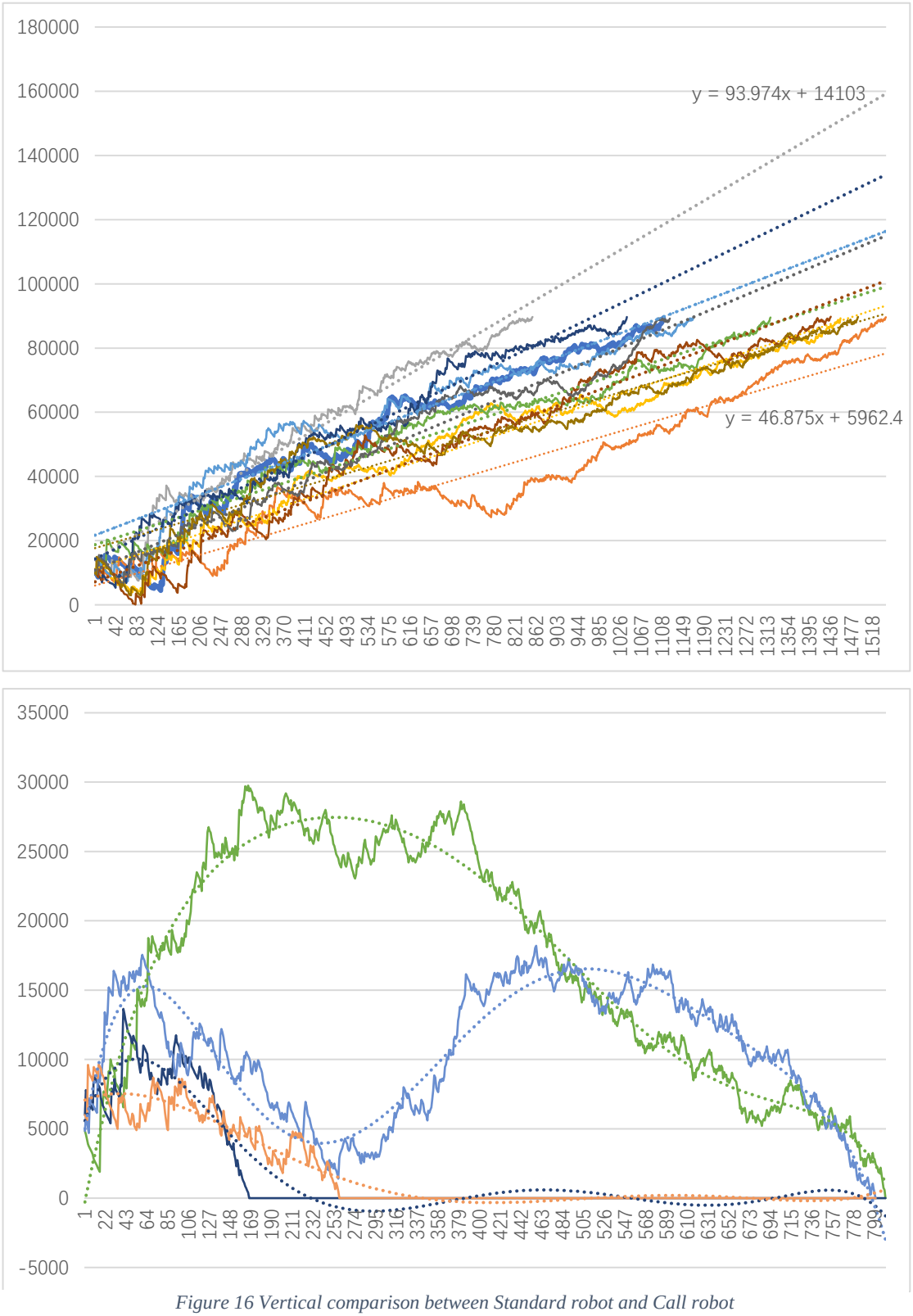

Figure 16 Vertical comparison between Standard robot and Call robot

Test2: Standard robots against each other:

In theory, the performance of each robot should be at the same level when the same type of robot against each other. If the variation of the same algorithm is more significant than acceptable level, then the algorithm should have some functional issues and not suited for the regular algorithm. From the chart below we can see that except the last one, other robots indicate stable performance with less variation.

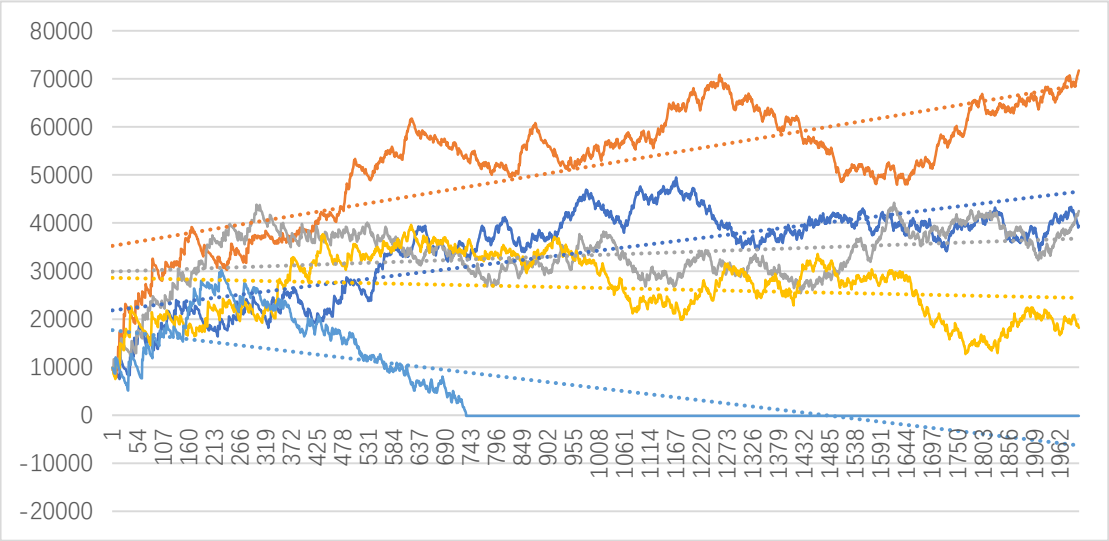

Figure 17 Standard robot against each other

Test3: 3 Standard robots vs 3 AI V1

After the determination of game environment, I started to test the performance for AI V1. Theoretically, even though the AI is just using simple simulations, it should still stronger than Standard AI. Thus, there is no need to implement a test between Call robot and AI V1. But the initial result is beyond my expectations. The Standard robot performs better than AI V1 this time. (Figure 18)

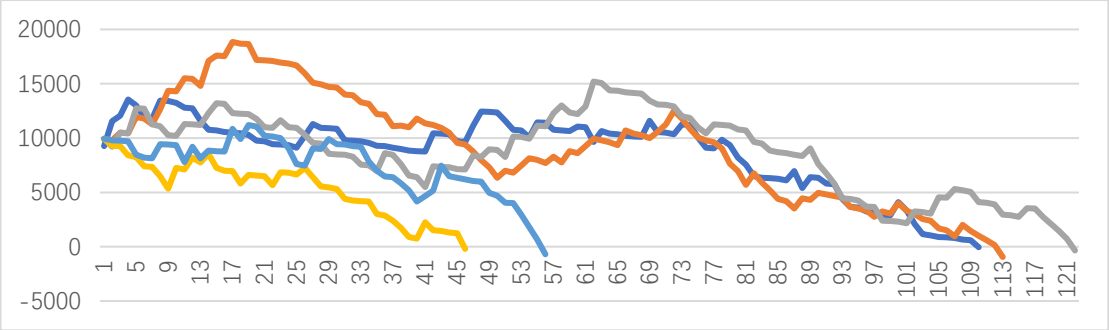

Figure 18 AI V1 was lost at the beginning

Tracing back to the game process, I have realized that the issues might be originated by the decision of 'Bluffing'. As I mentioned before, the Standard robot is relatively enthusiastic and would not be easily scared if it has good hands. Saying nothing of Call robot which would call all the time. There seems to be no better way except removing the bluffing decision, which makes the AI less human-like but fit the environment. Figure 19 shows the changes after disabling bluffing.

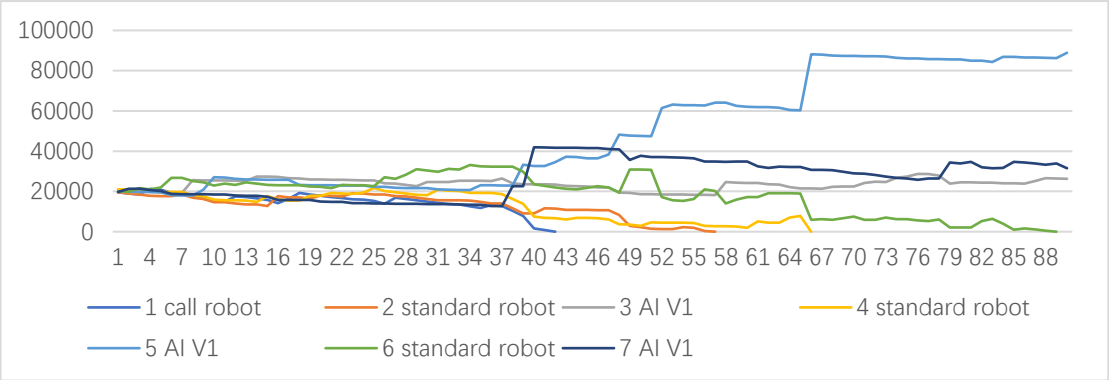

Figure 19 Improved AI V1

The adjustment is a success for this environment, and the AI V1 beats the Standard robots easily. (All of the AI V1 were alive when other robots were out of competition)

Test4: Final test

The final step is to run the test environment entirely. For the last evaluation, I use 1 Call robot, 3 Standard robots, 2 AI V1 and 1 AI V2 to estimate the overall strength of each algorithm. Unlike test1 and test2, test3 and test4 reduce the total number of rounds for each game, which indicated the AI is more aggressive than a regular robot. This is a success by some means because my initial concept was not to build an algorithm that the only defence. (Speak of this, the success of Libratus is because it uses a complete defence mode to beat a human player because the human will make mistakes but the machine would not.) From the diagram below, the performance of AI V2 is relatively stable even in a multiplayer environment. Unlike other robots, AI V2 seems capable of seeking the right opportunity to earn a lot from one round and can stop loss if the situation is not suitable for itself.

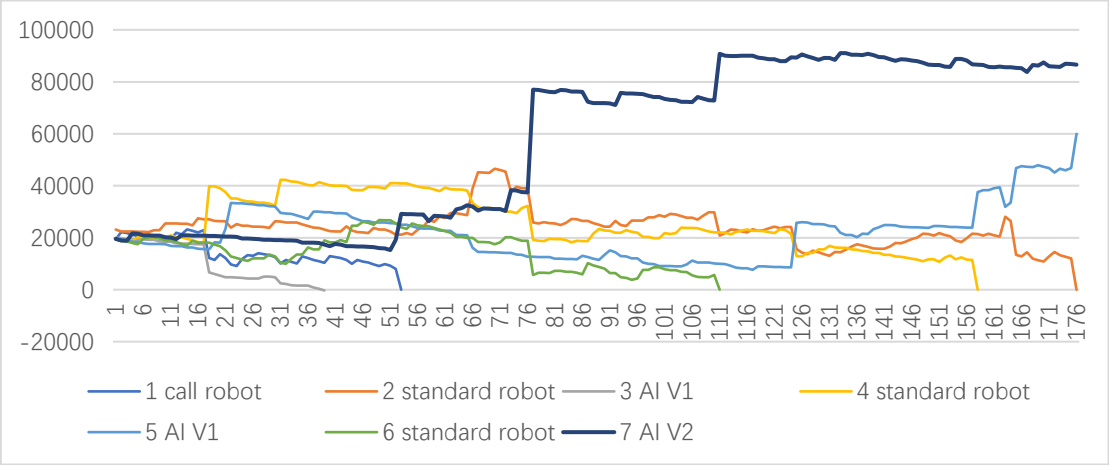

Figure 20 Samples of final test environment

For the strength comparison result, **AI V2 >= AI V1 > Standard Robot > Call Robot**.

Test5: Reinforcement learning for different personalities

The last part of the test process is the reinforcement learning. Due to the incomplete of AI V3, the automatic simulation is impossible if the updated parameter cannot pass through games. Thus, I have run the simulation manually for several times. The outcome may not be perfect, but the trends are worth mentioning.

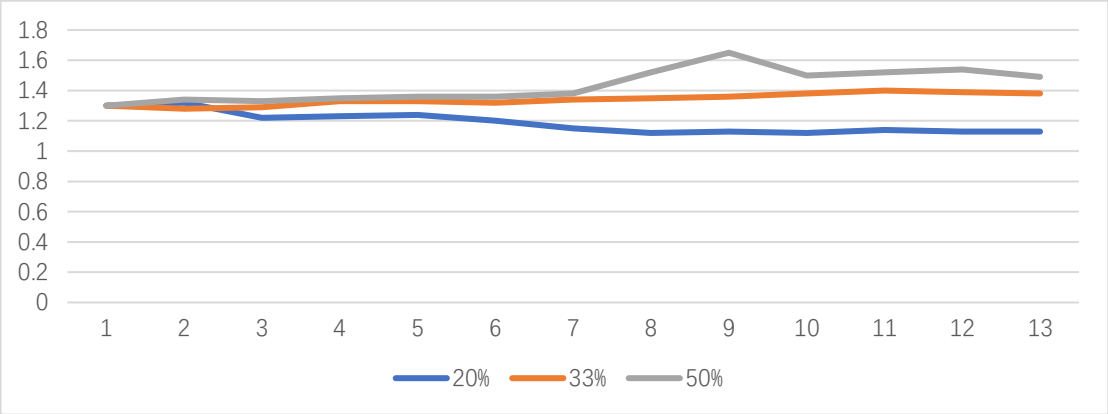

Figure 21 Threshold under different expectations

Under the expectations of 20% (-0.04 / +0.01), the threshold becomes smaller, and AI would play more rounds with mid-strength cards and looking for a draw. The threshold number for 20%, 33%, 50% expectations under test4 environment should be around **1.18, 1.38 and 1.5**.

4.3 Difficulties and Bugs

During the development of a new project, challenges and mistakes are inevitable, like the multiple Raise decision problem and the bluffing problem I mentioned before. Besides, there were also some interesting bugs but do no harm to the program.

- In the beginning, the type value for highcards is 0 (now is 1). That was because the program would ignore the type nibble and miscalculate the hand value, result in the highcards was the best type among all. Theoretically, 0 can be calculated as a type value but failed. The reason for that remaining mystery.

- Sometimes I would receive an error report says,' Cannot divide by Zero' when the algorithm calculates the rate of return. This seems to be a systematic glitch when attempting float number division. Problem fixed by adding

```
from __future__ import division
```

to the top of the code.

- Another systematic error is caused by windows terminal itself. During the debug process, the terminal would randomly drop the [IOERROR] signal when the print function was used rapidly. Pypy is free from this problem.

- Texas Hold'em poker is a zero-sum game theoretically. Which means wealth would not generate or disappear by itself. Whether how many times the funds

have been transferred, the total amount should be the same. But the settlement of multiple Raise decision was not implemented well and will bring in or take away money from the whole game sometimes. It is difficult to debug multiple iterations by substep, So I have just controlled the deviation to an acceptable level as it would not affect the overall performance.

The difficult part of this project was to be the implementation of high-level AI algorithms. Due to the underestimate of the complexity of game construction, the difficulties changed into debugging and testing.

4.4 Result

The comparison between different algorithms is successful. Although the simulation algorithm is still a simple one with only a few high-level knowledge, the other improvement could consider as the extra modules for further study. The game environment is playable and can be tested automatically. Here's my conclusion based on the experience of development:

One of the problems with existing program is the environment. AI V2 was trained on the environment of particular opponents. Once the environment changes, the threshold or strategies would not perform as well as before. In order to get a universal variable, it must adjust itself to the new environment, and the environment keeps changing as well. Thus, the hypothesis of having a global variable which solves all of the problems is wrong. The reasonable way of constructing a broad adaption algorithm is a learning algorithm and update itself all the time.

Although AI V2 performs better than all the other algorithms, the achievement of this project is insignificant for incomplete information game with multiple players. Libratus is the best Texas Hold'em AI in the world at present but still uses plenty of domain knowledge. I have reasons to believe that Texas Hold'em game is not a game that can simply conquer by math or by computational ability, because it's not a game that player against the system, it's a game that player against each other. In other words, one's victory is always built on other's failure. But that doesn't mean they are not a good player.

Compared to other poker game, Texas Hold'em needs more mathematical knowledge to get a higher grade. But there is another essential part that cannot be ignored – psychology. The best player would not focus what they have already. They can win the game by 'reading' their opponents' behaviour and make judgements – I got a weak hand, but I'm stronger than him.

In the end, I believe there is plenty of room to improve the existing algorithms. Furthermore, the direction for further improvements should focus on opponent modelling and a new algorithm for incomplete information game.

Chapter 5 - Conclusion and Further Work

5.1 Summarize

Tracing back to the initial project concepts, I was trying to implement Neural Network to the model of Texas Hold'em game. Throughout the research stage, I realised that Neural Network might not be suitable for a game with incomplete information. Then I changed my objects to applying different high-level algorithms like CFR+ or MCTS and make a comparison. Due to the complexity of abstracting the game environment, the project ends up achieving a simple algorithm with slight knowledge of reinforcement learning. My anticipation was great, but the reality reflected that I had underestimated the difficulty thoroughly. The difficulties are embodied in two aspects: Game with incomplete information and multiple player games have been the most complicated questions for decades. It is suitable to learn something instead of trying something new at present. Secondly, it was better to develop a supporting algorithm instead of constructing the whole game environment.

During the development process, I have learned that never downgrade a topic that you were not familiar. Knowing the limits of your own, focusing on the existing implementations and scheduling the time more flexible.

5.2 Further Study

For the algorithm, several improvements could be made. Opponent modelling, personality modelling, probabilistic search, the implementation of game theory and Nash Equilibrium are considerable. From the project aspect, next time I would try a simpler game with complete information such as simplified chess game, to focus on the algorithm rather than the game environment. The standard chess game is suitable for score calculation and generates the self-playing model.

Chapter 6 – Bibliography

- En.wikipedia.org. (2018). Texas hold 'em. [online] Available at: https://en.wikipedia.org/wiki/Texas_hold_%27em [Accessed 21 Apr. 2018].

- Partypoker.com. (2018). Poker Hands | Official Poker Hand Rankings | partypoker.com. [online] Available at: https://www.partypoker.com/how-to-play/hand-rankings.html [Accessed 21 Apr. 2018].

- Partypoker.com. (2018). How to play Texas Hold'em Poker - Hands and Rules | partypoker. [online] Available at: https://www.partypoker.com/how-to-play/texas-holdem.html [Accessed 21 Apr. 2018].

- Wenkai, L. (2013). Hand Strength Forecast Model and NPC Design for Texas Hold'em Poker (Master's thesis, Nanjing University).

- Brown, N. and Sandholm, T. (2017). Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. Science, 359(6374), pp.418-424.

- Pypy Team. (2018). PyPy - Welcome to PyPy. [online] Pypy.org. Available at: https://pypy.org/ [Accessed 22 Apr. 2018].

- AI era, (2017). Can AlphaZero beats Libratus? [online] Available at: https://zhuanlan.zhihu.com/p/32154212 [Accessed 23 Apr. 2018].

- Mick, W. (2005). Programming Poker AI. [online] Available at: http://cowboyprogramming.com/2007/01/04/programming-poker-ai/ [Accessed 23 Apr. 2018].

- Pokerstars.uk. (2018). Poker Hands Order - Poker Hand Rankings. [online] Available at: https://www.pokerstars.uk/poker/games/rules/hand-rankings/?no_redirect=1 [Accessed 23 Apr. 2018].

- PokerVIP, J. (2018). Poker Starting Hand Charts | PokerVIP. [online] PokerVIP. Available at: https://www.pokervip.com/strategy-articles/texas-hold-em-no-limit-beginner/starting-hand-charts [Accessed 23 Apr. 2018].

- Pokerstarsschool.com. (2018). Pot Odds and Expected Value. [online] Available at: https://www.pokerstarsschool.com/article/Pot-odds-and-expected-value [Accessed 24 Apr. 2018].

- Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. [online] Arxiv.org. Available at: https://arxiv.org/abs/1712.01815 [Accessed 20 Apr. 2018].

- Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. and Bowling, M. (2017). DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. Science, 356(6337), pp.508-513.

- Bowling, M., Burch, N., Johanson, M. and Tammelin, O. (2017). Heads-up limit hold'em poker is solved. Communications of the ACM, 60(11), pp.81-88.

- Lorentz, R. (2016). Using evaluation functions in Monte-Carlo Tree Search. Theoretical Computer Science, 644, pp.106-113.

- Pęski, M. (2014). Repeated games with incomplete information and discounting. Theoretical Economics, 9(3), pp.651-694.

- Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T. and Szafron, D., (2003). Approximating game-theoretic optimal strategies for full-scale poker. In IJCAI, pp. 661-668.

- Zinkevich, M., Johanson, M., Bowling, M. and Piccione, C., (2008). Regret minimization in games with incomplete information. In Advances in neural information processing systems, pp.1729-1736.

- **Game.py**

```
import CallRobot
import Dealer
#import FoldRobot
import StandardRobot
#import RandomRobot
import AI_V1
import AI_V2
import xlsxwriter
import time

class test(object):

    def __init__(self):

        self.data = [[],[],[],[],[],[],[],[]]
        self.counter = 0

        self.minbet = 100
        self.ante = 50
        self.initStack = 20000
        self.robot1 = CallRobot.robot('1 call robot',
self.initStack, self.minbet)
        self.robot2 = StandardRobot.robot(
            '2 standard robot', self.initStack, self.minbet)
        self.robot3 = AI_V1.robot('3 AI V1', self.initStack,
self.minbet)
        self.robot4 = StandardRobot.robot(
            '4 standard robot', self.initStack, self.minbet)
        self.robot5 = AI_V1.robot('5 AI V1', self.initStack,
self.minbet)
        self.robot6 = StandardRobot.robot(
            '6 standard robot', self.initStack, self.minbet)
        self.robot7 = AI_V2.robot('7 AI V2', self.initStack,
self.minbet)
        self.initial_game_list = [
            self.robot1, self.robot2,
self.robot3,self.robot4,self.robot5,self.robot6,self.robot7]
        self.fixed_game_list = [self.robot1,
self.robot2,self.robot3,self.robot4,self.robot5,self.robot6,self.
robot7]
        self.GameEnd = False

```

```
    def test2(self, i):

        self.raiseAmount = i.betAmount
        self.BET = self.raiseAmount
        self.POT += self.BET
        self.israised = True
        for p in self.move_list[self.move_list.index(i)+1:]:

            dec = p.reRaiseDecision()
            if dec == 'fold':
                self.in_game_list.remove(p)
                self.move_list.remove(p)
                return False

        for p in self.move_list[:self.move_list.index(i)]:

            dec = p.reRaiseDecision()
            if dec == 'fold':
                self.in_game_list.remove(p)
                self.move_list.remove(p)
                return False

    def test1(self, deci, i):

        if deci == 'check':
            self.BET = 0
        elif deci == 'fold':
            self.in_game_list.remove(i)
            self.move_list.remove(i)
        elif deci == 'call':
            self.BET = i.betAmount
            self.POT += self.BET
        elif deci == 'raise':
            self.BET = i.betAmount
            self.raiseAmount = i.betAmount
            self.POT += self.BET
            i.isRaise = True
            self.israised = True
        elif deci == 'All-in':
            self.raiseAmount = i.betAmount - i.previousBet
            self.BET = i.betAmount
            self.POT += self.raiseAmount
            i.isRaise = True
            self.israised = True

        if i.isRaise:

            for p in self.move_list[self.move_list.index(i)+1:]:
```

```
•
•                   if p.name != '3 AI V1' and p.name != '5 AI V1'
    and p.name != '7 AI V2':
•                       dec = p.raiseDecision(self.BET, self.POT)
•                   else:
•                       dec = p.raiseDecision(self.BET, self.POT,
    len(self.move_list)-1)
•
•                   if dec == 'fold':
•                       self.in_game_list.remove(p)
•                       self.move_list.remove(p)
•                   elif dec == 'call':
•                       self.POT += p.needToBet
•                       # p.money -= p.needToBet # REMEMBER THIS
    BUG!!!
•                   elif dec == 're-raise':
•                       print 'xxx'
•                   elif dec == 'All-in':
•                       self.test2(p)
•                       i.isRaise = False
•                       return False
•
•               for p in self.move_list[:self.move_list.index(i)]:
•
•                   if p.name != '3 AI V1' and p.name != '5 AI V1'
    and p.name != '7 AI V2':
•                       dec = p.raiseDecision(self.BET, self.POT)
•                   else:
•                       dec = p.raiseDecision(self.BET, self.POT,
    len(self.move_list)-1)
•
•                   if dec == 'fold':
•                       self.in_game_list.remove(p)
•                       self.move_list.remove(p)
•                   elif dec == 'call':
•                       self.POT += p.needToBet
•                   elif dec == 're-raise':
•                       print 'xxx'
•                   elif dec == 'All-in':
•                       self.test2(p)
•                       i.isRaise = False
•                       return False
•
•               i.isRaise = False
•
•
•               return False
•
```

```
            return True

        def initialGame(self):

            self.dealer = Dealer.dealer()

    self.initial_game_list.append(self.initial_game_list.pop(0))
            self.in_game_list = []
            self.in_game_list.extend(self.initial_game_list)
            self.move_list = []
            self.move_list.extend(self.in_game_list)
            self.POT = 0
            self.raiseAmount = 0
            self.israised = False

            return

        def roundReset(self):

            for i in self.move_list:
                i.clean()
                i.preStack = i.money
                i.minbet = self.minbet
                i.hand = self.dealer.dealHandCards()
                i.money -= self.ante
                self.POT += self.ante

            return

        def preflopOver(self):

            self.israised = False
            self.boradCards = self.dealer.dealBoradCards()
            #self.move_list = []
            # self.move_list.extend(self.in_game_list)

            for i in self.in_game_list:
                i.clean()
                if i.decision == 'All-in':
                    self.move_list.remove(i)

            return

        def turnStart(self):

            for i in self.in_game_list:
                i.boradCards = self.boradCards

```

```
            return

        def turnOver(self):

            self.raiseAmount = 0
            #self.move_list = []
            # self.move_list.extend(self.in_game_list)
            self.boradCards += self.dealer.dealTwoMoreCards()

            for i in self.in_game_list:
                i.clean()
                if i.decision == 'All-in':
                    if i in self.move_list:
                        self.move_list.remove(i)

            return

        def riverStage(self):

            tmp = len(self.move_list)

            for i in range(len(self.move_list)):

                if i < len(self.move_list):
                    obj = self.move_list[i]
                    obj.addPosition([i, len(self.move_list)])
                    obj.pot = self.POT
                    obj.boradCards = self.boradCards

                    if obj.name != '3 AI V1' and obj.name != '5 AI
    V1' and obj.name != '7 AI V2':
                        deci = obj.makeDecision()
                    else:
                        deci = obj.makeDecision(len(self.move_list)-
    1)

                    if not self.test1(deci, obj):
                        break
                    '''
                    if len(self.move_list) != tmp:
                        obj = self.move_list[i-1]
                        obj.addPosition([i, len(self.move_list)])
                        obj.pot = self.POT
                        obj.boradCards = self.boradCards

                        if obj.name != 'AI V1':
                            deci = obj.makeDecision()
                        else:
```

```
                        deci =
    obj.makeDecision(len(self.move_list))

                    if not self.test1(deci, obj):
                        break
                '''

        return

    def result(self):

        minstrength = 0
        self.winner = ''
        for i in self.in_game_list:

            if i.getHandStrength() > minstrength:
                minstrength = i.getHandStrength()
                self.winner = i.name

            # print i.getHandStrength()

        print "Winner is " + self.winner

        for i in self.fixed_game_list:
            if i.name == self.winner:
                i.money += self.POT
            if i.money <= 0:
                if i in self.initial_game_list:
                    self.initial_game_list.remove(i)
            if i.name == '7 AI V2':
                if self.winner != i.name:
                    i.reinforcement(0)
                else:
                    i.reinforcement(1)
                print i.threshold1,i.HELLO


        for i in range(len(self.fixed_game_list)):
            obj = self.fixed_game_list[i]
            self.data[i+1].append(obj.money)




        tmp1 = 0
        tmp2 = 0
        for i in self.fixed_game_list:
            tmp1 += i.money
```

```
                tmp2 += i.money-i.preStack
                print i.name, i.money, i.money-i.preStack
                if i.money == 0:
                    i.preStack = 0
            if tmp2 != 0:
                print self.POT
            print tmp1, tmp2  # fix the allin problem
            tmp1 = 0
            tmp2 = 0

            playerCount = len(self.initial_game_list)

            if playerCount == 1:
                self.GameEnd = True
            '''
            elif playerCount <= 4:
                self.ante = 150
                self.minbet = 300

            elif playerCount <= 5:
                self.ante = 500
                self.minbet = 1250
            elif playerCount <= 7:
                self.ante = 100
                self.minbet = 250
            '''

            if playerCount <= 3:
                for i in self.in_game_list:
                    if i.name[2:] != 'AI V1' or i.name[2:] != 'AI
    V2':
                        self.GameEnd = False
                        break
                    else:
                        self.GameEnd = True

            if playerCount == 2:
                self.GameEnd = True

            for i in self.in_game_list:
                i.clean()

            print "Next Round.\n\n"

        def gameOver(self):

            print "GAME OVER"
```

```
            for i in self.initial_game_list:
                print i.name, i.money
                if i.name == '7 AI V2':
                    print i.threshold1,i.threshold2

            print 'Press enter to Exit.'
            raw_input()

            return

        def start(self):

            while not self.GameEnd:

                #print 'Press enter to start.'
                #raw_input()
                self.counter += 1
                if self.counter >= 2000:
                    self.GameEnd = True

                print "Initial the game..."
                self.initialGame()
                print "Game started.\n\nPre-flop stage:"
                self.roundReset()

                self.utg = self.move_list[0]
                self.utg.position = 'Utg'
                self.utg.betAmount = self.minbet
                self.utg.previousBet = self.utg.betAmount
                self.utg.money -= self.utg.betAmount
                self.utg.pot = self.POT
                self.POT += self.utg.betAmount

                print self.utg.name, self.utg.convert(
                    self.utg.hand), 'bet', self.utg.betAmount,
    'pot:', self.utg.pot, 'money:', self.utg.money

                for i in self.move_list[1:]:
                    i.addPosition([self.move_list.index(i),
    len(self.move_list)])
                    i.pot = self.POT
                    if i.name != '3 AI V1' and i.name != '5 AI V1'
    and i.name != '7 AI V2':
                        deci = i.startHand()
                    else:
                        deci = i.startHand(len(self.move_list)-1)
                    if not self.test1(deci, i):
                        break
```

```
•
•               self.utg.pot = self.POT
•               if not self.israised:
•                   if self.utg.name != '3 AI V1' and
    self.utg.name != '5 AI V1'and self.utg.name != '7 AI V2' :
•                       utgdeci = self.utg.startHand()
•                   else:
•                       utgdeci =
    self.utg.startHand(len(self.move_list)-1)
•                   self.test1(utgdeci, self.utg)
•
•               if len(self.in_game_list) == 1:
•
•                   self.in_game_list[0].money += self.POT
•                   self.POT = 0
•                   #print 'winner ??'
•                   continue
•
•               self.preflopOver()
•
•               print "\nTurn stage:\nThe borad cards
    are: %s,%s,%s" % (self.robot1.convert(
•                   self.boradCards)[0],
    self.robot1.convert(self.boradCards)[1],
    self.robot1.convert(self.boradCards)[2])
•
•               self.turnStart()
•
•               self.riverStage()
•
•               self.turnOver()
•
•               print "\nRiver stage:\nThe borad cards
    are: %s,%s,%s,%s,%s" % (self.robot1.convert(self.boradCards)[0],
    self.robot1.convert(
•                   self.boradCards)[1],
    self.robot1.convert(self.boradCards)[2],
    self.robot1.convert(self.boradCards)[3],
    self.robot1.convert(self.boradCards)[4])
•
•               self.turnStart()
•
•               self.riverStage()
•
•               print "Result:"
•
•               self.result()
•
```

```
            self.gameOver()

            #self.printData()

        def printData(self):

            #print 'filename?:'
            #input1 = raw_input()
            name =
    str(time.localtime().tm_mon)+'.'+str(time.localtime().tm_mday)+'.
    '+str(time.localtime().tm_hour)+'.'+str(time.localtime().tm_min)
            workbook = xlsxwriter.Workbook(name+'.xlsx')
            worksheet = workbook.add_worksheet('sheet')

            row = 0
            col = 1

            for i in self.data[1:]:
                for y in i:
                    worksheet.write(row,col,y)
                    col += 1
                    if y <= 0:
                        break
                col = 1
                row += 1

            for k in range(len(self.fixed_game_list)):
                worksheet.write(k,0,self.fixed_game_list[k].name)

            workbook.close()
            print name

    def main():

        #tmp = test()
        t = time.time()
        circle = 1
        list1 = []

        for i in range(circle):
            list1.append([])
            tmp = test()
            tmp.start()
            list1[i].append(tmp.data)
            for i in tmp.fixed_game_list:
                tmp.data[0].append(i.name)

        '''
```

```
    name =
str(time.localtime().tm_mon)+'.'+str(time.localtime().tm_mday)+'.
'+str(time.localtime().tm_hour)+'.'+str(time.localtime().tm_min)
    callbook = xlsxwriter.Workbook('call'+name+'.xlsx')
    callsheet = callbook.add_worksheet('sheet')
    standardbook = xlsxwriter.Workbook('standard'+name+'.xlsx')
    standardsheet = standardbook.add_worksheet('sheet')

    print name

    row1 = 0
    col1 = 0
    row2 = 0
    col2 = 0

    for k in list1:
        for c in range(len(k[0])-1):

            if k[0][0][c] == 'AI V1':
                for i in k[0][c+1]:
                    callsheet.write(row1,col1,i)
                    col1 += 1
                    if col1 >= 2000:
                        break
                    if i <= 0:
                        break

            elif k[0][0][c] == 'standard robot':
                for i in k[0][c+1]:
                    standardsheet.write(row2,col2,i)
                    col2 += 1
                    if col2 >= 2000:
                        break
                    if i <= 0:
                        break
        col1 = col2 = 0
        row1 += 1
        row2 += 1

    callbook.close()
    standardbook.close()

    '''

    print time.time() - t


main()
```

- **CallRobot.py**

```
class robot(object):

    def __init__(self,name,money,minbet):

        self.name = name
        self.money = money
        self.minbet = minbet
        self.pot = 0
        self.hand = []
        self.boradCards = []
        self.isRaise = False
        self.betAmount = 0
        self.preStack = 0
        self.MAPPING =
{'A':12,'K':11,'Q':10,'J':9,'10':8,'9':7,'8':6,'7':5,'6':4,'5':3,
'4':2,'3':1,'2':0}
        self.R_MAPPING =
{'12':'A','11':'K','10':'Q','9':'J','8':'10','7':'9','6':'8','5':
'7','4':'6','3':'5','2':'4','1':'3','0':'2'}
        self.Type_MAPPING = {'High card':0,'One pair':1,'Two
pairs':2,'Trips':3,'Straight':4,'Full house':5,'Quads':6}
        self.R_Type_MAPPING = {'0':'High card','1':'One
pair','2':'Two pairs','3':'Trips','4':'Straight','5':'Full
house','6':'Quads'}

        return

    def clean(self):

        self.pot = 0
        self.boradCards = []
        self.isRaise = False
        self.betAmount = 0

        return

    def convert(self,cards): # int in list eg.[11,2]

        cards.sort(reverse=True)
        vision = []

        for i in cards:

            tmp = self.R_MAPPING[str(i)]
            vision.append(tmp)
```

```
•
•           return vision
•
•       def calaulate_hold_strength(self,cards):
•
•           strength = '??'
•
•           if cards[0] == cards[1]:
•               if cards[0] >= 10:
•                   strength = 'Super Strong'
•               elif cards[0] >= 8:
•                   strength = 'Strong'
•               else:
•                   strength = 'Opportunity'
•           elif cards[0] == 12:
•               if cards[1] >= 10:
•                   strength = 'Super Strong'
•               elif cards[1] >= 8:
•                   strength = 'Strong'
•               else:
•                   strength = 'Dangerous'
•           elif cards[0] == 11:
•               if cards[1] >= 10:
•                   strength = 'Strong'
•               elif cards[1] >= 8:
•                   strength = 'Dangerous'
•               else:
•                   strength = 'Weak'
•           elif cards[0] == 10:
•               if cards[1] >= 8:
•                   strength = 'Opportunity'
•               else:
•                   strength = 'Weak'
•           elif cards[0]-cards[1] == 1 and cards[1] >= 6:
•               strength = 'Opportunity'
•           else:
•               strength = 'Weak'
•
•           return strength
•
•       def raiseDecision(self,totalBET,totalPOT):
•
•           self.pot = totalPOT
•           self.previousBet = self.betAmount
•           self.needToBet = totalBET - self.previousBet
•           self.betAmount = self.needToBet
•           self.decision = 'call'
```

```
        if self.betAmount < self.minbet and self.previousBet ==
0:
            self.betAmount = self.minbet
        if self.money < self.minbet * 4 or self.needToBet >
self.money:
            self.decision = 'All-in'
            self.betAmount = self.money
            self.money = 0
        else:
            self.money -= self.betAmount

        print
self.name,self.convert(self.hand),self.decision,self.betAmount,se
lf.pot,self.money

        return self.decision

    def reRaiseDecision(self):

        self.decision = 'fold'

        print
self.name,self.convert(self.hand),self.decision,self.betAmount,'p
ot:',self.pot,'money:',self.money

        return self.decision

    def addPosition(self,position):

        self.position = ''

        return

    def startHand(self):

        self.decision = 'call'
        self.betAmount = self.minbet

        if self.position == 'Utg':
            self.betAmount = 0

        if self.money < self.minbet * 4:
            self.decision = 'All-in'
            self.betAmount = self.money
            self.money = 0
        else:
            self.money -= self.betAmount

```

```
            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money

            return self.decision

        def makeDecision(self):

            # standard robot, decided by position, if back and no
    raise then raise(bluff)
            # if got one pair or above then raise 1/2 pot

            self.decision = 'check'
            self.betAmount = 0

            if self.money < self.minbet * 4:
                self.decision = 'All-in'
                self.betAmount = self.money
                self.money = 0

            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money

            return self.decision

        def getHandStrength(self):

            best,self.types,strength =
    self.find_my_best(self.hand+self.boradCards)

            return strength

        def find_my_best(self,cards):

            cards.sort(reverse=True)
            unique = len(set(cards))
            length = len(cards)
            keys = []
            tmp_list = list(set(cards))
            tmp_list.sort(reverse=True)
            types = 999

            for i in tmp_list:
                if cards.count(i) >= 2:
                    keys.append(i)

            if length - unique >= 3:
```

```
                types = 666

                for i in tmp_list:
                    if cards.count(i) >= 5:
                        print 'OOOOOOut of range!!!'
                    elif cards.count(i) == 3:
                        if len(keys) == 1:
                            types = 4 # trips
                            break
                        elif len(keys) >= 2:
                            types = 6 # full house
                            break
                        else:
                            print 'jesus'

                for i in tmp_list:
                    if cards.count(i) == 4:
                        types = 7 # quads
                        key6 = i
                        break

                if len(keys) == 3:
                    types = 3 # two pairs

            elif unique == length - 2:
                types = 6666
                for i in tmp_list:
                    if cards.count(i) == 2:
                        types = 3 # two pairs
                        break
                    elif cards.count(i) == 3:
                        types = 4 # trips
                        break
                if unique == 5 and tmp_list[0] - tmp_list[4] == 4:
                    types = 5 # straight
                    cards = tmp_list
            elif length - unique <= 1 :
                types = 2 # one pair
                for i in range(unique-4):
                    if tmp_list[i] - tmp_list[i+4] == 4:
                        types = 5 # straight
                        del tmp_list[:i]
                        break
                    elif length == unique:
                        types = 1 # high card
                        #break
                    elif length - unique == 1:
                        types = 2 # one pair
```

```
                        #break
            else:
                types = 2 # one pair

            best = []

            if types == 7:
                tmp_list.remove(key6)
                best = [key6,key6,key6,key6,tmp_list[0]]
            elif types == 6:
                key2 = []
                key3 = 999
                for i in tmp_list:
                    if cards.count(i) == 2:
                        key2.append(i)
                    elif cards.count(i) == 3:
                        key3 = i
                key2.sort(reverse=True)
                if key2 == []:
                    for i in tmp_list:
                        if cards.count(i) == 3:
                            key2.append(i)
                    key2.sort(reverse=True)
                    try:
                        best =
    [key2[0],key2[0],key2[0],key2[1],key2[1]]
                    except:
                        print cards, types, key2, tmp_list, length,
    unique
                else:
                    best = [key3,key3,key3,key2[0],key2[0]]
            elif types == 5:
                best = tmp_list[:5]
            elif types == 4:
                tmp_list.remove(keys[0])
                best = keys * 3 + tmp_list[:2]
            elif types == 3:
                keys.sort(reverse=True)
                for i in keys[:2]:
                    best.append(i)
                    best.append(i)
                best.append(tmp_list[0])
            elif types == 2:
                tmp_list.remove(keys[0])
                best = keys * 2 + tmp_list[:3]
            elif types == 1:
                best = cards[:5]

```

```
            if types == 666:
                print cards, best
            strength = self.calaulate_hand_strength(best,types)

            return best,types,strength

        def calaulate_hand_strength(self,cards,types):

            strength = hex(types * 16 ** 5 + cards[0] * 16 ** 4 +
    cards[1] * 16 ** 3 + cards[2] * 16 **2 + cards[3] * 16 +
    cards[4])
            # print hex(cards[0] * 16 ** 4 + cards[1] * 16 ** 3 +
    cards[2] * 16 **2 + cards[3] * 16 + cards[4])
            return strength

```

- **StandardRobot.py**

```
    import random

    class robot(object):

        def __init__(self,name,money,minbet):

            self.name = name
            self.money = money
            self.minbet = minbet
            self.pot = 0
            self.hand = []
            self.boradCards = []
            self.isRaise = False
            self.betAmount = 0
            self.preStack = 0
            self.MAPPING =
    {'A':12,'K':11,'Q':10,'J':9,'10':8,'9':7,'8':6,'7':5,'6':4,'5':3,
    '4':2,'3':1,'2':0}
            self.R_MAPPING =
    {'12':'A','11':'K','10':'Q','9':'J','8':'10','7':'9','6':'8','5':
    '7','4':'6','3':'5','2':'4','1':'3','0':'2'}
            self.Type_MAPPING = {'High card':0,'One pair':1,'Two
    pairs':2,'Trips':3,'Straight':4,'Full house':5,'Quads':6}
            self.R_Type_MAPPING = {'0':'High card','1':'One
    pair','2':'Two pairs','3':'Trips','4':'Straight','5':'Full
    house','6':'Quads'}

            return

```

```
    def clean(self):

        self.pot = 0
        self.boradCards = []
        self.isRaise = False
        self.betAmount = 0

        return

    def convert(self,cards): # int in list eg.[11,2]

        cards.sort(reverse=True)
        vision = []

        for i in cards:

            tmp = self.R_MAPPING[str(i)]
            vision.append(tmp)

        return vision

    def calaulate_hold_strength(self,cards):

        strength = '??'

        if cards[0] == cards[1]:
            if cards[0] >= 10:
                strength = 'Super Strong'
            elif cards[0] >= 8:
                strength = 'Strong'
            else:
                strength = 'Opportunity'
        elif cards[0] == 12:
            if cards[1] >= 10:
                strength = 'Super Strong'
            elif cards[1] >= 8:
                strength = 'Strong'
            else:
                strength = 'Dangerous'
        elif cards[0] == 11:
            if cards[1] >= 10:
                strength = 'Strong'
            elif cards[1] >= 8:
                strength = 'Dangerous'
            else:
                strength = 'Weak'
        elif cards[0] == 10:
            if cards[1] >= 8:
```

```
                    strength = 'Opportunity'
                else:
                    strength = 'Weak'
            elif cards[0]-cards[1] == 1 and cards[1] >= 6:
                strength = 'Opportunity'
            else:
                strength = 'Weak'

            return strength

        def raiseDecision(self,totalBET,totalPOT):

            # need update

            self.pot = totalPOT
            self.previousBet = self.betAmount
            self.needToBet = totalBET - self.previousBet
            self.betAmount = self.needToBet
            self.potodds = self.betAmount / (totalPOT+self.betAmount)


            if self.potodds <= 0.2:
                self.decision = 'call'
                if self.betAmount >= self.money:
                    self.betAmount = self.money
                self.money -= self.betAmount
            else:
                self.decision = 'fold'

            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,se
    lf.pot,self.money

            return self.decision

        def reRaiseDecision(self):

            self.decision = 'fold'

            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money

            return self.decision

        def addPosition(self,position):

            self.oppoentNumber = position[1]-1
```

```
•
•
•           pos = float(position[0]+1) / float(position[1])
•
•           if pos <= 1.0/3.0:
•
•               self.position = 'Front'
•
•           elif pos <= 2.0/3.0:
•
•               self.position = 'Middle'
•
•           else:
•
•               self.position = 'Back'
•
•           return
•
•       def startHand(self):
•
•           if self.hand == [12,12] or self.hand == [11,11]: # AA and
    KK
•               self.decision = 'raise'
•           elif self.hand == [10,10]:
•               self.decision = 'raise'
•           elif self.hand == [12,11]:
•               self.decision = 'raise'
•           elif self.hand == [9,9] or self.hand == [8,8] or
    self.hand == [7,7]:
•               self.decision = 'raise'
•           elif self.hand[0] == self.hand[1] and self.hand[0] <= 6:
•               self.decision = 'call'
•           elif self.hand[0] == 12 and self.hand[1] >= 8:
•               self.decision = 'call'
•           elif self.hand[0] == 12:
•               self.decision = 'call'
•           elif self.hand[0] <= 11 and self.hand[0] >= 10 and
    self.hand[1] <= 10 and self.hand >= 9:
•               self.decision = 'call'
•           elif self.hand[0] - self.hand[1] == 1 and self.hand[1] >=
    4:
•               self.decision = 'call'
•           else:
•               self.decision = 'fold'
•
•           if self.decision == 'call':
•               self.betAmount = self.minbet
•               if self.position == 'Utg':
```

```
                    self.betAmount = 0
            elif self.decision == 'raise':
                self.betAmount = 3 * self.minbet
            elif self.decision == 'fold':
                self.betAmount = 0
                if self.position == 'Utg':
                    self.decision = 'call'

            self.money -= self.betAmount

            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money

            return self.decision

        def makeDecision(self):

            # standard robot, decided by position, if back and no
    raise then raise(bluff)
            # if got one pair or above then raise 1/2 pot

            if self.position == 'Back':
                if not self.isRaise:
                    self.decision = 'raise'
                    self.betAmount = self.minbet
                    self.isRaise = True
            else:
                self.decision = 'check'
                self.betAmount = 0

            self.getHandStrength()
            if self.types >= 3:
                self.decision = 'raise'
                self.isRaise = True
                if self.position == 'Front':
                    self.betAmount = self.minbet
                elif self.position == 'Middle':
                    self.betAmount = self.minbet * 2
                elif self.position == 'Back':
                    self.betAmount = self.minbet * 3

            self.money -= self.betAmount

            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money
```

```
            return self.decision

        def getHandStrength(self):

            best,self.types,strength =
    self.find_my_best(self.hand+self.boradCards)

            return strength

        def find_my_best(self,cards):

            cards.sort(reverse=True)
            unique = len(set(cards))
            length = len(cards)
            keys = []
            tmp_list = list(set(cards))
            tmp_list.sort(reverse=True)
            types = 999

            for i in tmp_list:
                if cards.count(i) >= 2:
                    keys.append(i)

            if length - unique >= 3:
                types = 666

                for i in tmp_list:
                    if cards.count(i) >= 5:
                        print 'OOOOOOut of range!!!'
                    elif cards.count(i) == 3:
                        if len(keys) == 1:
                            types = 4 # trips
                            break
                        elif len(keys) >= 2:
                            types = 6 # full house
                            break
                        else:
                            print 'jesus'

                for i in tmp_list:
                    if cards.count(i) == 4:
                        types = 7 # quads
                        key6 = i
                        break

                if len(keys) == 3:
                    types = 3 # two pairs
```

```
        elif unique == length - 2:
            types = 6666
            for i in tmp_list:
                if cards.count(i) == 2:
                    types = 3 # two pairs
                    break
                elif cards.count(i) == 3:
                    types = 4 # trips
                    break
            if unique == 5 and tmp_list[0] - tmp_list[4] == 4:
                types = 5 # straight
                cards = tmp_list
        elif length - unique <= 1 :
            types = 2 # one pair
            for i in range(unique-4):
                if tmp_list[i] - tmp_list[i+4] == 4:
                    types = 5 # straight
                    del tmp_list[:i]
                    break
                elif length == unique:
                    types = 1 # high card
                    #break
                elif length - unique == 1:
                    types = 2 # one pair
                    #break
        else:
            types = 2 # one pair

        best = []

        if types == 7:
            tmp_list.remove(key6)
            best = [key6,key6,key6,key6,tmp_list[0]]
        elif types == 6:
            key2 = []
            key3 = 999
            for i in tmp_list:
                if cards.count(i) == 2:
                    key2.append(i)
                elif cards.count(i) == 3:
                    key3 = i
            key2.sort(reverse=True)
            if key2 == []:
                for i in tmp_list:
                    if cards.count(i) == 3:
                        key2.append(i)
                key2.sort(reverse=True)
                try:
```

```
                    best =
[key2[0],key2[0],key2[0],key2[1],key2[1]]
                except:
                    print cards, types, key2, tmp_list, length,
unique
            else:
                best = [key3,key3,key3,key2[0],key2[0]]
        elif types == 5:
            best = tmp_list[:5]
        elif types == 4:
            tmp_list.remove(keys[0])
            best = keys * 3 + tmp_list[:2]
        elif types == 3:
            keys.sort(reverse=True)
            for i in keys[:2]:
                best.append(i)
                best.append(i)
            best.append(tmp_list[0])
        elif types == 2:
            tmp_list.remove(keys[0])
            best = keys * 2 + tmp_list[:3]
        elif types == 1:
            best = cards[:5]
        '''
        time.sleep(0.1)
        print types
        '''
        if types == 666:
            print cards, best
        strength = self.calaulate_hand_strength(best,types)

        return best,types,strength

    def calaulate_hand_strength(self,cards,types):
        '''
        time.sleep(0.1)
        print cards
        '''

        strength = hex(types * 16 ** 5 + cards[0] * 16 ** 4 +
cards[1] * 16 ** 3 + cards[2] * 16 **2 + cards[3] * 16 +
cards[4])
        # print hex(cards[0] * 16 ** 4 + cards[1] * 16 ** 3 +
cards[2] * 16 **2 + cards[3] * 16 + cards[4])
        return strength

```

- **Dealer.py**

```
import random

class dealer(object):

    def __init__(self):

        self.initDECK = [0,1,2,3,4,5,6,7,8,9,10,11,12] * 4

        tmp_deck = self.initDECK
        random.shuffle(tmp_deck)
        self.Deck = tmp_deck

        return

    def dealHandCards(self):

        Hand1 = self.Deck[:2]
        Hand1.sort(reverse=True)
        del self.Deck[:2]

        return Hand1

    def dealBoradCards(self):

        Boradcards = self.Deck[:3]
        del self.Deck[:3]
        Boradcards.sort(reverse=True)

        return Boradcards

    def dealTwoMoreCards(self):

        cards = self.Deck[:2]
        cards.sort(reverse=True)
        del self.Deck[:2]

        return cards
```

- **AI_V2.py**

```
from __future__ import division
import random

class robot(object):

    def __init__(self,name,money,minbet):

        self.name = name
        self.money = money
        self.minbet = minbet
        self.pot = 0
        self.hand = []
        self.boradCards = []
        self.isRaise = False
        self.betAmount = 0
        self.position = ''
        self.HELLO = False
        self.threshold1 = 1.3
        self.threshold2 = 0.9
        self.MAPPING =
{'A':12,'K':11,'Q':10,'J':9,'10':8,'9':7,'8':6,'7':5,'6':4,'5':3,
'4':2,'3':1,'2':0}
        self.R_MAPPING =
{'12':'A','11':'K','10':'Q','9':'J','8':'10','7':'9','6':'8','5':
'7','4':'6','3':'5','2':'4','1':'3','0':'2'}
        self.Type_MAPPING = {'High card':1,'One pair':2,'Two
pairs':3,'Trips':4,'Straight':5,'Full house':6,'Quads':7}
        self.R_Type_MAPPING = {'1':'High card','2':'One
pair','3':'Two pairs','4':'Trips','5':'Straight','6':'Full
house','7':'Quads'}

        return

    def clean(self):

        self.pot = 0
        self.boradCards = []
        self.betAmount = 0
        self.isRaise = False

        return

    def prt(self):

```

```
            print
    self.name,self.convert(self.hand),self.decision,self.betAmount,'p
    ot:',self.pot,'money:',self.money,str(self.winrate*100)+'%'

            return

        def convert(self,cards): # int in list eg.[11,2]

            cards.sort(reverse=True)
            vision = []

            for i in cards:

                tmp = self.R_MAPPING[str(i)]
                vision.append(tmp)

            return vision

        def calaulate_hold_strength(self,cards):

            strength = '??'

            if cards[0] == cards[1]:
                if cards[0] >= 10:
                    strength = 'Super Strong'
                elif cards[0] >= 8:
                    strength = 'Strong'
                else:
                    strength = 'Opportunity'
            elif cards[0] == 12:
                if cards[1] >= 10:
                    strength = 'Super Strong'
                elif cards[1] >= 8:
                    strength = 'Strong'
                else:
                    strength = 'Dangerous'
            elif cards[0] == 11:
                if cards[1] >= 10:
                    strength = 'Strong'
                elif cards[1] >= 8:
                    strength = 'Dangerous'
                else:
                    strength = 'Weak'
            elif cards[0] == 10:
                if cards[1] >= 8:
                    strength = 'Opportunity'
                else:
                    strength = 'Weak'
```

```
            elif cards[0]-cards[1] == 1 and cards[1] >= 6:
                strength = 'Opportunity'
            else:
                strength = 'Weak'

            return strength

        def raiseDecision(self,totalBET,totalPOT,opponents):

            self.pot = totalPOT
            self.previousBet = self.betAmount
            self.needToBet = totalBET - self.previousBet
            self.betAmount = self.needToBet
            self.potodds = self.betAmount / totalPOT
            self.winrate = self.simulation_outside(1000,opponents)
            returnrate =
    self.returnrate(float(self.winrate),float(self.potodds))

            if returnrate >= 1.3:
                self.decision = 'call'
                self.money -= self.betAmount
            else:
                self.decision = 'fold'

            self.prt()

            return self.decision


        def reRaiseDecision(self):

            self.decision = 'fold'

            self.prt()

            return self.decision

        def addPosition(self,position):

            self.oppoentNumber = position[1]-1


            pos = float(position[0]+1) / float(position[1])

            if pos <= 1.0/3.0:

                self.position = 'Front'

```

```
            elif pos <= 2.0/3.0:

                self.position = 'Middle'

            else:

                self.position = 'Back'

            return

        def startHand(self,opponents):

            winrate = self.simulation_outside(1000,opponents)
            self.winrate = winrate
            potodds = self.calpotodds(self.minbet,self.pot)
            returnrate =
    self.returnrate(float(winrate),float(potodds))

            tmp = random.randint(0,99)
            if returnrate < 0.8:
                self.decision = 'fold'
            elif returnrate < 1:
                if tmp <= 20:
                    self.decision = 'call'
                else:
                    self.decision = 'fold'
            elif returnrate < 1.3:
                if tmp <= 59:
                    self.decision = 'call'
                else:
                    self.decision = 'raise'
            else:
                if tmp <= 29:
                    self.decision = 'call'
                else:
                    self.decision = 'raise'

            if self.decision == 'call':
                self.betAmount = self.minbet
                if self.position == 'Utg':
                    self.betAmount = 0
            elif self.decision == 'raise':
                self.betAmount = 4 * self.minbet
            elif self.decision == 'fold':
                self.betAmount = 0
                if self.position == 'Utg':
                    self.decision = 'call'

```

```
            self.money -= self.betAmount

            self.prt()

            return self.decision

        def makeDecision(self,opponents):

            winrate = self.simulation_outside(1000,opponents)
            self.winrate = winrate
            potodds = self.calpotodds(self.minbet,self.pot)
            returnrate =
    self.returnrate(float(winrate),float(potodds))

            tmp = random.randint(0,99)
            if returnrate < 1:
                self.decision = 'check'
            elif returnrate < self.threshold1:
                if tmp <= 59:
                    self.decision = 'check'
                else:
                    self.decision = 'raise'
                self.HELLO = True
            else:
                self.decision = 'raise'
            if self.decision == 'check':
                self.betAmount = 0
            elif self.decision == 'raise':
                self.isRaise = True

                #t = random.randint(3,8)
                self.betAmount = self.minbet
                if self.winrate >= 0.98:
                    if self.money >= 2 * self.pot:
                        self.betAmount = 2 * self.pot
                    else:
                        self.betAmount = self.money
                elif self.winrate >= self.threshold2:
                    self.betAmount = 10 * self.minbet

            self.money -= self.betAmount

            self.prt()

            return self.decision

        def reinforcement(self,outcome): # 1 for win, 0 for lose

```

```
            if outcome == 1:
                if self.HELLO:
                    self.threshold1 -= 0.01
            elif outcome == 0:
                if self.HELLO:
                    self.threshold1 += 0.01

            self.HELLO = False

        def getHandStrength(self):

            best,self.types,strength =
    self.find_my_best(self.hand+self.boradCards)

            return strength

        def find_my_best(self,cards):

            cards.sort(reverse=True)
            unique = len(set(cards))
            length = len(cards)
            keys = []
            tmp_list = list(set(cards))
            tmp_list.sort(reverse=True)
            types = 999

            for i in tmp_list:
                if cards.count(i) >= 2:
                    keys.append(i)

            if length - unique >= 3:
                types = 666

                for i in tmp_list:
                    if cards.count(i) >= 5:
                        print 'OOOOOOut of range!!!'
                    elif cards.count(i) == 3:
                        if len(keys) == 1:
                            types = 4 # trips
                            break
                        elif len(keys) >= 2:
                            types = 6 # full house
                            break
                        else:
                            print 'jesus'

                for i in tmp_list:
                    if cards.count(i) == 4:
```

```
                    types = 7 # quads
                    key6 = i
                    break

            if len(keys) == 3:
                types = 3 # two pairs

        elif unique == length - 2:
            types = 6666
            for i in tmp_list:
                if cards.count(i) == 2:
                    types = 3 # two pairs
                    break
                elif cards.count(i) == 3:
                    types = 4 # trips
                    break
            if unique == 5 and tmp_list[0] - tmp_list[4] == 4:
                types = 5 # straight
                cards = tmp_list
        elif length - unique <= 1 :
            types = 2 # one pair
            for i in range(unique-4):
                if tmp_list[i] - tmp_list[i+4] == 4:
                    types = 5 # straight
                    del tmp_list[:i]
                    break
                elif length == unique:
                    types = 1 # high card
                    #break
                elif length - unique == 1:
                    types = 2 # one pair
                    #break
        else:
            types = 2 # one pair

        best = []

        if types == 7:
            tmp_list.remove(key6)
            best = [key6,key6,key6,key6,tmp_list[0]]
        elif types == 6:
            key2 = []
            key3 = 999
            for i in tmp_list:
                if cards.count(i) == 2:
                    key2.append(i)
                elif cards.count(i) == 3:
                    key3 = i
```

```
            key2.sort(reverse=True)
            if key2 == []:
                for i in tmp_list:
                    if cards.count(i) == 3:
                        key2.append(i)
                key2.sort(reverse=True)
                try:
                    best =
[key2[0],key2[0],key2[0],key2[1],key2[1]]
                except:
                    print cards, types, key2, tmp_list, length,
unique
            else:
                best = [key3,key3,key3,key2[0],key2[0]]
        elif types == 5:
            best = tmp_list[:5]
        elif types == 4:
            tmp_list.remove(keys[0])
            best = keys * 3 + tmp_list[:2]
        elif types == 3:
            keys.sort(reverse=True)
            for i in keys[:2]:
                best.append(i)
                best.append(i)
            best.append(tmp_list[0])
        elif types == 2:
            tmp_list.remove(keys[0])
            best = keys * 2 + tmp_list[:3]
        elif types == 1:
            best = cards[:5]
        '''
        time.sleep(0.1)
        print types
        '''
        if types == 666:
            print cards, best
        strength = self.calaulate_hand_strength(best,types)

        return best,types,strength

    def calaulate_hand_strength(self,cards,types):
        '''
        time.sleep(0.1)
        print cards
        '''

```

```
        strength = hex(types * 16 ** 5 + cards[0] * 16 ** 4 +
cards[1] * 16 ** 3 + cards[2] * 16 **2 + cards[3] * 16 +
cards[4])
        # print hex(cards[0] * 16 ** 4 + cards[1] * 16 ** 3 +
cards[2] * 16 **2 + cards[3] * 16 + cards[4])
        return strength

    def simulation_outside(self,circles,opponents):
        win = 0
        total = 0
        for i in range(circles):

            tmp_deck = [0,1,2,3,4,5,6,7,8,9,10,11,12] * 4
            random.shuffle(tmp_deck)

            for x in (self.hand+self.boradCards):
                tmp_deck.remove(int(x))

            my_holds = self.hand

            tmp = 5-len(self.boradCards)
            final_borad = self.boradCards + tmp_deck[:tmp]
            del tmp_deck[:tmp]
            opp_holds = []
            for i in range(opponents):
                opp_holds.append(tmp_deck[:2])
                del tmp_deck[:2]

            counter = 0
            for i in opp_holds:
                if self.find_my_best(my_holds + final_borad) >
self.find_my_best(i + final_borad):
                    counter += 1

            if counter == opponents:
                win += 1

            total += 1
        return win/total

    def calpotodds(self,bet,pot):

        return bet/(bet+pot)

    def returnrate(self,winrate,potodds):

        return winrate/potodds

```