Building a Strong Fighting Game Player

Vu Ngoc Quang

School of Information Science

Japan Advanced Institute of Science and Technology

September 2016

Acknowledgements

I would like to express my sincere gratitude to many people who have been with me throughout the years.

The first person I would like to thank is my supervisor, professor Iida, who has been helping me on study, research and life for the past two years. I especially appreciate his trust on me, which have been giving me the best studying experience I ever have.

Next, I would like to thank my family and friends. Many who are not here but always try to encourage me to thrive. Many who are here and always with me when I need. In particular, I would give special thanks to my teammate, who have been working with me everyday in the project that led to this thesis.

I also thanks a lot for the people here in Japan. Their help makes my life here a pleasure experience.

Contents

List of Figures

List of Tables

Abbreviations

MCTS	**M**onte-**C**arlo **T**ree **S**earch
RL	**R**einforcement **L**earning
AI	**A**rtificial **I**ntelligence
UCB	**U**pper **C**onfidence **B**ound
UCT	**U**pper **C**onfidence Bound applied to **T**ree

Chapter 1

Introduction

For a long time, games have been chosen as one of the testbeds for human intelligence. Chess, for example, has been one of the most prominent games and used to be a measure of intelligence for both human and computers for a long time. In the beginning, researches about games mostly focused on how to make a strong computer program, also called AI, and board games such as Chess and Checkers were the default testbed. As time progresses, computer programs' strength increased and surpassed the human champion level in these games. Gomoku was solved in 1993 by Victor Allis [1]. The Checkers program Chinook by Jonathan Schaeffer was considered to be equal to the World Champion Emeritus Marion Tinsley [2]. Chess champion Garry Kasparov lost to IBM's Deep Blue in 1997 [3]. And most recently, Go champion Lee Sedol lost to Google's AlphaGo in March 2016[1]. In most of these cases, the strength of computer programs comes from the combination of human expertise and sheer computational power of computers. Thus, it is fairly safe to say that in that kind of games, which is two-player zero-sum deterministic game, the computer will eventually surpass human. However, the strength of AI in other kinds of games is not so impressive. Video game is an area where computer programs are still lacking. A huge amount of efforts have been invested in creating powerful computer programs for many types of video games. The most famous success might be from Google DeepMind group in Atari games [4]. They use the Deep Reinforcement Learning technique which allows the program to train by self-playing. The technique was also employed in Google AlphaGo and contributed a fair share to its success [5]. However, in the world of video games, those Atari games are among the simplest ones. Researches in more

[1] https://deepmind.com/alpha-go

complex games have not enjoyed much successes. There currently are many video game competitions for computers to support researches. It is a very exciting area in the game research. In this thesis, we study the problem of real-time fighting video game.

1.1 Problem Statement

The challenge posed by fighting games is very different from board games. Board games is typically classified as a two-player zero-sum turn-based deterministic game. Fighting game is a real-time zero-sum non-deterministic game. The main difference is the real-time property. The player needs to issue a command to the character in a very short time frame. Because of this time limit, most tree search algorithm in board games cannot be applied directly. Therefore, many of AIs for fighting games are rule-based. Those approaches are fine when building a playable agent, however, human players can quickly recognize these rules and exploit them. Because of that, in most popular fighting games in the market, the mode human versus human is considered real games while human versus computer AI is considered to be practice sessions.

In this thesis, we aim to build a strong and adaptable AI for a fighting game testbed called Fighting ICE. Our expectation is to build an AI that can learn from past matches and adapt to new opponents. To do so, we use Monte-Carlo Tree Search as the base of our player and support it by Reinforcement Learning.

1.2 Thesis Structure

The thesis consists of 6 chapters. We introduce the background of our problem in this chapter. In chapter 2, we present a comprehensive view of our testbed, the Fighting ICE project. In chapter 3, we present related works on our approach, namely Monte-Carlo Tree Search and Reinforcement Learning. In chapter 4, we present the detail of our approach in solving the two research questions. In chapter 5, we present and discuss the results of our experiments. Finally, in chapter 6, we conclude the outcome of this research.

Chapter 2

The Fighting ICE Environment

In this chapter we introduce our game of choice: Fighting ICE. The game has been developed by the Intelligent Computer Entertainment Lab. (ICE Lab.) from Ritsumeikan University [6]. It was first introduced in [7] in CIG 2013 and its competition has been held annually since then.

2.1 Game Details

The game features two characters fighting each other in a two-dimensional arena, i.e., the character can only move up, down, left or right. We show, in Figure 2.1, a screen-shot of the game. On the top of the screen, there are the basic information of players: Hit Point (HP) and Energy. Hit Point is used to calculate the score of the game. It starts at O and decreases whenever a character gets hit by the opponent, hence the negative value is seen in the screen-shot. Energy is needed to perform certain actions, which is gained from hitting the opponent or hit by the opponent.

There are three kinds of actions:

- **Move** actions will change the position of a character and while moving, the character will always face the opponent. Some of the possible movements are RUN FORWARD, DASH BACKWARD, JUMP FORWARD or JUMP BACKWARD.

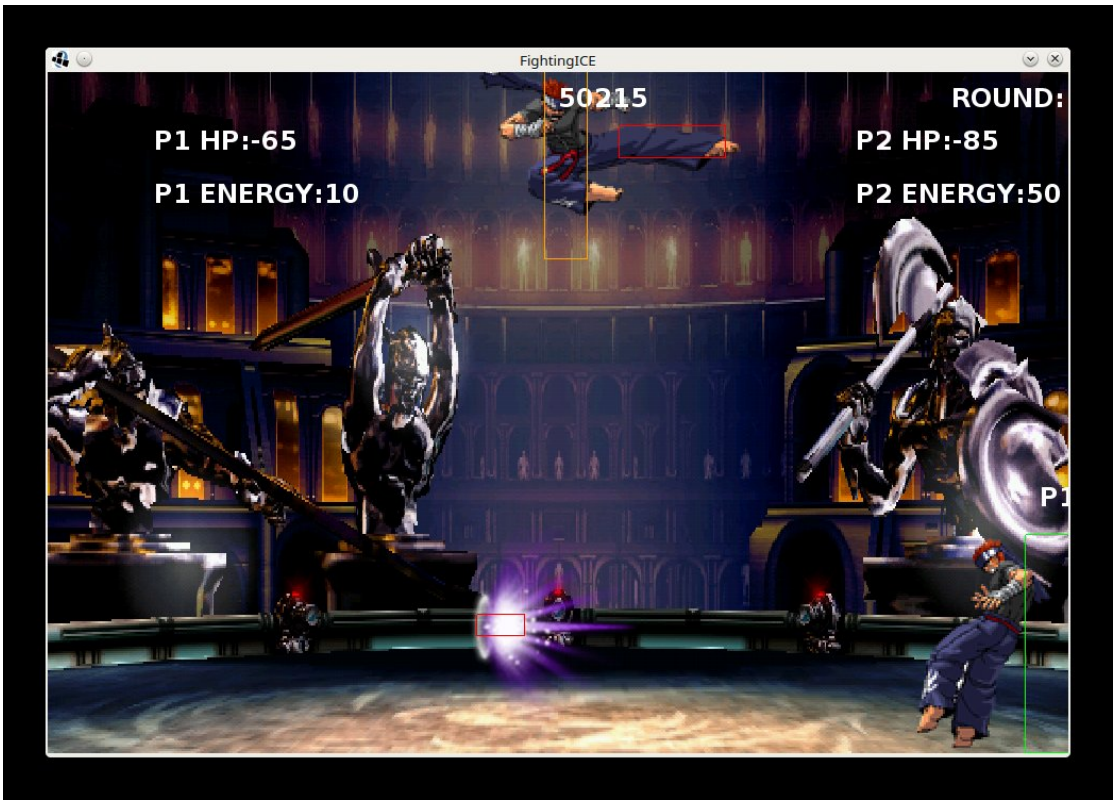

FIGURE 2.1: An in-game screenshot showing the two characters, named ZEN. The purple flying object is a THROW attack. The number at the top center is the remaining time as millisecond.

- **Attack** actions are actions that can damage the opponent (decrease their HP). Four types of attacks are available: HIGH, MIDDLE, LOW and THROW. Each type has a different properties, most notably related to GUARD actions. THROW or projectile attacks have the distinction of being separate from the user, moving independently in a fixed direction for several seconds.
- **Guard** actions can block or reduce damage from the opponent's attacks. There are 3 types of guard:
  - STAND_GUARD: block or reduce damage from HIGH and MIDDLE attacks.
  - CROUND_GUARD: block or reduce damage from LOW attacks.
  - AIR_GUARD: block or reduce damage from attack on AIR (i.e. while jumping).

An attack has 3 phases: STARTUP, ACTIVE and RECOVERY phases. The STARTUP phase is when the character starts to perform an action, but no damage

is done to the opponent. The ACTIVE phase is when the opponent can be hit. The RECOVERY phase is after the ACTIVE phase, before the character returns to the normal state. Moreover, some attacks have a CANCELABLE phase where the character can immediately perform another attack without waiting for the RECOVERY phase to end. Figure 2.2, taken from the official website[1] illustrates these phases.

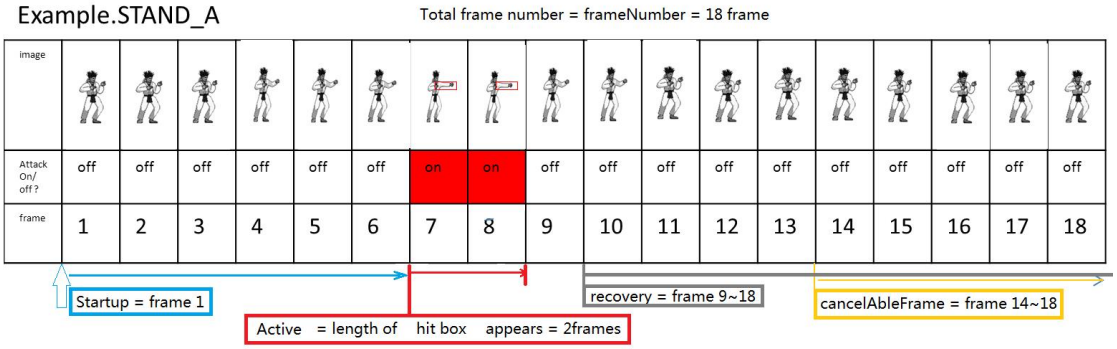

FIGURE 2.2: Sample motion data

One round in Fighting ICE last 60 seconds. To simulate a real-time environment, each second is divided into 60 frames, which means that 1 frame correspond to 16.67ms. At the start of a frame, the game manager will send game information to both players. In order to execute an action in a frame, a player needs to decide its action within that frame. Moreover, there is a 15-frame delay in the information sent, which means that the information that players received is of 15 frames ago. This lag, which mimics the delay of human perception, ensures that player cannot use simple counter tactics by just choosing an action that can counter the opponent's action of the previous frame. The organizer hopes that this condition will promote more human-like AIs. At the current version, the game has 3 characters: Zen, Garnet and LUD. Each character has different actions as well as properties such as sizes.

2.2 Competition

The game is featured in Fighting Game AI Competition, organized by ICE Lab. and hosted by Computational Intelligence & Games (CIG). The competition has

[1] http://www.ice.ci.ritsumei.ac.jp/ ftgaic/index-2a.html

been held annually since 2013 and the next competition will be held in September, 2016.

To maintain fairness between different algorithms, some restrictions are imposed. First, multi-thread processing is prohibited. All players must run on a single CPU. Second, the memory available is limited to 512MB and limited file I\O. As a result, simple algorithms cannot run better through the use of multi-threading or using large precomputed tables. Each player will be given 5 seconds prior to a match to prepare. A match consists of 3 rounds, each is 60-second long. At the end of a round, the score for a player will be calculated by the following formula:

$$\frac{opponent's\ HP}{player's\ HP + opponent's\ HP} * 1000$$

The player who has a higher score wins the round and the player who wins 2 rounds wins the match.

This year, 2016, submitted AIs must be able to play all three characters in three leagues. In a league, all AIs will play the same character with others in a round-robin format. The final result will be based on the results of three leagues. More detail information on the competition can be found in [2].

2.3 Works on Fighting ICE

Because of the real-time nature of the games, the agent has very limited time, only less than 17ms to decide an action. The game also has a large state-space as other games in this category. As a result, common algorithms in board games cannot be applied directly to the game. In fact, most of agents participated in the previous competitions are rule-based. The champion of 2015 competition is Machete, a rule-based agent. It chooses its actions based on several conditions such as the distance to the opponent, current position or amount of energy.

As a result, it only uses a few very effective actions and wins because it acts faster than most opponents. The champion of 2014 competition was CodeMonkey which used Dynamic Scripting [8], a method that utilises the reinforcement learning idea to dynamically switching the rules on-line. It has more than 20 highly refined rules. Each rule has a priority given by programmers. At the start of a match, it will

[2]http://www.ice.ci.ritsumei.ac.jp/ ftgaic/index-3c.html

choose a small set of rules to use. The action which the highest-priority matched rule chooses will be used. Every 4 seconds, it will re-estimate the effectiveness of rules and swap ineffective rules out. The authors reported that it can adapt to a new opponent in 12-32 seconds.

There are several agents based on machine learning techniques but generally weaker. For example, MizunoAI [9], an agent from ICE Lab., uses k-nearest neighbor algorithm together with a simulator to predict the move of the opponent. From the start of a match, it stores the opponent moves in memory. Then it simulates the match with the above opponent's actions and chooses an action which gives the best result. Later, the improved version, JerryMizuno AI [10], also incorporates fuzzy control to better deal with the problem of lacking information about the opponent at the start of a match.

This year, the ICE lab publishes a base MCTS AI for the Fighting ICE competition. Although the implementation is quite simple, its strength is quite good. We will discuss more about this player in chapter 4.

Chapter 3

Related Works

In our player, we use a combination of Monte-Carlo Tree Search and Reinforcement Learning. Both methods have been successfully applied to various problems, especially in game playing. Monte-Carlo Tree Search is currently one of the most popular topics while Reinforcement Learning have been a long time prominent learning algorithm. In this chapter, we summarise general information about Monte-Carlo Tree Search and Reinforcement Learning.

3.1 Game Tree Search

Game tree search is the basic concept in game programming that dated from 1950s by Shannon[11]. Commonly, game tree search is associated with Minimax algorithm and board games such as Chess. In these board games, there are 2 players that take turn to play. Figure 3.1 illustrates a simple tree in Minimax framework. In the tree, nodes represents game position and edges represents moves. Each layer, or ply, in the tree represents a turn for a player. The root node corresponds to the current game position, from where the search are conducted. Game positions in children nodes are reachable from the game positions in parent node represent by the linking edge. Each node associates with a value that are an estimated number represents the relative advantage between the two players. A positive number may signify that the current player has an advantage and a negative number may signify the opponent has an advantage. Thus, a player will try to gain the most advantage by choosing move that lead to a disadvantage for the other player. In the Minimax framework, that means the player in the Max

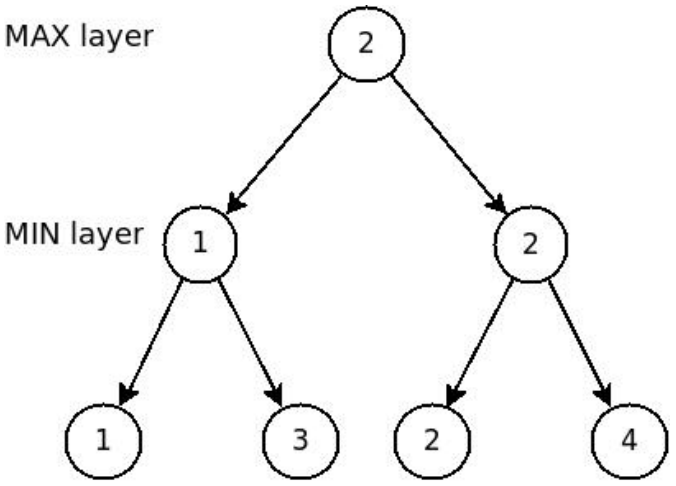

FIGURE 3.1: A game tree in Minimax framework. The value of a node in MAX layer is the maximum value of its children while the value of a node in MIN layer is the minimum value of its children.

layer will choose move lead to situation with maximum value while the player in the Min layer will choose move lead to situation with minimum value. The tree is usually expanded to a given limit and then an evaluation function is used to estimate the value of the game positions in leaf nodes. The estimated values are then backup to the root of the tree and a decision can be made upon these values.

3.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a very flexible method in building game AIs. Monte-Carlo is a term often used to denote simulation-based techniques. In the early 2000s, Monte-Carlo based methods were successfully applied to games such as Scrabble [12] and Poker [13]. Since the breakthrough in computer Go [14][15][16], Monte-Carlo based techniques have become very popular in games research. Monte-Carlo Tree Search is a game tree search algorithm which uses Monte-Carlo techniques. The advantage of MCTS is that it relies little on human. In Minimax tree search, it is essential to have a good evaluation function which is often resulted from years of human experience on the game. As a result, in games such as Go, where it is very hard to find a good evaluation function, Minimax based methods failed to reach professional level. MCTS provides a convenient and effective way of estimating an evaluation function. Later in General

Game Playing, a framework designed to prevent the existing of a strong heuristics or evaluation functions, MCTS quickly became the prominent approach[17]. Recently, the famous AlphaGo [5] also incorporates MCTS.

The concept of MCTS is very simple. MCTS builds a search tree incrementally by repeatedly runs many iterations. A node in the tree corresponds to a game situation with the current one is the root node. The game situation in a child node is usually reachable from the game situation in parent node in some predefined steps, usually corresponding to a move or action in game. Each iteration to build the tree in MCTS consists of 4 phases:

- Selection: An urgent leaf node is selected. The selected leaf node is the result of balancing between exploitation, keep visiting promising nodes, and exploration, choose nodes that have not been visited frequently.
- Expansion: The selected leaf node is expanded.
- Simulation: A simulation or playout is run from the selected node. A simulation is a sequence of actions, could be choose randomly or by some predefined distributions, that leads the game situation to the terminal state or a chosen limit.
- Back-propagation: The result of the simulation is updated backward from the selected leaf node.

The best move is usually decided by the aggregated result of all iterations. Figure 3.2, taken from [18], illustrates the 4 phases of MCTS.

An important point in MCTS is how to balance between exploitation and exploration. It need to explore new node in order to find better actions while it also need to exploit known good nodes to gain more confidence in these corresponding actions. The common way to solve this dilemma is to use UCB1 [19]. UCB stands for Upper Confidence Bound, a concept from the multi-armed bandit problem, and UCB1 is the most simple formula to calculate it. Thus, the most common type of MCTS is call UCT (Upper Confidence Bound applied to Tree) [20]. From a node n, a selection policy which use UCB1 will choose a child node c which maximizes:

$$UCB1(c) = \overline{R(c)} + C * \sqrt{\frac{ln\ visit(n)}{visit(c)}} \tag{3.1}$$

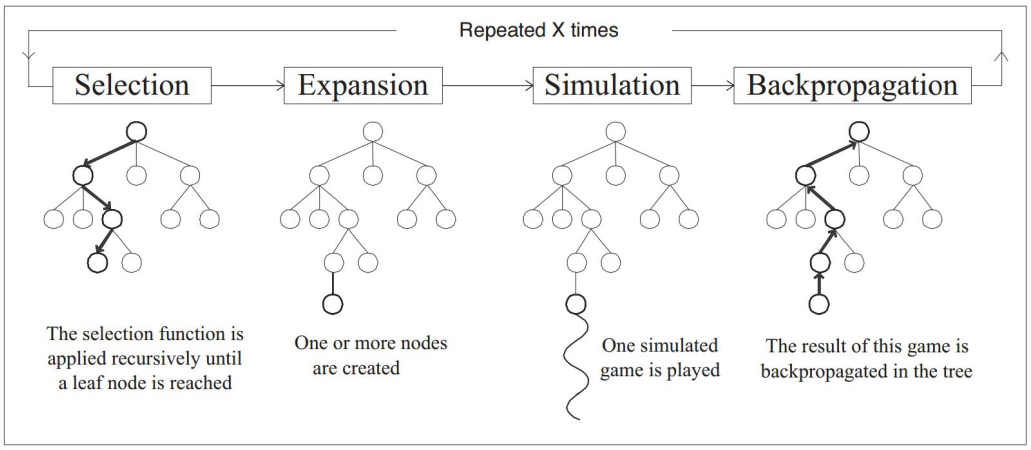

FIGURE 3.2: The four phases of Monte-Carlo Tree Search

In the above equation,$\overline{R(c)}$ is the average reward of node c. $visit(n)$ is the number of times node n has been visited. $visit(c)$ is the number of times node c has been visited. C is a constant to balance between the first term and the second term. The first one encourages choosing known good nodes, i.e. encourages exploitation. The second one encourages choosing less visited nodes, i.e. encourages exploration.

Monte-Carlo Tree Search has been applied to real-time games. In Ms Pac Man vs Ghost, several attempts have been made in using MCTS and some have found great successes. However, in those attempts, most divides the continuous game into small discrete steps, such as when a character meets a junction or another character, and then applies the MCTS to the now discrete game [21] [22]. Tron is another real-time game where MCTS has been applied. Once again, the game is conveniently divided into small step and the typical MCTS is applied [23] [24].

3.3 Reinforcement Learning

Reinforcement Learning (RL) is a class of learning methods in which an agent learns by trial-and-error through interacting with the environment. Its history dated back to the early days of Artificial Intelligence and has been applied to creating game AIs many times. The most famous achievement of RL in game AIs is probably the Checkers program [25] by Samuel and TD-Gammon [26] by Tesauro. Recently, another huge achievement with RL is [4] and [5] from Google DeepMind group. Outside of game playing, RL is often applied to Robotics and Control [27].

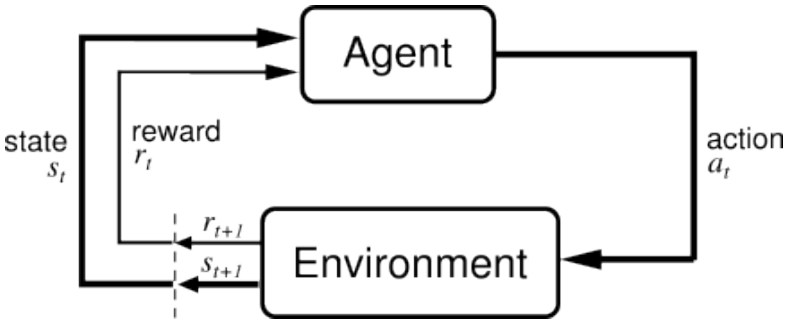

FIGURE 3.3: The agent-environment interaction in reinforcement learning

Reinforcement Learning methods are methods that take advantage of some special properties of Reinforcement Learning problem. In a RL problem, there usually are a learner, also called an agent, an environment for the agent to interact with and a goal that the agent needs to achieve. Usually, the goal is to maximize the expected sum of immediate rewards that the agent can accumulate overtime by interacting with the environment. Figure 3.3, taken from [28], shows the relation between components of a RL problem. Here, the agent interacts with environment in discrete time step $t = 0, 1, 2...$ Formally, we can define a RL problem as:

- A set of states of the environment S
- A set of actions available to the agent A
- A set of scalar rewards, often is real number

The agent need to learn a optimal policy $\Pi = S \mapsto A$ that maximizes the goal.

A RL method can be viewed by how rewards are calculated. There are 3 common ways [29]:

- **Finite horizon model**. The agent try to maximize the expected reward in a fixed limit l: $E(\sum_{t=0}^{l} r_t)$. The limit could be time or number of transition between states.
- **Infinite horizon discounted model**: the agent try to maximize the expected reward in unlimited time with rewards in the future have a geometrically discounted factor $\gamma, 0 < \gamma < 1$: $E(\sum_{t=0}^{\infty} \gamma^t r_t)$

- **Average reward model**: the agent try to maximize the average of intermediate reward in unlimited time: $\lim_{h\to\infty} E(\frac{1}{h}\sum_{t=0}^{h} r_t)$

Different view of the reward may lead to different policy learned.

A RL method can also be viewed by how it learn the optimal policy.

- The first way is to learn a value function V. A value function could be the expected long term reward for a state $V(s)$,or a pair of state and action $V(s, a)$. Once established, the agent can execute the optimal policy by choosing the action that lead to state with maximum expected value.
- Another way is to learn the optimal policy directly. The agent may start with arbitrary policy π. Then, it try to calculate a value function $V(s|\pi)$ that results from the policy π and try to improve π by assuming $V(s|\pi)$ is the correct value function, as in learning through value function. The above procedure is repeated until a desired policy is obtained.

Chapter 4

Our Approach

We design our player with 3 goals:

- The player will be able to play competitively in most situations.
- The player will have the ability to improve on its own.
- The player will can quickly adapt to new opponent.

We use Monte-Carlo Tree Search to build a flexible player, which can satisfy the first goal. We improve the player with Reinforcement Learning to satisfy the second goal. The third goal is currently left out, we will focus on that in near future. In this chapter, we describe our player and how the first two goals are achieved. First, the Monte-Carlo Tree Search module is described. Then, the Reinforcement Learning and how the two modules work together are presented.

4.1 Real-Time Monte-Carlo Tree Search

This year, the organizer of Fighting ICE competition released a new based AI, called MctsAi. As the name suggested, it uses Monte-Carlo Tree Search. The new based AI is decently strong. It can defeat 3 of the top 4 competitors in last year tournament, losing only to the champion. We have investigated it and seen that its method is sound. Therefore, we based our player on it. First, we describe the detail of the base MctsAi. Then, we describe how we improve upon it.

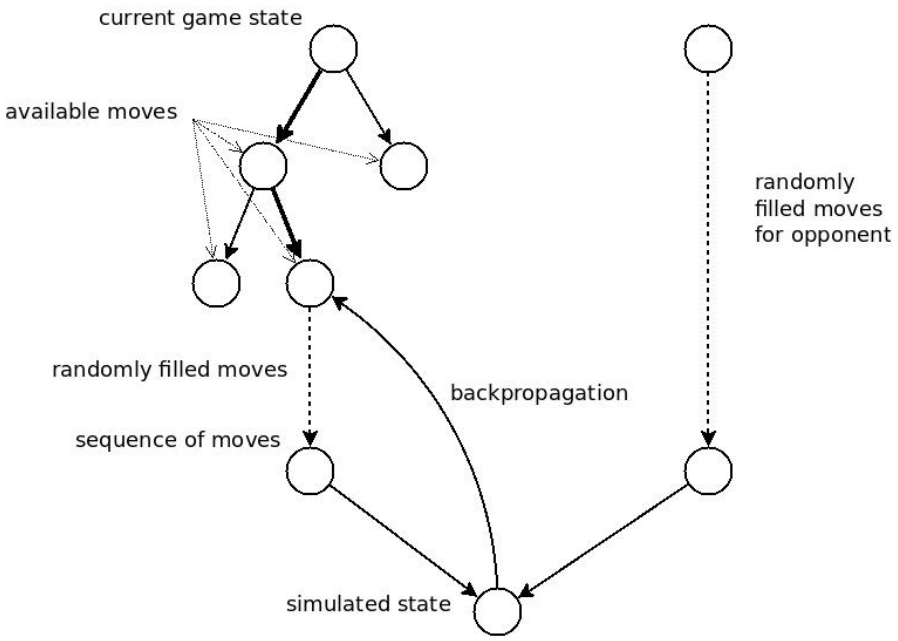

FIGURE 4.1: The search procedure of MctsAi.

4.1.1 MctsAi

In Fighting ICE environment, the convenient method of dividing time into well-defined steps is not effective. There are more than 30 possible actions for the player to choose. Each action has different execution times. Thus, it is possible but very inefficient to do so. In MctsAi, they have to relax the search procedure because of the above reason. Instead of the common game tree with 2 players, they build a one-player tree. The root node is still correspond to current game situation. However, the child node just represents one possible actions from current situation without concrete game information. A leaf node becomes a sequence of moves of a player from current game situation. A playout from a leaf node is carried out as a simulation from current game situation for 60 frames with 2 sequences of moves. One sequence is the sequence from the root node to current leaf node with filled random chosen moves to have a predefined length. The sequence for the opponent is a randomly generated sequence of possible moves of the same length. Since the playout will not reach end game, the simulated game situation is evaluated by a simple evaluation function which returns the amount of HP difference changed from current game situation. Figure 4.1 summarizes the search procedure of MctsAi.

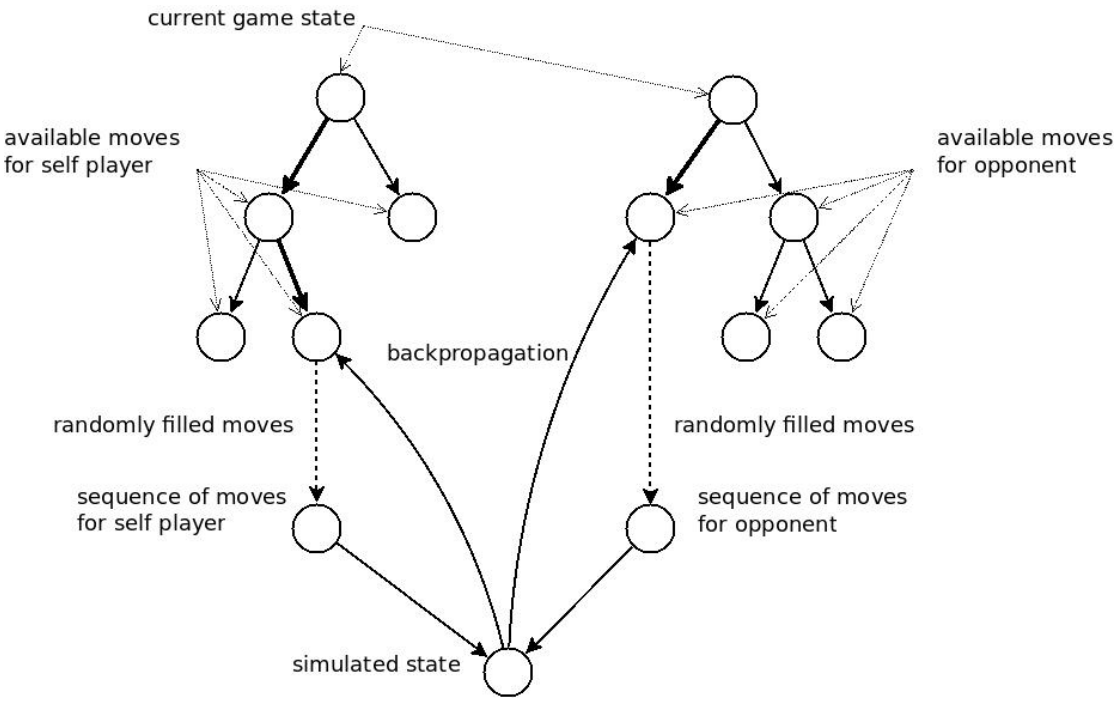

FIGURE 4.2: The search procedure of our player. Improved from MctsAi.

4.1.2 Our Improvements

Our first improvement is a better opponent model. In MctsAi, the opponent is essentially modeled by a random AI. We think it could be better simply by mirroring the tree search procedure for the opponent. The new procedure is shown in Figure 4.2 Our investigation shown that the new player is slightly better than MctsAi. In a 60 rounds test, the improved MctsAi won 32 rounds vs 28 rounds won by MctsAi. It also have better records again Machete, the last year champion. Also in 60 rounds test, MctsAi loss 23-37, while the improved one loss 27-33. The result is significant enough for us to conclude about the effectiveness of the improvement. We also believe that it will become even better with the second improvement.

The second improvement is reducing the number of considered moves in a game state. Without any reduction, for a state, there are about 20 possible moves. For 16ms, we found out that only about 350 simulations can be run. That number is quite small. We have tried to improve that but it is very hard because of the limitation of using only a single thread and the high computational cost of using the simulator. Therefore, to adequately explored a move, the number of considered moves in a game state must be reduced. Reinforcement Learning is chosen to prune less promising moves. Details on how we use Reinforcement Learning is presented in the next section.

4.2 Reinforcement Learning with Monte-Carlo Tree Search

The task of our Reinforcement Learning module is to suggest promising moves in any given states. Our MCTS will only choose from those suggested moves. By doing so, the moves will be explored more carefully by MCTS and their estimated values will be more accurate. After that, the estimated values from MCTS will be used by RL to improve the suggestions. Figure 4.3 summarizes the above procedure.

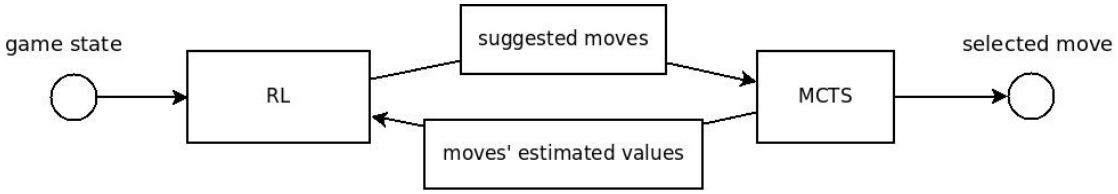

FIGURE 4.3: The design of our player. Reinforcement learning module suggests moves for Monte-Carlo Tree Search and uses the evaluated values to update.

Our RL learns a state-action value function, which estimates the amount of HP gained or lost of our player in a small time window, which is experimentally chosen to be 1s in current implementation. The endgame score is not chosen because the long time of a match and the large number of actions and states make estimating endgame score very hard. Also, we think that there is little relation between states that are several seconds apart in this game.

We represent a game state s as a vector of several chosen features of two players such as distance, energy, etc. For the purpose of generalization, similar game states are sorted into a hashed state $H(s)$. Thus, all state-action values of states with same $H(s)$ is stored together using a look-up table.

To select moves, we rank moves in descending order by UCB1 and select the top moves. We use a similar formula as in MCTS:

$$UCB1(m) = \overline{R}(H(s), m) + C * \sqrt{\frac{ln\ visit(H(s))}{select(H(s), m)}}$$

$\overline{R}(H(s), m)$ is the average state-action values of moves m in hashed state $H(s)$. $visit(H(s))$ is the number of times hashed state $H(s)$ has been visited. $select(H(s), m)$ is the number of times move m has been selected by RL in hashed state $H(s)$.

After evaluated by MCTS, all suggested moves will be updated in the same way as in MCTS.

$$select(H(s), m)_{t+1} = select(H(s), m)_t + 1$$

$$visit(H(s))_{t+1} = visit(H(s))_t + 1$$

$$\overline{R}(H(s), m)_{t+1} = \overline{R}(H(s), m)_t + \frac{1}{select(H(s), m)} * r_{t+1}$$

Currently, we do not use any real game results, so there are some oversight in the estimation of MCTS. However, we think that it is not very important. As several moves are suggested, there are a high chance that a good move is presented to MCTS. With less moves to considered, the speed and accuracy of MCTS will be enhanced and RL will benefit from more accurate estimations and multiple updates.

Chapter 5

Result and Discussion

We have described our player in previous chapter. In this chapter, we report how we train our player. Then, we compare the performance of our player with five others and present the results. After that, we discuss about strengths and weaknesses of our player and future improvements.

5.1 Training

Our player have been trained for 10000 rounds. Currently, it only trained for the character Zen because of the lack of comparable AIs for other characters. We choose five opponents for the training: Machete, Jay_Bot, Ni1mir4ri, RatioBot and MctsAI. The first four are the top competitors of the 2015 Fighting ICE Competition, taken from the competition official website[1]. The last one, MctsAI, is the base player for this year competition. Our player repeatedly played with each of them for 2 matches before changing to the next. Since our reinforcement learning module only used estimated results from Monte-Carlo Tree Search, our training method is essentially self-play. Different opponents only help advance the game to different states. Figure 5.1 shows the change of average score of our player during the training. It can be seen that our player started at a relatively good performance with average score of 350 in matches with the others. It is because of the ability of Monte-Carlo Tree Search. Even when only several randomly chose moves are nominated for a given state, MCTS can choose the best of them. The

[1]http://www.ice.ci.ritsumei.ac.jp/ ftgaic/index-R.html

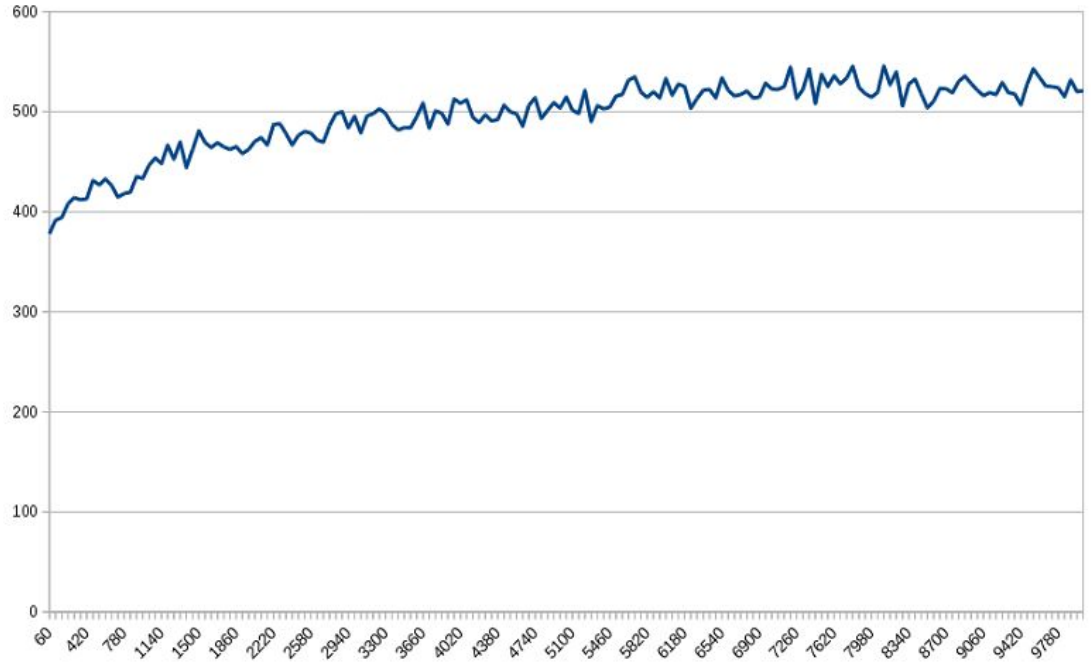

FIGURE 5.1: Change of average score of 60 rounds of our player in training period. The score varies from 0 to 1000.

	Machete	MctsAi	Jay_Bot	Ni1mir4ri	RatioBot	Our player
Machete		**76**	46	59	**84**	56
MctsAi	44		**100**	**69**	**90**	**73**
Jay_Bot	**74**	20		16	**96**	46
Ni1mir4ri	**61**	51	**104**		**99**	56
RatioBot	36	30	24	21		22
Our player	**64**	47	**74**	**64**	**98**	

TABLE 5.1: Comparing our player and 5 other players. Each plays with another for 120 rounds. The number on the cell shows the number of times that the player on the row won the player on the column. Numbers larger than 60 are marked bold.

score peaks after 6000 rounds. To further improve it, we need to improve on MCTS and state representation of RL.

5.2 Experiment

We compare our player's performance with the same 5 players in training. A player will fight 40 matches or 120 rounds with each other players. The results are shown on Table 5.1. We can see that, the main difference between our player and MctsAi is that while our player have better performance against Machete, MctsAi

has better performance against some others. We can also see the evidence of the rule-based approach in Machete, Jay_Bot, Ni1mir4ri, and RatioBot. RatioBot is rather weak and lost to all others. Machete lost to Jay_Bot and Ni1mir4ri while having best records against 2 Monte-Carlo Tree Search approaches. Jay_Bot won Machete and Ni1mir4ri won Jay_Bot with significantly high scores while lost to both search approaches. Our observation revealed that, our player has weaker resistance to projectile attacks than MctsAi and lost many matches because of that reason. And Jay_Bot, Ni1mir4ri and MctsAi all use projectile attacks frequently. The reason MctsAi frequently uses projectile attacks while our player does not may attribute to the more random nature of the opponent modelling in MctsAi. While our player may predict that the attack will not hit, MctsAi may predict otherwise because the opponent in MctsAi's perspective acts randomly. Moreover, these projectile attacks have high damage, consequently, high evaluated values will be returned by the evaluation function. In our player, that attack may not be suggested as a good move and, in the long run, the action to dodge that attack will also not be suggested. On the other hand, the more wins over Machete of our player are noteworthy. It shows that by limiting the number of considered moves, Monte-Carlo Tree Search has the potential to overcome strong rule-based approaches. After the remaining issues are solved, we can expect that our player can dominantly defeat the others.

5.3 Future Improvements

Having discussed the strengths and weaknesses of our player, we have identified several point for future improvements. The points in this section are points that we think most important and have significant impacts on the performance of our player.

- **Parameters tuning**. This is a importance aspect in any machine learning application. In our system, both Monte-Carlo Tree Search and Reinforcement Learning have their own set of parameters. In total, there are about 10 parameters in need of tuning. We have empirically set all of them, but still, we feel that the chosen parameters are not optimal.
- **Better state representation**. Currently, we choose a rather simple state representation. It is because of the limit of time as well as space. This

limits the refinement of the state and as a result, the refinement of the suggested actions. Another reason is the lack of generalization in the simple representation. For example, there are states which are only different by one variable which has little effect on the chosen action. However, our player does not know that and treats them totally unrelated. As a result, the learning speed is reduced. We would like to improve it by a better representation and we have been working on with a neural network representation which is expected to resolve both issues above.

- **Better in-game adaptation**. There are many different opponents and different play styles. Also, we cannot say there is an optimal way to defeat all of them. The way we limit our actions to speed up the search also limit our ability to quickly adapt to new opponents. Hence, this is also a much need improvement in our player.

- **Horizon effect**. Because the speed of the simulator is not great, the more we simulate, the more time it takes for a playout. Our experience shows that longer simulation time significantly reduce the number of playout per time frame. As a result, we agree with the 60-frame limit from MctsAi. But there are many cases where 60 frames is too short. The projectile attacks' issue discussed in previous section is one of the problems. Because projectile attacks generally last more than 3 seconds, which is 180 frames. Thus the chosen limit practically ignore most of projectile attacks that are far away. This is evident when we watching the AI plays. It seldom uses projectile attacks unless in close range, and it ignores opponent's projectile attacks that are far away and later could not dodge because of the chosen actions.

- **Improve Monte-Carlo Tree Search**. Monte-Carlo Tree Search is the core of our player. The strength of our player totally depends on the potential of MCTS. The purpose of Reinforcement Learning is to improve the speed and accuracy of MCTS. But, MCTS has certain weaknesses and those weaknesses still exists in our player. Thus, we need to improve MCTS in order to improve our player's strength.

Chapter 6

Conclusion

In this thesis, we have experimented a method of combining Monte-Carlo Tree Search and Reinforcement Learning in building a fighting game player in Fighting ICE environment. MCTS is the core of the AI strength. Both the game playing engine and feedback for RL rely on MCTS. The job of RL is to specify a small number of promising actions for MCTS, thus increase its accuracy on differentiating the best of those promising actions. The method shown its potential in its learning capability and improved strength. Although there exists some issues, it is still stronger than all top 4 AIs from last year competition.

Among the remaining issues, the most urgent one is state representation. We expect that a better representation as a neural network can solve multiple issues such as better generalization and finer distinction between states at the same time. For that reason, it has the top priority in our future works before the next Fighting ICE competition in this September.

In conclusion, we have shown that the combination of Monte-Carlo Tree Search and Reinforcement Learning is very suitable for fighting video games. The AI is more flexible and stronger than commonly used ruled-based approaches. It can also adapt to new opponents, although at current implementation, the speed of adaptation is not very fast.

Bibliography

[1] Louis Victor Allis, Hendrik Jacob Herik, and MPH Huntjens. *Go-moku and threat-space search*. University of Limburg, Department of Computer Science, 1993.

[2] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996.

[3] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[6] Fighting ICE. http://www.ice.ci.ritsumei.ac.jp/ ftgaic/index.htm.

[7] Feiyu Lu, Kaito Yamamoto, Luis H Nomura, Syunsuke Mizuno, YoungMin Lee, and Ruck Thawonmas. Fighting game artificial intelligence competition platform. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, pages 320–323. IEEE, 2013.

[8] Kevin Majchrzak, Jan Quadflieg, and Günter Rudolph. Advanced dynamic scripting for fighting game ai. In *International Conference on Entertainment Computing*, pages 86–99. Springer, 2015.

[9] Kaito Yamamoto, Syunsuke Mizuno, Chun Yin Chu, and Ruck Thawonmas. Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–5. IEEE, 2014.

[10] Chun Yin Chu and Ruck Thawonmas. Applying fuzzy control in fighting game ai. *77* , 4:02, 2015.

[11] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[12] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

[13] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201–240, 2002.

[14] Bernd Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[15] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[16] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[17] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, volume 8, pages 259–264, 2008.

[18] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[19] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[20] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[21] Spyridon Samothrakis, David Robles, and Simon Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

[22] Nozomu Ikehata and Takeshi Ito. Monte-carlo tree search in ms. pac-man. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 39–46. IEEE, 2011.

[23] Spyridon Samothrakis, David Robles, and Simon M Lucas. A uct agent for tron: Initial investigations. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 365–371. IEEE, 2010.

[24] Niek GP Den Teuling and Mark HM Winands. Monte-carlo tree search for the simultaneous move game tron. *Univ. Maastricht, Netherlands, Tech. Rep*, 2011.

[25] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[26] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995.

[27] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, page 0278364913495721, 2013.

[28] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[29] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.