Library Release Form

Name of Author: Shuyi Zhang

Title of Thesis: Improving Collectible Card Game AI with Heuristic Search and Machine Learning Techniques

Degree: Master of Science

Year this Degree Granted: 2017

. . . . . .

Shuyi Zhang
ATH 111
University of Alberta
Edmonton, AB
Canada, T6G 2G6

Date: . . . . . .

Improving Collectible Card Game AI with Heuristic Search and Machine Learning Techniques

by

Shuyi Zhang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Improving Collectible Card Game AI with Heuristic Search and Machine Learning Techniques** submitted by Shuyi Zhang in partial fulfillment of the requirements for the degree of **Master of Science** in *Computing Science*.

. . . . . .
Prof. Michael Buro

. . . . . .
Prof. Martin Mueller

. . . . . .
Prof. Scott Dick

Date: . . . . . .

To Love, Peace, and the Brotherhood of Man

Preface

All the work conducted in this thesis is under the supervision of my supervisor, Professor Michael Buro. Chapter 3 and 4 are published as S. Zhang and M. Buro, "Improving Hearthstone AI by learning high-level rollout policies and bucketing chance node events," in IEEE Conference on Computational Intelligence in Games (CIG 2017). I also appreciate all the helpful ideas and advice from David Churchill, Marius Stanescu, Nicolas Barriga, Christopher Solinas, Douglas Rebstock, and many other people who helped me during my graduate study.

Abstract

Modern board, card, and video games are challenging domains for AI research due to their complex game mechanics and large state and action spaces. For instance, in Hearthstone — a popular collectible card (CC) (video) game developed by Blizzard Entertainment — two players first construct their own card decks from hundreds different cards and then draw and play cards to cast spells, select weapons, and combat minions and the opponent's hero. Players' turns are often comprised of multiple actions, including drawing new cards, which leads to enormous branching factors that pose a problem for state-of-the-art heuristic search methods.

This thesis starts with a brief description of the game of Hearthstone and the modeling and implementation of the Hearthstone simulator that serves as the test environment for our research. Then we present a determinized Monte Carlo Tree Search (MCTS) based approach for this game and two main contributions of this approach. First, we introduce our chance node bucketing method (CNB) for reducing chance event branching factors by bucketing outcomes with similar outcomes and pre-sampling for each bucket. CNB is incorporated to the in-tree phase of the determinized MCTS algorithm and improves the search efficiency. Second, we define and train high-level policy networks that can be used to enhance the quality of MCTS rollouts and play games independently. We apply these ideas to the game of Hearthstone and show significant improvements over a state-of-the-art AI system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years there have been remarkable game artificial intelligence (AI) research achievements in challenging decision domains like Go, Poker, and classic video games. AlphaGo, for instance, won against Ke Jie who is currently the No.1 Go player in the world with the help of deep networks, reinforcement learning, and parallel Monte Carlo Tree Search (MCTS) [1], and recently an AI system based on deep network learning and shallow counterfactual regret computation running on a laptop computer won against professional no-limit Texas Hold'em players [2]. In addition, deep Q-learning based programs have started outperforming human players in classic Atari 2600 video games [3]. However, modern video strategy games, like collectible card (CC) or real-time strategy (RTS) games, not only have large state and action spaces, but their complex rules and frequent chance events also make the games harder to model than traditional games. Thus, it is challenging to build strong AI systems in this domain, and the progress has been slow.

1.1 Hearthstone as a Test-Bed

CC games are a sub-genre of video strategy games. They feature complex game rules and mechanics. In this kind of game, hundreds of unique cards with different special effects make the game fun to play yet difficult to master for human players. In 2017, there are millions of people playing video CC games online and a lot of professional players play in tournaments all around the world. Hearthstone, a CC game initially released by Blizzard in 2014, is currently the most popular video CC game. This game has many interesting properties besides its action and state complexity, such as non-determinism, and partial observability. Lastly, thanks to its big fan base, there are many open-source Hearthstone game simulators available online. Having these open-source projects makes it possible to do AI research in CC games.

1.2 Research Challenges and Related Approaches

To build strong AI players for computer strategy games, especially CC games, we have the following difficulties to overcome:

- **Complex Game Mechanics**

  Computer games have more complex rules and game mechanisms compared with traditional games. The game state of computer strategy games usually consists of multiple sub-states like resources, technologies, and armies. The number of types of actions is also larger than in traditional games. For example, the only type of action in Go is placing a stone. In contrast, players in Hearthstone can execute more types of actions like minion attack, hero attack, and playing cards.

  The complexity of game mechanics causes difficulties in the implementation of simulators. In Hearthstone, we need to have scripts for all different cards since each of them has unique special effects. The complex rules require the implementation of many testing modules to make sure the game logic works correctly. Another important drawback is that the "undo move" functionality is hard to implement due to the complex mechanics. Without the "undo" function, we have to copy states during the search and this slows it down.

  It's fortunate that there are a lot of open-source simulators of popular computer games. Since those games are usually closed-source, the engineers put a lot of efforts in remaking the entire game from their game playing experiences. In the case of Hearthstone, there are simulators like Silverfish [4], Metastone [5], and Nora [6], which provide much help to AI researchers in this area.

- **State and Action Space Complexity**

  Due to multiple sub-states in computer games, players often have to consider multiple objectives during their gameplay. CC game players, for instance, need to manage different aspects including mana resources, hand resources, army composition, or even individual combat units at the same time. As solving each sub-problem alone can be computationally hard already, having to deal with multiple objectives in strategic computer games is compounding the complexity.

  It is therefore infeasible to apply heuristic search algorithms to the original search spaces, and abstractions have to be found to cope with the enormous decision complexities. In the past few years several ways for reducing search complexity have been studied. For instance,

Hierarchical Portfolio Search [7]'s idea is to utilize scripts to reduce search complexity. It considers a set of scripted solutions for each sub-problem to generate promising low-level actions for high-level search algorithms. Likewise, *Puppet Search* [8] instead of searching in the original game's state space, traverses an abstract game tree defined by choice points given by non-deterministic scripts. Lastly, in [9] simple scripts for generating low-level moves for MCTS are used for reducing the branching factor in the CC game "Magic: The Gathering."

- **Large Branching Factor caused by Chance Events**

  In addition to large branching factors in decision nodes, many modern games feature chance events such as drawing cards, receiving random rewards for defeating a boss, or randomizing weapon effects. In Hearthstone, chance events are everywhere such as summoning a random minion, or cast a random spell to random targets. If the number of chance outcomes is big, the presence of such nodes can pose problems to heuristic search algorithms such as *ExpectiMax* search or the in-tree phase of MCTS, even for methods that group similar nodes and aggregate successor statistics [10] or integrating sparse sampling into MCTS [11].

1.3 Thesis Goals and Contributions

1.3.1 Software

The first goal of this thesis was to design and implement a fast simulator for the game of Hearthstone. It can simulate the game of Hearthstone including the player settings, card settings, and game loop. At the same time, we expected it to meet the requirements of clear code design and fast execution speed since these are beneficial for later research and code reuse. In Chapter 2 we present our software contribution, the Hearthstone simulator based on open-source software. It supports fast complete game simulation and custom AI agent implementation, and serves as the test environment we used in this thesis.

1.3.2 Improving AI Strength

The second goal was to improve the AI strength in the game of Hearthstone. The built-in AI players in many video games are considered weak, but there are still some strong state-of-the-art AI players developed by 3rd-party authors. We aimed to design a general approach (algorithm) to create a strong AI player for CC games. Then using Hearthstone as a test bed, we apply our approach to it and try to beat the state-of-the-art AI players. In Chapters 3 and 4, we show how we applied our approach to a state-of-the-art Hearthstone AI system to improve its playing strength.

1.3.3 Flexible Game AI Design

The recent successes of using deep machine learning to tackle complex decision problems such as Go and Atari 2600 video games [1,3,12] have inspired us to study how such networks can be trained to improve the AI playing strength in CC games. Also, unlike traditional games, video games are frequently updated. When using rule-based AI systems, developers may therefore need to rewrite AI scripts according to the patches. In such cases, a self-improving AI approach can save more human resources compared with changing the scripts manually. We were therefore motivated to design an approach that can be generally applied to Hearthstone and later updates without much manual tuning. In Chapter 4 we present an end-to-end machine-learning based approach that can be used in CC games to improve the AI playing strength for different card decks.

1.4 Thesis Outline

In Chapter 2, we first describe the game mechanics of our research test-bed, Hearthstone. Then we describe the implementation of one of the state-of-the-art Hearthstone AI player, Silverfish [4], and its simulator and essential parts of the modeling and implementations of our Hearthstone simulator based on Silverfish's. Chapter 3 first describes the details of the Determinized MCTS algorithm applied to the game of Hearthstone. Then we introduce the chance node bucketing (CNB) method that can deal with the problem of large branching factors in CC games. At the end of Chapter 3, we show that empirically DUCT combined with CNB can improve the AI strength. Chapter 4 explains how to apply machine learning techniques to improve the MCTS rollout policies in Hearthstone. Chapter 3 and 4 are based on our recent paper [13] presented at IEEE's 2017 Conference on Computational Intelligence in Games (CIG 2017). Chapter 5 concludes the thesis and discusses possible future work to improve CC AI systems even further.

Chapter 2

Hearthstone AI Systems

In this chapter, we first describe the game of Hearthstone, which is one of the most popular CC video games, to make the reader familiar with the game for which we will later present experimental results. In the second part we introduce previous work on simulators and AI systems for Hearthstone and the implementations of our Hearthstone simulator.

2.1 Game Description

2.1.1 Key Concepts

Hearthstone is a 2-player turn-based zero-sum strategy game with imperfect information. It starts with a coin flip to determine which player will go first. Players then draw their starting cards from their constructed 30 card decks. In regular games neither player knows the opponent's deck initially. The game GUI is shown in Fig. 2.1. The key concepts in Hearthstone are:

• **Mana crystals.** Mana crystals (mana) are needed to play cards from the hand. On the first turn, each player has one mana. At the beginning of each turn, the limit of each player's mana is increased by 1, and all the mana are replenished.

• **Game state.** The game state has seven components: 2 heroes, the board, 2 hands, and 2 decks. The **hero** is a special type of minion that has 30 health points (HP). A hero can only attack when equipped with a weapon and the number of attacks depends on the weapon. The game ends if and only if one hero's health value is ≤ 0. The **board** is the battlefield where minions can attack each other. It is important to evaluate who is leading on the board because, in most games, the winning strategy is to take control of the board by trading minions and then using the minions on the board to defeat the opponent's hero. In their **hands** players hold cards that are hidden from the opponent. A player can use minion cards to capture the board or use spells to remove his opponent's minions and deal damage to the opponent's hero. Usually, having more cards in their hand allows players to handle more complex board configurations. However, just holding cards without playing them may

Figure 2.1: Hearthstone GUI

Player 1: (1 hand) (2 mana) (3 hero) (4 minions) (5 deck)
Player 2: (6 hand) (7 mana) (8 hero) (9 minions) (10 deck)

lead to losing control of the board.

The **deck** is a collection of cards that have not been drawn yet. If a player plays all cards without ending a game, he will take fatigue damage every time he needs to draw a card from the deck. In professional tournaments held by Blizzard, players usually know the opponent's deck. Therefore, in the experiments reported later, we assume the same condition.

• **Cards.** Cards represent actions that a player can take by playing that card and consuming mana crystals. There are three main types: minion, spell, and weapon cards. **Minion** cards are placed into the board area. When a minion card is played, a minion is summoned according to the description of the card. Summoned minions have HP and attack (ATK) values and can attack Heroes and other minions. Most minions have unique abilities (e.g. minions with "Taunt" ability can protect their allies by forcing the enemy to deal with them first). If, for instance, the minion card "Mechwarper" (Fig. 2.2) is played, a 2-mana "Mechwarper" minion with 2 ATK and 3 HP is summoned to the board . The minion combat happens when one minion attacks another. Each attacked minion loses HP equal to the other minion's ATK. If the HP of a minion becomes smaller or equal to 0, the minion will die.

Spell cards are played directly from a player's hand and have an immediate special effect. For example, when a 4-mana "Fireball" (Fig. 2.3) spell card is played to a minion or one player's hero,

Figure 2.2: Minion card "Mechwarper"

Figure 2.3: Spell card "Fireball"

it will instantly deal 6 damage to the target. **Weapon** cards, like spells, are also played straight from a player's hand. They add a weapon to a player's arsenal allowing him to attack directly with his hero.

2.1.2 Action Types

The actions in Hearthstone can be categorized as follows:

- **Card-play.** Card-play is a type of action that the active player (P_a) chooses to play one playable card from the hand. Note that a card is playable when P_a has enough mana to play the card, and the game state meets the prerequisites of the card (e.g. the card "Execute" can only be played if there are opponent's (P_o) minions on the board). In addition, we introduce the functional form of actions for the sake of simplicity. The functional form of card-play action is $CP(C)$ where C is a playable card in P_a's hand.

- **Target-selection for a card.** Some cards require a target after being played. In this case, P_a needs to choose a target for the card. The functional form is $TS(C, T)$ where C is the card to play and T is the target.

- **Target-selection for a minion.** After being summoned to the board, a minion will sleep for one turn. In the next turn, the minion's status changes to "ready" which means that the minion can attack the opponent's minions or hero. The active player needs to choose one target for a

"ready" minion. The functional form is $TS(M, T)$ where M is the minion controlled by P_a and T is the target.

- **End turn.** P_a can end the turn proactively anytime during his turn. When P_a runs out of available actions, P_a is forced to terminate the turn. The functional form is $ET()$.

2.1.3 Game Play

- **Pre-game.** Before the game starts, two players will draw different numbers of cards from their decks. The player who goes first draws three cards and the player who goes second draws four cards and gains a special card called "The Coin". Both players can then swap out any of their starting cards for other cards from the top of their deck. The cards they swap out are then shuffled back into the deck.

- **Game Turn.** Before a turn starts, the system draws one card for the active player. He can then choose which cards to play (card-play actions) subject to mana availability. Some card-play actions will be followed by a target-selection action. The player can also select a minion to attack an opponent's minion. Players usually end turns when their objective has been accomplished or there are no more actions available.

- **Game End.** During any phase of the game, if a hero's HP value drops to 0 or below, the game ends. When the game ends, a player wins if the player's hero is alive. A draw can happen if both players' heroes die simultaneously (e.g. both heroes die from an area effect spell).

To illustrate these concepts we give an example of a game turn. The starting state is shown in Fig. 2.4: the active player (P_a) has 2 minions M_1 and M_2, and the opponent (P_o) has two minions M_3 and M_4 on the board; P_a has 6 mana available this turn and executes the following actions:

- Play the card "Fireball", which can deal 6 damage, to the M_4. This action kills M_4 because it only has 5 HP (Fig. 2.5).

- Choose M_1, which has 4 ATK and 5 HP, to attack M_3 with 2 ATK and 2 HP. M_1 takes 2 damage from M_3 and M_3 dies from 4 damage from M_1 (Fig. 2.6).

- Play the "Mechwarper" card on the board. This action summons the "Mechwarper" minion (M_5) which has the effect that all "Mech" minion cards in P_a's hand will cost 1 mana less (Fig. 2.7).

Figure 2.4: Turn start

Figure 2.5: Play "Fireball" card to M_4

Figure 2.6: Choose M_1 to attack M_3

Figure 2.7: Play "Mechwarper" to summon M_5

Figure 2.8: Choose M_2 to attack P_o's hero

Figure 2.9: End player's turn

- Choose M_2, which has 5 ATK and 4 HP, to attack opponent's hero with 24 HP, which takes 5 damage. M_2 takes no damage because minions don't take damage from attacking heroes (Fig. 2.8).

- End turn (Fig. 2.9).

The functional representation of the action sequence is

$$[CP(C_{Fireball}), TS(C_{Fireball}, M_4), TS(M_1, M_3), CP(C_{Mechwarper}), ET()]$$

.

2.2 Hearthstone Simulators and AI Systems

This subsection describes Hearthstone simulators and AI systems including the state-of-the-art AI player, "Silverfish".

• **Nora** is a Hearthstone AI player that learns from random replays using a random forest classifier to choose the action [6]. It is able to defeat the random player in 90% of the games but it still loses against simple scripted players. Nora's game simulator models an early version of Hearthstone.

• **Metastone** is a feature-rich and well maintained Hearthstone simulator [5], that features a GUI and simple AI systems, like greedy heuristic players, within the simulator, but its playing strength is not very high.

• **Silverfish** is a strong search-based Hearthstone AI system [4]. It features a powerful end-of-turn state evaluation that has been tuned by human expert players, a move pruning system, an opponent modeling module that can generate commonly played actions, and a 3-turn look-ahead search module that utilizes opponent modeling. Silverfish's simulator is compatible with Hearthstone Blackrock Mountain (BRM) expansion pack[1]. Silverfish can beat rank-10 players, which is considered above the average human player strength.

2.3 Silverfish's AI System

Silverfish is one of the best AI players in Hearthstone: BRM version, the AI system benefits from its knowledge database and search algorithm. In this section, we describe Silverfish's AI systems components.

[1] `http://hearthstone.gamepedia.com/Blackrock_Mountain`

2.3.1 Silverfish's Move Generator

Silverfish's move generator enumerates all available moves, and meanwhile prunes moves by using a rule-based pruning function written by expert players. In this way, bad moves like "play 'Fireball' card on P_a's hero" will not be returned.

2.3.2 Silverfish's State Evaluation Function

Silverfish has an end-of-turn state evaluation function to evaluate the state in the view of P_a who ended the turn. This evaluation function combines the following sub-evaluations:

- **The global Feature Evaluation Function** evaluates global features including two players' mana, HP values, the numbers of hand cards, and total HP and ATK on the board, and P_a's cards drawn and damage lost during the turn.
- **The Board Evaluation Function** evaluates the advantage of P_a's minions on the board over P_o's minions. The returned value is the result of P_a's board score minus P_o's board score.
- **The Hand Evaluation Function** evaluates P_a's hand score after the turn ends.
- **The Action Evaluation Function** evaluates how good the actions played were during the last turn. For instance, using the card "Fireball" on a 1 HP minion is considered a bad play because it overkills the minion by 5 damage.

Silverfish's end-of-turn state evaluation will take the linear combination of the evaluation scores above to get a overall evaluation of an end-of-turn state. The weights were hand-tuned by experts from the Hearthstone AI community.

2.3.3 Silverfish's Opponent Modeling Module

Silverfish has an Opponent Modeling Module (OMM) to handle the imperfect information problem of Hearthstone. In order to perform the search algorithm of Hearthstone, this module enumerates a set of highly possible card-play actions based on expert knowledge at P_o's turn to simulate P_o's plays. For example, OMM will play area-of-effect (multiple enemies can be affected) spells or summon powerful minions to mimic all possible strategies from P_o.

2.3.4 Silverfish's Search Algorithm

Silverfish's search algorithm works as follows. In P_a's turn, Silverfish uses the move generator to generate promising move sequences to the end of the turn. During P_o's turn, Silverfish uses OMM to generate the possible imperfect information moves, and still get perfect information moves by using

Algorithm 1 Silverfish's MiniMax

```
1:  procedure SF-MINIMAX(d, n)
2:      if d = 0 or n.GameEnd then
3:          return Eval(n)
4:      end if
5:      if n is P_a's turn then
6:          best ← ∞
7:          children ← GENERATEMOVE(n)
8:          for child in children do
9:              v ← SF-MINIMAX(child, d − 1)
10:             best ← max(best, v)
11:         end for
12:     else
13:         best ← −∞
14:         children ← GENERATEMOVE(n)
15:         for child in children do
16:             v ← SF-MINIMAX(child, d − 1)
17:             best ← min(best, v)
18:         end for
19:     end if
20:     return bestMove
21: end procedure
22:
23: procedure GENERATEMOVE(n)
24:     if n is P_a's turn then
25:         Play-card moves ← MoveGenerator.GetPCMoves(n)
26:     else
27:         Play-card moves ← OMM.GetPCMoves(n)
28:     end if
29:     Minion-attack moves ← MoveGenerator.GetMinionMoves(n)
30:     End-turn-moves ← enumerate(Play-card moves, Minion-attack moves)
31:     return End-turn-moves
32: end procedure
```

the move generator. Then the final move sequences are the permutations of perfect and imperfect information moves. At the turn level, Silverfish performs a Minimax search to find the best one among move sequences.

For the work reported in this thesis, we use Silverfish as the baseline to be compared with. Silverfish has a simulator that limits the AI to 3-ply searches. To compare with Silverfish, we added features to enable Silverfish to play complete games for specific decks. There are some difficulties in implementing Hearthstone AI: First, there are over 700 cards with different effects. For each card, we need to write specific scripts. Second, the game rules and mechanisms are complicated, and all the cards have special effects, so the simulator needs to have multiple checkers to handle all

the complex situations caused by action interactions. Even the real game itself is not bug-free. We spent considerable time on adding functions to the simulator to make it work in our experiments.

2.4 Implementations of the Hearthstone Simulator

Based on the Silverfish's simulator, we implemented our own Hearthstone simulator that can play the complete 2-player Hearthstone games. The following sections describe some important parts of our simulator's implementation. Note that the `Courier-font` text represents the variables or class names that appear in our implementation.

2.4.1 Cards

Cards are the most interesting part in Hearthstone because each card has a distinct effect. Therefore, the implementation of all cards is very complicated. In our simulator, each card is inherited from the `CardTemplate` Class and implements its `OnPlay()` method. For a minion card, the `OnPlay()` method will be called when an instance of a minion is summoned on the board. For a spell card, the `OnPlay()` method creates a corresponding instant effect on the board (e.g `call DealDamage()` on a minion) or players' hands (e.g. `DrawCards()`). Besides the `OnPlay()` method, there are also other callback methods like `OnDeathRattle()` (called when a minion is dead). If there are special effects when a minion is played, the method `OnBattleCry()` will be called. There are in total 732 card classes implemented in our simulator. We implemented over 100 card classes ourselves, and modified some of Silverfish's card implementations.

2.4.2 Minions

A minion has different attributes like ATK, HP, `isSilenced`, `isFrozen`, `divineShielded` and so on. We also use `List` to store the special effects on a minion. For instance, we use the `OnAttackEffectList` to keep the special effects to be triggered. When the minion attacks, the special effects will be triggered in first-in-first-out fashion. We also implemented specific functions to compute the results after a series of special effects being triggered.

2.4.3 Actions

The action object contains key attributes including `actionType`, `handcard`, `source`, and `target` where the `actionType` is an enum of `AttackWithHero`, `AttackWithMinion`, `PlayCard`, `UseHeroPower`, and `EndTurn`. The `handcard` variable is the reference to a hand card of P_a when `actionType` is `PlayCard` and null in other cases. When `actionType` is

Algorithm 2 GameLoop

```
1:  procedure GAMELOOP(P_1, P_2)
2:      state ← initializeGameState()
3:      currentPlayer ← P_1
4:      while Not state.GameEnd do
5:          time ← Time of a turn
6:          currentPlayer.updateState(state)
7:          while Not state.GameEnd or Not state.TurnEnd do
8:              move ← getMovesForPlayer(currentPlayer, time)
9:              time ← time−elapsed time
10:             if time > 0 then
11:                 state.doMove(move)
12:                 currentPlayer.updateState(state)
13:             else
14:                 state.doMove(EndTurnMove)
15:                 break
16:             end if
17:         end while
18:         currentPlayer = togglePlayer(currentPlayer, P_1, P_2)
19:     end while
20:     return state.GetResult()
21: end procedure
```

`AttackWithHero` or `AttackWithMinion`, `source` is the reference of the minion or hero to attack and `target` is a reference of the action target.

2.4.4 Game Loop

The pseudo code of the game loop is shown in Algorithm 2. Before the game starts, two instances of `PlayerAgent` are initialized as `playerOne` and `playerTwo`. The game is initialized by the `GameManager` class. The `PlayerAgent` class is extended into customized AI agents like `Silverfish`, which is modified from original Silverfish's AI, and `PlainMCTSPlayer` which is the vanilla MCTS player with no optimization. After the initialization of players, the main game loop starts as follows: The game state is initialized first; `playerOne` goes first and is set to the current player. `currentPlayer` first synchronizes his state with the public state. Then in a given frame of time budget (time or number of iterations), `currentPlayer` will do a sequence of moves until the turn ends, time is up, or the game ends. After the turn ends, the other player will become the `currentPlayer` in the next turn. The main game loop ends until the game is finished (win, loss, or draw).

2.5 Summary

In the first part of this chapter, we introduced the mechanisms of Hearthstone and demonstrate how the game-play of a turn works. Second, we described the AI system of Silverfish. Silverfish's search algorithm is a variant of the Mini-Max search algorithm with opponent modeling. Besides Silverfish's opponent modeling module, Silverfish also has a powerful end-of-turn evaluation function that is a rule-based evaluation with expert knowledge. Lastly, we described some key implementation details of our Hearthstone simulator, which serves as a test environment for later experiments. We made the design of our simulator simpler compared to other simulators with more features like Metastone so it can run simulations faster.

Chapter 3

Improving Silverfish by Using MCTS with Chance Event Bucketing

In this chapter we showcase how we improve Silverfish by using MCTS and bucketing chance events. We start by describing MCTS and the determinized UCT algorithm, which is a variant of determinized MCTS [14]. We then discuss the bucketing scheme we use to reduce the large chance node branching factors in Hearthstone. Lastly we present experimental results that indicate a significant performance gain comparing with the baseline.

3.1 Monte Carlo Tree Search

Monte Carlo Tree Search is a family of search algorithms for solving sequential decision problems. MCTS can be considered as a type of heuristic search in which the search direction is guided by the statistics of the results of a large number of rollout simulations. Since MCTS's invention around 2006 [15, 16], it has achieved great results in games that have large branching factors like Go. Additionally, its stochastic rollouts can implicitly handle the problem of randomness that appears in many video games.

The MCTS search algorithm can be decomposed into the following 4 phases:

- **Selection:** from the root node, a selection function is applied recursively to determine the next node to traverse until a leaf node is reached. This phase is also called the *in-tree* phase and the selection criterion is of referred to as *in-tree* policy.
- **Expansion:** after the selection phase, a leaf node is selected and one of its children is randomly added to the game tree for expansion.
- **Simulation:** starting from the leaf node selected in the previous phase, a rollout is run until the game ends or a depth limit is reached. The rollout is preformed based on a *rollout policy*

(*default policy*) which is uniform random in vanilla MCTS.

- **Back-propagation:** the result of the rollout simulation is backpropagated along all nodes in the path until it reaches the root.

Among all MCTS variants, Upper Confidence Bound applied to Trees (UCT) [16] is the most commonly used MCTS in-tree policy. It selects the next node to traverse based on the UCB1 [17] formula:

$$UCT(n) = Q(n) + c\sqrt{\frac{\log(N(n))}{N(p)}},$$

where $N(n)$ and $N(p)$ represent the visit count of node n and its parent node p respectively, $Q(n)$ is the average expected reward of node n so far, and c is a constant that balances exploitation and exploration.

Vanilla MCTS can be slow to converge to the optimal move and it cannot handle the problem of large branching factors well. Past research on improving MCTS can be categorized into two types: improving the *in-tree* policy and improving the rollout policy. Methods like *progressive bias* [18] and value initialization [19] introduce prior domain knowledge to the *in-tree* policy to generate better moves earlier. In a similar way, previous work on games like Go [1] and Hex [20] showed that an improved rollout policy can improve MCTS results significantly.

In the work presented in this thesis, we investigate the application of MCTS to the game of Hearthstone. We concentrate on improving the effectiveness of MCTS applied to games with large chance node branching factors and hierarchical actions by first reducing search complexity in the selection phase of MCTS, and then improving move selection in the simulation phase.

3.2 Determinized UCT (DUCT) for Hearthstone

Since Hearthstone is an imperfect information game, to improve Silverfish using search, we chose to use determinized search algorithms that yield good results in Contract Bridge [21], Skat [22] and "Magic: The Gathering" [23]. Specifically, we use a variant of determinized UCT (DUCT) [14], which is the UCT variant of Algorithm 3. This algorithm samples a certain number of worlds from the current information set in advance, and then in every iteration picks one and traverses down the sub-trees that fit the context of the world. If multiple worlds share the equivalent action, the statistics of that action are aggregated and used for action selecting based on the UCB1 formula. When the time budget is used up, the algorithm returns the most frequently visited move at the root node.

Algorithm 3 Determinized MCTS

```
1: procedure DETERMINIZED MCTS(I, d)
2:     // I: information to construct the information set, d: turn limit
3:     worlds ← Sample(I, numWorlds)
4:     while search budget not exhausted do
5:         for n in worlds do
6:             e ← TRAVERSE(n)
7:             l ← EXPAND(e)
8:             r ← ROLLOUT(l, d)
9:             PROPAGATEUP(l, r)
10:        end for
11:    end while
12:    return BestRootMove()
13: end procedure
14:
15: procedure TRAVERSE(n)
16:    while n is not leaf do
17:        if n is chance node then
18:            n ← SampleSuccessor(n)
19:        else
20:            n ← SelectChildDependingOnCompatibleTrees(n)
21:        end if
22:    end while
23:    return n
24: end procedure
25:
26: procedure ROLLOUT(n, d)
27:    s ← 0
28:    while n not terminal and s < d do
29:        s ← s + 1
30:        n ← Apply(n, RolloutPolicy(n))
31:    end while
32:    return Eval(n)
33: end procedure
```

3.2.1 Action Shared by Multiple Worlds

In Hearthstone, an action consists of 4 major parts: `actionType`, `handcard`, `source`, and `target`, where `handcard` is the reference of the card in P_a's hand, `source` is the reference of the attacking minion, and `target` is the reference of the target minion. In order to determine whether two actions are equivalent, we define two types of equalities in our implementation:

- **Strict Equality:** All attributes are recursively taken into account for equality check.
- **Soft (Hash) Equality:** We calculate the hash value of an action based on only some of its attributes, then compare the hash values of two actions to determine their equality. For instance, the attacking minion's `position` is only important when there is a minion that can buff adjacent ones, which does not happen in most decks, so the `position` attribute is

not taken into account for the hash calculation of an `AttackWithMinion` action.

We use Soft Equality we defined above to check whether actions are equivalent between worlds. For instance, if there is a_1 executed in $world_1$ and a_2 executed in $world_2$, if a_1 softly equal a_2, we consider them equivalent and their statistics are aggregated during the UCT selection phase.

3.3 Action Sequence Construction and Time Budget

In Hearthstone, P_a can play a sequence of actions in one turn. Therefore, we need to optimize the time budget management of our search algorithm to construct the best move sequence. Another difficulty is that the number of moves to be played is unknown, so we cannot distribute the search time equally among all moves. We investigate the following approaches of time budget strategy:

3.3.1 Multiple-Search Strategy

The first action sequence selection method allocates a time budget to search for the best move from the starting state n_0; after the time is up, it selects the move with the most visits, and then searches the next move in the same way. Finally, the best move sequence is constructed by the selected moves in multiple searches while reusing the tree to save time. In this method, we allocate a fraction $T \cdot \alpha_m$ of the remaining time T to each move. The time budget of the search starting from state n_i is

$$Time(n_i) = min(max(T \cdot \alpha_m, LB(n_i)), T \cdot \beta),$$

where the constant fraction β is greater than α_m and smaller than 1 to make sure we don't use up the remaining time, and $LB(n_i)$ is a lower bound of the time allocated for the search starting from n_i. The formula of $LB(n)$ is

$$LB(n) = \tau \cdot T / BF(n),$$

where τ is a fraction parameter between 0 and 1 and $BF(n)$ is the branching factors of state n. However in the case of $Time(n_{i+1}) > Time(n_i)$, we ensure that the search time for n_i is at least the same as n_{i+1} by adding an compensation time, $Time_{comp} = (Time(n_{i+1}) - Time(n_i))/2$, to continue the search starting from n_i.

3.3.2 One-Search Strategy

Another idea is instead of doing multiple searches, to try to do only one search and construct the best move sequence by recursively selecting the most visited child in current turn. However, if we return such a move sequence the last actions in this sequence may have low visit counts. In this case, we need to do an extra search starting from the node preceding the first rare move. We first

allocate a fraction, α_o of the remaining search time T for the initial search: $Time(n_0) = T \cdot \alpha_o$. After searching for $Time(n_0)$, we recursively select the most visited node to construct the move sequence until a node n_i is reached, whose visit count is smaller than ψ (a constant). If such node n_i exists, we start a new search from the node n_{i-1} using the multiple-search strategy; otherwise, the remaining time will be used to complete the original search with the starting node n_0.

Both one-search and multiple-search strategies spend more time on searching of initial moves than on later moves because the earlier moves have higher decision complexity. In later sections, we are going to compare these two methods empirically.

3.4 Utilizing Silverfish Functionality

Our DUCT search module utilizes Silverfish's rule-based evaluation function that was tuned by expert-level human players. This function only evaluates the end-of-turn game state by taking the hero, minion, hand, the number of cards drawn in the last turn, and penalty of actions executed during the last turn into account. We use this function in DUCT because it is fast (since it's rule-based) and comprehensive. We also expect it to provide good evaluations because it contributes to Silverfish's playing strength. We also use parts of the rule-based pruning code in Silverfish's move generator to prune bad moves, such as dealing damage to our hero.

Our algorithm uses rollout depth d. If the game ends within d turns following the starting state, 1 (win) or 0 (loss) is backed-up. If after d turns of simulation, the game has not ended and is in state n, we will call Silverfish's evaluation function to evaluate n and backup the evaluation value $r \in (0, 1)$.

3.5 Chance Event Bucketing and Pre-Sampling

In Hearthstone, Chance events can happen both before and during turns. Fig. 3.1 shows that the active player P_a's turn starts after drawing a card from his deck, and he can then play multiple actions including the ones with random outcomes until running out of actions or choosing to end the turn.

To mitigate the problem of high branching factors in chance nodes we propose to group similar chance events into buckets and reduce the number of chance events by pre-sampling subsets in each bucket when constructing search trees. Fig. 3.2 describes the process by applying above steps to a chance node C with $S = 12$ successors. To reduce the size of the search tree we form $M = 3$ buckets containing $S/M = 4$ original chance events each. We then pre-sample $N = 2$ events from each bucket, creating $(S/M) \cdot N = 6$ successors in total which represents a 50% node reduction.

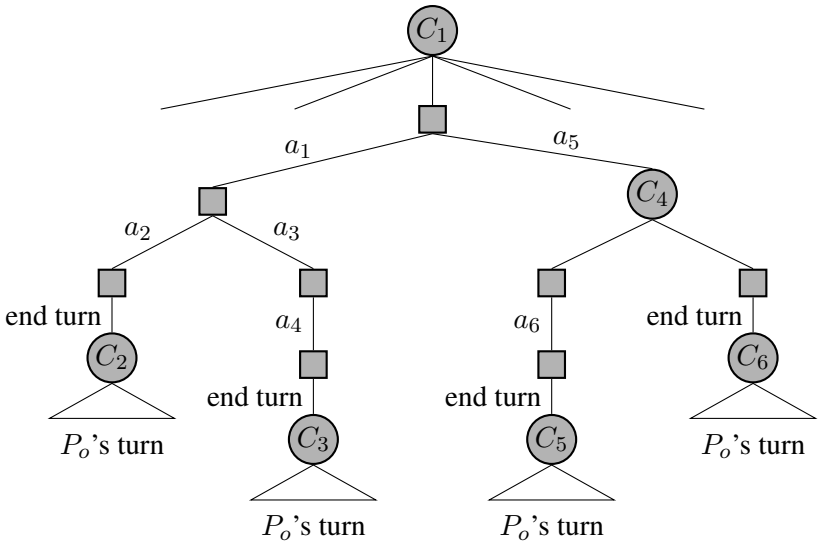

Figure 3.1: A sub-tree representing a typical turn in a Hearthstone game. P_a is to move after a chance event (e.g., drawing a card). **Squares** represent P_a's decision nodes, **circles** represent chance nodes, and edges represent player moves or chance events. After P_a ends the turn, P_o's turn is initiated by a chance node (C_2, C_3, C_5, C_6).

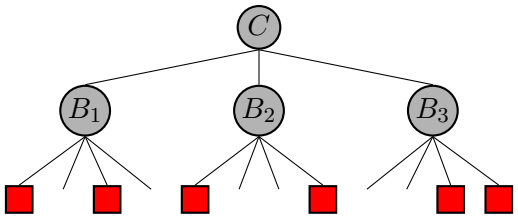

Figure 3.2: Bucketing and pre-sampling applied to a chance node C with 12 successors. There are $M = 3$ buckets abstracting $12/M = 4$ original chance events each. Among those $N = 2$ samples are chosen for constructing the actual search tree (red nodes).

In practice, the probability of each bucket is different and search agents should consider each bucket according to its probability. For the extreme case of a very skewed distribution, we can allocate a greater sample budget to the larger buckets and a lesser budget to the smaller ones. Also, M and N should be chosen with respect to the search space and bucket abstraction. For simple state abstractions, M can be small. If the nodes in the buckets are very different, N can be large. Also, there is a trade-off between more accurate sampling and smaller search efficiency when choosing the value of M and N.

3.5.1 Chance Events in Hearthstone

In Hearthstone's BRM version, chance events can happen in the following cases:

- **Card-drawing:** P_a draws one or more cards from the deck in a row (in one action). Card-drawing, which happens every turn, is the most frequent chance event. In extreme cases like

drawing 4 cards from the deck by using the "Sprint" card, a card-drawing event can produce over a thousand possible outcomes.

- **Random target:** For certain special card or minion effects, the system will choose a target randomly. For example, the minion "Ragnaros" will randomly deal 8 damage to a minion at the end of P_a's turn. In this case, the branching factors is small because the number of valid targets on the board is below 17.

- **Summon a random minion:** the branching factors of this kind of random event can vary a lot. For example, the card "Bane of Doom" only summons a random demon minion, which introduces around 10 possible outcomes, while the card "Piloted Shredder", which summons a random 2-cost minion, has around 100 possible outcomes.

- **Get (not draw) a random card** is similar to summoning a random minion. In the BRM version, most "get a random card" events have a low branching factor like the card "Y'sera" that can produce 5 dream cards, and the "Clockwork Gnome" that produces 7 "Spare Part" cards.

3.5.2 Bucketing Criterion

Among different types of chance events in Hearthstone, we can afford to enumerate all possible chance outcomes in the search for "Random target" and "Get a random card" events. "Summon a random minion" events do not happen in our test decks. Lastly, we found that only card-drawing happens frequently and the number of its possible outcomes is enormous. To mitigate this combinatorial explosion we apply chance event bucketing as follows. In Hearthstone's competitive decks, cards with similar mana cost usually have similar strengths. We can therefore categorize cards by their mana cost to form M buckets. The actual bucket choice depends on the card deck we are using and can be optimized empirically. In the experiments that will be reported later we used the buckets shown in Table 3.2. For determining the number of pre-samples N we experimented with various settings depending on the number of cards to be drawn. Empirically, The most effective choice was $N = 2$ in case one card is drawn, and $N = 1$ if more cards are drawn. We get this value setting by running a round-robin tournament using the open-hand UCT with 10000 rollouts on the Mech Mage deck for various N settings (Table 3.1).

To demonstrate the flexibility of our approach, we choose 3 decks to represent 3 different styles of Hearthstone games. Face Hunter is a *rush* deck that is designed to rush the opponent's hero down in the early game stage. Hand Lock is a *control* deck and its strategy is to control the board and win

the late game. Mech Mage is a *mid-range* deck fusing *rush* and *control* styles. The detailed deck information is listed in Appendix A.

3.6 Experiments

3.6.1 Impact of Imperfect Information

Hearthstone is an imperfect information game in which only a small part of the game state, P_o's hand cards, is invisible from P_a's view. However, the board and two players' deck information are known. We first investigate how different levels of perfection of inference affect the AI's play strength. In the experiment, we set a parameter a, which is the accuracy (levels of perfection) of the inference the AI can achieve. For instance, if we set $a = 1$, the AI has access to the perfect information state, while if we set $a = 0$, the AI guesses the imperfect information part completely wrong. To implement this method, we first copy the perfect information state; then for each card in P_o's hand, we randomly generate a number r between 0 and 1, if $r > a$, we swap the card with a different card in P_o's deck. In the experiment, we use a UCT AI (10000 rollouts, $d = 5$) agent with $a = 1$ playing 200 games against the same AI with $a \in \{0.2, 0.33, 0.5, 0.66, 0.8\}$, to see how much advantage the AI with $a = 1$ has.

From the results shown in Table 3.3, we can observe that having accurate inference will help to improve the playing strength. However, the advantage of UCT with $a = 1$ over UCT with $a = 0.5$, $a = 0.66$, and $a = 0.8$ is not significant. The result indicates that for the match-up we test, the board advantage is more important than the correct inference of opponent's hand. On the other hand, it shows that a good inference system is helpful to build a strong AI player for Hearthstone. It is interesting that the agent with a good inference (60% correct) can do as well as the perfect information agent. A possible explanation is that even with a perfect inference of P_o's hand, it is still hard to predict the future better than a 60% correct inference agent due to the high chance event branching factors.

Table 3.1: Win Rates of UCT with Different CNB Setttings

N when drawing 1 card	N when drawing 2+ cards	Win % (stderr)
1	1	46.2 (2.2)
2	1	54.0 (2.2)
3	1	53.4 (2.2)
3	2	52.6 (2.2)
4	1	47.0 (2.2)
4	2	46.8 (2.2)

Table 3.2: Card bucketing by deck and mana cost in Hearthstone

Deck	Buckets
Mech Mage	[1] [2] [3] [4,5] [6..10]
Hand Warlock	[1,2,3] [4] [5] [6] [7..10]
Face Hunter	[1] [2] [3..10]

Table 3.3: Win Rates of UCT ($a = 1$)

Opponent	Win % (stderr)
UCT ($a = 0$)	74.5 (3.1)
UCT ($a = 0.2$)	69.5 (3.3)
UCT ($a = 0.33$)	60.0 (3.5)
UCT ($a = 0.5$)	55.0 (3.5)
UCT ($a = 0.66$)	50.5 (3.5)
UCT ($a = 0.8$)	53.5 (3.5)

3.6.2 Search Time Budget Policy

To determine which search time management policy works better, we compare two policies by integrating both into an open-handed UCT search Agent with chance node bucketing (CNB) and Silverfish's evaluation function. In this experiment, we first pick the best performing settings of one-search ($\alpha_o = 0.66, \psi = 75$, $\alpha_m = 0.33$) and multiple-search ($\alpha_m = 0.33$) policy of by trying different parameters ($\beta = 0.8$ and $\tau = 1$ for both policies). Then we compare these best settings directly by playing games against each other. The result is shown in Table 3.4.

We can observe that after parameter tweaking, the performances of two policies are quite similar. We also observe that the one-search policy tends to spend more time on very first moves, while multiple-search policy tries to distribute time to moves equally. Implementation wise, the multiple-search policy is easier to implement and well-tune comparing with one-search policy.

3.6.3 Parameter Selection for DUCT

In our version of DUCT for Hearthstone, there are a few parameters that can be tuned to achieve better performance:

- rollout depth d
- exploration constant c
- number of the worlds sampled

Table 3.4: Win Rates of Time Management Two Policies

Opponent	Win % (stderr)
UCT with One-Search Policy	51.7 (2.5)
UCT with Multiple-Search Policy	48.3 (2.4)

Since finding the best 3-parameter combination using a full 3D grid search is extremely time-consuming, we ran a few experiments in advance with various parameter value combinations to select a candidate set of values for each parameter. For each parameter to test here, we fix other variables to a value that showed good results in previous small-scale experiments. Then we select the best value from the candidate value set for this parameter. We run a round-robin tournament between each pair of values in the parameter's candidate value set and finally conclude the best configuration of all parameters. Note that the parameter selection experiments here are run with both players using the Mech Mage deck. We chose the Mech Mage deck because it is a *mid-range* one in which both *rush* and *control* strategies can happen. We ran mirror matches because it is fair for both players to reduce the variance of results. Additionally, we fixed the number of rollouts in these experiments to 10000 (it takes approximately 3 seconds for DUCT) to allow us to run many experiments.

• **Rollout depth d**: We first investigate the impact of d by selecting it from the candidate set:

$$\{1, 3, 5, 7, 9, GameEnd\},$$

where 1 means to stop rollout simulation right after the end of current turn, and $GameEnd$ means rollout until the end of the game. The other parameters are kept fixed as $c = 0.7$ and $numWorld = 10$, we also fix the number of iterations of MCTS to 10000. The result is shown in Table 3.5.

The result shows that from $d = 1$ to $d = 5$, the increasing rollout depth leads to an increasing win rate as we expected. However, the win rates of DUCT with $d > 5$ have similar performance with DUCT with $d = 5$. The possible reason is that pure random rollouts and chance events introduce

Table 3.5: Round-Robin results of DUCT with various d

Player	Win % (stderr)
DUCT ($d = 1$)	35.8 (2.1)
DUCT ($d = 3$)	45.8 (2.2)
DUCT ($d = 5$)	56.0 (2.2)
DUCT ($d = 7$)	53.4 (2.2)
DUCT ($d = 9$)	54.6 (2.2)
DUCT ($d = GameEnd$)	54.4 (2.2)

more noise in the final reward signal. The result also indicates that Silverfish's evaluation is a good turn-end evaluation for the mirror Mech Mage deck setup.

• **Exploration Constant c**: We then evaluate the impact on the AI's strength of the exploration constant c. The exploration constant c is used to balance the exploration and exploitation in Monte Carlo Tree Search and the optimal c value varies in different domains. In this experiment, our candidate set of c values is

$$\{0.1, 0.3, 0.5, 0.7, 0.9, 1.2, 1.5\}$$

The other parameters are kept fixed as $d = 5$ and $numWorld = 10$, we also fix the number of iterations of DUCT (One-Search) to 10000. The result is shown in Table 3.6.

From the table we can see that in this setting, from 0.3 to 0.9, the performance is relatively similar. 0.7 has a slight advantage over other parameter settings. We also observe a diminishing return when c is greater than 0.9. Due to our limiting the number of iterations to 10000, too large exploration rate may lead to an unstable action that is returned. If we offer more time or number of iterations in the experiment, the best c may be different.

• **Number of Worlds:** Lastly, we evaluate the impact of the number of worlds sampled for DUCT considering the following values

$$\{1, 3, 5, 10, 20, 40\}$$

For this experiment, we fix $c = 0.7$ and the other parameters are kept the same as the previous experiment except for $numWorld$. The experiment results are shown in Table 3.7.

We can observe that there is a performance gain from 1 to 10 worlds, but the values greater than 10 do not seem to provide a stronger performance. We find that if we sample too few worlds, our search may not be able to reach some possible plays from the opponent and thus lead to a weaker performance. Since Hearthstone information sets are defined by both the perfect (the board) and the imperfect information part (the hands) and the imperfect information part does not contain a large amount of information, a reasonable guess is that the board control is more crucial than the

Table 3.6: Round-Robin results of DUCT with various c

Opponent	Win % (stderr)
DUCT ($c = 0.3$)	50.2 (2.2)
DUCT ($c = 0.5$)	55.0 (2.2)
DUCT ($c = 0.7$)	55.4 (2.2)
DUCT ($c = 0.9$)	54.0 (2.2)
DUCT ($c = 1.2$)	42.6 (2.2)
DUCT ($c = 1.5$)	42.8 (2.2)

Table 3.7: Round-Robin results of DUCT with various $numWorld$

Opponent	Win % (stderr)
DUCT ($numWorld = 1$)	37.8 (2.1)
DUCT ($numWorld = 3$)	45.2 (2.2)
DUCT ($numWorld = 5$)	51.8 (2.2)
DUCT ($numWorld = 10$)	56.0 (2.2)
DUCT ($numWorld = 20$)	55.2 (2.2)
DUCT ($numWorld = 40$)	54.0 (2.2)

hand inference in the decks we tested. Therefore, 10 worlds are sufficient for the search to perform well. On the other hand, sampling more worlds increases the implicit branching factors of DUCT search and causes the playing strength decrease. This result also agrees with Subsec. 3.6.1 which shows a larger number of samples may not perform better due to the random draws that increase the uncertainty.

In the end, we chose the following parameter configuration: $c = 0.7, d = 5, numWorlds = 10$, and the one-search policy that preformed slightly better than the multiple-search one. We used this parameter setting to test the play-strength of the DUCT algorithm against Silverfish that is the baseline in our experiments.

3.6.4 Playing Games

To evaluate the effect of adding DUCT and CNB to Silverfish we ran two experiments on an Intel i7-4710HQ CPU 3.5 GHz Windows 8.1 computer with 16 GB RAM. In the first experiment we let DUCT-Sf without CNB play 3 mirror matches, in which both players use the same deck (either Mech Mage, Handlock, or Face Hunter), against the original Silverfish player, allowing 5 seconds thinking time per move and using DUCT parameters $d = 5$, $numWorlds = 10$, UCT's optimized exploration constant $c = 0.7$ and time management one-search policy. The results are shown in Table 3.8 indicate that the performance of DUCT-Sf is superior to Silverfish's in all 3 matches. In the second experiment we let DUCT-Sf with CNB play against Silverfish. The results listed in Table 3.8 show an even greater playing strength gain.

Table 3.8: Win % (stderr) vs. Silverfish

Mirror Match	DUCT-Sf	DUCT-Sf+CNB
Mech Mage	66.5 (3.3)	76.0 (3.0)
Hand Warlock	54.0 (3.5)	71.5 (3.1)
Face Hunter	60.0 (3.5)	69.5 (3.2)
Combined	60.1 (2.0)	72.3 (1.8)

3.7 Summary

In this chapter we first presented our variant of determinized MCTS for the game of Hearthstone. We used tournament experiments to investigate the influence of each parameter and selected the best parameter settings empirically. We then demonstrated that the chance node bucketing approach can improve the strength of our search algorithm by reducing the branching factors caused by chance events. The core idea for dealing with non-determinism is sampling worlds and chance outcomes and chance event outcome bucketing.

Chapter 4

Learning High-Level Rollout Policies in Hearthstone

In this chapter we first describe the neural networks that we trained for making Hearthstone card play decisions in the MCTS rollout phase, and then present experimental results.

4.1 Learning High-Level Rollout Policies

In CC games actions can be categorized by levels of dependencies. For instance, "card-play" actions in Hearthstone can be considered high-level, while a "target-selection" action for that card can be regarded a dependent low-level action (Fig. 4.1).

In a turn that can consist of multiple actions, the most significant part is choosing high-level actions because they reflect the high-level strategy. For instance, if the active player P_a decides to attack, he will play more attacking high-level actions, and once the high-level actions are fixed, we only need to search the low-level actions that follow the high-level decisions. Fast heuristics or action scripts may be able to effectively handle this part. For instance, in Fig. 4.1, P_a's main goal is to remove all opponent's minions. So he chooses to play the "Fireball" and "Frostbolt" card to kill opponent's minions. "Target- selection" actions are trivial for P_a after deciding to play these two cards. If this is indeed the case, we can construct fast and informed stochastic MCTS rollout policies by training a high-level policy $\pi(a, s)$ that assigns probabilities to high-level actions a in states s, and — during the rollout phase — sample from π and invoke low-level action scripts to

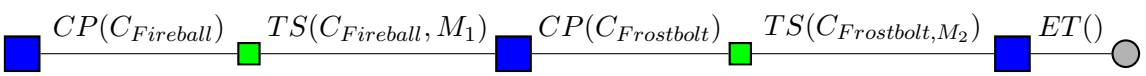

Figure 4.1: The visualization of a typical move sequence. High-level moves originate from blue nodes while low-level moves originate from green nodes. We can observe that the some high-level actions are followed by dependent low-level actions.

generate dependent actions. This idea is exciting, because the quality of rollout policies is crucial to the performance of MCTS, but up until now, only simple policies have been trained due to speed reasons. In games with complex action sets hierarchical turn decompositions allow us to explore speed vs. quality tradeoffs when constructing rollout policies, as we will see later in below sections.

4.2 Card-Play Policy Networks

A card-play policy network for Hearthstone maps a game state n to a card probability vector. The probabilities indicate how probable it is for card c_i to be in the turn card set

$$TCS(n) := \{c \mid c \text{ is played in turn starting with } n\}$$

Our goal is to train policy networks to mimic turn card sets computed by good Hearthstone players, which then can be used as high-level rollout policies in DUCT.

4.3 Training Data

To generate data for training our networks we let two DUCT-Sf+CNB players play three different mirror matches (using the Mech Mage, Handlock, Face Hunter decks), each consisting of 27,000 open-hand games using 10,000 rollouts per move. There are two benefits of using the open-handed data: first, the model learned from open-handed data can be directly used in determinized algorithms; second, it could be easier to learn counter-plays given the perfect state information in Hearthstone. Because drawing new cards in each turn randomizes states in Hearthstone we didn't feel the need for implementing explicit state/action exploration, but we may revisit this issue in future work.

The training target is the turn card set $TCS(n)$ for state n. For each triple $(n, TCS(n), n_{end})$ in the stored data set, where n is an intermediate game state and n_{end} is the turn end state reached after n, we have one training sample $(n, TCS(n))$. In fact, we use all intermediate state-TCS pairs as training samples, too. In total, we used about 4M samples.

4.4 State Features

Because Hearthstone's state description is rather complex we chose to construct an intermediate feature layer that encapsulates the most important state aspects. Our state feature set consists of three feature groups: global, hand, and board features. Also, recent achievements of convolutional neural networks (CNN) applied to games like Go [1], Poker [24], Atari games [3] and Starcraft [25] demonstrate its power of capturing the patterns from structured inputs. This motivated us to use

Algorithm 4 Card Compare Function

```
procedure COMPARE(C_1, C_2)
    if C_1.type = C2.type then
        return C_1.manacost - C_2.manacost
    else if C_1 is a minion card then
        return -1
    else
        return 1
    end if
end procedure
```

CNNs to learn the patterns of the structured board and hand features in Hearthstone. The features we used and the way that we encoded them for CNN models are follows:

•. **Global features:** two vectors encoding mana available until turn end, the opponent's available mana on the next turn, the Hero's health points (HP) (0-4 for each player, for a total of 25 different values), whether the active player is the starting player of the game, and whether the total ATK value of his minions is greater than the total HPs of the opponent's minions.

•. **Hand features:** a 2D vector V_h one-hot encodes the features of the cards in P_a and P_o's hands. Each column in V_h represents the features of a certain card that can appear in the game. These cards are sorted according to the compare function in Alg. 4. This way of sorting helps us to group cards with similar strengths together so that we get a better locality pattern for CNNs to learn from. Each row in V_h represents one binary (1: True, 0: False) hand feature related to the cards appears in the game. For instance, the j-th element in the i-th row encodes the i-th feature related to the card C_j. They are described here in order where the number represents the row index:

- 0-8: The number of instances (at most 2) of card C_j in P_a and P_o's hands:
  0: 0 instances in P_a's hand and 2 instances in P_o's hand.
  1: 0 instances in P_a's hand and 1 instances in P_o's hand.
  2: 1 instances in P_a's hand and 2 instances in P_o's hand.
  3: 0 instances in P_a's hand and 0 instances in P_o's hand.
  4: 1 instances in P_a's hand and 1 instances in P_o's hand.
  5: 2 instances in P_a's hand and 2 instances in P_o's hand.
  6: 2 instances in P_a's hand and 1 instances in P_o's hand.
  7: 1 instances in P_a's hand and 0 instances in P_o's hand.
  8: 2 instances in P_a's hand and 0 instances in P_o's hand.

- 9-12: The playability (1: playable, 0: not playable) of card C_j for P_a and for P_o:
  9: The card is not playable for P_a but playable for P_o.

10: The card is playable for both P_a and P_o.

11: The card is playable for P_a but not playable for P_o.

12: The card is not playable for neither P_a and P_o.

- Whether (1 or 0) P_a has a follow-up card-play after the card C_j is played:

  13: No follow-up card-play.

  14: There is a low-mana card-play.

  15: There is a high-mana card-play.

•. **Board features.** The features describing the board are represented as a 3D vector V_b. Each plane in V_b represents one binary board feature related to a minion appearing in the game. On all planes of V_b, each minion on the board is given a 2D index (i, j), where i is the minion's card index defined the same way as hand feature encoding, and j is mapped from its current HP value. The mapping from a minion's HP value to the index j is $[0 - 1 \rightarrow 0, 2 - 3 \rightarrow 1, 3 - 4 \rightarrow 2, 5 - 6 \rightarrow 3, 7+ \rightarrow 4]$. For instance, the (i, j)-th element in the k-th plane one-hot encodes the k-th feature related to a minion with the 2D index (i, j) on the board. There are 18 features described below (numbers represent the plane index):

- 0-8: The number of instances of the minion $M_{(i,j)}$ on P_a and P_o's sides of the board:

  0: P_a has 0 instances and P_o has 2 instances on the board.

  1: P_a has 0 instances and P_o has 1 instances on the board.

  2: P_a has 1 instances and P_o has 2 instances on the board.

  3: P_a has 0 instances and P_o has 0 instances on the board.

  4: P_a has 1 instances and P_o has 1 instances on the board.

  5: P_a has 2 instances and P_o has 2 instances on the board.

  6: P_a has 2 instances and P_o has 1 instances on the board.

  7: P_a has 1 instances and P_o has 0 instances on the board.

  8: P_a has 2 instances and P_o has 0 instances on the board.

- 9-17: The specialty level (Lv.2: legend minions, Lv.1: aura and battle-cry minions, Lv.0: other minions) of the minion $M_{(i,j)}$ on P_a and P_o's sides of the board:

  9: P_a's is level 0 and P_o's is level 2 on the board.

  10: P_a's is level 0 and P_o's is level 1 on the board.

  11: P_a's is level 1 and P_o's is level 2 on the board.

  12: P_a's is level 0 and P_o's is level 0 on the board.

  13: P_a's is level 1 and P_o's is level 1 on the board.

Table 4.1: Features from the view of the player to move

Feature(Modal)	Value Range	#CNN Planes
Max Mana (Global)	1-10	—
Heroes' HP (Global)	4 states	—
If active player is P_1 (Global)	0-1	—
Total attack $\geq$ enemy's board HP (Global)	0-1	—
Having each card (Hand)	9 states	9
Each card playable (Hand)	4 states	4
Next card after a cardplay (Hand)	3 states	3
Having each minion (Board)	9 states	9
Each minion's specialty (Board)	9 states	9

14: P_a's is level 2 and P_o's is level 2 on the board.

15: P_a's is level 2 and P_o's is level 1 on the board.

16: P_a's is level 1 and P_o's is level 0 on the board.

17: P_a's is level 2 and P_o's is level 0 on the board.

Table 4.1 summarizes the features we use in our experiments. We also tried some hand-crafted features but they didn't show merit, and we skipped some unimportant features like a minion's buff and debuff (power-ups or power-downs) to keep the model simple.

4.5 Network Architecture and Training

For approximating high-level card play policies we employ two network topologies:

- **"CNN+Merge."** Since there are inputs from different parts of the game state, we use a multi-module network architecture that consists of 3 sub-networks to receive the inputs from 3 feature groups (global, hand, and board) independently (Fig. 4.2). The global features are fed into a fully connected (FC) layer of 128 hidden units. The encoded board features are fed into one 2D convolution layer with 96 3×5 filters followed by one 2x2 max pooling layer and 3 to 5 2D convolution layer with 96 3×3 filters. The hand features are fed into 4 to 6 1D convolution layers with 96 1×3 filters. Finally, the outputs of sub-networks are flattened and merged to a merge layer by a simple concatenation, followed by 2 FC layers with 50% drop-out. The outputs layer has K (K is the number of different cards) sigmoid output neurons to compute the probability of each card to be played this turn. We use the Leaky ReLU [26] activation function ($\alpha = 0.2$) for all layers.

- **"DNN+Merge."** The network type also receives the inputs from the 3 feature groups, but the entire input is flattened into one long vector for each group (Fig. 4.3). Each group vector is

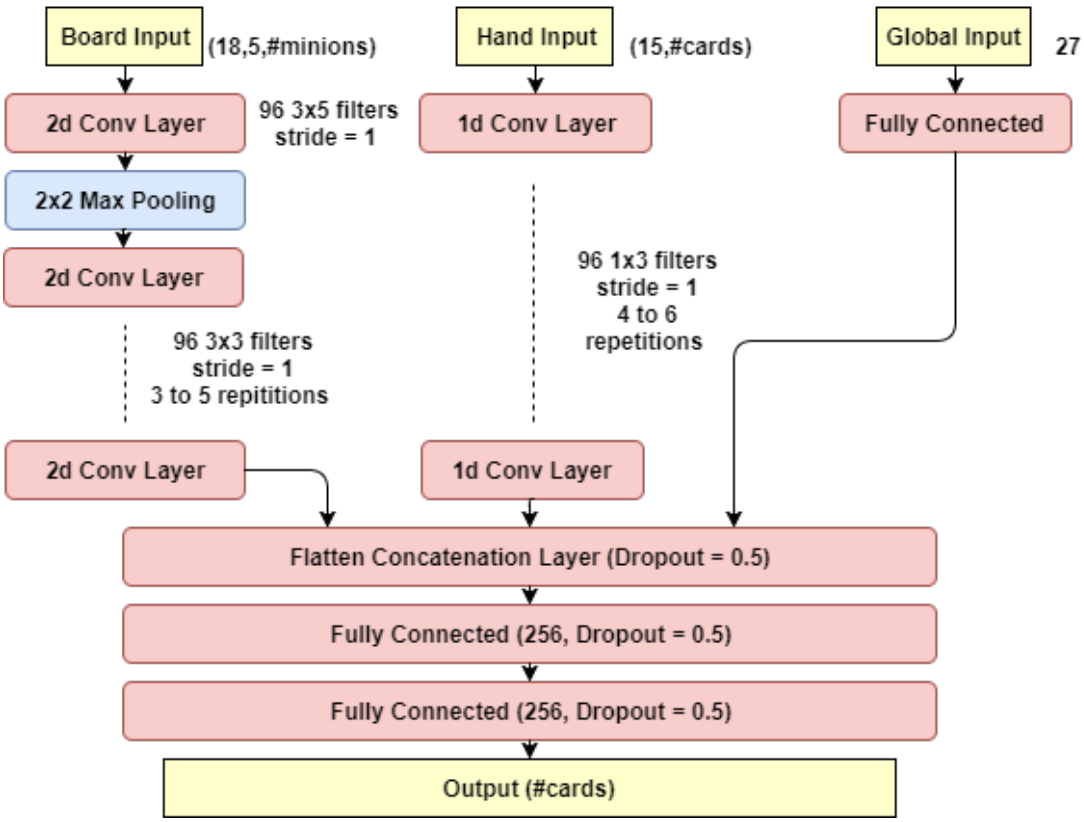

Figure 4.2: CNN+Merge Architecture: we tried different topologies of CNN models, the deepest one has 6 convolution layers in both board and hand module, while the shallowest on has 3 convolution layers. The board and hand input size can vary depending on the match-up.

then followed by one FC layer of Leaky ReLU units ($\alpha = 0.2$). Similar to the CNN+Merge type, the output of each group is fed into one concatenation (merge) layer and then followed by fully connected layers with using 0.5 drop-outs. The output layer has the same structure as the CNN+Merge networks.

When training both network types we used Xavier uniform parameter initialization [27]. We train several different models using similar settings. The largest one is a CNN+Merge network with 6 convolution layers having 1.75M parameters; the smallest one is the DNN+Merge network that has only 140K parameters. To tailor networks to different deck choices and maximum mana values we train them on data gathered from 3 mirror matches which we divided into 10 different sets with different initial maximum available mana values. For training we use the adaptive moment estimation (ADAM) with $\alpha = 10^{-3}$, decay $\sqrt{t/3}$, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. The mini-batch size was 200, and for one model, it typically took between 500 and 1,000 episodes for the training process to converge.

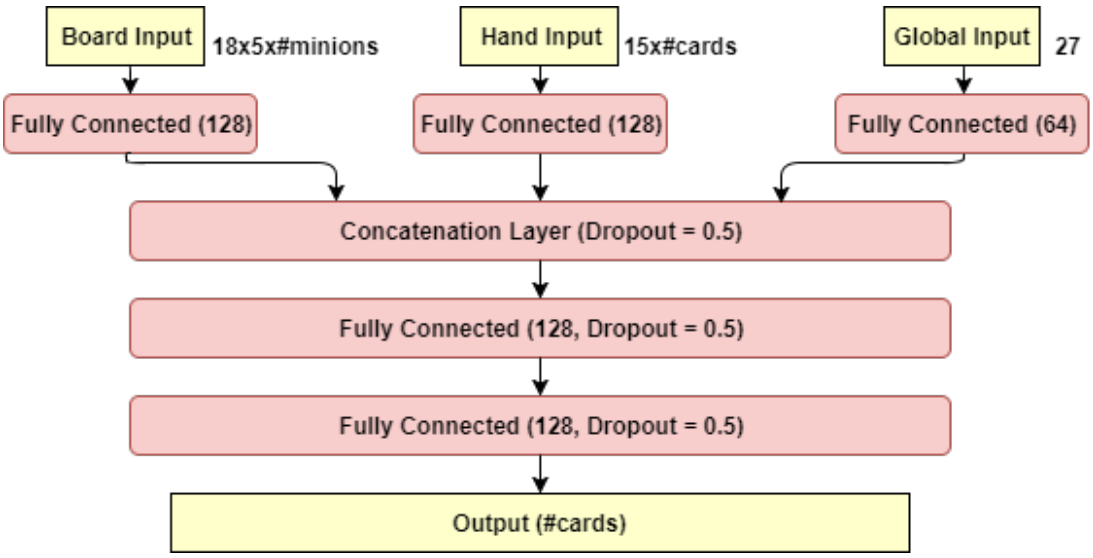

Figure 4.3: DNN+Merge Architecture: different from the CNN model, the inputs of DNN+Merge model are flattened 1D vectors and it has much fewer parameters to run the evaluations faster.

4.6 Experiment Setup

We trained and tested our neural networks with an NVIDIA GeForce GTX 860M graphics card with 4GB RAM using CUDA 7.5 and cnDNN4. The Hearthstone game simulator is written in C# and the networks are executed using Keras 1.1.2 [28] with the Theano 0.8.2 [29] back-end. For transmitting data between C# and Python 2.7.12 we used PythonNet [30] which introduced acceptable delays. One network evaluation including feature encoding only takes about 140 microseconds.

4.7 High-Level Move Prediction Accuracy

A high-level move prediction in Hearthstone is the cards to be played by a player in one turn. We compare the card selection of our learned high-level policy networks with the following move selectors:

- **Silverfish:** The original Silverfish AI with 3-ply search depth. We also enforce a 1 second search time limit because sometimes it takes too long for Silverfish to enumerate all possible 3-ply paths.
- **Greedy:** This action selector uses the cost-effect action evaluation heuristic $H(a)$, which we

Table 4.2: High-level policy prediction

Mana:	1-2	3-4	5-7	8-10
CNN+Merge (1.75m params)	91.9%	74.9%	76.6%	79.3%
CNN+Merge (290k params)	91.5%	71.7%	75.4%	77.8%
DNN+Merge (230K params)	89.9%	66.6%	69.2%	73.2%
Silverfish	86.7%	66.2%	67.0%	73.8%
Greedy	82.7%	50.3%	50.4%	55.5%

adapted from Silverfish's heuristics. It is defined as follows:

$$H(a) = \frac{value(a)}{cost(a)}, \text{ where} \tag{4.1}$$

$$value(a) = \sum_{m \in M_p} G(m,a) + \sum_{m \in M_o} L(m,a) \tag{4.2}$$

$$cost(a) = (\sum_{m \in M_p} L(m,a)) + a.ManaCost + 1 \tag{4.3}$$

$$L(m,a) = HpLoss(m,a) \cdot (m.ManaCost + 1)/m.MaxHp \tag{4.4}$$

$$G(m,a) = HpGain(m,a) \cdot (m.ManaCost + 1)/m.MaxHp \tag{4.5}$$

Here, a is the action to be evaluated, M_p, M_o represent the player to move's and the opponent's minion set, respectively, and $HpLoss(m,a)$ and $HpGain(m,a)$ denoting the loss and gain of minion m's health points when executing action a. $H(a)$ is a local heuristic that uses mana cost as scale for unifying the evaluation of gains and losses considering card-minion interactions. $H(a)$ is not very accurate for comparing actions from different levels, but it is better for comparing actions with the same precondition, such as finding the best target for a given card. The greedy action selector chooses the actions a with the highest $H(a)$ value in the current turn with the random tie-breaker.

For estimating the card selection quality we generated a total of around 1000 games the same way as the training data. We then picked ten states from each game with 1 to 10 available mana crystals, respectively. The accuracy metric we used is strict TCS equality, i.e., a card set prediction is accurate if $TCS_{pred}(n) = TCS(n)$. The results are presented in Table 4.2. They show that except for the beginning of the game, the trained networks are consistently better than Silverfish and Greedy at predicting turn card sets generated by high-level open-hand play, and that large networks are slightly better than the smaller networks. It is also interesting that near the end of the game the accuracy of all card selectors rises again. In the Face Hunter and Mech Mage games this may be caused by players running out of cards towards the end of the game which makes it easier to predict cards. The results also suggest that our CNN outperforms the DNN when using a similar number of parameters. However, the smaller DNN network only takes 60 microseconds to do one mini-batch

evaluation, whereas the CNN takes 10+ times longer.

4.8 Playing Games

We combined our deeper card-play policy networks with the low-level Greedy action chooser with the cost-effect heuristic (Eq. 4.1). Firstly, the networks receive the open-handed states and predict the probability of each card C_i to be played this turn: $P_{turn}(C_i|n)$. Then the agent greedily plays the card according to $\arg\max_C P_{turn}(C_i|n)$. If there are dependent low-level actions, he then greedily plays the low-level dependent action a with the best $H(a)$ value; otherwise he will play the next card. The agent keeps following this routine until there is no action available. We let this combined agent play 3 mirrors matches against different opponents including the random player, Greedy player with H(a) heuristic and Silverfish with 1 and 3-turn look-ahead search with 3 seconds of thinking time. The combined agent's win rates are shown in Table 4.3. The weakness of the Greedy action chooser is that it ignores mana management and it does not know which card to play to counter opponent's minions and hand cards. Card-play policy networks are complementary to such high-level decisions. We can observe that the combined agent beats the Greedy player and is on par with 1-turn Silverfish. However, the combined agent still cannot beat the search-based 3-turn look ahead Silverfish as expected.

4.8.1 Incorporating Card-Play Networks into DUCT

To make use of high- and low-level rollout policies in DUCT we replaced the original ROLLOUT function with Algorithm 5. This algorithm is tailored for games with multi-action turns and uses policies π_l and π_h to choose high- and low-level actions, respectively. It will execute multiple turns until either the turn limit or a terminal state is reached. If both high-level and low-level actions are available, it randomly selects either type and invokes the respective policy to generate an action. Otherwise, if an action is still available it uses the respective policy to generate one. Finally, the end-turn action is generated if no other actions are available. In the case of Hearthstone high-level policy $\pi_h(n)$ selects a card and low-level policy $\pi_l(n)$ then selects a suitable target.

Table 4.3: Win rate of CNN + greedy

Opponent	% Win Rate	% Std. Deviation
Random	99.7	0.3
Greedy	71.3	2.6
Silverfish 1-turn	54.7	2.9
Silverfish 3-turns	18.7	2.2

Algorithm 5 Rollout with Multi-Level Policy

```
1: // n: current state, d: turn limit
2: // π_h: high-level policy, π_l: low-level policy
3: procedure ROLLOUT(n, d)
4:     t ← 0
5:     while n not terminal and t < d do
6:         if n is chance node then
7:             a ← SampleSuccessor(n)
8:         else if high- and low-level actions available then
9:             if Random(0,1) > χ then
10:                a ← π_h(n)
11:            else
12:                a ← π_l(n)
13:            end if
14:        else if high-level actions available then
15:            a ← π_h(n)
16:        else if low-level actions available then
17:            a ← π_l(n)
18:        else
19:            a ← et                                   ▷ end turn
20:            t ← t + 1
21:        end if
22:        n ← Apply(n, a)
23:    end while
24:    return Eval(n)
25: end procedure
```

In our implementation we use the *softmax(a, τ)* function with temperature τ, defined as

$$P(a_i, \tau) = \frac{\exp(a_i/\tau)}{\sum_{j=1}^{K} \exp(a_j/\tau)}, \text{ for } i = 1, 2, \ldots, K \tag{4.6}$$

where a_i is the value of the i-th action, and τ is the temperature to control the randomness. If τ is large, all actions have nearly the same probability. We apply *softmax* to the less accurate but fast DNN outputs to define π_h based on card evaluations, and to the fast action evaluator H to form π_l based on heuristic target action evaluations. We set $\tau = 0.4$ for π_h and $\tau = 0.3$ for π_l to achieve the best performance based on small-scale experiments.

To reduce data transmission overhead when communicating between C# and Theano's Python code, we allocate a Numpy array and just send the indices of the entries to be filled. We also take advantage of the fact that the high-level policy network only has to be evaluated once when the turn begins. The multi-level policy rollout function we implemented is 5+ times slower than regular rollouts, but 10+ times faster than the bigger CNNs. To test the effect of the high-level rollout policy, we incorporated it into the strongest search-based AI without neural networks, namely DUCT with

Table 4.4: DUCT-Sf+CNB+HLR win rate against DUCT-Sf-CNB

Mirror Match	% Win Rate	% Std. Deviation
Mech Mage	53.4	2.2
Hand Warlock	62.6	2.1
Face Hunter	55.6	2.2
Combined	57.2	1.3

Table 4.5: DUCT-Sf+CNB+HLR win rate against Silverfish

Mirror Match	% Win Rate	% Std. Deviation
Mech Mage	74.5	3.1
Hand Warlock	79.0	2.8
Face Hunter	72.5	3.2
Combined	75.3	1.8

Silverfish's evaluation function and chance node bucketing (DUCT-Sf+CNB), and ran 500 games against DUCT-Sf+CNB for each mirror match, allowing 10 seconds thinking time per move and using $d = 5$, $numWorlds = 10$, $c = 0.7$, $\chi = 0.5$ and the one-search time management policy. The results are presented in Table 4.4. We can observe that overall our high-level rollout policy is superior to the pure random one, because the pure random rollout in Hearthstone has high variance and may lead to *blunders* [31]. As for deck-specific results, the high-level rollout policy improves in Hand Warlock game performance the most. One possible reason is that there are more self-harm cards like the "Hellfire", "Life Tap", and "Shadowflame" in the Hand Warlock deck, in which case our approach can help to avoid the *blunders* of using these cards improperly.

Lastly, we tested our DUCT-Sf+CNB+HLR agent with the same settings against our baseline, Silverfish with 5 seconds thinking time. The result is shown in Table 4.5. As we expected, this agent had a better result compared to DUCT-Sf-CNB. In the Hand Warlock games, it had a dominant performance and achieved a 79% win rate against Silverfish, which is much better than DUCT-Sf-CNB. The agent reaches 75.3% win rate overall, which is also superior to DUCT-Sf-CNB's 72.3%.

4.9 Summary

In this chapter, we first defined the card-play policy training target, then presented a set of Hearthstone game state features based on minimal domain knowledge. More importantly, The features are also suitable for different decks so that we do not need to change our code and network structures for other deck match-ups, and therefore serves our flexible AI design goals. Finally, we ran experiments to demonstrate the strength of our card-play networks. Combined with simple low-level heuristics, they can beat a simplified search-based Silverfish agent. Moreover we showed that

card-play networks improve the AI strength after being incorporated into DUCT.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis followed by a discussion of possible topics for future research.

5.1 Conclusions

In this thesis we have presented two improvements of MCTS applied to Hearthstone, and potentially other games with large chance node branching factors. We use bucketing and pre-sampling to deal with the issue of large branching factors caused by chance nodes. By using the optimized DUCT algorithm and Silverfish's evaluation function, our new search agent DUCT-Sf+CNB defeats the original Silverfish by 72% of the time.

We then defined a high-level policy for CC games and presented features for evaluating Hearthstone states that we feed into different neural networks which we trained from game data. We then applied the trained high-level networks in conjunction with low-level action heuristics to perform stochastic MCTS rollouts. Our experiments showed that the new AI system is even stronger than DUCT-Sf+CNB.

In addition, we re-designed and implemented the Hearthstone simulator based on Silverfish's code. This simulator is efficient and makes it easier for us to experiment compared to complex simulators like Metastone and Nora. Also, our bucketing and learning approaches only use the raw features without much domain knowledge. This approach can be easily adapted to other decks in future updates.

5.2 Future Work

- **Inference AI**

  In our approach, we handle the imperfect information by sampling multiple worlds. Although sampling a moderate number of worlds works well in our experiments, this method

still increases the branching factor of our search algorithm. In future research, we propose to machine-learn an inference module to predict the opponent's hand from the replay data. We can start with a simple Bayesian inference model similar to the one used in Skat [32] or a deep neural net to predict the probability of having each card given the move history.

- **Improving the chance event bucketing strategy**

  In our application, the outcomes of chance events are bucketed based on card mana cost. This bucketing criterion relies on the fact that in competitive decks each card's mana cost is proportional to its value. However, there are also other ways to bucket chance events, such as considering card effects. Machine learning techniques could be applied to the bucketing and sampling strategies to find a better way to classify the cards and provide a better bucketing criterion.

- **Improving the in-tree and low-level policy**

  In our research, we use the UCB1 formula to guide the in-tree policy and this method explores each untried leaf node once. Our experiments show that if we sample too many worlds, UCB1 for selection phase may suffer from large branching factors. However, with a learned policy of moves, we can employ algorithms like PUCB [33] or UCB with bias [18] to guide the in-tree traversal in a more efficient way. Besides improving the in-tree policy, we can also replace the scripted $H(a)$ heuristic with a machine learned heuristic function to improve the lower level rollout policy.

- **Reinforcement learning**

  Our high-level policy is currently learned by supervised learning. However, we can iteratively improve the policy by using reinforcement learning algorithms. The simplest way is batch reinforcement learning. The idea is to integrate the learned policy π_t to MCTS to generate the next batch of training data, and train a new classifier on the batch to get the new policy π_{t+1}. This is a variation of the DAGGER [34] algorithm.

- **Migrating to a better game engine**

  Newer Hearthstone simulators like Metastone are updated frequently by their communities to reflect changes in the original game. We are considering to use one of these simulators for future research because it can free us from tedious implementation issues.

Appendix A

Deck Lists

In this appendix, we list the specific information of 3 custom decks we used for experiments. They are Mech Mage (Fig. A.1), Hand Lock (Fig. A.2), and Face Hunter (Fig. A.3). Note that all cards are the version of Blackrock Mountain expansion.

Table A.1: Mech Mage Deck List

Card Name	Number of Copies
Clockwork Gnome	2
Coggaster	2
Mana Wyrm	1
Annoy-o-Tron	2
Frostbolt	2
Mechwarper	2
Snowchugger	2
Arcane Intellect	2
Harvest Golem	1
Spider Tank	2
Tinkertown Technician	2
Fireball	2
Goblin Blastmage	2
Mechanical Yeti	2
Loatheb	1
Archmage Antonidas	1
Dr.Boom	1
Flamestrike	1

Table A.2: Hand Lock Deck List

Card Name	Number of Copies
Mortal Coil	2
Ancient Watcher	2
Darkbomb	1
Sunfury Protector	2
Ironbeak Owl	1
Defender of Argus	1
Hellfire	2
Shadowflame	1
Twilight Drake	2
Voidcaller	2
Antique Healbot	1
Big Game Hunter	1
Loatheb	1
Studge Belcher	1
Emperor Thaurissan	1
Siphon Soul	1
Sylvanas Windrunner	1
Dr.Boom	1
Lord.Jaraxxus	1
Mal'Ganis	1
Mountain Giant	2
Molten Giant	2

Table A.3: Face Hunter Deck List

Card Name	Number of Copies
Abusive Sergeant	2
Leper Gnome	2
Southsea Deckhand	1
Worgen Infiltrator	1
Explosive Trap	2
Glavivezooka	2
Haunted Creeper	1
Knife Juggler	2
Mad Scientist	2
Quick Shot	2
Animal Companion	2
Arcane Golem	1
Eaglehorn Bow	2
Ironbeak Owl	1
Kill Command	2
Unleash the Hounds	2
Wolfrider	2
Leeroy Jenkins	1

Bibliography

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] M. Moravčík *et al.*, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, 2017. [Online]. Available: http://dx.doi.org/10.1126/science.aam6960

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[4] noHero123, "Silverfish," https://github.com/noHero123/silverfish, 2015.

[5] demilich1, "Metastone," https://github.com/demilich1/metastone, 2017.

[6] D. Taralla, Z. Qiu, A. Sutera, R. Fonteneau, and D. Ernst, "Decision making from confidence measurement on the reward growth using supervised learning: A study intended for large-scale video games," in *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016)-Volume 2*, 2016, pp. 264–271.

[7] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismata's robust AI architecture for games with large search spaces," in *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, 2015.

[8] N. A. Barriga, M. Stanescu, and M. Buro, "Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games," in *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[9] C. D. Ward and P. I. Cowling, "Monte Carlo search applied to card selection in Magic: The Gathering," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 9–16.

[10] N. Jouandeau and T. Cazenave, "Monte Carlo tree reductions for stochastic games," in *Technologies and Applications of Artificial Intelligence*. Springer, 2014, pp. 228–238.

[11] M. Lanctot, A. Saffidine, J. Veness, C. Archibald, and M. H. M. Winands, "Monte Carlo *-minimax search," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, pp. 580–586. [Online]. Available: http://dl.acm.org/citation.cfm?id=2540128.2540213

[12] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time Atari game play using offline Monte Carlo tree search planning," in *Advances in neural information processing systems*, 2014, pp. 3338–3346.

[13] S. Zhang and M. Buro, "Improving Hearthstone AI by learning high-level rollout policies and bucketing chance node events," in *IEEE Conference on Computational Intelligence in Games (CIG 2017)*.

[14] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information set Monte Carlo tree search." *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 2, pp. 120–143, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tciaig/tciaig4.html#CowlingPW12

[15] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. J. Van Den Herik, "Monte Carlo strategies for computer Go," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, 2006, pp. 83–91.

[16] L. Kocsis and C. Szepesvári, "Bandit based Monte Carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[17] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[18] G. M. J. Chaslot, M. H. Winands, H. J. V. D. Herik, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte Carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 03, pp. 343–357, 2008.

[19] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 273–280.

[20] S.-C. Huang, B. Arneson, R. B. Hayward, M. Müller, and J. Pawlewicz, "Mohex 2.0: a pattern-based MCTS hex player," in *International Conference on Computers and Games*. Springer, 2013, pp. 60–71.

[21] M. L. Ginsberg, "GIB: Imperfect information in a computationally challenging game," *Journal of Artificial Intelligence Research*, vol. 14, pp. 303–358, 2001.

[22] T. Furtak and M. Buro, "Recursive Monte Carlo search for imperfect information games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[23] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble determinization in Monte Carlo tree search for the imperfect information card game Magic: The gathering." *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 4, pp. 241–257, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tciaig/tciaig4.html#CowlingWP12

[24] N. Yakovenko, L. Cao, C. Raffel, and J. Fan, "Poker-CNN: a pattern learning strategy for making draws and bets in poker games," *arXiv preprint arXiv:1509.06731*, 2015.

[25] M. Stanescu, N. A. Barriga, A. Hess, and M. Buro, "Evaluating real-time strategy game states using convolutional neural networks," in *IEEE Conference on Computational Intelligence and Games (CIG 2016)*.

[26] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[27] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[28] F. Chollet, "Keras," https://github.com/fchollet/keras, 2017.

[29] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[30] D. Anthoff, "PythonNet," https://github.com/pythonnet/pythonnet, 2017.

[31] S. Fernando and M. Müller, "Analyzing simulations in Monte Carlo tree search for the game of go," in *International Conference on Computers and Games*. Springer, 2013, pp. 72–83.

[32] M. Buro, J. R. Long, T. Furtak, and N. Sturtevant, "Improving state evaluation, inference, and search in trick-based card games," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1407–1413. [Online]. Available: http://dl.acm.org/citation.cfm?id=1661445.1661671

[33] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[34] S. Ross, G. J. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 627–635.