APPROACHING HEARTHSTONE AS A COMBINATORIAL MULTI-ARMED BANDIT PROBLEM

A.J.J. Valkenberg

Master Thesis 19-10

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Science and Engineering
of Maastricht University

Thesis committee:

Dr. M.H.M. Winands
Dr. K. Driessens

Maastricht University
Department of Data Science and Knowledge Engineering
Maastricht, The Netherlands
May 13, 2019

Preface

This thesis was written under the supervision of Dr. M.H.M. Winands, to whom I owe a special thanks as without his guidance and patience this work could not have been finished. Additionally I would like to thank G.-J. Roelofs MSc., T. Aliyev MSc. and D. de Rydt MSc. for allowing me to base the framework for this research on their code and R. Arnoldussen MSc. for lending me the hardware to compute all of it. The code that was written for this thesis can be found on GitHub through the following link: `https://github.com/antonvalkenberg/ThesisCodeHSCMAB`. Last but not least, I would like to thank my family and friends for their trust and support during all of this.

Summary

This thesis investigates how to approach the popular digital card game Hearthstone as a Combinatorial Multi-Armed Bandit Problem. Hearthstone is a challenging domain for current search techniques because of their inherent complexity due to imperfect- and incomplete information. Another aspect of the complexity is the fact that players can execute multiple actions per turn, which results in a high branching factor of the search space.

Three search techniques based on the Monte Carlo Method are presented as approaches to handle the problem domain. These techniques are: Hierarchical Expansion, Naïve Monte Carlo Tree Search, and Linear Side Information. To adapt these techniques to fit the domain of Hearthstone, three search enhancements are presented as well as technique-specific enhancements. The three search enhancements are: determinisation of the game state, the use of multiple determinisations in an ensemble, and the improvement of playouts using the Move-Average Sampling Technique. Technique-specific enhancements include four types of dimensional orderings for Hierarchical Expansion and an adjustment to Linear Side Information to allow it to run on a time-limited budget.

Several experiments are conducted to determine appropriate settings for each of the three search techniques. The optimal length of the playouts for Hierarchical Expansion and Linear Side Information appears to be 4-ply, while for Naïve Monte Carlo Tree Search it seems 2-ply long playouts are optimal. For both Hierarchical Expansion and Linear Side Information, the Move-Average Sampling Technique does not significantly increase performance and therefore the actions in the playouts for these two techniques are selected randomly. Naïve Monte Carlo Tree Search does show a performance increase when using the Move-Average Sampling Technique with ϵ-greedy action selection. When using an ensemble of determinisations, Hierarchical Expansion does not benefit from more than one determinisation. Linear Side Information and Naïve Monte Carlo Tree Search however, do show an increase in performance when using multiple determinisations, with an ensemble of five determinisations providing the best results.

As for specific settings for the Hierarchical Expansion technique, results show a C-constant value of 0.2 for UCT to provide the best performance. For the T setting in Minimum-T Expansion, different values do not significantly affect performance, although a value of 0 does provide more iterations and a greater depth of the search tree. The experiment regarding the Dimensional Ordering problem for Hierarchical Expansion shows that the Descending Entropy ordering attains the best performance among the proposed orderings. For Linear Side Information, its technique-specific settings are focused on the computation budget. Results show that there is no difference in performance between the two proposed budget-estimation methods AverageSampleEstimation and PreviousSearchEstimation. However, the AverageSampleEstimation method achieves more iterations on average. It is also shown that when a high percentage of the computation budget is spent during the Explore phase, the performance of Linear Side Information decreases. For Naïve Monte Carlo Tree Search, its technique-specific settings are the ϵ value in the ϵ-greedy policies π_0 and π_g. Both policies show a significant increase in performance at high ϵ values, where a value of 0.75 provides the best results.

To determine the relative playing strength of the search techniques, a round-robin tournament is run between five agents; one for each of the search techniques and two agents representing a weak and a strong player. The results of this tournament show that each of the techniques significantly outperform the weak agent. However, none of the techniques can perform well against the strong agent. Two more tournaments are run; one where the evaluation function of each of the techniques is replaced, and one where the computation budget is varied between one and ten seconds. The second tournament shows that with this new evaluation function both the Hierarchical Expansion and Naïve Monte Carlo Tree Search agents perform at a level comparable to that of the strong agent, while the Linear Side Information agent does not improve. The third tournament shows that as the computation budget is increased, both the

Hierarchical Expansion and Naïve Monte Carlo Tree Search agents improve their performance, although a budget increase from ten seconds to sixty seconds does not significantly affect the performance of either agent. Overall, this thesis shows that when search techniques that are designed for the Combinatorial Multi-Armed Bandit problem are applied to Hearthstone, they can perform as well as an agent that finished second in the first Hearthstone AI Competition at IEEE CIG 2018.

Contents

Preface **iii**

Summary **v**

Contents **vii**

1 Introduction **1**
1.1 Game AI . . . 1
1.1.1 Card Games . . . 2
1.1.2 Search Techniques . . . 2
1.2 Problem Domain . . . 2
1.2.1 The Combinatorial Multi-Armed Bandit Problem . . . 2
1.3 Research Overview . . . 3
1.3.1 Problem Statement and Research Questions . . . 3
1.4 Thesis Overview . . . 3

2 Hearthstone **5**
2.1 The Game . . . 5
2.1.1 Pre-Game Setup . . . 7
2.1.2 Start of Turn . . . 7
2.1.3 Player Actions . . . 8
2.1.4 End of Turn . . . 8
2.2 Rules . . . 8
2.3 Game Characteristics . . . 8
2.3.1 Level of Determinism . . . 8
2.3.2 Level of Information . . . 9
2.4 Variants . . . 9
2.5 Strategies . . . 9
2.6 Hearthstone as a Combinatorial Multi-Armed Bandit Problem . . . 10
2.7 Current AI . . . 10
2.7.1 Hearthstone Simulators . . . 10
2.7.2 Bots for Hearthstone . . . 10
2.7.3 Agents for Hearthstone . . . 11

3 Monte Carlo Search Techniques **13**
3.1 The Monte Carlo Method . . . 13
3.1.1 Monte Carlo Search . . . 13
3.2 Monte Carlo Tree Search . . . 14
3.2.1 Hierarchical Expansion . . . 15
3.3 Naïve Monte Carlo . . . 15
3.4 Linear Side Information . . . 17

4 Searching in Hearthstone **19**
4.1 Search Enhancements . . . 19
4.1.1 Determinisation of the Search Space . . . 19
4.1.2 Ensemble Search . . . 20
4.1.3 Move-Average Sampling Technique . . . 21
4.2 Hierarchical Monte Carlo Tree Search for Hearthstone . . . 22
4.2.1 Move Completion . . . 22
4.2.2 Dimensional Ordering . . . 23
4.3 Naïve Monte Carlo Tree Search for Hearthstone . . . 24
4.4 Linear Side Information for Hearthstone . . . 24
4.5 Hearthstone State Evaluation Function . . . 25

5 Experiments & Results **27**
5.1 Experimental Setup . . . 27
5.1.1 Decks . . . 27
5.1.2 Default settings . . . 29
5.1.3 Playout Length . . . 29
5.1.4 Type of Playout . . . 30
5.1.5 Ensemble Size . . . 30
5.1.6 H-MCTS *C* Constant . . . 30
5.1.7 H-MCTS Expansion Threshold . . . 31
5.1.8 LSI Time Budget Estimation . . . 31
5.1.9 LSI Exploration Budget . . . 31
5.1.10 N-MCTS Policies . . . 32
5.2 H-MCTS Dimensional Ordering . . . 32
5.3 Round-Robin Tournament . . . 33
5.3.1 HeuristicAgent Evaluation Function . . . 33
5.3.2 Computation Budget . . . 34

6 Conclusion **37**
6.1 Research Questions . . . 37
6.2 Problem Statement . . . 38
6.3 Future Research . . . 38

References **39**

Chapter 1

Introduction

This chapter provides an introductory overview of this thesis, starting with Section 1.1 where its context and motivation are discussed. The application of Artificial Intelligence (AI) techniques to games is reviewed because this thesis focuses on the digital card game Hearthstone®. Search techniques are discussed due to their ability to cope with the large state spaces that are inherent to card games. Section 1.2 presents the problem domain before it is reviewed in detail in Chapter 2. Section 1.3 provides an overview of the research conducted in this thesis by presenting the problem statement and the four research questions that follow from it. Finally, Section 1.4 presents the general structure of this thesis.

1.1 Game AI

The field of AI has seen success in its application to games. Chess (Fang, Hsu, and Hsu, 2003) and Go (Silver *et al.*, 2016) are examples of games that, with their popularity and exposure, have increased the general public's awareness of the advancements in the field of Game AI. Games provide suitable domains to review the performance of new techniques. One of the reasons for this is the availability of strong human opponents. In many games it is not trivial for AI techniques to surpass a human player's level of play (Schaeffer, 2001).

A core problem in games is decision making: choosing the best action out of a collection of possible actions. When human players make a decision in a game they are predominantly concerned with finding a way to win. This can be described as a strategy: a plan of how to progress the game towards an end state where they win. The state of a game at any point during play is referred to as a *game state* and the collection of all possible game states is a game's *state space*.

How much a player knows about the game state is a characteristic of a game, it can either be perfect or imperfect. Chess is an example of a game with perfect information: all pieces are visible to each player which means all the information about the game state is known. For a game with perfect information a player could construct the entire state space and find a strategy that leads to a win from any game state (van den Herik, Uiterwijk, and van Rijswijck, 2002). However, some games' state spaces are too large to fully construct with current day computer hardware.

When some aspects of a game state are unknown, such as the cards in an opponent's hand in Poker or the identity of game pieces in Stratego®, the information is imperfect. Strategies that deal with these games have to take into account all the various possibilities that these unknowns can be. Poker is an example of a card game with imperfect information where Neural-Networks and Deep-Learning techniques have been successfully applied as strategies to combat the unknown information (Moravčík *et al.*, 2017).

1.1.1 Card Games

Many card games feature the players holding cards in their hand and having the identity of these cards hidden from the opponent. These types of games are attractive research subjects because of their inherent complexity due to imperfect information. Some card games that have become popular research topics over recent years have been games such as Poker and Bridge, but also digital card games such as Hearthstone. These digital games are convenient research domains because they have already been modelled on computers and therefore allow research to proceed quicker than when a certain problem domain would have to be modelled from scratch.

Another reason why card games are challenging research domains is because many of them have a large *branching factor*. A game's branching factor is the number of different actions available to a player each turn. Where some games give the player the option to play only one action per turn, some card games allow a sequence of actions, e.g. Hearthstone and Magic the Gathering®. Action sequences cause a combinatorial explosion in the size of the branching factor. State spaces with large branching factors provide a challenge for existing AI techniques due to their size, this is especially true for techniques that are based on searching.

1.1.2 Search Techniques

Search techniques explore the state space starting from the current game state and focus their computation budget on those parts that are the most relevant to the acting player. An evaluation function is used as a measure of how relevant a specific state is to that player. This allows search techniques to reduce the size of the space by pruning states that have been evaluated as less relevant (Knuth and Moore, 1975).

When state spaces become too large even pruning can fail to make the space sufficiently searchable. In spite of this, considerable progress has recently been made in the game of Go by using a combination of a search technique and a Machine-Learning technique; Monte Carlo Tree Search (MCTS) aided by Deep Neural-Networks (Silver *et al.*, 2016). MCTS is a search technique that explores the state space by constructing and traversing a tree structure. It uses random sampling to value the nodes in the tree and heuristics to determine the most promising nodes to analyse. Improvements to MCTS are still being researched, for example in domains that present a combinatorial *action space*, i.e. the collection of available actions. The Combinatorial Multi-Armed Bandit (CMAB) problem is an example of such a domain (Chen, Wang, and Yuan, 2013b).

1.2 Problem Domain

The game that this thesis focuses on is the digital strategy card game Hearthstone. Due to its many rules and complexities, we discuss the game in detail in Chapter 2. In this section we discuss the CMAB problem.

1.2.1 The Combinatorial Multi-Armed Bandit Problem

The Multi-Armed Bandit (MAB) problem can be formulated as a slot machine presenting the player with m arms, where the reward for choosing an arm is unknown prior to selecting it. The player is tasked with repeatedly choosing an arm to play each round and the objective is to get as close to the optimal reward as possible. The CMAB problem presents the player with a choice between super-arms as their play for a turn. These super-arms represent a set of arms of the MAB. A player's *combined action* becomes a set of *partial actions* that represent a choice for an individual arm. A combined action can consist of an arbitrary number of partial actions.

Let $R_t(C)$ be the reward function for combined action C when played in round t. The reward $R_t(C)$ could be a simple summation of reward for the partial actions p in C: $R_t(C) = \sum_{p \in C} R_t(p)$, but since the function is unknown it could also be nonlinear. A solution strategy for the CMAB problem tries to minimise the regret between the expected reward for a combined action and the actual reward received after executing the action. Recent strategies that deal with the CMAB problem include Naïve Monte Carlo (NMC) (Ontañón, 2013), Linear Side Information (LSI) (Shleyfman, Komenda, and Domshlak, 2014) and Hierarchical Expansion (HE) (Roelofs, 2015). These strategies are explained in Chapter 3.

1.3 Research Overview

Applying a standard MCTS algorithm to Hearthstone would not be feasible due to its combinatorial action space and the fact that executing certain game actions can lead to new actions becoming available. Current research regarding Hearthstone has focused on identifying strategic moves using a Neural-Network (Doux, Gautrais, and Negrevergne, 2016) or extensively applying heuristics to aid a standard MCTS algorithm (Santos, Santos, and Melo, 2017). Hearthstone has not yet been approached as a CMAB problem, even though its action space is combinatoric. Expert players indicate that the ordering of actions in Hearthstone is crucial to playing strength (TempoStorm, 2016). It would therefore be important to research various strategies to dimensional ordering within the algorithm of HE.

1.3.1 Problem Statement and Research Questions

From the problem and techniques discussed in this chapter the problem statement of this thesis follows as:

- *Can a card game such as Hearthstone be approached as a Combinatorial Multi-Armed Bandit Problem?*

To address the problem statement the following four research questions have been formulated:

1. How do we represent Hearthstone as a CMAB Problem?
2. How do we handle imperfect information in the CMAB Problem?
3. Can we find dimensional orderings for Hierarchical Expansion that improve performance?
4. How does Hierarchical Expansion perform compared to Naïve Monte Carlo and Linear Side Information in the domain of Hearthstone?

The first question is answered after reviewing the game in detail. To answer the second question, for each CMAB solution there will be a discussion as to how to adapt it to fit the game of Hearthstone. The last two questions are answered by performing experiments in an effort to analyse and compare the performance of the different techniques. Dimensional orderings that are specific to the domain of Hearthstone are also proposed and compared through experimentation.

1.4 Thesis Overview

This thesis starts with Chapter 1 that introduces the motivation for, and background of, the research conducted within. This chapter also briefly outlines the problem and current solutions to it. Chapter 2 presents the game of Hearthstone and reviews its rules in detail. Chapter 3 explains the techniques used in this thesis and Chapter 4 provides an overview of the adjustments and enhancements required to fit these techniques to the domain of Hearthstone. Chapter 5 presents the experiments and their results and provides analysis for them. Finally, Chapter 6 concludes this thesis and discusses any directions for future research.

Chapter 2

Hearthstone

This chapter provides an overview of the game of Hearthstone. Section 2.1 introduces the game and the board on which it is played as well as the turn structure. Section 2.2 discusses the rules of the game. Section 2.3 presents some characteristics of Hearthstone. Section 2.4 explains of the currently available modes of play. Section 2.5 describes which strategies are used in Hearthstone. Section 2.6 discusses how to model Hearthstone as a Combinatorial Multi-Armed Bandit problem. To close this chapter, Section 2.7 reviews the current state of game-playing agents available for Hearthstone.

2.1 The Game

Hearthstone is a digital strategy card game developed by Blizzard Entertainment®. It was published in the first quarter of 2014 and has since grown to have a reported player base of 70 million users as of May 2017 (Blizzard Entertainment, 2017).

Hearthstone presents a two-player turn-based zero-sum game where the objective is to reduce the opposing player's *health points* to zero. While a draw is possible (both player's health points would need to reach zero at the same time), it is a rare occurrence. The game is played on a board as can be seen in Figure 2.1. On this board the players are represented by a *hero* which tracks their remaining health points in its bottom right corner. The game client presents the board from the user's perspective, which means their opponent is represented by the top hero and the user's hero is the bottom one.

Nine classes are available for the player to choose from before starting the game. Each class can have multiple different heroes, but their difference is only cosmetic. The chosen class affects the *hero power* that is available to the player and it is shown to the immediate right of the hero portrait on the board. Some examples of a hero power are: 'Deal 1 damage', 'Restore 2 Health' or 'Deal 2 damage to the enemy hero'.

A class also restricts which subset of cards a player has access to when constructing their deck (the collection of cards they start the game with). Hearthstone decks contain 30 cards and allow at most two copies of the same card, except for cards of the *Legendary* rarity, of which only a single copy is allowed. A player's deck is shown on the far right of the board, next to the *End Turn* button. Cards in the deck are face-down since their order is hidden information.

Mana crystals are the game's resource and can be spent to play cards or use the hero power. This resource is commonly referred to as *mana*. A card's mana cost is represented by the number inside the mana crystal on the card, and hero powers have a fixed mana cost of 2. A player's available mana crystals are displayed in a bar to the right of the cards in their hand.

Hearthstone cards currently come in four different types (see Figure 2.2 for an example of each type):

1. Minion

2. Spell

3. Weapon

4. Hero

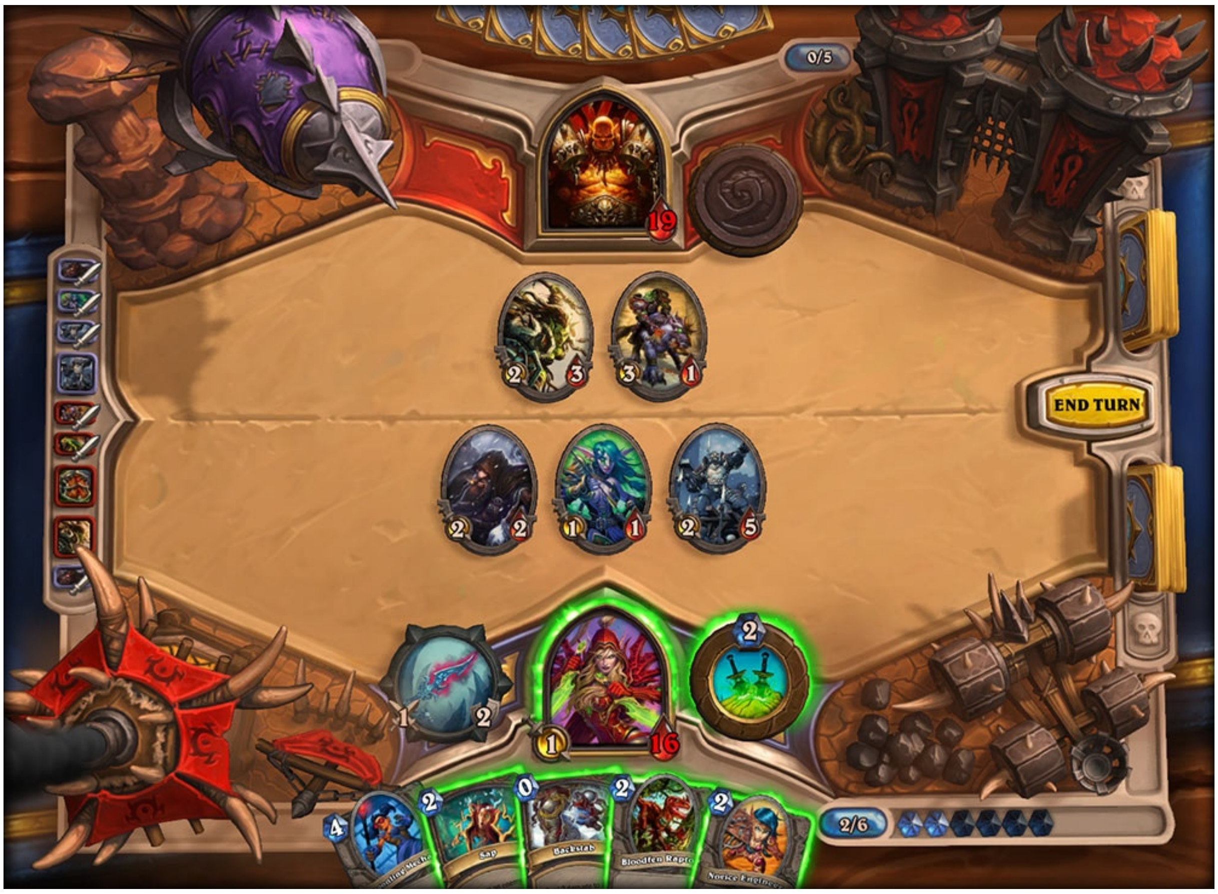


Figure 2.1: Example of a Hearthstone board state.

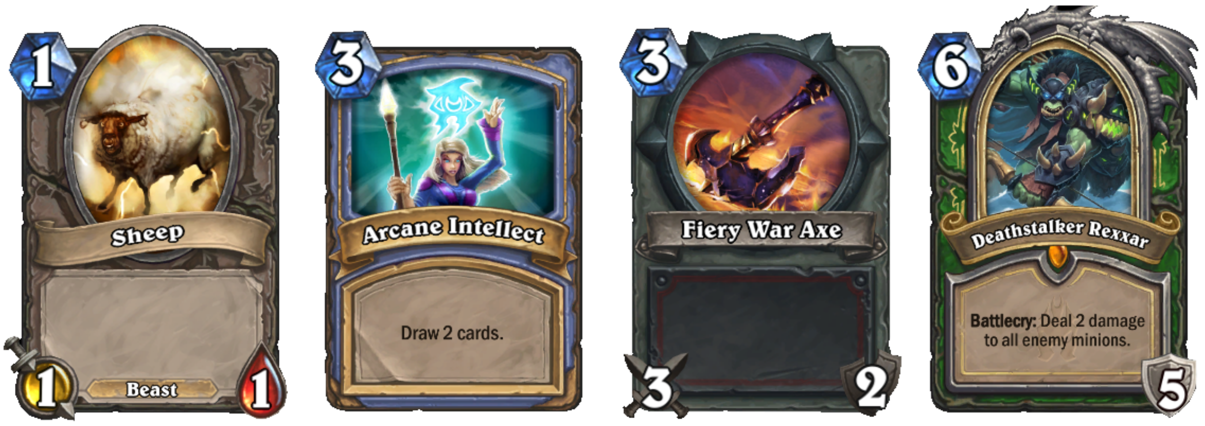


Figure 2.2: Examples of Hearthstone card types. From left to right: Minion, Spell, Weapon and Hero.

Minion cards create a *minion* on the board and a player can have a maximum of 7 minions on their side of the board at any one time. Minions have attack and health values, which are visible to the left and right of a minion, respectively. They can attack opposing minions or the opponent's hero and when they do they deal damage equal to their attack value. They also receive damage in return equal to the target's attack value. If their health ever falls to zero or less they die and are removed from play.

Spell cards are one-shot effects on the game, but the type of effect varies between spells. Some examples of these effects are: 'Draw 2 cards', 'Deal 6 damage', 'Destroy a random enemy minion' and 'Restore 8 Health'.

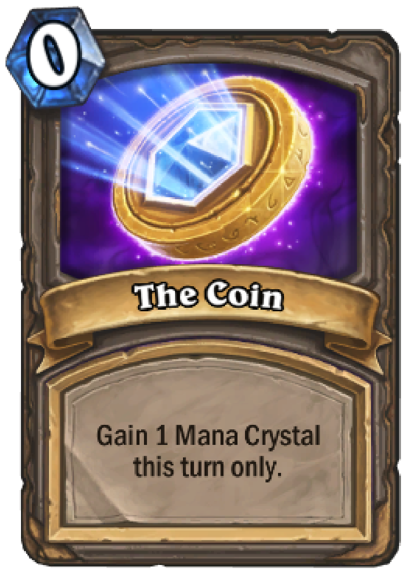


Figure 2.3: The Coin card.

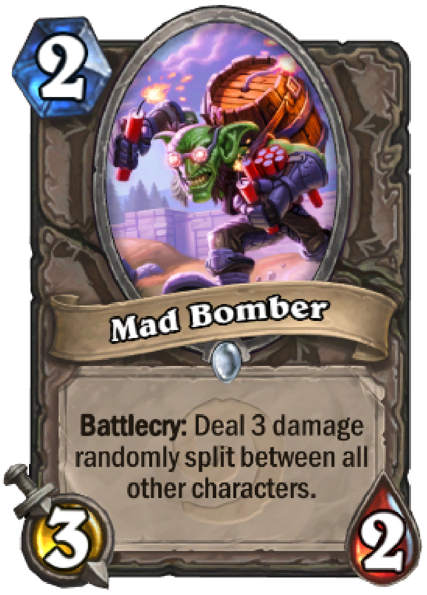

2
Mad Bomber
Battlecry: Deal 3 damage randomly split between all other characters.
3
2

Figure 2.4: Mad Bomber card.

Weapon cards equip the player's hero with a weapon that can be used to have the hero directly attack an opposing target. When a hero has a weapon equipped, it is shown to the left of its portrait. Weapons have an attack value and a durability value and when a hero attacks with a weapon it costs one point of durability. When a weapon's durability reaches zero, it is removed from play.

Hero cards replace the player's hero power with a specific version unique to that card for the remainder of the game. The hero portrait is also changed but this is a cosmetic change only.

Minions, spells, weapons and heroes have many different abilities, too many to discuss in detail here. Hearthstone currently contains nearly 1500 collectable cards and most have their own special abilities that affect the game.

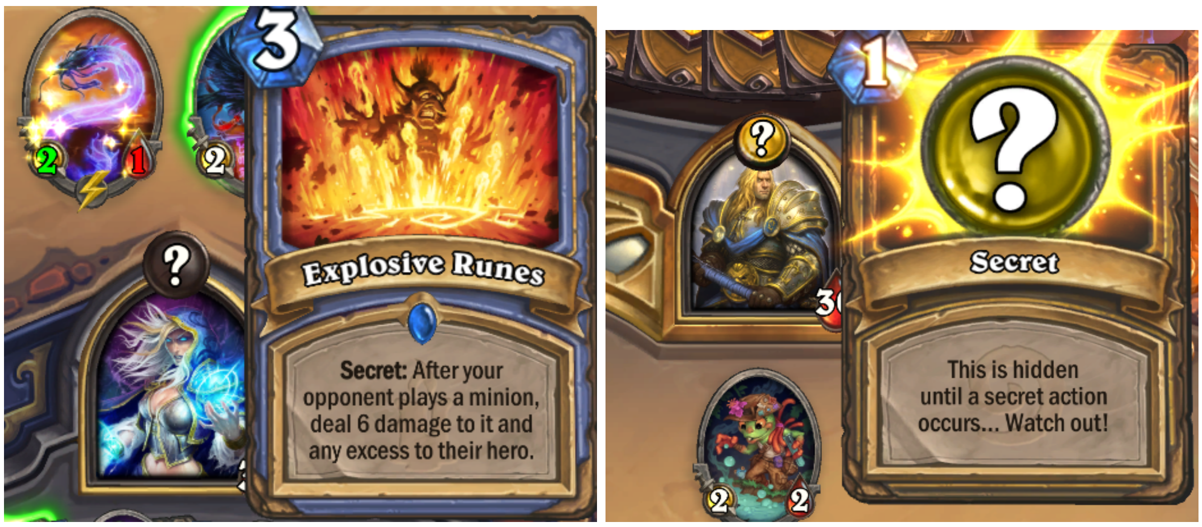
2.1.1 Pre-Game Setup

A player starts the game on 30 health points and with three randomly chosen cards from their deck in hand, or four cards if they are second to play (this is also known as being *on the draw*). While a player can choose which cards are in their deck, its ordering is randomised. Before the game starts both players can return any number of cards in their starting hand to the deck and randomly draw a replacement card, this is called the *Mulligan* phase.

Which player commences play first is determined by the game. To compensate for the inherent strategic advantage of playing first, the player who is on the draw receives a specific extra card called the *Coin* (see Figure 2.3), in addition to having four cards in their starting hand.

2.1.2 Start of Turn

A player's turn in Hearthstone starts with any 'at the start of turn' effects resolving. This is also the time when their mana crystals are replenished and increased by one up to a maximum of ten. The player then draws a card from their deck, but if they cannot do this because there are no more cards remaining, their hero receives *Fatigue* damage. This damage starts at one and is increased by one each time a card should be drawn, but cannot. Fatigue damage provides urgency to finish the game before running out of cards.



Secret: After your opponent plays a minion, deal 6 damage to it and any excess to their hero.

This is hidden until a secret action occurs... Watch out!

Figure 2.5: Example of a secret played by the player (left) and one played by the opponent (right).

2.1.3 Player Actions

Actions that a player can take while nothing is happening in the game are referred to as *Player Actions*. During this phase of the turn, the player can perform any action while they can afford the corresponding mana cost. Actions include playing a card, using the hero power, attacking with a minion or hero and ending the turn. These last two actions have no inherent mana cost.

There is a 75-second time limit to a player's turn and if this limit is reached the turn is automatically passed to the opponent.

2.1.4 End of Turn

When a turn is ended any 'at the end of turn' effects are resolved after which the opposing player commences their turn.

2.2 Rules

While there are no official detailed rules provided by Blizzard, a resource describing the game's processes has been created by a community effort (Curse, 2017). The assumptions used to create this resource are based on observations from the game client. It can therefore become inaccurate when something is changed in the client without communication from the manufacturer. However, as this is the most accurate and complete information on the subject, most Hearthstone simulators work on this set of rules.

2.3 Game Characteristics

As mentioned above, Hearthstone is a two-player turn-based zero-sum game. Some other characteristics that are worth discussing include whether or not the game is deterministic, if the available information is perfect or not and what the complexity of the game is.

2.3.1 Level of Determinism

Many cards in Hearthstone have effects that are non-deterministic. For example, the card *Mad Bomber* (see Figure 2.4) splits three damage randomly between all other *characters* (a character is a minion or hero) when it is played. The act of drawing cards from a deck in which the positions of cards are not known also makes Hearthstone non-deterministic.

2.3.2 Level of Information

There are several aspects of Hearthstone that make it a game with imperfect and incomplete information. First, the cards in an opponent's hand are hidden. This causes the player to not know which actions are available to their opponent. Second, the existence of *Secrets*. These are spell cards that create a *secret icon* on the player's hero, indicating that a secret is active. The player that played the secret can see what it is, but for the opponent its identity is hidden until a specific condition is met that triggers it to be revealed and take effect. See Figure 2.5 for examples of how secrets are represented in Hearthstone. These first two aspects cause information in Hearthstone to be imperfect. Finally, the omission of detailed rules by the publisher makes information in Hearthstone incomplete. Because rules and interactions are not confirmed and are subject to change (this has happened in the past (Curse, 2015)), the outcome of an action cannot be guaranteed.

2.4 Variants

Hearthstone has four different modes of play available at the moment:

1. Play
2. Solo Adventures
3. The Arena
4. Tavern Brawl

This research focuses on play-mode, the mode where one agent plays against another agent with their own pre-built decks. Play-mode can be played in a 'Ranked' setting where wins or losses influence a player's rating, or in a 'Casual' setting where no rating is maintained by the game.

Solo Adventures present a single player experience where the user can play against built-in AI opponents in a scripted series of adventures or general practice.

The Arena allows a player to build a new deck by repeatedly presenting them three cards, of which only one can be chosen for their deck. When the player has finished building their deck they play against human opponents until they reach either 12 wins or 3 losses, whichever comes first.

Tavern Brawl is only available for a limited time during the week and presents a new challenge each time; sometimes the player has to build their own deck and other times one will be provided for them. In a Tavern Brawl the player plays against human opponents.

2.5 Strategies

Due to its high number of unique cards, Hearthstone has seen many different strategies evolve. All of these can be broken down into four strategy *arche-types*:

1. Aggro
2. Midrange
3. Control
4. Combo

Aggro strategies aim to end the game quickly and are often heavily minion based. They focus on early aggression in order to overwhelm the opponent before they can set up a good defensive position.

Midrange strategies focus on controlling the middle phase of the game and often include a mix of minions and spells. They tend to be slower than aggro, but attempt to play cards with better effects that naturally cost more.

Control strategies try to control every phase of the game and often contain more spells than minions. They are often slower than midrange, but their late game power is so high that they will win the game if it gets there.

Combo strategies are a special kind, they can pretend to play either an aggro, midrange or control game, but aim to finish by playing a specific combination of cards. This strategy often wins outright when its *combo* is played and this is where the name comes from.

2.6 Hearthstone as a Combinatorial Multi-Armed Bandit Problem

Hearthstone has a combinatorial action space due to the possibility of performing multiple *game actions* per turn. The sequence of game actions is important, since performing one impacts the availability or effectiveness of other game actions. Hearthstone can be modelled as a CMAB problem in the following manner: the different game actions available to a player are the dimensions of the problem, whether to play them or not are the partial actions and a player's entire turn can be viewed as a combined action. The value to optimise would then be the likelihood of winning a game from the resulting board position after the player has executed their combined action for the turn. This value is calculated by an evaluation function, which is used by all the techniques that are researched in this thesis and is discussed in Section 4.5. This answers Research Question 1.

2.7 Current AI

To provide a sufficient overview of the current AI solutions available for Hearthstone, the concept of *simulators* should be discussed first. After this, several existing bots and agents that can play Hearthstone are discussed. The scripted agents available in the game of Hearthstone itself, i.e. a player's opponent in the single-player adventure mode, are not discussed. These agents are often referred to as 'Hearthstone AI', but this thesis does not review them as such, due to a lack of information about their design and inner workings.

2.7.1 Hearthstone Simulators

Because Hearthstone is a proprietary product of Blizzard Entertainment, its source code is not available to the general public. For research that involves Hearthstone it is therefore best to use one of the open-source simulators that are available. The website *HearthSim* (HearthSim, 2018) is created by a community of developers passionate about Hearthstone. It lists a number of known simulators, of which FIREPLACE and SABBERSTONE were the only ones being maintained at the beginning of this research. Simulators in general sometimes include a rudimentary AI, but these are often limited to heuristic rule-based system or shallow, greedy searches.

This thesis has chosen to work with SABBERSTONE. This is also the simulator used for the Hearthstone AI Competition (Dockhorn and Mostaghim, 2018), one of the competitions held at the Computational Intelligence and Games (IEEE CIG) conference in 2018. SABBERSTONE handles the creation of a game of Hearthstone and the processing of game actions performed by players. These game actions are represented in SABBERSTONE by the class *PlayerTask*. In this thesis a game of SABBERSTONE between two players is played by requesting a combined action from the active player and processing the set of PlayerTasks within that combined action. If a combined action does not end with the *EndTurn* task, it is added at the end of the set to ensure that the turn is passed to the next player, making them the active player. The game continues to ask players for actions until it has ended.

2.7.2 Bots for Hearthstone

Having an automated system, often called a *bot*, play the official game of Hearthstone is not permitted through Blizzard's *Terms of Service* and they actively pursue legislative action against bots they discover. Many of these bots are created and sold as part of a paid service and this makes finding information on their inner workings difficult.

A bot that was recently forced to discontinue its service was HEARTHBUDDY (Bossland GmbH, 2014). This came with some predefined behaviours such as 'Control' or 'Rush' and used simulation to look at most two turns ahead in order to find the best available actions. Two more bots that are currently still being maintained are SMARTBOT (HackHearthstone, 2018) and HEARTHRANGER (Rush4x, 2017). As both of these bots are paid services, their inner workings and AI systems are not disclosed.

2.7.3 Agents for Hearthstone

This thesis uses the term *agent* to refer to a system that can only play a game of Hearthstone on one of the aforementioned simulators, not the official version of the game. These systems are often developed to be part of a simulator, to show users an example of how the simulator can be used and encourage them to create their own agent for entertainment or research purposes. After the first Hearthstone AI Competition at IEEE CIG 2018 there are now several agents available for review that participated in the competition. Because this thesis focuses on researching the application and performance of CMAB techniques to the game of Hearthstone, only the agents submitted to the pre-made deck playing track are of interest. In this track, the agent that finished first used a MCTS approach, the agent that finished second used a heuristic approach and the agent that finished third used a flat Monte Carlo search. Other approaches used by agents in the competition include alpha-beta pruning and greedy searches, although not all participating agents have their approach described on the competition's website.

For the experiments in this thesis it is useful to have an agent available with a high level of playing strength. After reviewing the code of several of the agents that participated in the Hearthstone AI Competition at IEEE CIG 2018, it was determined that the agent that finished second could be used for this purpose. The adaptation of this agent to fit the code-framework created for this thesis will be referred to as HeuristicAgent. Unfortunately, the authors have not (yet) given their permission to publicly discuss the design of the agent in detail. Therefore, all that can be said about the agent's approach is that it uses a heuristic scoring function to determine the best PlayerTask to execute.

Chapter 3

Monte Carlo Search Techniques

This chapter discusses several techniques for searching through a state space, all of which are based on the Monte Carlo method that is introduced in Section 3.1. A basic version of Monte Carlo search is explained in Subsection 3.1.1. Section 3.2 presents Monte Carlo Tree Search. Subsection 3.2.1 describes an extension to this called Hierarchical Expansion. Section 3.3 reviews the Naïve Monte Carlo technique. The last technique that is discussed is Linear Side Information in Section 3.4.

3.1 The Monte Carlo Method

The namesake of this method is an area of Monaco, famous for its Casino, which points to a similarity amongst the techniques that are based on the method: all attempt to find an optimal solution through more-or-less random exploration of the state space (Metropolis, 1987). When applied to games, the method tries to approximate the value of a game state by simulating what the result of a game would be when played out to its conclusion. Such a *playout* of a game is generally performed by playing random moves.

Much like a gambler, the method is restrained by a budget, often measured in time or number of iterations. This budget can only be spent and not gained, and merely functions as a measure of how much computing is performed.

One of the advantages of this type of approach is that the only requirement is an evaluation function to determine the value of individual game states. This evaluation function does not require the use of domain-specific knowledge, which allows the Monte Carlo method to be an interesting tool in the field of general game playing (Björnsson and Finnsson, 2009; Gaina *et al.*, 2018). Due to the inherent randomness in the design of the method, convergence to the optimal solution can be slow. This can be seen as a disadvantage, but due to strong performance in a variety of domains, as mentioned in Subsection 1.1.2, randomness can also be considered to be a strength of the method. However, it should be noted that many of the enhancements to Monte Carlo techniques attempt to make them less random, such as adding domain knowledge to ignore certain parts of the state space or balancing the budget between exploring and exploiting. Another important aspect of the method is its dependence on the quality of the playout. The more accurately the playouts represent genuine gameplay, the more accurate state evaluations will be, which in turn helps to decide where to most effectively spend the computing budget.

3.1.1 Monte Carlo Search

This is the most basic version of searching using the Monte Carlo method, also known as Flat Monte Carlo Search. It only considers the child nodes of the root position and spends its budget on playouts to narrow down the value of the moves those nodes represent. This approach is essentially MCTS with a limited depth of one. Algorithm 1 shows a search where each available move is sampled once and the best evaluated move is returned as the solution to the search.

```
  input  : game state $s$, move set $M$
1 initialise $Scores \leftarrow [n]$
2 foreach $m \in M$ do
3     $s' = s$.Copy()
4     $s''$ = Play(m, s')
5     $Scores[m]$ = Playout(s'')
6 end
7 $m_{max} = \arg\max Scores$
  output: move $m_{max}$
```

Algorithm 1: Flat Monte Carlo Search

3.2 Monte Carlo Tree Search

MCTS is a best-first search (Coulom, 2007) that builds a search-tree by sequentially iterating over four phases (Chaslot *et al.*, 2008):

1. Selection: from the perspective of a node, select a child node according to a *selection strategy*.
2. Expansion: from the perspective of a leaf node, create one or more child nodes according to an *expansion strategy*.
3. Playout: from the perspective of a leaf node, play out the game according to a *playout strategy*.
4. Backpropagation: from the perspective of a leaf node, evaluate the simulated game state and propagate the evaluation back to the root node according to a *backpropagation strategy*.

See Figure 3.1 for a visual presentation of the four phases. This figure is based on the one presented in Chaslot *et al.* (2008). After the computation budget has been spent, the best child node of the root is selected as the final move, which is returned as the solution to the search.

A selection strategy selects one of the child nodes of the node that it is invoked upon. MCTS starts by selecting the root, which represents the current state of the game. Upper Confidence Bound applied to Trees (UCT) is an example of a selection strategy (Kocsis and Szepesvári, 2006). UCT selects the next node by using the following formula:

$$I_t = \max_{i \in K}\{\overline{R}_i + 2 \times C \times \sqrt{\frac{\ln V_p}{V_i}}\} \tag{3.1}$$

In Equation 3.1, I_t is the node selected at round t from the set of available nodes K, $\overline{R}_i$ is the average evaluation for node i, V_i is the number of times node i has been visited and V_p is the number of times i's parent node p has been visited. The constant C has to be tuned experimentally.

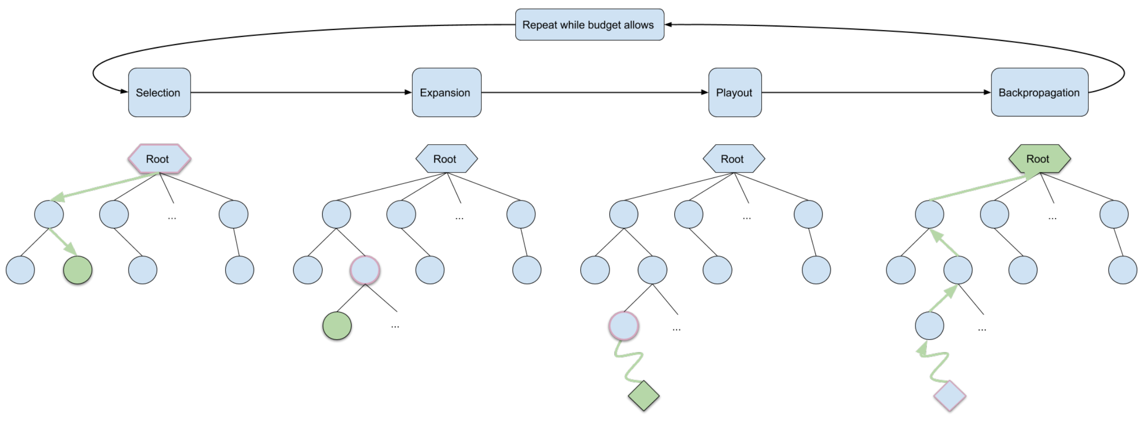


Figure 3.1: The four phases of Monte Carlo Tree Search.

An expansion strategy dictates when and how the child nodes of a node should be generated. The children of a node represent the set of game states that can be reached from the current node. An example of an expansion strategy is to only expand a node after it has been visited T number of times, this is referred to as Minimum-T Expansion in this thesis. The strategy discussed in Subsection 3.2.1 extends upon this expansion strategy.

The playout strategy starts with a copy of the game state as stored in the leaf node and continuously produces moves and executes them, moving the game forward until it reaches an end state or stop condition. This is also often referred to as a *simulation strategy.* A trivial strategy is to create random moves, but this can weaken the resulting play strength (Chaslot *et al.*, 2008).

A backpropagation strategy takes the evaluation score of a simulated game state from a leaf node and moves back up the game tree to the root node. Each node visited during this process has its value updated with the evaluation score. A straightforward way to calculate the value of a node is to take the average score over the number of visits.

3.2.1 Hierarchical Expansion

Hierarchical Expansion (HE) is an extension of MCTS and designed to cope with the action space of the Combinatorial Multi-Armed Bandit problem (Roelofs, 2015). In an effort to deal with a high branching factor as a result of the combinatorial action space, HE modifies the Expansion phase of MCTS. It expands into the partial actions for one dimension per level, which means one ply can encompass multiple levels. MCTS extended with HE is referred to as Hierarchical Monte Carlo Tree Search (H-MCTS) in this thesis. How HE changes the structure of the search tree can be seen in Figure 3.2 where a default MCTS expansion and HE are compared. These figures are based on those presented in Roelofs (2015). The combined action that is constructed in a single level in MCTS is in H-MCTS instead incrementally created and stored in the nodes. This approach allows exploration on the level of partial actions. When a game has partial actions that are significantly more valuable than others, HE will allow the search to be focused on branches of the tree that contain these valuable partial actions. HE assumes that some ordering for selecting which dimension to expand into is known or that such an ordering can be learned during the search. Because of its focus on partial actions and their order, HE can exploit high-value action sequences that are represented by a non-linear component in the reward function.

As a solution to the search, H-MCTS selects the best child node, starting at the root, until a combined action is constructed that contains a partial action for all dimensions. It should be noted that when the available computational budget is not adequate with respect to the complexity of the search space, it is possible that H-MCTS cannot create a deep enough tree to explore all partial actions. This means that when creating the solution to the search, H-MCTS might not be able to include a choice for all available dimensions. This is described in Roelofs (2015) as the Move Completion problem. How this problem is handled within the domain of Hearthstone is discussed in Subsection 4.2.1.

3.3 Naïve Monte Carlo

Naïve Monte Carlo (NMC) is an algorithm designed to solve the CMAB problem, using the Naïve Sampling process (Ontañón, 2013). This process works under the naïve assumption that partial actions do not influence each other, i.e. the reward function for a combined action can be rewritten as summation of rewards for individual partial actions. Each iteration of the process a choice is made whether to *Explore* or *Exploit*, according to a policy π_0. If Explore is chosen, a partial action is selected for each dimension independently according to a policy π_l, forming a combined action which is sampled and stored along with its value in a set of combined actions, G. If Exploit is chosen, a combined action is selected from G according to a policy π_g and sampled. At the end of the search the combined action in G with the best value is returned as the solution. In this thesis, the policies π_0 and π_g are ϵ-greedy policies and π_l creates random combined actions.

In Ontañón (2013) Naïve Monte Carlo Tree Search (N-MCTS) is presented. N-MCTS replaces the Selection and Expansion phases with the $NaïveSelectAndExpandNode$ algorithm, which is shown in Algorithm 2. Although it is designed for Real-Time Strategy games, the idea of incorporating the Naïve Sampling process into MCTS can still be applied within the domain of Hearthstone. Section 4.3 discusses how to adjust the $NaïveSelectAndExpandNode$ algorithm to fit this domain.

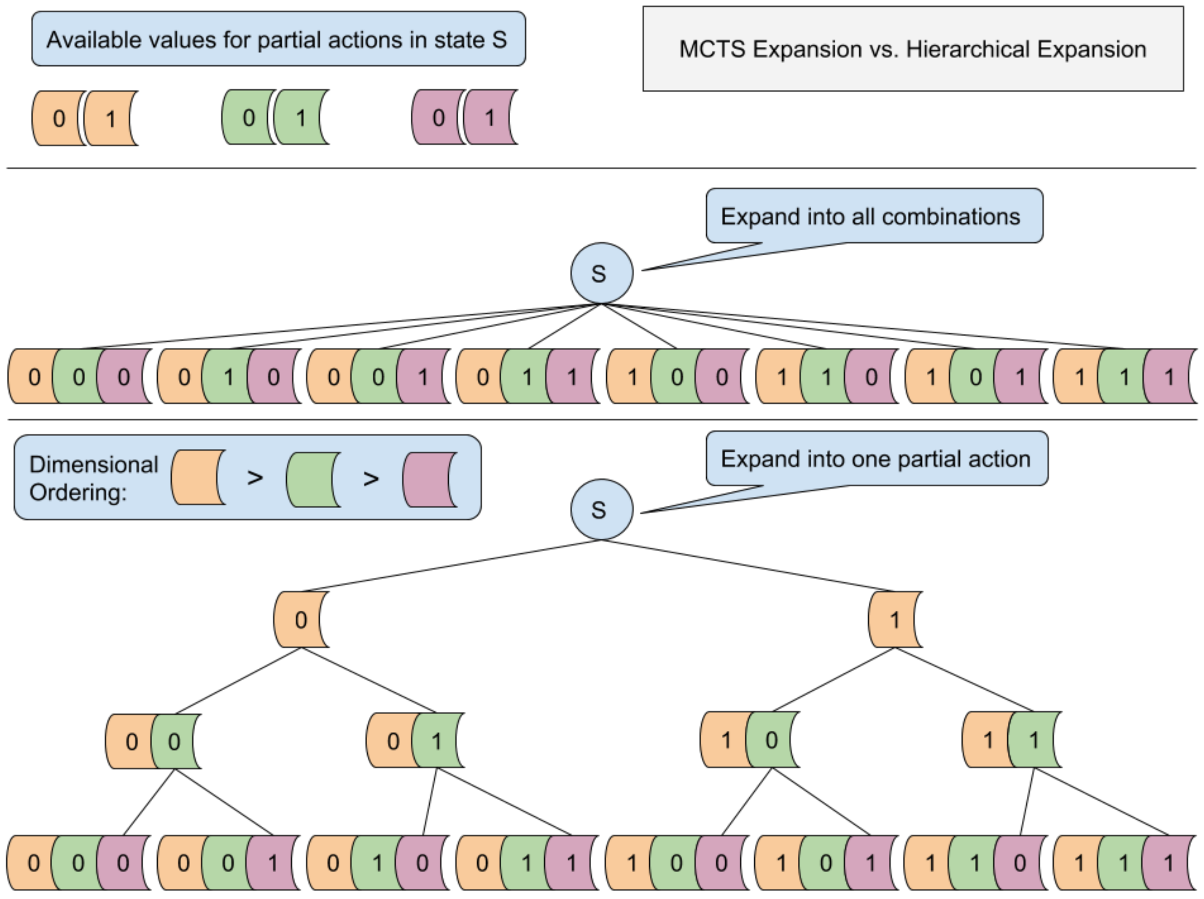

Available values for partial actions in state S
MCTS Expansion vs. Hierarchical Expansion
Expand into all combinations
S
Dimensional Ordering:
Expand into one partial action
S

Figure 3.2: MCTS Expansion (top) versus Hierarchical Expansion (bottom).

```
input  : node $n_0$
1  if canMove($max, n_0.state$) then
2      $player = max$
3  else
4      $player = min$
5  end
6  $\alpha$ = NaïveSampling($n_0.state, player$)
7  if $\alpha \in n_0.children$ then
8      return NaiveSelectAndExpandNode($n_0.child(\alpha)$)
9  else
10     $n_1$ = newTreeNode(fastForward($n_0.state, \alpha$))
11     $n_0$.addChild($n_1, \alpha$)
12     return $n_1$
13 end
output: selected node $n_s$
```

Algorithm 2: NaïveSelectAndExpandNode

3.4 Linear Side Information

Linear Side Information (LSI) (Shleyfman *et al.*, 2014) divides its search process into two phases: Explore and Exploit, each having a separate iteration budget. In the Explore phase, the set of all possible combined actions is reduced to a relatively small subset. To generate this subset a table of Side Information (Chen, Liu, and Lyu, 2013a) is created for each dimension and its available partial actions. This table contains the average reward after sampling each dimension's available partial actions an equal number of times, based on the available budget. To sample a partial action a playout is performed either within an otherwise empty combined action, or one that consists of randomly generated partial actions for the other dimensions. When this Side Information per dimension is normalised it can be used as a probability distribution for selecting partial actions during the generation of the subset of combined actions.

In the Exploit phase, the generated subset of combined actions is reduced by using Sequential Halving. This algorithm samples each combined action in the subset a number of times according to the available budget and then removes half of the combined actions; those that performed worse on average. This continues until only one combined action remains, which is returned as the solution to the search. The Sequential Halving algorithm as presented in Karnin, Koren, and Somekh (2013) is shown in Algorithm 3.

```
  input  : Exploitation budget $T$
1 initialise $S_0 \leftarrow [n]$
2 for $r = 0$ to $\lceil \log_2 n \rceil - 1$ do
3     sample each arm $i \in S_r$ for $t_r = \lfloor \frac{T}{|S_r| \lceil \log_2 n \rceil} \rfloor$ times, and let $\hat{p}_i^r$ be the average reward
4     $S_{r+1} \leftarrow$ the set of $\lceil |S_r|/2 \rceil$ arms in $S_r$ with the largest empirical average
5 end
  output: arm in $S_{\lceil \log_2 n \rceil}$
```

Algorithm 3: Sequential Halving

Chapter 4

Searching in Hearthstone

To be able to use the search techniques described in Chapter 3 within the domain of Hearthstone and the chosen simulator SabberStone, some adjustments and enhancements have to be made. Section 4.1 presents three enhancements that are applied to each of the search techniques. Section 4.2 describes how H-MCTS is modified to fit the domain of Hearthstone. Section 4.3 explains any modifications to N-MCTS and Section 4.4 does so for LSI. To close this chapter, Section 4.5 introduces the evaluation function that is used by all search techniques.

4.1 Search Enhancements

As a first step to making Monte Carlo search techniques applicable to Hearthstone this thesis restricts the domain to three decks. These decks are manually constructed to be playable by a single Hero class in an effort to not introduce the Hero class as a variable in the experiments. For the exact decks used during the experiments see Subsection 5.1.1. Note that the decks do not include any cards that use the discover mechanic, which introduces a significant increase in complexity. The following subsections discuss three enhancements that are applied to all search techniques.

4.1.1 Determinisation of the Search Space

One approach for a search technique to deal with hidden information in a game state is to create a *determinisation* of that state (Ward and Cowling, 2009). This is essentially a way of guessing what the hidden information is and then finding the correct action for that particular scenario. In the domain of Hearthstone we have the following hidden information in any given state:

- The order of the cards in the player's deck
- The cards in the opponent's hand
- The order of the cards in the opponent's deck

To create a determinisation of a game state in Hearthstone, each of these sets of unknown cards should have their content specified. Instead of using the entire pool of cards in Hearthstone to generate these sets, the cards in the player's deck can be limited to the deck of cards that they started the game with, as this information is known to the player. From this set, the cards that they have already played and the ones they have in their hand can be removed. The resulting set is then randomised and used to replace the player's deck in the game state.

For the opponent the determinisation process is similar, except that it is not known with which deck of cards they started the game. Expert players will often theorise or guess what cards could be in their opponent's deck based on the cards that they have seen them play so far. This thesis approaches the problem of predicting the opponent's deck in the same way. The opponent's played cards are compared to the cards in the known decks that the domain is restricted to. If this narrows the possibilities down to one deck, that deck is selected to represent the opponent's deck. If two or more decks share the cards played so far by the opponent, a default deck will be used, see Subsection 5.1.1 for its contents. Once a

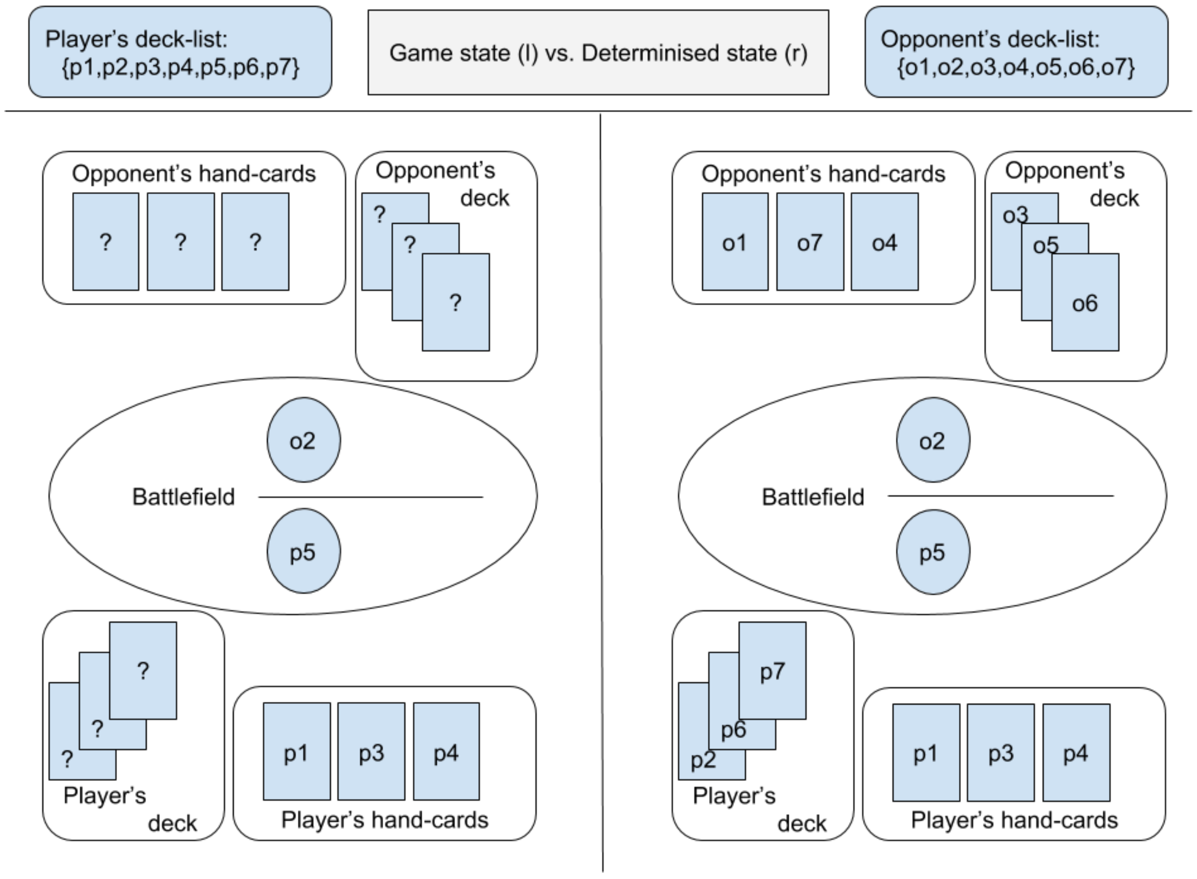


Figure 4.1: An example of a game state (left) versus a determinised state (right).

deck is selected for the opponent, any played cards are removed. The resulting set is then randomised and first used to replace the hidden information in the opponent's hand and after that the opponent's deck.

A simplified example of determinisation can be seen in Figure 4.1. It should be noted that any revealed cards, or cards whose position is known (such as the Coin), should be excluded from this process.

4.1.2 Ensemble Search

Creating a determinisation is essentially looking at one of many possible configurations of the hidden information with a game state. For the solution of a single determinised search to be useful, one would have to be able to accurately predict the most likely configurations of the hidden information. This level of prediction is often not possible and requires specific domain knowledge. However, using multiple different determinisations in an ensemble has shown to improve performance for determinised search techniques (Cowling, Ward, and Powley, 2012b). This thesis chooses to use the approach of *Determinised UCT* (Cowling, Powley, and Whitehouse, 2012a), which creates a search tree for each individual determinisation in the ensemble. The computation budget is evenly distributed among these trees and each provides a solution at the end of the search. Voting (Nijssen and Winands, 2012) can be used to determine the overall solution of a search that uses an ensemble of determinisations. However, due to the nature of Hearthstone, combined actions are not easily comparable (e.g. some may contain partial actions that other do not). This thesis therefore proposes a different solution algorithm for Ensemble Search in Hearthstone that votes on individual partial actions instead. In the case where multiple partial actions have the highest number of votes, one of them is chosen at random. This approach is presented in Algorithm 4.

```
input  : solution set $S$, game state $G$
1  initialise $A \leftarrow []$, $Votes \leftarrow [][]$
2  foreach $s \in S$ do
3      foreach $task \in s$ do
4          $Votes[task] = Votes[task] + 1$
5      end
6  end
7  do
8      $T \leftarrow G$.Options()
9      $T_v = T \cap Votes$
10     $t_{max} = \arg\max_t T_v[t]$
11     $A$.AddTask($t_{max}$)
12     $G \leftarrow G$.Process($t_{max}$)
13 while !$A$.IsComplete() && !$G$.IsDone()
output: solution $A$
```

Algorithm 4: Solution algorithm for Ensemble Search in Hearthstone.

4.1.3 Move-Average Sampling Technique

Move-Average Sampling Technique (MAST) (Björnsson and Finnsson, 2009) aims to improve the quality of the playouts that Monte Carlo techniques use. This technique creates a table containing a total value and visit count for each possible action in the game. At the end of a playout each executed action has its value updated with the evaluation of the playout. During a playout, the action with the highest average value is selected to be executed. A policy is used to balance exploration of different actions versus exploitation of the best action, such as ϵ-greedy or UCB1 (Powley, Whitehouse, and Cowling, 2013).

MAST can be used as a playout strategy in the domain of Hearthstone by recording a value for each partial action that is performed during a playout. This value is the evaluation of the state at the end of the playout. When the forward model has to decide what partial action to perform, it can reference the average value of each available partial action and choose one of them, using one of the aforementioned policies. Table 4.1.3 shows an example of some entries from a MAST table in the middle of a game. Note that the actual table contains many more entries, this example shows just a few manually selected entries. Because the values used in MAST are considered outside of any game context, the table can be re-used between searches.

Note the inclusion of the 'EndTurnTask' in the MAST table. This is the action that passes the turn to the opposing player and should always be included in a combined action for Hearthstone. It is also always an available option and because sometimes it is correct to do nothing, this thesis chooses not to disregard the task when using MAST.

PlayerTask	TotalValue	Visits
PlayCardTask =>[Player1] play 'Arcane Shot[150]'(SPELL) 'Rexxar[6]'	0	825
MinionAttackTask =>[Player1] 'Bloodfen Raptor[16]'(MINION) attack 'Rexxar[6]'	1748.5	10922
HeroPowerTask =>[Player1] using 'Steady Shot[5]'	383	9898
MinionAttackTask =>[Player1] 'Stonetusk Boar[13]'(MINION) attack 'Rexxar[6]'	-400.5	3758
EndTurnTask =>[Player1]	-544.5	70618

Table 4.1: Sample entries from a MAST table.

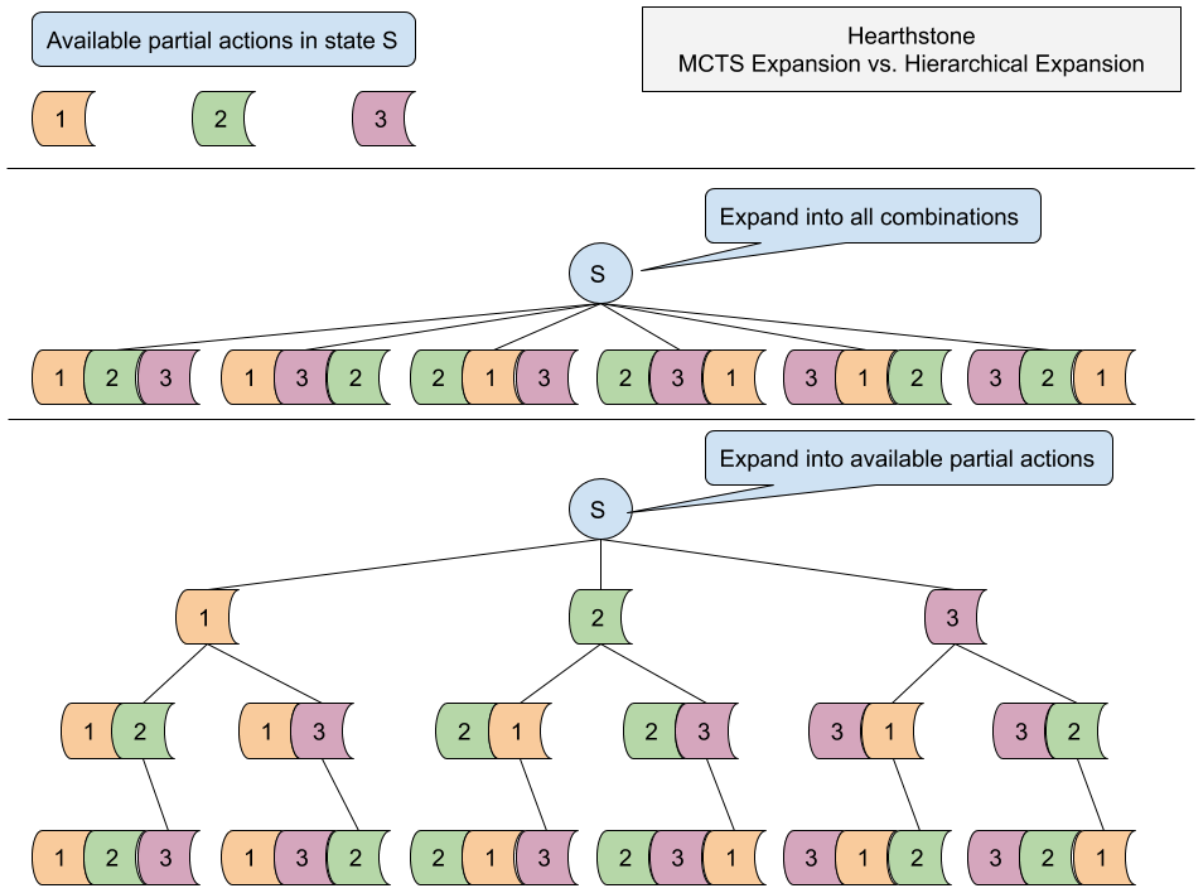

Available partial actions in state S
1
2
3
Hearthstone
MCTS Expansion vs. Hierarchical Expansion
Expand into all combinations
S
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
Expand into available partial actions
S
1
2
3
1 2
1 3
2 1
2 3
3 1
3 2
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Figure 4.2: Hearthstone adaptation of MCTS Expansion (top) versus Hierarchical Expansion (bottom).

4.2 Hierarchical Monte Carlo Tree Search for Hearthstone

The H-MCTS technique does not need any adjusting to the domain of Hearthstone, once we have already modelled the domain as a CMAB problem. Figure 4.2 shows how the Hierarchical Expansion phase is adjusted to fit the domain of Hearthstone. One aspect of this domain does change how the search tree is built during Hierarchical Expansion: the constrained availability of actions. It can happen that an action either becomes available, or is no longer available once another action is executed. An example of constrained availability of actions is shown in Figure 4.3. This tree structure is reminiscent of the structure of a trie, as discussed by Goodrich and Tamassia (2006), which is a tree-based data structure for storing strings. The main application of tries is in information retrieval from textual data by supporting pattern matching and prefix matching. An optimisation to the structure of tries is to compress it when a node has only one child. However, because ending the turn is a partial action that is always available to a player, the only positions where compression of the search tree could be applied are positions where the player has no other option that to end their turn. Due to this characteristic of the problem domain, this thesis does not look further into applying optimisations based on tries.

4.2.1 Move Completion

The problem of Move Completion as described in Subsection 3.2.1 can be solved by using the values from the MAST table to select partial actions when HMCTS does not return with a complete combined action. It should also be noted that simply adding the ‘EndTurnTask’ to the end of the incomplete combined action would solve the problem. However, because this thesis chooses not to disregard the ‘EndTurnTask’ in the MAST table, using those values will also consider this task and therefore provide a more complete solution to the problem of Move Completion.

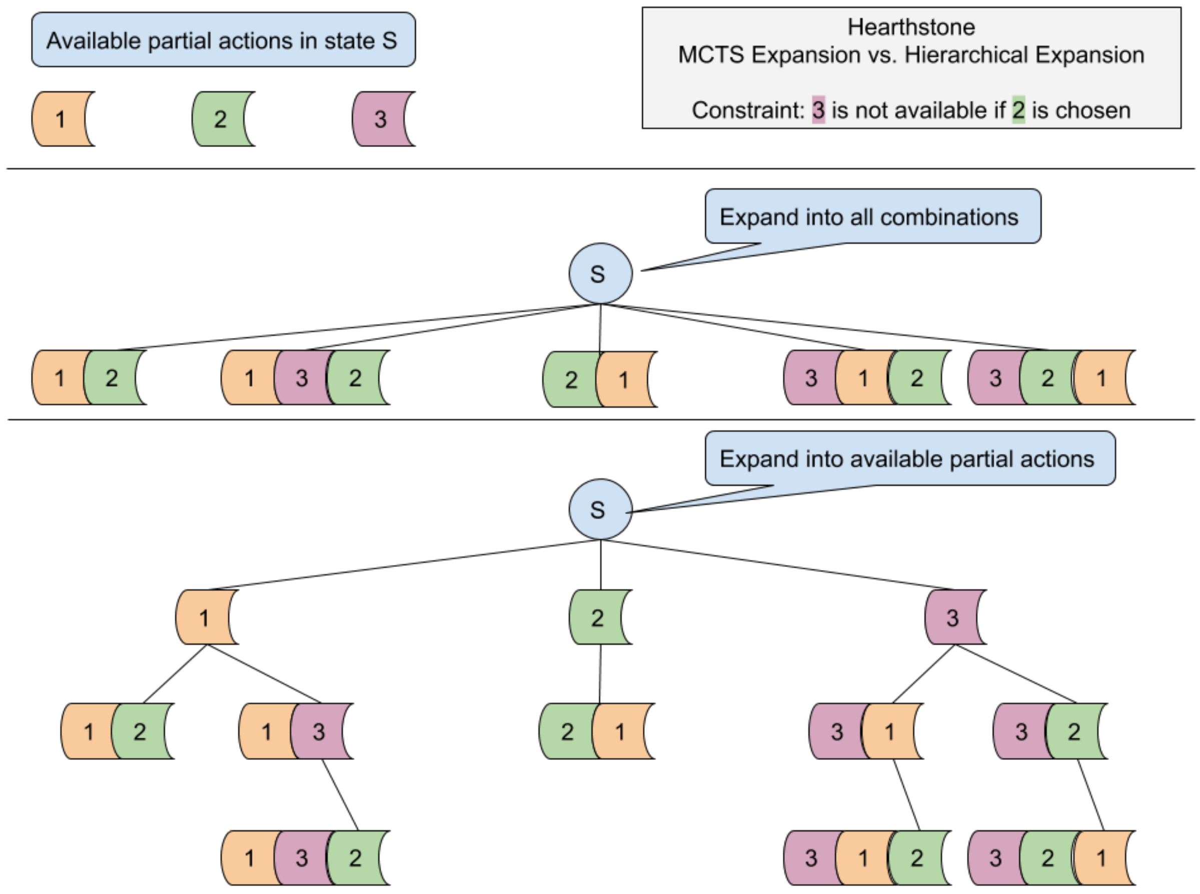

Available partial actions in state S
1
2
3
Hearthstone
MCTS Expansion vs. Hierarchical Expansion
Constraint: 3 is not available if 2 is chosen
Expand into all combinations
S
1 2
1 3 2
2 1
3 1 2
3 2 1
Expand into available partial actions
S
1
2
3
1 2
1 3
2 1
3 1
3 2
1 3 2
3 1 2
3 2 1

Figure 4.3: Illustration of constrained availability of actions in Hearthstone.

4.2.2 Dimensional Ordering

The problem of Dimensional Ordering for H-MCTS comes with an additional limitation in the domain of Hearthstone. Generally the dimensions of a CMAB problem have multiple different values. In Hearthstone however, a game action can either be executed or not. This means that the ordering of dimensions only changes the order of the creation of child nodes during the expansion phase.

The order in which an action sequence is executed can influence its value or effectiveness, as emphasised by the sarcastic phrase 'Draw last' sometimes used by Hearthstone fans. It is an attempt to indicate their dissatisfaction with a particular action sequence played by another player, which could have been ordered better to gain more information before making relevant choices. When given the option, it is most often better to draw cards first, because that gives the player more information about the game state for any other choices they have to make that turn. This is a typical play pattern in card games: prioritise actions that provide the player with more information. Another typical play pattern can be described as the iterative planning of a turn. An example of this is when a player has already decided that they are going to play a specific card, but have not yet chosen its target.

To order the game actions available to a player in a game of Hearthstone, the value of an action needs to be defined. This thesis researches the following ways to measure the value of an action in Hearthstone:

- Mana Cost
- Task Type
- Average Evaluation
- Shannon Entropy

Mana cost refers to the cost of either playing cards or activating the hero power. It therefore does not cover all available actions because attack actions have no inherent cost to them. Ordering by descending mana cost means that the search expands into the most expensive action first, which is often the most powerful action. Ordering by ascending mana cost means that the search expands into actions that contribute to taking multiple actions in a single turn before other actions.

Task type refers to the types of actions as defined by the SabberStone simulator (i.e. ‘PLAY_CARD’, ‘HERO_POWER’, ‘MINION_ATTACK’). Ordering by task type means that the search expands into actions of a specific type first over other types. It could for example be valuable to examine playing cards from the player’s hand before attacking with any minions.

Average Evaluation refers to the average value of a task’s evaluation values. To calculate this average the data generated by MAST can be used. Ordering by descending average evaluation means that the search first expands into actions that are more likely to lead to states that are beneficial to the player.

Shannon Entropy (Shannon, 1948) refers to the entropy of a task’s evaluation values. This value can be a measure for the amount of information contained in a task’s evaluation values. For this metric the data generated by MAST can be used when the values are stored as a collection instead of a total value. The entropy of a task X can then calculated by determining the frequency of its evaluation values and using those frequencies as $P(x)$ in the following equation:

$$H(X) = -\sum_{x} P(x)\log_2[P(x)] \tag{4.1}$$

Ordering by descending entropy means that the search expands into tasks that provide more information with respect to their evaluation value before others.

In addition to these measurements of the value of an action in Hearthstone, there is also the default ordering of actions as provided by the simulator. This ordering can be used as a baseline when comparing the performance of different orderings, which is an experiment presented in Section 5.2.

4.3 Naïve Monte Carlo Tree Search for Hearthstone

The N-MCTS technique requires only a minor adjustment to the domain of Hearthstone, once we have already modelled it as a CMAB problem. Because Hearthstone is turn-based, the *FastForward* function in Algorithm 2 can be simplified to having the game state process the sampled action and return the resulting state. It should be noted that when the $NaïveSelectAndExpandNode$ algorithm enters the recursive call (line 8), the iteration count should be incremented because there will be an additional sample generated.

4.4 Linear Side Information for Hearthstone

LSI does not require any specific adjustments to the domain of Hearthstone. Though it should be noted that the Side Information table is more compact than normal. In other CMAB domains, a row in the table would contain an average evaluation for each possible value that the partial action corresponding to that row can have. In the domain of Hearthstone however, a row only contains a single average evaluation, for the case when the partial action is played.

One general adjustment that LSI has to have is the ability to be run on a time-limited budget, since Hearthstone’s turns are limited by time. LSI divides its budget between the two phases at the start of the search, which means that an estimation needs to be made about the amount of computing time required for one iteration. Once this estimation is determined, the time budget can be converted into a number of iterations using the following formula:

$$B_i = \frac{B_t}{d_e \times \sigma} \tag{4.2}$$

In Equation 4.2, B_i represents the iteration budget, B_t represents the time budget, d_e represents the estimated duration of a single iteration and σ represents a factor which is used as a safety-margin to ensure that the time limitation is not exceeded when the estimation underestimated the actual duration.

This thesis researches two methods of estimating the duration of a single iteration of LSI. The first method calculates d_e as the average duration of 25 sample playouts from the root state and will be referred to in this thesis as *AverageSampleEstimation* (ASE). The second method calculates d_e as the average duration of an iteration in the previous search, falling back on ASE when this data is not available, and will be referred to in this thesis as *PreviousSearchEstimation* (PSE). The performance and accuracy of both methods is tested in an experiment presented in Subsection 5.1.8.

4.5 Hearthstone State Evaluation Function

As discussed in Chapter 3, each of the search techniques that use the Monte Carlo method have an evaluation function to determine the value of a state. This thesis presents an algorithm that is used by all search techniques as their evaluation function, which removes any bias when comparing the performance of the different techniques. To calculate an evaluation value for a state, the algorithm takes into account several properties of the state from the root player's point of view as itemised below:

- Number of minions controlled by the root player (M_r)
- Number of minions controlled by the opponent (M_o)
- Health points of the root player (HP_r)
- Health points of the root player's minions with taunt (HP_{rm^t})
- Power of the root player's minions that are able to attack (P_{rm})
- Power of the root player's weapon (P_{rw})
- Fatigue damage the root player would take at the start of their next turn (FD_r)
- Health points of the opponent (HP_o)
- Health points of the opponent's minions with taunt (HP_{om^t})
- Power of the opponent's minions that are able to attack (P_{om})
- Power of the opponent's weapon (P_{ow})
- Fatigue damage the opponent would take at the start of their next turn (FD_o)

Some of these properties are used to estimate the number of turns to reduce a player's health points to zero or less.

$$T_r = \frac{HP_r + HP_{rm^t} - FD_r}{P_{om} + P_{ow}}, T_o = \frac{HP_o + HP_{om^t} - FD_o}{P_{rm} + P_{rw}} \tag{4.3}$$

In Equation 4.3, T_r represents the estimated number of turns before the root player's health points reach zero and T_o represents this number for the opponent. In the case where a player has no minions that can attack and no weapon equipped, an arbitrarily high number is returned (e.g. MAXINT) to prevent division by zero. These estimations are then used in an algorithm that determines the actual evaluation value. The algorithm is presented in Algorithm 5. Lines 1 and 3 check to see whether either player can be killed within a single turn, if so, the appropriate value is returned (i.e. a 1 for win, -1 for loss). Line 5 checks to see whether either player is within four turns of dying. If neither player is, the algorithm proceeds to compare the health and number of creatures for each player in order to determine who is advantaged in a longer game. If either player is close to dying, the algorithm looks at the difference between these numbers to determine who is advantaged in the race. The evaluation values returned by the algorithm are positive when the root player is more likely to win the game and negative when the opponent is more likely to win, while zero indicates a stalemate.

```
input  : $T_r$, $T_o$, $HP_r$, $HP_o$, $M_r$, $M_o$
if $\lceil T_o \rceil == 1$ then
    $e = 1$
else if $\lceil T_r \rceil == 1$ then
    $e = -1$
else if $(T_r \geq 4) \wedge (T_o \geq 4)$ then
    if $(HP_r > HP_o) \wedge (M_r > M_o)$ then
        $e = 0.75$
    else if $(HP_r > HP_o) \wedge (M_r == M_o)$ then
        $e = 0.25$
    else if $(HP_r > HP_o) \wedge (M_r < M_o)$ then
        $e = 0.1$
    else if $(HP_r == HP_o) \wedge (M_r > M_o)$ then
        $e = 0.33$
    else if $(HP_r == HP_o) \wedge (M_r == M_o)$ then
        $e = 0$
    else if $(HP_r == HP_o) \wedge (M_r < M_o)$ then
        $e = -0.33$
    else if $(HP_r < HP_o) \wedge (M_r > M_o)$ then
        $e = -0.1$
    else if $(HP_r < HP_o) \wedge (M_r == M_o)$ then
        $e = -0.25$
    else
        $e = -0.75$
    end
else
    $T_\delta = T_o - T_r$
    if $T_\delta < -1$ then
        $e = 0.5$
    else if $T_\delta > 1$ then
        $e = -0.5$
    else
        $e = 0$
    end
end
output: $e$
```

Algorithm 5: Hearthstone State Evaluation

Chapter 5

Experiments & Results

This chapter presents the experiments that have been performed as part of this thesis and analyses their results. Section 5.1 introduces the experimental setup and the experiments performed to determine the appropriate settings for each search technique. Section 5.2 describes the experiment that compares the performance of the different approaches to Dimensional Ordering for H-MCTS. Section 5.3 explains the experiments that compare the performance of the CMAB techniques researched in this thesis.

5.1 Experimental Setup

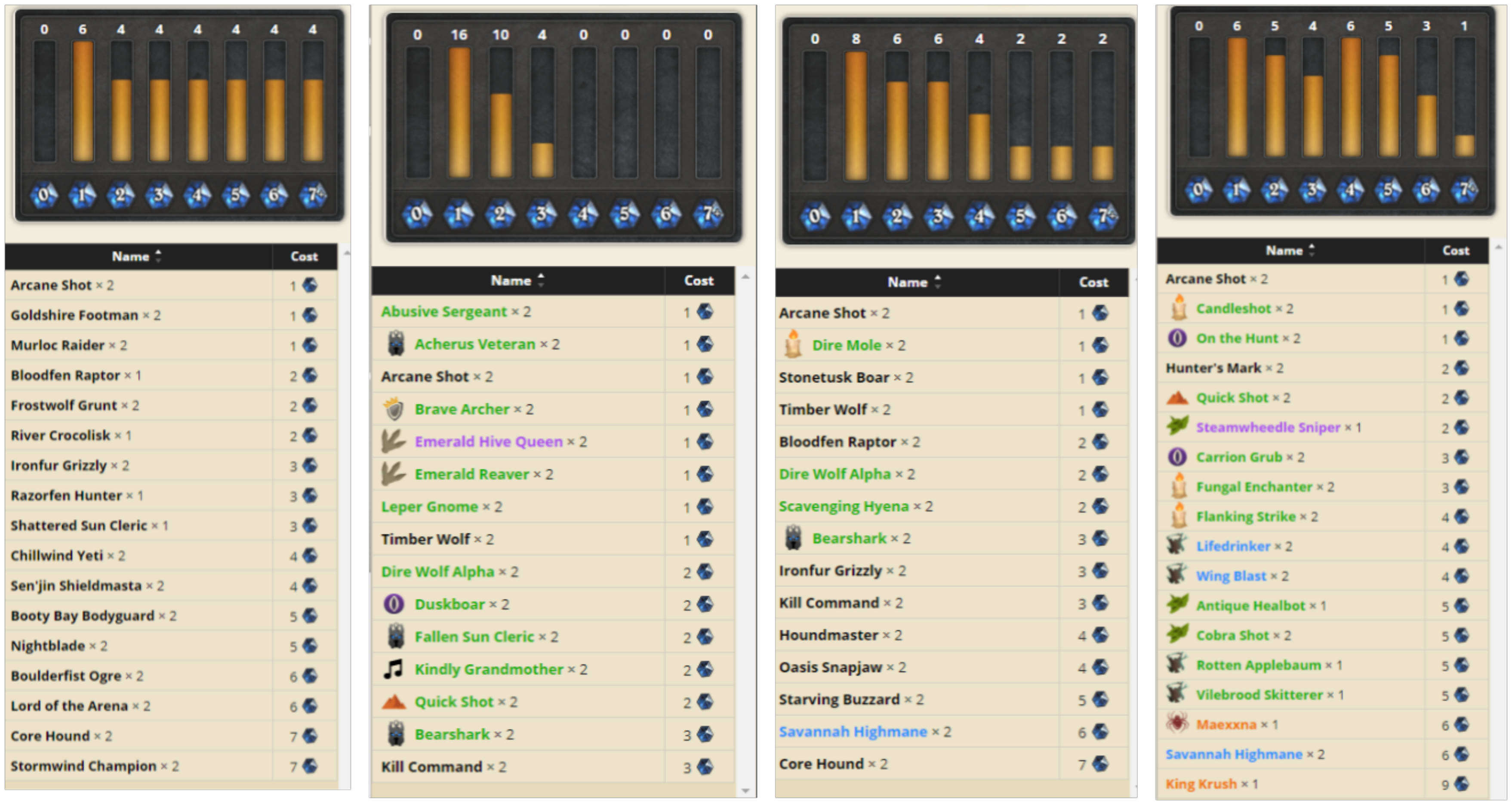
Two agents have been created to act as a performance baseline. RandomAgent is an agent that randomly selects an available action until the EndTurn task is selected. This represents a lower bound on the performance of an agent. HeuristicAgent is the adaptation of the submitted agent that finished second in the Hearthstone AI Competition 'Pre-made Deck Playing'-track at CIG 2018, as discussed in Section 2.7. This represents a high level of performance of an agent.

This thesis researches and experiments with three different techniques: H-MCTS, N-MCTS and LSI, as described in Chapter 3. Each of these techniques has its own settings for which appropriate values should be determined, but they also share the following settings: Playout Length, Type of Playout and Ensemble Size. Determining the optimal values for each of these settings are discussed in Subsections 5.1.3, 5.1.4 and 5.1.5, respectively.

To run a match between two agents, an even number of games are created using the SabberStone simulator. For each game the agents are randomly assigned one of the three decks, which can be the same for both agents. See Subsection 5.1.1 for a discussion on which decks are used. The player that starts a game alternates between the agents.

5.1.1 Decks

As discussed in Section 4.1, this thesis restricts the domain of Hearthstone to three decks. These decks are visually presented in Figure 5.1 in a form that is often used to present Hearthstone decks. A fourth deck, the default deck to be used during determinisation when an opponent's deck cannot be identified, is also included. The cards included in these decks are manually selected based on the author's preference, but also due to availability and compatibility with the simulator. The name and design of the decks; Aggro, Midrange and Control, resemble the strategies as described in Section 2.5 as much as possible. Combo was not chosen to be included as one of the deck's strategies because of the amount of domain knowledge that would have to be included in order for a deck of this type to be played effectively.



0	1	2	3	4	5	6	7
0	6	4	4	4	4	4	4

Name	Cost
Arcane Shot × 2	1
Goldshire Footman × 2	1
Murloc Raider × 2	1
Bloodfen Raptor × 1	2
Frostwolf Grunt × 2	2
River Crocolisk × 1	2
Ironfur Grizzly × 2	3
Razorfen Hunter × 1	3
Shattered Sun Cleric × 1	3
Chillwind Yeti × 2	4
Sen'jin Shieldmasta × 2	4
Booty Bay Bodyguard × 2	5
Nightblade × 2	5
Boulderfist Ogre × 2	6
Lord of the Arena × 2	6
Core Hound × 2	7
Stormwind Champion × 2	7

0	1	2	3	4	5	6	7
0	16	10	4	0	0	0	0

Name	Cost
Abusive Sergeant × 2	1
Acherus Veteran × 2	1
Arcane Shot × 2	1
Brave Archer × 2	1
Emerald Hive Queen × 2	1
Emerald Reaver × 2	1
Leper Gnome × 2	1
Timber Wolf × 2	1
Dire Wolf Alpha × 2	2
Duskboar × 2	2
Fallen Sun Cleric × 2	2
Kindly Grandmother × 2	2
Quick Shot × 2	2
Bearshark × 2	3
Kill Command × 2	3

0	1	2	3	4	5	6	7
0	8	6	6	4	2	2	2

Name	Cost
Arcane Shot × 2	1
Dire Mole × 2	1
Stonetusk Boar × 2	1
Timber Wolf × 2	1
Bloodfen Raptor × 2	2
Dire Wolf Alpha × 2	2
Scavenging Hyena × 2	2
Bearshark × 2	3
Ironfur Grizzly × 2	3
Kill Command × 2	3
Houndmaster × 2	4
Oasis Snapjaw × 2	4
Starving Buzzard × 2	5
Savannah Highmane × 2	6
Core Hound × 2	7

0	1	2	3	4	5	6	7
0	6	5	4	6	5	3	1

Name	Cost
Arcane Shot × 2	1
Candleshot × 2	1
On the Hunt × 2	1
Hunter's Mark × 2	2
Quick Shot × 2	2
Steamwheedle Sniper × 1	2
Carrion Grub × 2	3
Fungal Enchanter × 2	3
Flanking Strike × 2	4
Lifedrinker × 2	4
Wing Blast × 2	4
Antique Healbot × 1	5
Cobra Shot × 2	5
Rotten Applebaum × 1	5
Vilebrood Skitterer × 1	5
Maexxna × 1	6
Savannah Highmane × 2	6
King Krush × 1	9

Figure 5.1: The decks used during the experiments. From left to right: Default, Aggro, Midrange, Control.

H-MCTS	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
3-ply vs 2-ply	100	62.00% ±9.51%	7125	11400	5.5	5.5
4-ply vs 3-ply	100	62.00% ±9.51%	4955	6946	5.0	5.4
4-ply vs 2-ply	100	63.00% ±9.46%	4809	10539	4.9	5.3
LSI						
3-ply vs 2-ply	200	58.50% ±6.83%	3531	5013	1	1
4-ply vs 3-ply	100	68.00% ±9.14%	3005	4064	1	1
4-ply vs 2-ply	100	61.00% ±9.56%	2930	5385	1	1
N-MCTS						
3-ply vs 2-ply	400	47.25% ±4.89%	2682	3084	5.9	6.0
4-ply vs 3-ply	100	33.00% ±9.22%	1820	2158	5.6	5.8
4-ply vs 2-ply	100	34.00% ±9.28%	2285	3109	5.7	6.0

Table 5.1: Results of the Playout Length experiment.

5.1.2 Default settings

To experiment with the various settings that are available to each technique, some default settings are required. For the settings that are shared between techniques the following default values are used:

- Playout length of 3-ply
- MAST playout using ϵ-greedy selection ($\epsilon = 0.2$) (Powley *et al.*, 2013)
- Ensemble size of 1 determinisation

All three techniques also use the same evaluation function, as described in Section 4.5. H-MCTS uses UCT with a C constant set to $1/\sqrt{2}$ as its selection strategy and Minimum-T Expansion with T set to 50 as its expansion strategy. The policies π_0 and π_g used by N-MCTS have ϵ values of 0.75 and 0.0, respectively and are the same as set in Ontañón (2013). The Explore phase of LSI is assigned 25% of the iteration budget and the method used to estimate this iteration budget is AverageSampleEstimation with a σ setting of 1.5. This value was empirically determined to prevent LSI from violating the budget restriction.

For the experiments in the following subsections, each technique is limited to a computation budget of 5 seconds per turn. In the tables presenting the results of the experiments, $\overline{I}$ represents the average number of iterations per search and $\overline{D}$ represents the average depth of the search-tree (where applicable). These numbers are presented for each player, where $p1$ refers to player 1 and $p2$ refers to player 2. The win-rate column refers to the percentage of games won by player 1, with the upper and lower bounds noted next to them. These bounds are calculated with a confidence level of 95%. The number of games that were played for each match is included in the results because some matches required more games to reach a statistically significant result. The experiments in Subsections 5.1.3 through 5.1.10 are performed and analysed using the Self-Play framework presented in Heinz (2000).

5.1.3 Playout Length

This experiment is intended to determine an appropriate setting of the length of the playouts used by the search techniques. For each of the three techniques, a match was run between the default setting of a 3-ply long playout and settings of 2-ply and 4-ply long playouts. The results of this experiment are presented in Table 5.1. For H-MCTS, 4-ply long playouts perform significantly better than shorter playouts, although the number of iterations that can be performed is reduced. LSI shows similar results; longer playouts perform better at the cost of a reduced number of iterations. For N-MCTS however, 4-ply long playouts perform significantly worse than shorter playouts. There is no significant difference between 3-ply and 2-ply playouts, except for 2-ply reaching a higher number of iterations.

Based on these results, H-MCTS uses 4-ply long playouts for the experiments in Sections 5.2 and 5.3. LSI uses 4-ply long playouts and N-MCTS uses 2-ply long playouts for the experiment in Section 5.3.

H-MCTS	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
ϵ-Greedy vs Random	500	50.00% ±4.38%	8026	9053	6.5	6.6
UCB1 vs ϵ-Greedy	100	31.00% ±9.06%	8001	7069	5.9	5.4
UCB1 vs Random	100	19.00% ±7.69%	8692	9115	6.5	6.3
LSI						
ϵ-Greedy vs Random	100	38.00% ±9.51%	3275	4030	1	1
UCB1 vs ϵ-Greedy	100	21.00% ±7.98%	4960	4067	1	1
UCB1 vs Random	100	14.00% ±6.80%	5473	5064	1	1
N-MCTS						
ϵ-Greedy vs Random	200	62.50% ±6.71%	2425	2596	5.8	5.8
UCB1 vs ϵ-Greedy	300	53.33% ±5.65%	3035	2823	6.1	5.9
UCB1 vs Random	350	47.71% ±5.23%	2981	2936	6.1	5.9

Table 5.2: Results of the Type of Playout experiment.

5.1.4 Type of Playout

This experiment is intended to determine an appropriate setting for the type of playout used by the search techniques. For each of the three techniques, a match was run between the default setting of MAST playouts using ϵ-greedy selection, MAST playouts using UCB1 selection and random playouts performed by the RandomBot. The results for this experiment are presented in Table 5.2. For H-MCTS, UCB1-MAST performs significantly worse than both ϵ-greedy-MAST and Random playouts, but there is no significant difference in performance between those two, except for Random reaching a higher average number of iterations. For LSI, Random playouts perform significantly better than both UCB1-MAST and ϵ-greedy-MAST playouts. For N-MCTS, ϵ-greedy-MAST performs significantly better than Random, while UCB1-MAST shows no significant advantage over the other two strategies.

Based on these results, H-MCTS uses Random playouts for the experiments in Sections 5.2 and 5.3. LSI uses Random playouts and N-MCTS uses ϵ-greedy-MAST playouts for the experiment in Section 5.3.

5.1.5 Ensemble Size

This experiment is intended to determine an appropriate setting for the number of determinisations that make up the ensemble used by the search techniques. For each of the three techniques, a match was run between the default setting of 1 determinisation and settings of 2 and 5 determinisations. The results for this experiment are presented in Table 5.3. For H-MCTS, 1 performs significantly better than 2, but not significantly better than 5, while 5 does not have a significant advantage over 2. Using only 1 determinisation does generate more iterations and depth of the search tree. For LSI, 5 performs significantly better than 1, but not significantly better than 2, which does not perform significantly better than either other setting. For N-MCTS, while there is no significant difference in performance between 1 and 2, 5 does perform significantly better than both.

Based on these results, H-MCTS uses 1 determinisation for the experiments in Sections 5.2 and 5.3. Both LSI and N-MCTS use an ensemble of 5 determinisations for the experiment in Section 5.3.

5.1.6 H-MCTS C Constant

This experiment is intended to determine an appropriate setting for the C constant in UCT (see Equation 3.1). A match was run between the default setting of $1/\sqrt{2}$ and settings of 0.5 and 0.2. The results of this experiment are presented in Table 5.4. From these results it becomes clear that a setting of 0.2 performs significantly better than both other settings, while there is no significant difference between 0.5 and $1/\sqrt{2}$. Based on this, H-MCTS uses a setting of 0.2 for the C constant in UCT for the experiments in Sections 5.2 and 5.3.

H-MCTS	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
2 vs 1	250	42.40% ±6.13%	7532	8189	6.0	6.6
5 vs 2	250	53.60% ±6.18%	6849	7736	4.7	6.1
5 vs 1	250	45.60% ±6.17%	6985	8335	4.7	6.7
LSI						
2 vs 1	200	54.50% ±6.90%	4134	4181	1	1
5 vs 2	200	53.50% ±6.91%	4012	4301	1	1
5 vs 1	150	61.33% ±7.79%	3906	4259	1	1
N-MCTS						
2 vs 1	200	49.50% ±6.93%	2684	2731	5.7	5.9
5 vs 2	200	64.50% ±6.63%	2668	2776	5.4	5.7
5 vs 1	100	64.00% ±9.41%	2738	2910	5.4	5.9

Table 5.3: Results of the Ensemble Size experiment.

C constant	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
0.5 vs 0.2	200	41.00% ±6.82%	9256	10698	7.2	8.1
$1/\sqrt{2}$ vs 0.5	200	47.50% ±6.92%	8732	9279	6.8	7.2
$1/\sqrt{2}$ vs 0.2	200	42.00% ±6.84%	8339	9894	6.7	7.9

Table 5.4: Results of the H-MCTS C Constant experiment.

5.1.7 H-MCTS Expansion Threshold

This experiment is intended to determine an appropriate setting for T in the Minimum-T Expansion strategy. A match was run between the default setting of 50 and settings of 25 and 0. The results of this experiment are presented in Table 5.5. From these results it cannot be concluded that any setting performs significantly better than others, although it should be noted that a setting of 0 does reach a greater depth on average. Based on this, H-MCTS uses a setting of 0 for T for the experiments in Sections 5.2 and 5.3.

5.1.8 LSI Time Budget Estimation

This experiment is intended to determine the best method of estimating the number of iterations for LSI when a time-limited budget is used. A match was run between the default setting of AverageSampleEstimation (ASE) with a σ setting of 1.5 and the PreviousSearchEstimation (PSE) method with a σ setting of 1.0. The results of this experiment are presented in Table 5.6. In this table, $\overline{V}$ indicates the average number of budget limit violations per game, for each player. The σ setting for each method is noted next to its abbreviation, e.g. $ASE_{1.5}$ represents the ASE method with a σ setting of 1.5. As can be seen from the results, $PSE_{1.0}$ performs significantly better than $ASE_{1.5}$ and reaches more average iterations. However, this comes at the cost of nearly 5 budget limit violations per game, which is unacceptable. Therefore another two matches were run between the two methods, this time with equal σ settings of 1.3 and 1.5. From the results of those two matches it can be concluded that neither method performs better than the other does when the σ settings are equal, although ASE does attain more iterations on average. In order to bring the average number of budget limit violations per game down to an acceptable level, the σ setting should be set to 1.5. Based on these conclusions, LSI uses ASE with a σ setting of 1.5 for the experiment in Section 5.3.

5.1.9 LSI Exploration Budget

This experiment is intended to determine an appropriate setting for the percentage of the computation budget that is spent during the Explore phase of LSI. A match was run between the default setting of 25% and settings of 50% and 75%. The results of this experiment are presented in Table 5.7. From these results it can be concluded that 75% performs significantly worse against both other settings, although there is no significant difference between 50% or 25%. As a compromise between these settings, while still

Expansion Threshold	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
25 vs 0	300	50.33% ±5.66%	7481	8119	6.4	8.8
50 vs 25	300	47.33% ±5.65%	7587	7444	6.5	6.3
50 vs 0	350	52.57% ±5.23%	7506	7868	6.5	8.8

Table 5.5: Results of the H-MCTS Expansion Threshold experiment.

Estimation	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{V_{p1}}$	$\overline{V_{p2}}$
$ASE_{1.5}$ vs $PSE_{1.0}$	100	42.00% ±9.67%	3968	5518	0.3	4.8
$ASE_{1.3}$ vs $PSE_{1.3}$	200	50.00% ±6.93%	4783	4334	1.3	1.1
$ASE_{1.5}$ vs $PSE_{1.5}$	200	50.00% ±6.93%	3642	3257	0.5	0.3

Table 5.6: Results of the LSI Time Budget Estimation experiment.

leaving room for exploration of the search space, LSI uses a setting of 33% of the computation budget being spent in the Explore phase for the experiment in Section 5.3.

5.1.10 N-MCTS Policies

This experiment is intended to determine appropriate settings for the ϵ-greedy policies π_0 and π_g used by N-MCTS. The results for this experiment are presented in Table 5.8. For the policy π_0, a match was run between the default setting of 0.75 and settings of 0.5 and 0.25. From the results it can be concluded that 0.75 performs significantly better than both other settings. For the policy π_g, a match was run between the default setting of 0.0 and settings of 0.1 and 0.33. When it became apparent that there is no significant difference in performance between these settings, another match was run using higher settings for ϵ, in this case 0.75 and 0.5. From the results of these matches it can be concluded that a setting of 0.75 performs significantly better than 0.0. Based on these results, N-MCTS uses an ϵ-setting of 0.75 for both policies for the experiment in Section 5.3.

5.2 H-MCTS Dimensional Ordering

This experiment is intended to find a dimensional ordering for H-MCTS that improves its performance, in order to answer Research Question 3. A round-robin tournament consisting of 200-game matches was created to test the following orderings: Ascending Entropy, Descending Entropy, Descending Average Evaluation, Task Type, Descending Mana cost and the default ordering for the tasks as they are supplied by the SabberStone simulator. The order for Task Type is the following (in descending order): 'PLAY_CARD', 'HERO_POWER', 'MINION_ATTACK', 'HERO_ATTACK' and 'END_TURN'. The results for this experiment are presented in Table 5.9. This table contains the number of games won by the row-player against the column-player. Note that this experiment contains the first and only occurrence of a draw in all of the games played during this research. This illustrates how rare of an occurrence it is, but a possible one none the less. From these results we cannot conclude that any ordering is better than others with 95% confidence. However, if any ordering would have to be selected, the Descending Entropy ordering achieved the highest number of game wins in this experiment. The one-tailed hypothesis that its win-rate is greater than 50% gives a p-value of 0.0968. With this in mind, the H-MCTS technique uses the Descending Entropy ordering for the experiment in Section 5.3.

Exploration budget	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$
50% vs 25%	400	50.25% ±4.90%	4029	3978
75% vs 50%	100	38.00% ±9.51%	3898	4030
75% vs 25%	100	30.00% ±8.98%	3820	3894

Table 5.7: Results of the LSI Exploration Budget experiment.

π_0	# games	win-rate	$\overline{I_{p1}}$	$\overline{I_{p2}}$	$\overline{D_{p1}}$	$\overline{D_{p2}}$
0.50 vs 0.25	100	62.00% ±9.51%	2313	2719	8.2	12.5
0.75 vs 0.50	100	61.00% ±9.56%	2841	3291	5.8	8.6
0.75 vs 0.25	100	61.00% ±9.56%	2681	3526	5.8	12.9
π_g						
0.10 vs 0.00	300	49.67% ±5.66%	2737	2738	5.7	5.8
0.33 vs 0.10	300	50.33% ±5.66%	2703	2721	5.4	5.7
0.33 vs 0.00	300	50.67% ±5.66%	2713	2731	5.4	5.9
0.50 vs 0.00	200	55.50% ±6.89%	2735	2774	5.2	5.9
0.75 vs 0.50	200	54.50% ±6.90%	2587	2605	5.0	5.2
0.75 vs 0.00	300	56.67% ±5.61%	2647	2707	5.0	5.9

Table 5.8: Results of the N-MCTS Policies experiment.

5.3 Round-Robin Tournament

Now that the appropriate settings for each of the techniques have been determined, a round-robin tournament is set up to determine the relative play strength of the agents. A computation budget of 5 seconds per turn was set for the search techniques. The agents participating in the tournament, with their relevant settings, are the following:

- HeuristicAgent
- H-MCTS, 1 Determinisation, Random 4-ply playouts, $T = 0$, $C = 0.2$, Descending Entropy ordering
- LSI, 5 Determinisations, Random 4-ply playouts, Explore budget = 33%, $ASE_{1.5}$
- N-MCTS, 5 Determinisations, ϵ-Greedy 2-ply playouts, $\pi_0 = 0.75$, $\pi_g = 0.75$
- RandomAgent

The results for this experiment are presented in Table 5.10. This table contains the percentage of games won by the row-player against the column-player. Only 100 games were run for the matches involving the RandomAgent, as it quickly became apparent that all other agents were outmatching it. From these results it can be concluded that the HeuristicAgent is the strongest playing agent. Between the three search techniques there is no technique that performs significantly better than the other two. The arguably mediocre performance of the search techniques compared to that of the HeuristicAgent could be due to a number of factors, such as insufficient computing budget or a sub-optimal evaluation function. After reviewing the results, it was decided to replace the evaluation function described in Section 4.5 with the evaluation function used by the HeuristicAgent and adapt it for use by the search techniques. The next subsection discusses the experiment with the new evaluation function.

5.3.1 HeuristicAgent Evaluation Function

This experiment is intended to compare the performance of the search techniques using the heuristic evaluation function to that of the HeuristicAgent. It was decided to remove the RandomAgent from the list of participating agents, due to its performance in the previous experiment. The settings for all agents are identical to the previous experiment, with the exception of the evaluation function. The results for this experiment are presented in Table 5.11. This table contains the percentage of games won by the row-player against the column-player. From these results it can be concluded that the MCTS

Ordering	Entropy ↑	Entropy ↓	Evaluation ↓	Task Type	Mana ↓	Default	# games	win-rate
Entropy ↑	-	94	108	92	103	95	1000	49.20% ±3.10%
Entropy ↓	106	-	105	93	114	111	1000	52.90% ±3.09%
Evaluation ↓	92	95	-	111	100	106	1000	50.40% ±3.10%
Task Type	108	107	89	-	87	98	1000	48.90% ±3.10%
Mana ↓	97	86	100	113	-	83.5	1000	47.95% ±3.10%
Default	105	89	94	102	116.5	-	1000	50.65% ±3.10%

Table 5.9: Results for the H-MCTS Dimensional Ordering experiment.

Agent	HeuristicAgent	H-MCTS	LSI	N-MCTS	RandomAgent	# games	win-rate
HeuristicAgent	-	66%	75%	78%	98%	700	76.57% ±3.14%
H-MCTS	34%	-	52.5%	48.5%	97%	700	52.43% ±3.70%
LSI	25%	47.5%	-	45.5%	96%	700	47.43% ±3.70%
N-MCTS	22%	51.5%	54.5%	-	98%	700	50.57% ±3.70%
RandomAgent	2%	3%	4%	2%	-	400	2.75% ±1.60%

Table 5.10: Results of the Round-Robin Tournament experiment.

techniques gain the most from this new evaluation function, as their performance is now comparable to that of the HeuristicAgent. This increase in performance has to come from somewhere, and in this case it is LSI that decreases its performance against both MCTS techniques.

As discussed in the previous experiment, another factor that influences the performance of the MCTS techniques is the computing budget. The next experiment will therefore test the performance of the MCTS techniques with varying computing budgets.

5.3.2 Computation Budget

This experiment is intended to compare the performance of the MCTS techniques when the computation budget is changed from the default budget of 5 seconds. For this experiment, a round-robin tournament was created for the following agents: HeuristicAgent, H-MCTS and N-MCTS. Due to its poor performance in the previous experiment, LSI is not included. The settings for the agents are the same as in experiment 5.3.1. One match was run with a computation budget of 1 second and another match was run with a computation budget of 10 seconds. Table 5.12 presents the results of these matches. This table shows that H-MCTS improves its performance against HeuristicAgent when the computation budget increases from 1 second to 10 seconds (32.8% versus 44.8%, p=0.003), but the increase between 5 seconds and 10 seconds is not significant (47.5% versus 44.8%, p=0.284). N-MCTS does not show a significant improvement in performance between 1 and 10 seconds (48% versus 53.2%, p=0.123), although the improvement in performance between 5 and 10 seconds does show a significant difference (43.5% versus 53.2%, p=0.020). This leads to the conclusion that both techniques perform better when the computation budget is increased.

When the computation budget is set to 10 seconds, none of the three agents perform significantly better than either other agent over 250 games. From this experiment it can be concluded that both MCTS techniques are capable of performing at a level of play that is comparable to the second place finisher in the Hearthstone AI Competition at IEEE CIG 2018. When the computation budget is further increased to 60 seconds, H-MCTS wins 53 games out of 100 against the HeuristicAgent and N-MCTS wins 47 games out of 100. These sample sizes are too small to show any significant impact of such a large computation budget on the performance of the search techniques, but due to the extensive run-time required for these experiments it was not viable to run more than 100 games.

Agent	HeuristicAgent	H-MCTS	LSI	N-MCTS	# games	win-rate
HeuristicAgent	-	52.5%	72.5%	56.5%	600	60.50% ±3.91%
H-MCTS	47.5%	-	66.5%	50%	600	54.67% ±3.98%
LSI	27.5%	33.5%	-	25.5%	600	28.83% ±3.62%
N-MCTS	43.5%	50%	74.5%	-	600	56.00% ±3.97%

Table 5.11: Results of the HeuristicAgent Evaluation Function experiment.

1 second	HeuristicAgent	H-MCTS	N-MCTS	# games	win-rate
HeuristicAgent	-	67.2%	52%	500	59.60% ±4.30%
H-MCTS	32.8%	-	43.2%	500	38.00% ±4.25%
N-MCTS	48%	56.8%	-	500	52.40% ±4.38%
5 seconds	HeuristicAgent	H-MCTS	N-MCTS	# games	win-rate
HeuristicAgent	-	52.5%	56.5%	400	54.50% ±4.88%
H-MCTS	47.5%	-	50%	400	48.75% ±4.90%
N-MCTS	43.5%	50%	-	400	46.75% ±4.89%
10 seconds	HeuristicAgent	H-MCTS	N-MCTS	# games	win-rate
HeuristicAgent	-	55.2%	46.8%	500	51.00% ±4.38%
H-MCTS	44.8%	-	46.4%	500	45.60% ±4.37%
N-MCTS	53.2%	53.6%	-	500	53.40% ±4.37%

Table 5.12: Results of the Computation Budget experiment.

Chapter 6

Conclusion

In the final chapter of this thesis, the problem statement and research questions that were presented in the first chapter are reiterated and reviewed. After this, several topics and ideas for future research are discussed.

6.1 Research Questions

1. *How do we represent Hearthstone as a CMAB Problem?*

Section 2.6 showed that Hearthstone can be modelled as a Combinatorial Multi-Armed Bandit Problem by viewing the game actions that are available to a player as the dimensions of the problem. The arms of the problem represent the choice of whether or not to play those game actions. The super-arms are then comprised of an entire turn's worth of game actions.

2. *How do we handle imperfect information in the CMAB Problem?*

Subsection 4.1.1 presents the approach of creating a determinisation of Hearthstone's game state as a way to deal with the hidden information that is the identity and order of both player's decks and the cards in the opponent's hand. Using multiple determinisations in an ensemble is shown to improve performance for LSI and N-MCTS.

3. *Can we find dimensional orderings for Hierarchical Expansion that improve performance?*

Subsection 4.2.2 presents two dimensional orderings that are specific to the domain of Hearthstone (Mana Cost and Task Type) and two that are based on metrics created during the search process itself (Average Evaluation and Entropy). The performance of these orderings is compared to the default ordering provided by the simulator. Although the Descending Entropy ordering attains the highest number of game wins, the results for this experiment are inconclusive at the 95% confidence level.

4. *How does Hierarchical Expansion perform compared to Naïve Monte Carlo and Linear Side Information in the domain of Hearthstone?*

The performance of the different CMAB techniques is compared in three experiments. The first experiment shows that each of the three techniques outperform an agent that executes actions randomly, although they do not perform at the level of the agent that represents strong play. In the second experiment, the evaluation function that the techniques share is replaced by a stronger version. This significantly increases the performance of the H-MCTS agent and the N-MCTS agent, but the LSI agent does not show a similar increase. In the third experiment, an increase in the computation budget shows increased performance for the H-MCTS agent and also for the N-MCTS agent. At a computation budget of 10 seconds, both agents show comparable performance to that of the agent representing strong play. From these experiments it can be concluded that the Hierarchical Expansion technique performs better than Linear Side Information, and performs as good as Naïve Monte Carlo, in the domain of Hearthstone.

6.2 Problem Statement

- *Can a card game such as Hearthstone be approached as a Combinatorial Multi-Armed Bandit Problem?*

This thesis presents three techniques that are designed for the CMAB problem and adjusts them to fit the domain of Hearthstone. These adjustments include general ones, e.g. using an ensemble of determinisations of the game state to handle imperfect information and using the Move-Average Sampling Technique to improve the quality of playouts, but also include technique-specific adjustments.

The experiment discussed in Subsection 5.3.2 shows that an agent based on the Hierarchical Expansion technique, as well as an agent based on the Naïve Monte Carlo technique can perform at a level comparable to that of an agent that finished second at the Hearthstone AI Competition held at the IEEE Computational Intelligence and Games conference in 2018.

6.3 Future Research

First of all it should be noted that when dealing with games that have a high level of variance, the difference between strategies can sometimes be quite small. For instance, when the actual advantage of one strategy or technique is only 3%, i.e. it will win 53 out of 100 games on average, the number of games that should be played to achieve a statistically significant result with 95% confidence is 1064. When the computing budget is set to 10 seconds, it would take approximately 59 hours to run these games (2 players taking an average of 10 turns each over 1064 games). It would therefore be useful to investigate different and possibly more efficient ways of comparing and analysing experimental results for Hearthstone and similar games.

While N-MCTS did show a performance increase by using ϵ-greedy MAST playouts, this was not the case for both H-MCTS and LSI. A more general version of this same technique exists, called N-gram-Average Sampling Technique (NAST), which looks at the value of the Nth action in the context of the N-1 actions that preceded it. Due to the added value of certain action sequences in Hearthstone, NAST could be a better fit to the problem domain than MAST.

Another generalisation of one of the techniques used in this research could be to determinise the game state with types of cards rather than specific cards. For example, instead of placing card x_1 as the second card in the opponent's deck, a card of type A could be placed there. Players often think about what would happen if their opponent would draw a card of a specific type, such as a minion or a spell that interacts with one of the player's minions. This approach is similar to that of bucketing chance nodes as presented by Zhang and Buro (2017), but applied to cards in decks instead of random events.

Another direction for future research is to combine the power of Deep Neural Networks with the MCTS techniques, as such approaches have been successful in other complex games such as Go (Silver *et al.*, 2016) and StarCraft II$^{®}$ (DeepMind, 2018). Another complex game is Magic the Gathering (MtG), which is often described as a combination of Poker and Chess, but while both of those games have seen significant advances in scientific research, MtG has seen barely any research. The complexity of these cards games might make them the next subject for AI-research to find significant advances in.

References

Björnsson, Y. and Finnsson, H. (2009). CadiaPlayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 4–15. [13, 21]

Blizzard Entertainment (2017). 70 Million Players! `https://playhearthstone.com/en-us/blog/20720847`. [5]

Bossland GmbH (2014). HearthBuddy. `https://www.hearthbuddy.com/thebot/`. [10]

Chaslot, G. M. J-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M., and Bouzy, B. (2008). Progressive Strategies For Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [14, 15]

Chen, J., Liu, X., and Lyu, S. (2013a). Boosting with Side Information. *11th Asian Conference on Computer Vision (ACCV 2012)*, pp. 563–577, Springer, Berlin, Heidelberg. [17]

Chen, Wei, Wang, Yajun, and Yuan, Yang (2013b). Combinatorial Multi-Armed Bandit: General Framework and Applications. *30th International Conference on Machine Learning* (eds. S. Dasgupta and D. McAllester), Vol. 28 of *Proceedings of Machine Learning Research*, pp. 151–159, PMLR, Atlanta, Georgia, USA. [2]

Coulom, R. (2007). Monte-Carlo Tree Search in Crazy Stone. *12th Game Programming Workshop (GPW-07)*, Information Processing Society of Japan, Hakone Seminor House, Japan. [14]

Cowling, P. I., Powley, E. J., and Whitehouse, D. (2012a). Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 120–143. [20]

Cowling, P. I., Ward, C. D., and Powley, E. J. (2012b). Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 4, pp. 241–257. [20]

Curse (2015). Patch 3.0.0.9786. `https://hearthstone.gamepedia.com/Patch_3.0.0.9786`. [9]

Curse (2017). Advanced Rulebook. `https://hearthstone.gamepedia.com/Advanced_rulebook`. [8]

DeepMind (2018). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`. [38]

Dockhorn, Alexander and Mostaghim, Sanaz (2018). Hearthstone AI Competition. `https://dockhorn.antares.uberspace.de/wordpress/`. [10]

Doux, B., Gautrais, C., and Negrevergne, B. (2016). Detecting strategic moves in Hearthstone matches. *Machine Learning and Data Mining for Sports Analytics Workshop (ECML/PKDD)*, Riva del Garda, Italy. [3]

Fang, H. R., Hsu, T. S., and Hsu, S. C. (2003). Indefinite Sequence of Moves in Chinese Chess Endgames. *Computers and Games, Third International Conference (CG 2002)* (eds. M. Müller J. Schaeffer and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 264–279, Springer, Berlin, Heidelberg. [1]

Gaina, R. D., Couëtoux, A., Soemers, D. J. N. J., Winands, M. H. M., Vodopivec, T., Kirchgeßner, F., Liu, J., Lucas, S. M., and Perez-Liebana, D. (2018). The 2016 two-player GVGAI competition. *IEEE Transactions on Games*, Vol. 10, No. 2, pp. 209–220. [13]

Goodrich, M. T. and Tamassia, R. (2006). *Algorithm Design: Foundation, Analysis and Internet Examples.* John Wiley & Sons. [22]

HackHearthstone (2018). SmartBot. `https://hackhearthstone.com/`. [10]

HearthSim (2018). HearthSim. `https://hearthsim.info`. [10]

Heinz, E. A. (2000). Self-play Experiments in Computer Chess Revisited. *Advances in Computer Games 9* (eds. H. J. van den Herik and B. Monien), pp. 73–91, Universiteit Maastricht. [29]

Karnin, Z., Koren, T., and Somekh, O. (2013). Almost Optimal Exploration in Multi-Armed Bandits. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (eds. S. Dasgupta and D. McAllester), Vol. 28, pp. 1238–1246, JMLR Workshop and Conference Proceedings. [17]

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [2]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *17th European Conference on Machine Learning (ECML-06)*, Vol. 4212 of *LNAI*, pp. 282–293, Springer, Berlin, Heidelberg. [14]

Metropolis, N. (1987). The Beginning of the Monte Carlo Method. *Los Alamos Science Special Issue*, Vol. 15, pp. 125–130. [13]

Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. H. (2017). DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker. Computing Research Repository. [1]

Nijssen, J. A. M. and Winands, M. H. M. (2012). Monte Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 4, pp. 282–294. [20]

Ontañón, S. (2013). The Combinatorial Multi-Armed Bandit Problem and its Application to Real-Time Strategy Games. *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2013)*, pp. 58–64, Boston, USA. [2, 15, 29]

Powley, E. J., Whitehouse, D., and Cowling, P. I. (2013). Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pp. 1–8. ISSN 2325–4289. [21, 29]

Roelofs, G-J. (2015). Action Space Representation in Combinatorial Multi-Armed Bandits. M.Sc. thesis, Maastricht University, Department of Knowledge Engineering, Maastricht, The Netherlands. [2, 15]

Rush4x (2017). HearthRanger. `http://www.hearthranger.com`. [10]

Santos, A., Santos, P. A., and Melo, F. S. (2017). Monte Carlo Tree Search Experiments in Hearthstone. *Conference on Computational Intelligence and Games (CIG 2017)*, pp. 272–279, IEEE. [3]

Schaeffer, J. (2001). A Gamut of Games. *AI Magazine*, Vol. 22, No. 3, pp. 29–46. [1]

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, Vol. 27, No. 3, pp. 379–423. [24]

Shleyfman, A., Komenda, A., and Domshlak, C. (2014). On Combinatorial Actions and CMABs with Linear Side Information. *21st European Conference on Artificial Intelligence (ECAI 2014)*, pp. 825–830, Prague, Czech Republic. [2, 17]

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, Vol. 529, pp. 484–489. [1, 2, 38]

TempoStorm (2016). The Basics of Proper Sequencing. `https://tempostorm.com/articles/the-basics-of-proper-sequencing`. [3]

van den Herik, H. J., Uiterwijk, J. W. H. M., and van Rijswijck, J. (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, No. 1, pp. 277–311. [1]

Ward, C. D. and Cowling, P. I. (2009). Monte Carlo Search Applied to Card Selection in Magic: The Gathering. *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 9–16. [19]

Zhang, S. and Buro, M. (2017). Improving Hearthstone AI by Learning High-Level Rollout Policies and Bucketing Chance Node Events. *Conference on Computational Intelligence and Games (CIG 2017)*, pp. 309–316, IEEE. [38]