NTNU
Norwegian University of
Science and Technology

Programming a Hearthstone agent using Monte Carlo Tree Search

Markus Heikki Andersson
Håkon Helgesen
Hesselberg

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This thesis describes the effort of adapting Monte Carlo Tree Search (MCTS) to the game of Hearthstone, a card game with hidden information and stochastic elements. The focus is on discovering the suitability of MCTS for this environment, as well as which domain-specific adaptations are needed.

An MCTS agent is developed for a Hearthstone simulator, which is used to conduct experiments to measure the agent's performance both against human and computer players. The implementation includes determinizations to work around hidden information, and introduces action chains to handle multiple actions within a turn. The results are analyzed and possible future directions of research are proposed.

Sammendrag

Denne avhandlingen beskriver arbeidet ved å tilpasse Monte Carlo Tree Search til spillet Hearthstone, et kortspill med skjult informasjon og stokastiske elementer. Fokuset settes på å oppdage hvor passende MCTS er til dette miljøet, i tillegg til hvilke domenespesifikke endringer som er nødvendige.

En MCTS agent blir utviklet i en Hearthstone simulator, som brukes til utføre eksperimenter for å måle agentens ytelse mot både mennesker og datamaskinspillere. Implementasjonen inkluderer determiniseringer for jobbe rundt skjult informasjon, og introduserer handlingskjeder for å håndtere flere handlinger innenfor en tur. Resultatene analyseres og mulige retninger for fremtidig forskning blir foreslått.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter is inspired by the work done in the specialization project [Andersson and Hesselberg (2015)].

1.1 Background and Motivation

Video games are one of the quickest growing types of entertainment and is a popular passtime across the world, representing a billion-dollar industry just in the US [Newzoo (2014)].

Games will often include time constraints on a player's actions. Creating an AI to play these games therefore becomes a challenge, as they must be able to find solutions to problems quickly. This is why games are ideal to test applications of real-time AI research [Buro and Furtak (2003)].

This is not only true now, but also from the beginning of the computer era; many computer science pioneers have been using games like chess, checkers and others to research algorithms. Examples of such luminaries can be found by looking at Alan Turing, John von Neumann, Claude Shannon, Herbert Simon, Alan Newell, John McCarthy, Arthur Samuel, Donald Knuth, Donald Michie, and Ken Thompson [Billings (1995)].

There are a number of difficult problems represented in board games, card games, other mathematical games and their digital representations, and there are many reasons why their study is desirable. Usually they have all or some of these properties:

- Well-defined rules and concise logistics – As games are usually well-defined it can be relatively simple to create a complete player, which makes it possible to spend more time and effort on what is actually the topic of scientific interest.

- Complex strategies – Some of the hardest problems known in computational complexity and theoretical computer science can be found in games.
- Specific and clear goals – Games will often have an unambiguous definition of success, so that efforts can be focused on achieving that goal.
- Measurable results – By measuring either degree of success in playing against other opponents, or in solutions for related sub-tasks, it is possible to see how well an AI is performing.

Since a game can be looked at as a simplified situation from the real world, creating strong computer players for these games might have potential when applied to real world problems [Laird and VanLent (2001)].

There are many games where computer players will either cheat by using access to the game engine to obtain additional information, resources or units - or they won't be a match for an expert level human player. However, players usually enjoy games more when they are presented with a challenge appropriate to their skill [Malone (1980); Whitehouse (2014)]. Player enjoyment is easier to achieve when they perceive the game is played on equal terms, instead of fighting an inferior computer opponent who is receiving an unfair advantage.

Among the hardest problems known in theoretical computer science is the class of games with stochastic, imperfect information games with partial observability. This class includes a lot of problems which are computationally undecidable, but easy to express [Shi and Littman (2001); Gilpin and Sandholm (2006)].

A game which fits the above description well is Hearthstone: Heroes of Warcraft. It is a digital strategic card game developed by Blizzard Entertainment. Two players compete against each other with self-made decks, using cards specifically created for Hearthstone. The game is easy to grasp, but complexity quickly arises because of the interaction between cards – each card has potential to subtly change the rules of the game. It is because of this complexity that Hearthstone is an interesting area for AI research, especially for search in imperfect information domains and general game playing.

1.2 Research Questions

In the specialization project Hearthstone was analyzed and compared to other games to understand which techniques could be used to create a strong computer player [Andersson and Hesselberg (2015)].

It was discovered that the key points to consider when when creating an AI for Hearthstone were the large game tree, the difficulty of node evaluation and the way the game is constantly evolving. Monte Carlo Tree Search proved to be a viable solution. It can handle large game trees by using random sampling, can be run without a heuristic function and as such can deal with new elements being introduced.

This project is focused on further examining Monte Carlo Tree Search, implementing it to create a Hearthstone computer player and analyze its playing strength. As such the high level research questions are:

1. Can Monte Carlo Tree Search be implemented to create a strong Hearthstone computer player?
2. Which parameters and implementation choices are important for the agent's performance?

1.3 Research Method

The research method for this project is to acquire a deep understanding of the Monte Carlo Tree Search algorithm and implement it to create a strong Hearthstone computer player. The work can be divided into three parts

- Research and understand the underlying principles and theories behind MCTS.
- Implement MCTS specific to Hearthstone.
- Perform experiments to figure out which parameter values give the best results.

1.4 Thesis Structure

Chapter two describes Hearthstone in detail and describes it formally.

Chapter three explains the theories leading up to Monte Carlo Tree Search together with related research, and describes how MCTS works.

Chapter four describes the implementation of MCTS, which choices have been made and why, as well as the simulator being used.

Chapter five describes the agents and parameters used and sets up experiments to test the MCTS algorithm.

Chapter six describes the results of the experiments shown in the previous chapter and analyses them.

Chapter seven concludes the thesis.

Chapter eight describes future work.

Chapter 2

What is Hearthstone

This chapter is inspired by the work done in the specialization project [Andersson and Hesselberg (2015)].

2.1 Hearthstone Statistics

Developers	Blizzard entertainment
Publishers	Blizzard entertainment
Series	Warcraft
Engine	Unity
Platforms	Windows OS X iPad iPhone Android
Release dates	**Windows, OS X** March 11, 2014
	iPad April 16, 2014
	Android tablets December 15, 2014
	iOS, Android smartphones April 14, 2015
Genre	Collectible card game
Modes	Multiplayer, single-player

Hearthstone: Heroes of Warcraft is a collectible card game, which is online and free to play. It was developen and published by Blizzard Entertainment (2015). The game was announced first at the Penny Arcade Expo in March 2013 [Engadget (2013)], and released on 11th of March, 2014 [IGN (2014)].

New content is regularly released by Blizzard Entertainment in the form of new card sets, gameplay and game modes, which is usually an expansion pack or an adventure focusing on single-player.

It was reported by GameSpot in November 2015 that 40 million Hearthstone accounts had been created [GameSpot (2015)].

The game has gotten generally favorable reviews, receiving a score of 87.57% on GameRankings [GameRankings (2014)] as well as 88% on Metacritic [MetaCritic (2014)]

which is based on reviews made by several major video game critics. Later content releases have also received good ratings [GameInformer (2014a); PCGamer (2014); GameInformer (2014b); Eurogamer (2015)].

Hearthstone has won several awards:

- Game Awards (2014) - Best mobile/handheld game.
- GameSpot (2014) - Mobile game of the year.
- GameTrailers (2014) - Best overall game and multiplayer game of the year.
- Academy of Interactive Arts and Sciences (2014) - Mobile game of the year and strategy/simulation game of the year.
- BAFTA Video Games Award (2014) - Best multiplayer game.

2.2 Beginner's Guide to Hearthstone

2.2.1 Gameplay

Adapted from Hearthstone Gamepedia (2015d).

Every battle in Hearthstone is played as a one-on-one battle between two opponents. The gameplay is turn-based, with each player alternating playing cards to cast spells, summon minions to battle on their behalf or equip weapons to be able to attack.

The players are represented by their chosen 'hero', with different heroes being important characters from Warcraft lore. The different heroes are associated with a particular class, which determines what special cards and unique hero powers they have available to them.

Figure 2.1: Hearthstone hero Jaina Proudmoore

A hero has 30 'health', which is a system of points used to reflecting the remaining survivability of a character. A heroes health can be seen on the blood drop on the character portrait, like with Figure 2.1 showing Jaina Proudmoore, the hero representing the mage class. Once a hero's health is reduced to zero, the controlling player loses the game. The object of a Hearthstone game is simple, although often difficult to achieve – Get the enemy hero's health to zero before your opponent can do the same to your hero.

With the beginning of every turn, a player draws a card from their deck, which is a collection of 30 cards assembled before the match begins. Players are offered to play with pre-assembles 'basic' decks, or make their own. Neutral cards are a large part of the available cards, and can be used by any class. However, a substantial portion is limited to specific classes, which is what gives each hero their own strengths and unique abilities.

In each player's turn that player may attempt to use any of their cards, their hero power, attack with their minions or use their hero to directly attack if they have equipped a weapon. Most actions, however, require a player to spend mana crystals, and is often the limiting resource which forces players to plan out each of their moves strategically. At the beginning of the game a player has 1 mana crystal, and receives an additional crystal each turn up to a maximum of 10. Any mana crystal used is regenerated at the beginning of that players turn (all their crystals become filled). Any unspent mana does not carry over to the next turn, and is wasted if not used. Since players get a larger and larger mana pool, later rounds see increasingly expensive cards, which opens the game up to more impressive moves and powerful abilities.

Hearthstone has multiple strategic elements a player has to master before being able to compete at higher levels of play. Positioning and control of minions, assigning correct strategic importance to various targets, utilizing complex card synergies and interactions, as well as working with the unpredictable random elements and randomly selected cards drawn each round, are some of the parts which combined make Hearthstone an intricate game, where it is not always obvious which play is the best.

2.2.2 Battlefield

The game board, called the battlefield, is where the game takes place. An example of a battlefield is shown in Figure 2.2. It contains various UI elements like the player's hands, decks and mana crystals, as well as the heroes and minions.

There are several different battlefields with their own design and interactive elements, but these do not affect game play in anyway. A battlefield is chosen at random.

Your own hero is shown near the bottom of the battlefield, labeled in Figure 2.2 as *Player*. The opponent's hero is displayed on the opposite side, near the top, labeled *opponent*. To the right of your hero is the hero power, and below and to the right of that are the mana crystals. Below your hero are your cards. All of this is mirrored on your opponent's side. A history of the last few actions are shown to the left in the history panel, while each

Figure 2.2: Hearthstone battlefield

player's deck is found all the way to the right. Both player's minions are found in the center of the battlefield, on opposing sides of the line in the middle.

Players always view themselves as placed at the bottom of the battlefield, while their opponent is shown on the top. The battlefield is in effect an illusion, which allows each player to experience a standard view of the game each time they are playing. The players never find themselves playing on the opposite side of the screen. Minion placement is also preserved between players' screens; if a minion is placed on the left of the player's screen, it will be similarly placed to the left of the opponent's screen, despite an apparent rotation of the perspective.

2.2.3 Order of Play

The starting player is decided by a coin toss at the beginning of each game. Players are then shown cards randomly drawn from their own decks (three cards for the player who is going first, four cards for the player going second), which they can then choose to keep or replace individually. This is called a mulligan, and is shown in Figure 2.3. New cards are randomly drawn from the deck as replacements, and then the discarded cards are shuffled back into the deck. In addition to getting one more card, the player going second receives the coin itself as a special card, which can be used to grant the player an extra mana crystal until the end of the turn. This is to offset the disadvantage of going second.

At the beginning of their turn, the players have their mana crystals refilled, and then draw one card from their deck and add it to their hand. Players can have a maximum of 10 mana

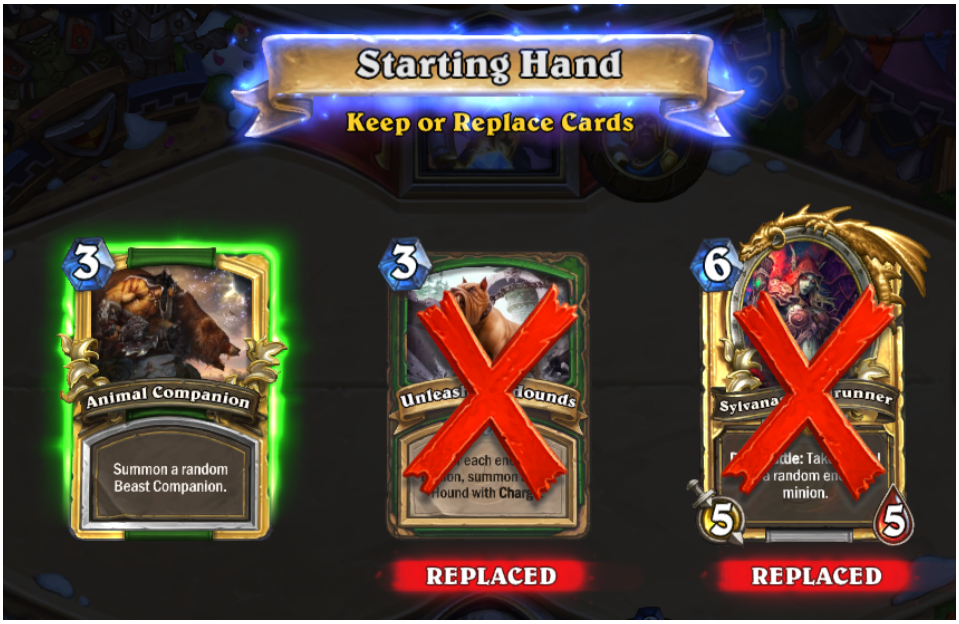


Figure 2.3: Mulligan – Replacing starting cards

crystals, as well as a maximum of 10 cards in their hand. Drawing an 11th card will cause the card to be revealed to both players, and then destroyed.

Provided you have enough mana, you can play any of your cards during your turn, using them to summon minions, casting spells or equip weapons. If you play a card, the amount of mana shown in the top-left corner of the card is depleted from your mana crystals.

You can command any viable minions to attack, use your hero's hero power, or use your hero to attack directly if your hero has a weapon equipped, or gotten an attack value another way.

A player's turn ends when they run out of time, or when they click on the End Turn button seen on the right side of the battlefield. A turn lasts a maximum of 75 seconds, with some additional time added to complete animations.

If players draw from their deck when there are no more cards remaining, they will take fatigue damage. Fatigue initially deals only one damage to the player, but the damage increases by one each time. Players can mouse over their own or their opponents deck to see how many cards are remaining. It's also possible to mouse over cards or hero powers to get a detailed description, as well as to view any enhancements on the cards.

2.2.4 Concluding the Game

A match ends when one of the following events happen:

- Either heroes' health reaches zero, or is destroyed. The player remaining is the winner.
- If a player concedes or leaves the game, the other player wins.
- If both heroes' are destroyed or get zero health at the same time, the game is a draw, although they will both see the screen showing defeat.
- When the turn limit of 90 turns is reached, the game ends in a draw. As before, both players see the defeat screen.

2.2.5 Additional Information

This guide is a short introduction intended only to get a new player familiar with some of the basic concepts of the game. A continuation of the guide is included in the appendix, which further explains cards, decks, minions, spells and weapons in more detail.

2.3 Hearthstone Game Description

2.3.1 Formal Game Specification

There is currently no official rulebook for Hearthstone which explains the underlying mechanics of the game, although a group of players have worked together to create a detailed, unofficial rulebook[Hearthstone Gamepedia (2015a)]. In addition to this there is no formal definition of the game as a scientific problem. This paper will define Hearthstone with the following properties, as used in the specialization project [Andersson and Hesselberg (2015)]:

- **Imperfect information** – There are various forms of uncertainty, and players have access to different types of information. This makes it necessary to both use and deal with deception, and to plan without being sure of your opponent's capabilities.
- **Stochastic outcomes** – There are significant amounts of chance in Hearthstone at most stages of the game. This creates uncertainty and uncontrollable outcomes, and it becomes difficult to make accurate assessments of performance due to the high degree of variance.
- **Partially observable** – A player does not see the opponent's hand or deck except when cards are played or otherwise revealed due to game mechanics. In addition, no cards are revealed at the end of the game. This makes it more difficult to learn of an opponent's strategy over the course of many games, as you won't always know what cards they chose not to play.

- **Scaling complexity** – Hearthstone is a card game with a large number of different cards, and the game's complexity can scale significantly based on which cards are included in the simulations. In addition, new cards are continuously being released. This makes it possible to test algorithmic performance at many levels.
- **Zero-sum** – The game is strictly competitive two-player, which means that one player's gain or loss is exactly balanced by the loss or gain of the other player.
- **Turn-based** – Game flow is partitioned into well-defined and visible parts. Hearthstone turns are sequential, with players alternating between whose turn it is until the game results in a victory.
- **Non-reactive** – Players are not allowed to act on their opponents turn.
- **Timed** – A round in Hearthstone lasts for 75 seconds, with additional time added to allow card animations to be completed. If a player runs out of time, their turn is forcefully ended and passed to their opponent.
- **Finite** – Hearthstone has a limited number of turns and cards, coming to a draw after 90 turns, although this limit is rarely reached.

2.3.2 The Complexity of Hearthstone

Hearthstone is not a card game which uses a standard deck of 52 cards like Poker or Bridge. Instead, Hearthstone uses cards made specifically for the game. As many of the cards change the rules of the game in different ways, and this again creates varied interactions between them, Hearthstone can quickly become complex.

Decks containing 30 cards are built by the players, and they choose from a pool of almost one thousand cards. In addition, there are regular releases of new expansions and adventures to keep the game fresh. Because of this, there is an enormous amount of possible interactions, which means that developing a competent AI player for Hearthstone can in some ways be looked at as creating an AI for general game playing. There can be great differences in each game depending on which decks are played.

The complexity increases for each card added, since each new card must be considered in relation to all other cards. As such, creating a rule-based AI can be difficult since it has to be updated after every new release, especially since new releases can introduce entirely new concepts to the game. This maintenance of a rule-based AI would grow exponentially more difficult. Therefore, in order to deal with the increasing complexity of Hearthstone, an algorithm that can deal with general game playing is preferred.

2.3.3 Estimate of Possible Moves

An estimate of the number of Hearthstone moves is found by looking at Oyachai (2014a).

A simple case to look at is to assume that each player has no cards in their hands, and that the board has **N** friendly minions and **M** enemy minions. The number of different ways the minions can attack is dependent on the *Attack* and *Health* values of the minions.

In the worst case scenario the player is not able to kill a single enemy minion, and each of the **N** minions have **M + 1** targets, including the enemy hero. As the orders the minion attack in matters, the total number of moves is the number of permutations multiplied by the number of different attacks.

The formula for the total number of moves **V** is

$$V = (M + 1)^N + N!$$

This is a number which can grow quickly. A board with seven minions on each side (**N** = 7, **M** = 7) has 10,569,646,080 permutations, which is around 10^{10}.

The table below contains the possible different moves with different **N** and **M**:

N	M							
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	1	2	3	4	5	6	7	8
2	2	8	18	32	50	72	98	128
3	6	48	162	384	750	1296	2058	3072
4	24	384	1944	6144	1.50E+04	3.11E+04	5.76E+04	9.83E+04
5	120	3840	2.92E+04	1.23E+05	3.75E+05	9.33E+05	2.02E+06	3.93E+06
6	720	4.61E+04	5.25E+05	2.95E+06	1.13E+07	3.36E+07	8.47E+07	1.89E+08
7	5040	6.45E+05	1.10E+07	8.26E+07	3.94E+08	1.41E+09	4.15E+09	1.06E+10

However, the above calculations are done when assuming no cards in each player's hand. By looking at the worst case scenario, where each player has **K** one-mana targetable spells in their hand. For each of these cards there are **N + M + 2** possible targets on which to use the cards. The order of play matters, so there are **K!** different orders. Each of the cards can be intertwined into a minion attack order in $\mathbf{(N + 1)^K}$ ways.

The formula for the total number of moves **V** in this case is given by

$$V = (M+1)^N N!(N+1)^K K!$$

This number grows larger more quickly. With seven minions on each side and a full hand of ten cards (**N** = 7, **M** = 7, **K** = 10), the number of possible permutations is approximately $5 * 10^{30}$.

In addition to this there are also different card effects and interactions which are difficult to calculate.

In a real game the number of possible permutations is considerably lower, and the calculations above are done to give an estimate of a possible upper bound. Minions will often die during move sequences, and a lot of permutations are in practice identical. The number of unique permutations is therefore much smaller. It's important to note, however, that these calculations do not take into consideration the different aspects of cards, and in certain cases the number of permutations could rise higher.

Chapter 3

Theory and Background

3.1 Theoretical Foundation

The following section outlines the theories which have led to the development of MCTS techniques, as well as describing research related to some of them. The key theories and insights are found in decision theory, game theory, Monte Carlo methods and bandit-based methods.

3.1.1 Decision Theory

Decision theory is a mathematical theory for choosing which rational decisions to make when there is uncertainty. This is done by combining utility theory with probability theory to create a framework where optimal choices can be made by looking at the consequences of an action together with the chance of each outcome. The words utility and reward, as well as event and outcome, will be used interchangeably.

A rational agent is defined as one which maximizes the expected utility, and can be formalized as follows [Wooldridge (2000)]:

- $A = \{\alpha_1, \alpha_2, ...\}$ – The set of possible actions.
- $\Omega = \{\omega_1, \omega_2, ...\}$ – The set of all outcomes, which are the result of the performance of an action.
- $P(\omega|\alpha)$ – The probability of an outcome $\omega \in \Omega$ given that the agent performs an action $\alpha \in A$.
- $R : \Omega \to \mathbb{R}$ – The reward of an outcome $\omega \in \Omega$.

$E(\alpha)$ denotes the expected reward of an action $\alpha \in A$, which simply means that $E(\alpha)$ represents the utility an agent can expect to obtain when performing action α.

$$E(\alpha) = \sum_{\omega \in \Omega} R(\omega)P(\omega|\alpha)$$

A rational agent can then be said to be an agent which chooses to perform an action α to maximize the expected utility $E(\alpha)$. In games it is normal to assume that all agents are rational, which means they will attempt to maximize their expected utility.

Markov Decision Processes

This section is based on "Markov Decision Processes in Artificial Intelligence" [Sigaud and Buffet (2013)].

Using decision theory it is possible to make rational decisions, but the theory does not capture sequential decisions, which are essential to most problem domains. To do this it is possible to use a Markov Decision Process (MDP). MDPs are used to model sequential decision problems in environments which are fully observable. MDPs are an extension of Markov chains, where the extension comes from allowing choices and rewards.

Markov Chains are processes which can undergo a transition from one state to another in a state space. In addition, these processes must have the property of no memory, meaning that the probability distribution of the next state is only dependent on the current state, not on the sequence of events which preceded it. This property of serial dependence only between adjacent states is called a Markov property. MDPs expand on Markov Chains by allowing choices with potential rewards in each state, while retaining the Markov property. If only one action is available to each state, and the rewards are the same, an MDP is reduced to a Markov Chain.

Markov Decision Processes are used to model sequential decision problems in environments which are fully observable, which means the new state resulting from choosing an action is known to the system. The following components are used to model MDPs:

- $S = \{s_0, s_1, ...\}$ - The set of possible states, where s_0 is the initial state.
- $A = \{\alpha_1, \alpha_2, ...\}$ – The set of possible actions.
- $f : (S, A) \rightarrow S'$ – The state transition function which determines the probability of getting to state s' if an action α is chosen while in state s.
- $R(s)$ – The reward function for state s.

Decisions are modelled as (*state*, *action*) pairs, where the next state s' is decided based on a probability distribution depending on the current state s and the chosen action α. It is assumed the states have the Markov property.

When using MDPs, a policy π specifies which action should be chosen in each state S, and the aim is to find the policy which will yield the highest expected reward $R(s)$, which

often has to be approximated. One way to solve for this is to use value iterations, starting at the goal state and updating values backward to get the accumulated expected values for each state.

Partially Observable Markov Decision Processes

The section above assumes that every state is observable. If the system state can't be determined, however, Partially Observable Markov Decision Processes (POMDP) can be utilized instead. The components from MDPs still apply, but an additional point is needed:

- $O(s, o)$ – The observation model which defines the probability of seeing an observation o in the state s.

A key notion is a belief state, which is a distribution over system states representing the state the system believes it is currently in. A POMDP retains the Markov property, and is fully observable relative to the belief state. This means that a POMDP can be reduced to an MDP for the corresponding belief state space [Russell and Norvig (2003)].

There are, however, difficulties with this approach. When the state space of the MDPs become too large, it is not possible to explicitly represent belief states. As a consequence, the conversion of POMDPs to MDPs can be impractical. Because of this, belief states have to be represented approximately [Givan and Parr (2001)].

3.1.2 Game Theory

The theories explained so far have built the foundation on which to model a single agent's choices in sequence based on actions with different probabilities and utility scores. While MDPs and POMDPs play against nature, game theory constitutes multiple rational agents interacting with each other. The basic assumptions underlying the theory is that decision makers are rational, as well as reasoning strategically, which means using the expectations they have of other decision-makers' behavior [Von Neumann and Morgenstern (2007)].

A game is defined by established rules allowing interaction of one or several players to produce a specific outcome. The following components are used to model a game:

- $S = \{s_0, s_1, ...\}$ - The set of possible states, where s_0 is the initial state.
- $S_T \subseteq S$ – The set of terminal states.
- $A = \{\alpha_1, \alpha_2, ...\}$ – The set of possible actions.
- $K = \{k_1, k_2, ...\}$ – The set of players.
- $f : (S, A) \rightarrow S'$ – The state transition function.
- $R_k(s)$ – The reward function for state s, which can differ for each player k.

Games can be viewed as having discrete time steps starting at t_0 and continuing over time $t = 1, 2, ..., n$. The initial state s_0 corresponds to t_0, and n corresponds to a terminal state s_n. At each state s_t one or more players k_i perform actions, sometimes known as moves, which leads to another state $s_t + 1$, as is defined by the state transition function f. As the transition function can vary between players, f is used rather than the transition model explained in previous sections. The players then receive a score or a value based on their choice of action, which is defined by the reward function R. These reward scores can have different values for each player, and can take many forms such as points or monetary gain, or may be opposite, giving negative values as costs. The reward function R_k is used instead of the previous reward function R to indicate the possibly different rewards for different players.

A players probability of selecting an action α in state s is given by that players policy π. A Nash equilibrium is defined as no player being able to benefit by unilaterally switching the strategies, and exists as the combination of all the players' policies. There is always a Nash equilibrium, but it's often intractable to compute it for real games [Daskalakis (2004)].

Combinatorial Games

The following are some of the properties which can be used to classify games:

- **Zero-sum** – A game is zero sum if one player's gain or loss is exactly balanced by the loss or gain of the other player(s). Whether the game is strictly competitive two-player, which means that one players gain or loss is exactly balanced by the loss or gain of the other player.
- **Information** – A game is said to be a complete information game if all players are able to observe every state of the game. If one or more players have different observation of one or more game states, the game is partially observable.
- **Determinism** – A game is deterministic if all outcomes are known. It is stochastic if it contains elements of chance.
- **Sequential** – A game is sequential if the players perform their moves in sequence. If players apply actions at the same time it is real-time.
- **Discrete** – A game is discrete if game flow is partitioned into well-defined parts, otherwise it is continuous.

Combinatorial games are those which satisfy the following qualities: zero-sum, perfect information, deterministic, discrete and sequential. Examples of combinatorial games are Go, Chess and Tic Tac Toe, as well as many others. Simpler combinatorial games, like Tic Tac Toe, are trivially solved. Other, more complex games, like Go, have shown themselves as significant research challenges.

Games can be described in extensive form, which allow forming of models that fit dynamic situations. Extensive form makes explicit the order in which players move, and what each

player knows when making each decisions. The extensive form can be viewed as a multi-player generalizations of a decision tree [Fudenberg and Tirole (1991)].

A game tree can be used to model a sequence of actions and the corresponding consequences, including chance event outcomes, resource costs and utility. Game trees have three kinds of nodes and two kinds of branches. The edges extending from a node are branches, where each branch represents one of the possible alternatives available at that point. The set of alternatives must be mutually exclusive, meaning that if one is chosen, the others can't be chosen, and collectively exhaustive, which means that all possible alternatives must be included in this set [Safavian and Landgrebe (1990)].

Minimax

This subsection is based on Artificial Intelligence: A Modern Approach – Stuart Russel and Peter Norvig [Russell and Norvig (2003)]

The minimax algorithm is the traditional algorithm for attempting to find an optimal strategy in combinatorial games.

Normally, the optimal solution would be to find a sequence of moves which leads to a goal state. In combinatorial games, however, both players are attempting to reach their own goal states, while trying to stop the opponent from reaching theirs. The players must therefore consider alternative strategies, specifying the moves available in states resulting from every possible response by the opponent, then the players' moves in states resulting from the responses to those moves and so on. Roughly speaking, an optimal strategy leads to outcomes as least as good as any other strategy when playing against an infallible opponent.

While in a game tree, an optimal strategy can be found by looking at the minimax value of each node or state, written as $M(s)$. For the player, the value of a node is the utility $R(s)$, assuming that both the player and the opponent play optimally from this point until the end of the game. Upon reaching a terminal state, the minimax value is just the utility or reward function $R(s_T)$. Given a choice, the player will prefer a state of maximum value, while the opponent will prefer a state of minimum value. This can be defined as follows:

$$M(s) = \begin{cases} R(s) & \text{if } n \in S_T \\ max_{c \in C} M(c) & \text{if } s \text{ is a max node} \\ min_{c \in C} M(c) & \text{if } s \text{ is a min node} \end{cases}$$

$C(s) = \{c_1, c_2, ...\}$ is the set of children from state s. A max node is where the current player tries to maximize its own utility, while for a min node it is assumed that the opponent tries to minimize the utility for the current player. The minimax algorithm will compute the minimax decision from the current node s, using a recursive computation of all minimax values of the successor states $c \in C(s)$, using the defining equations. This recursion proceeds down to the leaf nodes, then backpropagates the minimax values up through the tree.

The minimax algorithm without any enhancements simply performs a complete depth-first exploration of the game tree. If the tree depth is m with b legal moves from each point, the time complexity of the algorithm becomes $\mathcal{O}(b^m)$. This quickly becomes a time cost which is impractical for real games.

To somewhat reduce the the time complexity of the minimax algorithm, pruning can be implemented in order to eliminate large parts of the tree from consideration. The most widely used pruning technique in relation to minimax is $\alpha\beta$ pruning. Applied to a standard minimax tree, the same moves are returned as normal, but branches which could in no way influence the final decision are cut away.

The general principle can be described as follows: Considering a node n somewhere in the tree so that the player can choose to move into this node. If there exists a better choice m for the the player either at the parent node of n or further up in the tree, this means that n will never actually be reached in play. Once there is enough information about n from examining some of its descendants, it can be pruned.

The name $\alpha\beta$ pruning comes from the following parameters used in the algorithm:

- α – The value of the best choice found so far in any choice along the path for the current player.
- β – The value of the best choice found so far in any choice along the path for the opposing player.

The search updates the $\alpha\beta$ values as it goes along and prunes the remaining branches at a node as soon as the values are known to be worse than the current α or β values for the max or min player, respectively.

When pruning nodes, the minimax algorithm needs to examine only $\mathcal{O}(b^{m/2})$ nodes, effectively reducing the branching factor from b to $\sqrt{b}$. This means that minimax with $\alpha\beta$ pruning can look ahead roughly twice as far as minimax in the same amount of time.

The enhancement of $\alpha\beta$ pruning together with other powerful enhancements such as transposition tables (a hash table storing moves to avoid analysis of equal moves reached by different paths) has allowed the construction of computer Chess players unbeatable by humans. However, the biggest weakness of the minimax algorithm is its reliance on an evaluation function to create a heuristic for a board state, as well as being limited to ”small enough” search spaces, which is why minimax has struggled with games such as Go (where there are problems making a strong evaluation function, as well as having an enormous search space).

To illustrate the difference, Chess has an estimated number of legal positions to be between 10^{43} and 10^{47}, with a game-tree complexity of approximately 10^{123} [Chinchalkar (1996)], which is defined as the number of leaf nodes in the solution search tree of the initial position of the game. There is no definite number for the complexity of 19x19 Go, but there are many suggestions. A typical game between experts lasts about 150 moves, with an average of around 250 choices per move, which suggest a game-tree complexity of 10^{360} [Allis (1994)].

Expectimax

Expectimax is a variant of minimax which generalizes the algorithm to include stochastic elements. The expected value can be computed by averaging the minimax values of all possible chance events, looking at the probability of each of those outcomes occurring. Essentially, this means adding chance nodes to the tree. The children of a chance nodes are all the possible stochastic events which can occur according to the game rules. More precisely, an expectimax tree is the game tree of an extensive-form game of perfect, but incomplete information [Veness (2006)].

There are three different non-leaf node types; max, min and chance. The values for max and min nodes are the defined as the highest value of it's children for max nodes, and the lowest value for min nodes. The value of a chance node is the weighted sum of its children.

Given game state s, the set of possible outcomes Ω, s_ω as the state after an event $\omega \in \Omega$ has occurred and $R(s)$ being the reward from state s, expectimax value of a node can be described as:

$$Expectimax(s) = \sum_{\omega \in \Omega} P(\omega) R(s_\omega)$$

Using expectimax it is possible to search trees with chance elements, assuming the evaluation score is proportional to the likelihood of winning.

3.1.3 Monte Carlo Methods

Monte Carlo methods are the methods from which Monte Carlo Tree Search has been named. They came originally from physics where they were used to find approximations to intractable integrals, but have later been used in many domains, including research on games:

- Abramson showed it was possible to use sampling from Monte Carlo methods to also approximate the game-theoretic value of a game, using the games Othello, Tic-Tac-Toe and Chess [Abramson (1990)].
- Brügmann uses Monte Carlo methods for Go, although without convincing results [Brügmann (1993)].
- Monte Carlo methods have been successfully used for stochastic games:
  - Backgammon [Tesauro and Galperin (1996)].
  - Bridge [Smith et al. (1998)].
  - Poker [Billings et al. (1999)].
  - Scrabble [Sheppard (2002a)].

Gelly and Silver [Gelly and Silver (2011)] took the method further and calculated a Q-value of an action, which is the expected reward of an action, defined by the following formula:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a)\omega_i$$

Where:

- $N(s,a)$ – The amount of times an action a has been selected from the state s.
- $N(s)$ – The number of times games have been played while going through state s.
- ω_i – The results of the ith simulation which is played out from state s.
- $\mathbb{I}_i(s,a)$ – 1 if action a was selected from state s on the ith play-out from state s, otherwise 0.

In Monte Carlo methods where the actions of a given state are uniformly sampled, the approach is called flat Monte Carlo. Ginsberg [Ginsberg (2001)] and Sheppard [Sheppard (2002b)] have demonstrated the capabilities of flat Monte Carlo by using it to create computer players at world champion level in both Bridge (Ginsberg) and Scrabble (Sheppard).

Use of Monte Carlo methods gives rise to two important questions:

1. How is the quality of the Monte Carlo evaluation compared to a positional evaluation?
2. To provide a theoretically satisfactory evaluation, how many simulated games are needed (Is it possible to reach the required number of simulations)?

Question 1: It can be observed that programs based on Monte Carlo methods have been shown to be sufficiently accurate in several games, as have been cited in the list above. These are games where heuristics have been difficult to generate successfully. As such, the quality of Monte Carlo evaluation is able to match or surpass positional evaluation.

Question 2: Based on the work of Bouzy and Helmstetter [Bouzy and Helmstetter (2004)] it appears that the number of simulations required to reach meaningful evaluation with Monte Carlo methods is in the order of a few thousand for games like Go and Chess. This makes it difficult to use Monte Carlo evaluations as the only method of evaluation, as the required computation time could quickly become infeasible. An example would be the time required to evaluate one million nodes in chess; around 28 hours using Monte Carlo estimates when assuming 100,000 simulations per seconds, and using 10,000 simulations per evaluation. When playing in tournament conditions this would prohibit the algorithm from doing deep searches.

To bypass the time-consuming property and still reach a reasonable search depth, *lazy evaluations* can used, combined with $\alpha\beta$ search. Rather than analyzing a node completely,

the evaluations are stopped at a certain point, for instance when there is a 95% chance the value is lower than the α, or opposite for the β value [Chaslot (2010)].

Bouzy suggested an alternative way of combining search and Monte Carlo evaluations; growing a search tree by iteratively deepening and pruning unpromising nodes, while only keeping promising nodes. Leaf nodes are evaluated by Monte Carlo Evaluations. A problem with this approach is that good branches can be pruned due to the variance in the evaluations [Bouzy (2004)].

3.1.4 Bandit-Based Methods

A well-known class of sequential decision problems are bandit-based problems, and the multi-arm bandit problem is formally equivalent to a one-state MDP, where each action returns to the same state [Vermorel and Mohri (2005)]. A bandit-based problem is the choice of action which attempts to get the best reward over multiple turns. This is done by trying to choose the actions which maximize the cumulative reward.

However, the optimal action is rarely obvious, as the underlying reward structure is unknown, and the only way to approximate it is to analyze actions from previous turns. The policy deciding which action is best is based on the reward returned from each action in the past.

This leads to an interesting dilemma in which a player has to balance exploring new options together with exploiting what currently appears optimal, so that superior strategies can be found in the long run. More specifically, the algorithm has to balance the effect of exploiting the action currently expected to give the highest reward, with exploring other, assumed worse, actions to potentially find a better action.

A bandit is defined by the following variables:

- K – The number of arms on the bandit.
- $R_{i,n}$ – Reward of playing arm i on the nth pull.
- μ_i – The unknown expected value of arm i.

When trying to create a policy for choosing the optimal action, each actions is measured by regret of that action.

Regret

Regret is defined as the difference between the reward sum associated to an optimal strategy and the sum of the collected rewards. A player's policy should attempt to minimize that player's regret.

After n plays, the formula for regret is:

$$R_N = \mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)]$$

μ^* denotes the optimal expected reward $max(\mu_i), i \in \{1, 2, ..., K\}$, while $\mathbb{E}[T_j(n)|\pi]$ are the expected number of plays for action j in the first n trials, given the policy π. Another way to say this is that the regret is the expected loss when not choosing the optimal action.

A strategy where the average regret after each play tends to zero with a probability of 1 for any bandit problem when the horizon tends to infinity is called a zero-regret strategy. Zero-regret strategies converges to an optimal, but not necessarily unique, strategy if enough plays are made [Vermorel and Mohri (2005)]. Without unlimited plays there is a need for a policy which keeps the regret low. UCB1 has been shown to keep the regret logarithmic $\mathcal{O}(\ln n)$ in finite time, as well as being computationally efficient [Auer et al. (2002)].

Upper Confidence Bounds

UCB1 is an allocation strategy that achieves logarithmic regret uniformly over n plays without any preliminary knowledge about the reward distribution, apart from the fact that their support is in range $[0, 1]$. UCB1 is defined as the sum of two terms – The first term is the current average reward, and the second term is related to the size of the one-sided confidence interval for the average reward within which the true expected reward lands within with high probability [Auer et al. (2002)].

The policy dictates to play arm i that maximizes:

$$UCB1 = \bar{X}_i + \sqrt{\frac{2 \ln n}{n_i}}$$

where X_i is the average reward from i, n_i is the number of times arm i was played and n is the number of total plays so far. The first term encourages exploitation of high-reward choices while the second term encourages exploration of less explored choices [Browne et al. (2012)]. X_i is bound to the range $[0, 1]$ to normalize the reward from any domain. When $n_i = 0$ the exploration value is arbitrarily high, ensuring each arm is visited at least once. This assumes a finite number of arms. When n_i is a low value, the exploration term is large, and

$$\lim_{n_i \to \infty} \sqrt{\frac{2 \ln n}{n_i}} = 0$$

shows that the exploration terms becomes insignificant at higher values of n_i, showing that accumulated reward is the deciding factor given a sufficient amount of plays.

3.2 Monte Carlo Tree Search

3.2.1 Overview

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a game tree according to the results. MCTS is often used for move planning in combinatorial games, but its application extends to any domain which can be described in terms of *(state, action)* pairs and where simulation can be used to forecast outcomes. This means that MCTS is able to approximate solutions to MDPs.

Since MCTS was first described it has become the focus of much AI research; a short history can be seen in Figure 3.1. Monte Carlo methods have been known since around 1940, but it wasn't until Bruce Abramson explored the idea further in 1987 that Monte Carlo methods were first used in games like Tic-Tac-Toe, Othello and Chess [Abramson and Korf (1987)]. In 1993 Bernd Brügemann developed a computer Go program with decent playing strength, even with limited knowledge of the game. This created a new angle on how to approach the game of Go, compared to the traditional algorithms [Brügmann (1993)]. It was in year 2006 that Coulumb proposed Monte Carlo Tree Search and coined the term [Coulom (2006)]. The same year, Kocsis and Szepesvri developed a tree guiding algorithm which applied bandit-based methods [Kocsis and Szepesvári (2006)] and MoGo was developed by Sylvain et al. using UCB1 based on Kocsis and Szepesvári's research [Sylvain et al. (2006)]. The following years computer Go programs became stronger, leading up to 2016 where AlphaGo, Google's computer Go player, beat the world champion [Silver et al. (2016)].

MCTS uses Monte Carlo simulations to guide a best-first search through a game tree. Unlike classic tree search algorithms like minimax with alpha-beta pruning, MCTS does not require an evaluation function, and is therefore particularly interesting in domains where it can be difficult to evaluate game states. Rather than using an evaluation function, MCTS can base a node's value on pseudo-random exploration of the search space. Then, using knowledge gathered from previous explorations, the algorithm can build a game tree, becoming increasingly more accurate in estimating the values of promising moves.

The MCTS algorithm has certain requirements due to its characteristics; at minimum, the following criteria must be fulfilled before MCTS can be used [Chaslot (2010)]:

- Scores for winning or losing a game must be bounded – rewards must be finite.
- The algorithm must have access to complete information.
- The length of the game must be finite, so simulations can terminate.

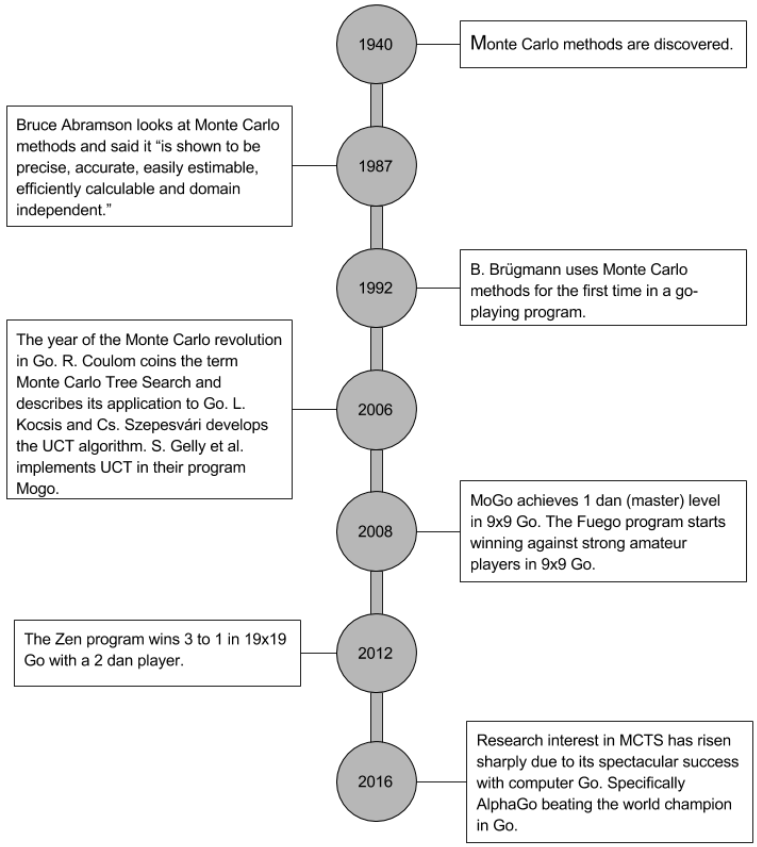


Figure 3.1: Timeline showing the discovery and evolution of Monte Carlo methods into MCTS

Basic structure

MCTS consists of two strongly coupled parts: a relatively shallow tree structure and deep simulated games. The structure of the game tree determines the first moves of the simulated games, while the result of these simulations are what shapes the game tree.

MCTS performs the following four main steps in a loop as long as the algorithm is within its computational budget, growing the tree with each iteration. The algorithm is shown in Figure 3.2 [Browne et al. (2012)]:

1. **Selection** – Starting at root node R, recursively select optimal child nodes using a selection policy π until a leaf node L is reached.
2. **Expansion** –If L is not a terminal node (i.e. it does not end the game), create one or more child nodes and select one (C).
3. **Simulation** – Run a simulated playout from C until a result is achieved.
4. **Backpropagation** – Update the current move sequence with the simulation result.

After the tree is built, a final step is needed; the move selected for play will be the 'best' child of the root node, where 'best' is dependent on the implementation of the algorithm.

Every node i in the game tree represents a different game state, while also holding two essential pieces of information:

- V_i - The current value of the state i.
- n_i - The number of times i has been visited.

The edges of the game tree represents actions. An action is done by a player to change the game state. The actions have to follow the legal options of the current state according to the rules of the game. Each node will generally have several edges corresponding to all allowed actions from itself leading to subsequent nodes.

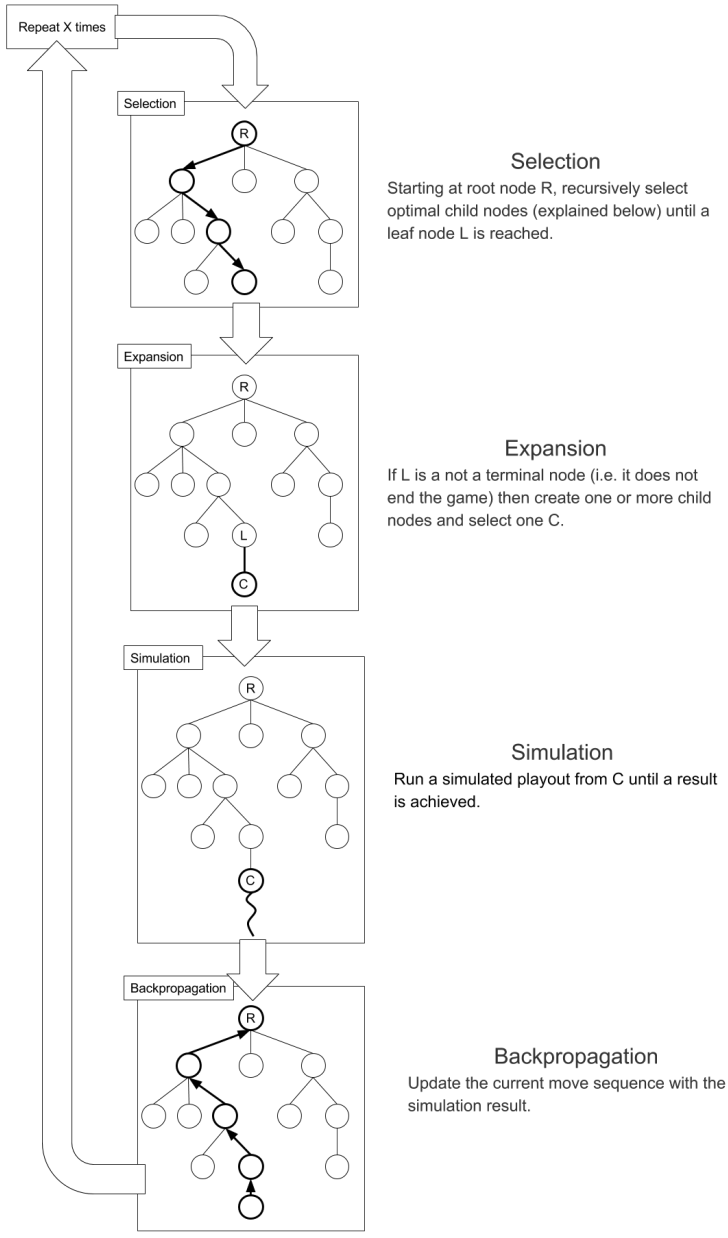


Figure 3.2: MCTS Algorithm

Pseudo-code

Listing 3.1 shows the pseudo-code of a simple MCTS implementation, without any enhancements.

Listing 3.1: MCTS pseudo-code

```
MCTS(node)
    while(withinComputationalBudget):
      # Traverse Tree with Select until leaf is reached
      while(!node.isLeaf):
        node = Select(node)

      # Expand leaf node
      node = Expand(node)

      # Simulate game to get result r
      r = Simulate(node)

      # Backpropagate results ending on the root node
      while(node.hasParent):
        updateScore(node, r)
        node = node.parent

    return finalChildSelection(node)
```

3.2.2 Selection

Selection is the process of choosing which path to traverse in the tree and figure out which areas are more promising to explore. This is done by starting at the root node, then recursively selecting children using a selection strategy until reaching a node which is not yet fully explored. A fully explored node is a node where all possible actions have been expanded into children.

When traversing the game tree, the goal is to choose the optimal path or optimal way of building the tree. To do this, nodes will have to be selected recursively according to the approximated node values generated by the simulation part of the MCTS algorithm. There are different types of selection strategies, but they are generally focused on balancing exploration versus exploitation, where exploration means weighing nodes higher if they have been visited less, and exploitation means further utilizing promising nodes and strengthening their statistical significance. If this balance is done correctly, new nodes will have a chance to be evaluated, while strong, known nodes will get more evaluation time. A commonly used node selection policy is Upper Confidence Bounds for Trees (UCT), which is based on UCB described in Section 3.1.4 [Browne et al. (2012)].

The idea is to treat the choice of child nodes as a multi-armed bandit problem where the value of the child is the approximation from the simulations, using a win and visit score, and with the UCT formula select the child that is the most promising based on the following formula:

$$V_i + B \times \sqrt{\frac{\ln N}{n_i}}$$

Kocsis and Szepesvári have shown that the probability of selecting the optimal play converges to 1 as the number of samples grow to infinity. This implies that MCTS with UCT converges to the optimal minimax tree given unlimited resources [Kocsis and Szepesvári (2006)].

The first term of the formula, V_i, is the current value of the node, usually the win rate of node i. V_i represent the exploitation term of the formula, as it is based on the strength of the prior simulations in the branch, and in cases with high success rate it will continue to explore that path. Depending on the reward function r_s mapping wins, draws and losses to an numerical value, V_i will be the average r_s of all visits to that node, residing in the range between the scores from only losses to only wins.

The second term in the function consists of two parts denoting the exploration part of the formula; B, which is the constant bias parameter (which now includes the $\sqrt{2}$ seen earlier) and $\sqrt{\ln N/n_i}$. $\ln N$ is the natural logarithm of the number of visits of the parent of the node, and n_i is the number of visits of the considered node. $\ln N$ promotes more exploration by increasing the weight of the second term more if the current branch has been visited often, making it more likely that new nodes are explored rather than exploiting win rate. n_i will differentiate between the children by increasing the value of nodes which have been visited less.

The bias B is a constant which gives optimal results when set to $\sqrt{2}$ when looking at regret, but should be tweaked to fit the current domain. Usually, with limited resources, $B < \sqrt{2}$ should be chosen. By tweaking the bias the exploration term would be weaker or stronger, and correctly tuning the bias is important to the overall strength of MCTS, as it decides the exploration vs exploitation trade-off in the selection process.

If the selection strategy has a bias for exploitation it will focus on branches where the win-rate is high. This will increase the the number of simulation in the same area of the tree and give more accurate results on those branches, but on the other hand, the search might miss other alternatives that would have shown more prominent given more simulations. On the other hand, if the bias is too large, the exploration term will distribute the searches evenly making it similar to a breadth first search, preventing deeper simulations in interesting areas of the game tree.

3.2.3 Expansion

The expansion component of MCTS is where nodes are added to the game tree. After the selection process terminates on a leaf node the expansion will generate or open a node, then add it to the game tree. As game trees for most domains are too big to be completely stored in memory, and also too large to fully search, the expansion step has to use a policy to decide how a node will be expanded to store one or more of its children in memory.

A popular expansion strategy by Rémi Coulom [Coulom (2006)] involves expanding one node per simulated game, where the node corresponds to the first position encountered during traversal which is not already stored. This is usually a randomly chosen child node of the leaf node. The strategy has several benefits; it's simple, efficient and easy to implement, as well as being low in memory consumption.

There are alternatives, for instance full node set expansion, which expands all possible children from a leaf node, or strategies involving expanding the tree to a certain depth before starting the search. The former example is often only available when a large amount of memory can be used, or it is possible to efficiently perform multiple simulations at the same time. The latter example reduces the memory usage, and will only give slightly reduced level of play. However, there is generally little effect of having different expansion strategies, and the simpler implementation of one expansion per simulated game is usually sufficient [Chaslot (2010)].

3.2.4 Simulation

The simulation process is a fast, approximated playout of the game from the current state. Each playout has to reach a game resolution, by either playing randomly or selecting pseudo-random moves following a simulation strategy.

A simulation strategy π is applied from node C to simulate playouts, until a result is achieved. The result comes from the reward function r_s which is determined by the game result of a terminal node.

A random simulation strategy is rarely optimal, as it assumes the opponent is playing random moves. As such, the use of a simulation strategy has been shown to significantly improve the level of play in MCTS in some games [Sylvain et al. (2006); Bouzy et al. (2008)]. Even so, a random simulation strategy will, with enough simulations, approximate strong play.

Such a policy is often based on domain specific knowledge that may be used to create a heuristic that can find interesting moves. An example is for MCTS in GO where heuristics can be based on pattern recognition, significance of captures and the proximity of the last move.

There are two trade-offs which must be considered before applying a policy to the simulations. First there is the trade-off between speed and accuracy. By applying knowledge

to the simulation policy the precision increases and variance of the results is reduced, giving more reliable simulations. However, the heuristic may be computationally expensive, increasing the simulation time. As the simulation cost increases the overall number of simulations per second will drop. This reduces the depth the MCTS can reach in the game tree, making the search more shallow, which in turn reduces playing strength.

Secondly there is the trade-off between exploration vs exploitation in the simulations. If there is no strategy or the strategy is too stochastic, it leads to more exploration. This can cause the simulations to become unrealistic as obviously suboptimal plays are explored. Therefore it will require a larger amount of total simulations to make up for lost thinking time spent on avoidable branches. On the other hand, the simulation strategy may be too deterministic and the same actions are always chosen from the same position, so the results will be biased towards the predicted play style following the simulation policy.

An effective simulations strategy will balance the two trade-offs, and should insure that sufficient playing strength is achieved while still being computationally effective. Creating a heuristic may be a huge challenge for some tasks and be unfeasible, but even using a simple heuristic has shown to increase playing strength [Browne et al. (2012)]. The importance of heuristics is shown by all the top performing MCTS Go programs that use domain specific knowledge, with the prime example being AlphaGo, recently beating the Go world champion [Silver et al. (2016)].

3.2.5 Backpropagation

Backpropagation is used to update the game with the information gained from the simulation step, recursively updating from node C where the simulation started, up to each parent until the root node R is reached. The reward score being propagated is dependent on whether the simulation returned a win, loss or a draw. For two player zero-sum games reward score would normally be counted positively $r_s = 1$ for games won, $r_s = 0$ for games lost, and $r_s = 0.5$ for draws, stemming from the utility scores most commonly used in game theory.

For each node visited a backpropagation strategy is used to compute a value V_i for the node. Usually V_i is the average of the scores for each node i, where $V_i = \frac{\sum r_s}{n_i}$. n_i is the number of visits [Coulom (2006)].

3.2.6 Final child selection

When the computational budget is reached, for instance a certain number of iterations or a time limit, the search terminates and returns the game tree with the updated nodes. From the rootnode R, one of the children or possible action will be selected based on their number of visits n_i and scores V_i.

The three most common child selection strategies are:

- Max child: Select the child with the highest reward V_i.
- Robust child: Select the child with the highest visit count n_i.
- Max-Robust child: Select the child that has the highest V_i and n_i.

The Robust child policy is the most common, as the selection process in MCTS will visit the most promising node throughout the search as long as the bias is weighted correctly [Browne et al. (2012)]. Max child should become the same as the Robust child during the search because of the exploitation in UCT, however with low amounts of iterations, Max child has been shown to be significantly weaker than the other options [Chaslot et al. (2007)]. Max-Robust child is a strong contestant, but there is no guarantee that such a child exists, and therefore require an uncertain amount of thinking time, which might be unfeasible in some cases.

3.2.7 Characteristics

Aheuristic

MCTS has the benefit of being aheuristic in its natural form. As such it is applicable to any domain which may be modelled using a tree, and doesn't require domain-specific knowledge. Although domain-specific knowledge is mentioned as an enhancement for a simulation strategy, the fact that MCTS can be applied to complex domains is advantageous for research on domains which are difficult to evaluate.

Anytime

As MCTS backpropagates to update the tree every cycle of the algorithm, it ensures that the game tree is up to date after every iteration. This gives the benefit of being able to stop anytime and use the current tree to make a decision. This enables the algorithm to be flexible with time consumption, and task that needs more accurate predictions may be given additional time.

Asymmetric

The selective sampling from UCT gives MCTS similarities to the human problem solving approach, where good moves are evaluated more compared to those which seem worse. This characteristic uniquely shapes the game tree for every game and each position the game is currently in [Williams (2010)]. This asymmetric shape give MCTS the ability to take advantage of high quality plays without exploring all possibilities in the tree to reach that position.

Chapter 4

Implementation

This chapter describes the implementation of the Monte Carlo Tree Search algorithm for Hearthstone together with the choices made, a heuristic based agent and the environment where the testing takes place.

4.1 Simulator

It is not possible have a computer agent interface with the official Hearthstone client directly, as the game does not have an API, nor are players allowed to use bots or AIs for playing. Additionally, it's important to note that for playing hundreds or thousands of games, using Hearthstone directly would be incredibly slow as it only allows play via the GUI.

The simulator chosen for this project is HearthSim. It is open source, and can be found at https://github.com/oyachai/HearthSim.

HearthSim is programmed in Java, and is made specifically for AI research. While incomplete (it does not yet have all the cards implemented), it is sufficiently developed to allow testing of a significant number of cards. Another important reason for choosing HearthSim is that it already has a simple board value heuristic AI incorporated, which can be used to test the MCTS agent.

4.2 Heuristic Brute Force AI

Description of how this AI is implemented is adapted from [Oyachai (2014b)].

The Heuristic Brute Force AI (HBF) is already implemented into the simulator by the simulator's creator, and will be used as one of the ways to measure the performance of the MCTS agents.

The AI is a simple rudimentary score maximizing model. Each turn the AI looks at all possible moves which can be played, and assigns a score to them based on the outcome. Thus, the AIs performance is going to be determined mostly by the function used to assign the scores. After searching as many moves as possible within a given computational budget, the sequence of moves with the best score is chosen.

4.2.1 Scoring Function

Variables with hat (ˆ) denote values associated with the opponent.

The scoring function can be written as follows:

$$S = S_b + \hat{S}_b + S_c + S_h + \hat{S}_h$$

where S_b is the friendly board score, $\hat{S}_b$ is the enemy board score, S_c is the hands score, $\hat{S}_h$ is the enemy hero' health score, and S_h is the own hero's health score.

S_b – Friendly board score

The friendly board score is the sum of all attack and health values of friendly minions on the battlefields. The formula is written as follows:

$$S_b = \sum_i (w_a a_i + w_h h_i)$$

where a is the attack value and h is the health value for all friendly minions i. The attack and health values are weighted by w_a and w_h respectively. The reasoning behind this scoring is the idea that the higher health and attack values your minions have, the better. This is often the case, and the heuristic will usually perform well if the cards used are relatively simple.

However, it is easy to find cases which makes this heuristic work against its intended purpose. One such case is the card Ancient Watcher, seen in Figure 4.1. This card has above average attack and health values, but it can not attack, and needs to be enabled by other cards to be useful.

Figure 4.1: Ancient Watcher

Tuning w_a higher means the heuristic puts a higher value on the attack of a minion, and less value if it is tuned lower. This is also the case for the tuning of w_h, with respect to health. In general, an aggressive deck will want a higher w_a tuning, to maximize damage dealt, while a defensive or control oriented deck will want a higher w_h tuning to absorb damage and defend minions.

$\hat{S}_b$ – Enemy board score

The enemy board score is the opposite of the friendly board score; it subtracts the enemy minion's weighted attack and health value.

$$\hat{S}_b = -\sum_j (\hat{w}_a a_j + \hat{w}_h h_j)$$

where a_j and h_j are the for all attack and health values for all enemy minions j. The higher the health and attack values of your opponent's minions, weighted by $\hat{w}_a$ and $\hat{w}_h$, the lower the score.

The same strengths and weaknesses seen in the friendly board score are also found here.

S_c – Hand (card) score

Cards in the hand can be viewed as a resource waiting to be spent. A player is doing better the more cards they have in their hand, as they will have more actions to choose from.

The hand score is calculated as follows:

$$S_c = w_m \sum_k m_k$$

where m_k is the mana cost of card k, with the weight w_m.

This is an extreme simplification, and it is debatable whether higher mana cost cards are actually more valuable or not. When considering cards with no card text, the mana cost will often match well with their attack and health values, and using these values will give a rough estimate of the strength of a hand. However, the issue comes when more advanced cards are analyzed, as they often have a mana cost based on their potential value for synergies with other cards, so the heuristic can easily be misleading or provide incorrect values of a players hand.

The weight m_k decides how much each card is valued based on their mana cost, with a higher m_k giving a higher value.

$\hat{S}_h$ – Enemy health score

It is usually assumed a player is doing better the less health the enemy hero has. The enemy health score captures this observation:

$$\hat{S}_h = \hat{w}_H \hat{H}$$

where $\hat{H}$ is the enemy hero's health value, and the weight $\hat{w}_H$ is the weight. If $\hat{H}$ is zero or below, $\hat{S}_h$ is set to an arbitrarily large number to make sure the AI always picks the winning move.

Although this score gives a good indication of a player's status in the game, experienced players know how to utilize health as a resource and know that depending on which deck you are playing against, different health thresholds have different values. For instance, playing against a class with a lot of direct damage spells, it's beneficial to stay above a certain health range so that you can't be killed in a single turn. Another example is the card Molten Giant, seen in Figure 4.2, which becomes stronger as your health becomes lower (the card gets cheaper to play). As such, there are nuances beyond this heuristic that have to be considered.

$\hat{S}_h$ – Your hero health score

Similar to the enemy health score being low, a player is often doing better the more health the hero has.

$$S_h = w_H H$$

Figure 4.2: Molten Giant

where H is the hero's health value and w_H is the weight. If H is zero or below, S_h is set to an arbitrarily large negative number to make sure the AI never kills itself.

The strengths and weaknesses of this score is similar to those seen in enemy health score above.

4.2.2 Move Generation

The Heuristic Brute Force AI attempt to look at all possible actions within it's turn. The number of possible moves scale exponentially with the number of minions on board and cards in hand, so the AI is limited to 30 seconds of computation, after which it gives up and plays the best sequence of moves that it has found thus far.

The HBF AI does not consider the possible moves beyond the current turn, as there are too many possibilities, which prohibits brute-force search approach.

4.2.3 Summary

By looking at the information above it's possible to give an overall assessment of the Heuristic Brute Force AI, and look at it's key strengths and weaknesses.

Strengths

- Simple board state evaluations – The AI is able to effectively analyze board states with relatively simple cards and perform efficient minion trading.
- Fast – The AI can go through a large number of board states quickly.

Weaknesses

- No look-ahead – The AI does not look past the current turn, which stops the AI from planning ahead. This means the AI can only consider how to optimize the current turn, without any form of long-term goals.
- Advanced cards – The relatively simple heuristic prevents the AI from using complex cards, which are generally needed for strong play.
- Predictability – The AI has a fixed scoring function, which makes it easy to predict.

4.3 Monte Carlo Tree Search Implementation

Although MCTS can be implemented without any changes or enhancements, it is beneficial to tailor the algorithm to the specific domain.

Show in listing 4.1 is the code used for implementing the core of the MCTS used in this project. The following subsections will go into the most important steps in depth, explaining the reasoning behind the code, as well as which choices have been made and what enhancements have been used.

Listing 4.1: Code implementing the skeleton of the MCTS algorithm.

```
1  private HearthTreeNode DoMonteCarlo(HearthTreeNode node) throws HSException {
2      int iterations = 0;
3      while(iterations < MAX_ITERATIONS/DETERMINIZATIONS) {
4          node = Selection(node);
5          node = Expansion(node);
6          int score = Simulation(node);
7          node = Backpropagation(node, score);
8          iterations++;
9      }
10     return node;
11 }
```

4.3.1 Difference in Computational Budget

There are two options when deciding the computational budget:

- Number of iterations – The algorithm runs a specific number of times before returning.
- Time – The algorithm runs as many iterations as possible within a time limit before returning.

As the simulation part of the MCTS algorithm requires the game to be played out to a terminal node, it takes a different amount of time to run depending on the stage of the game simulated from; Simulating from a game state at the start of the game will take longer than simulating from a state in the late game. This means that using a certain number of iterations will benefit from longer thinking time in the early turns, while using time will get an increased number of iterations in the later parts of the game.

In most of the experiments, number of iterations will be used as they are independent of the capacity of different computers, and are easier to use for measuring.

4.3.2 Node Structure

The tree is made up of two different kinds of nodes: Action nodes, which are nodes generated from a player performing an action on their turn, and end turn nodes, which come

from a player passing the turn to the opponent. This node structure also allows adding other node types, for instance chance nodes.

Both node types contain the following information:

- **State representation** – Information containing the current state of the game:
  - The player's **decks**.
  - The player's **hands**
  - The player's **health** values
  - The player's current and max **mana**.
  - The player's **minions**
- **Wins** – How many times a simulation from this or a descendant node has resulted in a win.
- **Visits** – How many times this or a descendant node has been visited.
- **Parent node** – The node to which this node is a child.
- **Children** – The list of child nodes this node has.
- **Possible actions** – All the possible actions which can be done from the node.

Action nodes

Upon instantiating a new node, action nodes are generated as possible actions, which can later be added to the list of children once they are explored in another iteration. There are three different types of actions nodes:

- **Attack children** – A node where the action involves one character attacking another.
- **Use card children** – A node where the action involves using a card.
- **Hero power children** – A node where the action involves using a hero power.

These nodes differ in how they are created, but are functionally the same. They simply represent the resulting state of a move.

End turn nodes

Hearthstone is a complex game in many respects, one of these being the possibility of performing multiple moves each turn. Because of this, when possible actions for a new node is created, an end turn node is always added as well, representing the action a player can take to pass the turn to the opponent. This allows the MCTS algorithm to handle both null-action turns as well as multi-action turns, since end turn nodes will function as leaf nodes within a players turn.

When an end turn node is created, the end turn step is performed for the current player, then the current player is swapped to the opponent, and the begin turn step is performed.

4.3.3 Selection

The selection strategy implemented is Upper Confidence Bounds for Trees (UCT), developed by Kocsis and Szepesvári [Kocsis and Szepesvári (2006)], and is described in Section 3.2.2. The method has been shown to balance exploration and exploitation, while giving the option of tuning with a bias parameter.

The line numbers referred to in this paragraph are from Listing 4.2, showing the code used for Selection. As long as the node is fully explored, and the node has children, selection will be performed (line 3). The children of the node are compared to each other, and the one with the highest score (based on calculateScore) is chosen (lines 9 to 13). Selection is then recursively performed until a leaf node is reached (line 18). Once a leaf node is reached, it is returned (line 21).

Listing 4.2: Code implementing the Selection part of MCTS.

```
1  private HearthTreeNode Selection (HearthTreeNode node) throws HSException {
2      // Check if there are no unexplored children (possible actions) and it's not a leaf node
3      if(node.possibleActions.isEmpty() && !node.getChildren().isEmpty()) {
4
5          HearthTreeNode candidateNode;
6          double candidateNodeScore = Integer.MIN_VALUE;
7
8          // Compare scores for all children, select the best one.
9          for (HearthTreeNode childNode : node.getChildren()){
10             double childScore = calculateScore(childNode);
11             if(childScore > candidateNodeScore){
12                 candidateNodeScore = childScore;
13                 candidateNode = childNode;
14             }
15         }
16
17         // Recursively select nodes until a leaf node is reached.
18         return Selection(candidateNode);
19     }
20
21     return node;
22 }
```

As selection is only performed on a fully explored node, it is ensured that every possible action is explored from a node before a deeper search can be allowed. This relates to the score of UCB1 as described in Section 3.1.4, where the exploration term will be arbitrarily high for an arm where $n_i = 0$, which is similar in MCTS to a child node which has not yet been expanded. The way this relation has been implemented is to stop selection upon reaching unexplored children, and move on to the expansion step, ensuring each child is visited at least once.

The code in Listing 4.3 shows the implementation of the UCT formula. The implementation is straight-forward, with the exploitation score being $V_i = wins/visits$ (line 2), and the exploration score being $B * \sqrt{\ln N/n_i}$ (lines 4 and 5).

Listing 4.3: Code implementing the exploration versus exploitation score for Selection.

```
1 private double calculateScore(HearthTreeNode node){
2     double exploitScore = node.wins/node.visits;
3
4     double exploreScore = Math.sqrt(Math.log(node.getParent().visits)/node.visits);
5     exploreScore = biasParameter * exploreScore;
6
7     return exploitScore + exploreScore;
8 }
```

The selection policy can be used regardless of which player is performing a move, as the results used for score calculation are always updated in relation to the player whose turn it is. This is further explained in Simulation (Section 4.3.5).

The bias parameter is crucial for the performance of the UCT formula, especially since multiple actions can be performed in a single turn. This will be discussed further in Final Child Selection (Section 4.3.7).

4.3.4 Expansion

The expansion strategy used is the one created by Rémi Coulom [Coulom (2006)], described in Section 3.2.3, as it is simple, efficient, easy to implement, relatively low on memory consumption, and has been shown to be sufficient for most domains.

The line numbers referred to in this section are from Listing 4.4, showing the code used for Expansion. From the list of possible actions not yet expanded, one is chosen randomly (lines 8 and 9). The chosen action is removed from the list of possible actions (line 13), and added to the node's children (line 14). Possible actions are then created for the new node depending on if it's a leaf node or not (lines 18-22). Lastly, the new node is returned (line 25).

Listing 4.4: Code implementing the Expansion part of MCTS.

```
private HearthTreeNode Expansion(HearthTreeNode node) throws HSException {

    // If node is a leaf node – no expansion
    if (node.possibleActions.isEmpty()){
        return node;
    }

    // Randomly select an action
    int randomNumber = random.nextInt(node.possibleActions.size());
    HearthTreeNode newNode = node.possibleActions.get(randomNumber);

    // Remove the new node from the node's possible actions, and add it to the node's children
    node.possibleActions.remove(newNode);
    node.addChild(newNode);

    // Generate possible actions for the new node, unless it's a leaf node
    // True if no players are dead
    if (!(newNode.data_.isDead(PlayerSide.CURRENT_PLAYER) ||
            newNode.data_.isDead(PlayerSide.WAITING_PLAYER))) {
        newNode.possibleActions = childNodeCreator.createChildren(newNode);
    } else {
        newNode.possibleActions = new ArrayList<>();
        newNode.setChildren(new ArrayList<>());
    }

    return newNode;
}
```

It's important to note the step for creating possible actions. The expansion process can be utilized to prune child nodes by not allowing certain possible actions to be appended to the newly expanded node. Doing this causes fewer moves to be considered by the MCTS algorithm. The result of this is reduction in generality, but can have practical domain specific enhancements if done correctly. It's also possible to prune known bad moves which exist merely because they are theoretically possible.

An example of pruning unnecessary child nodes can come from situations where the different permutations of minion placements are in essence the same. If no cards are played in either deck which rely on different positions on the board, like the card Alpha Wolf seen in Figure 4.3, then it doesn't matter where a minion is placed. Disallowing the different permutations to be created as children, but for instance only placing minions to the far left, can significantly reduce the size of the game tree. In a worst-case situation with 6 minions being on a players side, there are 7 possible placements for a new minion. This means that only using one of the possible permutations reduced the branching factor for this specific move from 7 to 1, which again greatly reduces branching further down in the tree.

Figure 4.3: Dire Wolf Alpha

4.3.5 Simulation

For the implementation a random simulation strategy has been chosen. This is similar to using flat Monte Carlo, discussed in Section 3.1.3, where actions are uniformly sampled, as nodes will on average receive a uniform distribution from the random selection of actions. Using this approach has led to world class computer players in Bridge and Scrabble, showing the viability of a random simulation strategy [Ginsberg (2001), Sheppard (2002b)].

Even though an informed simulation strategy has been shown to improve levels of play [Sylvain et al. (2006); Bouzy et al. (2008)], a random simulation strategy was chosen for the implementation, as Hearthstone is a game where creating an evaluation function to use with a simulation strategy is itself a difficult problem.

The code used for the simulation implementation is shown in Listing 4.5. The node is deep copied to ensure the simulations are independent from the tree (line 2). The while loop (line 7) is run until the simulations reach a terminal node where the result is returned (lines 9 to 16). All possible actions are created for the simulation node (line 19), then a random one is chosen for further simulation (lines 22 and 23).

Listing 4.5: Code implementing the Simulation part of MCTS.

```
1  private int Simulation(HearthTreeNode node) throws HSException {
2      HearthTreeNode simulationNode = new HearthTreeNode(node.data_.deepCopy());
3
4      // Remember whose turn it is
5      byte currentPlayerID = simulationNode.data_.getCurrentPlayer().getPlayerId();
6
7      while(true){
8          // Check if either player is dead, return score based on whose turn it was originally
9          if (simulationNode.data_.isDead(PlayerSide.CURRENT_PLAYER)
10                 || simulationNode.data_.isDead(PlayerSide.WAITING_PLAYER)) {
11             if (currentPlayerID == simulationNode.data_.getCurrentPlayer().getPlayerId()) {
12                 return 1;
13             } else {
14                 return 0;
15             }
16         }
17
18         // Create every possible action from the simulation Node
19         simulationNode.possibleActions = childNodeCreator.createChildren(simulationNode);
20
21         // Choose one of these possible actions at random to simulate further
22         int randomNumber = random.nextInt(simulationNode.possibleActions.size());
23         simulationNode = simulationNode.possibleActions.get(randomNumber);
24     }
25 }
```

The returned results are 1 for a win and 0 for a loss. It's not possible to get a draw. These numbers are used directly later as the reward score in backpropagation.

Depending on whose player the initial node in the simulation is affiliated with, the result is

returned from this player's perspective. This means that using the selection strategy on any parts of the tree will play both players as rational agents, such as for minimax explained in Section 3.1.2, but without using min and max nodes.

The generation of possible actions are the same as what is used when a new node is normally instantiated, as described in Node Structure (Section 4.3.2). This means that any action available to the player, including ending the turn, will be randomly selected and used for the next step of the simulation. The simulation is done as random self-play until a terminal node is reached, and the result is returned for use in Backpropagation.

It is also possible to initiate a new game from the initial simulation state, and use two agents which play randomly, but this was found to be too cost intensive to allow for a significant number of simulations. While this means that the choice of AI for the simulations can be used as a simulation strategy, a significant improvement must be made to offset the cost of initializing a new game for each simulation.

4.3.6 Backpropagation

The implemented backpropagation strategy is straightforward, updating the visits as well as the score up the game tree from the expanded node, as described in Section 3.2.5.

The code implementation for backpropagation can be seen in Listing 4.6. As in the simulation code, the current player's perspective is being saved (line 3). As long as the root node hasn't been reached, the score is updated relative to the player's perspective using UpdateScore (lines 7 to 16). Then node is set to be the parent, to backpropagate further up the tree (line 17). In the end the visits of the root node is incremented (line 20).

Listing 4.6: Code implementing the Backpropagation part of MCTS.

```
1  private HearthTreeNode Backpropagation(HearthTreeNode node, int score){
2      // Remember whose turn it is
3      byte currentPlayerID = node.data_.deepCopy().getCurrentPlayer().getPlayerId();
4
5      // While the node has a parent, backpropagate the result of the simulation up the game tree
6      while(node.getParent() != null){
7          if (node.getParent().data_.getCurrentPlayer().getPlayerId() == currentPlayerID) {
8              node = UpdateScore(node, score);
9          } else {
10             if (score == 0){
11                 node = UpdateScore(node, 1);
12             }
13             if (score == 1){
14                 node = UpdateScore(node, 0);
15             }
16         }
17         node = node.getParent();
18     }
19
20     node.visits++;
21
22     return node;
23 }
```

As mentioned in simulation, the reward score r_s takes the values 1 for wins and 0 for losses. This gives the exploitation score V_i a range of $[0, 1]$, which is effectively the win rate of the node.

The implementation code for the updateScore function is shown in listing 4.7. The code is relatively simple, iterating upon visits (line 2) and increasing wins based on the score (line 3).

Listing 4.7: Code implementing update score for Backpropagation

```
1 private HearthTreeNode UpdateScore (HearthTreeNode node, double score){
2     node.visits++;
3     node.wins+= score;
4     return node;
5 }
```

4.3.7 Final Children Selection - Action Chain MCTS

Unlike most combinatorial games, Hearthstone allows the possibility of zero to multiple actions for a player each turn. As a result, several issues must be addressed.

Unknown number of actions

As each turn in Hearthstone is different, it's not always clear how many actions a player can or should perform during a turn. The number of possible actions each turn grows as the number of resources increases, like how much mana or how many cards are available. An intuitive approach to this problem would be to run MCTS for the first action, performing it and then running MCTS for the second action and so on, until the player decides to end the turn. While this allows using the common method of child selection in MCTS, a problem arises when deciding the computational budget which should be allocated to each run of MCTS.

If a set computational budget was given to each action, the player receives more thinking time for a turn where more actions are performed, and less for fewer actions. For instance, a player using 10 seconds of thinking time for each move, would spend 10 seconds on a turn with one action and 100 seconds on a turn with 10 actions. To stop the player from going over the allowed thinking time, a harsh limit would have to be placed on the number of iterations, so that all actions could be performed. However, this would greatly weaken turns with fewer actions, as less thinking time would be used.

A solution to this problem is to allocate all available resources to constructing a single MCTS tree, and selecting actions downward the tree until an end turn node is reached. This is one of the reasons the end turn nodes were implemented, to function as a marker for the action chain. This means there is no need to guess the number of actions which

will be performed per turn, and the computational budget is distributed across the tree in accordance with the exploration versus exploitation balance.

Action Chain

An action chain is an ordered list of actions chosen in accordance with the child selection policy from the root node until a leaf node is reached in a sub-tree bounded by nodes that lead to the opponents turn. The action chain requires that there exist at least one branch where the leaf node for the current player is accessible. The definition of the leaf nodes has to be handled by the node structure of the tree, in this case by the end turn nodes. This also implies that other events such as stochastic elements have to be handled by the node structure as well, so that leaf nodes can be reached after nature events are explored.

Child Selection

The robust child policy, as described in Section 3.2.6, is applied to each step to determine which action should be selected. This means the child node with most visits is added to the chain. Robust child is chosen over Max child as turns with many actions can end up with few visits lower down the branch.

Bias parameter

The bias parameter is an important part in balancing exploration versus exploitation. This is especially so when considering an action chain, as there is a fall-off to the number of iterations for each depth of the tree. For instance, given 300 iterations spread evenly over three nodes will give only 100 iterations to further search below each of those nodes. If too many iterations are spent exploring insignificant paths, a long action chain might not receive enough iterations to find strong plays.

As mention in Section 3.2.2, a lower bias is needed when working with limited resources. Action chains require an even lower bias to force a significant enough amount of iterations so that the children of each step in the action chain can still receive statistical significant results.

4.3.8 Determinizations

A difficulty with implementing a computer Hearthstone player is that the game does not have perfect information, because the cards in the opponent's hand and the ordering of cards in both decks are unknown. This is an issue, since the agent finds itself in a set of possible belief states, rather than dealing with a fully observable environment.

A single turn in Hearthstone is in essence a Partially Observable Markov Decision Process (POMDP), which is described in Section 3.1.1. Given the set of belief states $B(s) = \{b_1, b_2, ...\}$ which are the different permutations of card combinations for a state s which make up the hidden information, the number of belief states $|B(s)|$ can be found by the following formula:

$$|B(s)| = (\hat{h} + \hat{d})! \times d!$$

where $\hat{h}$ is the amount of cards in the opponents hand, $\hat{d}$ is the amount of cards in the opponents deck and d is the amount of cards in your own deck. Even during the end of the game, where deck and hand sizes tends to be low, an example using $\hat{h} = 3$, $\hat{d} = 5$ and $d = 5$ gives $(3 + 5)! \times 5! = 4838400$ possible belief states. This is computationally infeasible as the branching factor for a state increases proportionally to the number of belief states. More specifically, it can be said that the number of branches $\mathbb{B}(s)$ from state s is the product of the number of belief states B and the number of actions $|A|$ available from that state:

$$\mathbb{B}(s) = |B(s)| \times |A|$$

It can be seen that even with a low number of possible actions, the branching factor becomes unmanageable.

The solution implemented in this project is to use only a subset of the belief states $D = \{d_1, d_2, ...\}$ where $D \subseteq B$, perform the MCTS algorithm on these belief states and then merge the resulting trees together [Cowling et al. (2012)].

These belief states, called determinizations, are chosen randomly. Each determinization is given $T/|D|$ thinking time, where T is the total computational budget for each turn. MCTS is performed on each determinization, creating a tree for each of them. These trees are then pruned to a sub-tree bounded by the end turn nodes. By looking at equal end turn nodes for the different determinization, the trees are merged into one tree containing the sum of scores for each node. If a determinization has not explored enough to reach an end turn node, that branch in the final tree does not get updates from this determinization. The reason some branches are not explored in certain determinizations is that for that belief state those plays were not deemed strong enough by the selection policy.

The main advantage of using this method is that the problem is reduced to an MDP, which makes it possible for MCTS to solve. Additionally, by only choosing a subset of the belief state, it's possible to work with the problem without taking every belief state into consideration.

There are, however, certain drawbacks. Increased number of determinizations greatly reduces the number of iterations available for MCTS, creating a more shallow search. There is also the chance that the determinizations don't correctly reflect the actual game state, which means the search will be lead into sub-par plays. It is necessary to consider the trade-offs between having a more shallow search, versus avoiding bad plays by acting on the wrong information.

4.3.9 Limitations

Although many Hearthstone cards are usable with the following implementation, there are certain limitations using the simulator and the HBF agent. First of all, there are some cards which have not yet been implemented in the simulator, and as such they can't be used. Secondly, the HBF agent does not handle hero powers, so a choice was made to not include them. Third, random effects were not handled correctly by the simulator, so cards including stochastic elements were excluded from the decks. Fourth, the mulligan stage of the game was not used, as the HBF agent does not support it.

There is a limitation with the MCTS implementation as well. Because of the way determinizations are implemented, it is currently not possible for the action chain to handle cards which allow card draw in the middle of the turn. As such, cards with card draw have been excluded.

Chapter 5

Experimental Setup

This chapter presents the different experiments which will be performed in order to evaluate the viability of the MCTS agent, as well as containing the description of the different agents used.

5.1 Agent Description

Three different agents have been implemented to use in the experiments:

1. Heuristic Bruteforce Search Agent
2. Cheating MCTS Agent
3. Determinization MCTS Agent

5.1.1 Heuristic Bruteforce Search Agent

This agent is implemented as described in section 4.2, and is used as a benchmark to test the other two agents. The variables are the ones used by the creator of the simulator and will remain the same for all experiments. They are set as follows:

- $w_a = 0.9$
- $w_h = 0.9$
- $\hat{w}_a = 1.0$
- $\hat{w}_h = 1.0$
- $w_m = 0.1$

- $w_H = 0.1$
- $\hat{w_H} = 0.1$

5.1.2 Cheating MCTS Agent

The cheating MCTS agent (CMCTS) agent is implemented as if it could see the opponents hand as well as see each card in the decks, hence the name 'cheating'. It is used to show the theoretically optimal play of the determinization MCTS agent, and should always play better than the determinization agent which is playing with hidden information. In essence, the cheating MCTS is set to have the correct belief state.

The different parameters which will be tested are:

- B - The bias parameter.
- T - The computational budget.

5.1.3 Determinization MCTS Agent

The determinization MCTS agent (DMCTS) works as described in the implementation, without any knowledge of the opponents hand other than how many cards it contain, as well as no information on the order of the cards remaining in the decks. The agent knows the content of the opponents deck, as is often the case in tournament conditions, but not the order of the cards. This knowledge is what is used to make the determinizations.

The different parameters which will be tested are:

- B - The bias parameter.
- T - The computational budget.
- d - The number of determinizations.

5.2 Experiment Scheme

Each experiment will be described by the following:

- Objective – The goal of this experiment or what is the desired outcome.
- Motivation – Why this experiment is useful; how this can be used for creating a stronger agent.
- Expected Outcome – A short description of which characteristics the expected results will have.
- Parameters – Which agents are part of the experiment and which variables they use.

The performance measure for the experiments will be the agent's win-rates. In all experiments each agent will play as the first player half of the time. The agents will all be playing with equal decks, shown in Figure B.1 (included in the appendix).

The basic deck is designed to cater to the strength of the HBF agent, as it would be trivial to include cards which make the agent fail. It is more interesting to measure the MCTS agents against an opponent which is as competent as possible. The advanced deck includes two extra cards which are considered more difficult to use properly to achieve maximum value.

The experiments are presented in the order performed, and each new experiment takes into account the data gathered from previous experiments.

5.3 Experiments

5.3.1 Bias Parameter

Objective

Establish a reasonable estimate of which bias parameter gives optimal balance of exploration and exploitation. Different bias parameters will be tried from low to high with steps in between to approximate a solution.

Motivation

As discussed in Section 3.2.2 and 4.3.7, having the correct bias parameter is important to balance exploration vs exploitation. The computational budget will be limited, and as the action chain requires a certain amount of exploitation to sufficiently evaluate nodes at lower tree depths due to the iteration drop-off caused by the cost of exploring, the tuning of the bias parameter is even more important than in regular MCTS.

Expected Outcome

Extreme low or high values will perform poorly. There will be a point where the bias parameter will have the optimal performance, which becomes worse as the bias deviates from that point. The value is expected to be below $\sqrt{2}$.

Parameters

- Agents used – CMCTS vs HBF.
- Decks used – Basic deck.
- T – As there is currently no upper bound on how much computational budget is required, 60 seconds will be used as this is usually how much time a player normally has to perform his turn. 10 and 30 seconds will also be tested to see if there is a significant difference when using less computational time. All experiments will be run on the same computer.
- B – The bias will be tested with the range $[0.25, 1.5]$ with a step size of 0.25.

5.3.2 Computational Budget

Objective

Establish a reasonable estimate of how much time the MCTS agents need to approach sufficient playing strength, as well as to see when it is no longer useful to provide the agents with additional time. As CMCTS plays as a DMCTS guessing the correct belief state, this experiment will be used to establish a lower bound of necessary iterations for each determinization.

Motivation

It's important to correctly balance finding a play in a reasonable amount of time, while also making sure the play is strong enough. The CMCTS plays with perfect information, and finding the required computational budget for the agent to play well indicates a lower bound on iterations needed for each belief state in the DMCTS agent. Assuming DMCTS approximates the correct belief state it would require at least this lower bound of iterations to get a deep enough search to perform well.

Expected Outcome

Going from lower to higher computation times will see an increase in win-rate. The win-rate will increase faster at first, but then see diminishing returns.

Parameters

- Agents used – CMCTS vs HBF.
- Decks used – Basic deck.
- $B = 0.5$.
- The different number of iterations which will be tested are:
  - $T = 1000$ iterations
  - $T = 2000$ iterations
  - $T = 3000$ iterations
  - $T = 4000$ iterations
  - $T = 5000$ iterations
  - $T = 10000$ iterations

5.3.3 Number of Determinizations

Objective

Figure out how many determinizations are optimal with a given computational time, to balance the fact that information is hidden with getting a sufficient number of simulations. Different computational budgets will be tested to see if the observations from the previous experiment holds true.

Motivation

A determinization uses $T/|D|$ computation time. Having too many determinizations will leave each tree shallow, while too few determinizations will fail to accurately estimate the opponent. Giving a determinization lower amount of iterations than a lower bound will show poor performance.

Expected Outcome

A high number of determinizations relative to the number of iterations will give poor results. There will be a point where the number of determinizations will have optimal performance, which becomes worse the more the number of determinizations deviate from this point. Longer computational time should increase overall performance relative to the number of determinizations.

Parameters

- Agents used – DMCTS vs HBF.
- Decks used – Basic.
- $B = 0.5$.
- $T = 5000$ iterations
  - $d = \{1, 2, 3, 4, 5, 10\}$
- $T = 10000$ iterations
  - $d = \{1, 2, 3, 5\}$

5.3.4 Advanced Cards

Objective

Test if the optimal number of determinizations change based on which cards are used. Two of the card Flamestrike (shown in Figure 5.1) are added to both player's decks, as it is a high impact card which requires planning over several turns to maximize it's value. This will be used to test if number of determinizations vary with card complexity. Only two cards of the same type are changed from the basic to the advanced deck to only change a single variable compared to the previous experiment.

Figure 5.1: Flamestrike

Motivation

It is interesting to see the performance of the DMCTS agent with a more advanced deck. Flamestrike is an iconic Hearthstone card, which is often used to turn the tide of battle. As Flamestrike is a card which can punish players who overextend, a higher number of determinizations might be required to plan around this card.

Expected Outcome

A higher amount of determinizations might perform better than in the previous experiment, as Flamestrike is an important card to play around. As the advanced deck has only changed two cards out of thirty, the impact won't be huge, but should show a noticeable change in win-rate for either agent.

Parameters

- Agents used – DMCTS vs HBF.
- Decks used – Advanced.
- $B = 0.5$.
- $T = 5000$ iterations
  - $d = \{1, 2, 3, 4, 5\}$
- $T = 10000$ iterations
  - $d = \{1, 2, 3\}$

5.3.5 Humans vs DMCTS

Objective

Test if the AI is able to perform well against human players. The experiment will look at who wins the games, as well as having the players answer a questionnaire about their experience, which can be found in Appendix D.

Motivation

It has always been interesting to see how computers perform against human players for every game. Hearthstone is still a game where the AI is far weaker than the best players, so it is interesting to see what level the DMCTS agent is at compared to human players.

Expected Outcome

The DMCTS agent will win some matches, but it's uncertain what the win-rate will be.

Parameters

- Agent used – DMCTS vs Human players
- Decks used – Advanced
- $B = 0.5$
- $T = 20000$ iterations
- $d = 2$

Chapter 6

Experimental Results and Analysis

6.1 Description scheme

Each experiment described in chapter 5.3 will be written in order, first showing the results, then pointing out which data is more interesting, followed by analyzing the data. All confidence intervals are calculated with a 95% confidence level.

6.2 Bias parameter

6.2.1 Results

Bias	10 seconds	30 seconds	60 seconds
0.25	0.55	0.72	0.76
0.5	0.67	0.76	0.8
0.75	0.65	0.78	0.7
1	0.59	0.73	0.67
1.25	0.5	0.68	0.74
1.5	0.44	0.66	0.78

Figure 6.1: Resulting win-rates depending on bias parameter and computational budget

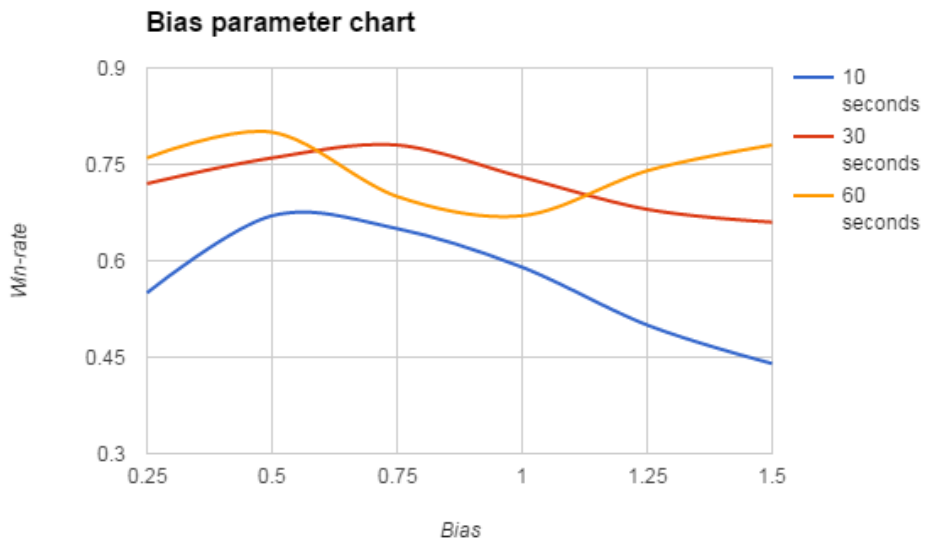


Figure 6.2: Bias results in graph form

6.2.2 Analysis

The goal of the experiment was to approximate an optimal bias parameter, which could be used for further experimentation.

The bias parameter was expected to perform poorly at very low or high bias, which can be seen in the bell curve of the graphs for 10 and 30 seconds, shown in Figure 6.2. 60 seconds shows an outlier with a high win-rate with 1.5 bias, although at 60 seconds the win-rate seems to be high for most bias parameters. It was also expected that the highest win-rates would be found below $\sqrt{2}$, which is true for 10 and 30 seconds, while 60 seconds seems to perform well with most values simply because of the large computational budget.

As can be seen in Figure 6.1. the data trends towards a higher win-rate around 0.5 to 0.75 bias, showing that a significant amount of exploitation is required, especially at lower computational budgets. As the bias increases, the win-rates drop. This is because the more exploration is performed, the less number of iterations are used for each step in the action chain, resulting in poorer overall turns when the last actions in the sequence are considered.

$B = 0.5$ showed the highest win-rate of 0.8, with a confidence interval of ± 0.07, as well as performing best for the lowest think time. This bias value will be used for the rest of the experiments.

6.3 Computational Budget

6.3.1 Results

Iterations	Win-rate
1000	0.345
2000	0.53
3000	0.6
4000	0.605
5000	0.65
10000	0.71

Figure 6.3: Resulting win-rates depending computational budget

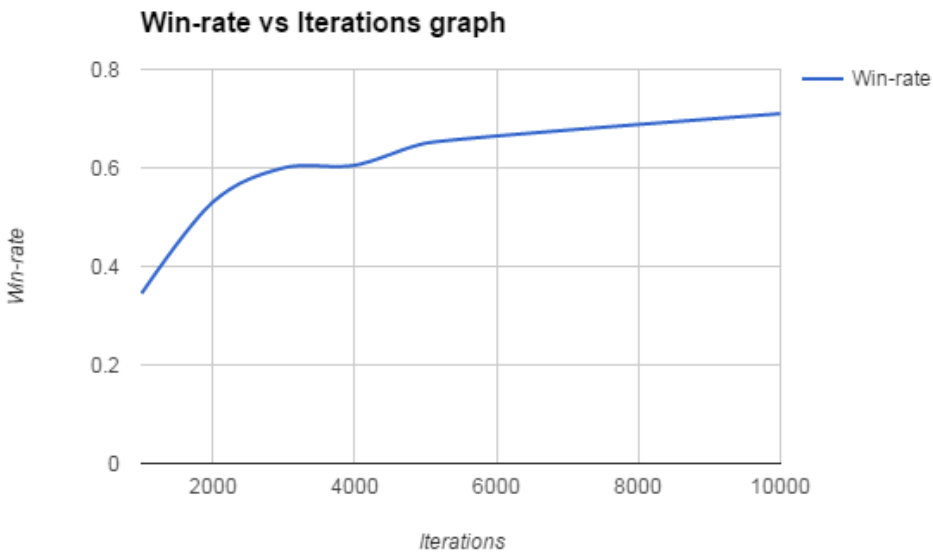


Figure 6.4: Computational budget results in graph form

6.3.2 Analysis

The objective of this experiment was to establish the relationship between win-rate and computational budget. As can be seen from Figure 6.4 there is a clear increase in win-rate as more computational budget is added.

The win-rate seems logarithmically proportional to the computational budget, which means that as long as more iterations are add, the win-rate will rise, although there will be diminishing returns. At 3000-4000 iterations the gradient decreases, showing lower increase in win-rate when adding more iterations.

Using CMCTS, a lower bound can be determined to establish the minimum amount of iterations required for each determinization. This bound can vary on the game environment and the agent's settings, but each determinization will not perform better than CMCTS using the same computational budget.

The goal of the DMCTS agent is to approximate the CMCTS agent. This requires a number of determinizations which can together approach a representation of the correct belief state. However, each determinization requires enough iterations to perform well enough. The balance between determinizations and iterations varies, but of course, the more computational budget is available, the better the performance. The correct balance for iterations per determinization is expected to be higher than the lower bound.

6.4 Number of Determinizations

6.4.1 Results

Determinizations	Win-rate
1	0.56
2	0.42
3	0.41
4	0.3
5	0.34
10	0.31

Figure 6.5: Resulting win-rates depending on determinizations for $T = 5000$

Determinizations	Win-rate
1	0.63
2	0.53
3	0.55
5	0.43

Figure 6.6: Resulting win-rates depending on determinizations for $T = 10000$

6.4.2 Analysis

The point of the experiment was to look at the performance of different number of determinizations, to establish the balance between deep searches versus accuracy in estimating belief states.

Figure 6.5 shows that the win-rates for 5000 iterations are higher for a low number of determinizations, especially for just a single determinization. This matches well with the earlier prediction considering the lower bound required for each determinization. 2 and 3 determinizations are close to the lower bound, and still perform ok, but increasing the number of determinizations further gives lower win-rates.

Figure 6.6 shows the win-rates for 10000 iterations. As expected the win-rates are higher with a larger number of iterations. The same drop-off in win-rate when spreading out iterations over several determinizations can also be seen. This shows that deep searches from a higher computational budget is more impactful than number of determinizations.

Both 5000 and 10000 iterations have lower win-rates than the ones for CMCTS with the same computational budget.

6.5 Advanced Cards

6.5.1 Results

Determinizations	Winrate
1	0.453
2	0.405
3	0.32
4	0.39
5	0.35

Figure 6.7: Resulting win-rates using advanced cards depending on determinizations for $T = 5000$

Determinizations	Winrate
1	0.52
2	0.44
3	0.39

Figure 6.8: Resulting win-rates using advanced cards depending on determinizations for $T = 10000$

6.5.2 Analysis

This experiment was similar to the previous one, but using two different cards, creating more advanced game play options. The card added (Flamestrike, seen in Figure 5.1) was specifically chosen as it is a card which often has to be played around if it is expected the opponent has it in the hand.

By comparing 5000 iterations in number of determinizations and advanced cards (Figures 6.5 and 6.7 it can be seen that it is similar win-rates overall, but significantly lower for 1 determinization while using the advanced deck. The low win-rate is probably caused by the inaccuracy from a single determinization, which may have made the DMCTS agent play to passively when it predicted the opponent to possess the Flamestrike in hand. As for the significant drop in win-rate from 2 to 3 determinizations shows that there is not enough iterations to support more determinizations. More determinizations would be required to accurately predict the true belief state, which is important when including cards such as Flamestrike.

Figure 6.8 shows increased performance compared to Figure 6.7 which is because of the increase of computational budget, again confirming that larger computational budget is the parameter with highest impact.

6.6 Humans vs DMCTS

6.6.1 Results

	Player 1	Player 2	Player 3
Number of wins	5	2	2
Number of games	9	3	3
Highest rank received	10	13	8
Usual amount of play time	1h/week	Twice a month	3h/week or nothing
Expected level of play from the AI	3	3	5
Perceived level of play from the AI	4	5	4
Player enjoyment	4	4	4
Obvious misplays or weird behavior from the AI?	5	5	5
Similarity to human play	5	5	5

6.6.2 Analysis

By looking at Figure 6.9 it can be seen that the players are among the top 5%-15% of Hearthstone players, which means the players are competent compared to the total player-base.

The DMCTS agent won 6 out of 15 games giving it a win-rate of 0.4, with many of the matches being close. This win-rate shows that the agent still has some way to go to perform well within the given parameters, but is strong enough to take games off of relatively skilled human opponents.

The players did not observe any obvious misplays or weird behavior from the AI, showing the robustness of the DMCTS agent. The players reported that the agent played as they would expect a human to play using these decks, as well as perceiving high level moves, setting up the board correctly and planning several turns ahead.

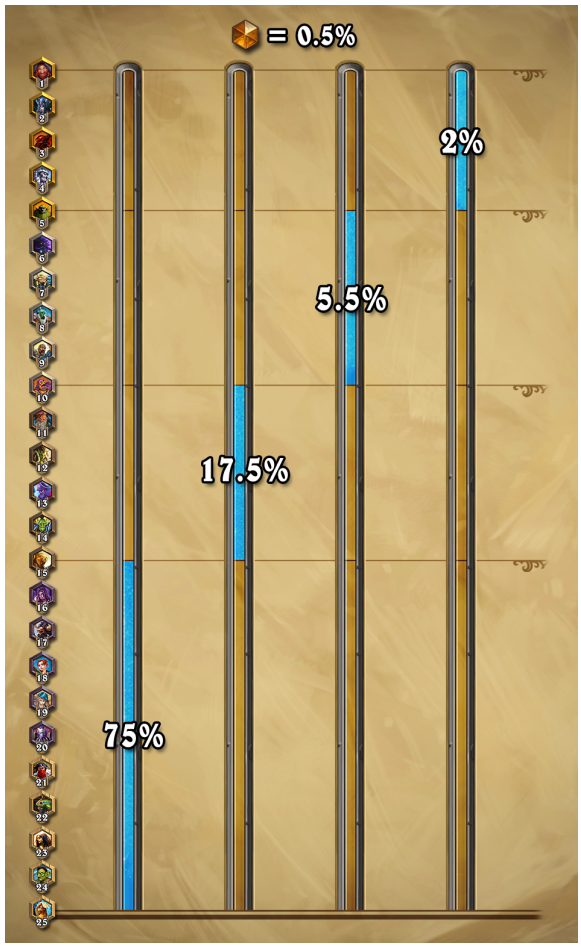


Figure 6.9: Top % Hearthstone players based on rank

Chapter 7

Conclusion

The research questions raised in Section 1.2 were:

1. Can Monte Carlo Tree Search be implemented to create a strong Hearthstone computer player?
2. Which parameters and implementation choices are important for the agent's performance?

To answer the first question, MCTS has been analyzed and implemented to create a Hearthstone computer player, which is able to handle multiple actions within a turn, deal with hidden information and perform well against both human and computer opponents.

Several implementation choices were made in order to create this agent, beyond what is considered the standard MCTS algorithm. A node structure adding end turn nodes was used to make each turn bounded, allowing the generation of the action chain, which means the algorithm can dynamically allocate computational time to the interesting part of the game tree, and return a full turn worth of actions.

The choice of bias parameter, balancing exploitation versus exploration, proved significant. This is because of the action chain, which required a certain amount of exploitation to reduce drop-off for each step in the chain.

Another addition was the use of determinizations to handle hidden information. This created a need to balance the computational budget against the number of determinizations to get strong performance. It was established that although the balance can vary for different parameters, a lower bound can be estimated for the amount of iterations per determinization.

Even though using a well-estimated bias and number of determinizations was important, the biggest factor contributing to the agent's performance was the number of iterations.

This was partly because increasing computational budget allows for deeper search, but also because more iterations allows for more determinizations.

It was shown that the DMCTS agent was able to perform well against a heuristic-based agent, as well as taking games off of strong human opponents, showing strong performance when using Monte Carlo Tree Search. Even though the agent is not able to play every aspect of Hearthstone, the work done in this thesis gives a strong indication of future possibilities.

Chapter 8

Future Work

This chapter describes future work which can be done either to improve the playing strength of the algorithm, or add functionality to make the agent more complete.

8.1 Handling of Stochastic Elements

An important aspect of Hearthstone is stochastic elements. As such, a necessary step towards making an agent which can handle all cards is to implement a way to handle randomness. One approach to this would be to add Chance nodes, which would branch into every possible outcome, and have a value similar to what is seen in expectimax, described in Section 3.1.2.

8.2 Mulligan

Another necessary step towards making a complete Hearthstone agent would be to include the mulligan step, described in Section 2.2.3. An approach to this could be to make an evaluation function for the mulligan step, relative to the opponent's hero, which is simpler than creating an evaluation function for a board state, or simply run Monte Carlo Tree Search based on the opening hand.

8.3 Duplicate Node Pruning

In Hearthstone, different sequences of moves can lead to equal board states. Pruning duplicate board states would reduce unnecessary branching. One way to do this would be

to create a hash which could uniquely identify a board state, and check each board state against a hash table whenever a new node is added to the tree.

There are issues with this approach, however, as duplicate nodes can exist in different parts of the tree, and backpropagation would have to account for this.

Another possibility for pruning nodes would be to look at nodes which are similar, but not equal. There are many situations in Hearthstone where states are in effect equal, and could potentially be merged. An example is dealing 4 damage to a minion with 3 health compared to dealing 4 damage to a minion with 4 health. The initial states are different, but the action chosen is the same.

8.4 Thread Optimization

Monte Carlo Tree Search is an algorithm where it is relatively easy to run different parts on multiple threads. There are several areas where this can be applied:

- Run MCTS on separate branches from the root node on different threads, then merge the resulting tree together.
- Run determinizations on different threads before merging them together.
- Instead of running a single simulation after expansion, multiple simulations could be run on different threads.

There are many possibilities to choose from when it comes to thread optimization, but the goal is either to increase the number of simulations or to increase the statistical significance of a node's value.

8.5 Simulation Strategy

As written in Section 3.2.4, having a simulation strategy can significantly increase level of play. Using a simulation strategy to approximate the opponent's play would give more accurate results. However, there is a need to balance the cost of a simulation strategy with the increase accuracy in rewards.

Bibliography

Abramson, B., 1990. Expected-outcome: A general model of static evaluation. Pattern Analysis and Machine Intelligence, IEEE Transactions on 12 (2), 182–193.

Abramson, B., Korf, R. E., 1987. A model of two-player evaluation functions. In: AAAI. pp. 90–94.

Academy of Interactive Arts and Sciences, 2014. 18th Annual DICE Awards. http://www.interactive.org/news/18th_dice_awards_winners_.asp, [Online; accessed 19-November-2015].

Allis, L. V., 1994. Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, Universiteit Maastricht.

Andersson, M., Hesselberg, H., 2015. Specialization project.

Auer, P., Cesa-Bianchi, N., Fischer, P., 2002. Finite-time analysis of the multiarmed bandit problem. Machine learning 47 (2-3), 235–256.

BAFTA Video Games Award, 2014. Bafta video game awards: Destiny triumphs, while indies dominate. http://www.theguardian.com/technology/2015/mar/13/bafta-video-game-awards-destiny-triumphs-indies-dominate, [Online; accessed 19-November-2015].

Billings, D., 1995. Computer Poker. Master's thesis, Department of Computing Science, University of Alberta.

Billings, D., Castillo, L. P., Schaeffer, J., Szafron, D., 1999. Using probabilistic knowledge and simulation to play poker. In: AAAI/IAAI. pp. 697–703.

Blizzard Entertainment, 2015. http://us.battle.net/hearthstone/en/, [Online; accessed 18-November-2015].

Bouzy, B., 2004. Associating shallow and selective global tree search with monte carlo for 9× 9 go. In: Computers and Games. Springer, pp. 67–80.

Bouzy, B., Helmstetter, B., 2004. Monte-carlo go developments. In: Advances in computer games. Springer, pp. 159–174.

Bouzy, B., Winands, M., Chaslot, G., 2008. Progressive strategies for monte-carlo tree search.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., 2012. A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4 (1), 1–43.

Brügmann, B., 1993. Monte carlo go. Tech. rep.

Buro, M., Furtak, T., 2003. RTS Games as Test-Bed for Real-Time AI research. In: Proceedings of the 7th Joint Conference on Information Science (JCIS 2003). pp. 481–484.

Chaslot, G., 2010. Monte-carlo tree search. Maastricht: Universiteit Maastricht.

Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., Bouzy, B., 2007. Progressive strategies for monte-carlo tree search. In: Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007). pp. 655–661.

Chinchalkar, S., 1996. An Upper Bound for the Number of Reachable Positions. ICCA Journal 19 (3), 181–183.

Coulom, R., 2006. Efficient selectivity and backup operators in monte-carlo tree search. In: Computers and games. Springer, pp. 72–83.

Cowling, P. I., Powley, E. J., Whitehouse, D., 2012. Information set monte carlo tree search. Computational Intelligence and AI in Games, IEEE Transactions on 4 (2), 120–143.

Daskalakis, K., 2004. The complexity of Nash equilibria.

Engadget, 2013. Blizzard announces digital collectible card game Hearthstone: Heroes of Warcraft. http://www.engadget.com/2013/03/22/watch-blizzards-pax-east-announcement-right-here/, [Online; accessed 19-November-2015].

Eurogamer, January 2015. Hearthstone: Goblins vs Gnomes review. http://www.eurogamer.net/articles/2015-01-08-hearthstone-goblins-vs-gnomes, [Online; accessed 20-November-2015].

Fudenberg, d., Tirole, J., 1991. Game theory. MIT press.

Game Awards, 2014. Dragon Age: Inquisition Wins GOTY at Game Awards. http://www.gamespot.com/articles/dragon-age-inquisition-wins-goty-at-game-awards/1100-6424005/, [Online; accessed 19-November-2015].

GameInformer, August 2014a. Hearthstone: Curse of Naxxramas. http://www.gameinformer.com/games/hearthstone_curse_of_naxxramas/b/pc/archive/2014/08/25/single-player-adventure-awaits.aspx, [Online; accessed 20-November-2015].

GameInformer, December 2014b. Hearthstone: Goblins Vs. Gnomes. http://www.gameinformer.com/games/hearthstone_goblins_vs_gnomes/b/pc/archive/2014/12/10/there-s-no-place-like-gnome.aspx, [Online; accessed 20-November-2015].

GameRankings, 2014. Hearthstone: Heroes of Warcraft. http://www.gamerankings.com/pc/710060-hearthstone-heroes-of-warcraft/index.html, [Online; accessed 19-November-2015].

GameSpot, 2014. Hearthstone: Heroes of Warcraft - Mobile Game of the Year. http://www.gamespot.com/articles/hearthstone-heroes-of-warcraft-mobile-game-of-the-/1100-6423909/, [Online; accessed 19-November-2015].

GameSpot, November 2015. Hearthstone Reaches 40 Million Players, Up 10 Million. http://www.gamespot.com/articles/hearthstone-reaches-40-million-players-up-10-milli/1100-6432063/, [Online; accessed 19-November-2015].

GameTrailers, 2014. GameTrailers Best of 2014 Awards - Best Multiplayer. http://www.gametrailers.com/videos/7e6kcd/gametrailers-best-of-2014-awards-best-multiplayer, [Online; accessed 19-November-2015].

Gelly, S., Silver, D., 2011. Monte-carlo tree search and rapid action value estimation in computer go. Artificial Intelligence 175 (11), 1856–1875.

Gilpin, A., Sandholm, T., 2006. A competitive Texas Hold'em Poker player via automated abstraction and real-time equilibrium computation. In: Proceedings of the National Conference on Artificial Intelligence. Vol. 21. p. 1007.

Ginsberg, M. L., 2001. Gib: Imperfect information in a computationally challenging game. Journal of Artificial Intelligence Research, 303–358.

Givan, B., Parr, R., 2001. An introduction to markov decision processes. Purdue University.

Hearthstone Gamepedia, 2015a. Hearthstone advanced rulebook. http://hearthstone.gamepedia.com/Advanced_rulebook, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015b. Hearthstone cards. http://hearthstone.gamepedia.com/Card, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015c. Hearthstone deck. http://hearthstone.gamepedia.com/Deck, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015d. Hearthstone gameplay. http://hearthstone.gamepedia.com/Gameplay, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015e. Hearthstone minion. http://hearthstone.gamepedia.com/Minion, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015f. Hearthstone spell. http://hearthstone.gamepedia.com/Spell, [Online; accessed 27-September-2015].

Hearthstone Gamepedia, 2015g. Hearthstone weapon. http://hearthstone.gamepedia.com/Weapon, [Online; accessed 27-September-2015].

IGN, 2014. Hearthstone: Heroes of WarCraft Review. http://uk.ign.com/games/hearthstone-heroes-of-warcraft/mac-163237, [Online; accessed 20-November-2015].

Kocsis, L., Szepesvári, C., 2006. Bandit based monte-carlo planning. In: Machine Learning: ECML 2006. Springer, pp. 282–293.

Laird, J., VanLent, M., 2001. Human-level AI's killer application: Interactive computer games. AI magazine 22 (2), 15.

Malone, T., September 1980. What makes things fun to learn? Heuristics for designing instructional computer games. In: Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems. ACM, pp. 162–169.

MetaCritic, 2014. Hearthstone: Heroes of Warcraft. http://www.metacritic.com/game/pc/hearthstone-heroes-of-warcraft, [Online; accessed 19-November-2015].

Newzoo, 2014. Newzoo Global Games Market Report.

Oyachai, October 2014a. Estimate on number of possible Hearthstone moves. http://buddypanda.com/?p=450, [Online; accessed 28-September-2015].

Oyachai, June 2014b. HearthSim – Intro. http://buddypanda.com/?p=30, [Online; accessed 28-September-2015].

PCGamer, September 2014. Curse of Naxxramas. http://www.pcgamer.com/hearthstone-curse-of-naxxramas-review/, [Online; accessed 20-November-2015].

Russell, S. J., Norvig, P., 2003. Artificial Intelligence: A Modern Approach, 2nd Edition. Pearson Education.

Safavian, S. R., Landgrebe, D., 1990. A survey of decision tree classifier methodology.

Sheppard, B., 2002a. Towards perfect play of scrabble. Ph.D. thesis, Maastricht university.

Sheppard, B., 2002b. World-championship-caliber scrabble. Artificial Intelligence 134 (1), 241–275.

Shi, J., Littman, M. L., 2001. Abstraction methods for game theoretic Poker. In: Computers and Games. Springer, pp. 333–345.

Sigaud, O., Buffet, O., 2013. Markov decision processes in artificial intelligence. John Wiley & Sons.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529 (7587), 484–489.

Smith, S. J., Nau, D., Throop, T., 1998. Computer bridge: A big win for ai planning. Ai magazine 19 (2), 93.

Sylvain, G., Wang, Y., Munos, R., Teytaud, O., 2006. Modification of uct with patterns in monte-carlo go. Technical Report RR-6062 32, 30–56.

Tesauro, G., Galperin, G. R., 1996. On-line policy improvement using monte-carlo search. In: NIPS. Vol. 96. pp. 1068–1074.

Veness, J., 2006. Expectimax enhancements for stochastic game players. Ph.D. thesis, UNIVERSITY OF NEW SOUTH WALES.

Vermorel, J., Mohri, M., 2005. Multi-armed bandit algorithms and empirical evaluation. In: Machine learning: ECML 2005. Springer, pp. 437–448.

Von Neumann, J., Morgenstern, O., 2007. Theory of games and economic behavior. Princeton university press.

Whitehouse, D., 2014. Monte Carlo Tree Search for games with Hidden Information and Uncertainty. Ph.D. thesis, University of York.

Williams, G., 2010. Determining game quality through uct tree shape analysis. Ph.D. thesis, MS thesis, Imperial Coll., London.

Wooldridge, M. J., 2000. Reasoning about rational agents. MIT press.

Appendix A

Beginners guide to Hearthstone continuation

This section is inspired by the work done in the specialization project [Andersson and Hesselberg (2015)].

A.1 Cards

Adapted from Hearthstone Gamepedia (2015b).

Cards are basic pieces which make up Hearthstone. There are three categories of cards: spells, weapons and minions, which all represent actions a player can take.

A typical Hearthstone card can be seen in Figure A.1. In the top left of the card, the mana cost is displayed, indicating how much mana a player must spend to play this card. The mana cost of a card is usually in the range from 0 to 10 mana. There are special cases though, like Molten Giant seen in Figure 4.2, which has an initial mana cost which is higher, but gets lower the less health the hero has.

Every card features a portrait and card text. The card text varies for each card, and can also be left blank. When it comes to minion and weapon cards, the card text will describe abilities or special effects the card possesses, while spell cards will have a description of the spells effect when played. Card text can be removed from a minion with a ”silence” effect. Seen in the bottom left corner of the minion card is the attack value, while in the bottom right is the health value. Weapons show similar values, displaying the attack strength of the weapon, and the durability. Spell cards do not have any values here.

Figure A.1: A typical Hearthstone card

A.2 Decks

Adapted from Hearthstone Gamepedia (2015c).

A deck is a set of 30 cards. In a game, the cards drawn by the players come from the deck they choose beforehand. There are pre-made basic decks available, or the players can make their own decks.

Figure A.2 shows how a deck list might look like. On the left side of the figure the mana costs are displayed, while on the right side are the number of copies included of that card. The name is displayed in the middle. There can only be two copies of each normal card in a deck, and only one copy of legendary cards.

Deck is a term which refers not only to the collection of cards the players bring into the game, but also to the stack of cards which is shown to the right hand side of the battlefield shown in Figure 2.2. In this context, the word deck is used to refer to the cards which a player has not yet drawn. Card draw and certain spells exhaust a player's deck, and when they run out of cards they begin taking fatigue damage. A player can mouse over the decks to see how many cards are remaining in each. The deck is randomly shuffled at the start of a match, after the mulligan step is performed.

Figure A.2: A Hearthstone decklist

Creating a custom deck is generally recommended, as a player can learn more about the game's deeper strategic elements. It is also usual for players to copy decks found online. A player is free to change their decks as often as they wish, outside the game.

Making decks is an important aspect of Hearthstone. There are many types of deck, with a huge amount of possible variations. Common archetypes include control decks, aggressive decks, and decks which try to balance the two, or use different strategies. As different classes have different cards, in addition to the neutral cards which can be used by all classes, certain classes synergize well with certain deck types.

An important detail to consider while building a deck is the size limit of 30 cards. Players are forced to cut down all the possible cards into a smaller set which synergize well and focus upon the player's chosen strategy.

A.3 Minions

Adapted from Hearthstone Gamepedia (2015e).

A.3.1 Overview

A minion is a creature which persists on the battlefield, controlled by a player. A minion is controlled by the player who summoned it, and is usually used to attack the opponent's minion or the enemy hero. Minions are an important aspect of Hearthstone, and is often responsible for most of the damage dealt in a game.

Figure A.3: A Hearthstone card affecting adjacent cards

A.3.2 Summoning

A minion on the battlefield is summoned by playing a minion card. To summon a minion, the mana cost, displayed in the top left corner of the card, must be payed. If all criteria is met, the minion card is transformed into a creature on the game board where the player chose to play it, and is represented by a portrait. After being summoned the minion persists on the board until it is destroyed or other cards interact with it to transform it or return it to the owners hand. A minion dies when it's health is reduced to zero, or a destruction effect is used to kill it directly.

It's important to note that if a minion is returned to the hand or shuffled into the deck, any enhancements gained while being on the board are lost, and the attack and health is set to the original values. However, a card which has been transformed into something else will not return as the original card, but as a new card representing the transformation.

It is sometimes important to place minions correctly on the board, as some cards have effects which depend on positioning. An example is Dire Wolf Alpha, shown in Figure A.3, which gives adjacent minions a buff to attack. Minions generally don't change their position once placed, but if a minion is destroyed, it's neighbour minions become adjacent to one another. If a minion is taken over by a spell or effect, or summoned through a special ability, they will always appear on the furthermost right position available.

Both player can have a maximum of seven minions played at the same time, and can't play more minions once the maximum is reached. Cards and effects which summon

more minions will not be usable, and effects which take control over enemy minions are wasted.

A.3.3 Actions

When a minion enters the battlefield they are considered exhausted. This is indicated by a ”zzz” above the card, and lasts for one turn. If a minion has the Charge descriptor they do not enter the battlefield exhausted, and can attack immediately. Unless a minion has Charge they are considered exhausted no matter how they are put onto the battlefield. Another term for exhaustion is summoning sickness.

Minions can attack once a turn. Minions with the Windfury description can attack two times. A minion can attack enemy minions or the opposing hero.

When one minion attacks another, it doesn't matter who attacked. Both minions will have their health values reduced by a number equal to the opposing minions attack value. Damage is dealt simultaneously. If a minion's health reaches zero it is destroyed and removed from the game board.

In combat between minions and heroes the order matters. If a minion attacks a hero, only the hero takes damage, but a heroes weapon will not have it's durability reduced. If a hero attacks a minion, both take damage. If the minion deals damage to a hero greater than the remaining health, the hero is destroyed and the other player wins.

Most minions have the abilities to attack, unless their card text specifically indicates otherwise, their attack value is 0, or the minion has been affected by a Freeze effect. A frozen minion can't choose to attack, but will participate in minion combat as normal if it is attacked.

Some minions have other abilities beyond normal attacks, giving them additional powers or having effects which trigger in response to some events.

A.3.4 Special properties

Following is a list of some of the most common types of minion abilities or properties:

- **Taunt** – This minions has to be attacked before other minions or the hero can be attacked.
- **Triggered effects** – This minion causes something to happen as a response to a certain event
- **Deathrattle** – An effect triggers when this minion dies.
- **Poison** – Any damage dealt by this minion destroys the opposing minion. This does not work on heroes.
- **Spell damage** – Improves the damage dealt by spells the controlling player casts.

- **Stealth** – This minion can't be targeted. Stealth is removed if the minion attacks or deals damage in some way.
- **Windfury** – This minion can attack twice.
- **Silence** – This minion has it's card text removed.
- **Enrage** – After taking damage, this minion provides a boon to the controlling player, or deals more damage
- **Freeze** – This minion is unable to attack.
- **Divine Shield** – The first damage this minion takes is blocked.

A.3.5 Minion types

Some minions belong to a specific type or tribe, as can be seen with the Murloc Warleader in Figure A.4, which belongs to the Murloc tribe. Tribe cards usually possess synergy, allowing powerful decks to be built around them. The Murloc Warleader provides all other murlocs with +2 attack and +1 health. A minion type is not removed if it is silenced.

Figure A.4: A Hearthstone card with a creature type

A.4 Spells

Adapted from Hearthstone Gamepedia (2015f).

A spell card is used to activate a one-time effect or ability, which is described in the card's text. Spells do not have attack or health values, only mana costs. A spell can only be cast on your turn.

Certain minions have synergy with spells, providing additional damage or effects when a spell is cast.

(a) A 'secret' card **(b)** A normal Spell

Figure A.5: Hearthstone spell cards

A special kind of spell, available only to certain classes, are secrets. They are the same as normal spells, except they have a delayed, hidden effect, which is only triggered when a specific event occurs. The opponent can see that a secret has been played, but not which secret it is. Secrets can only be activated on the opponent's turn, preventing a player from triggering it.

A.5 Weapons

Adapted from Hearthstone Gamepedia (2015g).

Weapons are cards which can be used to give heroes attack values. They are only available to certain classes. A weapon has an attack value and a durability value. Every time an attack is made, the durability of the weapon is reduced by one. If no durability is left, the weapon is destroyed. A hero can only have one weapon equipped at a time.

There are different ways to equip a weapon, although the most common method is by playing a weapon card. Other alternatives are from minions with battlecries or deathrattles with weapon effects, as well as the rogue hero power. There are also cards which synergize with weapons, or counter them.

Figure A.6: A Hearthstone weapon

Appendix B

Decks

(a) Deck with basic cards **(b)** Deck with advanced cards

Figure B.1: Decks used for experiments

Appendix C

Raw Data

The following subsections present the raw data gathered from running the experiments described in Chapter 5.

Note: As some experiments were run on multiple computers for different lengths of time, the number of games don't always match up perfectly. However, all simulations are displayed here for completeness.

C.1 Bias Parameter

Result of the bias parameter experiments (Section 5.3.1)

Bias	Wins	Games	Winrate
0.25	82	150	0.547
0.5	101	150	0.673
0.75	97	150	0.654
1	89	150	0.593
1.25	76	150	0.507
1.5	133	300	0.443

Figure C.1: CMCTS vs HBF, Basic decks, $T = 10$

Bias	Wins	Games	Winrate
0.25	72	100	0.72
0.5	96	126	0.761
0.75	112	144	0.778
1	140	192	0.729
1.25	68	100	0.68
1.5	82	125	0.656

Figure C.2: CMCTS vs HBF, Basic decks, $T = 30$

Bias	Wins	Games	Winrate
0.25	76	100	0.76
0.5	108	135	0.8
0.75	70	100	0.7
1	126	187	0.674
1.25	74	100	0.74
1.5	89	113	0.788

Figure C.3: CMCTS vs HBF, Basic decks, $T = 60$

C.2 Computational Budget

Iterations	Wins	Games	Winrate
1000	69	200	0.345
2000	106	200	0.53
3000	120	200	0.6
4000	121	200	0.605
5000	130	200	0.65
10000	142	200	0.71

Figure C.4: CMCTS vs HBF, Basic decks, $B = 0.5$

C.3 Number of Determinizations

Determinizations	Wins	Games	Winrate
1	56	100	0.56
2	42	100	0.42
3	41	100	0.41
4	30	100	0.3
5	34	100	0.34
10	31	100	0.31

Figure C.5: DMCTS vs HBF, Basic decks, $B = 0.5$, $T = 5000$

Determinizations	Wins	Games	Winrate
1	63	100	0.63
2	53	100	0.53
3	55	100	0.55
5	43	100	0.43

Figure C.6: DMCTS vs HBF, Basic decks, $B = 0.5$, $T = 10000$

C.4 Advanced Cards

Determinizations	Wins	Games	Winrate
1	68	150	0.453
2	81	200	0.405
3	32	100	0.32
4	39	100	0.39
5	35	100	0.35

Figure C.7: DMCTS vs HBF, Advanced decks, $B = 0.5$, $T = 5000$

Determinizations	Wins	Games	Winrate
1	52	100	0.52
2	44	100	0.44
3	39	100	0.39

Figure C.8: DMCTS vs HBF, Advanced decks, $B = 0.5$, $T = 10000$

C.5 Humans vs DMCTS

Player ID (3 responses)

1
2
3

Highest rank achieved (3 responses)

10
13
8

Usual amount of play time (3 responses)

1 hour/week
Twice a month
3-4 hours a week before the latest expansion, 0 now

What level of play did you expect from the AI before playing? (3 responses)

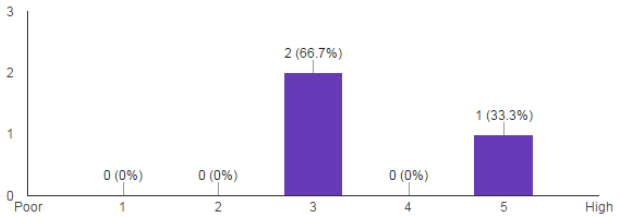
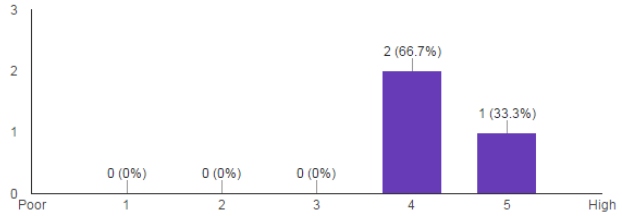


Figure C.9: Humans vs DMCTS results part 1

Comments to expected level of play (1 response)

I expected it to mostly find the optimal move for any given situation, but with some misplays here and there. I expected those misplays to cost it the game more often than it did however.

What level of play did you perceive after playing against the AI? (3 responses)



Comments to perceived level of play (1 response)

4, the AI played mostly optimal, but the AI was susceptible to high-damage flamestrikes and overextended more often than optimal

Did you enjoy playing against the AI? (3 responses)

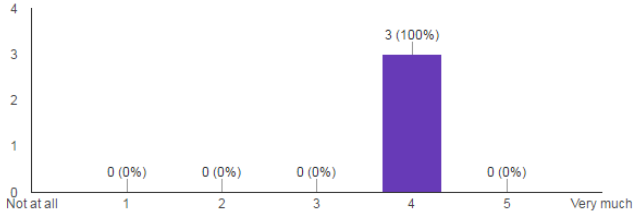
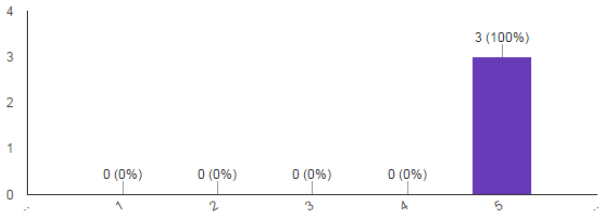


Figure C.10: Humans vs DMCTS results part 2

Comments to enjoyment (1 response)

The AI played good enough for me to have to think and play optimal to win. Could easily fall victim to a nasty Flamestrike if I was careless

Did the AI do any obvious misplays or display weird behavior? (3 responses)



Comments to AI misplays or weird behavior (1 response)

Noticed no abnormal behaviour or obvious bad plays

Did the AI play similar to a human? (3 responses)

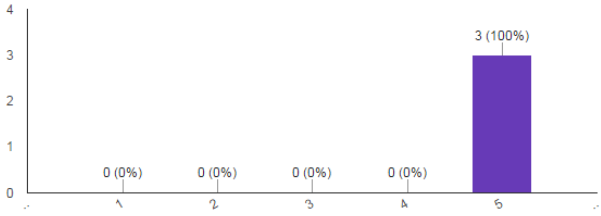


Figure C.11: Humans vs DMCTS results part 3

Comments to AI playing like a human (2 responses)

With the deck used, the computer played more or less the same as any good human would

Do you have any general feedback regarding the AI? (2 responses)

Impressive, I did not expect the AI to manage such a level op optimized play, even with such basic decks. If possible, give it a safer play-style when ahead. Better to hold back in case of Flamestrike than to throw out too much and lose a game in which it was ahead. Have it play only 5+ hp minions if Flamestrike could possibly arrive next turn when ahead.

Solid play from the AI, and good Hearthstone matches. Seem to be influenced by what player manages to snowball the early game. Although the last game was largely determined by top-decking, which is interesting because it's a pattern I've often seen in games with human players. It would also be nice if it could use a bit less time on it's initial moves, but it's not critical.

Figure C.12: Humans vs DMCTS results part 4

Appendix D

Humans vs DMCTS Questionnaire

Following is the questionnaire given to the players after playing against the DMCTS agent:

Human player vs MCTS

*Required

1. **Player ID** *

2. **Highest rank achieved** *

3. **Usual amount of play time** *

4. **What level of play did you expect from the AI before playing?** *
   Mark only one oval.

	1	2	3	4	5	
Poor	◯	◯	◯	◯	◯	High

5. **Comments to expected level of play**

Figure D.1: Human player vs MCTS page 1

6. **What level of play did you perceive after playing against the AI?** *
Mark only one oval.

	1	2	3	4	5	
Poor	◯	◯	◯	◯	◯	High

7. **Comments to perceived level of play**

_____

_____

_____

_____

_____

8. **Did you enjoy playing against the AI?** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very much

9. **Comments to enjoyment**

_____

_____

_____

_____

_____

Figure D.2: Human player vs MCTS page 2

10. **Did the AI do any obvious misplays or display weird behavior?** *
Mark only one oval.

	1	2	3	4	5	
Crazy behavior	◯	◯	◯	◯	◯	Normal behavior

11. **Comments to AI misplays or weird behavior**

_____

_____

_____

_____

_____

12. **Did the AI play similar to a human?** *
Mark only one oval.

	1	2	3	4	5	
Obviously a computer	◯	◯	◯	◯	◯	Might as well be a human playing

13. **Comments to AI playing like a human**

_____

_____

_____

_____

_____

Figure D.3: Human player vs MCTS page 3

14. **Do you have any general feedback regarding the AI?**

_____

_____

_____

_____

_____

Figure D.4: Human player vs MCTS page 4