Universiteit Leiden
Opleiding Informatica

Learning to Play Hearthstone

Using Machine Learning

Name:	Frank van Rijn
Date:	26/10/2016
1st supervisor:	Prof. dr. Aske Plaat
2nd supervisor:	Dr. Walter Kosters

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Learning to Play Hearthstone Using Machine Learning

Frank van Rijn: 0725102

October 28, 2016

Abstract

The subject of this thesis is a new game called Hearthstone. It is a strategy card game developed by Blizzard Entertainment, in which players duel with each other with cards they collected. The game of Hearthstone provides a challenge for developing an artificial intelligence (AI) agent. The agent has to be able to deal with unknown information and stochastic events in a large search space. In this thesis four different strategies are explored to create such an AI agent for playing the game of Hearthstone; a simple random bot, a rule-based bot, a Monte Carlo bot and a Monte Carlo Tree Search bot are implemented and compared with each other.

Contents

1 Introduction **4**
- 1.1 Topic . . . 4
- 1.2 Problem Statement . . . 4
- 1.3 Legitimization . . . 5
- 1.4 Method . . . 5
- 1.5 Contributions . . . 5
- 1.6 Structure . . . 6

2 Literature **7**
- 2.1 MCTS . . . 7
- 2.2 Determinization . . . 8

3 Hearthstone **12**
- 3.1 Example Game . . . 14

4 Bots **18**
- 4.1 Random Bot . . . 18
- 4.2 Rule-based Bot . . . 18
- 4.3 Monte Carlo Bot . . . 19
- 4.4 MCTS Bot . . . 21

5 Experiments and Results **22**
- 5.1 The Number of Determinizations . . . 22
- 5.2 Cheating MCTS versus Standard MCTS . . . 23
  - 5.2.1 Extra Experiment 1: Only Minions . . . 23
  - 5.2.2 Extra Experiment 2: Adding Spells . . . 24
  - 5.2.3 Extra Experiment 3: Extreme cards . . . 24
- 5.3 Tournament . . . 25

6 Conclusion and Future Work **27**

References **31**

A Cards **33**
- A.1 Extra Cards . . . 34

B Decks **35**
- B.1 Deck with only Minions . . . 35
- B.2 Deck with Flamestrike and Deathwing . . . 35
- B.3 Ordered deck with David and Goliath . . . 36
- B.4 Deck with Spells . . . 37

1 Introduction

In the area of Artificial Intelligence (AI) games have always been an interesting area of research. Many games have been subject of research and AI strategies have been developed to play games, such as Checkers [18], Chess [6], Poker [4], Go [19] and many more.

In this thesis a new game is added to that list, namely Hearthstone [10]. Hearthstone is a new game developed by Blizzard Entertainment, who describe their game as: *"A fast-paced strategy card game for everyone. Deceptively Simple. Insanely Fun"*. It is a free to play, online, turn based, collectible card game for two players. Players play the game and acquire cards to build their deck and use these decks to play versus other players in a ranked ladder system. With the ultimate goal to reach the legendary rank. In this thesis we will look into developing AI strategies for this new game.

1.1 Topic

The object of this thesis is to investigate different AI strategies for the game of Hearthstone. We are only interested in playing the game. Deck building and acquiring cards, which are also challenging problems in the game of Hearthstone; are not a part of this thesis. Collectible Card Games such as Hearthstone, are complex games for AI strategies to deal with. Issues that arise from these type of games are: having to deal with imperfect information and having a lot of options each turn. In this thesis a first attempt is made to tackle these known issues for the game of Hearthstone.

1.2 Problem Statement

In this thesis the main research question is: *Can a computer program, known as a bot, learn to play Hearthstone?* To be able to answer this question the following research questions will be addressed:

1. Does the field of Artificial Intelligence provide algorithms for creating a Hearthstone bot?
2. Is Monte Carlo Tree Search a suitable technique for playing Hearthstone?
3. Does determinization help to deal with unknown information?

1.3 Legitimization

Hearthstone is a relatively new game, the official release was on the 11th of March in 2014. The game is a huge success and by the 22th September of 2014 Blizzard Entertainment announced that they had 20 million individual players [9]. The popularity of the game is still growing. In February 2015 it was estimated that there are already 25 million registered Hearthstone accounts. With the development of an Android version of the game and a version playable on iPhone the number of players has only increased since. To the best of our knowledge no papers on the subject of Hearthstone have been published.

1.4 Method

In this thesis Hearthstone will be investigated by using experimental research; multiple experiments are done with different bots that play Hearthstone. Four strategies will be compared with each other: A random bot, a rule-based bot, a Monte Carlo bot and a Monte Carlo Tree Search (MCTS) bot will be implemented. The question how these strategies will match up against each other will be answered.

1.5 Contributions

In this thesis an implementation of MCTS for Hearthstone is proposed and compared to other AI strategies. Hearthstone is an interesting game, not only because of its popularity, but it is also interesting from a game theoretical perspective. It is a challenging game because any AI agent playing the game has to deal with drawing random cards and unknown information, the cards the opponent is holding are not known. In the game itself there are some game modes to play against an AI agent implemented in the game, but it is not a challenge for players with experience. Only by letting the AI agent "cheat" are they able to create a difficult opponent in the single player mode. The game lets the AI agent cheat by giving the agent extra starting life and extra powerful heroic abilities. It would be a much nicer experience if the AI agent would provide a challenge by playing intelligently. Contributions of this thesis are:

- Describing the new game called Hearthstone.
- Implementing MCTS for Hearthstone.
- Showing that using multiple determinizations does not always aid the algorithm.
- Showing that the addition of spells make Hearthstone a more complex game.

1.6 Structure

The rest of this thesis is structured as follows: In Section 2 literature is discussed and in Section 3 the game of Hearthstone is explained. Then in Section 4 the functionality of the different Hearthstone bots is explained. The different experiments and their results are described in Section 5 and a conclusion is drawn in Section 6, where also possible future work is described.

This thesis is written as part of the computer science master program at the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. It is written under the supervision of Prof. Dr. Aske Plaat and Dr. Walter Kosters.

2 Literature

Hearthstone, like many other card games such as Magic the Gathering, Poker and Bridge, is a stochastic game with imperfect information. It is a two player zero-sum game: there are two people playing each other and it is either a win for one of the players or a draw. Since this is a new game there is no known evaluation function for a given board state. It is not even known if such a reliable evaluation function can be found, since only parts of the game are visible to a player. Because a useful evaluation function is lacking, techniques are needed that do not require an evaluation function, such as MCTS. In this section literature regarding MCTS is described and literature regarding determinization; an extension of the MCTS algorithm in order to deal with imperfect information is presented.

2.1 MCTS

Monte Carlo Tree Search has caused huge progress in playing strength of AI in games where there is no good evaluation function and where bots have to deal with a very large search space. The best example is the game of Go. For a long time it seemed unlikely that computer players could challenge good human Go players, because of the large search space without a good evaluation function. However, the MCTS algorithm was able to compete with human players without the need for an evaluation function [15]. Recently even a professional Go player was defeated by a new Go playing bot using MCTS in combination with deep networks [19].

MCTS uses four steps to build up a partial subtree and explore promising areas of the search space. The four steps are selection, expansion, simulation and backpropagation. In the selection stage, starting from the root a new node is selected, using the Upper Confidence Bound for Trees (UCT) [14] algorithm until a leaf node in the Tree is reached. UCT algorithm selects the node that maximizes:

$$\bar{x_j} + \sqrt{\frac{2 \ln n}{n_j}} \tag{1}$$

where $\bar{x_j}$ is the average reward of child j gained by dividing the wins of child j by the number of times it has been visited; n is the total number of simulations so far and n_j is the number of times child j has been selected. This selection is repeated until a leaf node is selected. When a leaf node is selected using the UCT algorithm this node is expanded. If it is a terminal node this means nothing needs to be done and the result can be backpropagated. If it is a non-terminal

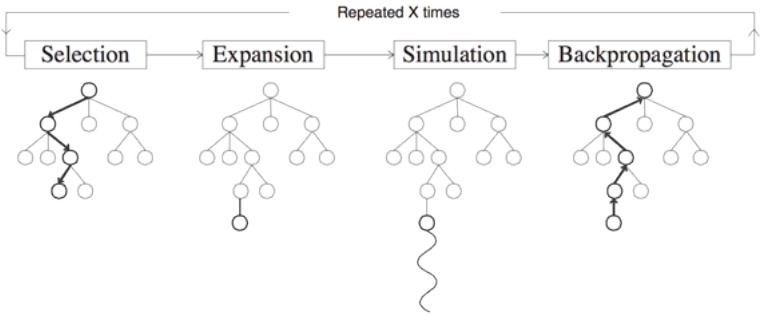

Figure 1 – Scheme of the Monte Carlo Tree Search algorithm [7].

node its children are added to the tree and the next phase is entered, the simulation phase. From the nodes added in the previous step a random playout is done and the result is being backpropagated in the next step. From the newest nodes added to the tree the result of the simulation is being updated upward in the tree. Each (grand)parent node of the added nodes gets an extra visit and if the playout is a win from that nodes perspective, the number of wins is also updated. In Figure 1 the four steps of the Monte Carlo Tree Search algorithm are shown.

2.2 Determinization

This standard version of MCTS is not directly applicable to games with imperfect information, since decisions have to be made for both players while constructing the game tree. Dealing with imperfect information has always been a problem for artificial agents. One way to deal with imperfect information is to take all the possible moves into account. However, for Hearthstone that is not feasible. Currently there are 889 different cards. Without taking into account other moves, even playing a single card that the opponent is holding would already result in a branching factor of 889. To circumvent this we assume the content of the opponent's deck is known. This brings the amount of possible cards back to 30 (the number of cards in the deck of the opponent). This looks like a big assumption, but in reality it is not that big. Since most good decks are shared on the internet it is possible to know what deck the opponent is using after seeing a couple of cards. There is a tool built for predicting the opponent deck using machine learning [5], which shows very good results. It achieves 97% successful prediction rate after seeing only three cards. Even with the information about the content of the deck of the opponent, considering all options results in quite a high upper bound branching factor. Imagine

a mid-game where both players have three minions and the opponent has one card in hand. This gives the opponent 12 possible attacks, adding 30 possible cards to play, plus ending the turn and using hero power makes a branching factor of 44 possible moves. Having such an high branching factor leads to very undeep search trees.

Determinization is a technique that deals with imperfect information without letting the branching factor explode. The concept of determinization is that all unknown information is guessed in such a way that it resolves into a viable game state. This technique has been used on different games and combined with MCTS it is also called Perfect Information Monte Carlo (PIMC). Especially in card games PIMC has been successfully implemented, card games such as Bridge [11], Poker [12] and Magic the Gathering [8, 20] to name a few.

Determinization is not a perfect technique. It suffers from two major problems, as is explained by Frank and Basin [3], called *strategy fusion* and *non-locality*. Figure 2 depicts an example of the problem of strategy fusion. The square represents a choice for the maximizing player, while the circle represents a choice for the minimizing player. In this example there are two possible determinizations, resulting in two possible trees. A random unknown factor determines which tree represents the correct rewards at the leaf nodes. The players do not know the outcome of this random event and thus do not know which tree has the correct reward values and which tree is a wrong derminization. There is clearly a superior choice for the maximizing player in the first decision point, since choosing the right option will always result in a reward of 1, regardless of choosing the correct determinization. However the MCTS algorithm does not prefer this over the left option, because it wrongly assumes it can make the correct choice later in the tree again. However if it happens to choose the wrong determinization, the algorithm will make the wrong choice and the maximizing player will wrongly choose a leaf node with score 0.

Figure 3 shows the problem of non-locality. Again the square represents a choice for the maximizing player and the circle a choice for the minimizing player. Two possible determinizations are shown by the two trees. The minimizing player does not know which tree has the correct reward values, the left world or the right world. The maximizing player does know whether the left world or the right world is the correct world. If the maximizing player makes the left move, and thus the next move is determined by the minimizing player, the minimizing player does not know if he or she should choose left or right since he or she does not know which world is the true world. However in reality the minimizing player can know the left world must be the correct world, since if it was the case that the right world is the correct

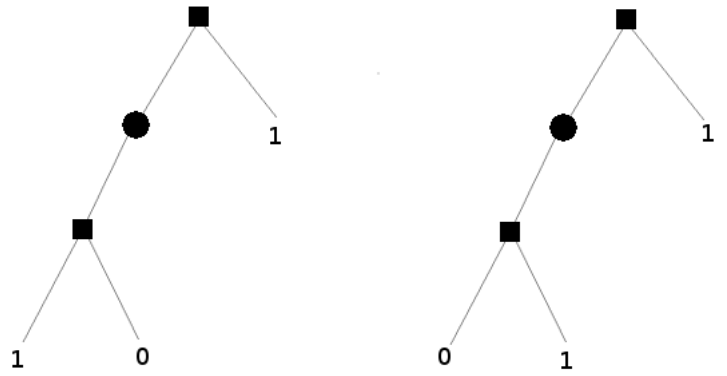

Figure 2 – An example of strategy fusion; MCTS does not prefer the right option over the left option.

world, the maximizing player would have chosen the right option to get a guaranteed outcome of 1. The problem lies in the fact that the algorithm makes choices based upon a certain game state, not taking into account the game history. MCTS will not detect this kind of situation and thus the algorithm will be wrong in some cases where it could have known what the correct play would have been.

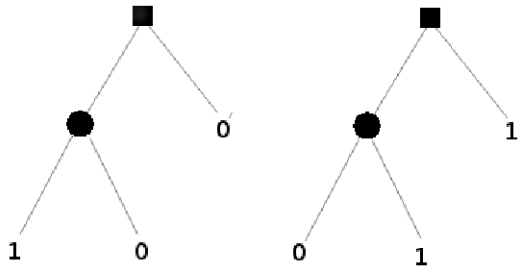

Figure 3 – An example non-locality; MCTS does not detect it can make the correct decision based on other parts of the search tree.

Even though there are theoretical shortcomings to the algorithm, it yields strong play in different games. Long, Sturtevant, Buro and Furtak try to explain in their paper [16] why this is the case and even try to predict what kind of games are suitable for MCTS with determinization. They come up with three properties that can predict whether a game is suitable for using determinizations. The three properties are:

- **Leaf Correlation**, the probability for terminal nodes with the same parent to have the same outcome.
- **Bias**, the probability the game is biased towards one player or the other.
- **Disambiguation Factor**, a factor that determines how quickly hidden information is revealed during the game.

Hearthstone has very high leaf correlation, when a player is close to winning very often there are many winning lines. The critical decisions are usually made in the mid game. Hearthstone is a little biased towards the starting player, but not by much. During the game players play cards revealing hidden information to their opponents, resulting in a relatively high disambiguation factor. These features would suggest that MCTS with determinizations is a good algorithm to play Hearthstone.

3 Hearthstone

In this section the basic rules of Hearthstone are explained.

Hearthstone is a 1 vs. 1 player card game with cards that are designed for this game, based upon the characters of the popular online multi-player game World of Warcraft. Each player chooses one out of nine possible heroes and builds a deck of 30 cards.

The game starts with both players having 30 life points and each player drawing their opening hand of three cards when starting and four cards when going second. The players get to resolve their mulligan; Each player can select any number of cards from their opening hand they wish to replace with a random card from their deck. The objective of the game is to reduce the life total of the opponent to 0 or below. The most common way this is done, is by attacking the opponent with minions. Figure 4 shows an example of such a minion. The name of the minion is shown in the middle, in this case *Bloodfen Raptor*. In the top left corner the *mana cost* of the minion is depicted by the value in the blue crystal; for the *Bloodfen Raptor* the mana cost is 2. Each minion has an *attack* value, shown in the bottom left corner, and a *health* value shown in the bottom right corner. In this thesis Attack and health value are notated inside square brackets, for example *Bloodfen Raptor* is a 2 mana [3,2] minion. The players start without mana crystals and at the start of each turn they gain a mana crystal and replenish all mana crystals used in the previous turn. This means that in a normal game a player has one mana in his or her first turn, two in his or her second, three in his or her third and so on. As the game progresses a player can play more powerful and more expensive minions, however the maximum of mana crystals a player can obtain is 10.

In addition to minions, there are also cards that are spells. These spells can, for instance, kill minions of the opponent, or gain you life, or even deal damage to opponents, such as the spell *Fireball* shown in Figure 2. Note that a spell does not stay on the battlefield after it is played and also does not have an attack and health value, its effect is resolved and then it is discarded.

At the start of each turn the player draws a card and then he or she can play out cards from his or her or her hand. He or she can attack with minions, however minions can only attack if they were in play at the beginning of that player's turn. This means that a minion that has just been played cannot attack, unless this minion has an ability called charge giving the minion the power to attack the turn it comes into play. There are several abilities that minions can have to grant them with extra power. A minion can also attack minions of

Figure 4 – An example of a minion: Bloodfen Raptor [1].

Figure 5 – An example of a spell: Fireball [1].

the opponent instead of directly attacking the opponent. The choice to attack certain minions or attack the opposing hero is one of the important decisions that a player has to make in the game.

Some minions have a keyword on them: these keywords grant minions an extra effect. The keywords used on the cards that are used in this thesis are:

- taunt
- charge
- battlecry

Taunt forces the opponent to first attack the minions with taunt. It is not possible to attack a non-taunt minion while the opponent has a taunt minion on the battlefield. Charge grants a minion to be able to attack the turn it comes into play, while normal minions have to wait a turn before being able to attack. Minions with battlecry have an effect when they are played. The effect is explained on the card itself. For example the card *Gnomish Inventor* has battlecry: Draw a card. So when you play *Gnomish Inventor* from your hand you draw a card.

Each of the nine possible heroes has a hero power which can be used by each player once during his or her turn for two mana. Some examples of hero powers are shown in Figure 6.

Figure 6 – Examples of hero powers of different heroes; in order from left to right: hunter, warlock, paladin and mage [1].

3.1 Example Game

Figure 7 – Screenshot of the game; it is the first turn of the hunter, a *Murloc Raider* has been cast.

In this Section an example game is shown to give a more in depth explanation of the game of Hearthstone for readers who are unfamiliar with the game. Readers who know the game already can skip this Section. In this game a hunter is playing a warlock, the game is shown

from the perspective of the hunter. The hunter is the starting player. In the first screenshot shown in Figure 7, mulligans have already been resolved and the hunter played its first minion the *Murloc Raider*. At the top and bottom in the middle of the screen the hero portraits are shown, with the life totals of the hunter and the warlock both at 30 life. At the bottom right the mana crystals are shown. This being turn one the hunter has access to one mana, which has been used to play the *Murloc Raider*. At the top above the warlock portrait, the mana crystals of the warlock are shown. Since the warlock has not had any turn it has 0 mana crystals. The backside of cards the warlock is holding are also visible. At the bottom under the hunter portrait the hand of the hunter is shown who is now holding *Bluegill Warrior, Gnomish Inventor* and *Ironfur Grizzly*. The only possible move left is the end turn button, which becomes green because there is no other move left.

Figure 8 – The beginning of the second turn of the hunter.

In Figure 8 the start of the turn of the hunter is shown. The hunter has drawn a card for this turn, namely *Frostwolf Warlord*, and has gained an extra mana crystal. He now has two out of two mana available. The warlock has used his first turn to use *The Coin* to cast an *Acidic Swamp Ooze*. On the left side there is a small log of the game.

All three cards played up to this point are shown there. The hunter has different options for this turn. Notice that the *Bluegill Warrior* in hand and the *Murloc Raider* on the board have a green border, this is because they respectively can be played and attack. The end turn button is now yellow and not green since the hunter has other options. The hero power of the hunter depicted at the right of the hunter portrait is also green, because it can also be used for two mana.

Figure 9 – The second turn of the hunter, the *Murloc Raider* is attacking the *Acidic Swamp Ooze*.

In Figure 9 the second turn of the hunter is played out. The hunter chose to play the *Bluegill Warrior* and since it has charge it can attack immediately. It opted to attack the opponent and now the Warlock is at 28 life. The *Murloc Raider* is about to attack the *Acidic Swamp Ooze*, which will result in both minions dying and being removed from the board.

Figure 10 – The final turn of the hunter, the hunter can put his opponent at 0 life and win the game.

In Figure 10 the game has been fast forwarded eighth turns now being turn ten of the hunter. The hunter has a total of twelve attack power and the warlock is currently at 14 life. The twelve attack plus two damage from the hero power of the hunter is enough to win this game. Thus the game will result in a win for the hunter, barring that the hunter player notices this and does not attack the minions of the warlock. Notice that the [5,4] *Bootybay Bodyguard* has a shield around it, this how the game visually shows that the minion has taunt.

4 Bots

In this section the four bots, the random bot, rule-based bot, the Monte Carlo bot and the Monte Carlo Tree Search bot and their workings are explained.

4.1 Random Bot

The first and most simple AI implementation is the Random bot. The Random bot finds all possible moves and randomly selects one of these moves. Every possible move has the same probability of being selected. This type of strategy does not require any domain knowledge of the game, which makes it easy to implement for new games. Random play usually does not result in strong performance, it does however function as a benchmark to test other strategies. Any useful strategy should outperform the random player. The Random bot is also used in other strategies. The algorithm for the random bot is shown in Algorithm 1.

Algorithm 1 RANDOM BOT

Input (Game G)

1: Find all legal moves $M = \{m_1, m_2, ..., m_k\}$ in G
2: Uniform randomly select a number i from $[1, k]$
3: **return** m_i

4.2 Rule-based Bot

The second algorithm is the Rule-based bot. The Rule-based bot is implemented by using domain knowledge to derive rules for the bot to follow.

The first rule is to check if there is a sequence of moves that defeats the opponent directly. This is done by checking if the opponent has no taunt minion and if the bot has enough attack power to defeat the opponent. If this is the case, the first move that attacks the opponent directly is selected. If that is not the case, the bot proceeds to the next rule.

The algorithm tries to find what is called a good trade. It tries to make an attack in which the opponent loses more than the bot does. A good example is attacking with a *Murloc Raider* into a *Bloodfen Raptor*. You lose 2 points of attack and 1 point health on your minions, but the opponent loses 3 attack and 2 points of health (see appendix A).

If there are no good trades available the next rule is to attack the opponent directly, unless the opponent has a taunt minion in which case the taunt minion is attacked. If no attacks are possible, the algorithm tries to play a minion, because in most cases playing a minion is better then not playing a minion. When none of the above rules led to the selection of a move, a random move is selected in the same way as for the Random bot. In Algorithm 2 the working of the Rule-based bot is shown.

Algorithm 2 RULE-BASED BOT

Input (Game G)

Find all legal moves $M = \{m_1, m_2, ..., m_k\}$ in G
$attval \leftarrow \sum$ attack value of minions
if ($attval > Oppolife$ **and not** Taunt) **then**
  select $m_i \in M$, where m_i attacks opponent
  return m_i
end if
for all $m_i \in M$ **do**
  if (m_i = value trade) **then**
    return m_i
  end if
end for
for all $m_i \in M$ **do**
  if (m_i = attack opponent) **then**
    return m_i
  end if
end for
for all $m_i \in M$ **do**
  if (m_i = play a minion) **then**
    return m_i
  end if
end for
$m_i \leftarrow Randombot(G)$
return m_i

4.3 Monte Carlo Bot

The third algorithm is the Monte Carlo bot. The Monte Carlo bot works by random sampling of game playouts to determine the best move. The algorithm has a budget, b, which determines the number of sample games the algorithm will play out. First, the bot finds all possible moves; suppose there are m of these possible moves. For each move b/m random playouts are executed. The move with the highest number of wins is most likely the best move and therefore chosen as the next move. Before the random playouts, the game-state needs to be randomized, or else the algorithm could learn information that is not available to the algorithm, for example, the next card that is drawn, or the cards the opponent is holding in its hand. During the playout of the game these unknown cards will influence the win percentages of the possible plays of the bot. This would be an unfair advantage because the bot has no way to know what the opponent is holding or what cards the bot is drawing next.

Randomizing the game state is done by shuffling the deck of the bot, then the cards the opponent is holding are put back into his or her deck. The deck of the opponent is shuffled and then the opponent draws as many cards as it was holding before randomizing the game state. The algorithm is shown in Algorithm 3.

Algorithm 3 Monte Carlo bot

Input (Game G, Integer b)

Find all legal moves $M = \{m_1, m_2, ..., m_k\}$ in G
for $i = 1$ **to** k **do**
  $G_c \leftarrow Copy(G)$
  Do move m_i on G_c
  for $\ell = 1$ **to** b/m **do**
    Do random playout of G_c using Random bot
    if win for current player **then**
      $win_i \leftarrow win_i + 1$
    end if
  end for
end for
Select move $m_i \in M$ with highest win_i
return m_i

4.4 MCTS Bot

The fourth algorithm is the Monte Carlo Tree Search bot. The MCTS bot works by constructing a search tree to find the best move. The algorithm has two parameters the budget, b; and the number of determinizations, d. First, the algorithm uses determinization to get rid of the random and the unknown information. Each determinization works the same as the randomization for the MC bot, except that for MCTS with determinizations a tree is constructed and for each determinization a new randomized state is created, with its own search tree. For all determinizations a search tree is build using the four steps of the MCTS algorithm. Recursively a leaf node is selected using the UCB algorithm. This leaf node is expanded (unless it is a terminal node), by doing a random playout for each possible move from that point. The new nodes are added to the game tree as leaf nodes. The result of the playout is then backpropagated through the tree, updating the amount of visits each node has had and the amount of winning playouts.

Algorithm 4 MCTS BOT

Input (Game G, Tree T, Integer b, Integer d)

for $j = 1$ **to** d **do**
  Game $H \leftarrow Determinize(G)$
  Find all legal moves $M = \{m_1, m_2, ..., m_k\}$ in G
  for $\ell = 1$ **to** b/d **do**
    Select(node) using UCB
    if selected node = leafnode **then**
      expand(T)
      simulate random playout using Random bot
      backpropagate result updating nodes in T
    end if
  end for
end for
Select move $m_i \in M$ with most visits
return m_i

5 Experiments and Results

In this section the different experiments that are conducted and their results are explained. Note that for this research only a subset of the complete Hearthstone game is implemented, to keep the programming task of recreating the game feasible. In the version of Hearthstone implemented only two out of the nine existing heroes are implemented; those two heroes are the hunter and the warlock.

The hero power of the Hunter is *Steady Shot*, for two mana the hunter deals two damage to the opponent. The hero power of the Warlock is *Life Tap*, for two mana the warlock deals two damage to itself and it draws a card. These specific powers were chosen because they are considered the best hero powers in the game. The warlock is strong for generating card advantage and the hunter power is strong for allowing a player to deal the final damage without having any minions on the board.

5.1 The Number of Determinizations

For the MCTS algorithm with determinizations, the parameter d, which determines the number of determinizations, is important for the performance of the algorithm. An experiment was set up to find a good value for this parameter. The algorithm was run for different values of d, namely $1, 5, 10, 15, 20, 30$ and 40. For each of those values of d two hundred playouts against all other values of d were done; first hundred times being the starting player and then a hundred times letting the other value start. This was done four times for each pair of hunter and warlock (hunter vs. hunter, hunter vs. warlock, warlock vs. hunter and warlock vs. warlock). Note that the pairs having same values of d, for instance $d = 5$ vs. $d = 5$, are omitted. This leads to 16800 played games. For this experiment a simple deck was used with only minions, that can be found in Appendix B.1. The results were added and are shown in Figure 11.

The lines show the number of wins versus the MCTS version on the horizontal axis, the legenda shows which version of MCTS is represented by which line.

The red line is the performance of MCTS with 1 determinization versus the other values of d. As can be seen the performance improves as it plays against higher values of d. This leads to the conclusion that using multiple determinizations leads to worse performance, which is a very unexpected result. To understand this a series of extra experiments are conducted.

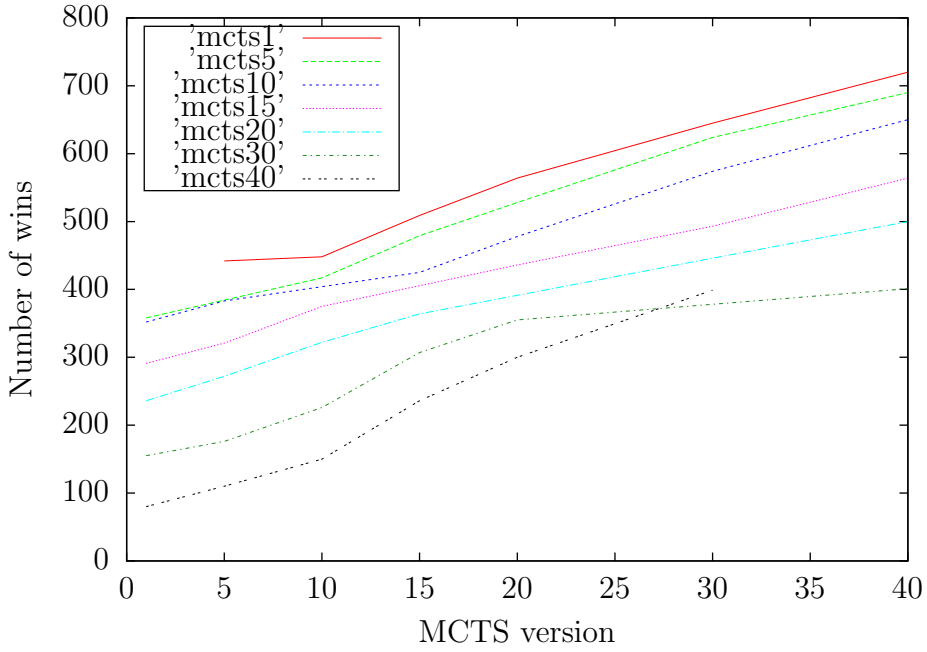

Figure 11 – Graph showing the results how Determinizations influence the win-percentages of the MCTS algorithm

5.2 Cheating MCTS versus Standard MCTS

In these extra experiments two versions of Monte Carlo Tree Search were tested against each other; a standard MCTS version with one determinization to deal with the imperfect information(MCTS-D) versus MCTS with perfect information (MCTS-PI), basically a cheating version of MCTS that can see all the cards. The hero class *Hunter* was used for all experiments for both players. Three different decks were used during the three extra experiments to understand how the cards used have an effect on the performance of the algorithm.

5.2.1 Extra Experiment 1: Only Minions

For this experiment the same simple deck, with only minions, as for the previous experiment was used, see Appendix B.1 for a complete list. Thousand games were played where MCTS-D was the starting player and another thousand games where MCTS-PI was the starting player. This resulted in an expected win ratio of 0.5375 for MCTS-PI. This was a bit unexpected, intuitively a larger percentage of the games should be won by MCTS-PI; Being able to see all the cards should give a player an unfair advantage. The non-cheating version

could still win over 50% of the games when it starts, winning 525 of the thousand games. In conlusion; starting is a larger advantage then being able to see all the cards.

5.2.2 Extra Experiment 2: Adding Spells

A second experiment was conducted. Some minions were removed and spells were added to create more complexity. The idea is that minions are too much alike. Thus, the advantage of knowing all the cards is almost non-existent, because they almost all do the same thing. The *Archmages* were removed from the deck and replaced with higher impact cards. The card *Flamestrike* was added. *Flamestrike* is a spell that deals four damage to all enemy minions. Another card called *Deathwing* was also added. *Deathwing* is an eight mana [7,6] minion that has a battlecry effect. When you cast it, you must discard your hand and it destroys all other minions on the battlefield. The idea behind the addition of these specific cards is that it is very useful to know whether the opponent is holding these cards. In case the opponent is holding cards that can destroy multiple minions, it might be better to not play all your minions and hold some in hand. The hypothesis is that this would give an advantage to MCTS-PI. The decklist used can be seen in the Appendix B.2.

The result is a win-ratio for MCTS-PI 0.546. It does indeed increase the win percentage of the cheating MCTS version, but maybe not as much as expected. In conclusion, even now starting is a bigger advantage than looking at the cards, MCTS-D wins 555 of the games on the play.

5.2.3 Extra Experiment 3: Extreme cards

A third experiment was done to see if MCTS-PI would win in an extreme situation where looking at cards clearly should give the algorithm the upper hand. A card called *Goliath* was created. Goliath is a two mana minion that is [15,10], which is completely absurd for the game. For comparison, a two mana minion is usually [2,3] or [3,2]. Also another card called *David* was added to the deck. *David* is a spell that checks if the opponent has *Goliath* in play and if so you win the game, if not you deal five damage to yourself. Knowing whether the opponent does or does not have *David* in hand clearly changes the outcome if you should play *Goliath* or not. For this experiment the decks were ordered and *not* shuffled to ensure that both players had access to both *David* and *Goliath*. The decks can be found in the Appendix B.3.

Now MCTS-PI has a win-ratio of 1.0, which is exactly what would be expected. The cheating version can see that the opponent is holding *David* and thus will never play *Goliath*, while the non cheating version has to make a determinization and since these determinizations are random it will eventually make the mistake in thinking that MCTS-PI is not holding *David*. If the opponent is not holding *David*, playing *Goliath* is a very strong move and will be chosen as the next move. Resulting the MCTS-PI playing *David* and winning the game.

5.3 Tournament

In this experiment the four bots are matched up against each other and play a thousand games against each other on the play and on the draw. All the bots use the same hero, namely the *Hunter* and they use the same deck with spells. The exact decklist can be found in Appendix B.4. The results are shown in Table 1. Each histogram shows the results of one of the four bots. The red part shows the number of wins while starting. The green part shows the number of wins while drawing. Note that each bot also plays thousand games against itself and always wins those thousand games.

As Table 1 shows, the performance of the random bot is really bad, which was to be expected. Playing randomly is a very poor strategy in this game. It manages to get a few wins against the rule-based bot, but out of two thousand games against both MC and MCTS it does not manage to win a single game. The rule-based bot is already an improvement over the random bot, almost winning all the games versus the random bot and winning some games against the MC and MCTS bots. Again the MC bot is much better than the rule-based bot, with a performance that is almost as good as the MCTS bot. In almost all the cases the MCTS bot is slightly better then the MC bot, except versus the rule-based bot on the draw it achieves the same amount of wins or better then the MC bot.

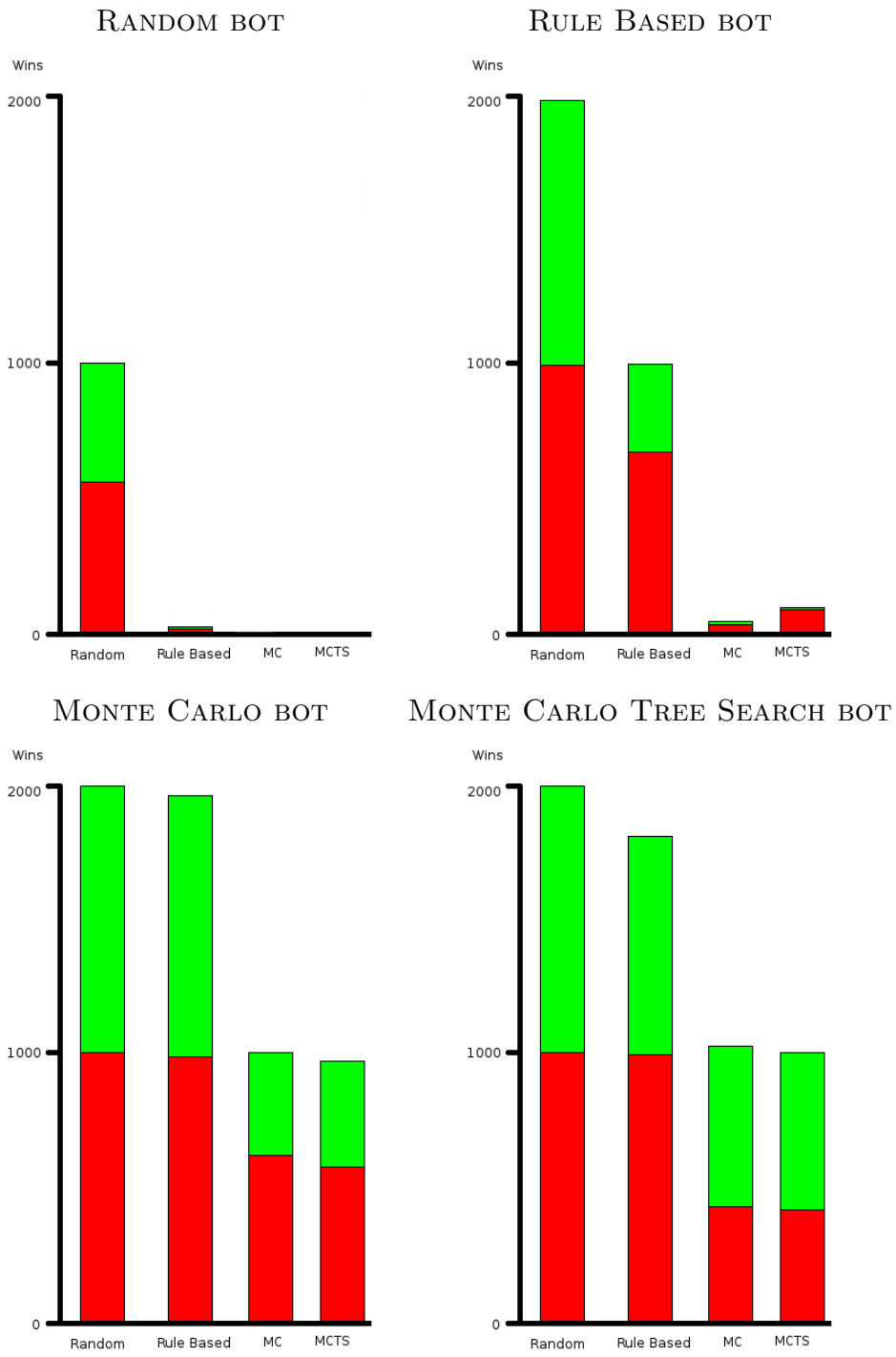

Table 1 – The performance of the four bots playing against each other.

6 Conclusion and Future Work

In this thesis the game of Hearthstone was investigated. In this section the research questions are answered and a conclusion is drawn. The first question as proposed in Section 1 is:

> Does the field of Artificial Intelligence provide algorithms for creating a Hearthstone bot?

If a game is too straightforward it does not make sense to use complex algorithms to solve the game of Hearthstone. For instance the game of Tic-Tac-Toe is very simple to solve and can be easily brute forced. Such trivial games are not very suitable for the use of machine learning algorithms. Hearthstone does provide a challenging problem space. The game is an imperfect information, stochastic zero-sum combinatorial game. Using a brute force algorithm, calculating all possible moves and their outcome, is definitely not a possibility. Finding the correct play is not trivial and since brute forcing is not possible, heuristic algorithms are necessary to find good plays for any Hearthstone bot. For this game there is no known evaluation function, nor is it trivial to find such a function, which limits the amount of useful algorithms. The game of Go has struggled with this issue, being a combinatorial game without a known evaluation function. A strong playing bot was thought to be many years away. The development and the use of Monte Carlo Tree Search was a huge breakthrough in the playing strength of Go bots, because it was able to deal with the size of the problem space of Go without the use of an evaluation function. This leads to the next research question:

> Is Monte Carlo Tree Search a suitable technique for playing Hearthstone?

As is shown in Section 2 standard MCTS can not directly be used to play Hearthstone. Standard MCTS is not designed to deal with imperfect information and with stochastic uncertainty. Not knowing what cards your opponent is holding or what cards you are going to draw makes it impossible for the algorithm to build a search tree. The solution is to create determinizations, just create a possible game state in which the algorithm does know everything. Most likely the determinized game state is not correct, but now MCTS can be used to play Hearthstone. So in conclusion, MCTS with determinizations is suitable for playing Hearthstone. The last research question is:

Does determinization help to deal with unknown information?

To be able to use MCTS for a game with imperfect information, some technique is necessary to deal with imperfect information. In some games it is possible to circumvent the use of determinizations by investigating all possible moves, for example this is done in the game of Scotland Yard [17]. Scotland Yard is a board game in which a group of players, representing the police, try to catch a criminal in London. The location of the criminal is the unknown factor. However, the number of possible locations where the criminal could be is limited. For the game of Hearthstone this is not possible, because the number of possible moves grows exponentially with the number of cards the opponent is holding, making it infeasible to investigate all possible moves. Thus in the game of Hearthstone the use of determinizations is very useful and does help to deal with the imperfect information.

The use of determinizations adds another parameter to the algorithm, namely the number of determinizations used. In Section 5 experiments were conducted to find the best value for this parameter and interestingly it was found that the best number was only one determinization. A series of extra experiments were conducted to investigate why this was the case. It was shown that being able to see all unknown information was not a great advantage in the simple version of Hearthstone. This could explain why one is the best number of determinizations, since using more determinizations is not free. For each determinization a new search tree is built and each tree has to share the budget (usually the budget is the number of total random playouts). The idea is that using more determinizations the best move is identified more often, since using only one determinization it could be that the guessed game state is an extreme game state that is far from the real game state. By using more determinizations the average best move is chosen and extremes are filtered out. Since having perfect information only helps a little in identifying the correct best move, spending budget on getting better information is not wise and just using more budget in constructing a bigger search tree results in better play. This could be true other games as well.

Adding complexity to the game in Hearthstone is done by adding spells to the game. They change the interaction in the game, a player being ahead on the board can easily be set back by spells. This is much harder to accomplish with only minions, since they cannot attack immediately when they enter the board. The opponent can attack those minions first and thus has the opportunity to make attacks that favor that player.

The main research question as proposed in Section 1 is:

> Can a computer program, known as a bot, learn to play Hearthstone?

In this thesis four strategies to play the game of Hearthstone were implemented and compared to each other. A random bot was created to be a very simple beatable strategy for other bots to compete against. It was expected that all other strategies would easily be able to beat the random bot. The second strategy was the rule-based bot; a bot that follows rules, made up by mimicking the train of thought of a human player playing the game. This bot could indeed easily beat the random bot. The next bot used a more intelligent strategy, namely Monte Carlo, which uses random sampling to evaluate different possible moves. This bot outperformed the random bot and the rule-based bot. The last bot was implemented using a more sophisticated strategy, namely Monte Carlo Tree Search. MCTS is a state-of-the-art algorithm and was expected to dominate all the previous strategies. It does indeed beat the random and the rule-based bot by a fair margin, however it does not dominate the performance of the Monte Carlo bot. The Monte Carlo bot does perform better against the rule-based bot. Between the two bots MCTS does win more than half of the games, but not as many as expected.

In conclusion the answer is yes, the computer can indeed learn to play Hearthstone. The random bot, which is the benchmark of a player who cannot learn the game, is easily beaten by all strategies. The rule-based bot could be seen as the equivalent of a novice player, and the MC and MCTS bot can definitely beat the rule-based bot. The MCTS bot in the current version could be seen as an advanced player, but not of the strength of an expert Hearthstone player. Improvements have to be made to get to the level of human expert players, which leads to the future work.

For future work there are many possibilities, exploring strategies in Hearthstone is not close to finished. For starters strategies in different areas of Artificial Intelligence come to mind. Natural computing techniques like genetic algorithms and neural networks and deep learning networks can come up with totally different solutions for playing this game. For a genetic algorithm an evaluation function would be needed, which might prove to be complex to formulate, but is not infeasible. For neural networks large data sets of games are needed, which are not available now. The game does not automatically store played games. Blizzard does have access to the games that are played on their servers, but they do not release this valuable data.

The rule-based bot proposed in this thesis is not a very refined version of a rule-based bot. With using more expert domain knowledge a better performing version of a rule-based algorithm is definitely a possibility.

The Hearthstone version used in this thesis should be extended to a full version of Hearthstone, if bots would like to compete against real players. At this point not all cards and mechanics are implemented, which makes it impossible to use the bot in the real game. The full version of Hearthstone also leads to the use of more spells and minions with more advanced mechanics which could add to the complexity of the game. As has been shown, adding complexity to the game helps the more advanced algorithms outperform the more straightforward algorithms. With a full implementation of the game, bots could be tested in the real game. Every month there is a ladder ranking system which lets players climb from rank 25 to rank 1 and if enough games are won at rank 1 a players reaches the legendary status. Each season around 0.5% of the players reach this rank. This would be a great achievement if a bot can be developed that reaches this rank consistently.

The game provides a real challenge for AI strategies and a Hearthstone league or competition would be a very nice platform for different algorithms to compete against each other, similar to world chess championships [13].

Another possible improvement could be opponent modeling. By using inference to make better determinizations and stick to those instead of using random ones every time. Now the bot has the tendency to play around cards for one turn and forget about it the next turn, and stop playing around it, resulting in inconsistent play. Adding opponent modeling to the MCTS algorithm could improve the algorithm. Another possible improvement for the MCTS algorithm could be the use of multithreading, especially when the algorithm uses multiple determinizations, the use of multiple threads could really speed up the algorithm allowing for more random playouts in the same time.

References

[1] http://hearthstone.gamepedia.com/Hearthstone_Wiki.

[2] http://hearthcards.net.

[3] Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1):87–123, 1998.

[4] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Poker as a testbed for ai research. In Robert E. Mercer and Eric Neufeld, editors, *Advances in Artificial Intelligence: 12th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI'98 Vancouver, BC, Canada, June 18–20, 1998 Proceedings*, pages 228–238, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[5] Elie Bursztein. Predicting Hearthstone opponent deck using machine learning. https://www.elie.net/blog/hearthstone/predicting-hearthstone-opponent-deck-using-machine-learning, 2015. Accessed:18-02-2016.

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

[7] Guillaume Chaslot, Mark H.M. Winands, H. Jaap van den Herik, Jos W.H.M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04(03):343–357, 2008.

[8] Peter I. Cowling, Colin D. Ward, and Edward J. Powley. Ensemble determinization in Monte Carlo Tree Search for the imperfect information card game Magic: The Gathering. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(4):241–257, 2012.

[9] Blizzard Entertainment. 20 million Hearthstone players. http://eu.battle.net/hearthstone/en/blog/15982639/20-million-hearthstone-players-22-09-2014. Accessed:06-04-2015.

[10] Blizzard Entertainment. Hearthstone: Heroes of Warcraft official game site. http://us.battle.net/hearthstone/en/, 2013. Accessed:31-03-2015.

[11] Matthew L. Ginsberg. GIB: Imperfect information in a computationally challenging game. *CoRR*, abs/1106.0669, 2011.

[12] Johannes Heinrich and David Silver. Self-play Monte-Carlo Tree Search in computer poker. 2014.

[13] ICGA. World computer software championship. http://icga.leidenuniv.nl. Accessed:28-07-2016.

[14] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[16] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information Monte Carlo sampling in game tree search. 2010.

[17] J (Pim) AM Nijssen and Mark HM Winands. Monte-Carlo Tree Search for the game of Scotland Yard. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 158–165. IEEE, 2011.

[18] Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Akihiro Kishimoto, Rob Lake, Paul Lu, Steve Sutphen, and Martin Muller. Solving checkers. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 292–297. Morgan Kaufmann Publishers Inc., 2005.

[19] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

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

A Cards

The cards that are used in the experiments for this thesis.

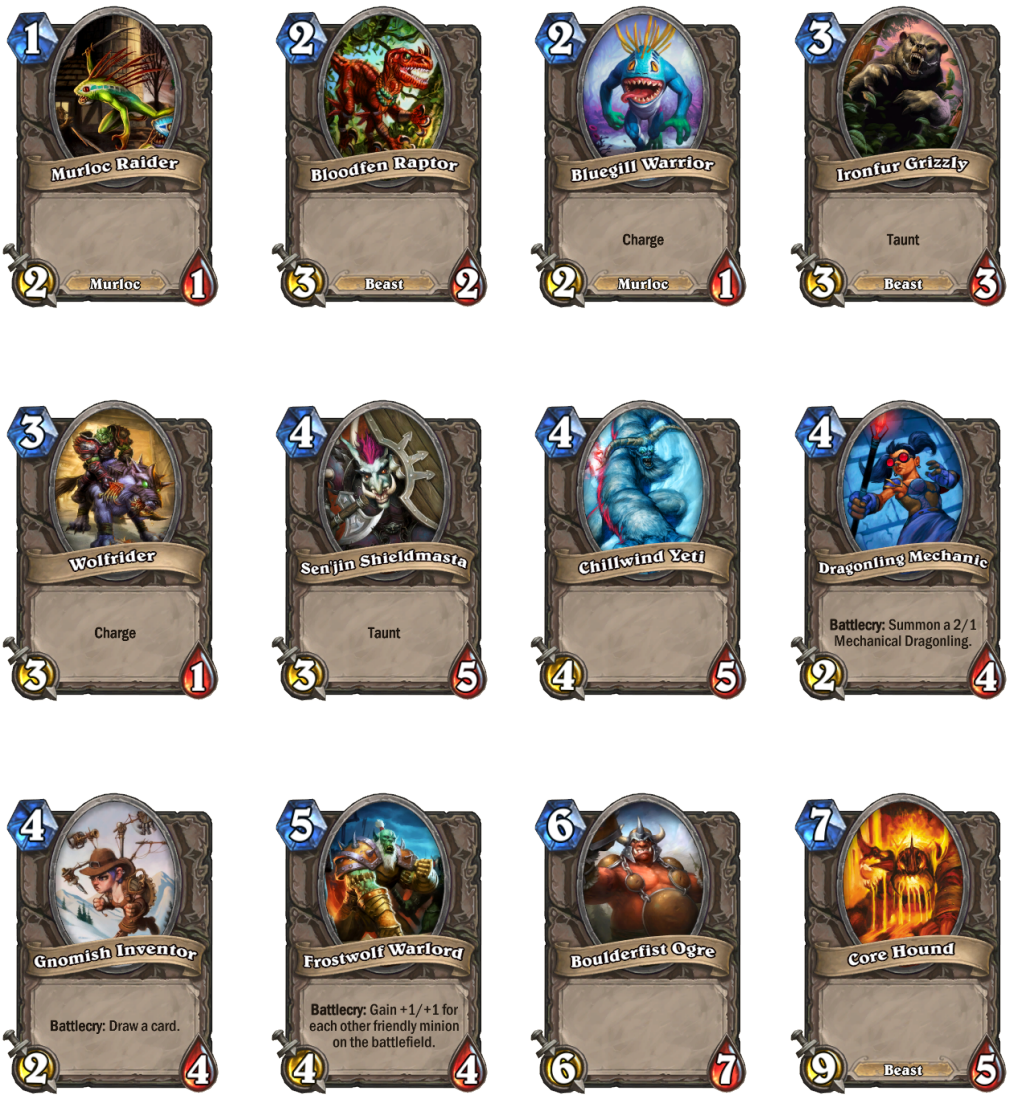

Figure 12 – The existing cards used in all experiments in this thesis [1].

A.1 Extra Cards

The cards that do not exist in the full version of Hearthstone, but were implemented for this thesis. *Deathwing* is an existing card in the full game, but in the full game it is a 10 mana [12,12] minion. Its mana cost was lowered to have more impact during the experiments, because sometimes games are ended before players reach 10 mana. The attack and health values were also altered to keep the card balanced, the battlecry effect is unchanged from the original card. The cards *David* and *Goliath*, were created specially for experiments and do not exist in the full version of Hearthstone.

Figure 13 – Cards that were created to do extra experiments [2].

B Decks

All the decklists of the decks that are used for the experiments in this thesis.

B.1 Deck with only Minions

A simple deck with only minions, similar to what a beginning player would use. This deck is used for the experiments in Sections 5.1 and 5.2.1.

```
2 Murloc Raider
2 Acid Swamp Ooze
2 Bloodfen Raptor
2 Bluegill Warrior
2 Iron Grizzly
2 Wolf Rider
2 Senjin Shield Masta
2 Chillwind Yeti
2 Dragonling Mechanic
2 Gnomish Inventor
2 Frostwolf Warlord
2 Bootybay Bodyguard
2 Archmage
2 Boulderfist Ogre
2 Corehound
```

B.2 Deck with Flamestrike and Deathwing

Flamestrikes and *Deathwing* are added to the simple deck. This deck is used for the experiments in Section 5.2.2.

```
2 Murloc Raider
2 Acid Swamp Ooze
2 Bloodfen Raptor
2 Bluegill Warrior
2 Iron Grizzly
2 Wolf Rider
2 Senjin Shield Masta
2 Chillwind Yeti
2 Dragonling Mechanic
2 Gnomish Inventor
2 Frostwolf Warlord
2 Bootybay Bodyguard
```

```
2 Boulderfist Ogre
2 Corehound
1 Flamestrike
1 Deathwing
```

B.3 Ordered deck with David and Goliath

This deck is used for the experiments in Section 5.2.3. The cards *David* and *Goliath* are added to the deck, in the experiments the deck is not shuffled.

```
1 Goliath
1 David
1 Deathwing
1 Flamestrike
2 Bloodfen Raptor
2 Bluegillwarrior
2 Bootybay Bodyguard
2 Boulderfirst Ogre
2 Chillwind Yeti
2 Corehound
2 Senjin Shield Masta
2 Dragonling Mechanic
2 Murloc Raider
2 Iron Grizzly
2 Frostwolf Warlord
2 Gnomish Inventor
2 Wolfrider
```

B.4 Deck with Spells

This deck is used for the tournament run in Section 5.3.

```
2 Murloc Raider
2 Bloodfen Raptor
2 Bluegill Warrior
2 Iron Grizzly
2 Wolf Rider
2 Senjin Shield Masta
2 Chillwind Yeti
1 Dragonling Mechanic
1 Gnomish Inventor
2 Frostwolf Warlord
2 Boulderfist Ogre
1 Corehound
2 Darkbomb
2 Fireball
1 Arcane Intellect
2 Flamestrike
```