TÉCNICO
LISBOA

Monte Carlo Tree Search Experiments in Hearthstone

André Miguel Leitão Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Francisco António Chaves Saraiva de Melo
Prof. Pedro Alexandre Simões dos Santos

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Francisco António Chaves Saraiva de Melo
Members of the Committee: Prof. João Miguel de Sousa de Assis Dias

June 2017

“It always seems impossible until it’s done.”

Nelson Mandela

Acknowledgments

First of all, I would like to appreciate my gratitude to my dissertation supervisors Professor Pedro Santos and Professor Francisco Melo, for their patience, inspiration and encouragement in the development of this thesis. I'm sure that without them, I probably would never ended such challenge in my life.

Also, I would like to thank my family, and in specially my Mom, for her friendship, encouragement and for caring me over in the last years.

Additionally, I would like to thank the Metastone developers, for assisting in the use of the platform, and particularly to the GitHub user `@demilich1` for all the assistance throughout the development of this thesis.

Last but not least, to all my friends and colleagues in Instituto Superior Técnico (IST), persons that helped me grow and were always there for me during the good and bad times in my life.

Thank you.

Abstract

In this work, we introduce a Monte-Carlo tree search (MCTS) approach for the game “Hearthstone: Heroes of Warcraft”, the most popular online Collectible Card Game, with 50 million players as of April 2016.

In Hearthstone, players must deal with hidden information regarding the cards of the opponent, chance, and a complex gameplay, which often requires sophisticated strategy.

We argue that, in light of the challenges posed by the game (such as uncertainty and hidden information), Monte Carlo tree search offers an appealing alternative to existing AI players.

Additionally, by enriching Monte Carlo tree search with a properly constructed heuristic, it is possible to introduce significant gains in performance.

We illustrate through extensive validation the superior performance of our approach against ”vanilla” Monte Carlo tree search and the current state-of-the art AI for Hearthstone.

Keywords

Artificial Intelligence, Monte Carlo Tree Search, Hearthstone, Game Tree Search, Search Algorithms, Collectible Card Games

Resumo

Neste trabalho é proposta uma abordagem com base nos métodos Monte-Carlo, para o jogo Hearthstone: Heróis de Warcraft, o jogo de cartas colecionáveis mais popular do momento e com mais de 50 milhões de jogadores registados em Abril de 2016.

No Hearthstone, os jogadores são continuamente posto á prova devido ao conceito de informação escondida, onde a mão do oponente é desconhecida, devido ao conceito de aleatoriedade existente, onde por exemplo as cartas são inicialmente baralhadas e devido a uma complexa jogabilidade, que muitas vezes requer uma estratégia bastante robusta e refinada.

Com o trabalho desenvolvido, argumentamos que, á luz dos desafios colocados pelo jogo (conceito de informação escondida e incerteza), a abordagem desenvolvida para o efeito oferece uma alternativa valida, face ao atual estado da arte neste domínio.

Adicionalmente, através do enriquecimento do algoritmo, mais especificamente através da introdução de informação especifica do jogo, é possível alcançar ganhos significativos de desempenho, relativamente a sua versão mais tradicional.

Palavras Chave

Inteligência Artificial, Monte Carlo Tree Search, Hearthstone, Árvore de Jogos, Algoritmos de Procura, Jogos de Cartas Colecionáveis

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

IST	Instituto Superior Técnico
AI	Artificial Intelligence
MCTS	Monte Carlo Tree Search
UCB	Upper Confidence Bound
CCG	Collectible card games
M:TG	Magic:The Gathering
GSV	Game State Value
EA	Evolutionary Algorithms

1 Introduction

Contents

The video game industry is increasingly growing. According to a new estimate from the Open Gaming Alliance, the global game software market is expected to pass 100 billion in 2019 [5]. Indeed, such success can be easily explained by today's games characteristics. They are extremely engaging, providing enjoyable and rewarding gaming experiences. In addition, games are a cultural phenomenon, being used for several purposes and even considered an interesting testbed for research and new ideas.

Due the nowadays requirements for the development of increasingly realistic games, several game companies encourage the development of games with fast and intelligent systems, using all kinds of Artificial Intelligence (AI) algorithms. Currently, AI can offer huge benefits, providing systems with intelligent behaviors, improving the challenge and experience that any game can supply.

Besides that, and especially in off-line games, the use of AI has become a necessity, covering "the behavior and decision-making process of the game-playing elements" [6]. Among the AI methods normally used, in the last years, several Monte Carlo Tree Search (MCTS) based methods emerged in the field of computer games, allowing the development of games "not be hard coded, but learned to play as a normal human" [7].

Collectible Card Games

Collectible card games (CCG) are one of the most popular forms of contemporary game play. Since the inception of Magic:The Gathering (M:TG) in the 90s—several such games emerged as popular forms of entertainment (both physical and electronic) and even training and education [8, 9].

They often-intricate gameplay allied with aspects of hidden information and chance, what also makes CCG appealing testbeds for AI research. For example, several works have explored both machine learning [10, 11] and planning approaches in CCG such as M:TG and Hearthstone [12, 13].

The former game, in particular, is the most popular online CCG, with over 50 million players as of April 2016 [14]. The game mechanics is a turn-based duel between two opponents, and can be played both in multiplayer and single player modes. In Hearthstone, players must deal with hidden information regarding the cards of the opponent, chance, a complex gameplay, which often requires sophisticated strategy, and a "wide combination of states, rules, cards that imply complex reactions" [15]. Several works in the literature have tackled different aspects of the game, such as deck building [15], card generation [10] and general game-play [16], where in the last one, the use of MCTS was clearly recommended.

1.1 Motivation and Goals

The development of Hearthstone suffered limitations in time and money. For example, the small team of 5 game developers had only 18 mounts to develop the game [17]. Such limitations, directly influenced the overall gaming experience of Hearthstone, and primarily, the development of Solo Adventures, a game mode ruled by AI and mainly developed for teaching purposes. Solo Adventures aims primarily for introducing new players to Hearthstone, and its characterized with the following features:

- The AI was designed to mimic an intermediate player. The developer's team interviewed several Blizzard employees, with different playing levels, for identifying the playing style of an intermediate player;
- The AI system rarely keeps cards in hand and prioritizes playing cards whenever is possible;
- Two difficult modes, beginner and expert;
- The difficulty levels are mainly adjusted with changes decks;
- The AI does not cheat or know the opponent's cards.

Solo Adventures, due its simplification, is challenging enough from a beginner point of view. However, the gamming community widely claims the expert AI can be beaten with the basics decks by a skilled player. Another common problem, often supported, is the rarity that the AI plays combos or keeps cards in hand intentionally. Also, some expert players mentioned that the AI developed, is not able play in medium and long term, leading to its own defeat.

So, in attempt to improve the gaming experience in Hearthstone, in this work, we develop an AI system, using MCTS methods for the card game of Hearthstone. This work, aims to improve the competitive level of the algorithm, by tacking in particular, the following problem:

Will the integration of expert knowledge in MCTS, improve the performance, regarding its "vanilla" version?

To answer such question, the approach uses a modified version of MCTS, which integrates expert knowledge in the algorithm and solves the hidden information problem presented in the game. MCTS "vanilla" version was adapted previously for Hearthstone, an adaptation that neither solved the hidden information problem or integrated expert knowledge of Hearthstone [4, 18]. Our results shown that our system is superior to "vanilla" MCTS and able to attain competitive results with state of the art AI for Hearthstone.

1.2 Contributions

The main contribution of this work consists on adapting the MCTS algorithm for Hearthstone. In the recent years, MCTS as becoming a de facto standard in game AI, being particularly suited to address the chance elements in Hearthstone [1,12,13,19]. In particular, we perform a modified version of MCTS, which integrates expert knowledge in the algorithm's search process. Such integration is done, on one hand, through a database of decks that the algorithm uses to cope with the imperfect information; and, on the other hand, through the inclusion of a heuristic that guides the MCTS roll out phase and which effectively circumvents the large search space of the game.

The heuristic represents a particular game strategy, and aims at supporting the selection and simulation process. We compare the performance of our proposed approach to that of the state-of-the-art AI for the game; by using an adequate heuristic, we are able to attain a competitive performance.

Summarizing, the contributions of this document are as follows:

- The first contribution consists in using a deck database to address the problem of hidden information in the game;
- The second and main contribution is the integration of a heuristic to handle the large search space of the game.

1.3 Thesis Outline

The rest of the document is organized as follows. Chapter 2 provides a general overview of some theoretical concepts, the game and the simulator used. Chapter 3 discusses the state-of-the art relevant for the research presented. Chapter 4 describes the details of the proposed approach. Chapter 5 describes the methodology used to fine-tune our approach, while Chapter 6 discusses the results achieved with our approach. Chapter 7 concludes the document, presenting the possible future work. Appendix A and B provides information regarding the parameters used the experiments and the Evolutionary Algorithms (EA) used to train the heuristic.

1.4 Terminology

There are terms in the literature, widely used in a variety of ways and sometimes inconsistently, leading to confusions, regarding the characteristics of MCTS. So, for the remainder of this thesis, we consider:

- ”vanilla” MCTS: the standard MCTS version, that neither integrates expert knowledge or solves the hidden information problem present in Hearthstone;
- UCB: commonly used Tree Policy;
- UCT: MCTS with Upper Confidence Bound (UCB) in the Tree Policy;

- Tree Policy: Another terminology for selection and expansion steps of MCTS;
- Default Policy: Another terminology for the simulation step of MCTS;
- Roll-out: The independent simulations performed. During the simulation stage, one or more roll-outs can be run, to estimate the current expanded node's value.

2 Background

Contents

In this chapter, its provided an examination of important theories and preliminary concepts that are imperative to understand the work performed.

2.1 Game Types

With the enormous variety of games, there is the necessity for AI development purposes, to classify them according some features. Essentially, games can be categorized in a distinction between deterministic versus stochastic and perfect information versus imperfect information (see table 2.1) [20]. Such features are explained as follows:

- **Perfect information vs imperfect information**: in games of perfect information, players can observe all the elements and features present. In board games like chess, the board and pieces are visible for both players. On the contrary, in imperfect information games, some core elements are invisible, what introduces uncertainty. Most card games are classified of imperfect information, as player's deal constantly with hidden information.

- **Deterministic vs stochastic**: in a deterministic game, the next state is solely determined by the actual state and actions taken. An example, is the game of Go, where the next position is only influenced by the player's action. On the other hand, in stochastic games random features affect the actual state. For example, in Hearthstone, the deck is initially shuffled, and some cards have random abilities, aiming primarily to affect the current outcome.

According the above features, we can mention that Hearthstone is a stochastic game of imperfect information. In Hearthstone, the player's hand is invisible, from one or another, and the game state is not solely determined by a sequence of moves of both players.

2.2 Game Tree Search

Game tree search is a technique commonly used in computer game playing [20]. Generally, a game tree is gradually created, with nodes that represents game positions and edges game movements. At the root node, the initial game state is usually represented.

	Perfect Information	**Imperfect information**
Deterministic	Chess, Checkers	Battleship, Mastermind
Stochastic	Blackjack, Monopoly	Poker, Hearthstone

Table 2.1: Game types table example

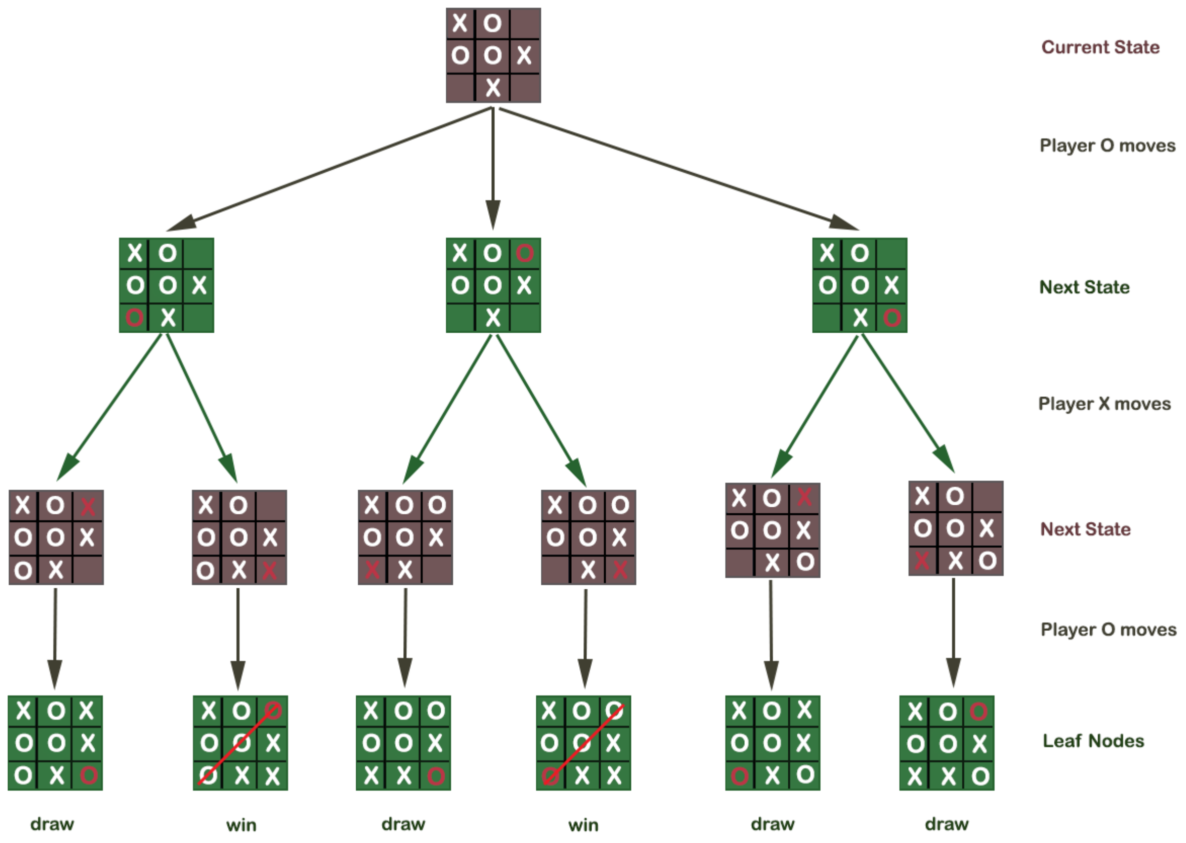


Figure 2.1: Example of a game tree in Tic Tac Toe

With game trees, not only it's possible the analysis of movements and their consequences, but also the appreciation of the new game states created from that. Actually, there are several algorithms that uses game trees in the decision-making process, in which Minimax[1] and MCTS are one of the popular ones.

Figure 2.1 illustrates an example for the game of Tic Tac Toe[2]. In the game tree, the edges represents possible moves and the boards new game states. From the current state, the 0 player can make 3 possible moves, leading to new game states. In response to that movements, the X player have 2 possible moves. The game alternates between the X and O players, until a leaf node is reached.

2.3 Monte Carlo Tree Search

MCTS is a family of search methods designed to address sequential decision problems . MCTS methods rely on sampling to handle both large branching factors (as observed in games such as Go

[1] `https://en.wikipedia.org/wiki/Minimax`
[2] `https://en.wikipedia.org/wiki/Tic-tac-toe`

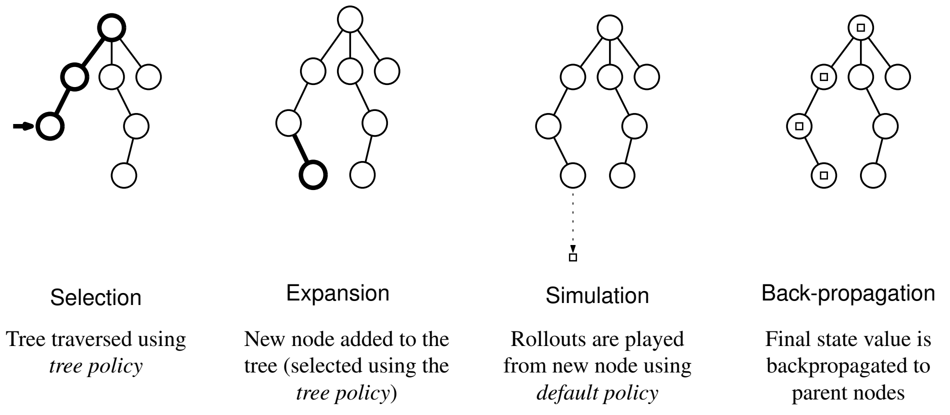

Selection
Tree traversed using *tree policy*
Expansion
New node added to the tree (selected using the *tree policy*)
Simulation
Rollouts are played from new node using *default policy*
Back-propagation
Final state value is backpropagated to parent nodes

Figure 2.2: Diagram representing the 4 steps of MCTS. In the first two steps, the tree is traversed using the Tree Policy, until a leaf node is reached and marked for expansion. The expanded node is selected again using the tree policy. The algorithm then simulates trajectories of the system, using some default policy, until a terminal state is reached. The value of that state is then back-propagated to its parents. Diagram adapted from [1].

[21–23]) and the randomness (as observed in games such as Hearthstone).

MCTS iteratively builds a game tree in memory, from the current state of the game, until some computational budget has been reached, usually, time constrain or number of iterations. The 4 main steps, applied per search iteration in MCTS, are (see Fig. 2.2):

1. Selection: Starting at the root node, a selection function is recursively applied to determine the next node to expand. Selection is mostly based in the information stored in each node, and continues until a leaf node is reached.

2. Expansion: As soon as the algorithm reaches a leaf node, one or more child nodes are added to the game tree, according the available actions.

3. Simulation: From each of the nodes expanded in the previous stage, one or more simulations (roll-outs) are run until a terminal state is reached. The simulations are obtained using a predefined policy, which can be as simple as random selection. The value of the terminal state provides a (noisy) estimate of the value of the previous states.

4. Back-propagation. Once the simulation ends, the result is back-propagated up to the root node, allowing all the node values being constantly updated. Back-propagation is the final step of an MCTS iteration.

These 4 main steps, are commonly grouped in the literature into 2 distinct policies:

1. Tree Policy: Selection and creation of leaf nodes, from the ones already present in the tree (selection and expansion);

2. Default Policy: Produce a value estimation of the actual expanded node (simulation).

The back-propagation step doesn't need to use a policy itself, as only updates the nodes statistics that are used to inform future tree and default policies.

Algorithm 2.1: General MCTS approach

```
function MCTS($state\ s0$)
while within computational budget do
    $state\ s1$ ← TreePolicy($s0$)
    $reward\ r$ ← DefaultPolicy($s1$)
    BACKUP($s1$, $r$)
end
return BestChild($s0$)
```

The 4 main steps of MCTS are summarized in the pseudo Algorithm 1. After the initialization, is created the initial state $s0$. Then, both policies are continuous applied to guide the algorithm: $TreePolicy$ is a function applied in both selection and expansion steps, responsible for returning the new expanded state $s1$; $DefaultPolicy$ is a function applied in the simulation that returns the reward associated r. The $BACKUP$ function represents the final step of MCTS, where the reward value r is back propagated to the earlier expanded node $s1$. Finally, and after the computational budget is reached, the $BestChild$ function returns the action associated to the best child node of the initial state $s0$.

Figure 2.3 illustrates five iterations of MCTS. The algorithm starts with the initialization process, where is added the root node to the tree. Each node is represented by the number of victories/visits, respectively. After the initialization, each of the four steps takes place, being represented by the policies previously mentioned. In the end of each iteration, the simulation result is back propagated, allowing all the node values being constantly updated.

2.3.1 Upper Confidence Bound

Let us consider the 4 mains steps of MCTS in more detail. The goal of MCTS is to quickly estimate the value of the current state (root node) and potential subsequent states, so that the game tree can be used to guide the action selection of the agent. For this reason, each node contains information regarding:

- The number of times that the node was visited in all simulations;
- The number of simulations from that node that resulted in victories.

MCTS uses this information to guide the selection of the next node to visit/expand. It does so by means of the tree policy that balances exploration—i.e., experimenting actions and situations seldom experienced before—and exploitation—i.e., taking advantage of the knowledge built so far (also known as the exploration/exploitation dilemma).

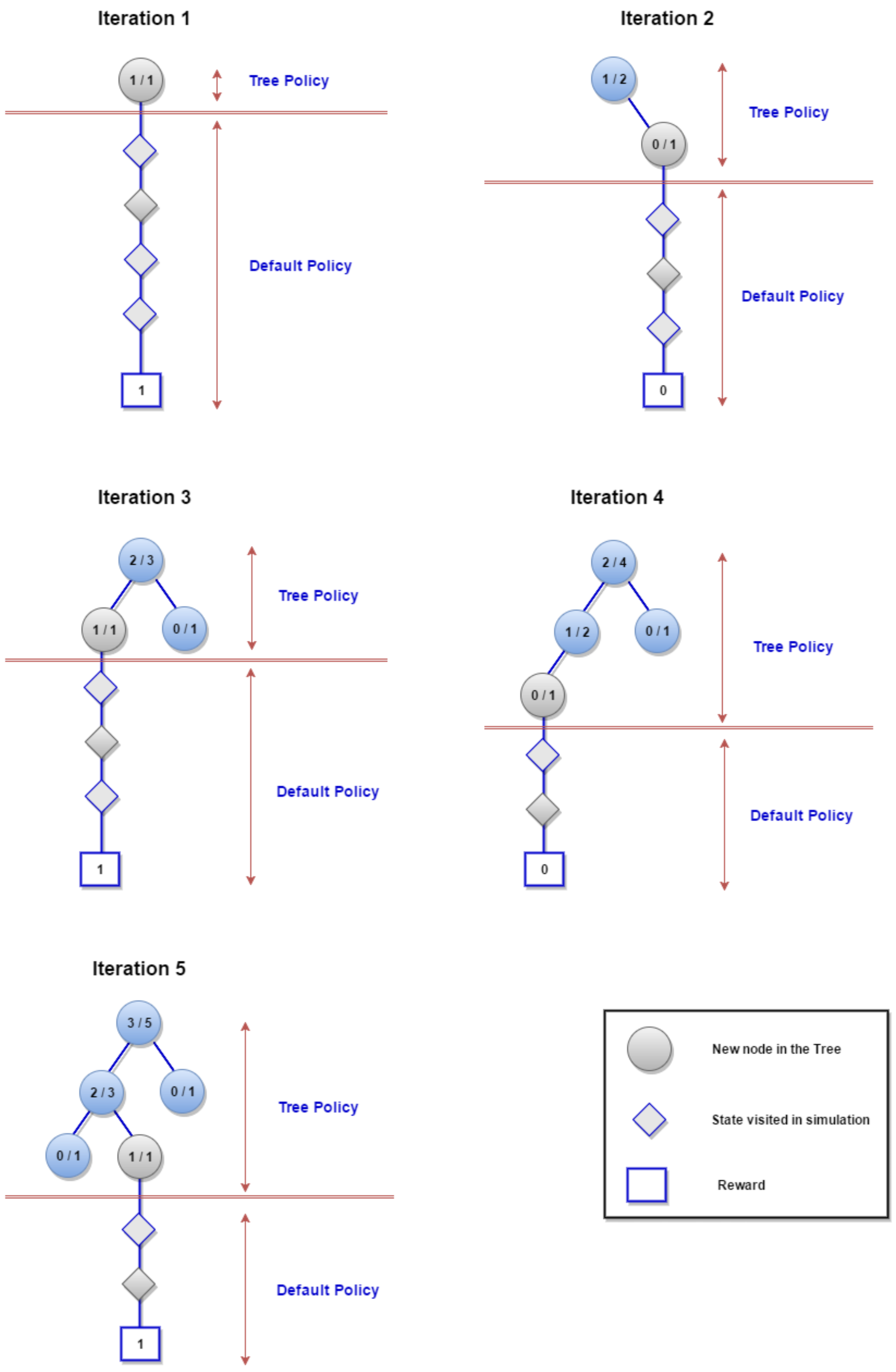


Figure 2.3: Example of 5 iterations in MCTS

A commonly used tree policy relies on the so-called UCB [1, 24, 25], and selects each child node v' of v, according the following formula:

$$v' = \underset{v' \in \ children \ \ of V}{\arg\max} \ \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}, \tag{2.1}$$

where c is a constant, $N(v)$ is the number of visits to node v and $Q(v)$ is the number of victories. These values are updated during the back-propagation stage by increasing both N and Q as necessary.

Analyzing the formula, its noticeable the balance between exploration and exploitation: while the reward term $\frac{Q(v')}{N(v')}$ encourages the exploitation of higher reward-nodes, the second term $c\sqrt{\frac{2\ln N(v)}{N(v')}}$ encourages the exploration of less visited ones.

Using UCB, MCTS is guaranteed to converge to the minimax tree as the number of simulations from each node grows to infinity [24]. However, MCTS is an anytime algorithm: it continues to run until a computational budget is reached, at which time it provides the best answer so far (see Chapter 2.3.4 for the MCTS' characteristics).

2.3.2 Progressive Strategies for MCTS

MCTS is a search algorithm that doesn't need any specific domain knowledge. The algorithm selects the best move by exploring the search space at pseudo-randomly. While the simplest instances of MCTS can run without any specific domain knowledge, it has been shown in practice that the algorithm's performance can be improved significantly with additional knowledge [1, 26, 27].

Progressive bias is a method for integrating domain heuristic knowledge into MCTS: "when a node has been visited only a few times and its statistics are not reliable, then more accurate information can come from the heuristic value $H(v)$" [1]. So, Progressive bias modifies the tree policy, by taking the form of an additional term included in (2.1):

$$v' = \frac{H(v)}{1 + N(v)}, \tag{2.2}$$

Analyzing the formula, its noticeable the balance between exploration and exploitation: the influence of such modification is very important within flew iterations, but rapidly decreases, ensuring that Progressive Bias converges to a selection strategy [1].

2.3.3 EA in MCTS

One of the most significant benefits and appealing features of MCTS, is the fact that the algorithm can be used without any domain specific knowledge: "it works reasonably well in its vanilla form on a variety of problems" [28]. However, "it is also well known and not surprisingly, that the appropriate use of a heuristic function can significantly boost the performance" [29].

So, a new adaptive MCTS algorithm was introduced, that uses EA to rapidly optimize the algorithm's performance. The main idea behind the Fast-Evolutionary MCTS approach, is to embed the EA fitness function with the MCTS' algorithm, where each MCTS' iteration directly contributes for the fitness evaluation.

In other words, in each MCTS' iteration, an individual parameter vector is drawn from the population, where each individual is characterized by a set of heuristic weights. The parameter vector is then used to be integrated and influenced both the Tree and Default Policies, as seen for example in 2.3.2. Apart such influence, the MCTS algorithm functions as normal, being the result of each iteration used as the fitness function for the EA.

After running the MCTS within the computational budget, the algorithm returns the estimate of the best parameter vector computed so far.

2.3.4 MCTS Characteristics

MCTS has characteristics, that make this algorithm to be considered a new framework for AI and a popular choice for a variety of domains [1, 23, 30]:

- Aheuristic: One of the most significant benefits of MCTS, is the lack of need for expert knowledge. As such, the algorithm is ready applicable for any domain, that can be modeled as a tree.
- Anytime: As MCTS back-propagates the result of each simulation, in every iteration of the algorithm, all the node's values are constantly updated. Such benefit ensures that MCTS can be stooped anytime.
- Asymmetric: The tree policy in MCTS, and in particular UCB, allows the algorithm to favour the more promising regions, where nodes with higher reward, are selected often, over those that seem worse (despite the exploration/exploitation dilemma). This characteristic, leads to a asymmetric tree construction over time, offering MCTS the ability of taking advantage of high quality plays, without exploring all possibilities in the tree.

2.4 Hearthstone: Heroes of WarCraft

"Hearthstone" is a turn-by-turn online CCG with matches played between two opponents [2, 15, 31] (see Fig. 2.4). There are several game modes available, being essentially divided in multiplayer and single player modes.

Each player can be represented by one of the 9 heroes available. A hero has a special power, base cards and begins with 30 life points. The goal, is to reduce the opponent hero's life points to 0. To do so, players can summon minions, use spell cards or applying damage to the opponent ones already present on the board.

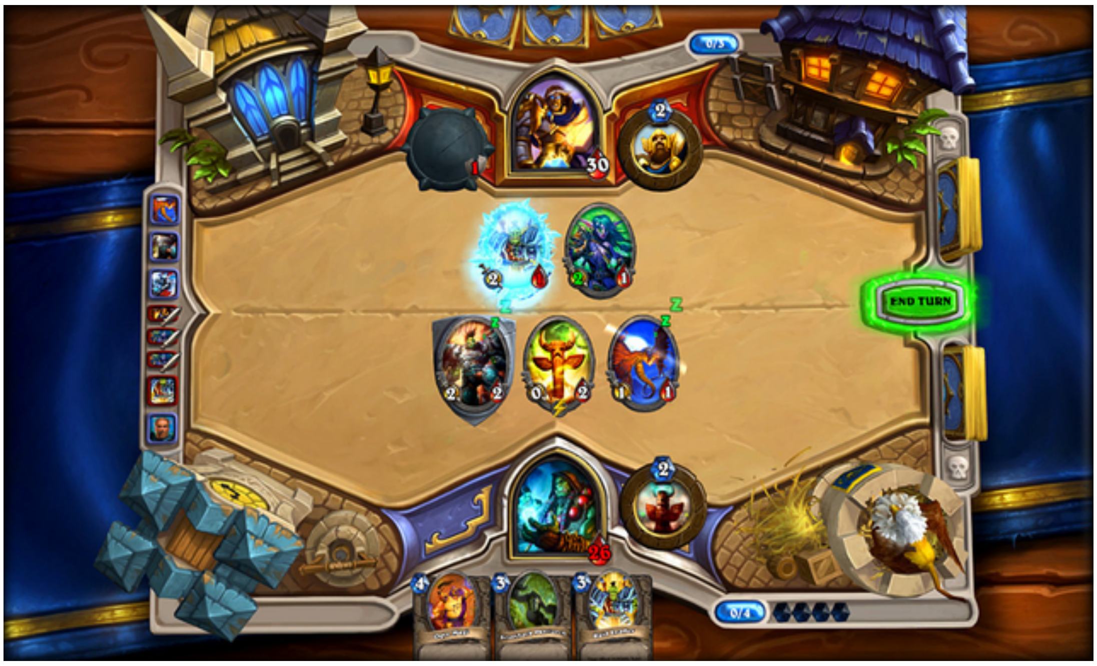


Figure 2.4: Screenshot of a Hearthstone game in process. ©Copyright Blizzard Entertainment.

In Hearthstone, the player's hand is invisible to the opponent. At the beginning, each player receives her own deck, composed with 30 cards. In addition, the decks are initially shuffled, meaning that even expert players doesn't know the cards that are going to draw initially.

In each turn, each player receives a random card and one mana crystal. Mana crystals are the resource used to play cards and use hero powers, and increases by 1 until each turn until 10's, where players control a maximum of 10. The game evolves as each player receives and uses mana crystals to play new cards.

2.4.1 Hearthstone Cards

The collectible Hearthstone cards, are at the core of its gameplay and are one of its most appealing features, as powerful cards may provide the player with a significant advantage. Cards can be grouped into three main types: Spells[3], Minions[4] and Weapons[5]. Spells activate a one-time ability or effect. Minions are persistent creatures that remain in the battlefield (until they are destroyed). Weapons are special cards used by the hero to attack.

Each card is associated to a mana cost, a description and effects or abilities (see Fig. 2.5). Card effects range from situational and local (e.g. a target minion gains life points) to changing the rules of the game (e.g. players draw more cards). The changing rules creates an additional challenge to artificial

[3]http://hearthstone.gamepedia.com/Spell
[4]http://hearthstone.gamepedia.com/Minion
[5]http://hearthstone.gamepedia.com/Weapon

players.

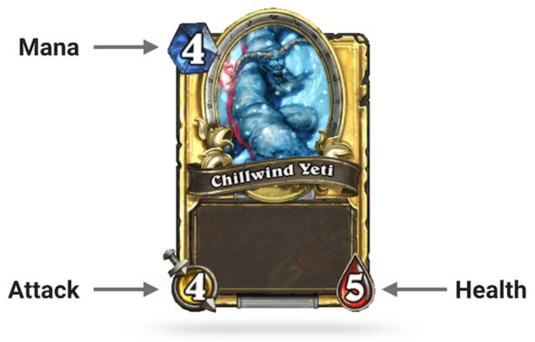


Figure 2.5: Minion card example. ©Copyright Blizzard Entertainment.

2.4.2 Hearthstone Card Abilities

As previously described, minions are persistent creatures with some abilities[6]. There are in total 14 different types, what supplies distinct special powers and effects. Here its summarized some abilities:

- Charge ability: minions with this ability, are able to attack the opponent in the same turn that they were placed in the battlefield.
- Freeze ability: minions with this ability, can freeze the opponent, removing its next possible attack.
- Taunt ability: the enemy hero and his minions, are forced to target this minion, before attacking others.
- Divine shield ability: special ability, that allow minions to ignore the first attack received.
- Windfury ability: minions with this ability, are able to attack twice per turn, instead of only once.
- Enrage ability: when the minion with this ability is damaged, the stated effect in the card description becomes active. Usually, its increased the minion's attack by k points.

2.4.3 Hearthstone Strategies

In Hearthstone, building the best deck possible is an essential skill. Because players can select the decks to play, the deck used governs the player's strategy. Normally, it's not unusual to associate "standard decks" with common strategies in the game [2, 15], and the decks are characterized "by the distribution of the mana cost, of the cards they contain, which is referred as mana curve" [2].

Common decks/strategies include:

[6]http://hearthstone.gamepedia.com/Ability

- Aggro (meaning “aggression”): Low mana curve deck, that is built with cheap cards with the main purpose of finishing the game as quickly as possible. These decks consume a significant amount of cards, as they seek to inflict the maximum damage, and exhaust themselves if are not able to quickly kill the opponent.

- Mid-range: Balanced curve deck, very flexible and primarily designed for responding to the opponent’s moves. Its objective is to gain power in the mid-game turns, where the player can access powerful finishers (something that, for example, aggro players cannot afford).

- Control: Higher mana curve deck, that prioritizes the survival in the first turns. Control decks usually pose huge threats with just few minions, but without a careful early game, any aggro or mid-range strategy can defeat it. Control decks are designed to gain control in the last stages of the game.

2.4.4 Hearthstone Battlefield

In Hearthstone, the battlefield[7] represents the board where the actions takes place (see Fig. 2.6). The battlefield incorporates the different UI elements available in Hearthstone (e.g. the player’s hands and minions), and exists different types of them, with specific design that incorporates distinct interactive elements. The interactive elements available, don’t affect the game in anyway, and initially the battlefield used, is chosen at random from the available ones.

The hero selected is shown near to the bottom, being the opponent one displayed in the opposite side, close to the top. To the right of the selected hero is the hero power, and bellow the player’s hand and mana crystals, the resource used to place new cards into the battlefield.

A history of plays is preserved and shown in the history panel, while the number of cards presented in both decks can be found all the way to the right. Both player’s minions are found in the center of the battlefield, being its placement preserved between screens. All of items mentioned are mirrored in the opponent’s side.

2.4.5 Hearthstone Mulligan and Fatigue

In Hearthstone, the player who goes first, is decided by tossing a coin in the beginning of the game (commonly named as mulligan in the community). Then, random cards are drawn from the player’s decks (3 for the player who goes first and 4 for the player who goes second), where they can choose the ones to keep in hand or to replaced individually (see Fig 2.7). The discarded cards are shuffled back, and new ones are randomly drawn as replacements for the player’s hand.

[7] http://hearthstone.gamepedia.com/Battlefield

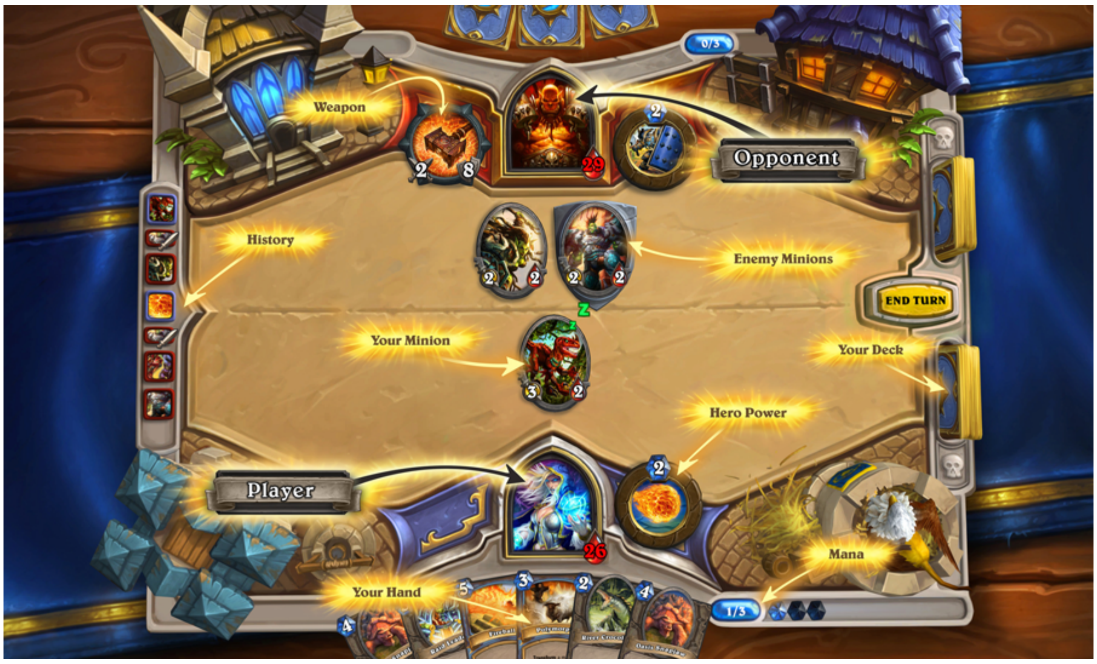


Figure 2.6: Hearthstone battlefield. ©Copyright Blizzard Entertainment.

For balancing the game, and in addition for receiving 1 more card, the second player also receives an additional special card, named coin, that can be used to grant an extra mana crystal.

The game evolves, as both players use the mana crystals, to place new threads into the battlefield. The player's turn ends, when they run out of time, or in the moment they select the End Turn button. Usually, a turn lasts a maximum of 75 seconds, with some additional time for animations. If players drawn all the cards in the deck, they go into the fatigue stage. Fatigue initially deals 1 damage, in the beginning of the player's turn, and increases by 1 each time is triggered.

2.4.6 Hearthstone Game Specification

In Hearthstone, players must deal with hidden information regarding the cards of the opponent, chance, randomness, and a complex gameplay, which often requires sophisticated strategy. In addition to the properties mentioned in 2.1, the work in [30] also defines Hearthstone as:

- Scaling complexity game: Hearthstone features more than 1000 playable cards, with different properties that can be grouped to define a complex strategy. In addition, new expansions are continuously being released, increasing the game's complexity at many levels.
- Zero-sum game: Hearthstone is an adversarial game of 2 players, meaning that the gains of one are strictly balanced by the losses of the other.
- Non-reactive game: The player who is waiting, can't interfere the opponent's turn.

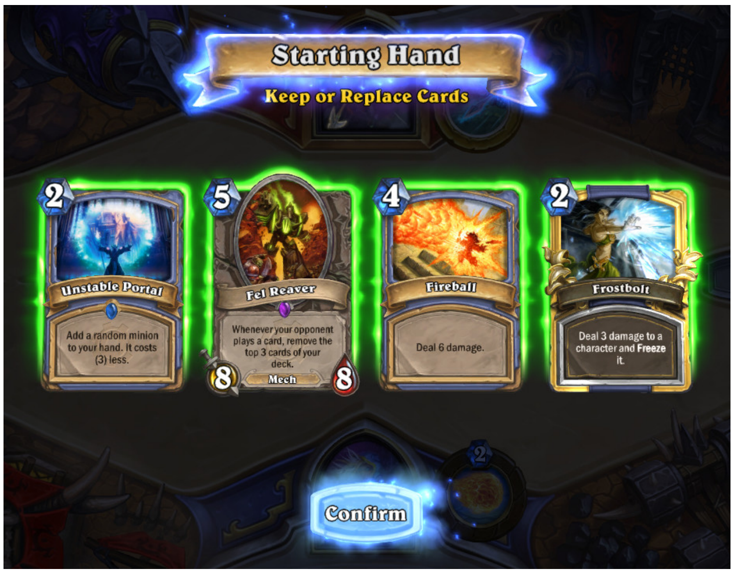

Starting Hand
Keep or Replace Cards
Unstable Portal
Add a random minion to your hand. It costs (3) less.
Fel Reaver
Whenever your opponent plays a card, remove the top 3 cards of your deck.
Mech
Fireball
Deal 6 damage.
Frostbolt
Deal 3 damage to a character and Freeze it.
Confirm

Figure 2.7: Mulligan stage of the second player. ©Copyright Blizzard Entertainment.

- Timed game: A turn in Hearthstone lasts at least 75 seconds, with some additional time for animations. If such time is reached, the actual turn is forcefully ended and then passed to the other player.
- Turn-based game: The game flow of Hearthstone is partitioned with sequential turns, what means that the player's turn alternate, until the game results in a victory.
- Finite game: Hearthstone has a limited number of 90 turns, coming to a draw after such condition.

2.4.7 Hearthstone Simulators: Metastone

The Hearthstone community is massive and essentially characterized with devoted players. From forums, where gamers exchange new strategies, to the existence of simulators that emulates the main mechanics in Hearthstone, there are many tools available for experienced players.

Since the collectible cards are the Hearthstone basis, there is the necessity for other tools, not present in the real game, to quickly allow the testing of new decks. Simulators thus represent an important role, and besides the emulation of the main gameplay mechanics, they provide additional means for the creation and evaluation of decks recently built. For instance, Fireplace, Metastone and Hearthbreaker are examples of simulators available to the community. In particular, Metastone [32] includes functionalities for allowing the simulation of a large number games between different heroes, decks and AI systems, providing summarized statistics after the matches. The simulator already includes some AI systems that can be tested, including:

- Random: Agent that selects actions at random and provides a naive baseline for comparison.

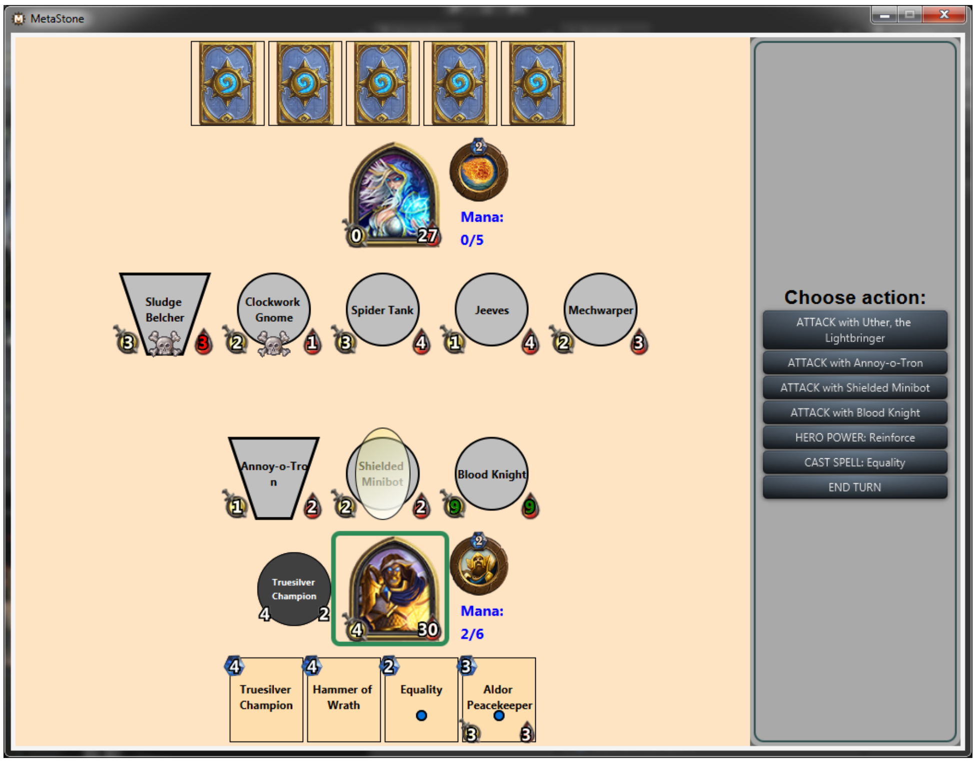


Figure 2.8: Metastone. ©Copyright Metastone.

- No Aggression: Agent that does not attack the opponent. The AI randomizes between playing cards or simply performing the "end-turn" action.
- Greedy: Myopic player whose actions are driven by a heuristic, built on several game metrics, and whose weights were tuned using an evolutionary approach (see Chapter 4).
- Game State Value (GSV): Player that uses a recursive alpha-beta algorithm driven by the aforementioned heuristic. To the extent of our knowledge, this is the best AI system available, and several existing players report disappointing performances against it [4].

Given the functionalities it provides, we adopt Metastone as the testbed in which we evaluate our AI system.

3

Related Work

Contents

This chapter presents a description of several works performed, that were relevant to our research. Despite Hearthstone being a recent game, there are already a considerable number of existing methodologies, that considered different aspect of the game. We start by presenting some works performed in Hearthstone and similar games, proceeding with others that directly played a significant role in our methodology.

3.1 Predicting the Opponent's Deck

"Hearthstone" features more then $1,000$ playable cards that can be collected. Despite the resulting number of possible decks being extremely large, the decks normally used features a predictable structure:

- Some cards are specially designed to work with others;
- Hearthstone have cards that are restricted to a specific class or hero;
- A significant number of cards are under-powered, representing bad choices for any deck.

Besides that, another interesting factor that corroborate this predictable structure is the net-decking phenomenon, representing the the act of using a deck made by another player. So, to study this predictable structure, a statistical learning algorithm was used in [2], aiming to predict the most probable and playable future cards. The algorithm, by modeling the relationship between cards and using a database of game's replays, returns a list of possible future cards with the corresponding probability.

In particular, the algorithm uses a modified version of Markov chains, that modulates the card relations as a set of n-grams. The idea, is through the opponent's cards, already played, extract bi-grams (sequence of 2 cards) from the games database and then counting their frequency. Figure 3.1 illustrates the point. Lets suppose the opponent already played 2 cards, being Deadly Poison and Shiv. From those cards, the algorithm extracts all the bi-grams present in the games database, being then counted and sorted by frequency. The bi-grams with higher frequency are returned, representing the cards more often associated with the ones previously played.

This study, is an attempt to address the issue of predicting the opponent's next action using Markov chains. The approach was able to achieve an accuracy above 95% after analyzing $50,000$ game replays, indicating that, in fact, the number of effective decks that the players choose can be significantly narrowed down.

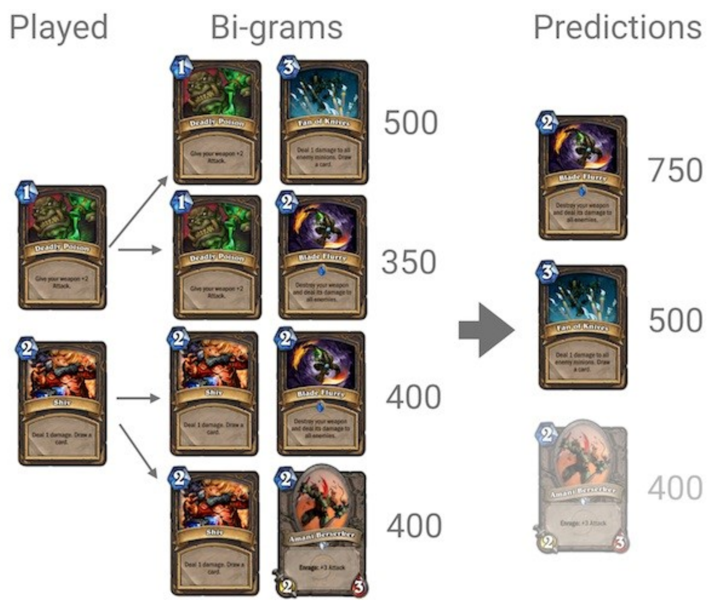


Figure 3.1: Prediction opponent cards example. Image adapted from [2].

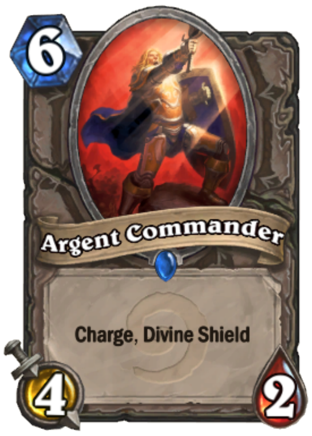
3.2 Evaluating Cards in Hearthstone

The collectible Hearthstone cards are the core of its gameplay, and one of its mots appealing features. As powerful cards may provide the player with a significance advantage, the work in [3], revealed an algorithm to find these cards. As explained in Chapter (2), to each card is associated a set of properties. So, the algorithm creates an equation system, where the manna cost of each card is treated as the sum of it's properties. After gathering cards with similar properties and solving the equation system with the least squares method, the real value of each card is calculated.

Figure 3.2 illustrates the point, where the card argent commander is modulated. For instance, if after solving the equation system, the card's properties were equal to 1, according the example in the Figure 3.2, we will obtain the following equation:

$$\begin{aligned} 4a + 2h + c + d + i \Leftrightarrow \\ 4(1) + 2(1) + (1) + (1) + (1) = 9 \end{aligned} \tag{3.1}$$

According equation 4.3, since the real value of Argent Commander would be 9, a value higher than is 6 of mana, this card would represent a overpowered one. Applying this approach to 134 cards, the author concluded that the real value of most cards were very close to the mana cost. At the time, he was able compile a list, representing the most overpowered cards, where Light's justice and Soulfire were the top 2.



mana = attack + health + charge + divine shield + intrinsic value

$6 = 4a + 2h + c + d + i$

Figure 3.2: Example of one equation. Image adapted from [3].

3.3 Automated Play for Card Games

3.3.1 Supervised Learning Approaches

In this work, David Taralla developed $Nora$, an intelligent agent for the game of Hearthstone [33]. Using a supervised learning algorithm, more particularly, decision trees, $Nora$ was able to predict a (discrete) value for each action and state. The result of such classifier was used for the action selection.

Regarding the training process of $Nora$, the author generated the training examples with an impressive number of $640,000$ games between 2 random agents, where was used a heuristic function for evaluating such examples. The heuristic function, was developed in cooperation with expert players of Hearthstone, and reflected a typical control strategy, a strategy that reproduced the process of gaining board control for preventing the opponent's victory. For this purpose, several game metrics were used, changing and influencing the gamming strategy of $Nora$:

- Minion advantage (MA): number of minions the player controls over her opponent.
- Tough Minion advantage (TMA): number of powerful minions the player controls over her opponent.
- Hand advantage (HA): number of hand cards the player has minus her opponent's hand cards.
- Trade advantage (TrA): factor that represents how good the minions on the board are to lead to advantageous trades.

$$H(s) = 1.75 * MA(s) + 2.50 * TMA(s) + 1.00 * HA(s) + 3.75 *_{TrA} (s). \tag{3.2}$$

Against a random agent, $Nora$ presented good results, performing a win-rate near to 93%. However, against an agent that implemented a mid-level playing strategy, $Nora$ just obtained a win-rate close to 10%.

In this work, the author put forward the claim that the process of defining the representation of states and actions (heuristic) was difficult, being such improvement even more challenging, since would required even more knowledge of Hearthstone. Also, the author mentioned that the the random training process used by $Nora$ lead her to underestimate the value of the play actions. Surprisingly, despite this training process, $Nora$ was even successful at aiming the correct targets. As future work, the author suggested that would be interesting to work with a simulation based algorithm like MCTS.

3.3.2 MCTS for Card Games

3.3.2.A Poker

Poker is a card game with similarities to Hearthstone. Its characteristics of imperfect information and stochastic outcomes, make this game a subject of great interest for AI researchers. So, the work in [26], proposed the use of MCTS, with the integration of expert knowledge for the game of Poker. In particular, the approach integrated a probabilistic model that estimated the opponent's behavior in unseen situations. The probabilistic model predicted 2 sources of information: the opponent's private cards and future actions. More concretely:

- The prediction of cards was used to circumvent the hidden information problem, by sampling the opponent's cards.
- The prediction of actions was used to assist the tree and default policies during iterations, whenever the opponent performed actions in the game tree.

For evaluating the real impact of such integration into MCTS, the authors compared the results using the "vanilla" version of MCTS, in a one-o-one game against different AI. In general, the MCTS version with expert knowledge always performed better than its "vanilla" version, performing a higher aggression and deciding to bet constantly in early stages. In particular, versus a more challenging opponent, MCTS with expert knowledge had a small margin of victories, unlike the "vanilla" version that lost quite severely. For further research, was suggested the performance of more elaborate tests, for optimizing the parameters used in MCTS.

3.3.2.B Magic

The work in [12], studied hybrid solutions for general game play in M:TG. The same solutions merged 3 different AI approaches, that covered the 3 gaming decisions points considered in the experiments (see

Table 3.1: 12 different AIs developed

AI Name	Attack Strategy	Blocking Strategy	Card Play Strategy
RA_RB_RP	Random	Random	Random
RA_RB_SP	Random	Random	Rule-based
RA_RB_MC	Random	Random	MCTS
RA_SB_RP	Random	Rule-based	Random
RA_SB_SP	Random	Rule-based	Rule-based
RA_SB_MC	Random	Rule-based	MCTS
SA_RB_RP	Rule-based	Random	Random
SA_RB_SP	Rule-based	Random	Rule-based
SA_RB_MC	Rule-based	Random	MCTS
SA_SB_RP	Rule-based	Rule-based	Random
SA_SB_SP	Rule-based	Rule-based	Rule-based
SA_SB_MC	Rule-based	Rule-based	MCTS

table 3.1):

- Attack strategy: deciding which creatures to attack;
- Blocking strategy: deciding how to defend and how to block the opponent's attacking creatures;
- Card selection: deciding which hand's cards to play.

In an initial stage, was studied the advantages of being the first player to draw. Such results, indicated that starting at first, didn't offer a significant advantage, where the former players achieved a performance of 52% of victories.

In a second test set, was studied the performance of the MCTS based solutions for the card selection stage. The results obtained, suggested that existed a significant advantage for the same solutions, over the rule-based and random ones. In particular, the MCTS hybrid solutions consistently scored more, performing better win-rates and specially over the random ones.

In a final test set, was investigated the effect of the number of roll-outs in MCTS. With only 10 roll-outs per node, in average, these hybrid solutions performed a win-rate close to 61%, whereas the increasing number continued to improve the performance, reaching the plateau of 80% at 600 roll-outs.

This work shown that MCTS, could successful be applied to a domain of card selection in M:TG. Also, the inclusion of MCTS with other AI approaches, namely a rule based solution, yielded an convincing performance, with the only few simulations used.

3.3.2.C Hearthstone

As stated in Chapter 2, the development of Hearthstone suffered limitations that directly influenced the gaming experience. So, in order to improve such experience, the works in [18] and [4] proposed the development of a game agent using the MCTS "vanilla" version for the game of Hearthstone. Both works were developed in Metastone, and used the different AI systems available, for evaluating the algorithm's performance (see Chapter 2).

The major bottleneck and difficulty at the time was the hidden information concept of Hearthstone. To circumvent such difficulty, both projects performed modifications in the simulator, allowing MCTS to know the opponent's deck.

In [18], besides the evaluation with the different AI systems, was compared the algorithm's performance with different decks, that represented different gaming strategy's. Generally, none MCTS agent won less than 75% of the games against the most challenging Metastone's AI present at the time, the rules-based agent. Also, for evaluating the real competitive level of the approach, the authors made additional tests with human players of different competitive levels. To do so, were performed empirical modifications in the MCTS' parameters, for adapting the approach. Such results indicated that the MCTS agent got slightly less than 50% victories against a human player, with moderate and skilled level, but 0% against a master player.

Regarding the work in [4], a similar approach was conducted. Against the current state of the art of Metastone's AI, the GSV, the approach performed a poorly performance of just 9%. However, versus a less skilled opponent, the Random agent, the approach obtained a performance of 69%.

In both works, the authors concluded that MCTS performed better than expected. They reported, that the agents occasionally made extremely smart moves, playing combos and keeping cards in hand intentionally, behaviors that even the actual AI system implemented in Hearthstone performs poorly. Also, was recommended, the integration of expert knowledge and the realization of more elaborate test to optimize the MCTS' parameters.

3.4 Conclusions

To conclude this chapter, we present a brief overview of the various works presented:

- Predicting the opponent's deck were able to achieve successful results on the complex task of general gameplay, indicating that, in fact, the number of effective decks that the players choose in Hearthstone can be significantly narrowed down;
- Evaluating cards in Hearthstone revealed an approach that creates an equation system to find overpowered cards in the game. Each card was modulated as the sum of its properties, properties that can be use for instance, to evaluate the opponent's game state (e.g. sum of the player's minions properties, for evaluating their quality regarding the opponent ones);
- Supervised Learning Approaches used decision trees, for the creation of an intelligent agent for Hearthstone. The same work also revealed an heuristic function, that evaluated the training exam-

ples, and can be use in the future, to integrate expert knowledge into MCTS;

- MCTS For Card Games adapted different MCTS methods, for different card games (see table 3.2). Regarding the ones applied to Hearthstone, the behaviors achieved, the performance and the possible improvement with expert knowledge of the game, were conclusions widely discussed.

	Expert Knowledge	**Hidden Information**	**Notes**
MCTS for Poker [26]	yes	yes	vs the ”vanilla” version
MCTS for Magic [12]	no	no	hybrid MCTS approach
MCTS for Hearthstone [18]	no	no	tested with humans
MCTS for Hearthstone [4]	no	no	9% vs GSV; 69% vs random

Table 3.2: Overview of the MCTS works for card games

4 Approach

Contents

This chapter presents the solution of our approach, describing each component developed, with the goal of understanding the proposed enhancements to MCTS.

◇

In the last chapter, we have seen different methodologies that considered distinct aspect of Hearthstone. From those that considered general game play, the application of the ”vanilla” version of MCTS’ clearly stand out (a version that neither solves the hidden information problem or integrates any type of knowledge).

Despite the disappointment performances against the best AI present in Metastone [4], the GSV (see Figure 4.1), it has been shown in practice that the algorithm’s performance can be improved significantly with the integration of expert knowledge.

So, in attempt to improve the MCTS competitive level and solving the hidden information problem of Hearthstone, we propose to use of MCTS with the integration of expert knowledge of the game’s domain. The knowledge used, can be divided in two parts:

- a heuristic that guides the tree and default policies of MCTS;
- a deck’s database that allows the algorithm to reason about the possible cards that the opponent may have.

Our approach is summarized in Figs. 4.2 and 4.3, and the details are discussed in the continuation.

4.1 Node Structure

The node structure developed, contains all the information necessary for MCTS to perform the different steps per iteration. The node structure contains the basic statistics considered in the literature,

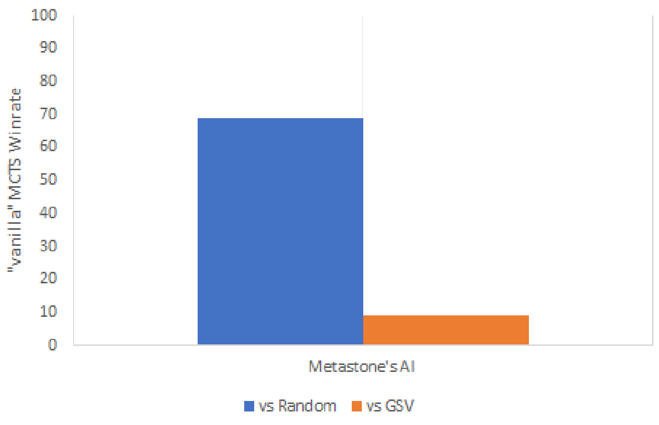


Figure 4.1: ”vanilla” MCTS version vs Metastone’s AI [4]

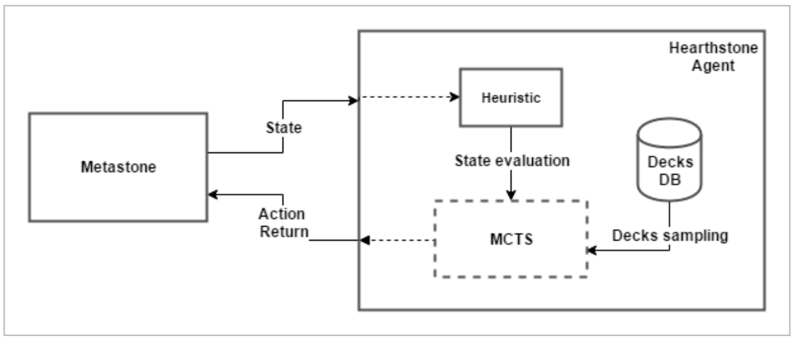


Figure 4.2: High-level overview of the interaction between our MCTS agent and the Metastone game engine.

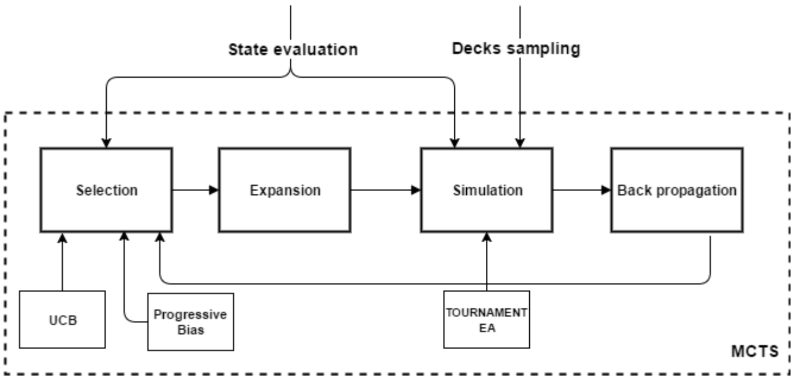


Figure 4.3: Detail of the MCTS block from Fig. 4.2.

with some additional information that assists the expansion and back-propagation steps of MCTS. In particular, we considered:

- State: All the information regardless the current state:
  - Active Player in such node:
    - Hero:
      - Hero Power;
      - Weapon.
    - Health and Attack points;
    - The AI used for returning a single action;
    - Graveyard;
    - History of plays;
    - Hand Cards;
    - Deck: List of cards without the hand ones;
    - Minions in board;

- Number of Victories: How many times, a certain node, has been victorious;
- Number of Visits: How many times a certain node has been visited;
- Parent Node: Relation to the node to which the current is a child. This relation is used to perform the back-propagation. Also, the root node doesn't have a parent one;
- Children Nodes: List of child nodes;
- Available Actions: List of the remaining available actions per node. If the current list is empty, then the current node is not expandable;
- Action: The action that originated such node. When the computational budget is up, we return this structure;
- Player: The player associated to such node.

4.1.1 Actions in Nodes

When a node is being expanded, a child node is generated according the available actions. Metastone considers 5 different types of actions:

- Play Card: the most primitive type of action in Metastone. Occurs when a card is played into the battlefield or activated. All entities in Metastone are initially cards, being then transformed to a more particular set, after they are played (e.g. initially we have the card minion, and only after the card has been played, the entity minion is generated and placed into the battlefield);
- Battlecry: action activated, when the card with such property is played;
- Discover: action that allows the player to select, from a given number of auto-generated random cards, the one to add to her hand;
- Physical attack: action that happens, when the active player commands one character, to attack another;
- End Turn: action that ends the actual player's turn. Plays a very important role, since to a node is only associated a single action. So, this action enables MCTS to handle the multiple actions per turn, functioning as leaf nodes between the player's turn.

As an important observation, is the fact that, since MCTS relies on a search tree, its important to consider all the possible targets for a given action. In other words, if during the expansion, a certain physical attack action is select, from the available ones, we considered all the possible targets, including the friendlies and enemies ones for this action (e.g. minions with the $enrage$ ability improve the attack as they suffer damage, so expert players often provoke damage on purpose to these minions, to make it stronger).

As another example, is the possible positions that some card or minion can be placed in the battlefield, being such position very strategical in Hearthstone. So, each time the Play Card action is selected to expand, we considered all the possible target positions that can be placed.

4.2 Tree Policy

4.2.1 Selection

In our approach, we want to accurately select the next node to expand. Its main difficulty, regards the balance between the exploitation of the most promising nodes and exploration of alternatives ones, that in future iterations, may become to be extremely beneficial. Not only we want to preserve such balance but also to refine it, with expert knowledge of Hearthstone.

So, the selection strategy implemented, merges 2 distinct methods:

- The UCB algorithm, that has been show in practice to balance the dilemma of the exploration versus exploitation;
- Progressive Bias, as a mean to integrate expert knowledge of Hearthstone, in a form of an heuristic in MCTS.

In other words, we enrich the UCB selection rule with an extra term, that accounts for the domain-specific knowledge. So, each time a node is selected in MCTS, its choice maximizes the following formula:

$$v' = \underset{v' \in \ children \ \ of V}{\arg\max} \ \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}} + \frac{H(v)}{1+N(v)} \tag{4.1}$$

In other words, all the children of the node v, are compared to each other, being returned the one with the highest score. The selection, is then recursively applied, until a leaf node is reached. Also, this process in only performed on fully expanded nodes (nodes with none available actions), what certifies that all the actions are explored, before a deeper search is allowed.

4.2.2 Expansion

After the node's selection, MCTS reaches a leaf node. So, as expansion policy, we use the one proposed in [34], where the available actions are chosen at random, from ones not yet expanded. The list of possible actions is then updated, by removing the chosen action from the same list. Finally, the new expanded node is returned for the next step of MCTS.

4.3 Default Policy

Regarding the hidden information problem of Hearthstone, where its unknown the player's hand and remaining cards in deck, its impossible to perform the standard default policy that choses actions at random, until a leaf node has been reached.

So, our default policy, implements a series of adaptations, that not only handles the hidden information problem, but also integrates expert knowledge, as has been shown in practice to improve the levels of play [35]. Our default policy uses:

- Deck's database: to circumvent the hidden information problem, aiming to predict the current opponent's deck;
- EA Selection method: to add expert knowledge during the roll-outs phase.

As an important observation, initially, the recently expanded node is cloned, where all the operations are performed. This aims to ensure, that the default policy is independent from the nodes present the tree. The motivation behind such adaptations is as follows:

4.3.1 Using a Database of Decks

"Hearthstone" features more then $1,000$ playable cards, from which the players can build their 30 card deck. For competitive play it is crucial to predict what cards can be played by the opponent, which creates an added difficulty for the tree and default policies of the MCTS algorithm, when simulating the opponent's moves.

Fortunately, the choice of hero is known from the beginning, and the efficiency of a deck strategy implies a more limited choice of cards. In the lack of game data from which probable decks can be estimated, we instead adopt a simpler alternative that relies on community-build decks as representatives of the most common game strategies (Aggro, Mid-Range and Control). In particular, we use a set of decks built from those used by professional players in recent tournaments.

So, we develop an deck database that effectively reduces the search space that MCTS needs to consider. Intuitively, it can be understood as the artificial counterpart to the knowledge of a master player, who is aware of common decks types and is able to (approximately) infer the opponent's deck type with the most likely cards from its hero.

In our approach, we use the cards played by the opponent so far, to select/sample one deck from the deck database that is compatible therewith. In other words, our approach counts the common references present in both the cards played by the opponent so far, and each deck at the database. The deck selected is the one with most common references, which will be the one that most accurately translates the opponent's strategy and which has, therefore, the best predictive ability. After removing from the selected deck the cards already played, we obtain a collection of the likely cards to be played, that MCTS then uses in its search.

As an example, lets assume the opponent already played so far 4 cards, and has in hand 5. So, from the 4 cards already played, we select the deck in the database that preferably reproduces such cards. After removing the cards played from the deck selected in the database (e.g. 30-4=26), is selected randomly 5 new cards to replace the new opponent's hand (being such cards after removed, 26-5=21). Finally, we have a collection of the new 21 cards to be drawn in the roll-out, that will replace the remaining opponent's deck in the cloned node.

4.3.2 Adding Expert Knowledge

Given the large branching factor in Hearthstone, the default policy of MCTS will generally require a large number of iterations, before each node is properly explored and an accurate estimated of each action's value can be obtained. Such extensive process is time-consuming, which is inconvenient given the limited time to play imposed by the game.

To circumvent this difficulty and to integrate expert knowledge, we adopt the tournament selection approach commonly used in EA [36, 37]. In particular, at each step of the default policy, k actions are sampled at random from the set of allowed actions.

Each of these k actions is then "scored", according to a pre-defined heuristic function, that evaluates the game state resulting from executing such action.[1] It is the value of k and the heuristic function that, in our approach, define the roll-out phase int the default policy. For example, if $k = 1$, the resulting default policy reduces to standard random sampling. On the other hand, if $k = N_A$ (where N_A is the number of currently admissible actions), the resulting default policy is greedy with respect to the heuristic.

This approach is used in both the player's game states, is tested with different values of k, and uses different policies, according the player's turn for selecting actions. Indeed we assume rational players:

- Player's policy: it's selected the action, from the k sampled ones, that maximizes the heuristic value (roll-out in player's turn);
- Opponent's policy: it's selected the action, from the k sampled ones, that minimizes the heuristic value (roll-out in opponent's turn);

As an example, lest assume the values of $k = 75\%$ for the player, and $k = 50\%$ for the opponent. Also, lets assume in the current default policy state s_t, we have 50 available actions, and is the player the one to intervene in the state s_t. So, in such conditions, if $k = 75\%$, we randomly select 38 actions (e.g. 0.75*50=38), where the one it the maximum heuristic value is selected. On the contrary, if the opponent is the one to return an action, we randomly select 25 actions (e.g. 0.50*50=25), where the one with the

[1] It is worth emphasizing that the heuristic used in the simulation stage need not be the one used in the selection stage.

lowest value is selected. After such effort, we move to the new state $s_t + 1$, where the roll-out continues, until a leaf node is reached.

4.4 Back-Propagation

The default policy, aims to estimate the current expanded node's value. This is followed, by the back-propagation step of MCTS, where the nodes statistics are updated. In our approach, this is performed by means of an function, that always increase the visits count in 1, and the number of victories according the default policy's result. As long as the root node hasn't been reach, all the nodes and their respective fathers are updated, using the same effort.

4.5 Heuristics

Both the tree and the default policies used in the selection and simulation stages of MCTS, relies on a heuristic function that evaluates subsequent states and informs the action selection. In order to assess the impact of such heuristic in the performance of the method, we consider two distinct heuristics. Both heuristics were constructed as linear combinations of a small number of features extracted from the state s, taking the general form

$$H(s) = \sum_{n=1}^{N} \alpha_n \phi_n(s), \tag{4.2}$$

where $\phi_n(s)$ is the value of feature n at the state s. The difference between the two heuristics lies on the features used. Another important observation, is the fact that both heuristics are symmetric, meaning that if one state is considered very good for one player (e.g $H(s) = 30$), in the same way is considered very bad for its opponent (e.g. $H(s) = -30$).

4.5.1 Heuristic 1

The first heuristic, included a small number of hand-picked features, that reflect a game control strategy, reproducing the process of gaining board control and preventing the opponents victory (see 3.3.1). The first heuristic can be written as follows:

$$H(s) = \alpha_{MA}\phi_{MA}(s) + \alpha_{TMA}\phi_{TMA}(s) + \alpha_{HA}\phi_{HA}(s) + \alpha_{TrA}\phi_{TrA}(s) + \alpha_{BM}\phi_{BM}(s). \tag{4.3}$$

where MA is the minion advantage, TMA is the tough minion advantage, HA is the hand advantage, TrA is the trade advantage and BM the board mana advantage.

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

4.5.1.A Minion advantage - MA

The minion advantage represents the number of minions that one player controls over her opponent. In Hearthstone, the more minions one player controls, more tempted is the opponent to kill them all. So this feature, is pretty straightforward: "with more minions than my enemy, I can more easily prevent him from putting minions for himself" [16]. In other words, the minion advantage aims clearing the opponent's creatures, and after the trade, still having a higher overall number of threads in the battlefield. The resulting formula can be written as follows:

$$MA = \sum_i (m_i) - \sum_j (\hat{m_j}) \tag{4.4}$$

where m_i is the friendly minion i and $\hat{m_j}$ the enemy minion j.

4.5.1.B Tough Minion advantage - TMA

The tough minion advantage represents the number of powerful minions that one player controls over her opponent. Its an important modification from MA due the characteristics of Hearthstone: if a player at her side a board full of minions, but if they are all weak, that's not considered as having control. In Hearthstone, "there are too many chances that the opponent casts a devastating spell, that he reserved specially for board control situations" [16]. An example, lets consider the *Flamestrike* spell:

"Deal 4 damages to all enemy minions."

So, the tough minion advantage captures these observations, by subtracting the number of powerful minions, with health over 4, that one player controls over its opponent:

$$TMA = \sum_i (tm_i) - \sum_j (\hat{tm_j}) \tag{4.5}$$

where tm_i is the tough friendly minion i and $\hat{tm_j}$ the tough enemy minion j.

4.5.1.C Hand advantage - HA

In Hearthstone, the number of cards that one player has in hand is very representative of the actual board control situation. This metric assumes: "with more cards than my opponent, more chances I have to react my opponent's tentatives of taking control over the board" [16]. Another important observation, is the fact that one player is doing better, as more cards he has in hands, because he will have more actions to choose from. So, the hand advantage subtracts the number of cards the one player has in hands over her opponent:

$$HA = \sum_i (hc_i) - \sum_j (\hat{hc_j}) \tag{4.6}$$

where hc_i is player's hand card i, and $\hat{hc_j}$ the opponent's hand card j.

4.5.1.D Trade advantage - TrA

The trade advantage reflects how good the execution of an action was, and facilitates the possible subsequent actions for a given player. This metric, is a subset of a more general one named as card advantage: "Card advantage is ultimately an expression of the difference between the rate that on player gains and lose cards in the battlefield" [16].

For example, if for killing a friendly minion, the opponent used 2 cards, this action lead to a generation of "2-1" card advantage for the player. Another important observation, is the trading metric that represents the physical attack actions that one player performs. So, and to summarize everything that has been said, the trade advantage, is the advantage generated after the trading is done: "The process of finding the best trade is not trivial, as an exhaustive search would result in evaluating all possible combinations of possible attacks" [16].

The trade advantage performed in this thesis, captured these observations, being modulated as the difference of the health and attack points, between the 2 players, that was generated in an action/trade. In other words, if a good trade was performed, must be generated a positive difference in points, regarding the player who played (e.g. if during a trade, 1 of the enemy's minions was killed, and the friendly minion still lives, the enemy minion's statistics no longer influences the trading function, being then generated a positive difference in points). The resulting formula is as follows:

$$TrA = \sum_i (a_i + h_i) + H + A - \sum_j (\hat{a_j} + \hat{h_j}) - \hat{H} - \hat{A} \tag{4.7}$$

where a_i is the attack and h_i the health for the friendly minions i, H the hero's health, A the hero's attack, $\hat{a_j}$ and $\hat{h_j}$ the attack and health, respectively, for the enemies minions j and finally, $\hat{H}$ and $\hat{A}$ the opponent's health and attack respectively.

4.5.1.E Board mana advantage - BM

The board mana advantage reflects the difference between the sum of the mana for the player's cards versus the sum of the mana for the opponent's cards in the battlefield:

$$BM = \sum_i (m_i) - \sum_j (\hat{m_j}) \tag{4.8}$$

where m_i is the mana cost of the friendly minion i and $\hat{m_j}$ the mana cost of the enemy minion j. According the work in [2], this metric is a better indicator than the number of minions, as the author implied that a 3-cost minion is more powerful that 2 minions with cost 1.

◇

Several of the features above, were already used in the literature, although in a different setting (see Chapter 3). The value of the weights was optimized using EA, by having a myopic greedy agent, driven only by the heuristic, play numerous games against the greedy Metastone player [36–38].[2] As expected, the weights depend greatly on both decks. However, since the opponent's deck is unknown, we optimized our weights against different decks, selecting the configuration that performs best against all decks in average (see Appendix B).

4.5.2 Heuristic 2

As a second heuristic, we use the one driving the Greedy and the GSV Metastone's players. This heuristic takes the form in (4.2), includes similar metrics than the ones used by heuristic 1, also adding new ones, that account for example, the hero's life points. According the hero's life points, the heuristic adapts to a certain strategy (e.g. defensive or offensive).

4.6 Computational Budget

When the computational budget is reached, MCTS stops the search and returns the best action associated to the root node. As mentioned in Chapter 2.4.5, in Hearthstone, a turn last a maximum of 75 seconds, where players can perform multiple moves each turn.

So, following this line of thinking, MCTS needs to return different actions, in different instances of the algorithm and during the same turn. As an example, and assuming the turn just began, initially is started a new instance of MCTS, by creating the root node t and started the search. Then, and after the computational budget has been reached, is returned the best child action a_t, associated to the root t. So, if a_t doesn't represent the End Turn, MCTS stills remain the active player, being then solicited to return a new action (what will initiate a new instance of MCTS).

So, our approach, approximate the different instances performed by MCTS, where in each of them is returned a single action, to the maximum time of 75 seconds, for completing the turn. In other words,

[2]In fact, our myopic greedy agent is similar to Metastone's, although using a different heuristic.

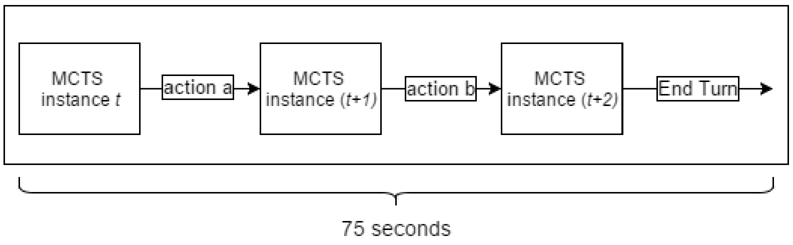


Figure 4.4: Different MCTS instances during the max time's turn

the MCTS player planes and returns all the actions, during her respective turn, in a maximum time of 75 seconds (see Fig 4.4).

To do so, before a new instance of MCTS is started, it's checked if the maximum of time per turn was already been reached. If not, a new instance of MCTS is started. Otherwise, we perform the end turn action, forcefully ending the MCTS' turn.

4.7 Search Tree Reuse

Traditionally, being an online planning algorithm, MCTS is a restarted at every new instance t of the execution, bearing as root node the state s_t of the system. In other words, at each new instance t, the agent builds a MCTS tree from state s_t for as long as it is allowed to plan; when the computation time is up, MCTS prescribes an action a_t (the action associated to the state s_t) and the system moves to a new state, s_{t+1}. The process then repeats, constructing a new tree rooted at s_{t+1}. This means that the tree constructed in one instance (e.g. instance t) (and the outcome of the corresponding simulations) are discarded, losing all the previous information, when is constructed a new tree rooted at s_{t+1}.

In order to maximize the use of information, from one instance to the next, we explore the possibility of reusing the tree, from the previous instance, in growing the new one, by continuing MCTS's search from the previous returned node.

Fig. 4.5 illustrates the point, being represented 2 distinct instances of MCTS. After the computational budget is up, the best child action of the rote node A is returned, represented in the example by the action associated to the node C. Then, in the next instance $t+1$, MCTS continues the search, by assuming the node C as the new root node, what allows the refinement of the knowledge already present. Such reuse is reminiscent of search seeding, wherein the values at each node are not started at 0.

4.8 Final Action Return

Being an anytime algorithm, when the computational budget is exhausted, MCTS returns the best candidate action at the root node, v_{root}. Following the recommendations in [1, 39], we compared 4

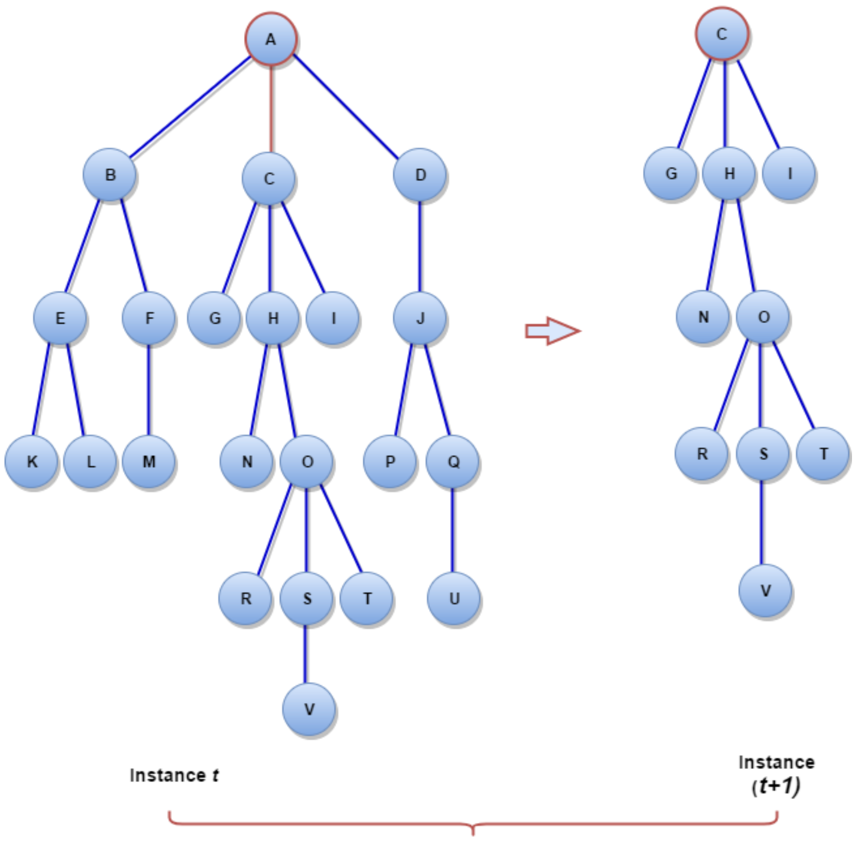


Figure 4.5: Search tree reuse example.

different criteria selection methods, for selecting the candidate action in MCTS:

- Max-child: returns the (action) child node v at the root with the highest number victories, i.e.,

$$v_{\text{vic}} = \underset{v \in C(v_{\text{root}})}{\arg\max}\; Q(v),$$

  where we write $C(v)$ to denote the set of children of v.

- Robust-child: returns the child node v at the root with the highest number of visits, i.e.,

$$v_{\text{vis}} = \underset{v \in C(v_{\text{root}})}{\arg\max}\; N(v).$$

- Max-robust-child: returns the child node v at the root with the highest combined number of victories and visits, i.e.,

$$v_{\text{rob}} = \underset{v \in C(v_{\text{root}})}{\arg\max}\; (Q(v) + N(v)).$$

- Secure-child: returns the child node v at the root that maximizes the lower confidence bound, i.e.,

$$v_{\mathrm{lcb}} = \operatorname*{arg\,max}_{v \in C(v_{\mathrm{root}})} \frac{Q(v')}{N(v')} - c\sqrt{\frac{2 \ln N(v)}{N(v')}}.$$

5

Parameter Selection Methodology

Contents

As seen in the previous chapter, our proposed approach to Hearthstone, includes a number of adaptations whose impact should be tested. In addition, the different works performed with MCTS for Hearthstone, never considered or studied such parameters, performing an empirical adjustment according the computational time of the algorithm.

So, in this chapter, we explain the conducted validation process, aimed at establishing the impact, in our approach's performance, of the:

- number of iterations: the number of nodes available in the tree. With more nodes, the algorithm becomes more knowledgeable about the domain;
- number of roll-outs per iteration: the number of independent simulations performed to a leaf node. By increasing this parameter, its possible to perform more quickly estimations of the current expanded node, although, the amount of computational time quickly grows in such conditions;
- value of sampling the parameter k in the simulation: the number of k actions randomly selected from the available ones. By increasing k, is expect the simulation to become more knowledgeable about the domain;
- heuristics used in the tree and default policies: 2 different heuristics are used, with different gaming metrics;
- action selection criterion: when the computational budget is exhausted, its tested 4 different methods, that considered the different node's statistics implemented;
- search tree reuse: using the previous tree knowledge, in the construction of the new tree.

To correctly optimize these parameters, it's necessary to perform all the possible combinations between them. Due the large number of possibilities and variables present, we investigated the impact, of each of the above components, by varying one while keeping the other fixed (details of the parameters used in the different experiments can be found in appendix A). For example, if is currently being tested the number of iterations, the other parameters not related are fixed, being only changed such parameter.

These experiments, aims to find the best configuration to use, regarding overall performance and computational time. The last one is particularly important, since a configuration with good performance but huge computational time it's not ideal

For each configuration, we measure both the performance against Metastone's GSV (see Chapter 2) and the computational time.[1] Computational times were measured on a 2.6GHz Intel Core i7 processor with 16GB of RAM memory.

The base configuration for the validation process is:

[1]As mentioned before, the computational time is an important performance measure to consider, since Hearthstone players have a limited amount of time to play.

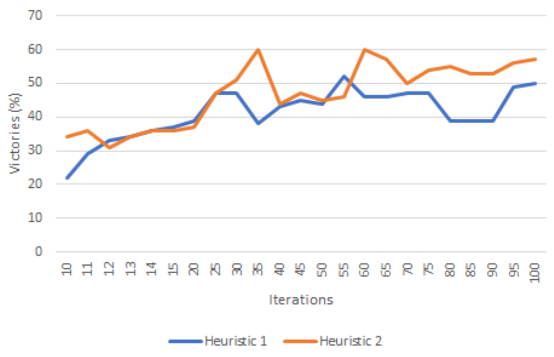
- **Player 0 (our approach):**
  - *Hero:* Warlock hero;
  - *Deck:* Tarei's warlock zoo deck.
- **Player 1 (Metastone's GSV):**
  - *Hero:* Warlock hero;
  - *Deck:* Tarei's warlock zoo deck.

5.1 Number of Iterations

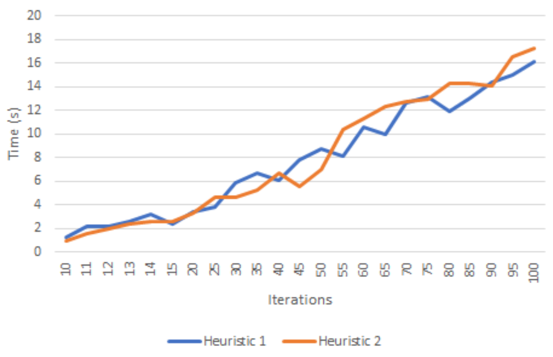
In a first test, we evaluated the impact of the number of iterations allowed to MCTS both in terms of time and game performance, using each of the two simulation heuristics. The results are summarized in Fig. 5.1, and correspond to averages over 250 independent runs. We report as performance the percentage of games won against Player 1 (Metastone's GSV player), and as time the average time-per-play.

Several observations are in order. First, both heuristics perform similarly in terms of computation time. This is not surprising, since they both involve a small number of operations. There is a slight overhead in Heuristic 2, since it requires the computation of a larger number of features, but the difference is not significant.

In terms of performance, however, Heuristic 2 does seem to offer an advantage, that tends to increases with the number of iterations. This is also not surprising, since Heuristic 2 includes more information than Heuristic 1. Also unsurprisingly, this effect is negligible when the number of iterations of MCTS is small (i.e., the tree is shallow), but increases with more interactions.

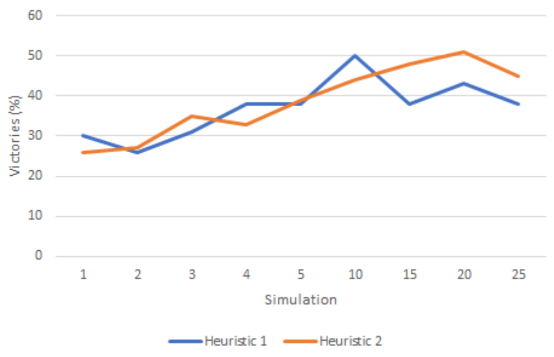


(a) Performance vs. number of MCTS iterations.

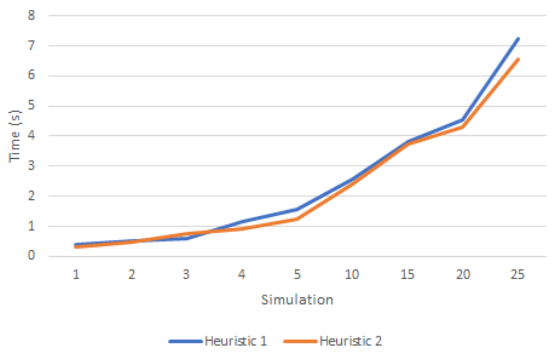


(b) Time vs. number of MCTS iterations.

Figure 5.1: Game and computational performance of our proposed approach as a function of the number of MCTS iterations. Results correspond to averages of 250 independent runs.



(a) Performance vs. number of roll-outs during simulation.



(b) Time vs. number of roll-outs during simulation.

Figure 5.2: Game and computational performance of our proposed approach as a function of the number of roll-outs performed during the simulation stage. The results correspond to averages of 250 independent runs.

Finally, we note in Fig. 5.1(a) that there is some variability in the observed performance. Such variability can be explained by the chance aspects of the game, since two games played exactly with the same decks may turn out to be very different.

5.2 Number of Roll-outs

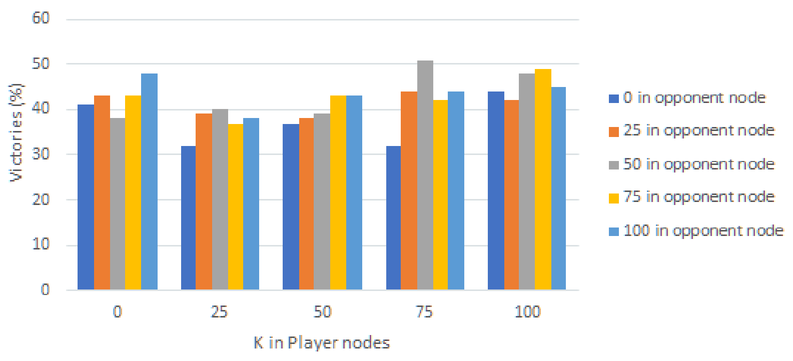
A second parameter that influences the performance of our approach is the number of roll-outs performed per iteration in the simulation stage of the game. The results are summarized in Fig. 5.2, and correspond to averages over 250 independent runs. We again report as performance the percentage of games won against Player 1 (Metastone's GSV player), and as time the average time-per-play.

The results are qualitatively similar to those observed in Section 5.1, with both heuristics performing equivalently in terms of computation time, while Heuristic 2 showing a small advantage in terms of performance.

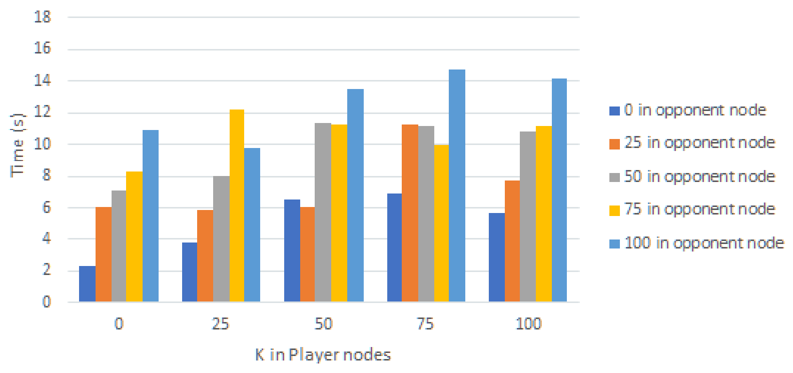
A curious observation is that both heuristics seem to drop somewhat in performance as the number of simulations grows beyond 20. While this may simply be due to the inherent stochasticity of the game, it may also be the case that the large number of simulations makes the UCB heuristic too greedy too soon, preventing sufficient exploration.

5.3 The Parameter K

We also investigated the impact of parameter k (which is combined with the heuristic to control the default policy during simulation) on the algorithm—again both in terms of performance and computation



(a) Performance vs. k for Heuristic 1.



(b) Computation time vs. k for Heuristic 1.

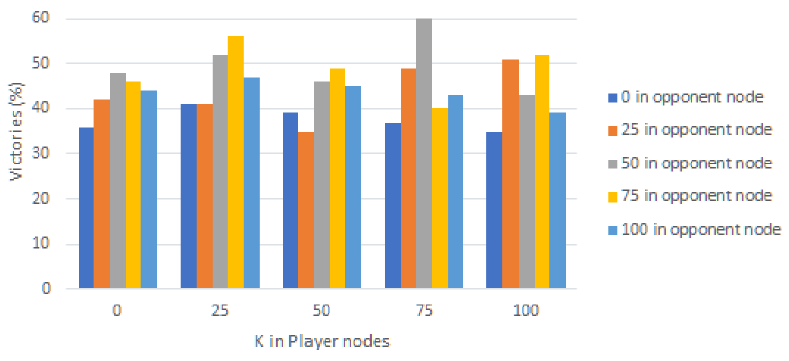
Figure 5.3: Game and computational performance of our approach as k varies between 0% and 100% of the admissible actions. The results correspond to averages of 250 independent runs.

time. We varied k between 0% and 100% of the admissible actions, both in the nodes of Player 0 (where the action is selected to maximize the heuristic) and in those of Player 1 (where the actions are selected to minimize the heuristic). The corresponding results are reported in Fig. 5.3 for Heuristic 1 and Fig. 5.4 for Heuristic 2.

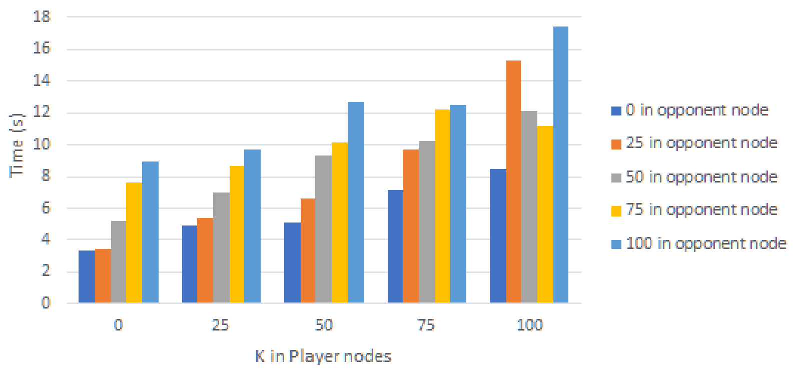
Regarding performance, two observations are in order. First, the performance of our agent does not change significantly with k. A second observation is that the best results are achieved with different values of k for the Player 0 nodes and the Player 1 nodes, namely when $k_0 = 75\%$ and $k_1 = 50\%$.[2]

We also note that extreme values of k (for example, $k_0 = k_1 = 0\%$ or $k_0 = k_1 = 100\%$) lead to poorer performance—in one case because the heuristic is not used and MCTS is, therefore, unable to properly handle the large branching factor of the game, while in the other the simulation is bound to the heuristic and unable to properly handle the differences between the predicted and actual behaviors of the opponent.

[2]We write k_i to denote the value of k for player i.



(a) Performance vs k for Heuristic 2.



(b) Computation time vs k for Heuristic 2.

Figure 5.4: Game and computational performance of our approach as k varies between 0% and 100% of the admissible actions. The results correspond to averages of 250 independent runs.

We conclude by noting, from Fig. 5.3(b) and Fig. 5.4(b), that the amount of computational time required grows with k since, for larger values of k, the algorithm must go over a larger number of alternatives and select the best, according to the heuristic.

5.4 Final Action Return

We also compared the performance of the different action selection alternatives discussed in Chapter 4. The results are summarized in Fig 5.5, and indicate that:

- Max-child selects the action with most victories, and is the best-performing action-selection criterion.
- Robust-child selects the most visited action. However, since the correlation between the number of visits and its impact towards victory is a less direct indicator of the quality of an action, the resulting performance is, expectedly, worse. In our experiments, we face situations that nodes with

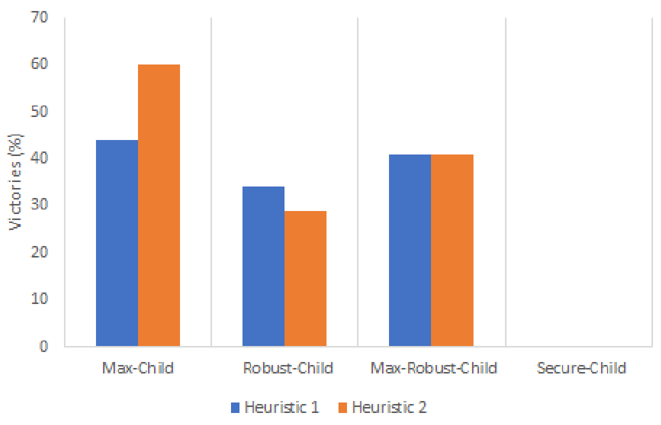


Figure 5.5: Performance of our proposed approach with the different action selection criteria. The results correspond to averages of 50 independent runs.

extremely high visits but lower victories were even selected, which may explain the performance drop.

- Max-robust-child selects the action that jointly maximizes the number of visits and victories. Interestingly, its performance lies exactly in the middle between the Max- and Robust-child players.
- Finally, the secure-child is far too conservative and is unable to ever lead to a winning state.

5.5 Search Tree Reuse

Finally, we investigated the impact of tree reuse in the performance of the algorithm. In particular, we compared a first policy (police 1) obtained when the tree is reused between different instances and a second policy (police 2) obtained when the search tree is rebuilt at each new instance.

The results are depicted in Fig. 5.6, and clearly show that tree reuse does, in fact, lead to improved performance, reaching a maximum gap of 15% if used heuristic 2 and police 1. Regarding policy 2, the same results are verified, but without a gap so larger between the 2 heuristics as seen earlier.

In general, such results suggest that policy 1 performs better, by allowing the refinement of the knowledge already present in the game tree.

5.6 Conclusions

In this chapter, we have seen the importance that the parameters used by MCTS, can have in the algorithm's performance. We conclude by presenting a brief overview of the results, followed by the final parameters chosen, to be used in the final evaluation:

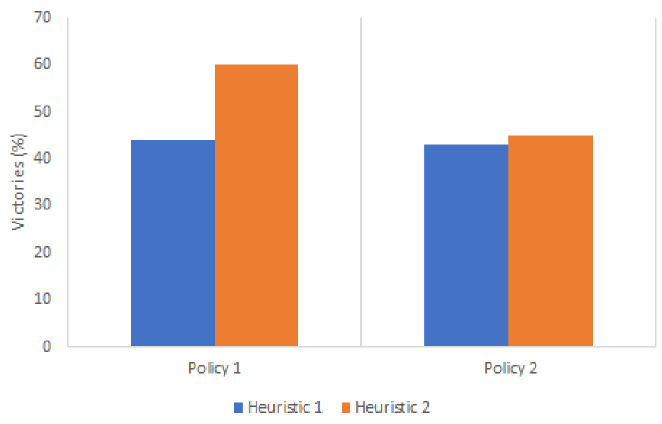


Figure 5.6: Impact in performance of tree reuse. Policy 1 corresponds to the policy obtained when the search tree is maintained between execution steps. Conversely, Policy 2 is obtained by rebuilding the tree at each execution step.

- Number of iterations: it's noticeable that the performance improved between iterations. Indeed, as we increased this parameter, the game tree becomes larger and knowledgeable about the domain, being progressively added new nodes.

- Number of roll-outs: This step of MCTS, aims to estimate the current expanded node value. So, in theory as we increased this parameter, more quickly can a node be estimated. On the other hand, our approach integrates heuristic knowledge, that in situations where a larger value of roll-outs are being used, the simulation became too greedy, preventing sufficient exploration.

- The parameter K: An informed simulation strategy generally performed better. Analyzing the results, using K equal to 0%, achieves worse results than other K configurations, regardless the player's node or the heuristic to use. Another similar behavior is the performance drop when it's used a K equal to 100%. As expected in such conditions, the simulation becomes too greedy, being even contrary to the MCTS principals of randomness, for dealing with large branching factors.

- Action return: Analyzing the different methods, the max child is the one that accomplishes the best results. As this method chooses the candidate action based on victories, being victories directly related to the node's performance, this result followed the expected.

- Search tree reuse: Was clear, that continuing iterating performs better results than not to, suggesting that this approach allowed the refinement of the knowledge already present in the search tree.

5.6.1 Final parameters

- Algorithm: Monte Carlo tree search

- Heuristic: 2;
- Hero: (Target hero);
- Deck: (Target deck);
- N. iterations: 60;
- N. simulations: 20;
- Parameter k: 75% for player 1; 50% for player 2;
- Action selection criterion: Max-child;
- Tree reuse: yes.

6

Results

In this chapter, the discussion will point to the conducted validation process aimed to appreciate the improvements of our approach, regarding the actual state of the art for Hearthstone.

◇

In order to validate the improvements and contributions of our approach into MCTS, we paired it with the ”vanilla” version of the algorithm and against the different AIs in Metastone (see Chapter 2):

- Random: Selects actions at random and provides a naive baseline for comparison;
- No Aggression: Does not attack the opponent. This AI randomizes between playing cards or simply performing the “end-turn” action;
- Greedy: Myopic player whose actions are driven by a heuristic (heuristic 2), built on several game metrics, and whose weights were tuned using an evolutionary approach (see Chapter 4).
- GSV: Player that uses a recursive alpha-beta algorithm driven by the aforementioned heuristic (heuristic 2). To the extent of our knowledge, this is the best AI system available in Metastone, and several existing players report disappointing performances against it [4].

where the different AIs represented different competitive levels of difficulty. In addition, we also studied how our approach adapted in different gaming scenarios. To do so, we used 3 different decks, that represented a wide variety of gaming strategies:

- Tarei’s Warlock Zoo: A moderate aggro-based deck that aims to control the board while damaging the opponent.
- JasonZhou’s N’Zoth Control Warrior: control-based deck, one of the most consistent in Hearthstone. It aims to exhaust the opponent’s resources, dominating in late-game turns.
- Che0nsu’s Yogg Tempo Mage: Mid-range based deck, consisting of heavy minions with a higher curve compared to Aggro ones.

For each deck, we run a total of 250 games and record the percentage of victories. The results are summarized in Fig. 6.1, Fig. 6.2, Fig. 6.3 and Fig. 6.4, where Approach 1 is our approach and Approach 2 is the “vanilla” MCTS player. Both approaches used the deck database to handle hidden information and used the same parameters throughout (see Appendix A). [1]

As a initial test, we first decided to evaluate both approaches against the Random Player present in Metastone. Each MCTS version played into a round-robin, being used all the possible combinations of

[1] MCTS was run for 60 iterations per play, 20 simulations per node. Both approaches used sampling in simulation with $k_0 = 75\%$ and $k_1 = 50\%$, Max-child output selection and tree reuse.

decks between them. Since the complexity of the game prevents the Random Player from any meaningful play, not surprisingly, both approaches almost crushed the opponent, performing a win-rate near of 100% in every scenario.

In a second test set, we used the No Aggression Player as opponent. Since this AI in particular, randomizes between fishing her turn and playing cards, not participating in attack actions, we faced similar results as the ones seen previously, for both approaches.

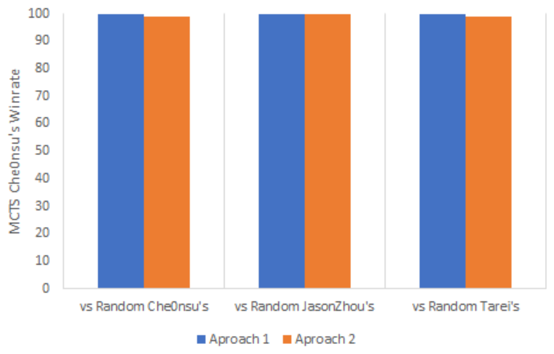
In a third test set, we tested the performance against the Greedy Player. Several observations are in order:

- Since this AI represents an myopic player, whose actions are driven by a heuristic, our MCTS adaptation clearly out-performed the opponent, presenting a win-rate in average of 61%;
- In average, the "vanilla" MCTS version presented a win-rate close to 40%, being a gap close to 20%, between our adaptation;
- An exception, is the one seen in Fig.6.3(b), when the opponent uses the Che0nsu's deck. In such conditions, the deck used by MCTS, is not particularly suitable against the one used by the opponent. However, our adaptation even overcomes, but this time, by a close margin of 4%.

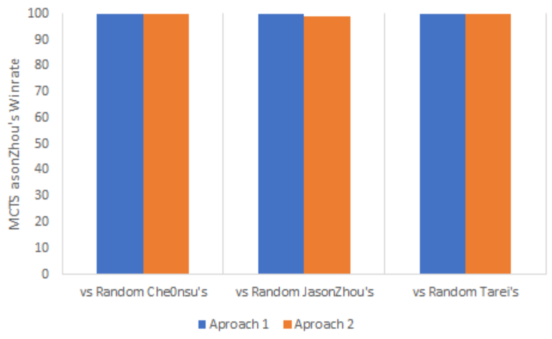
Finally, we tested the performance of both approaches against the actual-state-of-the-art of Metastone's AI, the GSV:

- As expected, against a more competitive player, some decks were better matched than others, which explained the variation across the different decks;
- In average, our adaptation to MCTS performed a win-rate close to 42%;
- Overall, the MCTS "vanilla" version presented an win-rate close to 21%, being a gap close to 20%, considering our adaptation;
- Another important observation, is that the fact that the heuristic parameters are optimized to perform well against all decks, what eventually hampers the performance of the agent (as seen, for example, with the JasonZhou's deck).

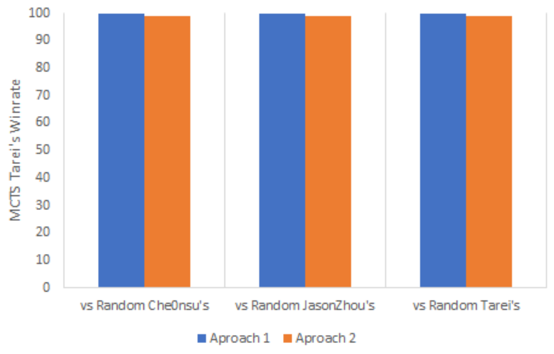
In any case, against the different strategies and AIs opponents, our results show that the MCTS approach with expert knowledge clearly outperformed the "vanilla" approach in all situations, often by a large margin, proving that MCTS with the integration of expert knowledge is able to attain more competitive results.



(a) Che0nsu's deck

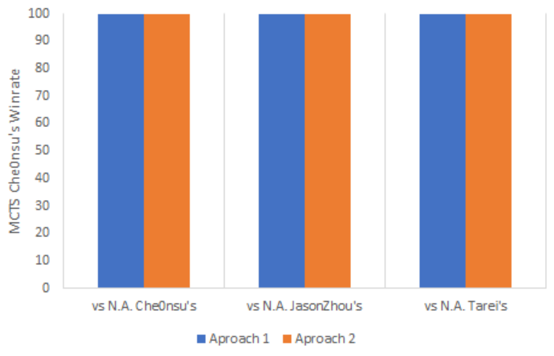


(b) JasonZhou's deck

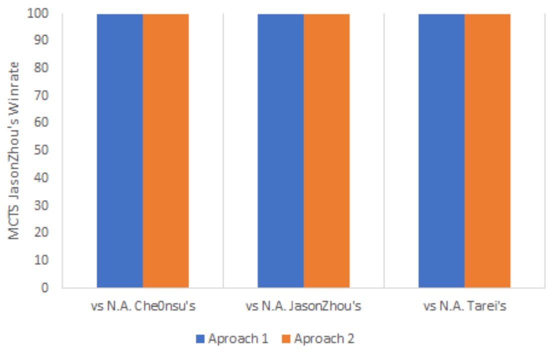


(c) Tarei's deck

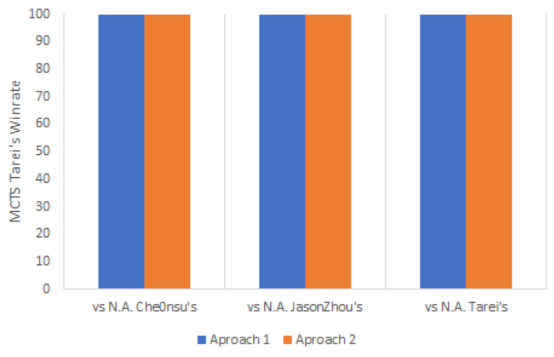
Figure 6.1: Performance of MCTS players against Metastone's Random player with different decks.



(a) Che0nsu's deck

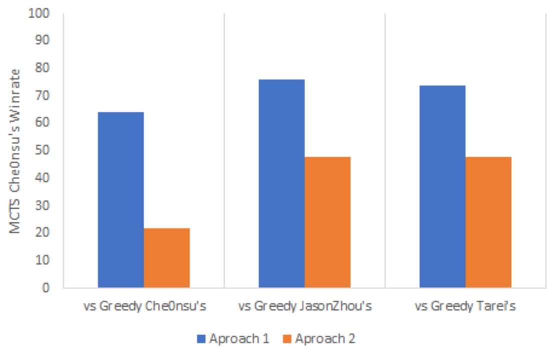


(b) JasonZhou's deck

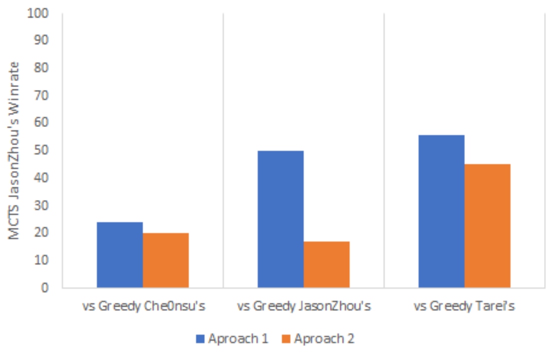


(c) Tarei's deck

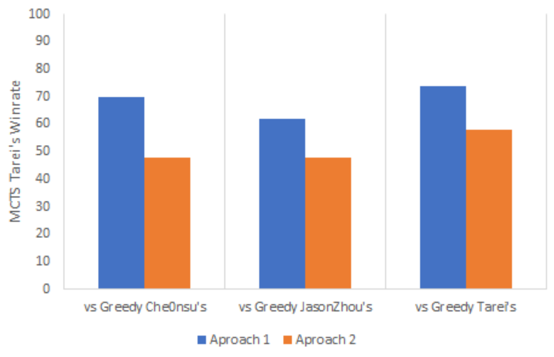
Figure 6.2: Performance of MCTS players against Metastone's No Aggression player with different decks.



(a) Che0nsu's deck

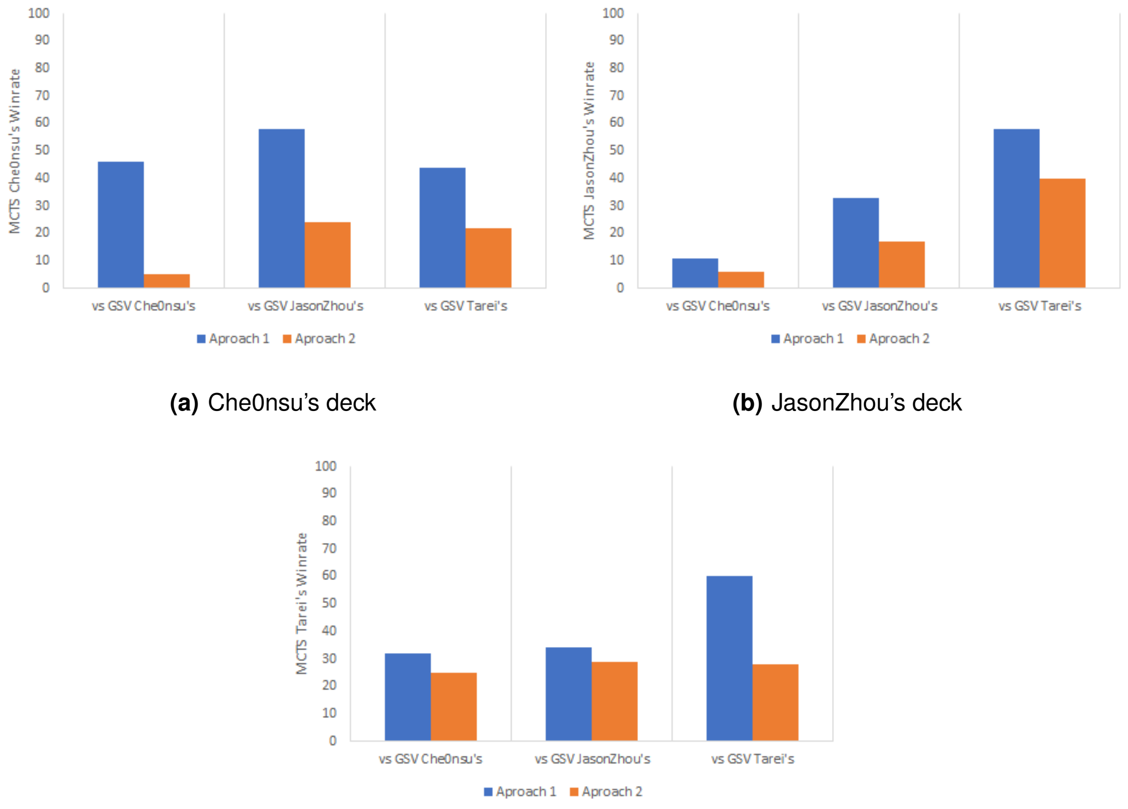


(b) JasonZhou's deck



(c) Tarei's deck

Figure 6.3: Performance of MCTS players against Metastone's Greedy player with different decks.



(a) Che0nsu's deck

(b) JasonZhou's deck

(c) Tarei's deck

Figure 6.4: Performance of MCTS players against Metastone's Game State Value player with different decks.

7 Conclusions and Future Work

Contents

7.1 Conclusions

In the beginning of this work, we have seen that the AI in Hearthstone, suffered limitations that directly affected the gaming experience. In order to improve such experience, several works were performed with different methodologies and motivations, that considered different aspects of the game.

From all methodologies developed, the use of MCTS for general game play, clearly stands out in Hearthstone. As stated in Chapter 2 and 3, the results and behaviors achieved in recent applications, and the ability to be used without domain specific knowledge, are examples of benefits that lead this algorithm to be considered as a new framework for AI in games. In addition to such benefits, and the possibility of improving the algorithm's performance with expert knowledge, are benefits to be taken into account.

So, in this work, we developed an MCTS-based approach for Hearthstone. Our approach was boosted MCTS with domain specific knowledge of 2 types: a database of decks that mitigates the impact of imperfect information, and a heuristic that guides the tree construction. Regarding the heuristics, and since it represents a sum of different game metrics, we used an EA approach that targeted a specific deck to learn the weights to use.

During experiments, we conducted a simple grid search across the space of possible values, for optimizing the MCTS's parameters. For each configuration, we measured both the performance against Metastone's GSV and the computational time.

During the same tests, we noticed some interesting behaviors. Our approach was able to quickly predict the opponent's deck, performed good trades, played correctly cards in the right position and even hurt himself if needed, for gathering attack bonus in minions with $enrage$ ability.

Our results, shown that our approach is superior to “vanilla” MCTS, being able to attain competitive results with state of the art AI for Metastone. In particular, our approach achieved very good results against a Random player, performing a win-rate close to 100% in every case scenario.

Against the more competitive player present in Metastone's, the GSV, some decks were better matched than others, which explained the variation across the different decks. Nevertheless, our approach achieved an win-rate optimal near to 60%, representing a performance quite superior to the one presented in [4] of just 9%.

To conclude, in any case, our results show that the MCTS approach with expert knowledge clearly outperformed the “vanilla” approach in all situations and often by a large margin. The available evidence, seems to suggest that MCTS with the integration of expert knowledge is able to attain more competitive results than his ”vanilla” version.

7.2 Future Work

Our results also open avenues for future research. First we would like to recommend to update the simulator used in this work to the latests version. Being Metastone a stable platform, also presents bugs and features that clearly influenced our MCTS's approach. In addition to such reasons, is the fact that the latest expansions, release in Hearthstone, will be available in Metastone, upgrading the development environment for the continuation of this work.

Secondly, we strongly encourage the use of any alternative methodologies, to optimize the MCTS' default policy, where this particularly phase performed a significant role in MCTS computational time. For example, since the roll-outs performed, at the recently expanded nodes, are independent from one and another, we recommend to start all the roll-outs at the same time (e.g. threads), updating the node's statistics after such effort.

Along similar lines, we recommend to study different selection methods of EA, to improve the integration of expert knowledge into the default policy (see Chapter 4.3.2). During experiments, we faced the successful application of these methods, because in particularly, they merged stochastic information with expert knowledge, allowing the refinement of this peculiarly stage.

Also, we would like adapt new Progressive Strategies, namely Progressive Unpinning, for studying how the branching factor influences MCTS (see Chapter 2.3.2 and 4.2.1). During the research conducted, we have seen that MCTS performs poorly, when the branching factor is high and the computational budge is low. So, the idea is to reduce artificially the branching factor in the beginning of MCTS, increasing progressively with the number of iterations [40] (also including the Progressive Bias method).

Regarding particularly the expert knowledge, in our approach, the heuristic governing the tree construction uses a single set of weights that is deck-dependent. We observe that an improvement in performance could probably be achieved by considering specialized weights for the deck or even by using a more specialized EA method for the learning phase (see the Fast-Evolutionary MCTS in Chapter 2.3.3).

Still considering the expert knowledge, we would like to develop our own heuristic, with a dynamic approach, for targeting a specific game strategy. The dynamic approach should be directly influenced by the actual HP, in which for instance, when in low HP conditions, a defensive strategy should be used, and otherwise, an attacking instead. Hopefully, this would allow to optimize MCTS for specific behavior, instead a specific deck.

Additionally, we desire to study new methods for selecting the candidate root action in MCTS. In our experiments, we only adapted approaches that returns a single action. The idea would be to optimize the MCTS' turn, by selecting the best moves, from the root node to the end turn, that maximizes a certain criteria (e.g. number of victories).

It's also important to investigate new forms to deal with the principal characteristic of MCTS, the variance of results. During experiments, we faced this problem , having some discrepancy in the results obtained. As such, we performed series of 250 games, averaging the results obtained, what really slowed our validation process.

Last but not least, we really suggest to perform tests with professional player of Hearthstone, and preferably with different ranks. As seen in Chapter 3, MCTS could be adapted with different competitive players, so the idea, would pass to study the parameters to use in different players with different ranks.

Bibliography

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Computational Intelligence and AI in Games*, vol. 4, no. 1-49, 2012.

[2] E. Bursztein, "I am a legend: Hacking "Hearthstone" using statistical learning methods," in *Proc. 2016 IEEE Int. Conf. Computational Intelligence in Games*, 2016.

[3] ——. (2016, April) How to find undervalued Hearthstone cards automatically. Accessed 15-January-2016. [Online]. Available: https://www.elie.net/blog/hearthstone/how-to-find-automatically-hearthstone-undervalued-cards

[4] G. Tzourmpakis. (2014, April) HearthAgent, A Hearthstone Agent, Based on the Metastone project. Accessed 15-January-2017. [Online]. Available: http://www.intelligence.tuc.gr/~robots/ARCHIVE/2015w/Projects/LAB51326833/

[5] O. G. Alliance. (2017, May) Global Game Software Market Forecasted To Reach $100 Billion In 2019. Accessed 10-May-2017. [Online]. Available: https://opengamingalliance.org/press/details/global-game-software-market-forecasted-to-reach-100-billion-in-2019

[6] A. El Rhalibi, K. W. Wong, and M. Price, "Artificial intelligence for computer games," *International Journal of Computer Games Technology*, 2009.

[7] I. Millington and J. Funge, *Artificial Intelligence for Games, Second Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[8] R. Steinman and M. Blastos, "A trading-card game teaching about host defense," *Medical Education*, vol. 36, no. 12, pp. 1201–1208, 2002.

[9] T. Denning, A. Lerner, A. Shostack, and T. Kohno, "Control-alt-hack: The design and evaluation of a card game for computer security awareness and education," in *Proc. 2013 ACM-SIGSAC Conf. Computer and Communications Security*, 2013, pp. 915–928.

[10] W. Ling, E. Grefenstette, K. Hermann, T. Kǒciský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," in *Proc. 54th Annual Meeting of the Assoc. Computational Linguistics*, 2016, pp. 599–609.

[11] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. Senior, "Latent predictor networks for code generation," *CoRR*, vol. abs/1603.06744, 2016.

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

[13] P. Cowling, C. Ward, and E. Powley, "Ensemble determinization in Monte Carlo tree search for the imperfect information card game "Magic: The Gathering"," *IEEE Trans. Computational Intelligence and AI in Games*, vol. 4, no. 4, pp. 241–257, 2012.

[14] Polygon. Hearthstone now has 50 million players.

[15] P. García-Sánchez, A. Tonda, G. Squillero, A. Mora, and J. Merelo, "Evolutionary deck-building in "Hearthstone"," in *Proc. 2016 IEEE Int. Conf. Computational Intelligence in Games*, 2016.

[16] D. Taralla, "Learning artificial intelligence in large-scale video games," Master's thesis, Faculty of Engineering, University of Liège, 2015.

[17] B. Schwab. (2014, April) Building the AI for Hearthstone. Accessed 25-May-2015. [Online]. Available: http://www.gamasutra.com/view/news/224101/Video_Building_the_AI_for_Hearthstone.php

[18] S. Hill and E. Merrill. (2016, April) Intelligent agents for Hearthstone. Accessed 15-January-2017. [Online]. Available: http://rebrn.com/re/intelligent-agents-for-hearthstone-1986863/

[19] R. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *Proc. 21st Int. Joint Conf. Artificial Intelligence*, 2009, pp. 40–45.

[20] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[21] A. Rimmel, O. Teytaud, C. Lee, S. Yen, M. Wang, and S. Tsai, "Current frontiers in computer Go," *IEEE Trans. Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 229–238, 2010.

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

[23] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, “Monte-carlo tree search: A new framework for game ai.” in *AIIDE*, 2008.

[24] L. Kocsis and C. Szepesvari, “Bandit based Monte-Carlo planning,” in *Proc. 17th Eur. Conf. Machine Learning*, 2006, pp. 282–293.

[25] P. Auer, N. Cesa-Bianchi, and P. Fisher, “Finite-time analysis of the multiarmed bandit problem,” *Machine Learning*, vol. 47, pp. 235–256, 2002.

[26] M. Ponsen, G. Gerritsen, and G. Chaslot, “Integrating opponent models with monte-carlo tree search in poker,” in *Proceedings of the 3rd AAAI Conference on Interactive Decision Theory and Game Theory*, ser. AAAIWS’10-03. AAAI Press, 2010, pp. 37–42.

[27] B. Bouzy and U. R. Descartes, “Monte-carlo go reinforcement learning experiments,” in *In IEEE 2006 Symposium on Computational Intelligence in Games*. IEEE, 2006, pp. 187–194.

[28] S. M. Lucas, S. Samothrakis, and D. Perez, “Fast evolutionary adaptation for monte carlo tree search,” in *European Conference on the Applications of Evolutionary Computation*. Springer, 2014, pp. 349–360.

[29] D. Perez, S. Samothrakis, and S. Lucas, “Knowledge-based fast evolutionary mcts for general video game playing,” in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[30] M. H. Andersson, “Programming a Hearthstone agent using Monte Carlo Tree Search,” Master’s thesis, Norwegian University of Science and Technology, 2016.

[31] P. Lynch, “Hearthstone Critical Eye,” University of Virginia, PDF, May 2015. [Online]. Available: https://cs4730.cs.virginia.edu/materials/HearthstoneCriticalEye.pdf

[32] Metastone. Metastone Simulator.

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

[34] R. Coulom, “Efficient selectivity and backup operators in monte-carlo tree search,” in *Proceedings of the 5th International Conference on Computers and Games*, ser. CG’06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 72–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=1777826.1777833

[35] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, “Modification of UCT with Patterns in Monte-Carlo Go,” INRIA, Research Report RR-6062, 2006. [Online]. Available: https://hal.inria.fr/inria-00117266

[36] B. Miller and D. Goldberg, “Genetic algorithms, tournament selection, and the effects of noise,” *Complex Systems*, vol. 9, pp. 193–212., 1995.

[37] T. Blickle and L. Thiele, “A comparison of selection schemes used in evolutionary algorithms,” *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[38] M. Negnevitsky, *Artificial Intelligence: A Guide to Intelligent Systems*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[39] F. Schadd, “Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis,” Master’s thesis, Department of knowledge Engendering, Maastricht University, 2009.

[40] G. Chaslot, M. Winands, H. van den Herik, J. Uiterwijk, and B. Bouzy, “Progressive strategies for Monte-Carlo tree search,” *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

A

Parameters used in experimenters

In this appendix, we specify the parameters used in the different experiments reported in the paper. Player 1 refers to the MCTS player, being always held fixed and described in Chapter 5.

A.1 Number of Iterations

- Algorithm: Monte Carlo tree search
- Heuristic: Both 1 and 2;
- Hero: Warlock hero;
- Deck: Tarei's warlock zoo deck;
- N. iterations: every 10 between 10 and 100;
- N. simulations: 20;
- Parameter k: 75% for player 1; 50% for player 2;

- Action selection criterion: Max-child;
- Tree reuse: yes.

A.2 Number of Roll-outs

- Algorithm: Monte Carlo tree search
- Heuristic: Both 1 and 2;
- Hero: Warlock hero;
- Deck: Tarei's warlock zoo deck;
- N. iterations: 30;
- N. simulations: 1 through 35;
- Parameter k: 75% for player 1; 50% for player 2;
- Action selection criterion: Max-child;
- Tree reuse: yes.

A.3 Action Selection

- Algorithm: Monte Carlo tree search
- Heuristic: Both 1 and 2;
- Hero: Warlock hero;
- Deck: Tarei's warlock zoo deck;
- N. iterations: 60;
- N. simulations: 20;
- Parameter k: 75% for player 1; 50% for player 2;
- Action selection criterion: $\{$Max-child, Robust-child, Max-robust-child, Secure-child$\}$;
- Tree reuse: yes.

A.4 Search Tree Reuse

- Algorithm: Monte Carlo tree search
- Heuristic: Both 1 and 2;
- Hero: Warlock hero;
- Deck: Tarei's warlock zoo deck;
- N. iterations: 60;
- N. simulations: 20;
- Parameter k: 75% for player 1; 50% for player 2;
- Action selection criterion: Max-child;
- Tree reuse: $\{$yes, no$\}$.

A.5 Parameter k

- Algorithm: Monte Carlo tree search
- Heuristic: Both 1 and 2;
- Hero: Warlock hero;
- Deck: Tarei's warlock zoo deck;
- N. iterations: 60;
- N. simulations: 20;
- Parameter k: $\{$0%, 25%, 50%, 75%, 100%$\}$ for player 1; $\{$0%, 25%, 50%, 75%, 100%$\}$ for player 2;
- Action selection criterion: $\{$Max-child, Robust-child, Max-robust-child, Secure-child$\}$;
- Tree reuse: yes.

A.6 Final Parameters

- Algorithm: Monte Carlo tree search
- Heuristic: 2;
- Hero: (Target hero);
- Deck: (Target deck);
- N. iterations: 60;
- N. simulations: 20;
- Parameter k: 75% for player 1; 50% for player 2;
- Action selection criterion: Max-child;
- Tree reuse: yes.

B
Genetic algorithm parameters

We now describe the parameters of the genetic algorithm used to optimize the weights in the heuristic (4.3).

- Player 1 AI: Greedy;
- Player 1 Deck (Target deck): “Tarei’s warlock zoo deck”;
- Player 2 AI: Metastone’s Greedy;
- Player 2 Decks: {“Amnesiac´s Secret Face”; “Jasonzhou’s N’zoth Control Warrior”; “Frozen’s N’zoth Paladin”; “Yulsic’s Midrange Shaman”; “Che0nsu’s Yogg Tempo Mage”; “Tareis’s Warlock Zoo”; }
- Population size: 50;
- Population restarts: 10;
- N. games used to compute fitness: 30;
- Selection methods: {Random, Rank, Roulette, Tournament};

- Crossover probability: 0.7;
- Mutation probability: 0.01;

The resulting weights were

$$\boldsymbol{\alpha} = \begin{bmatrix} 1.0457 & 0.2058 & 9.6557 & 0.9407 & 5.2249 \end{bmatrix}.$$