ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Graduado en Ingeniería Informática

Análisis de la técnica 'Monte Carlo Tree Search' aplicada como Inteligencia Artificial en el videojuego 'Hearthstone: Heroes of Warcraft'

Analysis of the technique 'Monte Carlo Tree Search' applied as Artificial Intelligence in the videogame 'Hearthstone: Heroes of Warcraft'

Realizado por

Álvaro Florencio de Marcos Alés

Tutorizado por

Antonio José Fernández Leiva

Pablo García Sánchez

Departamento

Lenguajes y Ciencias de la Computación

Ingeniería Informática (Universidad de Cádiz)

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2019

Fecha defensa: julio de 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

El objetivo que se persigue con este trabajo es analizar el rendimiento del método Monte Carlo Tree Search (MCTS), en el entorno del juego de cartas coleccionables Hearthstone: Heroes of Warcraft^{TM}, buscando maximizar el número de victorias contra distintos oponentes. Para cumplir dicha tarea se ha realizado una versión muy parametrizada del método, que nos permitirá probar distintas versiones y analizar según sus valores de entrada, sus resultados.

Palabras clave: Inteligencia artificial, Monte Carlo Tree Search, parámetros, Hearthstone:Heroes of Warcraft^{TM}.

Abstract

The objective pursued with this project is to analyse the performance of the method Monte Carlo Tree Search(MCTS), using the collectible card game Hearthstone:Heroes of Warcraft^{TM} as environment, maximizing the number of victories against different opponents. To accomplish this goal a highly parameterized version of the method has been created, which will allow us to test different versions and to analyse theirs results according to their input values.

Keywords: Artificial intelligence, Monte Carlo Tree Search, parameters, Hearthstone:Heroes of Warcraft^{TM}.

Contents

Chapter 1

Introduction

This chapter will inform the reader about the contents of this project, its motivation and how it is structured.

1.1. Motivation

Games are considered to be a great scenario for testing artificial intelligence, as they are closed and most of them have a huge search space where the performance of complex algorithms can be analysed.

What's more, games are also the biggest entertainment industry, generating $ 137.9 billion per year [1]. It is expected to keep growing for 2020. Also, the number of people playing games is insane. As a consequence the relevance and impact of games in human's lives is astounding, from affecting popular culture to increase the speed of human's decisions making.

Its researching has also an increasing tendency, as scientists find them really interesting as a subject of study, because of the complex problems that arise from the creation of engines that offers unique experiences to the players. [2]

Furthermore, *Monte Carlo Tree Search*(MCTS) [3] has succeeded in different types of games from board games as Chest and Shogi to real-time video games such as *Total War: RomeII* campaign AI [4]. In addition, it has also been tested in games where the information was incomplete like Poker.

This, is the case of Hearthstone^{TM} [5] where the search space is overwhelming and the hand of the opponent is unknown, the aim of the project is to analyse whether the MCTS has success in terms of win rate over other algorithms or not.

It is also worth mentioning, this method can be applied to any other type of problem which search space can be represented as a tree, as a consequence its analysis will allow us to gain more insight on its usefulness.

1.2. Goals

This project main goal is to create an agent that it is able to play Hearthstone^{TM} 's games maximizing the number of victories. To accomplish it, a Monte Carlo Tree Search method will be implemented as parameterized as possible in order to test some variations of the method and compare its performance against other versions and algorithms.

1.2.1. Sub-goals

- Research information about the State of the art of Artificial Intelligence and Games.
- Implement an agent using Monte Carlo Tree Search techniques to play Hearthstone^{TM} .
- Create an environment to test the agent against other agents, with different decks.
- Analyse those results to gain some insight of why some input values lead to better or worse results.
- Finally, explain problems encountered and future ideas to improve its performance that have not been feasible.

1.3. Report structure

- Chapter 2 **Background:**

  It will be explained what Hearthstone^{TM} is, how has artificial intelligence been applied to video games and the functioning of the plain Monte Carlo Tree Search.

- Chapter 3 **Methodology, design and solution implementation**

  It will be explained the actual implementation and the techniques used to build the agent.

- Chapter 4 **Experimentation and results:**

  It will show the conditions in which the results were made, the results and their analysis.

- Chapter 5 **Conclusions:**

  It will be presented problems encountered, future improvements to the project and self learned contents.

Finally, a bibliography will close the report.

Chapter 2

Background

In this chapter, concepts and rules necessary to understand the project environment will be discussed.

2.1. Brief History of Artificial Intelligence and Games

This section aims to accomplish the researching information about AI and Games sub-goal.

Since the beginning of computers, artificial intelligence and games have been closely related, as games are a great scenario for testing AI [2] . Firstly, with Alan Turing as one of the fathers of theoretical computer science, (re)invented the *Minimax* [6] algorithm to play Chess.

Later on, Arthur Samuel invented the form of machine learning, nowadays known as Reinforcement Learning [7] and tested it by making it play Chekers against itself. It was thought board games captured the essence of thought, as with a simple set of rules they were so challenging for humans.

After several decades researching tree search techniques, in 1994 *Chinook* [8] an agent who played Checkers beat the World Checkers Champion Marion Tinsley.

And a few later, Chess also fell in front of computers by IBM's *Deep Blue* [9], which was mainly a minimax with some handcrafted constraints.

The last milestone was reached in 2016 with the game Go, where an AI managed to win to the World Champion Ke Jie, it was developed by Google DeepMind and was called *AlphaGo* [10], it features a deep reinforcement learning approach mixed with a tree search approach specifically Monte Carlo Tree Search.

Not only does AI have been applied to board games but also is becoming popular the problem of **procedural content generation** [11] which open a huge field of opportunities for AI to explore. This phenomenon starts with the game Rogue in 1980, and still is far from disappearing with the latest approaches such as the Chalice Dungeons of *Bloodborne* (Sony Computer Entertainment, 2015) or *No Man's Sky* (Hello Games, 2016).

There are also other projects related between Hearthstone^{TM} and AI like automated playtesting through evolutionary algorithms [12] or data mining challenges. [13]

2.2. Monte Carlo Tree Search

Monte Carlo Tree Search, known also as MCTS is a best-first search technique that doesn't need a evaluation function as priors algorithms such as minimax do.

The core concept of the algorithm revolves around the idea of building a tree of possible future states of the game, throwing multitude of games simulations and learning from its results.

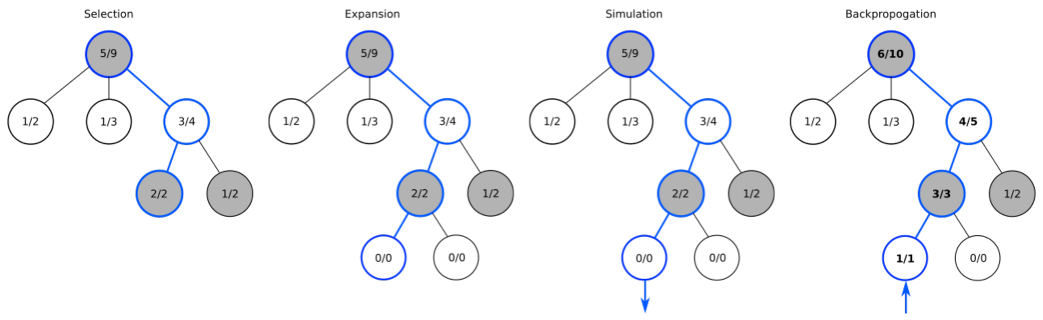


Figure 2.1: Monte Carlo Tree Search stages

A definition of its behaviour can be understood through its following stages as shown in Figure 2.1:

2.2.1. Selection

Only if the state is in the tree, a selection of which child of a certain node should be used is made. This is usually done by a mixing of exploitation of the best nodes so far, and a factor of exploration of the less visited nodes.

2.2.2. Expansion

When a leaf is found, meaning the next game state is not found, the tree adds the next state and starts to simulate.

2.2.3. Simulation

From that point it will simulate the rest of the game assigning random moves to each player until it ends, then when it reaches the end, it will return if the game was won or lost.

2.2.4. Backpropagation

All the nodes that took part in the process from the beginning of the tree down to the point where the game started to be simulated updates their values depending on the results of the simulations and always increasing the number of times each node was visited.

2.2.5. Advantages

One of the main advantages it has is not requiring an evaluation function in order to work properly as it retrieves this information from the numerous simulations until the end of the game.

Other interesting fact is, it can be stopped at any time yielding a decision based on the most promising movement already found. It has not a definite end, so a maximum time should be given.

Its tree grows asymmetrically as it balances between exploration and exploitation.

2.3. Hearthstone: Heroes of Warcraft ™

Hearthstone™ is a collectible Card Game released by Blizzard Entertainment in 2014, available for PC and phone, that had achieved over 100 million players by November 2018.

It is a turn-based card game where the player plays against other player or against an artificial intelligence, victory is achieved by reducing the health points of the foe to zero.

2.3.1. Collectible Card Games

It is a type of strategic games [14] where sets of cards are released and specifically designed to be collected and played with.

Usually this type of games first start with a classic/basic set of cards and afterwards new expansions add new cards, that interacts with the older sets changing the best decks that were currently discovered by that time.

One if not the most representative game of this type is *Magic: The Gathering* [15], released in 1993 by Wizards of the Coast, which lays the foundations of both players having a board where they could place creatures with attack and health to fight and reduce each players health points.

Nowadays this design choices can be found in many collectible card games, however at that time, it was the first of its kind. Hearthstone™ core mechanics are based on Magic's designs ideas, but has also done enormous contributions to this genre by adding fresh mechanics, solving some concepts issues and developing one of the first online and user friendly collectible card game.

2.3.2. Rules and Interface

Hearthstone^{TM} consists of two opponents battling in turns until one of theirs health points decrease to zero or less. A player can only interact with the game while it is its turn.

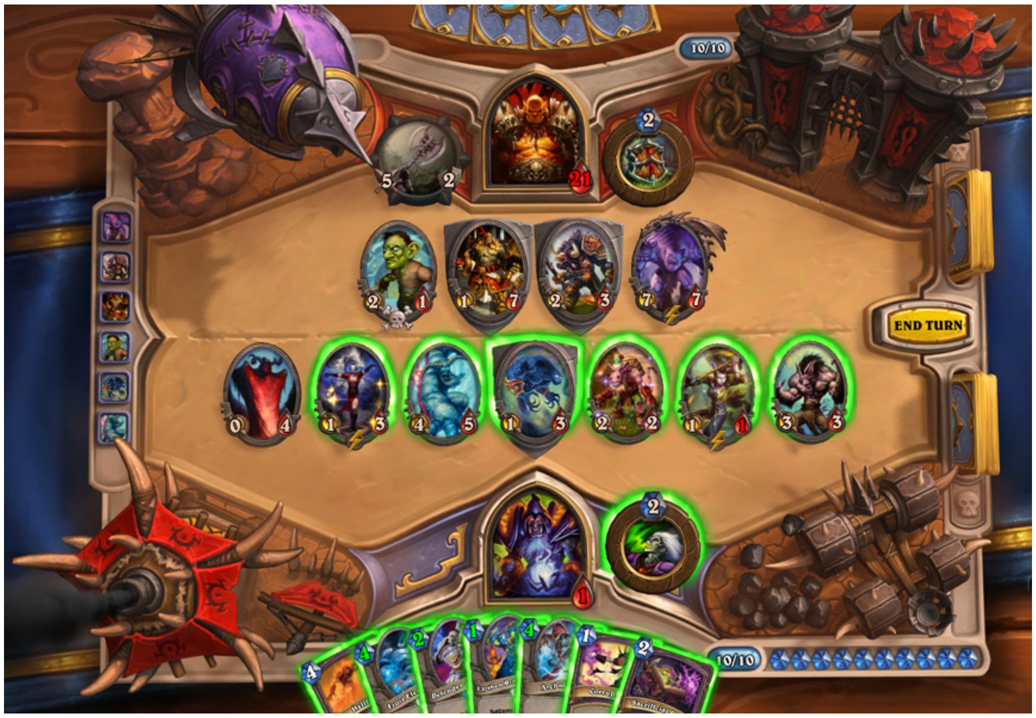


Figure 2.2: Example of a Hearthstone^{TM} game

Each player has a hand with cards, and a board where minions can be summoned or played as it can be seen in Figure 2.2. Each card has a mana cost that must be fulfilled in order to play it, each player is given a maximum mana increase when their turn starts, draw a card and their previous amount of mana refilled. The maximum amount of mana is 10. [16]

Both players start the game with a deck made of 30 cards. At the beginning, the first player starts with 3 cards and is able to choose for each card if they want it to be in their hand or throw it back to the deck and then draw a new card for each one they have returned.

The second player, to equalize the fact they start second, starts with 4 cards plus a coin, that is a special card that gives an extra mana for the turn it is played.

If there is a situation where a deck is empty, each time that player draws a card from it, instead it will receive a point of damage for the first time, the second time it will receive two and so on.

2.3.3. Possible actions per turn

A Hearthstone^{TM} turn lasts a maximum of 75 seconds, and the player can do whatever number of actions he wants within the time of the turn, provided they do not break any of the other rules mentioned before, such as playing a card if they do not have mana enough or attacking with a minion if it has already attack or was played that turn.

- End turn task
- Hero power

  Each player depending on its class has a different hero power, this is an action that can only be done once per turn and always costs 2 mana.
- Play a card

  There are several types of cards:

  - Minion: it has attack, health, a cost, and may have also a text that produces an effect, when played goes into the board and from the turn after is played it can attack.
  - Spell: it produces an instant effect upon playing its mana cost, it doesn't remain on the board as minions do.
  - Weapons: it is equipped to your hero when the mana cost is paid and only one can be active at a time, it has an attack value and durability values that stand for the number of attacks left before it gets destroyed.

- Minion attack

  Minions, a turn after being played can attack the enemy hero or the enemy minions. If an enemy minion has *taunt* ability it has to be killed first to be able to attack other minions or the enemy hero. Minions by default can only attack once per turn.

2.4. Hearthstone^{TM} AI Competition

It is a Hearthstone^{TM} competition focused on AI [17] that has been celebrated in 2018 and will be celebrated again in July 2019, this time it will be part of the *IEEE Conference on Games* [18].

Its main purpose is to make a tournament of AI agents that will play against the others using 3 concrete decks known and other 3 decks that are unknown to the participants, in order to make an agent that knows how to play with any given deck.

There are several rules as not cheating looking the opponents hand or having a maximum time to make a decision for each turn.

There is a list in its official web page [19] of the agents resulting from 2018, so to test the behaviour of the approach followed, those agents will be used to test its quality.

Chapter 3

Methodology, design and solution implementation

This chapter aims to explain the environment in which the project has been executed, the behaviour the crafted agent has and the parameters it receives to change its behaviour.

3.1. Environment and technologies

Right after the idea of creating an AI for Hearthstone$^{\text{TM}}$, a deep research began on how to connect the agent with the real game and which programming language would suit the project necessities.

This research ends up with these results:

3.1.1. Simulators

At the end, the only choice was to use a simulator between the following options as the game's code was private and could not be accessed.

- **Metastone**

  It is written in Java, and its main focus is for analyzing card values, deck building tools and performance evaluation. It also allows users to create new custom cards and offers a fair simulator of the behaviours of Hearthstone^{TM} .

  It was an option to be taken as it is reliable as a simulator of the game, and especially because there are several AI that has been tested in this simulator before. [20]

- **Brimstone**

  It is a simulator written in C#, which focuses on being as fast and simple as possible for developers to build and test AI, card balance, new mechanics etc.

  It would definitely be the option to choose if it were not because it is under heavy development at the moment(June,2019) [21]

- **Sabberstone**

  This simulator places all the focus into AI for Hearthstone^{TM} developers, to be easy to understand, offers great communication with the game states and supports calculating all the available options for a player and manage the tasks that are available for the agent. It is written in C# .Net Core.

  This was the option taken as its features are a great support to develop and experiment with the created agent. It is also under development, but at a much later stage of the process. [22]

3.1.2. Visual Studio

Visual Studio 2017 Community has been used as the Integrated Development Environment as well as C# [23] as the programming language for the project. It is important to have .Net core 1.1 installed when installing Visual Studio for a correct functioning of Sabberstone.

The project its divided in several packages, where SabberstoneCoreAI has all the important files that may be modified by the developer. Other packages like SabberstoneCore mainly, contribute to the proper functioning of the Hearthstone^{TM} s mechanics and its simulation.

3.2. Implementation

In this section it will be explained the behaviour of the created agent. The only requirements to communicate with Sabberstone is to extend a class called *Abstract Agent* and implement its methods with the desired behaviour.

3.2.1. Monte Carlo Tree Search adaptation to Hearthstone^{TM}

When applying the Monte Carlo Tree Search technique to Hearthstone^{TM} there are some aspects to consider.

At first glance, a possible approach would be to perform a MCTS execution for the whole turn and afterwards return the sequence of actions resulting from it. But, as Hearthstone^{TM} 's cards do not have a deterministic behaviour when played, an increase in its performance will be that, after one MCTS execution an action is made. Then, the turn has not necessarily ended. Therefore, another MCTS could be launched until an end turn task is performed and the turn ends.

Following this approach every time an action is going to be made we know the exact state of the game resulting from previous actions. So, the actions are more accurate to the current game state. This, was the followed approach. [24]

Another adaptation that has to be done because of Hearthstone^{TM} restrictions happens while simulating, MCTS is defined to reach the end of the game where whether a player win or lose is known so, it can backpropagate the results.

On the contrary, Hearthstone^{TM} does not allow us to simulate random moves from the opponent, because although we would not look directly at the opponent's hand, when playing his cards, its behaviour can be inferred in such a way that at the end it will be cheating. To avoid that problem an estimator function of the state has been developed to guess on what percentage a state is a lose or win for a player.

3.2.2. Nodes structure

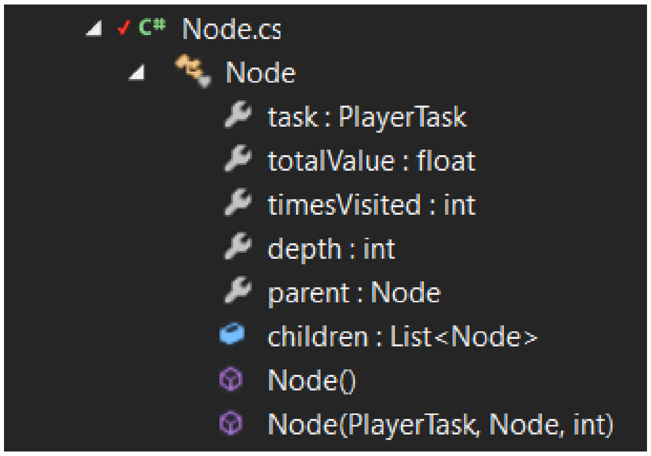

```
Node.cs
  Node
    task : PlayerTask
    totalValue : float
    timesVisited : int
    depth : int
    parent : Node
    children : List<Node>
    Node()
    Node(PlayerTask, Node, int)
```

Figure 3.1: Node structure class view

MCTS algorithm requires a tree made of nodes which have the following attributes as it can be seen in Figure3.1.

- **Task**

  It holds an action represented by the node.

- **Total value**

  It represents the number of guessed wins the node would get if its action is made.

- **Times visited**

  It represents the number of times that node has been visited.

- **Depth**

  It stores at what depth of the tree the node is. It is information required for the *tree maximum depth* parameter to work.

- **Parent**

  It stores the node right above the current node. Or null if it is the root node.

- **Children**

  It stores a list of the nodes right under the current node.

3.2.3. Main loop

If there is only one possible action, it is directly selected without running the Monte Carlo Tree Search algorithm.

Firstly, the tree gets initialized by having an empty root with all the possible actions as children, the information of what actions are allowed, is carried out by the simulator.

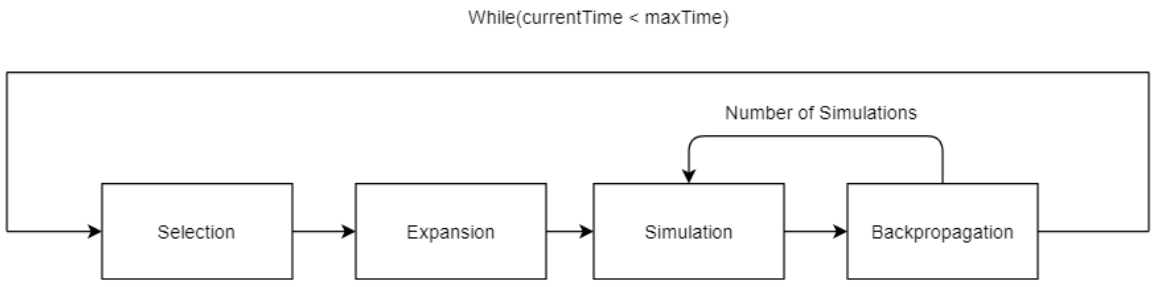


Figure 3.2: Main loop diagram

As mentioned in subsection 2.2.5. the MCTS algorithm does not have an end per se. As a consequence, the different stages of the MCTS will iterate over and over until a maximum time its reached, as shown in the Figure 3.2. For each iteration of the main loop, the stages simulation and backpropagation will be repeated depending on the *number of simulations* parameter.

Finally, a decision of which action is selected is made based on the *final decision making policy* parameter. Summarizing it will select the task of one of the root's children nodes.

3.2.4. Selection

This method is in charge of iterating the tree selecting the most interesting paths until it reaches a leaf.

For each child of the root node, a score is calculated to evaluate how interesting each node is, for this purpose two *tree policies* has been developed, the node with the maximum score is used and the task inside of it is simulated to have a copy of what could have happened after doing the task of the node.

But a decision has not been made yet, only if the selected node is a leaf in the tree we go to the next stage. If not, it recurs with the selected node as the root of the next recursion.

3.2.5. Expansion

This method is in charge of expanding the tree with available actions after selection.

In case the node coming from selection, has an end turn task, where we could not expand because it will no longer be our turn if we do so, or the node has been visited zero times or the *tree maximum depth* is reached, then expansion would return the current node, given from selection, to start simulating from it.

If none of those cases occurs then the tree will be expanded with the available options from the node given by selection. The first node created will be chosen, as no information about them is known it will not matter which one is chosen, a simulation of the new node's task will be done, so the virtual state of the game for future simulations is maintained and the simulation stage will start from this new node.

3.2.6. Simulation

Its main purpose is to simulate from the node given until an end turn task is found, meaning the end of my turn.

While the game do not reach the end or an end turn task is found, it will keep simulating updating in each of them the virtual state of the game, the selection of which node will be the next to simulate is done by a *simulation policy* explained in the parameters below. When the loop ends by any of the two reasons the *estimator function* will be called and its value returned.

3.2.7. Backpropagation

It is in charge of backpropagating the results, from the node the simulation started all the way up to the root of the tree. Updating all the nodes in the path, adding one to the *number of visits* of all the nodes traversed and adding the value of the estimation, given from simulation, to the *total value* of the nodes traversed.

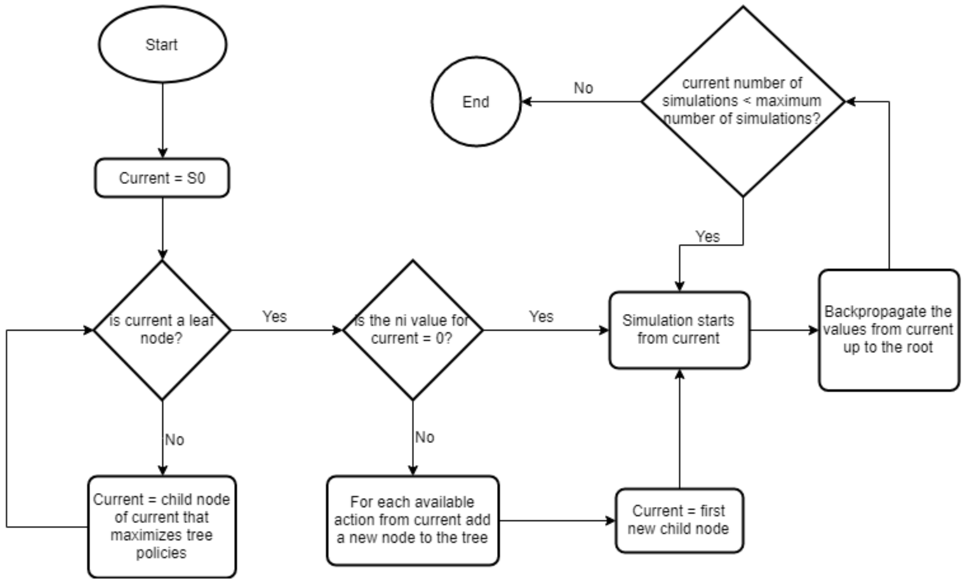

Start
Current = S0
Is current a leaf node?
Yes
No
Current = child node of current that maximizes tree policies
Is the ni value for current = 0?
Yes
No
For each available action from current add a new node to the tree
Current = first new child node
Simulation starts from current
Backpropagate the values from current up to the root
current number of simulations < maximum number of simulations?
Yes
No
End

Figure 3.3: An iteration of the algorithm

To summarize, a more detailed flow diagram as shown above in Figure 3.3 give an idea of an iteration of the algorithm.

3.3. Parameters

A way of making variations to the MCTS could have been done in several ways, but the followed approach was to create a set of parameters that could influence the behaviour of the MCTS, so different variations could be easily created and it could be analysed which parameters were the most influential to the agent. After doing so the implementation sub-goal mentioned in subsection 1.2.1 will be fulfilled.

The followings parameters are the ones used to create variations. Most of them have already been mentioned above.

We will use this names as an easy way to refer to them:

- let total victories of a node be vi
- let times visited of a node be ni
- let the number of iterations of the main loop be N
- let the explore constant be C
- let the state of the tree be Si
- let the heuristic function be H

3.3.1. Parameters affecting the Main Loop

- **Max time**

  It is the maximum time the algorithms is allowed to be running, as mentioned before, it could stay running forever and also be stopped at any given time. As much time is given it should get better results because more simulations could be done, therefore the search tree will be more explored.

  The range of values considered was: [1.0, 1000.0] ms for each action the player do.

- **Number of simulations**

  It is the number of times a simulation is done from a node. Plain MCTS would only do one simulation from a node. But, by doing several simulations there is a lot of time saving that could be use for more simulations, instead of traversing the tree in order to do just one simulation at a time as plain MCTS does.

  That time saving comes with a cost, as more simulations are done from a single path, if time runs out to quickly, there will be only a few branches explored, so it is not likely to get a good result.

  The range of values considered was: [1.0, 10.0]

- **Final decision making**

  It is in charge of deciding which node of the root's children should be selected after the maximum time its reached. This are the approaches considered.

  - **Maximum victories**

    The action taken would be the one with the maximum number of total victories.

    $$Action(Si) = \max(vi)$$

  - **Maximum visited**

    The action taken would be the one with the maximum number of times visited.

    $$Action(Si) = \max(ni)$$

  - **Maximum victories and visited**

    The action taken would be the one with the maximum number of times visited plus the number of total victories.

    $$Action(Si) = \max(ni + vi)$$

- **Maximum victories over visited**

  The action taken would be the one with the maximum the number of total victories over the number of times visited.

$$Action(Si) = \max(\frac{vi}{ni})$$

- **Maximum UCB**

  The action taken would be the one with the maximum Upper Confidence Bound(UCB).

$$Action(Si) = \max(\frac{vi}{ni} + C * \sqrt{\frac{\ln N}{ni}})$$

3.3.2. Parameters affecting Selection

- **Tree Selection Policy**

  When selecting, there should be a policy to decide which paths should be explored so, two approaches has been developed:

  - **UCB1**

    The Upper Confidence Bound(UCB) manages to generate a balance between exploitation and exploration since the nodes that have never been explored obtain a boost in its score so, there are no unexplored branches and at the same time the most promising ones also obtain score from their number of victories, but not as much as it starts to eclipse the other branches.

$$Score(Si) = \frac{vi}{ni} + C * \sqrt{\frac{\ln N}{ni}}$$

- **UCB1 Heuristic**

  This approach add some specific domain knowledge to the UCB approach, this knowledge is added with an heuristic that acts as an evaluation function. Such heuristic owner is Pablo García Sánchez, co-director of this project, who had previously made an agent for the mentioned in Section 2.4., and which agent used the heuristic, I am going to use.

  The final score will be as follows:

$$Score(Si) = \frac{vi}{ni} + C * \sqrt{\frac{\ln N}{ni}} + HeuristicImportance * \frac{H}{ni}$$

  The mentioned heuristic used the following parameters, that were optimized by Pablo García Sánchez using evolutionary algorithms:

  [hero health reduced, Hero attack reduced, minion health reduced, minion attack reduced, minion killed, minion appeared, secret removed, mana reduced, minion health, minion attack, minion has charge, minion has deathrattle, minion has divine shield, minion has inspire, minion has life steal, minion has steal, minion has taunt, minion has windfury, minion rarity, minion mana cost, minion poisonous]

- **Importance between Heuristic and UCB1**

  This parameter balances the importance given to the specific domain knowledge of the game over the UCB1 scoring system. If it is too high it will almost transform the MCTS in an algorithm similar to a greedy approach, but even worst because it would be disturbed by the other stages of the MCTS. On the other hand if it's too small it will transform UCB1 Heuristic approach to be almost identical to UCB1 approach.

  The range of values considered was: [0.01, 5.0]

- **UCB1 Explore constant**

  It balances how much importance should the exploration factor have over the exploitation. As much higher this value becomes it will tend to explore all the paths possible but will not focus enough time into exploiting, the opposite will happen if it is too small.

  The range of values considered was: [1.0, 3.0]

3.3.3. Parameters affecting Expansion

- **Tree maximum depth**

  It determines how big the tree is allowed to be made. By changing this value it changes how many far nodes are created from the root node that is the actual node that has to made the decision at the end. So, if it is too big it may happen that there is information gathered that is so far away from the actual state of the game that may disturb the current decision values. In contrary, if it is too small the only information that is going to be affecting to the decision are the closest nodes to the root.

  The range of values considered was: [1, 15]

3.3.4. Parameters affecting Simulation

- **Tree Simulation Policy**

  When simulating there are a few possible approaches to select which is the node that will follow the one that is currently been simulated.

  The approaches considered are the following:

  - **Random policy**

    It will select a random node between the possible children from the node that is currently been simulated. It is the approach the plain MCTS uses and it has the advantage that is really fast.

- **Greedy policy**

  It uses the score resulting from the score method of the agent created by Pablo García Sánchez to select which of the nodes to simulate next seems more appealing to the MCTS agent.

$$Node(Si) = \max(H_i)$$

- **Children considered while simulating**

  This parameter operates within the Greedy policy and it chooses which percentage of the possible nodes should be taken into account as the scoring method of the greedy might be too much time consuming. If it is too small it will lose too much information as only a few of the children would be taken into consideration, but if it is too large it will consider every possibility but it may do less iterations of the whole algorithm.

  Once a percentage is selected it will get that percentage of the children available at random rounding up.

  The range of values considered was: [0.0, 1.0]

- **Estimation function**

  As mentioned in Subsection 3.2.1. an estimation function is required in order to evaluate how won or lost a game is, as Hearthstone^{TM} rules does not allow us to do anything in the opponent's turn.

  Firstly, a way of scoring a given state of the game has to be developed and the following approaches were considered:

  The parameters shown in upper case in the following formulas are inputs values that has not been optimized, but handcrafted.

– **Linear estimation**

It will consider all the values as if they have a linear progression.

Score(Si) = heroHealth * HEALTH_IMPORTANCE +
heroArmor * HEALTH_IMPORTANCE +
(weaponAttack * WEAPON_ATTACK_IMPORTANCE +
weaponDurability * WEAPON_DURABILITY_IMPORTANCE) +
statsOnBoard * BOARD_STATS_IMPORTANCE +
handSize * HAND_SIZE_IMPORTANCE +
deckRemainingCards * DECK_REMAINING_IMPORTANCE +
maximumMana * MANA_IMPORTANCE +
secretsNumber * SECRET_IMPORTANCE -
overloadMana * OVERLOAD_IMPORTANCE

– **Gradual estimation**

It will consider all the values as linear but health, armor, hand size and cards remaining in the deck.

Score(Si) = $\sqrt{heroHealth}$ * HEALTH_IMPORTANCE +
$\sqrt{heroArmor}$ * HEALTH_IMPORTANCE +
(weaponAttack * WEAPON_ATTACK_IMPORTANCE +
weaponDurability * WEAPON_DURABILITY_IMPORTANCE) +
statsOnBoard * BOARD_STATS_IMPORTANCE +
$\sqrt{handSize}$ * HAND_SIZE_IMPORTANCE +
$\sqrt{deckRemainingCards}$ * DECK_REMAINING_IMPORTANCE +
maximumMana * MANA_IMPORTANCE +
secretsNumber * SECRET_IMPORTANCE -
overloadMana * OVERLOAD_IMPORTANCE

– **Value estimation**

It will consider, instead of the stats of the cards and minions, the cost of them. Assuming Hearthstone^{TM} is balanced, the cost of a card should give a fair average of the value of that card.

Score(Si) = heroHealth * HEALTH_IMPORTANCE +
heroArmor * HEALTH_IMPORTANCE +
weaponCost * WEAPON_COST_IMPORTANCE +
minionsOnBoardCost * MINION_COST_IMPORTANCE +
secretCost * SECRET_COST_IMPORTANCE +
handCardCost* CARD_COST_IMPORTANCE +
deckRemainingCards * DECK_REMAINING_IMPORTANCE +
maximumMana * MANA_IMPORTANCE -
overloadMana * OVERLOAD_IMPORTANCE

Once the scores of each player has been collected, the following formula created by myself to try to normalize the differences between the scores of the players, will give us how won or lost the game is.

Meaning 1, as the result of the estimation, the MCTS agent is absolutely winning with the state given, provided the game ended at that time, 0 the MCTS agent is definitely losing the game with the state given and 0.5 the MCTS agent and the opponent have equal possibilities of winning.

Let $Score1$ and $Score2$ be the scores resulted from one of the approaches shown above for each player.

$$WinEstimation(Si) = \frac{Score1 - Score2}{\max(Score1, Score2) * 2} + 0.5$$

Chapter 4

Experimentation and results

In this chapter, it will be explained how the results from the experimentation were gathered, which were the actual results and an analysis of them.

4.1. Experimentation environment

All the experimentation was developed on my personal computer and several scenarios were created to test the performance of the MCTS agent variations.

4.1.1. Machine characteristics

The machine used to carry out this tests was a laptop-Q80D2RF7 with Windows 10 as its operative system, an Intel(R) Core(TM) i7-6700HQ 2.6 GHz processor, 8GB of RAM memory.

4.1.2. Decks employed

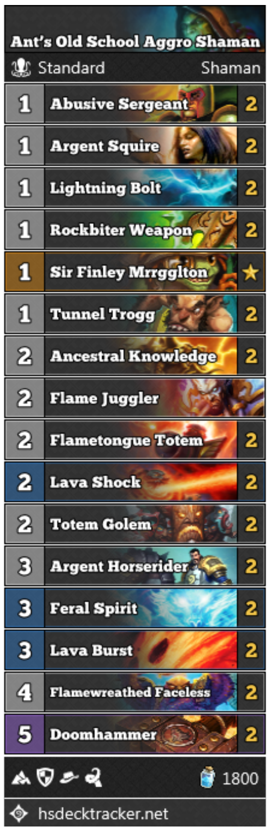
For this test three different decks were selected according to the best decks in Whisper of the Old Gods expansion [25]:

Meaning *aggro*, *mid-range* and *control* the pace at which each deck is meant to be played:

- Aggro, really fast it should capitalize on killing the opponent quickly
- Mid-range, capitalizing on having a strong enough early plays, but having some cards with high mana costs.
- Control, it capitalize on stalling the game in the early plays, so after a high maximum mana is reached they have higher costs cards than their opponents.

The decks used were:

Aggro shaman



Ant's Old School Aggro Shaman

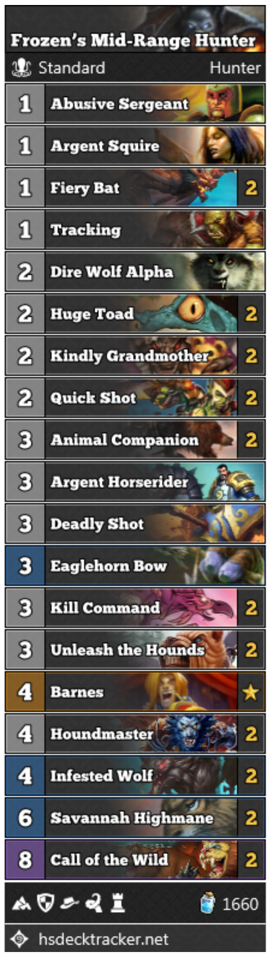
Standard — Shaman

1	Abusive Sergeant	2
1	Argent Squire	2
1	Lightning Bolt	2
1	Rockbiter Weapon	2
1	Sir Finley Mrrgglton	★
1	Tunnel Trogg	2
2	Ancestral Knowledge	2
2	Flame Juggler	
2	Flametongue Totem	2
2	Lava Shock	2
2	Totem Golem	2
3	Argent Horserider	2
3	Feral Spirit	2
3	Lava Burst	2
4	Flamewreathed Faceless	2
5	Doomhammer	2

1800

hsdecktracker.net

Mid-range hunter



Frozen's Mid-Range Hunter

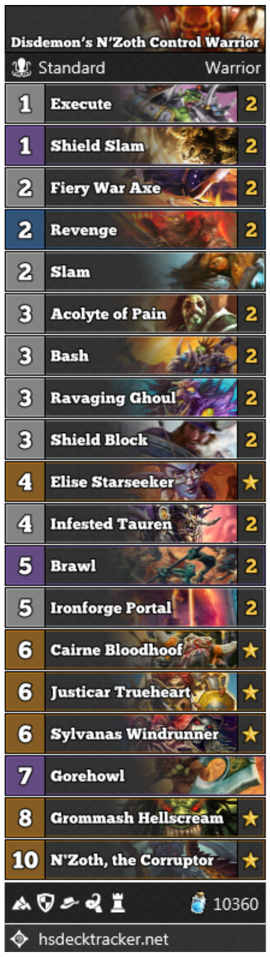
Standard — Hunter

1	Abusive Sergeant	
1	Argent Squire	
1	Fiery Bat	2
1	Tracking	
2	Dire Wolf Alpha	
2	Huge Toad	2
2	Kindly Grandmother	2
2	Quick Shot	2
3	Animal Companion	2
3	Argent Horserider	
3	Deadly Shot	
3	Eaglehorn Bow	
3	Kill Command	2
3	Unleash the Hounds	2
4	Barnes	★
4	Houndmaster	2
4	Infested Wolf	2
6	Savannah Highmane	2
8	Call of the Wild	2

1660

hsdecktracker.net

Control warrior



Disdemon's N'Zoth Control Warrior

Standard — Warrior

1	Execute	2
1	Shield Slam	2
2	Fiery War Axe	2
2	Revenge	2
2	Slam	
3	Acolyte of Pain	2
3	Bash	2
3	Ravaging Ghoul	2
3	Shield Block	2
4	Elise Starseeker	★
4	Infested Tauren	2
5	Brawl	2
5	Ironforge Portal	2
6	Cairne Bloodhoof	★
6	Justicar Trueheart	★
6	Sylvanas Windrunner	★
7	Gorehowl	
8	Grommash Hellscream	★
10	N'Zoth, the Corruptor	★

10360

hsdecktracker.net

Figure 4.1: Decks used for the tests

4.1.3. Agents employed

The different agents used for the tests were:

- **Greedy**

  This was the agent developed by Pablo García Sánchez, co-director of this project. It uses a greedy approach having an evaluation function for all the possible actions and choosing the one who has the biggest score. It finished second in the Hearthstone^{TM} AI competition of 2018.

- **Tyche**

  Kai Bornemann developed this agent and finished third in the Hearthstone^{TM} AI competition of 2018. It follows a MCTS approach too, with some variants.

The following five agents are variants with different parameters of the MCTS developed in this project. Some common values for the parameters of this five agents, chosen by my personal experience are:

– UCB1 Explore constant = 2

– Number of simulations = 1

– The parameters used for the Heuristic of Pablo García Sánchez used in the Selection tree policy

– Parameters for Linear estimation were always this:
  Weapon Attack = 0.7 , weapon durability = 0.4 , health importance = 0.4 , board stats importance = 0.9 , hand size importance = 0.4, deck remaining importance = 0.01 , mana importance = 0.02 , secret importance = 0.4 , overload importance = 0.3

– Parameters for Gradual estimation were always this:
  The same values as linear but the followings: health importance = 4.5 , hand size importance = 2.5 , deck remaining importance = 0.08

- **MCTS1**

  Final decision making = Maximum victories over visited

  Importance between Heuristic and UCB1 = 10

  Selection tree policy = UCB1

  Tree maximum depth = 1

  Tree simulation policy = Greedy policy

  Estimation function = Linear estimation

- **MCTS2**

  Final decision making = Maximum victories over visited

  Importance between Heuristic and UCB1 = 0.1

  Selection tree policy = UCB1

  Tree maximum depth = 1

  Tree simulation policy = Greedy policy

  Estimation function = Gradual estimation

- **MCTS3**

  Final decision making = Maximum victories over visited

  Importance between Heuristic and UCB1 = 1

  Selection tree policy = UCB1

  Tree maximum depth = 3

  Tree simulation policy = Random policy

  Estimation function = Linear estimation

- **MCTS4**

  Final decision making = Maximum victories

  Importance between Heuristic and UCB1 = 0.1

  Selection tree policy = UCB1 Heuristic

  Tree maximum depth = 1

  Tree simulation policy = Greedy policy

  Estimation function = Linear estimation

- **MCTS5**

  Final decision making = Maximum victories over visited

  Importance between Heuristic and UCB1 = 0.35

  Selection tree policy = UCB1 Heuristic

  Tree maximum depth = 10

  Tree simulation policy = Random policy

  Estimation function = Linear estimation

4.1.4. Description of the tests

A class called *Tournament* was created to perform all the competitions depending on the number of decks and agents involved. After the program finish it will create a .csv file with all the information related to the games played.

The tests were divided in four stages having the following conditions. And each of the stages being repeated a number of times to reduce the randomness of the games played, being for this case 13 repetitions.

Stages one, two and three followed the same pattern of all the players playing two games against each other, one playing first and one playing second and using the same deck for the whole stage for all the agents, one deck for each stage.

In stage four all agents play against all other agents twice one playing first and one playing second. But this time with all available decks, unless both have the same deck, because this should have been tested in stage one, two or three depending of the deck.

4.2. Stage one: Aggro shaman

After 546 games have been played using only Aggro shaman deck, the following results were gathered:

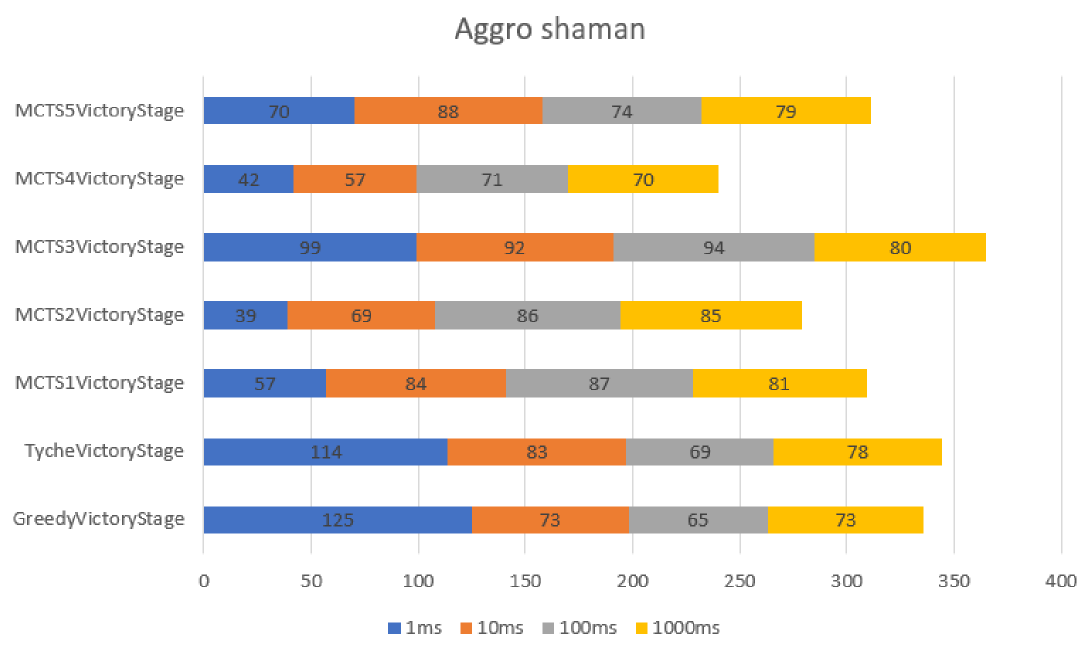


Figure 4.2: Stage one: Aggro shaman results

As it can be seen in Figure 4.2 in average it seems that MCTS3 is the best agent who plays aggro shaman, this deck is fast and capitalize on doing damage to the enemy hero. For the longest time 1000ms the winner is MCTS1.

4.3. Stage two: Mid-range hunter

After 546 games have been played using only Mid-range hunter deck, the following results were gathered:

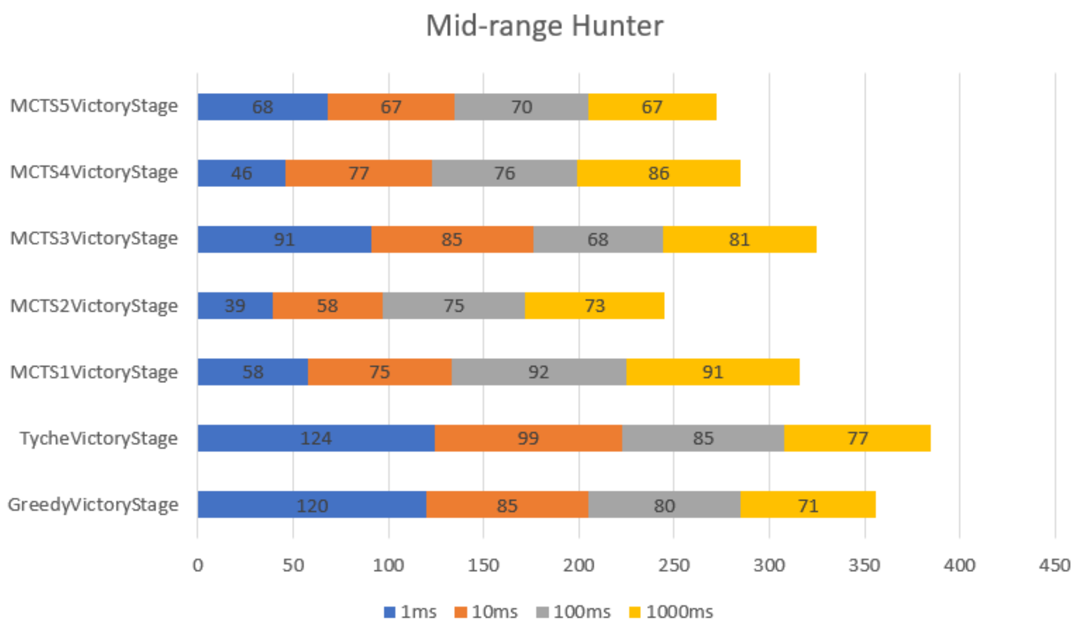


Figure 4.3: Stage two: Mid-range hunter results

Figure 4.3 shows that Tyche is the average winner of this stage, the required behaviour to win with this deck would be a balance a between using their cards in the best moments but without being to slow. For the longest time the winner is MCTS1 again.

4.4. Stage three: Control warrior

After 546 games have been played using only Control warrior deck, the following results were gathered:

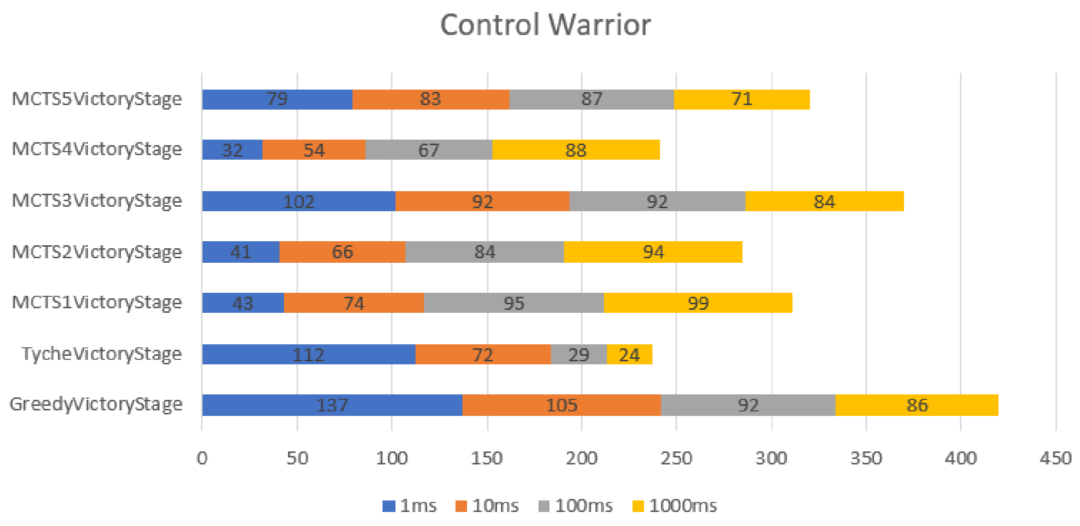


Figure 4.4: Stage three: Control warrior results

Figure 4.4 is a match were the agents should optimized their resources available up to the maximum point. Greedy is the winner in average and another time MCTS1 wins for the longest time.

4.5. Stage four: Free-for-all

After 3276 games have been played using all the decks between all agents, the following results were gathered:

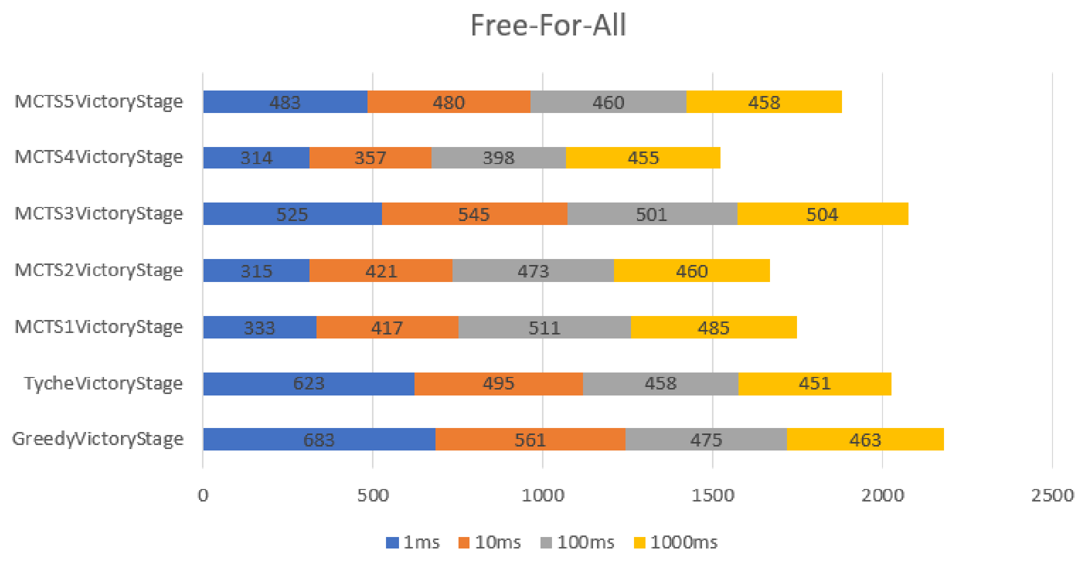


Figure 4.5: Stage four: Free-for-all results

Up to this point it can be seen that although Greedy and Tyche approaches have varied they results a lot between the different decks the MCTS variants have remain with similar results independent of the deck being used because they adapt to the deck requirements through the simulation process.

As it shows the Figure 4.5 the average winner is Greedy because it not depends on the time used. However the longest time winner is MCTS3 in this case closely followed by MCTS1. My personal impression of why now MCTS3 have more wins in free for all when MCTS1 has been winning all the other stages is that MCTS3 uses a bigger Tree Maximum Depth allowing him to gather more complex information resulting from the battles between different decks.

4.6. Final results

This results were obtained by adding up all the stage's scores, giving the amount of 4914 games been played for each time.

4.6.1. Decision time: 1ms

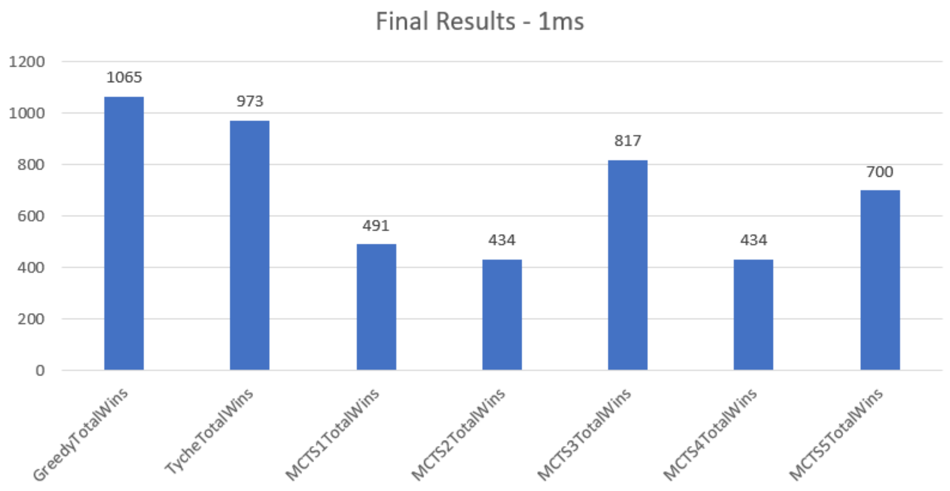


Figure 4.6: Final results for 1ms

As it can be seen in Figure 4.6 the Monte Carlo Tree Search approaches seems not to work well with too less time, probably because they do not have time enough to explore the sufficient amount of tree paths. It seems that the two approaches with a higher Tree maximum depth, MCTS3 and MCTS5 achieve better results as a consecuence of gathering information faster by storing it in several nodes of the tree instead of just one, as MCTS1, MCTS2 and MCTS4 does.

4.6.2. Decision time: 10ms

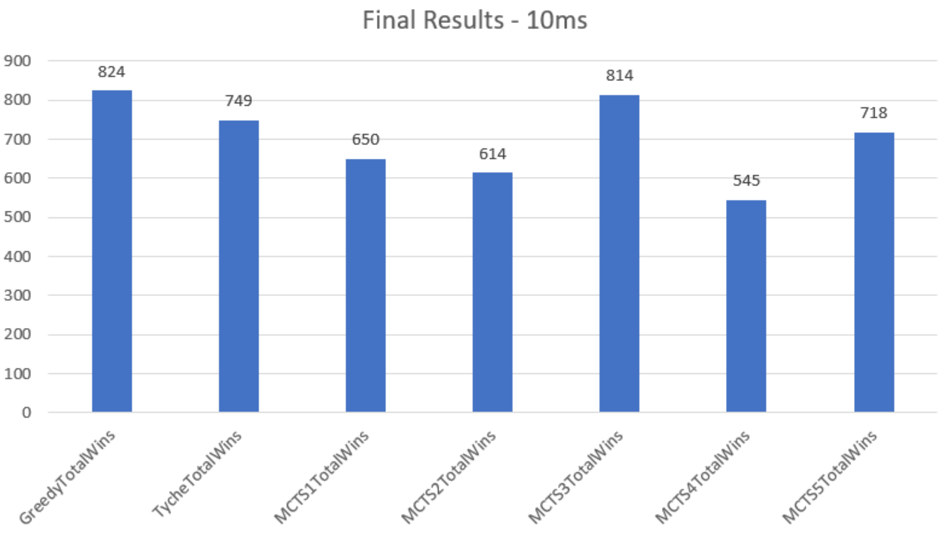


Figure 4.7: Final results for 10ms

At first glance, Figure 4.7 shows that after having a bit more time the MCTS approaches starts to explore most of the tree so, they are getting closer results to the reference agents, Greedy and Tyche. MCTS3 had the highest increase in comparison with 1ms and this happens because it is using a random policy for simulating which explores more than exploit the best path, in contraposition with the greedy one that focuses more on the best paths so also requires more time to get to its full potential exploring the most hidden paths.

4.6.3. Decision time: 100ms

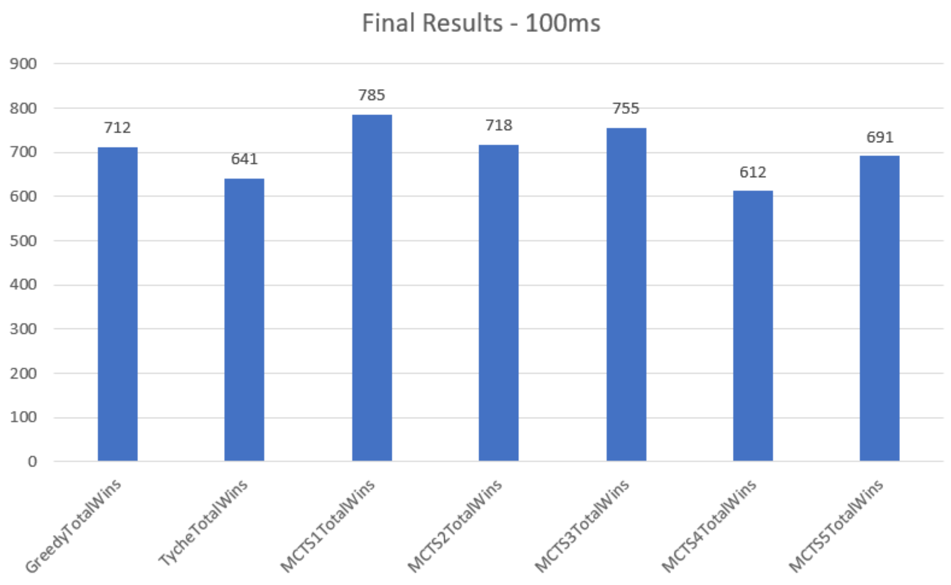


Figure 4.8: Final results for 100ms

As it can be seen in Figure 4.8 MCTS3 seemed to have reached the point with 10ms to explore most of the tree and that is why it has not keep increasing its performance at the same ratio. This, is also the first time that some of the variants wins to the reference agents and not only one but three of them MCTS1, MCTS2 and MCTS3, being MCTS1 and MCTS2 identical with the exception of the estimation function. The reasons why these variations have reach so good results is because by using a greedy policy for simulating and having time enough those hidden paths mentioned before can now be reached.

4.6.4. Decision time: 1000ms

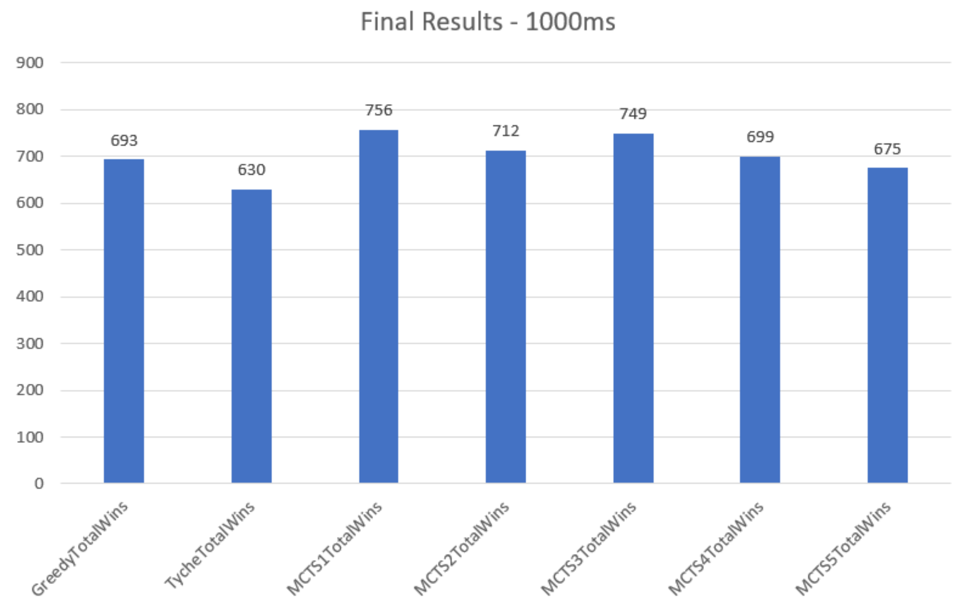


Figure 4.9: Final results for 1000ms

Finally the last and most important test, Figure 4.9 shows clearly how MCTS seems to operate better with the sufficient amount of time. This time all the variants but MCTS5 has won to the reference agents. MCTS4 due to the use of the heuristic in the selection policy and in the simulation policy has been the one who has needed more time than anyone else to reach to a point were the most of the tree its explored.

4.7. Best average agent and conclusions

This section should not be considered as a reference of the performance of the agents, rather it is a representation of how fast the MCTS explores their trees or a reminder that the Greedy approach does not use time to work so they results in all times is the same one, with the variance of the other agents improving or worsen.

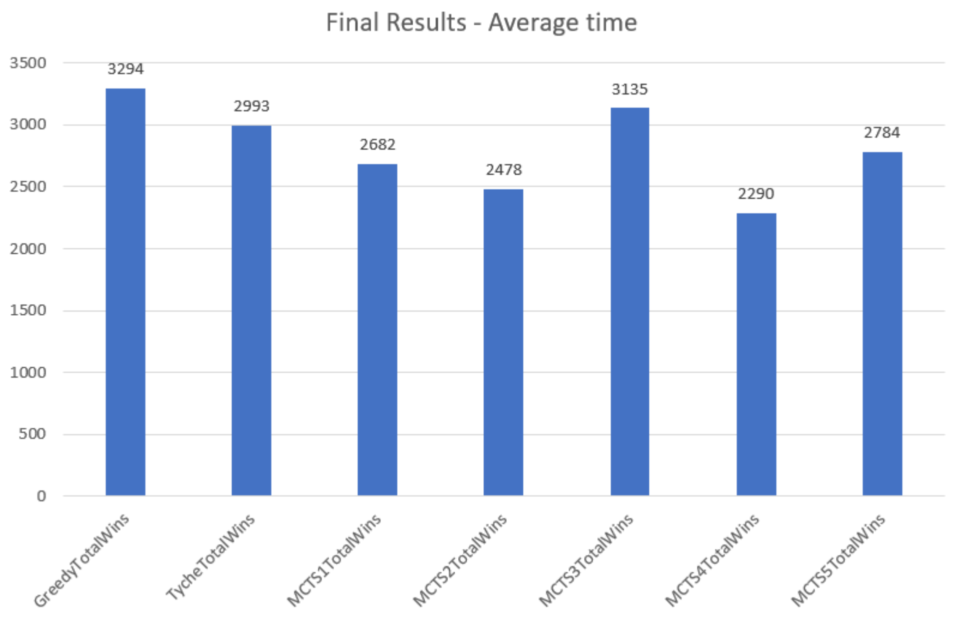


Figure 4.10: Final results average

It can be seen in Figure 4.10 that MCTS1, MCTS2 and MCTS4 until a time of 100ms or 1000ms did not quite explore most of the paths, therefore their average values are not that good.

4.7.1. Conclusion of results

- The winner of this tournament is MCTS1 because it has achieved the most number of games won at the maximum amount of time. It have to be considered that a Hearthstone^{TM} turns longs for 75 seconds, and with an average of 4-5 actions per turn in my own experience playing, means that having 1000ms per action to "think" it is not an issue.

- It is also worth mentioning what parameters of MCTS1 makes him the winner. One of the key parameter was the Tree maximum depth = 1, because it seems that if the algorithm focuses only on the next action to do and gather information with simulations from those points that information seem to be better for deciding, also having a greedy policy for simulating did give him some advantage over others similar like MCTS3. However, I do not think that parameter did too much of a difference for its performance in comparison with the other possibility, Random policy. I can not assure that the linear estimation function is better than the gradual one (MCTS1 vs MCTS2) as the parameters of the gradual one are really complicated to guess them right for a human.
- Although being MCTS1 the winner of this tournament, after all this tests it can be concluded all of the variants, even though working so differently on the inside, reach a similar results when they get to explore most of their trees but, require different amounts of time.

Chapter 5

Conclusions

Through the development of this project, has been explained what is Monte Carlo Tree Search technique and a parameterized approach to how could an agent play Hearthstone^{TM} .

The obtained results are satisfactory, winning to two well rounded agents that achieved a great results in the Hearthstone^{TM} AI Copetition means the resulted agent achieve an efficient performance, and will probably keep getting better if their parameters get optimized.

Therefore, the Monte Carlo Tree Search technique seems to be pretty good for Hearthstone^{TM} environment due to the inherited randomness of the game. It appears to find the best paths of the tree quicker than other tree search techniques does, because it relies mainly on simulations to discover it, instead of an evaluation function.

5.1. Improvements and future development

Some other functions as heuristics to the tree selection policy could be added to see if they are better than the one used.

Other improvement could be allowing the simulation stage, to go through the opponent's turn using only the information that is already known to the MCTS agent, as minions on board or the number of cards in hand, so the agent will not only take into account its own turn but also some of the opponents plays and the MCTS agent next turn.

Without doubts the most important improvement and future development would be to optimize, by an optimizer, the parameters involved in the current state of the agent, as most of this parameters have been handcrafted by personal experience.

In addition, it is consider to try to search for other agents that uses different techniques as evolutionary algorithms or similar to test MCTS performance against other Artificial intelligence techniques that may not have been tested yet in Hearthstone^{TM} environment.

Finally, I will try to submit the resultant agent to the 2019 Hearthstone AI Competition and a further article might be written when the parameters are optimized.

5.2. Personal learning

I have learnt a whole lot of things by developing this project.

- Starting with, using Latex technology to write this report, that I have never used before and turned out to be pretty handy and powerful.
- Secondly by practising my writing English skills, by doing this report.
- I have learnt how to develop a whole project from the beginning: researching, reading documents, implementing and testing as well as organizing myself to fulfilled the milestones I proposed.
- I have learnt the Monte Carlo Tree Search technique and how interesting could it be to think about possible improvements for a given algorithm.
- Finally, I learnt how tedious is to do experimentation of Artificial Intelligence, since it takes a long time for the tests to finish, and the number of games in this case should not be smaller or there may be too much variance involved in the results.

5.3. Technical problems encountered

- Firstly, I found quite late in the process of development, I was cheating because I was simulating the future plays of the opponent without looking directly to his hand, but in the end using its cards so the information of which cards it is going to play could be inferred by the MCTS. Afterwards this issue was solved by stopping the simulations when my turn ends.

- A parameter to reuse the tree for all the decisions involved in the same turn was considered, meaning for the first execution of the MCTS in the same turn a tree is generated and after an action is selected, the turn does not necessarily have finished, so the next MCTS to make a decision in the same turn could reuse parts of the previous tree to require less time to run. This approach, was discarded because playing a Hearthstone^{TM} 's card do not produce a deterministic effect therefore some information would be lost.

- Another problem encountered was to test all the parameters by hand to be able to show some potentials results of the agent in this report, even though it has not reached its most potential yet, it had reached a good enough performance.

- Understand how an agent could be connected with Sabberstone simulator, how information about the state of the game could be gathered and the adaptations that MCTS would require to be implemented in Hearthstone^{TM} 's environment were some problems that required a lot of time to be fully understood.

Chapter 5

Conclusiones

A lo largo del desarrollo del proyecto, se ha explicado la técnica del 'Monte Carlo Tree Search', así como se ha creado un agente parametrizado capaz de jugar a Hearthstone^{TM}.

Los resultados obtenidos son muy satisfactorios, el hecho de ganar a dos agentes que obtuvieron grandes resultados en la 'Hearthstone^{TM} AI Copetition' significa que el agente realizado en el proyecto tiene un gran rendimiento y probablemente mejore una vez sus parámetros sean optimizados.

También, la técnica del 'Monte Carlo Tree Search' parece funcionar muy bien para 'Hearthstone^{TM}' debido a la inherente aleatoriedad del juego. Parece que es capaz de encontrar los mejores caminos del árbol más rapido que otras técnicas de busqueda en árboles. Ya que, se basa en realizar muchas simulaciones para descubrirlos, en vez de utilizar una función de evaluación.

5.1. Mejoras y desarrollo futuro

Añadiría otras heurísticas a las políticas de selección en el árbol para ver si estas pudiesen ser mejores que la usada actualmente.

Otras mejoras podrían ser permitir que en la fase de simulación, permitiera al algoritmo pasar por el turno del oponente utilizando unicamente la información que pudiese ser conocida en ese instante, como el número de esbirros en mesa, o el número de cartas en mano del oponente, de esta manera el agente no solo tendra en cuenta su propio turno, sino que también tendra en cuenta algunas de las posibles jugadas que pudiese hacer su oponente en el siguiente turno.

Sin duda la mejora más sustancial para el proyecto sería optimizar los valores que se le pasan como parámetros al agente, a través de un optimizador, ya que los parámetros actuales se han decidido a mano según mi experiencia personal.

Además, se considerara buscar otros agentes que usen distintas técnicas, como algoritmos evolutivos o similares para probar el rendimiento del 'Monte Carlo Tree Search' contra otros tipos de inteligencias artificiales, que no hayan sido probadas anteriormente en 'Hearthstone^{TM}'.

Finalmente, intentare participar en la competición '2019 Hearthstone AI Competition' con el mejor agente resultante del proyecto y puede que escriba un árticulo futuro cuando los parámetros ya hayan sido optimizados.

5.2. Aprendizaje personal

He aprendido un monton de cosas a lo largo del desarrollo del projecto.

- Empezando por aprender como utilizar la herramienta de Latex para escribir esta memoria. Nunca la había utilizado y a resultado ser de gran utilidad.
- También, considero que he afianzado mi nivel de inglés al escribir esta memoria.
- He aprendido a desarrollar un proyecto completo, de inicio a fin pasando por todas sus fases desde la investigacion hasta la implementación y prieba de resultados. Así como he aprendido a organizarme para cumplir las metas propuestas en su tiempo.
- He aprendido como funciona la técnica del 'Monte Carlo Tree Search' y como de interesante puede ser buscar posibles formas de mejorar el algoritmo.

- Finalmente, he aprendido como de tedioso puede resultar la experimentación en la inteligencia artificial ya que toma grandes cantidades de tiempo para poder probar el rendimiento de los agentes al tenerse que usar un número de partidas lo suficientemente elevado como para que se palien los efectos de la aleatoriedad.

5.3. Problemas técnicos encontrados

- Primero, he encontrado en una fase tardía del desarrollo del proyecto que el agente realizado estaba haciendo trampas ya que en la fase de simulacion se simulaban acciones usando las cartas del rival. Aunque no se mirara a sus cartas directamente, al utilizarse después de muchas iteraciones de simulaciones el algoritmo estaba infiriendo cuales eran o sus efectos, información que no debería usar. Más tarde este problema fue resuelto haciendo que las simulaciones se terminasen cuando se acabase el turno del jugador.

- Se consideró un parámetro destinado a reutilizar el árbol para todas las decisiones que se realizaran dentro del mismo turno, consiguiendo así que no se tuviera que crear desde el inicio el árbol para cada decision, solo se crearía desde el principio para la primera acción del turno. Esta opción fue finalmente descartada, pues se pensó que debido a que las cartas en 'Hearthstone$^{\text{TM}}$' cuando se juegan no producen un efecto determinista por lo que reutilizar el árbol implicaría que no se estaría teniendo en cuenta información util que si que era accesible.

- Otro problema que se encontró fue ir probando a mano distintos valores de los parámetros para poder enseñar en esta memoria resultados buenos, incluso si nunca se llegaría a obtener los mejores resultados posibles por parte del algoritmo ya que se estaban optimizando a mano sus valores.

- Entender como el agente se podría conectar con el simulador Sabberstone, fue otro de los problemas. Saber como la informacion de cada estado podía obtenerse del simulador y también que adaptaciones del 'Monte Carlo Tree Search' se requerirían para adaptarlo a 'Hearthstone^{TM}'.

Bibliography

[1] "The video games' industry is bigger than hollywood." https://lpesports.com/e-sports-news/the-video-games-industry-is-bigger-than-hollywood. (Accessed in June 2019).

[2] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*, vol. 2. Springer, 2018.

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

[4] A. J. Champandard, "Monte-carlo tree search in total war: Rome ii's campaign ai," *AIGameDev. com: http://aigamedev. com/open/coverage/mcts-rome-ii*, 2014 (Accessed in June 2019).

[5] "Hearthstone main page." https://playhearthstone.com/es-es/. (Accessed in June 2019).

[6] M. Sion *et al.*, "On general minimax theorems.," *Pacific Journal of mathematics*, vol. 8, no. 1, pp. 171–176, 1958.

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[8] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, pp. 21–21, 1996.

[9] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[10] F.-Y. Wang, J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, and L. Yang, "Where does alphago go: From church-turing thesis to alphago thesis and beyond," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 2, pp. 113–120, 2016.

[11] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[12] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo, "Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone," *Knowledge-Based Systems*, vol. 153, pp. 133–146, 2018.

[13] A. Janusz, T. Tajmajer, and M. Świechowski, "Helping ai to play hearthstone: Aaia'17 data mining challenge," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 121–125, IEEE, 2017.

[14] "Collectible card games information website." `https://en.wikipedia.org/wiki/Collectible_card_game`. (Accessed in June 2019).

[15] "Magic: The gathering main page." `https://magic.wizards.com/en`. (Accessed in June 2019).

[16] "Hearthstone advanced rulebook." `https://hearthstone.gamepedia.com/Advanced_rulebook`. (Accessed in June 2019).

[17] A. Dockhorn and S. Mostaghim, "Hearthstone ai competition." `https://arxiv.org/pdf/1906.04238.pdf`, 2018. (Accessed in June 2019).

[18] "Ieee cog 2019 competitions." `http://ieee-cog.org/competitions_conference/`. (Accessed in June 2019).

[19] A. Dockhorn and S. Mostaghim, "Hearthstone ai competition's bots." https://dockhorn.antares.uberspace.de/wordpress/bot-downloads/. (Accessed in June 2019).

[20] demilich1, "Metastone hearthstone simulator." https://github.com/demilich1/metastone. (Accessed in June 2019).

[21] darkfriend77, "Brimstone hearthstone simulator." https://github.com/darkfriend77/Brimstone. (Accessed in June 2019).

[22] hearthsim, "Sabberstone hearthstone simulator." https://hearthsim.info/sabberstone/. (Accessed in June 2019).

[23] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# language specification.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] A. Santos, P. A. Santos, and F. S. Melo, "Monte carlo tree search experiments in hearthstone," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 272–279, IEEE, 2017.

[25] "Vs data reaper report 20." https://www.vicioussyndicate.com/vs-data-reaper-report-20/. (Accessed in June 2019).

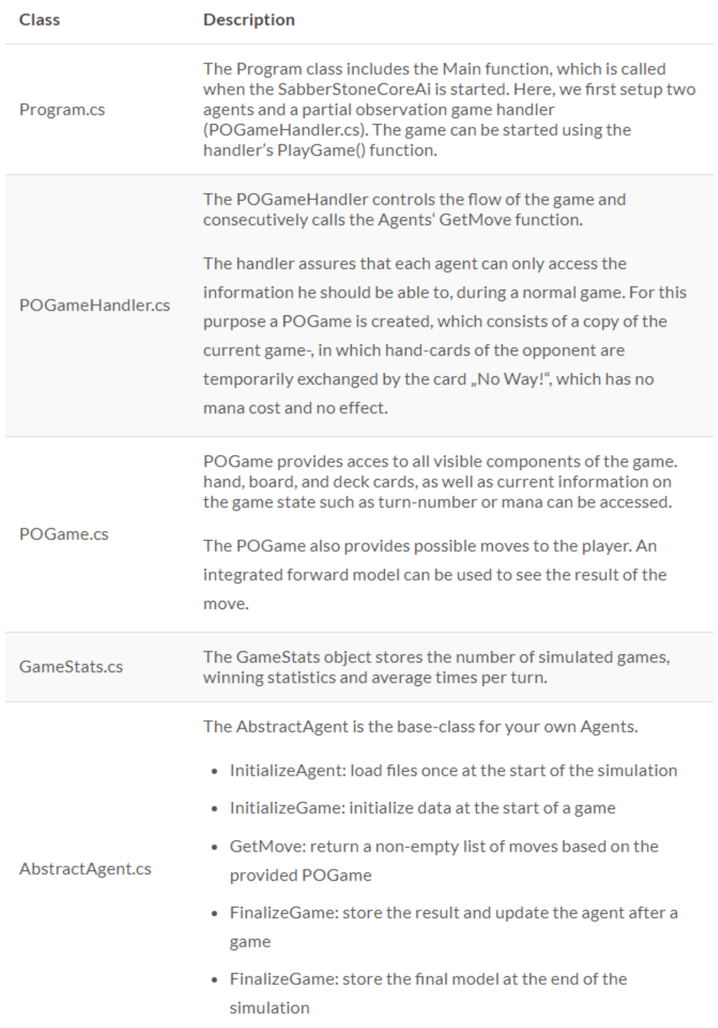
[26] A. Dockhorn and S. Mostaghim, "Hearthstone ai competition's classes." https://dockhorn.antares.uberspace.de/wordpress/tutorial/. (Accessed in June 2019).

Appendix A

Project set up and structure of the project

First, it will be started by downloading the contents of this Github repository: https://github.com/ADockhorn/HearthstoneAICompetition

Second, an IDE is recommended to manage the different packages that were downloaded. The one used for this project, although others should also work, was Visual studio 2017. It is important to install when asked .Net Core 1.1. once all the installations finished. A file called SabberStone.sln should be opened, it contains all the packages being SabberStoneCoreAI the only one you should modify. [26]



Class	Description
Program.cs	The Program class includes the Main function, which is called when the SabberStoneCoreAi is started. Here, we first setup two agents and a partial observation game handler (POGameHandler.cs). The game can be started using the handler's PlayGame() function.
POGameHandler.cs	The POGameHandler controls the flow of the game and consecutively calls the Agents' GetMove function. The handler assures that each agent can only access the information he should be able to, during a normal game. For this purpose a POGame is created, which consists of a copy of the current game-, in which hand-cards of the opponent are temporarily exchanged by the card „No Way!", which has no mana cost and no effect.
POGame.cs	POGame provides acces to all visible components of the game. hand, board, and deck cards, as well as current information on the game state such as turn-number or mana can be accessed. The POGame also provides possible moves to the player. An integrated forward model can be used to see the result of the move.
GameStats.cs	The GameStats object stores the number of simulated games, winning statistics and average times per turn.
AbstractAgent.cs	The AbstractAgent is the base-class for your own Agents. • InitializeAgent: load files once at the start of the simulation • InitializeGame: initialize data at the start of a game • GetMove: return a non-empty list of moves based on the provided POGame • FinalizeGame: store the result and update the agent after a game • FinalizeGame: store the final model at the end of the simulation

Figure A.1: Important classes

In Figure A.1 it can be seen which classes need to be understood for using SabberStone. To create an agent all the information needed is to go to the Agent folder and create a class that extends AbstractAgent and implements its methods.

For the concrete files of this project, they are organized as follows:
There is a folder called *AlvaroMCTS*, that should be inside of folder Agents mentioned above, with the main class of the algorithm called *AlvaroAgent* from this class all the other class objects are created or called for assigning parameters to the MCTS.

- Node.cs

  Contains all the information that a node is required to store.

- DeckManager.cs

  It is in charge of building the decks so, there are some "getters" methods returning each of the decks.

- SelectAction.cs, TreePolicies.cs, SimulationPolicies.cs and Estimator.cs

  Are factory classes for the different ways of implementing those parameters and also contains its implementations.

TycheAgent should be downloaded from:
https://dockhorn.antares.uberspace.de/wordpress/bot-downloads/

ParametricGreedyAgent will be with the code of the project but belongs to Pablo García Sánchez.

The following classes should be in folder src:

A class Tournament.cs and AgentTournament.cs was also created as mentioned in subsection 4.1.4. Description of the tests for managing all the experimentation.

Finally, the class Program.cs that contains the function main and is the first executed, contains the set up of the decks and agents with its parameters and a call to start the tournament.

Appendix B

CSV format document

After executing the tournament, when all games have been played a .csv is created with the information of the games in the following format.

First, it will store all the games played between each two agents as follows:

Headers: Stage, League, Agent1, Deck1, Agent2, Deck2, Victory1, Turns
An example could be: 2, 5, MCTS1, MidrangeHunter, Greedy, MidrangeHunter, 1, 20
"League" stands for the number of a certain stage's repetition.

Second, it will store the summatory of the results of each league as follows:

Headers:Stage, League, Agent1Name Victories, Agent2Name Victories, TotalGamesLeague, Agent1Name Turns, Agent2Name Turns ... depending on the number of agents

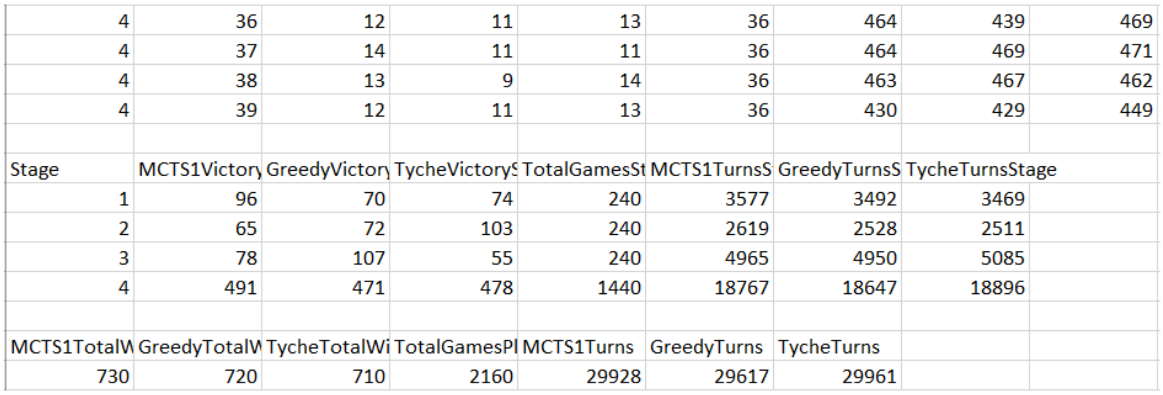
Then, it will store the summatory of the results of each stage as follows:

Header: Stage, Agent1Name Victories, Agent2Name Victories, TotalGamesStage, Agent1Name Turns, Agent2Name Turns ... depending on the number of agents
And below each agent it will appear the number of victories accomplished by the whole stage

Finally it will appear the final results of the test:

Header: Agent1 Victories, Agent2 Victories... ,totalGames, Agent1 Turns, Agent2 Turns... And below the total wins adding all stages.

This, will be an example of one test:



4	36	12	11	13	36	464	439	469
4	37	14	11	11	36	464	469	471
4	38	13	9	14	36	463	467	462
4	39	12	11	13	36	430	429	449
Stage	MCTS1Victory	GreedyVictory	TycheVictoryS	TotalGamesSt	MCTS1TurnsS	GreedyTurnsS	TycheTurnsStage	
1	96	70	74	240	3577	3492	3469	
2	65	72	103	240	2619	2528	2511	
3	78	107	55	240	4965	4950	5085	
4	491	471	478	1440	18767	18647	18896	
MCTS1TotalW	GreedyTotalW	TycheTotalWi	TotalGamesPl	MCTS1Turns	GreedyTurns	TycheTurns		
730	720	710	2160	29928	29617	29961		

Figure B.1: Example of a resulting .csv document

Where the first 4 lines are a chunk of all the information related to the results from the summatory of each league in each stage, the next table contains the information related to the summatory of victories in each stage and finally the last table contains the final number of victories per agent.

The information of each of the games played should be above all the previous information.