Monte Carlo Tree Search Experiments in Hearthstone

André Santos*, Pedro A. Santos†, Francisco S. Melo‡
Instituto Superior Técnico/INESC-ID
Universidade de Lisboa,
Lisbon, Portugal
Email: *andre.l.santos@tecnico.ulisboa.pt, †pedro.santos@tecnico.ulisboa.pt, ‡fmelo@inesc-id.pt

***Abstract*—In this paper, we introduce a Monte-Carlo tree search (MCTS) approach for the game "Hearthstone: Heroes of Warcraft". 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 MCTS with a properly constructed heuristic, it is possible to introduce significant gains in performance. We illustrate through extensive empirical validation the superior performance of our approach against vanilla MCTS and the current state-of-the art AI for Hearthstone.**

***Index Terms*—Monte Carlo Tree Search, Artificial intelligence for games, Hearthstone.**

I. Introduction

Collectible card games (CCGs) are one of the most popular forms of contemporary game play. Since the inception of "Magic, the Gathering"—back in the 90s—several such games emerged as popular forms of entertainment (both physical and electronic) and even training and education [1], [2].

The often intricate gameplay allied with aspects of hidden information and chance also make collectible card games appealing testbeds for artificial intelligence (AI) research. For example, several works have explored both machine learning [3] and planning approaches in CCG games such as "Magic, the Gathering" [4], [5].

In this paper, we develop an AI system for the card game "Hearthstone: Heroes of Warcraft", the most popular online CCG game, with 50 million players as of April 2016 [6]. Hearthstone is a turn-based CCG 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, and a complex gameplay, which often requires sophisticated strategy. Several works in the literature have tackled different aspects of the game, such as deck building [7], card generation [3] and general game-play [8]. The former work, in particular, uses a supervised learning approach to predict a (discrete) value for each action and each state. The result of such classifier is then used for action selection.

We propose the use of Monte Carlo Tree Search (MCTS), as it is becoming a *de facto* standard in game AI and is particularly suited to address the chance elements in Hearthstone [4], [5], [9], [10]. In particular, we propose 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 rollout 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 paper 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.

The rest of the paper is organized as follows. Section II provides a general overview of the game and the simulator used. Section III describes the details of the proposed approach. Section IV describes the methodology used to fine-tune our approach, while Section V discusses the results achieved with our approach. Section VI concludes.

II. Hearthstone

We start by presenting an overview of the game and the main challenges it poses in terms of AI.

A. Hearthstone: Heroes of Warcraft

"Hearthstone" is a turn-by-turn online CCG with matches played between two opponents. Before a match, each player can build a 30 card deck using one of nine heroes available. Each hero has a special power and base cards that the player can choose, together with "common" cards that can be chosen for any hero. Having selected her or his hero and deck, the player can then enter a match with an opponent. Each player starts with 30 life points and the main goal of the game is to reduce the life points of the opponent's hero to zero. To do so, players can summon minions or apply damage to the minions that the opponent currently holds on the battlefield, besides directly damaging the opponent's hero. 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. The match evolves as each player receives and uses mana crystals to play new 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, Minions and Weapons. 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. 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 players.

B. Hearthstone strategies

Because players select the deck of 30 cards with which they play, the deck governs the player's strategy, and it is not unusual to associate "standard decks" with common strategies in the game. Common decks/strategies include:

- *Aggro* (meaning "aggression"): It is a deck comprising 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*: It is a flexible deck, 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).
- *Contro*l: It is a strategy that prioritizes 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.

C. Metastone

Metastone [11] is an open-source simulator available for the Hearthstone community. The simulator includes all the main gameplay mechanisms. Additionally, it includes functionalities allowing the simulation of a large number of games between different heroes, decks and AI systems, providing summarized statistics after the matches. The simulator already includes some AI systems against which other artificial Hearthstone players can be tested. The systems included in the simulator are

- A random player that selects the actions at random and provides a naive baseline for comparison.
- A "no aggression" player, corresponding to a player that does not attack the opponent. The AI randomizes between playing cards or simply performing the "end-turn" action.
- A "greedy" player, corresponding to a myopic player whose actions are driven by a heuristic built on several game metrics and whose weights were tuned using an evolutionary approach (see Section III).
- The Game state Value (GSV), a recursive alpha-beta algorithm driven by the aforementioned heuristic. To the extent of our knowledge, this is the state-of-the-art, and several existing players report disappointing performances against it.

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

III. Proposed Approach

Our approach builds on the well known Monte Carlo tree search family of methods. Therefore, before discussing our proposed enhancements to MCTS, we provide an overview of this class of methods.

A. Monte Carlo tree search

Monte Carlo tree search 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 [12], [13]) and the randomness (as observed in games such as Hearthstone).

MCTS iteratively builds a search tree from the current state of the game. The 4 main steps in MCTS are (see Fig. 1):

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 to the available actions.
3) *Simulation:* From each of the nodes expanded in the previous stage, one or more simulations (rollouts) is run until a terminal state is reached. The simulations are obtained using a predefined *default 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. Backpropagation is the final step of an MCTS iteration.

Let us consider each of these stages 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 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.

The tree policy 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.

A commonly used tree policy relies on the so-called *upper confidence bound*, or UCB [14], and selects at each node v the successor node v^* such that

$$v^* = \arg\max_w \frac{Q(w)}{N(w)} + c\sqrt{\frac{2\ln N(v)}{N(w)}}, \qquad (1)$$

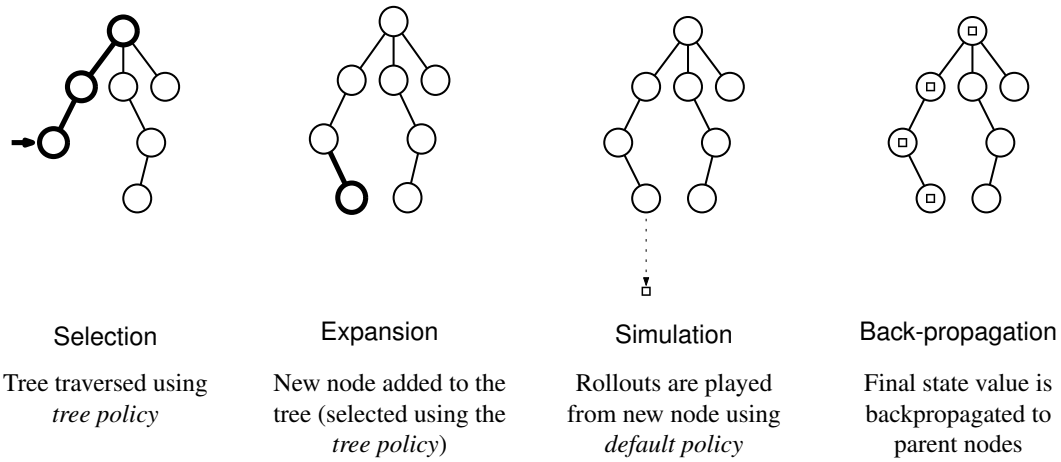


Fig. 1: Diagram representing the 4 steps of MCTS. In the first two steps, the tree is traversed using the *Tree Policy*, until a leaf node is reached and marked for expansion. The expanded node is selected again using the tree policy. The algorithm then simulates trajectories of the system, using some default policy, until a terminal state is reached. The value of that state is then backpropagated to its parents. Diagram adapted from [9].

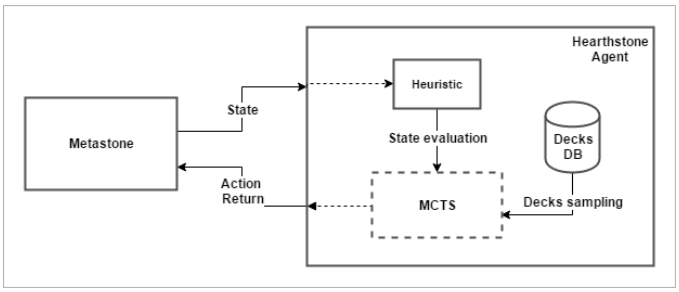


Fig. 2: High-level overview of the interaction between our MCTS agent and the Metastone game engine.

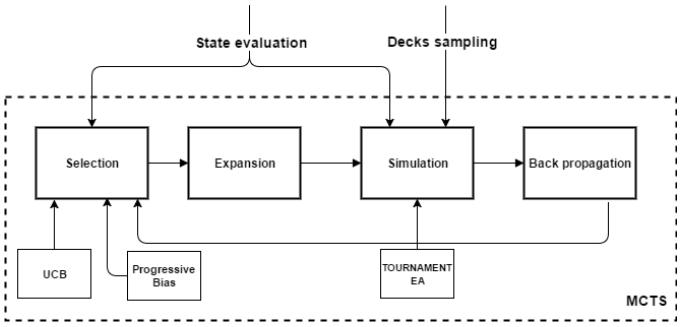


Fig. 3: Detail of the MCTS block from Fig. 2.

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

Using the UCB, MCTS is guaranteed to converge to the minimax tree as the number of simulations from each node grows to infinity [15]. 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.

◇

While the simplest instances of MCTS can run without any specific domain knowledge, it has been shown in practice that the performance of MCTS can be improved significantly by providing the algorithm with additional knowledge.

In our approach, we propose the integration of expert knowledge regarding the game of Hearthstone. In particular, we propose the use of (i) a heuristic that guides the selection and simulation steps of MCTS; and (ii) a deck database that allows the algorithm to reason about possible cards that the opponents may hold.

Our approach is summarized in Figs. 2 and 3. We discuss the details in the continuation.

B. Adding expert knowledge to the selection step

We use *progressive bias* [16] as a mean to integrate expert knowledge in the selection step of MCTS. Progressive bias takes the form of an additional term included in (1):

$$v^* = \arg\max_w \frac{Q(w)}{N(w)} + c\sqrt{\frac{2\ln N(v)}{N(w)}} + \frac{H(w)}{1+N(w)}, \qquad (2)$$

where H is a heuristic function representing the domain knowledge. In other words, we enrich the UCB selection rule with an extra term that accounts for the domain-specific expert knowledge.

C. 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 selection and simulation steps 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. For example, in [17], a statistical learning algorithm was used to predict the most probable future card between turns three and five of a match. The algorithm 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 from can be significantly narrowed down.

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.

The use of a deck database 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 thus able to (approximately) infer the opponent's deck type and most likely cards from its hero and the cards played in the early stages of the game.

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.

D. Sampling in simulation

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

To circumvent this difficulty, we adopt the *tournament selection approach* commonly used in evolutionary computation [18], [19]. In particular, at each step of the simulation, 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 default policy used for simulation. 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.

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

E. Tree and default policies heuristic

Both the tree and the default policies used in the selection and simulation stages of MCTS rely on a heuristic function that evaluates subsequent states and informs action selection. In order to assess the impact of such heuristic in the performance of the method, we considered 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

$$H(s) = \sum_{n=1}^{N} \alpha_n \phi_n(s), \tag{3}$$

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.

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. In particular, it uses the following features:

- *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.
- *Board mana advantage (BM)*: the difference between the sum of mana for the player's cards and opponent's ones.

Several of the features above were already used in the literature, although in a different setting (see, for example, [8]). The resulting heuristic becomes:

$$\begin{aligned} H(s) = {} & \alpha_{MA}\phi_{MA}(s) + \alpha_{TMA}\phi_{TMA}(s) \\ & + \alpha_{HA}\phi_{HA}(s) + \alpha_{TrA}\phi_{TrA}(s) + \alpha_{BM}\phi_{BM}(s). \end{aligned} \tag{4}$$

The value of the weights was optimized using genetic programming, by having a myopic greedy agent driven only by the heuristic play numerous games against the greedy Metastone player.[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.

2) Heuristic 2: As *second heuristic*, we use the one driving the greedy and GSV Metastone players. The heuristic also takes the form in (3), but includes a number of additional features that account, for example, for the number of life points that the hero still has.

F. Action return

Being an anytime algorithm, when the computational budget is exhausted, MCTS returns the best candidate action at the

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

root node, v_{root}. We compared four selection methods used in the literature [9]:

- *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.,

$$v_{\text{lcb}} = \underset{v \in C(v_{\text{root}})}{\arg\max}\ \frac{Q(w)}{N(w)} - c\sqrt{\frac{2\ln N(v)}{N(w)}}.$$

G. Search tree reuse

Traditionally, being an online planning algorithm, MCTS is restarted at every step t of the execution, bearing as root node the state s_t of the system. In other words, at each execution step 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 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 iteration (and the outcome of the corresponding simulations) are discarded between execution steps.

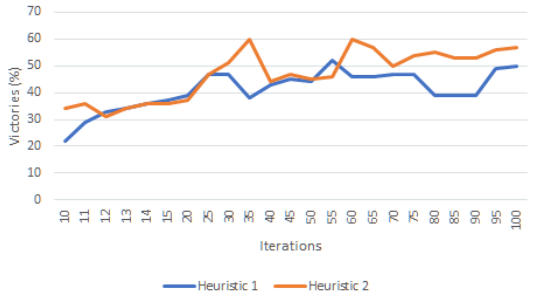
In order to maximize the use of information from one execution step to the next, we explore the possibility of reusing the tree from the previous iteration in growing the new tree. Such reuse is reminiscent of *search seeding*, wherein the values at each node are not started at 0.

IV. Parameter Selection Methodology

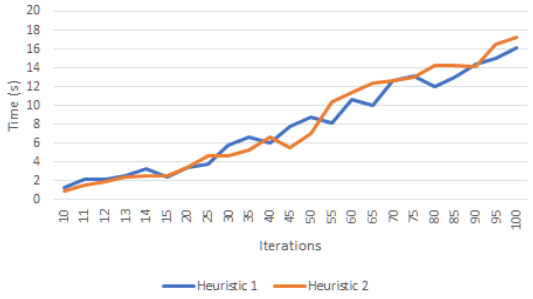
As seen in the previous section, our proposed MCTS approach to Hearthstone includes a number of adaptations whose impact should be tested. We thus conducted an extensive validation process aimed at establishing the impact, in our approach's performance, of the:

- number of iterations vs. the number of rollouts per iteration;
- value of the parameter k in the simulation;
- heuristic used in the tree and default policies;
- action selection criterion;
- search tree reuse.

In particular, we investigated the impact of each of the above components by varying one while keeping the other fixed (the exact parameters and full results are described in the appendix of the extended version of this paper). For each of the free parameters (such as k) we conducted a simple grid search across the space of possible values. For each configuration we measure both the performance against Metastone's *GSV AI* (see Section II) and the computational time.[3] Computational times were measured on a 2.6GHz Intel Core i7 processor with 16GB of RAM memory.



(a) Performance vs. number of MCTS iterations.



(b) Time vs. number of MCTS iterations.

Fig. 4: 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.

The base configuration for the validation process is:

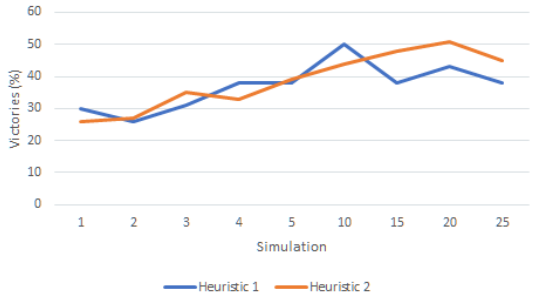
- **Player 0 (our approach):**
  - *Hero:* Warlock hero;
  - *Deck:* Tarei's warlock zoo deck.
- **Player 1 (Metastone's Game State Value):**
  - *Hero:* Warlock hero;
  - *Deck:* Tarei's warlock zoo deck.

A. Number of iterations

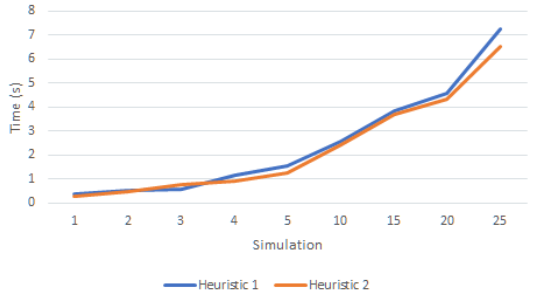
In a first test, we evaluated the impact of the number of iterations allowed to MCTS both in terms of time and game performance, using each of the two simulation heuristics. The results are summarized in Fig. 4, and correspond to averages over 250 independent runs. We report as performance the percentage 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

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

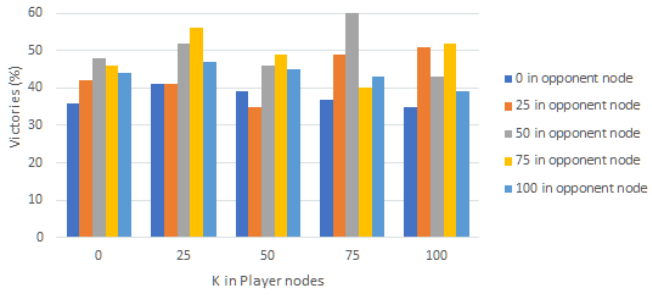


(a) Performance vs. number of rollouts during simulation.

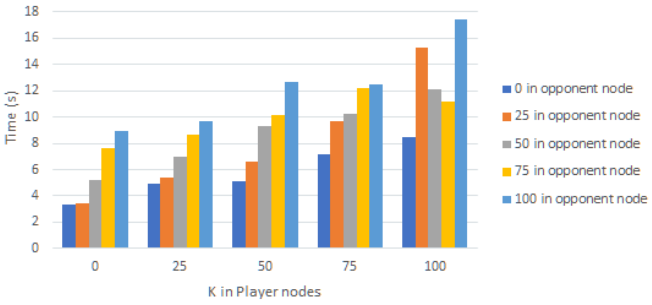


(b) Time vs. number of rollouts during simulation.

Fig. 5: Game and computational performance of our proposed approach as a function of the number of rollouts performed during the simulation stage. The results correspond to averages of 250 independent runs.



(a) Performance vs. k.



(b) Computation time vs. k.

Fig. 6: 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.

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.

Finally, we note in Fig. 4a 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.

B. Number of rollouts

A second parameter that influences the performance of our approach is the number of rollouts performed per iteration in the simulation stage of the game. The results are summarized in Fig. 5, 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 IV-A, 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.

C. 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 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. 6 for Heuristic 2 (results for Heuristic 1 are similar and can be found in the in the appendix of the extended version of this paper).

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 $k_1 = 50\%$.[4]

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

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

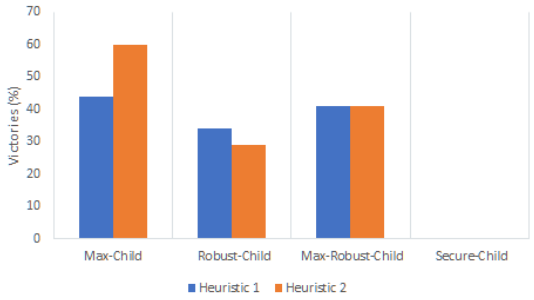


Fig. 7: Performance of our proposed approach with the different action selection criteria. The results correspond to averages of 250 independent runs.

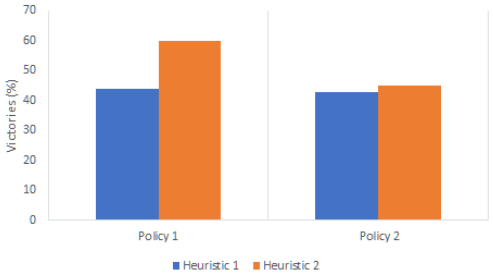


Fig. 8: Impact in performance of tree reuse. Policy 1 corresponds to the policy obtained when the search tree is maintained between execution steps. Conversely, Policy 2 is obtained by rebuilding the tree at each execution step.

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.

We conclude by noting, from Fig. 6b, 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.

D. Action return

We also compared the performance of the different action selection alternatives discussed in Section III. The results are summarized in Fig 7. The results 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.
- 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.

E. Search tree reuse

Finally, we investigated the impact of tree reuse in the performance of the algorithm. In particular, we compared a first policy obtained when the tree is reused between execution steps and a second policy obtained when the search tree is rebuilt at each execution step. The results are depicted in Fig. 8, and clearly show that tree reuse does, in fact, lead to improved performance.

V. Empirical Evaluation

In order to perform a comparative analysis between our approaches, we paired them against the different AIs existing in Metastone:

- Random Player;
- No Aggression Player;
- Greedy;
- GSV Player.

which represent 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:

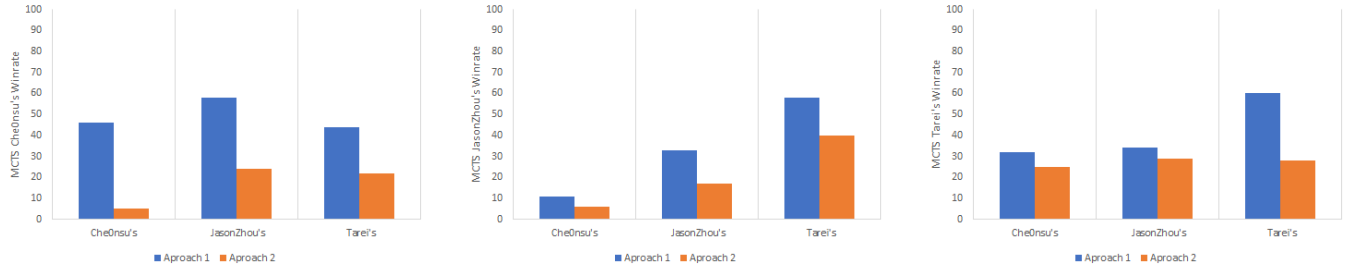
- *Tarei's Warlock Zoo*: A moderate aggro-based deck that aims to control the board while damaging the opponent.
- *JasonZhou's N'Zoth Control Warrior*: control-based deck, one of the most consistent in Hearthstone. It aims to exhaust the opponent's resources, dominating in late-game turns.
- *Che0nsu's 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. Both approaches used the deck database to handle hidden information and used the same parameters[5] throughout (the exact parameters and full results are described in the extended version of this paper). Each MCTS version played into a round-robin, being used all the possible combinations of decks between them.

Against the *Random* and the *No Aggression* players our approaches attained a win-rate near of 100% in every scenario. This is not surprising, since the complexity of the game prevents those players from meaningful play. The *Greedy* player was more even-matched, the "vanilla" MCTS version on average presenting a win-rate close to 40% while the full approach presented win-rates of 54% to 60%.

Finally, we tested the performance of both approaches against the actual-state-of-the-art of Metastone's AI, the *GSV*. The results are summarized in Fig. 9, where *Approach 1* is our approach and *Approach 2* is the "vanilla" MCTS player. In average, our adaptation to MCTS performed a win-rate close to 42%, while the MCTS "vanilla" version presented an win-rate close to 21%. Some decks are better matched than others, which explains the variation across the different

[5]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.



(a) Che0nsu's deck (b) JasonZhou's deck (c) Tarei's deck

Fig. 9: Performance of MCTS players against Metastone's Game State Value player with different decks.

decks. 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 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.

VI. Conclusions and Future Work

In this paper, we develop an MCTS-based approach for Hearthstone. Our approach boosts 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. Our results show that our approach is superior to "vanilla" MCTS and are able to attain competitive results with state of the art AI for Hearthstone.

Our results also open venues for future research. In our approach, the heuristic governing the tree construction uses a single set of weights that is deck-independent. We observe that an improvement in performance could be achieved by considering specialized weights in the deck used by the player.

Also, we would like to develop our own heuristic, with a dynamic approach for targeting a specific game strategy. Hopefully, this would allow to optimize MCTS for a more specific behavior, instead a specific deck. Also, we desire to develop a new method for selecting the candidate root action in MCTS. In our experiments, we only develop 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 the number of victories.

Acknowledgment

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and by the EC H2020 project RAGE (Realising an Applied Gaming Eco-System) http://www.rageproject.eu; grant agreement No 644187. The authors 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 work.

References

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

[2] 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.

[3] W. Ling, E. Grefenstette, K. Hermann, T. Kočiský, 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] 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.

[11] Metastone. Metastone Simulator.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

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