Proceedings of the 16th Conference on Computer Science and Intelligence Systems pp. 101–110
DOI: 10.15439/2021F3
ISSN 2300-5963 ACSIS, Vol. 25

A Practical Solution to Handling Randomness and Imperfect Information in Monte Carlo Tree Search

Maciej Świechowski
QED Software, Warsaw, Poland
Email: maciej.swiechowski@qed.pl

Tomasz Tajmajer
Institute of Informatics
University of Warsaw, Poland
and *QED Software*, Warsaw, Poland

***Abstract*—This paper provides practical guidelines for developing strong AI agents based on the Monte Carlo Tree Search algorithm in a game with imperfect information and/or randomness. These guidelines are backed up by series of experiments carried out in the very popular game - Hearthstone. Despite the focus on Hearthstone, the paper is written with reusability and universal applications in mind. For MCTS algorithm, we introduced a few novel ideas such as complete elimination of the so-called nature moves, separation of decision and simulation states as well as a multi-layered transposition table. These have helped to create a strong Hearthstone agent.**

I. Introduction

Games of various kinds and forms have been challenging human minds since the ancient times [1]. Many games encode abstract problems, solving which requires high intelligence and well-developed reasoning ability. It is natural that people have started using them for more than just pure entertainment. With the inception of modern computers and artificial intelligence (AI), games have been often employed as testing environments [2], [3], [4], [5]. As of now, artificial intelligence in games is still a hot and growing research topic.

One of the recent trends in AI in games revolves around creating universal game-playing agents. Such an approach is believed to be closer to the roots of the AI and it is perceived as a step towards general intelligence. The most active research projects in this area are General Game Playing (GGP) [6], [7], [8] and General Video Game Playing (GVGP) [9]. Both projects include annual competitions open for the strongest programs. Both in GGP and GVGP, MCTS has become the state-of-the-art method. Since 2007, all winners of the GGP competition have used this algorithm. A particular reason for its success in general domains is the fact that MCTS requires only the rules of a game. Although it can take advantage of domain knowledge, as shown in [10], [11], it is fully operational without it.

Because MCTS is a statistics-based algorithm, it has helped to tackle non-deterministic hidden-information games, which had been particularly difficult for other tree search algorithms [12], [13], [14]. Nevertheless, MCTS is also used in games without non-determinism or hidden-information such as Go [15], Hex [16], Othello[17] or Havannah [18].

This research was co-funded by the Smart Growth Operational Programme 2014-2020, financed by the European Regional Development Fund under a GameINN project POIR.01.02.00-00-0207/20, operated by The National Centre for Research and Development.

We describe a relatively complete approach to using MCTS in a game with hidden information and random effects [19], [20]. However, the specific game we have chosen for this study is Hearthstone: Heroes of Warcraft, developed by Blizzard Entertainment [21]. Hearthstone is an immensely popular game having around 70 million active players. It is a video card game that consists of series of duels. Players prepare 30-card decks of a chosen hero class. There are ten available hero classes. The goal is to use such a combination of minions, spells, weapons, secrets and alternative heroes to reduce the opponent's life total to zero. Minions can attack or be attacked. Spells usually deal damage, control the board (e.g., destroy minions), draw more cards or provide positive buffing/healing effects.

The paper is organized as follows. The next section is devoted to imperfect information, including Hearthstone-specific aspects of it (Section II-D), and the MCTS algorithm. Tree management using a multi-layered transposition table is shown in Section III. Section IV is devoted to the problem of dealing with randomness. All aspects presented in sections II to IV make up a complete MCTS-based agent. Based on the descriptions and pseudocode provided, the reader should be able to reproduce the agent and adapt it to a new game. The experiments and results for our MCTS-based agent in Hearthstone are presented in Section V. Finally, the last section is devoted to conclusions.

II. Imperfect Information

A. MCTS

This section is a short introduction to the Monte Carlo Tree Search algorithm (MCTS). We assume some familiarity of readers with the topic and recommend the survey [22] as a relatively exhaustive source of knowledge about MCTS.

The MCTS algorithm is the *state-of-the-art* method of performing the game tree search. The idea behind MCTS is to construct game tree iteratively, as depicted in Figure 1, by adding one node in each iteration. The algorithm uses simulations to gather statistical evidence about the quality of actions. The statistics include the average score, the number of visits to a node (state) and the number of times an action has been chosen in the iterations so far.

The aim of a selection policy is to maintain a proper balance between exploration (of not well-tested actions) and exploitation (of the best actions so far). The most common

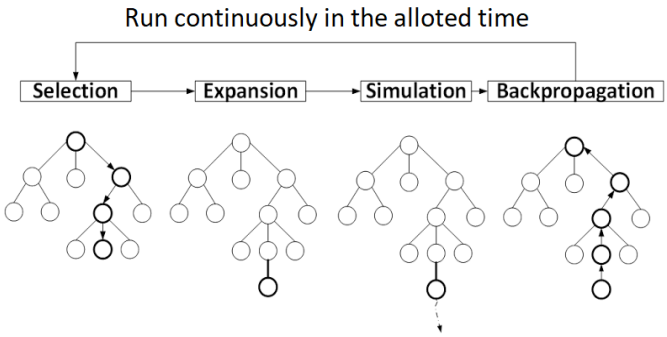

Fig. 1. The schema of the MCTS algorithm.

algorithm, which was also used for the experiments in this work, is called Upper Confidence Bounds applied for Trees (UCT) [23], [24].

B. The State-of-the-Art Approach

A game is of imperfect information if participating players cannot observe the complete state that affects the game. Even if some portion of the state is hidden only to certain players or only at certain moments, the game should still be considered as having imperfect information. Like randomness, imperfect information increases the combinatorial complexity of game tree search algorithms because the algorithms do not have access to the actual state (unless they cheat) and have to consider many potential states instead. There are two state-of-the-art methods of tackling hidden information with MCTS:

1) Perfect Information Monte Carlo Tree-Search [25] (PIMC): which performs determinization of hidden information and after that considers the game as a perfect-information one. Each determinization symbolizes a possible (parallel) world, in which a regular MCTS algorithm is executed. The standard PIMC algorithm performs many determinizations at the root level and combines statistics and decisions from them. The key problems with this approach are strategy fusion and nonlocality, both discussed in papers [25], [26]. The strategy fusion is manifested whenever an algorithm combines strategies determined for various worlds into a single optimal strategy. This can often lead to weak play, because the actual (unobservable) state is represented only by one of these worlds. Another effect of the strategy fusion with determinizations is manifested when an opponent is to make a partially observable move. After a determinization, each move is fully deterministic, so the PIMC algorithm can make different decisions based on a particular determinization of the opponent's partially observable move by the as discussed in work [26]. This is a problem of overfitting to specific determinizations. The nonlocality problem stems from the fact that determinizations might have different likelihoods of being accurate. In particular, some determinizations may be extremely unlikely, rendering their solutions irrelevant to the overall process.

2) Information Set Monte Carlo Tree Search [26] (ISMCTS): this algorithm introduces so-called information sets, which cluster states that are indistinguishable from a particular player's point of view. The ISMCTS algorithm greatly reduces the effects of strategy fusion and nonlocality, which are present in PIMC. However, ISMCTS requires a much more complex game model, which creates a huge implementation workload for developers. The authors of work [26] wrote that in ISMCTS, "*the player's choices of actions must be predicated on information sets, not on states*". In other words, the game simulator must allow for making both fully observable moves (for the sake of the regular game playing) and partially observable moves (for the sake of AI agents using ISMCTS). Working with Hearthstone, we have found out that this approach can be impractical for more complex games, especially video games with complex moves.

Another method, mentioned in studies [26], [27], is called Belief Distributions, which consists in modelling the decision process of players that have different access to information (typically, the mutual opponents). The history of observed actions forms an input for calculating the probability distributions of possible states. Such a method requires a good model, which is usually hand-crafted by experts. It has been applied in games such as Poker [28], in which one of the main aspects of the game is to guess what cards the opponent is holding.

C. Our Approach

Our solution combines determinizations and information sets. We introduce two distinct interfaces for representing the game state:

1) Game simulation state (GS-state): this interface allows performing all of the game's logic, such as determining legal moves, applying moves and updating states, checking if the game is in a terminal state, who won the game etc. It can only be used with complete information – either by some kind of game server / game-master that maintains the complete state (knows the correct one) or by a player after performing a determinization (guessing the state).

2) Information Set state (IS-state): this interface represents all information about the state of the game that is available to a particular player and is used by him or her to make decisions. The second property is very important - the information set may ignore, i.e., not contain, some information that is visible to the player if it is not that important from a decision process perspective. For example, let us assume that a health property of a player is a number from 0 to 100. The game AI designer may decide not to represent the health of a player as a numerical value, but rather a set of buckets $[0, 0], [1, 20], ..., [81, 100]$. Therefore, the maximum number of unique values of the health property is reduced from 101 to 6 in the IS-state. However, the GS-state has to operate on the maximum resolution to comply with the rules of the game properly. For the optimization of parameters chosen to be stored in information sets, machine learning algorithms can be used. Similarly, the IS-state may contain some redundant information from the GS-state point of view, e.g., whether a particular card has already been played in the game. The main

consequence is that the proposed method allows for **complete separation of the state used for decisions (and gathering statistics of actions) and the state that is required for simulations**.

We believe that such a distinction has universal applicability and makes it easy to apply the ISMCTS algorithm in various games, no matter how the game simulator is written. In particular, the AI component based on the MCTS algorithm can be added after the game logic engine is written because the AI component will not put any constraints on the engine. This approach allows creating a game simulator separately from the AI module (a good software engineering practice), without making any sacrifices required for the AI. This property has been invaluable during development of the simulator for Hearthstone, especially in terms of how actions are represented and applied to a state. The IS-state is a static snapshot of transformed game state data, so there is no concept of applying actions to IS-states. The IS-state can be viewed as a so-called plain data object. As it will be shown in Section III, devoted to the storage of knowledge, the only functionality apart from storing data IS-state provides is the equality comparison. The relatively exhaustive pseudocode of the proposed approach is shown in Algorithm 1. The procedures which are not explained in detail in Algorithm 1 are discussed below:

- **updateRoot** - this procedure changes the root node in the tree when the current state of the game is changed. The procedure is explained in detail in Section III-A.
- **determinize** - this procedure guesses the hidden information in the game based on naive sampling among possible realizations with a uniform probability.
- **propagate** - the standard back-propagation phase of the MCTS.
- **simulate** - the standard simulation phase of the MCTS. This procedure starts with a given state, simulates till the end and gets players' scores.
- **tt.findOrCreate** - the procedure that finds a tree node that corresponds to the information set. It is explained in detail in Section III-B.

Algorithm 1 The pseudocode of the proposed MCTS implementation.

```
1:  procedure ITERATE(gs_state)
2:      rootNode ← updateRoot(gs_state)
3:      node ← rootNode                          ▷ current node
4:      while elapsedTime < allotedTime do
5:          gs_movingState ← determinize(gs_state)
6:          GLOBALS::SELECTION ← RUNNING
7:          while GLOBALS::SELECTION is RUNNING do
8:              if gs_movingState.terminal ≠ true then
9:                  node ← node.select(gs_movingState)
10:             end if
11:             propagate(simulate(gs_movingState))
12:         end while
13:     end while
14: end procedure
15:
16: procedure NODE.SELECT(gs_movingState)
17:     moves ← gs_movingState.getMoves()
18:     curEdges ← []
19:     for each move in moves do
20:         edge ← allEdges[move]
21:         if edge not found then
22:             edge ← new edge(move)
23:             allEdges[move] ← edge
24:         end if
25:         edge.N ← +1          ▷ increment observed count
26:         curEdges.push(edge)
27:     end for
28:     chosenEdge ← selection(curEdges)  ▷ Using UCT formula
29:     chosenMove ← chosenEdge.getMove()
30:     chosenEdge.V ← +1          ▷ increment visit count
31:     if chosenEdge.V == 1 then
32:         GLOBALS::SELECTION ← FINISHED
33:     end if
34:     gs_movingState.apply(chosenMove)
35:     is_state ← createInformationSet(gs_movingState)
36:     tt ← mcts.getTranspositionTable()
37:     chosenEdge.nextNode ← tt.findOrCreate(is_state)
38:     return chosenEdge.nextNode
39: end procedure
```

D. Information Sets in Hearthstone

We constructed the information sets in Hearthstone that consist of: (1) global publicly available information about the game such as the active player, game stage (e.g., mulligan or choose target), (2) data about Player1 and (3) data about Player2. The data about a player is modeled as a polymorphic structure with two possible types: the base *ISAnyPlayer* or *ISObservedPlayer* that inherits from *ISAnyPlayer*. The former type represents the perfect information portion of a player, i.e., everything that is visible about the player to all players. This includes such properties as the HP, armor, current crystals, maximum crystals, minions on board etc. However, to simplify the model and reduce the combinatorial complexity of states, we first sort minions by their ID numbers when doing the comparison of *IS-state* objects. As long as the two states contain the same set of minions (with the same attack values, health and ID numbers), their information sets are considered equal, even if positioning of the minions is different. The position of minions only matters when it affects attack or health. The ID is a number encoding a card's name, e.g., each "Prince Keleseth" card has the same ID. We also include a few specific properties of a player that could be theoretically derived from previous actions and states, such as whether a player has played an elemental card last turn (some cards gain extra effects based on this) or whether the "Prince Keleseth" buff has been applied in this game, which is a very unique effect that increases attributes of all minions in a player's deck.

The *ISObservedPlayer* comes with additional data about

the hidden information in the game: the hand and the secret zone. In an IS-state, we use a simpler model of a hand than the one used in a simulator. The hand is just a multi-set of cards' ID numbers, and any other properties of cards in the hand, such as effective mana cost, are ignored. The order of cards in hand is irrelevant. The secret zone is a set of secrets' ID numbers, because it is not possible to have more than one secret of a kind at the same time. Such a representation makes it possible to compare various approaches to modeling imperfect information:

1) Both players are modeled as *ISObservedPlayer* - this variant makes the assumption that the agent, which the tree is constructed for (the tree owner), determinizes the opponent and treats different determinizations as different states. This variant has the biggest granularity of states. **An exemplar consequence:** the sets of cards in both players' hands affect state comparisons.
2) Only the tree owner is modeled as *ISObservedPlayer* - here, we (i.e., the agent that is constructing the tree) do not differentiate between the states that differ with information we cannot see. **An exemplar consequence:** only the set of cards in the tree owner player's hand affects state comparison.
3) Only the currently active player is modeled as *ISObservedPlayer* - the idea is similar to (2), in (3), the tree owner hides the information about itself when simulating the opponent. This variant can be regarded as a symmetric version of (2). **An exemplar consequence:** the set of cards in a player's hands affects state comparison only if this player is to make a move in the state that is currently considered.

We have measured that both variants (2) and (3) work similarly without any significant difference in the playing strength of the resulting agent. The first variant, however, leads to significantly weaker bot because of the higher combinatorial complexity. In this variant, there are many more unique nodes in the transposition tables and therefore each node has less statistics. The strategy fusion problem arises with this approach as well.

III. Transposition Tables

Transposition tables [29], [30] were originally proposed as an enhancement to the alpha-beta algorithm, which reduces the size of minmax trees. As shown in paper [29], within the same computational budget, the enhanced algorithm significantly outperforms the basic one without the transposition tables. The term "transposition" refers to a state in the game that can be achieved in different ways.For simpler management, the MCTS tree is often modeled in such a way, that each unique sequence of actions leads to a state with a unique node in the tree. This leads to duplication of nodes, even for indistinguishable states. However, one of the benefits of such a duplication, is the fact that, whenever an actual action in the game is performed, the tree can be safely pruned into the sub-tree defined by the state the action lead to. All nodes that are either above the current one or on an alternative branch cannot be visited anymore, so there is no need to store them anymore. The problem is more complicated when transpositions are taken into account, and there is no longer one-to-one mapping between states and nodes. In such a case, the structure is no longer a tree per se, but a directed acyclic graph (DAG). When an action is played in the game, it is non-trivial to decide which nodes can be deallocated and which cannot because they might be visited again. In general, it would require a prediction model that can decide whether a particular state is possible to be encountered again. Storing all nodes, without deallocation, is detrimental not only for performance, but also the memory usage, which can become too high very quickly. Therefore, a worthwhile idea is to consider a probability a state will be encountered again and some threshold value based on which the nodes are pruned. Such an approach is suitable for a game-specific scenario but not for a universal case because there has not been proposed a general way of computing such a probability or setting the threshold. Therefore, we propose a solution based on reference counting described in the following subsection.

A. The Update Root Procedure

When an action is played in the actual game (not in a simulation), the algorithm first resets reference counts of all nodes. The node that corresponds to the state after the played action is the new root candidate. Next, the algorithm recursively traverses the nodes starting from the root candidate, increments the reference count in the currently visited node and proceeds through edges that lead to nodes with reference count equal to 0. After the recursive process terminates, all nodes with zero reference count are marked to be deallocated.

The above procedure is designed to be executed only once per action made in the game, so it does not bring high CPU overhead compared to the time required for simulations. We have measured this in Hearthstone. The profiling session consisted of 4 complete games with a one second clock for the moves. The average number of actions was 70 per game, what gives 280 executions of the update root procedure. The Monte Carlo simulations took 62% of the total time, whereas the update root procedure, described in the previous paragraph, took only 3.2% of the total time. However, because its only purpose is to reduce the memory usage, in the case of games, in which the memory footprint is low anyway, the procedure can be executed less frequently (e.g. every k actions or even once per game) to save some time. In Hearthstone, however, this procedure is required, because turning it off results in an out of memory exception after running for enough time (typically after a minute) with the setting of 20000 simulations (or more) per action.

B. Original Two-Layered Design

The proposed solution to storing the game tree is based on a two-layered structure of hash lookup tables (hashmaps). In many programming languages, the built-in hashmaps require keys to be integer values, so we decided to comply with this requirement. The whole idea is to find a node, if such

exists, given an information set as the input. Our approach requires developers to implement two things. The first one is a boolean method *equals(otherIS)*, that checks whether two information sets are equal or not and returns *true* or *false*, respectively. Usually, the *equals()* method is used with the accompanying *hash()* method to speed up the retrieval of elements from a collection. Our approach is based on a more complex structure of two, preferably but not necessarily orthogonal, ways of computing hash values. Therefore, we introduce a two-dimensional hash function (or, equivalently, a pair of functions, one per dimension) that returns two values, each being an integer number, that represent distinct hashes of a state represented by an information set:

Let A and B be two information sets. The critical constraint on their two hash values is as follows:

$$(A = B) \Rightarrow (A.h^{'} = B.h^{'}) \wedge (A.h^{''} = B.h'') \qquad (1)$$

Such a two-level structure has been proposed based on analysis of complex games such as Hearthstone. For simple enough games, the hash function can return the same value in both dimensions, i.e., by setting $h^{'} = h^{''}$. In Hearthstone, any hashing function we tested that produced one number led to many collisions and therefore had a huge negative impact on the performance of searching nodes. Extending the hash into two dimensions allows for using independent measures without the need of dimension reduction and combining them within one equation. As a result, we managed to decrease the average time of finding nodes by an order of magnitude in comparison with a single-dimensional hash. The hash functions, we used in Hearthstone are as follows:

```
h' = ActivePlayerID +
     sum(for_each(P in Players):
     {
         4*P.MaxCrystals + 41*StageType +
         90*(P.HP + P.Armor +
         40*P.MinionsCount
         + 581*P.HandSize))
     }
h'' = sum(for_each(P in Players)
       {
          P.WeaponDurability +
          for_each(M in P.Minions):
             M.HP + 170*M.ID +
             21*(M.Attack+1)
       }
```

where $ActivePlayerID$ is the index of the player, encoded as 0 or 1 for Player 1 and Player 2, respectively; $StageType$ is $0, 1, 2, 3,$ or 4 for *BasicAction*, *ChooseTarget*, *ChooseOne*, *Discover*, or *Mulligan*, respectively.

C. Using Transposition Tables in the Game

In our approach, transposition tables are used with the idea of separating the simulation game state and the decision game state. The decision game state is represented as an information set (IS-state). Firstly, we make a one-to-one mapping between information sets and nodes, so each IS-state has exactly one corresponding node. Information sets have already been defined in the previous sections as game state abstractions. A node is a functional structure that contains data required by the MCTS algorithm to operate. We managed to simplify nodes to only store a **collection of edges**. Because the MCTS version presented in this paper works with non-deterministic effects of actions, the edges cannot be stored as a fixed-sized array populated when a node is created. Instead, we store edges as a hashmap to satisfy the requirement that different sets of actions might be possible in consecutive visits by the MCTS algorithm to the same information set:

```
edges = hashmap<key: action, value: edge>
edge is composed of:
{
    stats: mcts_statistics
           (observedCount,
            visitCount,
            totalScore)
    current_next_node: node
}
```

Only the simulation game state is allowed to compute the legal moves, and it is possible that different simulation game states are mapped to the same information set. For example, there can be two simulation game states that differ with hidden information (indistinguishable from one player's perspective). Another example might be, when the AI designer/developer purposefully wants to simplify information sets and cluster more states together by ignoring some information available in the game. Therefore, we always look at the possible moves at the moment (by using the simulation game state) and then find the appropriate edge dynamically that corresponds to each specific move. An amortized cost of searching with hashmaps is $O(1)$. When a move is chosen, it is applied to the current GS-state and, based on the resulting GS-state, the IS-state (information set) is created. This information set is then used to retrieve the corresponding node.

IV. Randomness

Randomness is defined as a property of a game that some actions can have more than one outcome, i.e., that actions can lead to more than one distinct state, based on some arbitrary probability distributions. Players that participate in the game are not supposed to know the actual outcomes of the random effects before they materialize. They, however, may know the probability distributions underpinning random actions. In computer games, the game logic engine that is responsible for running the game (often referred to as the game server or game master) performs randomization in secrecy and informs the players when the effects of random actions become visible. A few examples of actions with nondeterministic outcomes are: shuffling a deck of cards, drawing a random card from a deck or rolling a die. Examples of games with randomness are Backgammon, Bridge, Poker, Settlers of Catan, Dungeons and Dragons, Magic the Gathering, and naturally, Hearthstone.

Fig. 2. Examples of cards including non-deterministic effects: drawing unknown cards (on the left) and random damage assignment (on the right).

Figure 2 shows two examples of cards with randomness in Hearthstone. Whether drawing a card is a random effect is a matter of interpretation and needs some further clarification. The rules of this game can be implemented in two ways. One way is to shuffle the deck of cards once and then each consecutive card is drawn from the top. In this case, drawing a card is not really a random effect but rather unveiling hidden information. The second way is not to shuffle the deck at all and give a random card to a player each time a card needs to be drawn. Both approaches are equivalent, because the deck is always supposed to be in a random order, and there are no effects in the game such as putting cards in a specific place in the deck.

Randomness raises the combinatorial complexity of a game. Let's consider a fully deterministic game with a branching factor of N. Now, if we make a change that each action on average has R possible random outcomes, the branching factor increases to N multiplied by R. Randomness is especially prevalent in card games. For instance, there are $52!$ ways a deck of 52 cards can be shuffled. Moreover, non-determinism of actions increases the difficulty of implementing the game engine to conform to the requirements of tree search algorithms such as MCTS. Consider the following problems:

- The MCTS algorithm stores statistics of players' actions: average score, total number of the action being observed, total number of the action being chosen. Let us now consider an action of playing the Arcane Missiles card shown in Fig. 2. In a given state, this action's average score should not be affected by how the damage will be split among the enemies, because the player could not know this outcome at the moment of playing the card. This suggests storing only one edge for this action. However, in the MCTS tree, there must be different nodes (states) that correspond to various outcomes of this action, which suggests having a separate edge for each result.
- The original Information Set Monte Carlo Tree Search algorithm requires the following property to be held. Let S be a state without perfect information. If identical sequences of actions $(a_1, ..., a_k)$ are applied from this state, then each one must end up with the same player active having the same set of available actions.

To tackle both mentioned problems, non-deterministic actions are split into a rational player's part and the so-called nature player's part. Continuing with the example of Arcane Missiles, the action would be split into:

```
PLAYER-1: Play Arcane Missiles
NATURE-P: Split 3 dmg among all enemies
```

or

```
PLAYER_1: Play Arcane Missiles
NATURE_P: Deal 1 dmg to a random enemy
NATURE_P: Deal 1 dmg to a random enemy
NATURE_P: Deal 1 dmg to a random enemy
```

The nature player is an artificial player that does not choose actions intentionally but rather performs them according to rules defined by the game (e.g., a probability distribution). The rational player observes such moves as random. A nature move determines the result of random calculations (it is one of the possible concrete realizations), therefore, after the move is generated, its outcome is deterministic. However, such a separation of player and nature moves as well as encoding determinations of random outcomes in the nature moves can be extremely difficult and time-consuming from the implementation point of view, that it might even be completely inapplicable in practice. In particular, video games are not designed in such a way to comply with the above mentioned model, so, if this model is used, one cannot necessarily separate the development of the actual game engine and the game AI.

Our goal was to make the development of the game engine maintainable for various games, so we decided to propose a model, in which actions can include any number of nondeterministic effects. The solution works in an integrated fashion with the information sets stored in transposition tables. Each time an action is made, the information set corresponding to the current state of simulation is searched for in the transposition table. Please refer to lines 34–37 in Algorithm 1 (pseudocode) for a possible implementation. Such an approach allows us not to introduce any explicit comparison operator for actions. In the proposed approach, two actions are different if they lead to different information sets, and they are equal otherwise. We keep only one edge for an action no matter how many resulting states it may have, so it represents a player's choice. The result of an action may vary in each MCTS iteration and because information sets and nodes correspond to each other, the same action might lead to different nodes in subsequent iterations. The UCT statistics gathered for the action are aggregated, so without any need for special treatment, the probability of random effects is taken into account. States which are more probable to occur will be visited more frequently by the MCTS, because it applies actions with their built-in randomness. At the same time, strong actions by means of the expected score (weighted

by the probability of occurrence) will be chosen more often in the selection phase.

V. RESULTS

A. Foreword

Before proceeding to the actual experiments results aimed at measuring efficiency of players, we wanted to first verify whether algorithms developed specifically to tackle the problems of randomness and incomplete information work in the game of Hearthstone.

The first test consisted in running 1,000,000 random simulations of the game. Each simulation had several unit tests to ensure the rules of Hearthstone are not violated. The second test consisted in running 100,000 matches between players that use the MCTS algorithm as defined in the paper. At this point, we were only interested whether there were no run-time errors at any point or problems with obtaining unequal states after applying the same sequence of actions from the same (stored and loaded) starting state. Both tests were successful. As a result, we can comment that that our way of implementing MCTS with dedicated algorithms for randomness and imperfect information is effectively error-free and does not put many constraints on how the simulator (forward-model) for a game is implemented. It is worthwhile noticing that our Hearthstone simulator was not specifically prepared to work with MCTS, yet the method is still easy to integrate with it.

B. Experimental setup

Our experimental setup consisted of the following AI players (controllers):

1) **Random** - an agent that performs actions according to the uniform random distribution among the currently available actions. The main purpose of including this agent is to make it serve as a baseline. A random controller can be also useful as a comparative benchmark in games, in which agents generally do not win against this agent 100% of the time; the relative difference of win-rates can be useful as a measure.
2) **EnhMCTS** - employs the MCTS algorithm as described in this paper. It is essentially a combination of the plain MCTS + our enhancement for randomness, incomplete information and transposition tables. We expected this player to consistently beat the *Random* player and not fall too much behind the "cheating" players described below.
3) **Hand Cheater** - is a *EnhMCTS* in which the player has perfect knowledge about the cards in the opponent's hand (hence, the cheater) but does not have access to the ordering of cards in decks.
4) **Full Cheater** - is a player that has perfect knowledge both about the opponent's hand and the ordering of cards in both players' decks. Such a player does not need to perform determinizations.

Both cheating controllers are used to find the upper bound on the performance of handling randomness and imperfect information.

The performance of players in Hearthstone is influenced by the decks they are using. For the experiments, we included four various types of decks that have also been used in the Hearthstone-based data mining competition [31]:

1) **Randomness** - this deck is made of cards with random effects exclusively. Examples of such cards are: *Arcane Missiles*, *Primordial Glyph* or *Animal Companion*. The idea of this deck was to increase the need of having a proper algorithm to handle randomness. This is also a deck that in theory gives the random player more chances to play better than more sophisticated approaches. The chosen hero for this deck was Mage.
2) **Minions** - this Hunter-based deck contains minion cards only. The goal was to have a relatively easy and streamlined deck to play.
3) **Control** - a "Cube Warlock" deck, which is very complex in terms of tactics. This deck allows agents to display their full potential as it requires long-term planning and tactical mastery.
4) **Aggro** - an average difficulty Paladin deck that is very fast and strong.

Each combination of decks and players was tested using 400 matches, which is above average in these kind of experiments.

C. Results with AI players

Let us start with baseline tests, which are depicted in Figure 3. Each grid corresponds to an experiment played between two players: *P1* and *P2*. The actual players used for the experiment are given in the top-most caption in each figure. Rows' labels are decks used by *P1* (the starting player), whereas columns are labeled by decks used by *P2*. The values in cells (i.e., on the intersections) contain the win ratio from the perspective of *P1*. A value of 1.0 means that *P1* has won all the games, a value of 0.5 represents a tie, and 0.0 is a perfect win for *P2*. If *P1* is equal to *P2*, then the main diagonal shows experiments played between identical players (so-called mirror matches); therefore, the results denote the first-player biases, i.e., how much more likely the player that goes first is to win in a particular setup.

The first test uses the weakest – Random – controllers. The main diagonal represents first move advantage, whereas other cells show biases between particular decks. For example, the *Aggro* deck has a clear advantage over the *Randomness* deck if it plays as first, whereas the *Minions* deck raises an upper hand against the *Control* when going first.

The next experiment, which is in the middle grid in Fig. 3, shows the same information for a pair of Full Cheaters, which are the strongest bots. It is interesting to note that these biases differ from those for the Random bots. The right-most part of Fig. 3 depicts those differences for each cell. In the experiment with Full Cheaters, $P1$ playing as *Aggro* significantly wins against any other deck. In comparison with Random vs. Random matches, Full Cheater is able to have a score higher by 0.20. The baseline results show that there

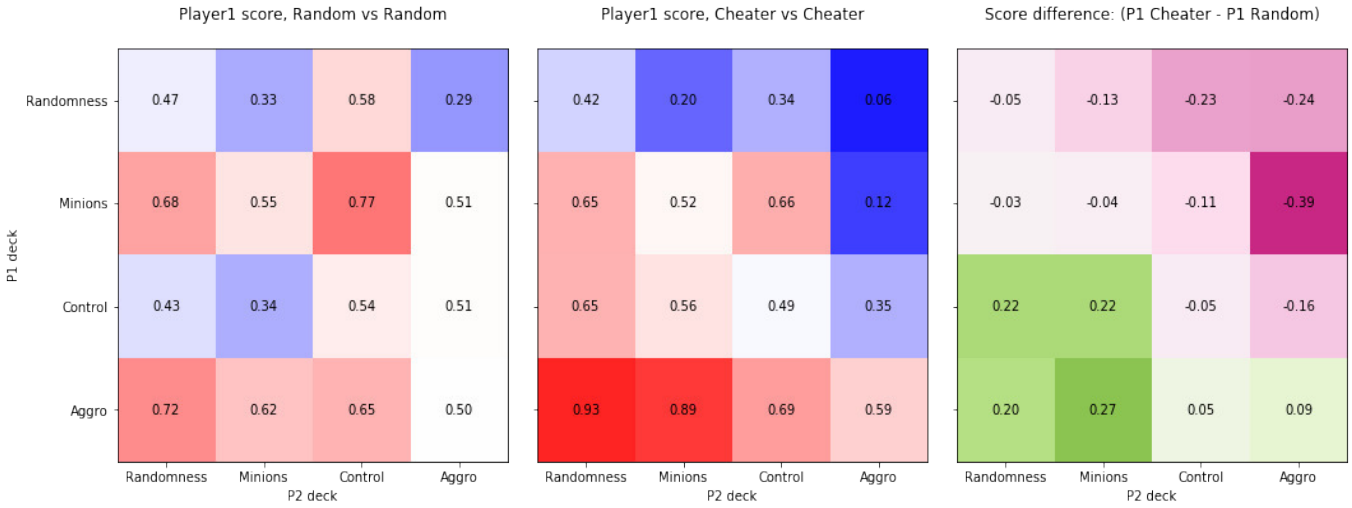

Fig. 3. The baseline scores. From the left: (1) Random Player vs Random Player, (2) Full Cheater vs Full Cheater and (3) the average score bias of Full Cheater compared to Random player

TABLE I

THIS TABLE CONTAINS THE AVERAGE SCORES OVER ALL MATCHES PLAYED BETWEEN THREE PAIRS PLAYERS THAT ARE DESCRIBED IN SECTION V-B. EACH TRIPLET OF COLUMNS IS DEVOTED TO ONE PARING OF PLAYERS. PLEASE NOTE THAT THE SCORES (P1 VS P2) AND (P2 VS. P1) ARE SHOWN FROM THE PERSPECTIVE OF THE PLAYER DENOTED BY P1.
THE THIRD COLUMN IN EACH TRIPLET DENOTES THE ADVANTAGE OF P2, WHICH IS CALCULATED ACCORDING TO THE FORMULA SHOWN IN EQUATION 2. THIS IS A DIFFERENCE IN TOTALSCORE(P2) - TOTALSCORE(P1).

		Hand Cheater (P1) vs. Full Cheater (P2)			**EnhMCTS (P1) vs Full Cheater (P2)**			**EnhMCTS (P1) vs Hand Cheater (P2)**		
		Score of P1		**Advantage**	**Score of P1**		**Advantage**	**Score of P1**		**Advantage**
P1 Deck	**P2 Deck**	**P1 vs P2**	**P2 vs P1**	**of P2**	**P1 vs P2**	**P2 vs P1**	**of P2**	**P1 vs P2**	**P2 vs P1**	**of P2**
Rnd	Rnd	0.35	0.58	0.07	0.39	0.54	0.07	0.47	0.53	0.00
Rnd	Minions	0.23	0.73	0.04	0.21	0.74	0.05	0.24	0.75	0.01
Rnd	Control	0.38	0.61	0.01	0.34	0.59	0.07	0.45	0.57	-0.02
Rnd	Aggro	0.05	0.94	0.01	0.07	0.91	0.02	0.08	0.93	-0.01
Minions	Rnd	0.62	0.4	-0.02	0.59	0.27	0.14	0.58	0.29	0.13
Minions	Minions	0.40	0.42	0.18	0.41	0.34	0.25	0.5	0.36	0.14
Minions	Control	0.62	0.26	0.12	0.57	0.17	0.26	0.73	0.19	0.08
Minions	Aggro	0.13	0.83	0.04	0.1	0.72	0.18	0.12	0.75	0.13
Control	Rnd	0.58	0.37	0.05	0.53	0.35	0.12	0.48	0.46	0.06
Control	Minions	0.48	0.39	0.13	0.39	0.34	0.27	0.42	0.46	0.12
Control	Control	0.45	0.49	0.06	0.38	0.34	0.28	0.46	0.36	0.18
Control	Aggro	0.24	0.6	0.16	0.24	0.47	0.29	0.28	0.54	0.18
Aggro	Rnd	0.94	0.06	0.00	0.93	0.04	0.03	0.94	0.06	0.00
Aggro	Minions	0.84	0.08	0.08	0.84	0.04	0.12	0.9	0.06	0.04
Aggro	Control	0.70	0.17	0.13	0.67	0.15	0.18	0.71	0.16	0.13
Aggro	Aggro	0.50	0.34	0.16	0.46	0.3	0.24	0.54	0.36	0.10
Column reference		I	II	III	IV	V	VI	VII	VIII	IX

are fundamental biases of decks and positions, which can be exploited by a good player – this is a general statement for Hearthstone.

One of the reasons we included the Random controller was to compare it to other agents. However, it turned out that each of our MCTS-based agents, i.e, Enhanced Vanilla MCTS, Hand Cheater and Full Cheater, achieves a 100% winrate over the random player. We can compare this result to 93% winrate reported in a literature [32], where the developed agent was also pitted against a (uniform) random player. This proves that **our implementation and setup of the MCTS algorithm itself is already very strong**.

Another experiment is aimed at measuring the impact of having full knowledge (Full Cheater), partial knowledge (Hand Cheater) or relying only on *fair* algorithms for incomplete knowledge (EnhMCTS). The baseline for the experiment is Full Cheater vs Full Cheater. All the results are presented in Table I. Please note that each experiment involves two players that switch sides after half of matches to avoid the starting role bias. However, to avoid confusion, the scores are always presented from the perspective of the first player, i.e., P1. The score of P2 can be calculated by $1 - Score(P1)$. Each third column contains an adjusted advantage in scores for P2. It is calculated as the total score of $P2$ minus the total score of P1. The total scores can be computed as follows:

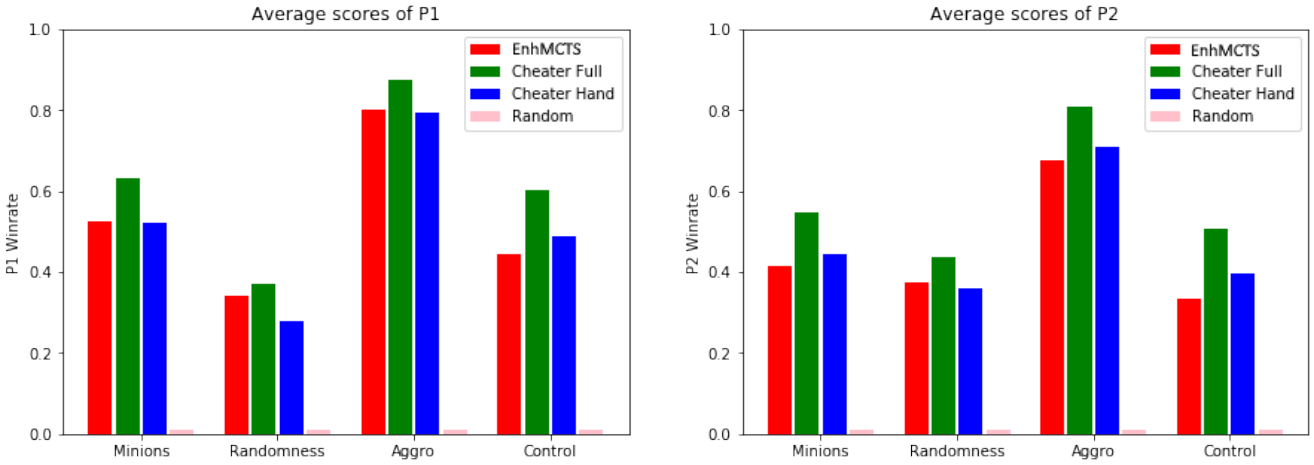

Fig. 4. Comparison of average win-rates for decks and bots. Scores are averaged for matches for any given deck-bot pair against all other decks and bots.

$$Score(P1) = Score(\text{P1 vs P2}) + Score(\text{P2 vs P1})$$
$$Score(P2) = 1 - Score(\text{P1 vs P2}) + 1 - Score(\text{P2 vs P1})$$

The advantage of P2 expressed as a difference in scores reduces to:

$$\begin{aligned} &advantage(P2) = Score(P2) - Score(P1) = \\ &= 2 - 2 * Score(\text{P1 vs P2}) - 2 * Score(\text{P2 vs P1}) = \\ &= 1 - Score(\text{P1 vs P2}) - Score(\text{P2 vs P1}) \end{aligned} \quad (2)$$

Let us show that Hand Cheater is not much worse than Full Cheater. The results of this experiment are shown in columns I-III of Table I (please see the last row for the columns references). The third column shows the advantage of Full Cheater over Hand Cheater, and most of the values are not far from 0.00, which would denote equal performance. Only in three matchups is the total Full Cheater's score over 0.15 higher than the total score of Hand Cheater. In one of the 16 tested cases, i.e., *Minions* vs *Rnd*, Hand Cheater achieved better score.

Compared to the baseline scores in Fig 3, when Hand Cheater is playing as the first player, the biggest drop in performance is with *Minions* vs *Minions*, *Control* vs *Aggro* and *Aggro* vs *Aggro*. When Hand cheater is playing as the second player, the biggest drop in performance is for *Aggro* vs *Control*.

EnhMCTS is indeed a weaker player than Full Cheater, but the total score advantage of the latter was not greater than 0.29 in all cases. Please note that the theoretical maximum possible advantage is equal to 2.00, which would happen if one player wins all the games when playing both sides (the first and second to go). Such a case occurs when either of the three presented players – Full Cheater, Hand Cheater or EnhMCTS, faces the Random controller. With this in mind, the results achieved by EnhMCTS are promising.

When comparing the scores to the Cheater vs. Cheater baseline in Fig. 3, the worst cases for EnhMCTS playing as the first player are *Control* vs *Minions*, *Control* vs *Randomness* and *Aggro* vs *Aggro*. When EnhMCTS is the second to go, the worst cases are *Control* vs *Control*, *Control* vs *Aggro* and *Minions* vs *Aggro*.

Finally, EnhMCTS is slightly weaker than Hand Cheater as shown in column IX in Table I. It is worth noticing that, in overall, the advantage of Hand Cheater over EnhMCTS is similar to the advantage of Full Cheater over Hand Cheater.

Figure 4 shows the scores of each player averaged over each performed experiment using Random controller, EnhMCTS, Hand Cheater and Full Cheater with respect to the deck used. Unsurprisingly, Full Cheater is the strongest player. However, all of the MCTS-based players are relatively close to each other. The results show great potential of the methods for tackling incomplete information and randomness applied in EnhMCTS.

VI. Conclusions

The MCTS algorithm is the *state-of-the-art* method for searching the space of combinatorial games to find a good action to play in the current state. However, while the algorithm is clearly established for deterministic perfect-information games, it does not transfer directly onto games with randomness and imperfect information. Such games pose many challenges, and it is often unclear how to adapt MCTS for them. In this paper, we showed a very practical solution to this problem that is generic enough to be applied to any combinatorial game with randomness and incomplete information. The solution is based on three main pillars. The first one is a new approach to Information Set Monte Carlo Tree Search that operates on two levels of granularity. Information sets are used together with determinizations. The second pillar is dynamic resolution of randomness by matching states that result from random moves on-the-fly. The third pillar is a two-layered Transposition Table that is very fast to query and works particularly well with both

the dynamic randomness resolution and information sets. Our approach does not enforce special constraints on how the game forward-model (simulator) is implemented, which is a huge advantage in a practical scenario, especially for commercial games.

To prove the method's efficacy, we have chosen the very complex game Hearthstone. The obtained results are very promising. Not only does the proposed algorithm work as intended, but also, as summarized in Figure 4, our methods for handling randomness and imperfect information fall only slightly behind a "cheating" agent that has full information about the game state.

For future work, we plan to perform further tests with different complex games with random effects and imperfect information. Next, we want to focus on the realm of non-card video-games, which feature huge combinatorial complexity and new challenges to overcome.

REFERENCES

[1] L. Kurke, "Ancient Greek Board Games and How to Play Them," *Classical Philology*, vol. 94, no. 3, pp. 247–267, 1999.

[2] J. McCarthy, "Chess as the Drosophila of AI," in *Computers, chess, and cognition*. Springer, 1990, pp. 227–237.

[3] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[4] C. E. Shannon, "XXII. Programming a Computer for Playing Chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[5] M. Buro and T. Furtak, "RTS Games as Test-Bed for Real-Time AI Research," in *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)*, 2003, pp. 481–484.

[6] M. R. Genesereth, N. Love, and B. Pell, "General Game Playing: Overview of the AAAI Competition," *AI Magazine*, vol. 26, no. 2, pp. 62–72, 2005.

[7] M. Świechowski and J. Mańdziuk, "Self-Adaptation of Playing Strategies in General Game Playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 367–381, Dec 2014.

[8] M. Świechowski, H. Park, J. Mańdziuk, and K.-J. Kim, "Recent Advances in General Game Playing," *The Scientific World Journal*, vol. 2015, 2015.

[9] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General Video Game Playing," *Dagstuhl Follow-Ups*, vol. 6, 2013.

[10] S. Sharma, Z. Kobti, and S. Goodwin, "Knowledge Generation for Improving Simulations in UCT for General Game Playing," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2008, pp. 49–55.

[11] S. Haufe, D. Michulke, S. Schiffel, and M. Thielscher, "Knowledge-Based General Game Playing," *KI-Künstliche Intelligenz*, vol. 25, no. 1, pp. 25–33, 2011.

[12] I. Szita, G. Chaslot, and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*. Springer, 2009, pp. 21–32.

[13] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 4, pp. 241–257, 2012.

[14] G. Van den Broeck, K. Driessens, and J. Ramon, "Monte-Carlo Tree Search in Poker Using Expected Reward Distributions," in *Asian Conference on Machine Learning*. Springer, 2009, pp. 367–381.

[15] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[16] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010.

[17] D. Robles, P. Rohlfshagen, and S. M. Lucas, "Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*. IEEE, 2011, pp. 305–312.

[18] F. Teytaud and O. Teytaud, "Creating an Upper-Confidence-Tree Program for Havannah," in *Advances in Computer Games*. Springer, 2009, pp. 65–74.

[19] M. Swiechowski, T. Tajmajer, and A. Janusz, "Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms," in *2018 IEEE Conference on Computational Intelligence and Games, CIG 2018, Maastricht, The Netherlands, August 14-17, 2018*, 2018, pp. 445–452. [Online]. Available: https://doi.org/10.1109/CIG.2018.8490368

[20] 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)*. IEEE, 2017, pp. 121–125.

[21] A. K. Hoover, J. Togelius, S. Lee, and F. de Mesentier Silva, "The Many AI Challenges of Hearthstone," *KI-Künstliche Intelligenz*, vol. 34, no. 1, pp. 33–43, 2020.

[22] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte Carlo Tree Search: A Review of Recent Modifications and Applications," 2021, submitted to Springer-Nature AI Reviews Journal. [Online]. Available: https://arxiv.org/abs/2103.04931

[23] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Proceedings of the 17th European conference on Machine Learning*, ser. ECML'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 282–293.

[24] S. Gelly and Y. Wang, "Exploration Exploitation in Go: UCT for Monte-Carlo Go," in *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, Dec. 2006. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00115330

[25] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak, "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search." in *AAAI*, 2010.

[26] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 120–143, 2012.

[27] I. Frank and D. Basin, "Search in games with incomplete information: A case study using bridge card play," *Artificial Intelligence*, vol. 100, no. 1-2, pp. 87–123, 1998.

[28] M. J. Ponsen, G. Gerritsen, and G. Chaslot, "Integrating Opponent Models with Monte-Carlo Tree Search in Poker." in *Interactive Decision Theory and Game Theory*, 2010.

[29] A. Kishimoto and J. Schaeffer, "Transposition Table Driven Work Scheduling in Distributed Game-Tree Search," in *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, ser. AI '02. London, UK, UK: Springer-Verlag, 2002, pp. 56–68.

[30] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989.

[31] A. Janusz, T. Tajmajer, M. Świechowski, Ł. Grad, J. Puczniewski, and D. Ślęzak, "Toward an Intelligent HS Deck Advisor: Lessons Learned from AAIA'18 Data Mining Competition," in *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2018, pp. 189–192.

[32] D. Taralla, "Learning Artificial Intelligence in Large-scale Video Games: A First Case Study with Hearthstone: Heroes of Warcraft," Ph.D. dissertation, Université de Liège, Liège, Belgique, 2015.