Enhancing Monte Carlo Tree Search for Playing Hearthstone

Jean Seong Bjorn Choe
School of Electrical Engineering
Korea University
Seoul, South Korea
garangg@korea.ac.kr

Jong-Kook Kim
School of Electrical Engineering
Korea University
Seoul, South Korea
Jongkook@korea.ac.kr

***Abstract*—Hearthstone is a popular online collectible card game (CCG). Hearthstone imposes interesting challenges in developing a search algorithm for the game AI. As a CCG, it has a considerable amount of hidden information from each player's private hand and deck. Moreover, the action space is full of stochastic actions compared to other similar games. That is, instead of a single move, each player is allowed to build a move sequence via various combinations of atomic actions. Therefore, when applying any heuristic search algorithm, the branching factor of the search space is extremely large. In this paper, we explore the use of Monte Carlo Tree Search (MCTS) with approaches to reduce the complexity of the search space and decide on the best strategy. First, we utilise state abstraction to present the search space as a Directed Acyclic Graph (DAG) and introduce a variant of Upper Confidence Bound for Trees (UCT) algorithm for the DAG. Next, we apply the sparse sampling algorithm to handle imperfect information and randomness and reduce the stochastic branching factor. This paper presents empirical evaluations of the proposed framework for Hearthstone and the experimental results suggest that our approach is well suited for developing a better AI agent.**

***Index Terms*—Monte-Carlo tree search, Hearthstone, artificial intelligence for games**

I. Introduction

Collectible Card Games (CCGs) contain challenging properties for applying operational AI. Players of CCGs must face informational uncertainty from their opponent's private hand, and innate stochasticity of drawing cards from own shuffled deck. Furthermore, CCGs allow deck building with innumerable combinations of selected cards to play against others, rather than use a predetermined set of cards. This characteristic poses a considerable similarity to General Game Playing (GGP), making it difficult for an agent to construct a precise evaluation function for overall gameplay. Moreover, these games usually encourage players to think and perform with the flexibility to play creative combinations of cards. Plus, the action of each turn is often a sequence of atomic actions rather than a single move.

Hearthstone[1] is one of the most popular CCGs. According to Blizzard Entertainment, the publisher of Hearthstone, there are more than 100 million players of Hearthstone. In addition to the popularity, it also has gained in popularity among AI researchers in recent years. Some group of researchers studied the deck building aspect of Hearthstone [1]–[4]. While some other authors focused their researches with developing an AI agent for playing Hearthstone [5]–[7]. There also have been several data mining/AI competitions about Hearthstone, including the one held in the CIG conference of the last year.

Three major factors create difficulties in applying AI to Hearthstone. First, AI agents must deal with an abundance of hidden information and randomness. Moreover, the stochasticity of actions in Hearthstone is relatively higher compared to those in other similar games, such as *Magic: The Gathering*[2]. Another defining feature of Hearthstone is its freedom of combining any kind of in-game action. For example, there is a certain type of action that commonly exists in CCGs called the 'attack move.' While usually performed separately from playing cards, these moves can be executed at any moment of a turn in Hearthstone. Therefore, searching for the optimal action sequence for each turn is more difficult in this game compared to others.

In this study, we present Monte Carlo Tree Search (MCTS) [8], [9] based approaches that mitigate the difficulties described above. To reduce the complexity of the state space, we show the utilisation of state abstraction to integrate rule-based heuristics and reflect the information set of each player. With the use of state abstraction, we are able to construct a Directed Acyclic Graph (DAG) instead of a tree for Monte Carlo search. Then, we demonstrate schemes for running Monte Carlo search on the DAG. For the imperfect information and the stochastic actions, we use *Expectimax* algorithm. In other words, a state ahead of uncertainty is presented as a chance node in the DAG, and subsequent uncertain states are sampled from the chance node. We also introduce an approach based on sparse sampling [10], in order to reduce stochastic branching factor of the chance nodes.

The structure of this paper is organised as follows: Section II presents existing research on AI in the domain of Hearthstone and compares it with our study. Our approaches and algorithms for playing Hearthstone are presented in Section III. Then, in Section IV, approaches to combine expert knowledge into the

[1]Blizzard Entertainment, Inc. https://playhearthstone.com

[2]Wizards of the Coast LLC. https://magic.wizards.com

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

framework is introduced. Section V describes the setup and results of the experiments. Finally, Section VI provides the conclusions.

II. Related Work

There are researches on the use of MCTS for Hearthstone AI [5], [6], [11]. In [11], the authors proposed the use of a deck database for opponent prediction and tested the range of parameters of MCTS with two heuristic state evaluation functions for rollout and tree policies. In this work, we allow our agents to know the list of cards in the opponent's deck. Also, we do not consider any form of evaluation functions. Zhang et al. proposed an approach to handle large branching factors in chance node, called *Chance event bucketing* [5]. Chance event bucketing groups pre-sampled outcomes of a chance node with domain-knowledge based, empirically tuned criteria. Then, the algorithm considers only a portion of events for each bucket. On the other hand, we use sparse sampling to reduce the stochastic branching factor of chance nodes. The search technique proposed in [6] takes similar approaches with the methods in this paper. In [6] authors proposed an algorithm to handle imperfect information with MCTS by constructing a DAG with information sets as keys of transposition table. We also build up a DAG of state abstractions, which represent the information set of the corresponding player. However, we use sampling-based methods to handle hidden information, in contrast with the algorithm suggested in [6] picking determinization at the beginning of the iteration of MCTS.

III. Proposed Method

A. Overview

In this section, we present approaches to deal with the difficulties of playing Hearthstone using MCTS. Our framework consists of three essential methods: (1) State abstraction that combines domain knowledge of Hearthstone and handles imperfect information of the game. (2) Modified implementation of Upper Confidence Bound for Trees (UCT) algorithm for DAG structure of the search space of abstractions. (3) Sparse sampling approach for randomness and incomplete information.

B. State Abstraction

The ground states of Hearthstone contain a lot of distinguishable features. The board, where both players can observe, may consist of up to 7 Minions, a Hero and its Hero Power and Weapon for each player's side. Each minion is characterised with a card from the pool of approximately 1,000 different cards including non-collectible ones. The cards also have approximately 20 distinctive attributes; to name some of them, there are Health, Zone Position, and also hidden ones like Order-Of-Play. Nevertheless, expert human players often consider only a small part of the ground state. For instance, zone positions of minions on board are usually neglected by players except for some specific situations. Inspired by this idea, we construct state abstractions with the minimum information of the ground states. With the introduction of state abstraction, the size of search space is adequately reduced, as many similar states are aggregated that ought to be separately considered without abstraction. We represent the space of abstractions using a transposition table [12], with the keys of the table are the abstractions. Thus, the search space becomes a single rooted Directed Acyclic Graph (DAG) where each node corresponds to a unique abstraction instead of a ground state. Consequently, the number of possible sequences of atomic actions is effectively reduced.

C. Modified Upper Confidence Bound for Directed Acyclic Graph (UCD)

Saffidine et al. [13] proposed an extension of Upper Confidence Bound for Trees (UCT) algorithm [8] for DAGs called Upper Confidence Bound for Directed Acyclic Graph (UCD). Unlike the plain MCTS algorithm, UCD stores statistics in the edges of DAG, rather than the nodes. Then, during the selection phase, UCD uses the modified UCT formula to select an edge from the outgoing edges of a node:

$$\arg\max_{e \in C(s)} Q_{d_1}(e) + c\sqrt{\frac{\ln N_{d_2}(e)}{n_{d_3}(e)}} \tag{1}$$

with:

$$d_1, d_2, d_3 \in \mathbb{Z}_0^+, \tag{2}$$

where $C(s)$ is the set of outgoing edges of a node s, $Q_{d_1}(e)$ is the average value of descendants edges down to depth d_1 of e, $n_{d_2}(e)$ is the sum of the visit counts of descendants edges down to depth d_2 of e, and $N_{d_3}(e)$ is defined as the sum of $n_{d_3}(e')$ for the siblings e' of e. Thus, if $d_1 = 2$, $Q_{d_1}(e)$ is average payoff over outgoing edges from each successor node of outgoing edges of the successor of the edge e.

Here, we propose a modified implementation of the UCD algorithm, which uses 2 parameters instead of 3 and has a more integrated structure than the implementation proposed in the previous work. First, to use the same form of the UCT algorithm, where the bias term of each option is determined by the total number of tries and the number of times the option is chosen, we replace N_{d_2} in equation 1 with N_d, the sum of the parametric visit count n_d of outgoing edges. N_d is accumulated in each node during Backpropagation. Moreover, since $d_1 = 0$ parameterisation means the value of a node can be calculated according to the preceding node, we only consider parameterisation of $d_1 > 0$. Q_d is also accumulated in each node during backpropagation. Therefore, the modified selection formula becomes:

$$\arg\max_{e \in C(s)} Q_{d_1}(s(e)) + c\sqrt{\frac{2\ln N(p(e))}{n_{d_2}(e)}}, \tag{3}$$

where $s(e)$ and $p(e)$ refer to the successor and the predecessor of an edge, respectively. $Q_{d_1}(s)$ and $n_{d_2}(e)$ are defined in the same way as in the original UCD framework.

We also present a simpler backpropagation algorithm to incrementally update $Q_d(s)$, $n_d(e)$ and $N(s)$. The basic idea is updating visit count and the payoff from leaf nodes to nodes

and edges in the traversed path recursively. By storing statistics in the nodes in the path, the algorithm naturally accumulates the values from all outgoing edges of each node. Also, with the given parameterisation, the algorithm backpropagate values in off-path nodes and edges recursively. This ensures each node to have statistics from all descendent nodes down to the given depth parameterisation. Thus, this backpropagation algorithm removes the need for recursive calculation for each statistic during the tree phase. The entire algorithm presented in the Algorithm 1.

D. Handling Imperfect Information

There are several methods have been suggested to employ MCTS to imperfect information games [9], [14]. In this work, we focus on sampling-based methods to treat imperfect information and randomness in the same manner. We introduce two typical approaches, *Perfect Information Monte Carlo* (PIMC) and *Information Set Monte Carlo Tree Search* (ISMCTS), then present our approach.

1) Perfect Information Monte Carlo (PIMC): PIMC [15], also called Determinized UCT, is a classical approach for domains of imperfect information. PIMC starts with generating a set of determinizations, the sampled game states from the information set, which contains indistinguishable states from the point of view of a player, of the root state. Then at the beginning of each iteration of MCTS, it chooses a determinization from the set and continues the search as if the domain has perfect information. Thus, PIMC uses a subtree of states with perfect information, for each generated determinization as the root of the subtree. The work in [16] studied that this approach can be useful for CCGs.

2) Information Set Monte Carlo Tree Search (ISMCTS): In Information set MCTS [17], each node in the search tree corresponds to the information set of a particular player. Similar to PIMC, it selects a determinization each time and searches the relevant part of the tree of information sets. Since states with different action sets are allowed to be aggregated in a node, selecting the best child is treated as subset-armed bandit problem here. The authors of [17] proposed two variants of ISMCTS, *Single-Observer ISMCTS* (SO-ISMCTS) and *Multiple-Observer ISMCTS* (MO-ISMCTS). SO-ISMCTS constructs a single tree whose nodes are information sets of the root player, while MO-ISMCTS generates a separate tree for each player.

In this work, we handle imperfect information with a single DAG. In the DAG, each node of state abstraction only contains the observable information of the player about to act. Thus, each node represents an information set of the active player. This is somewhat between SO-ISMCTS and MO-ISMCTS, because we maintain a single graph but information sets of both players can appear. The main difference in our approach is that determinizations are generated when hidden information is needed. In other words, we use *Expectimax* method and chance nodes for the hidden information. Note that since we have considered DAG, where only unique information sets of each player exist, the size of the search graph will not be

Algorithm 1 Modified UCD Algorithm

procedure MCTS(s)
  while search duration **do**
    while s is not terminal and not a leaf node **do**
      Select a best edge among the outgoing edges of s according to the equation (3).
      Set s as the successor of the selected edge.
    end while
    Run Expansion and Simulation phase from the current node unless the node is terminal.
    BACKPROPAGATION(s, r)
  end while
end procedure

procedure BACKPROPAGATION(s, r)
  repeat
    Update the total rewards and the update count for rewards of the current node with r.
    Update $p(s)$ of the current node s.
    for each incoming edge e of the current node **do**
      if e is not in the current traversed path **then**
        RECURSIVEUPDATE($e, r, d1, d2$)
      else
        Increase the edge visit count $n(e)$ by 1.
      end if
    end for
    Backpropagate to the next node on the current traversed path.
  until The root node is reached.
end procedure

function RECURSIVEUPDATE($e, r, d1, d2$)
  if $d1 > 1$ **then**
    Update the total rewards and the update count for rewards of the predecessor of e with r.
  end if
  if $d2 > 0$ **then**
    Increase the e's the edge visit count by 1.
    Increase the predecessor of e's the total edge visits $N(p(e))$ by 1.
  end if
  for each incoming edge ϵ of the incoming edges of the predecessor of e **do**
    RECURSIVEUPDATE($\epsilon, d1 - 1, d2 - 1$)
  end for
end function

intractably ramified despite the introduction of chance nodes. For example, although the number of all possible outcomes of a particular chance node representing the unknown opponent's hand arrangement can be unmanageable, the number of possible final resulting states after the opponent's turn is relatively feasible.

This approach, combined with sparse sampling we will explain in the next subsection, can be regarded as a simpler version of POMCP algorithm [18]. Whereas POMCP deals with exact histories, in our framework, the opponent appeared in planning repeatedly forgets his previous actions as his turn ends and the search graph expands to the nodes of the root player. Also, as the number of iterations increases, the virtual opponent in the search graph eventually "knows" the information of the root player. Despite this problem, we could not observe the effect of this theoretical flaw in experiments because we did not allow agents to think enough time to construct such a deep graph.

E. Sparse Sampling for Chance Nodes

Our system treats stochastic actions and opponent sequences with the expectimax algorithm. While classic expectimax requires prior knowledge of the distribution of the outcomes of each chance node, we presume that is practically impossible in our domain. To tackle this problem, the authors of [19] proposed the application of Sparse sampling algorithm [10] to MCTS, called *Sparse UCT*. This method samples a new outcome each time a chance node is visited until the total number of samples reaches a predetermined threshold. Thus, it constructs an approximated probability distribution for chance nodes with a limited number of samples. Here, we make full use of the transposition table to construct a discrete distribution of outcome nodes. So, when an outcome is sampled, we immediately look up the transposition table and find a matching transposition. And if there is one, instead of adding a separate node to the graph, we increase the sample count of the matching outcome. In this way, we are able to obtain the approximated probability distribution for chance nodes while maintaining a graph without duplication.

Double progressive widening (DPW) [20] algorithm also constraints the maximum number of samples for chance node. In contrast with the sparse UCT, DPW determine the threshold k by the visit count of the chance node n, with $k = \lceil Cn^{\alpha} \rceil$.

In our search algorithm, chance nodes appear too often with the 'end turn' nodes and many stochastic actions of Hearthstone. Therefore, the combinatorial complexity of each traversed path from the root to leaves easily becomes intractable as the search space expands. To handle this problem, we propose a new method called *Damped sampling* (DS). DS decreases the threshold for the number of samples with the number of encountered chance nodes having a large number of outcomes. For example, suppose that the first chance node in the traversed path has a threshold value k and k sample nodes have been expanded from the chance node. Then the next chance node encountered during the tree phase would have a threshold value of less than k. Therefore, this algorithm effectively decreases the number of outcomes with low realization probability and allows deeper search by reducing stochastic branching factor. Thus, the formula for the threshold k with DPW and DS combined becomes,

$$k = \lceil Cn^{\alpha} \eta^{-\mu} \rceil, \tag{4}$$

where μ is the count of encountered chance nodes that the number of outcomes exceeds the *threshold parameter* d, the η is the *damping parameter*. d and η are hyperparameters that should be determined empirically.

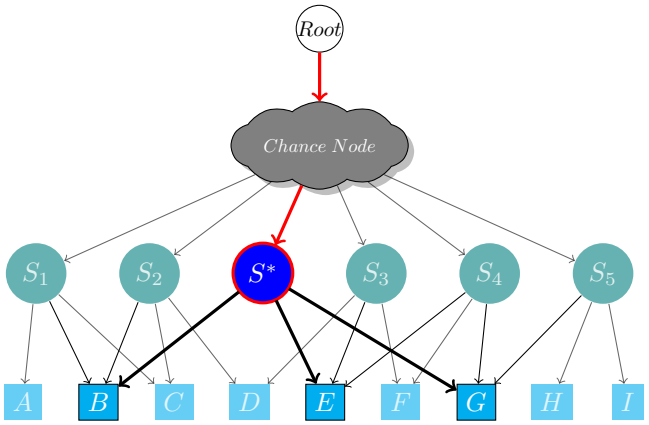

Fig. 1. Example of decision graph when a chance node representing *Discover* mechanic is selected. The chance node is selected from the current root and the outcome S^* is observed. Discover allows a player to choose one of three randomly selected choices from the corresponding pool of cards. Here, each sample S_i represents the three choices and squares with an alphabet means the resulting states from the choices. Although a new root with the state S^* is created because the observed outcome S^* has not been sampled from the chance node until the selection, it is possible to reuse the subgraphs rooted at B, D, and E, by maintaining the transposition table.

F. Search Time Allocation

In Hearthstone, it is often difficult to obtain all available action sequences due to the enormous number of outcomes from random actions. In other words, to attain a complete action sequence, a player should play an atomic action and observe the outcome, if it is stochastic. Thus, it is natural to have a search procedure that returns an optimal atomic action, rather than an optimal action sequence. If we consider each move selection as a process of *Absolute pruning* [21], we are able to reuse the statistics from previous iterations. Specifically, after an atomic action is selected, we run MCTS searching for the next atomic action with the selected node or the outcome of the selected chance node as a root node for the MCTS. This approach is also studied in [11] as *Search tree reuse* and in [22] as *Bridge-burning MCTS*. We also maintain the transposition table along with the consecutive searches in our implementation. This is extremely useful for handling mechanics like *Discover*. A more detailed explanation is shown in Figure 1.

With the given action selection scheme, as Zhang et al. reported [5], fixed time budget for the whole sequence would result in the unbalanced allocation of search duration. Meanwhile, in Hearthstone, most of the actions in one turn limit

the subsequent range of actions. For example, an action of playing cards consumes Mana, which is limited for each turn. Therefore, allocating time search time in a decreasing manner can be efficient for general cases. Thus, we use exponentially decreasing search time for a turn. For this, we introduce four parameters, k, T_{max}, T_{min}, and T_0 and allocate $T(i) = \gamma^{i-1} T_0$ seconds for the search for the i th move selection, where γ is determined by solving $T_{max} = \sum_{i=1}^{k} T(k)$. Therefore, the search time $T(n)$ for the n th selection is

$$T(n) = \begin{cases} T_0 \gamma^{n-1} \ , & \text{if } n \leq k \\ T_{min} \ , & \text{otherwise,} \end{cases} \tag{5}$$

where γ is determined by solving $T_{max} = \sum_{i=1}^{k} T_0 \gamma^{i-1}$.

IV. Integrating Domain-Specific Knowledge

A. Overview

In this section, we briefly explain heuristic approaches that integrate domain-specific knowledges for boosting MCTS for playing Hearthstone.

B. Category-based Action Filtering

Hearthstone often allows a player to perform self-destructive actions or wasteful actions that give disadvantages to the player in most of the situations. Thus, these kinds of actions can seriously harm the performance of the Monte Carlo simulation, especially in the absence of any sophisticated simulation policies. Therefore, we use heuristic categories based on the text information of cards. Each category of cards is subjected to a hand-crafted criterion based on domain knowledge, which examines the validity of playing the cards of the corresponding category using the given game state. For example, playing "Fireball" or any similar cards, which deal a certain amount of damage to a character, are filtered out when the target character is friendly. We observed applying category-based filtering to the uniform random rollout policy improves the performance of overall MCTS and even reduces the running time of the simulation phase. We also use the filtering in the tree phase of MCTS in a limited manner to prevent it from removing "niche" actions.

C. Obliged Actions

In some situations, there are actions that unconditionally profitable and should be performed without any consideration. A fine example of this kind of action is attacking the opponent's Hero in the absence of any obstacles. We can oblige these actions by removing turn-ending action. This heuristic is very simple to implement and observed to benefit both the tree phase and the simulation phase of MCTS.

V. Experiments

A. Setup

We use *SabberStone* [23], an open-source Heathstone simulator written in C# and the competition environment for CIG'18 and COG'19 Hearthstone AI competition [24]. Specifically, our testing environment is on par with Hearthstone client of Patch 12.2.0.27358, which was released on October 18, 2018, and played until November 4, 2018.[3] To evaluate our methods in various game environments, we used 3 competitively played decks: *Deathrattle Hunter*, *Even Warlock*, and *Odd Paladin*. In the rest of this paper, we simply put them as *Hunter*, *Warlock*, and *Paladin*, respectively. In all tested games, both players carried the same deck. Moreover, mulligan is skipped and the starting player for each game is randomly chosen. Also, we empirically determined the required parameters such as UCT constant for each deck. However, we used the parameters for allocating search time differently, based on the machine specification used for testing, to sync average search count per turn around various machines, but we did not change the value of γ. The determined parameters are listed in the Table I. We allocated search time for agents with the given parameters and the formula described in III-F.

TABLE I
Baseline parameters for applied methods

Symbol	Description	Hunter	Warlock	Paladin
c	UCT constant	0.25	0.4	0.8
w	Sparse UCT sampling width	48	32	48
η	Damping parameter		2	
k	Search	3	3	5
T_{max}	Time	20	20	30
T_{min}	Allocation		3	
T_0	parameters		10	

We employed parallelism by combining root parallelization and leaf parallelization [25] throughout the experiments. Particularly, we ran 6 independent MCTS, with each MCTS performs 2 simultaneous simulations. Then, we aggregated statistics gathered in each root and used it for the final move selection. When an action is chosen, sometimes some of the roots are incongruous with the resulting state abstraction from the chosen action due to the looseness of abstraction or randomness of the action. In that case, we replaced these roots with new roots but keeping the same transposition table of the previous root. By preserving the transposition table, we are able to minimise information loss.

Finally, all heuristic approaches have been introduced in Section IV are applied to all experiments as a part of the baseline.

B. UCD

We first evaluated the use of our modified UCD framework by comparing UCT without transposition table and UCD with different parameterisations. All agents used sparse UCT combined with Damped Sampling as their sampling scheme. The relative performance against UCT agent is in Table II.

[3]https://hearthstone.gamepedia.com/Patch_12.2.0.27358

TABLE II
RESULTS OF UCD WITH DIFFERENT PARAMETERISATIONS AGAINST UCT WITHOUT TRANSPOSITION TABLE.

Deck	$d1$	$d2$		
		0	1	2
Hunter	1	52.5%	49%	59%
	2	51%	61%	46.5%
	3	49%	63%*	51%
Warlock	1	47%	47%	53%
	2	49.5%	53%	47%
	3	48%	54.5%	57%*
Paladin	1	51%	50.5%	45%
	2	50%	52%	53.5%*
	3	41%	44%	52.5%

The win percentage from 200 games between two players with the same deck. The best parameterisation for each deck is marked with asterisk(*).

TABLE III
RESULTS OF DPW+DS WITH DIFFERENT PARAMETERISATIONS AGAINST SPARSE UCT

Deck	α	C		
		0.25	0.5	0.75
Hunter	0.4	53%	58%*	47%
	0.5	49%	57%	49%
	0.6	51%	50%	49%
Warlock	0.2	56%	64%	55%
	0.3	59%	67%*	56%
	0.4	59%	63%	56%
		0.5	0.75	1.00
Paladin	0.5	52%	42%	57%
	0.6	58%*	57%	50%
	0.7	51%	53%	50%

The win percentage from 100 games between two players with the same deck. The best parameterisation for each deck is marked with asterisk(*).

C. DPW and DS

Here, we investigated the effectiveness of the double progressive widening algorithm, combined with our damped sampling method. We tested empirically chosen interval for the parameters C and α of DPW algorithm and ran 100 games against sparse UCT agents. In these experiments, all agents used UCD, with the estimated best parameters for each deck. The results are presented in Table III. The result shows that Warlock benefits greatly from the use of DPW and DS. The result can be attributed to the fact that playing Warlock deck involves more chance nodes corresponding to drawing cards.

D. Comparison with TycheAgent

In order to investigate the performance of our agent in the COG'19 Hearthstone AI competition scheduled in 2019 [26], we conducted an experiment that compares our agent with the TycheAgent [27], which took place the third rank in both "Premade Deck Playing" and "User Created Deck Playing" tracks of the CIG'18 Hearthstone AI Competition. TycheAgent is also the best performing bot of the competition that the source is publicly available at the time of writing.

We arranged two experiments for the two of three decks[4] used for the premade deck playing track of the competition of the last year and 100 matchups are played with the same deck for both agents. In this experiment our agent ran on single thread and had shorter search time with the allocation parameters $T_{max} = 10$, $T_{min} = 1.5$, and $T_0 = 5$. The other parameters are determined based on the previous experiments and we did not do any deck-specific fine-tuning for the parameters. The results are presented in Table IV. Results show that our framework certainly out-performed TycheAgent.

[4]RenoKazakusMage deck was not compatible with SabberStone client of the version we used for experiments.

TABLE IV
A SUMMARY OF RESULTS FROM 100 GAMES AGAINST TYCHEAGENT

Deck	Winrate
MidRangeJadeShaman	66%
AggroPirateWarrior	78%
Parameters used	
c	0.3
d_1, d_2	2, 1
C, α, η	0.15, 0.5, 2
k, T_{max}, T_{min}, T_0	3, 10, 1.5, 5

VI. CONCLUSIONS

In this paper, we presented an MCTS-based framework to handle various challenges of playing Hearthstone. First, by adopting state abstraction, we are able to tackle three problems at once: the size of search space, the commutativity of atomic actions, and representing the partial observability. Second, we proposed an MCTS enhancing algorithm that exploits the structure of DAG in the selection phase and the backpropagation phase of MCTS. Finally, we introduced the use of sparse sampling to handle large uncertainty. We also proposed a complementary algorithm called *Damped sampling*, that mitigates the stochastic ramification of the search graph. Thus, we provide a generic framework for games of similar difficulties and open-ended baseline for future improvement. Our experimental result shows that our approach is well fitted for various environments in the domain of Hearthstone.

REFERENCES

[1] Z. Chen, C. Amato, T. H. D. Nguyen, S. Cooper, Y. Sun, and M. S. El-Nasr, "Q-DeckRec: A Fast Deck Recommendation System for Collectible Card Games," *IEEE Conf. Comput. Intell. Games, CIG*, vol. 2018-Augus, pp. 1–8, 2018.

[2] A. Stiegler, C. Messerschmidt, J. Maucher, and K. Dahal, "Hearthstone deck-construction with a utility system," *Ski. 2016 - 2016 10th Int. Conf. Software, Knowledge, Inf. Manag. Appl.*, pp. 21–28, 2017.

[3] L. F. W. Goes, A. R. Da Silva, J. Saffran, Á. Amorim, C. Franc, T. Zaidan, B. M. Olímpio, L. R. Alves, H. Morais, S. Luana, and C. Martins, "Honing stone: Building creative combos with honing theory for a digital card game," *IEEE Trans. Comput. Intell. AI Games*, vol. 9, no. 2, pp. 204–209, 2017.

[4] A. K. Hoover, A. Bhatt, J. Togelius, S. Lee, F. de Mesentier Silva, and C. W. Watson, "Exploring the hearthstone deck space," 2018, pp. 1–10.

[5] S. Zhang and M. Buro, "Improving hearthstone AI by learning high-level rollout policies and bucketing chance node events," *2017 IEEE Conf. Comput. Intell. Games, CIG 2017*, pp. 309–316, 2017.

[6] M. Swiechowski, T. Tajmajer, and A. Janusz, "Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms," *IEEE Conf. Comput. Intell. Games, CIG*, vol. 2018-Augus, pp. 1–8, 2018.

[7] I. Kachalsky, I. Zakirzyanov, and V. Ulyantsev, "Applying reinforcement learning and supervised learning techniques to play hearthstone," *Proc. - 16th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2017*, vol. 2018-Janua, pp. 1145–1148, 2018.

[8] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[9] D. Whitehouse, P. Rohlfshagen, S. Tavener, S. M. Lucas, C. B. Browne, S. Colton, D. Perez, E. Powley, S. Samothrakis, and P. I. Cowling, "A Survey of Monte Carlo Tree Search Methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[10] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large MDPs," *Ijcai*, pp. 193–208, 1999.

[11] A. Santos, P. A. Santos, and F. S. Melo, "Monte Carlo tree search experiments in hearthstone," *2017 IEEE Conf. Comput. Intell. Games*, pp. 272–279, 2017.

[12] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," *2008 IEEE Symp. Comput. Intell. Games, CIG 2008*, pp. 389–395, 2008.

[13] A. Saffidine, T. Cazenave, and J. Méhat, "UCD: Upper confidence bound for rooted directed acyclic graphs," *Knowledge-Based Syst.*, vol. 34, pp. 26–33, 2012.

[14] J. Heinrich and D. Silver, "Smooth uct search in computer poker," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[15] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak, "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search," *24th AAAI Conf. Artif. Intell.*, pp. 134–140, 2010.

[16] 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 Trans. Comput. Intell. AI Games*, vol. 4, no. 4, pp. 241–257, 2012.

[17] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 2, pp. 120–143, 2012.

[18] D. Silver and J. Veness, "Monte-Carlo planning in large POMDPs," *Adv. Neural Inf. Process. Syst.*, pp. 2164–2172, 2010.

[19] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower Bounding Klondike Solitaire with Monte-Carlo Planning," *Proc. 19th Int. Conf. Autom. Plan. Sched.*, pp. 26–33, 2009.

[20] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous upper confidence trees," in *International Conference on Learning and Intelligent Optimization*. Springer, 2011, pp. 433–445.

[21] J. Huang, Z. Liu, B. Lu, and F. Xiao, "Pruning in uct algorithm," in *2010 International Conference on Technologies and Applications of Artificial Intelligence*. IEEE, 2010, pp. 177–181.

[22] H. Baier and P. I. Cowling, "Evolutionary MCTS for Multi-Action Adversarial Games," *IEEE Conf. Comput. Intell. Games, CIG*, vol. 2018-Augus, pp. 253–260, 2018.

[23] C. Decoster, J. S. B. Choe *et al.* Sabberstone. [Online]. Available: https://github.com/HearthSim/SabberStone

[24] A. Dockhorn and S. Mostaghim. Hearthstone ai competition. [Online]. Available: http://www.is.ovgu.de/Research/HearthstoneAI.html

[25] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 2008, vol. 5131 LNCS, pp. 60–71.

[26] A. Dockhorn and S. Mostaghim, "Introducing the hearthstone-ai competition," 2019.

[27] K. Bornemann. Tycheagent. [Online]. Available: https://dockhorn.antares.uberspace.de/wordpress/wp-content/uploads/2018/11/HearthstoneBot-Bornemann-MCTS.zip