Monte Carlo tree search with state merging for reinforcement learning in Regular Decision Processes

Gabriel Paludo Licks[a,*], **Fabio Patrizi**[a] **and Giuseppe De Giacomo**[a, b]

[a]Sapienza University of Rome
[b]University of Oxford

Abstract. This paper introduces a novel algorithm for Reinforcement Learning (RL) in Regular Decision Processes (RDPs), a model of non-Markovian decision processes where dynamics and rewards depend on regular properties of the history. Our algorithm is inspired by Monte Carlo tree search (MCTS), yet it is improved with state merging capabilities. Performing merges allows us to evolve the tree model into a graph over time, as we periodically perform similarity tests borrowed from automata learning theory to learn states that are equivalent to one another. This results in improved efficiency and scalability over standard MCTS. We present empirical results that demonstrate orders of magnitude performance improvement over the state-of-the-art RL algorithms for RDPs.

1 Introduction

The use of Monte Carlo tree search (MCTS) has shown to be highly successful in a variety of applications in the field of reinforcement learning, with outstanding performance when dealing with large state spaces [14, 8, 23]. The study of techniques for state merging, state abstraction, or state aggregation, is widely present in the literature of reinforcement learning, and a number of these have been applied to the case of MCTS [11, 12, 27, 15, 24]. A common observation is that these techniques assume that the observations returned by the environment constitute a complete representation of the current state of the world. However, most real-world applications and domains are naturally non-Markovian [6]. That is, the dynamics and rewards are functions of the entire history. Therefore, complete histories of interactions with the environment have to be considered in order to behave optimally.

A recently introduced formalism, called Regular Decision Processes (RDPs) [6], captures a well-behaved class of NMDPs. While the dynamics of NMDPs can show an arbitrary dependency on the history of past observations, RDPs work under the assumption that the reward and dynamics are regular functions of the history, and therefore representable by finite-state machines. This implies the existence of an underlying state space where states are determined by histories. This is a key property that makes them amenable to a range of solution techniques. We describe an example of an RDP to further help in drawing an intuition of the concepts introduced throughout the text.

Example 1 (Flickering Grid [21])**.** *In a Flickering Grid, an agent is assigned a simple navigational task: it has to reach a designated goal location from its initial location. The agent can observe its current position in the grid. However, sometimes its sensors malfunction, and the agent fails to observe its current position. Although with a low probability, this can repeat for many consecutive time steps. It is clear that, as long as the agent keeps track of the history of actions, its current location can be deduced from the history with probability 1. Observations make it easier, but they are not essential. This makes a flickering grid a Regular Decision Process. The state is the current agent's location, and it can always be exactly inferred from the history.*

In this paper we study how to devise a Monte Carlo tree search algorithm with state merging for reinforcement learning in RDPs. More specifically, we employ state merging techniques from probabilistic automata [4] learning and seamlessly combine them within MCTS. Previous algorithms for reinforcement learning in Regular Decision Processes have also used automata learning techniques to learn the model of the decision process, however they solved it using planning algorithms such as policy iteration and showed low scalability. The motivation for MCTS is exactly its ability to deal with larger state spaces, which we show to greatly improve when combined with state merging. Since automata learning is in itself a hard problem, we further adapt MCTS to generate states on demand for every history such that, even with poor estimates of its values, the agent is still able to behave in a best-effort fashion in states it has not visited often enough. Our contributions include the introduction of a novel solution that combines MCTS and state merging, closely designed to tackle the current drawbacks in recent works in RDPs, and an empirical evaluation of the algorithm in comparison to state-of-the-art techniques in the literature of RDPs.

2 Preliminaries

We start by introducing the key definitions of the setting of this paper.

Non-Markov Decision Processes. A *Non-Markov Decision Process* (NMDP), cf. [6], is a tuple $\mathcal{P} = \langle A, O, R, \zeta, \mathbf{T}, \mathbf{R} \rangle$ with components defined as follows. A is a finite set of *actions*, O is a finite set of *observations*, $R \subseteq \mathbb{R}_{\geq 0}$ is a finite set of non-negative *rewards*, ζ is a special symbol that denotes *episode termination*. Call the elements of $\mathcal{H} = (AOR)^*$ *histories* and the elements of $\mathcal{E} = \mathcal{H}A\{\zeta\}$ *episodes*. Then, $\mathbf{T} : \mathcal{H} \times A \rightsquigarrow (O \cup \{\zeta\})$ is the *transition function*, and $\mathbf{R} : \mathcal{H} \times A \times O \rightsquigarrow R$ is the *reward function*. The transition and reward functions can be combined into the *dynamics function* $\mathbf{D} : \mathcal{H} \times A \rightsquigarrow (OR \cup \{\zeta\})$, which describes the probability to observe next a certain pair of observation and reward, or termination,

* Corresponding Author. Email: licks@diag.uniroma1.it.

given a certain history and action. Namely, $\mathbf{D}(or|h,a) = \mathbf{T}(o|h,a) \cdot \mathbf{R}(r|h,a,o)$ and $\mathbf{D}(\zeta|h,a) = \mathbf{T}(\zeta|h,a)$. We often write an NMDP directly as $\langle A, O, R, \zeta, \mathbf{D} \rangle$. A *policy* is a function $\pi : \mathcal{H} \rightsquigarrow A$. The *uniform policy* π_{u} is the policy defined as $\pi_{\mathrm{u}}(a|h) = 1/|A|$ for every a and every h. The *dynamics of* $\mathcal{P}$ *under a policy* π describe the probability of an episode remainder, given the history so far, when actions are chosen according to a policy π; it can be recursively computed as $\mathbf{D}_\pi(aore|h) = \pi(a|h) \cdot \mathbf{D}(or|h,a) \cdot \mathbf{D}_\pi(e|haor)$, with base case $\mathbf{D}_\pi(a\zeta|h) = \pi(a|h) \cdot \mathbf{D}(\zeta|h,a)$. Since we study episodic reinforcement learning, we require episodes to terminate with probability one, i.e., $\sum_{e \in \mathcal{E}} \mathbf{D}_\pi(e|\varepsilon) = 1$ for every policy π.[1] This requirement ensures that the following value functions take a finite value. The *value of a policy* π *given a history* h, written $\mathbf{v}_\pi(h)$, is the expected sum of future rewards when actions are chosen according to π given that the history so far is h; it can be recursively computed as $\mathbf{v}_\pi(h) = \sum_{aor} \pi(a|h) \cdot \mathbf{D}(o,r|h,a) \cdot (r + \mathbf{v}_\pi(haor))$. The *optimal value given a history* h is $\mathbf{v}_*(h) = \max_\pi \mathbf{v}_\pi(h)$, which can be expressed without reference to any policy as $\mathbf{v}_*(h) = \max_a \left(\sum_{or} \mathbf{D}(o,r|h,a) \cdot (r + \mathbf{v}_*(haor)) \right)$. The *value of an action* a *under a policy* π *given a history* h, written $\mathbf{q}_\pi(h,a)$, is the expected sum of future rewards when the next action is a and the following actions are chosen according to π, given that the history so far is h; it is $\mathbf{q}_\pi(h,a) = \sum_{or} \mathbf{D}(o,r|h,a) \cdot (r + \mathbf{v}_\pi(haor))$. The *optimal value of an action* a *given a history* h is $\mathbf{q}_*(h,a) = \max_\pi \mathbf{q}_\pi(h,a)$, and it can be expressed as $\mathbf{q}_*(h,a) = \sum_{or} \mathbf{D}(o,r|h,a) \cdot (r + \mathbf{v}_*(haor))$. A policy π is *optimal* if $\mathbf{v}_\pi(\varepsilon) = \mathbf{v}_*(\varepsilon)$. For $\epsilon > 0$, a policy π is ϵ*-optimal* if $\mathbf{v}_\pi(\varepsilon) \geq \mathbf{v}_*(\varepsilon) - \epsilon$.

Probabilistic Automata. We follow [3]. A *Probabilistic Deterministic Finite Automaton* (PDFA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \tau, \lambda, \zeta, q_0 \rangle$ where: Q is a finite set of states; Σ is an arbitrary finite alphabet; $\tau : Q \times \Sigma \to Q$ is the transition function; $\lambda : Q \times (\Sigma \cup \{\zeta\}) \to [0,1]$ defines the probability of emitting each symbol from each state ($\lambda(q,\sigma) = 0$ when $\sigma \in \Sigma$ and $\tau(q,\sigma)$ is not defined); ζ is a special symbol not in Σ reserved to mark the end of a string; $q_0 \in Q$ is the initial state. An important characteristic in PDFA is called state distinguishability, which allows for identifying states in the automaton that produce different probability distributions of emitting symbols. For a quantity $\mu > 0$, we say that $\mathcal{A}$ is μ*-distinguishable* if $\max_x |\lambda(q_1,x) - \lambda(q_2,x)| \geq \mu$ for every string x and every two distinct states q_1 and q_2. This is a key characteristic exploited by algorithms for learning probabilistic automata.

Regular Decision Processes. A *Regular Decision Process (RDP)* [6] is an NMDP $\mathcal{P} = \langle A, O, R, \zeta, \mathbf{T}, \mathbf{R} \rangle$ admitting a *finite automaton* $\mathcal{A}$ that describes its dynamics. Specifically, for every history h, the output of the transducer is $\mathcal{A}(h) = \mathbf{D}(\cdot|h,\cdot)$, i.e. the distribution on the next observation and reward given an action.[2] Results in [20] show that probabilistic automata, specifically PDFA, is the automata model that accurately captures the dynamics of an RDP.

Markov Decision Processes. A *Markov Decision Process (MDP)* [5, 18] is a decision process where the transition and reward functions (and hence the dynamics function) depend only on the last observation in the history, taken to be an arbitrary observation when the history is empty. Thus, an observation is a complete description of the state of affairs, and it is customarily called a *state* to emphasise this aspect. Hence, we talk about a set S of states in place of a set O of observations. All history-dependent functions—e.g., transition and reward functions, dynamics, value functions, policies—can be seen as taking a single state in place of a history.

Episodic RL. Given a decision process $\mathcal{P}$ and a required accuracy $\epsilon > 0$, Episodic Reinforcement Learning (RL) for $\mathcal{P}$ and ϵ is the problem of an agent that has to learn an ϵ-optimal policy for $\mathcal{P}$ from the data it collects by interacting with the environment. The interaction consists in the agent iteratively performing an action and receiving an observation and a reward in response, until episode termination. Specifically, at step i, the agent performs an action a_i, receiving a pair of an observation o_i and a reward r_i, or the termination symbol ζ, according to the dynamics of the decision process $\mathcal{P}$. This process generates an episode of the form $a_1 o_1 r_1 a_2 o_2 r_2 \dots a_n \zeta$. The collection of such episodes is the data available to the agent for learning.

Monte Carlo methods. Monte Carlo (MC) methods in reinforcement learning [25] are simulation-based methods, or rollout-methods, for estimating the value function. They are characterised by sampling experiences, i.e. episodes, from the environment and use their average returns to estimate the value function. With enough sampling, the averages should converge to the corresponding expected values, which is the principle idea behind Monte Carlo methods. Many algorithms rely on Monte Carlo methods, e.g. Monte Carlo tree search. A background on Monte Carlo tree search is given in Chapter 4 as we concomitantly describe the concepts we borrow from it to devise our novel algorithm.

3 Related Work

There exists in the literature a number of related studies to this one with the goal of improving the performance of MCTS by means of state merging, state abstraction, or state aggregation. Here we highlight and point out the differences between existing work and our approach. [11] propose a value-based state aggregation, and studies how aggregating states according to their expected value can improve the performance of MCTS in MDPs. [12] and [27] show how abstracting the state space to a smaller dimension helps in the performance of MCTS by exploiting approximate state homomorphisms in MDPs. [15] provide a theoretical regret analysis on the benefit of merging states in Monte Carlo tree search and operating on a graph instead of a tree, however it does not provide a practical algorithm and merge criteria for doing so. [24] show a method for state abstractions based on the geometry of the state space, aggregating newly expanded nodes if similar to a neighbour node in the same depth of the tree. The common characteristic of the studies mentioned above are that none of them considers the setting of NMDPs. That is, state merging, abstraction, or aggregation, are preformed with the assumption that each state holds enough information in order to be aggregated, and not considering the whole history. In contrast, our techniques aim to improve the efficiency of the above algorithms in the non-Markov settings, taking into consideration that both the reward and the dynamics function are history-dependant.

The closest approaches that take into consideration both rewards and dynamics of the decision process are [1, 20, 21]. These approaches focus on Regular Decision Processes, that assume that the reward and dynamics are representable by finite automata. Given this characteristic, these approaches rely on automata learning techniques to effectively cluster histories into one representative state. However, there are fundamental differences on the approaches [20], [21], and our approach, which directly impacts the number of samples before

[1] A constant probability p of terminating at every step amounts to a discount factor of $1 - p$, see [18].

[2] In [6] the functions $\mathbf{T}$ and $\mathbf{R}$ are represented using the temporal logics on finite traces LDL_f. Here instead we use directly finite automata to express them. Note that all $\mathbf{T}$ and $\mathbf{R}$ representable in LDL_f are indeed expressible through finite automata.

achieving a near-optimal policy. [20] propose an iterative algorithm that, at each iteration, learns an automaton by sampling a batch of episodes randomly and solves it by value iteration, with guarantees that better automata and policies are returned at each iteration. [21] introduce a smarter way of sampling episodes by taking advantage of an automaton that is built incrementally, while using RMax to guide exploration and sample episodes that lead the automata learning algorithm to learn states with higher return, and solving it by value iteration. Our approach also builds an automaton incrementally, but we will consider a new state for every visited history, i.e. we will learn directly on histories while learning states concomitantly. This allows us to take complete advantage of the sampled data and compute (initially rough) value estimates for every history, and therefore have a policy with higher average return earlier on. As a drawback, we initially face a large tree, which motivates our decision on employing an algorithm such as MCTS, in conjunction with the fact that MCTS does not need a complete and accurate model of the environment, in contrast to dynamic programming methods such as value iteration. The tree eventually converges to a graph as states get merged, and MCTS progressively refines the value estimates for every node, guided by an exploration policy such as UCB1 [13].

4 Monte-Carlo tree search with state merging

We start this chapter by describing the basic concepts of Monte Carlo tree search (MCTS) and its execution steps. We proceed by describing the main ideas and concepts in the literature of Probabilistic Deterministic Finite Automata (PDFA) learning and its state merging procedure. Finally, we show a detailed overview of our algorithm, specifically on how we connect MCTS and state merging from PDFA-learning to devise a novel algorithm for reinforcement learning.

4.1 Monte Carlo tree search

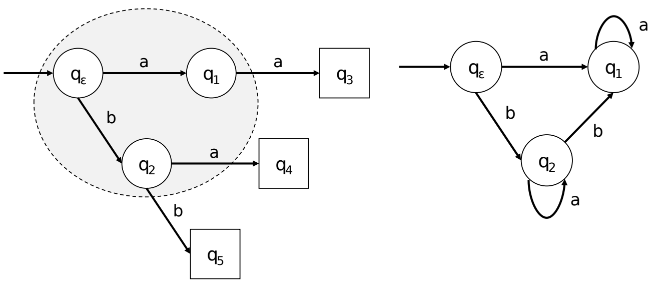
We describe the classic MCTS execution scheme [13, 8]. The algorithm operates over a tree structure, where every node in the tree represents a history h, and transitions between nodes are defined by applying an action a, and observing a pair of observation o and reward r, such that $h' = haor$ then maps to the next node. Every node in the tree keeps track of basic statistics throughout the algorithm's execution, namely $N(h)$ for the number of times a node has been visited, $X(h)$ for the empirical sum of returns from that node onwards, such that the expected average return value for a given history can be computed as $\hat{\mathbf{q}}(h) = X(h)/N(h)$ [13].

The algorithm works by iteratively performing the following four procedures: selection, expansion, simulation, and backup. The *selection* procedure traverses the tree from the root node up to a leaf node and, and selects nodes according to some tree policy that is usually the UCB1 for trees action selection policy π_s [2, 13]:

$$\pi_s = \pi_{UCT}(h) = \underset{a \in A}{\arg\max}\, \hat{\mathbf{q}}(ha) + c\sqrt{\frac{2 \ln N(h)}{N(ha)}}, \qquad (1)$$

such that c is a constant that controls the exploration factor of the selection policy, and any action where $N(ha) = 0$ yields ∞.

Once the selection procedure reaches a leaf node, the algorithm performs an *expansion* to the tree by adding a new child node from the previously selected node. Expansions usually follow a criterion that relies on a minimum number of times that a node in the tree should be visited before expanding it further, assuming enough simulations are performed to estimate its expected return. A *simulation* is performed when a leaf node or a newly expanded node is reached, and it consists of executing a *rollout* policy π_r up to a specific depth or to the end of the episode, that traditionally consists of sampling actions uniformly at random. The total return of a simulation is then used to perform the *backup* procedure, which backpropagates the return to update the value of each traversed node of the tree. The backpropagation procedure at a given iteration conventionally consists of updating the empirical returns $X(h)$ and visit counts $N(h)$, resulting in an update of the expected average return $\hat{\mathbf{q}}(h)$ for every visited node in the traversed history.



(a) Hypothesis automaton. (b) Target automaton.

Figure 1: Incremental learning of hypothesis automaton with safe states as circles and candidate states as squares, and target automaton. $\Sigma = \{a, b\}$.

4.2 Learning probabilistic automata and state merging

Probabilistic Deterministic Finite Automata (PDFA) is the model that accurately captures the dynamics of RDPs [20], hence the choice for this formalism. We now present a general description of a PDFA-learning algorithm, highliting the concepts and features of algorithms in [19, 9, 17, 3, 4] that are borrowed for developing our algorithm. These algorithms work by building a *hypothesis* automaton that approximates the true *target* automaton, and incrementally constructs this hypothesis by means of adding new states and/or merging states. Let us illustrate by taking Figure 1 as an example. Figure 1b shows the graph of a target automaton, and Figure 1a shows the graph of a hypothesis automaton learned so far. States in the hypothesis automaton are classified as *safe* (circles) or *candidate* states (squares). Safe states in the hypothesis are one-to-one with states in the target automaton, while candidate states are the frontier of the current hypothesis. As a state is promoted to safe, the frontier is then expanded by adding new candidate states for every possible transition from the recently promoted state.

The core of the algorithm are statistical tests, which are periodically performed to determine when a candidate state can be considered safe if distinct from all other states, or merged to another state if equivalent to another state:

$$Test(q, q', \mu, \delta, \epsilon), \qquad (2)$$

where q and q' are states in the hypothesis and μ is the distinguishability parameter [3]. The parameters δ and ϵ are the desired confidence and accuracy, respectively. Note that the implementation of the $Test$ function may differ from author to author. However, we consider here a general definition that considers the probably approximately correctly (PAC) framework [26], specifically the one in [21], such that the tests output the correct answer with high confidence and accuracy, based on the distinguishability μ of the automata states.

Algorithm 1: MCSM

```
1  $\mathcal{G} \leftarrow$ initialise graph with root node as initial state
2  foreach episode do
3      $h \leftarrow \varepsilon$;
4      $s \leftarrow \mathcal{G}(h)$;
5      while episode is not done do
6          $a \leftarrow$ choose action according to policy $\pi_s$ if $s$ is safe, otherwise rollout policy $\pi_r$;
7          $o, r \leftarrow$ apply action $a$;
8          $h \leftarrow haor$;
9          $s \leftarrow \mathcal{G}(h)$;
10     $merges$ and $promotions \leftarrow$ perform statistical tests on $\mathcal{G}$;
11     $\mathcal{G} \leftarrow$ apply new merges and promotions to the graph $\mathcal{G}$, if any;
12     backpropagate total return collected from episode $h$;
```

With enough data collected for each state the algorithm is able to estimate the empirical transition probabilities between states with high accuracy, such that it is able to confidently take decisions to distinguish states and approximate the hypothesis graph to the true target with PAC guarantees.

4.3 Combining MCTS and state merging from automata learning

We have introduced the basic concepts and procedures of MCTS, probabilistic automata learning, and state merging. Now we describe how these can seamlessly be put together as one. Figure 2 provides an illustration of this process.

Let us start with the first MCTS procedure, *selection*, and let $\Sigma = AOR$ be such that every symbol in the alphabet is a transition triple of action, observation, and reward. All nodes in the tree that are traversed during selection will correspond to *safe* states, except for leaf nodes that will correspond to *candidate* states. It is easy to see that the concept of nodes in the tree that have already been expanded in classic MCTS aligns to the one of safe nodes in probabilistic automata learning. That is, the collected data and statistics each of node/state expanded so far is enough to confidently represent its possible outcomes. To illustrate, Figure 2a shows a tree structure where circle nodes represent safe states where the selection phase of MCTS takes place, and square nodes represent candidate states.

Following the selection procedure, the *expansion* will correspond to the promotion of candidate states (leaf nodes) to safe states. As a state is promoted to safe, new candidate states are created. That is, the newly promoted safe state is expanded and new candidate states, i.e. child nodes that represent its direct outcome of applying an action, are added to the tree as leaves.

The *simulation* procedure will correspond to the execution of an exploration policy (the *rollout* policy) after a candidate state is reached. Notice that each history and its prefixes generated from simulations past the candidate state are added to the tree as a node. In other words, each candidate state holds a prefix tree of possible outcomes from the rollout policy. As an example, Figure 2a illustrates smaller square nodes that are created according to the simulations from each candidate state (bigger square nodes). Finally, the *backup* procedure will use the total return of the complete history to backpropagate the value of all nodes in the tree in which the history traverses.

As mentioned in the previous section, statistical tests will be performed periodically in order to merge or promote candidate states to safe. The results from these statistical tests will determine when a node will be promoted or merged. It is simple to see that promoting a candidate state to safe affects the tree by expanding it. However, merging states remains a more elaborate process. To illustrate it, let us take the example in Figure 2b and assume that a statistical test returns that candidate state c_1 and safe state q_5 are in fact the same, i.e. their empirical distributions over history suffixes are similar enough such that the test considers them equal. Merging candidate c_1 to safe q_5 will amount to redirecting to q_5 all transitions that once pointed to c_1 and propagating the prefix tree of c_1 to q_5, as illustrated in Figure 2c. Note that merges can also introduce cycles to the graph, as illustrated in Figure 2d, where candidate c_0 is merged to its parent safe state q_1. Given that the two states reached from different histories are actually equal, we can refine the count and expected return estimates of the safe state that the candidate was merged onto by taking $N(q_1) = N(q_1) + N(c_0)$ and $X(q_1) = X(q_1) + X(c_0)$, and recursively to all its children. With subsequent applications of merges, the tree eventually converges to a graph, and every node that previously represented a history, now represents equivalence classes of histories. With the use of MCTS as heuristic to guide the search, we manage to test and learn states that expand the model towards the goal, as data is collected towards higher rewarding nodes.

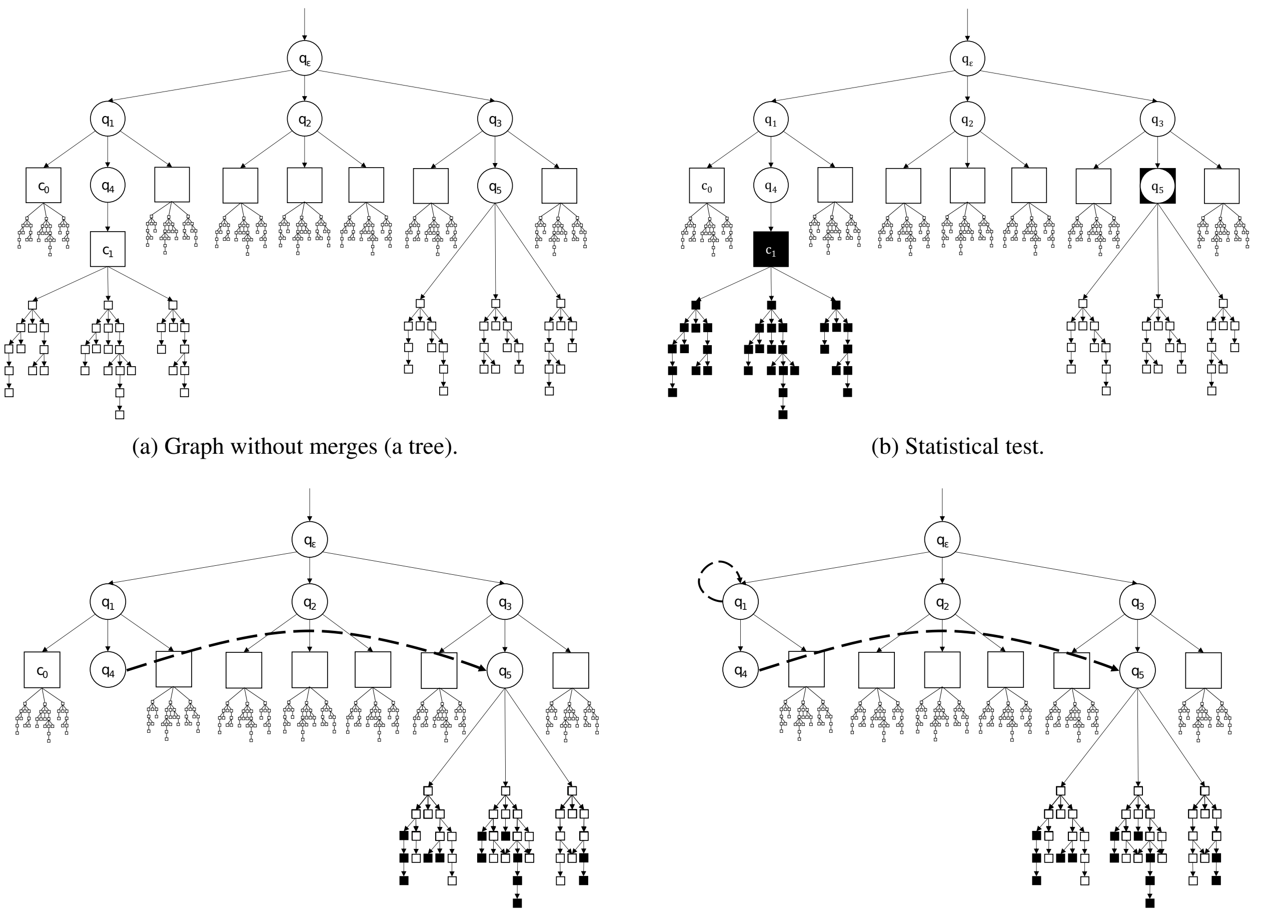
Finally, with the guarantees mentioned in the previous section for the similarity tests we employ from PDFA learning literature, we learn the correct and minimal model of the dynamics [3], i.e. the MCTS tree will converge to the minimal graph over time. Regarding the policy, we inherit guarantees from [13] that it converges in the limit as UCB1 progressively approximates the value function's expected return.

4.4 Algorithm overview

We now describe our algorithm MCSM, a novel algorithm that combines MCTS with state merging from probabilistic automata learning theory. An overview of MCSM is presented in Algorithm 1, developed over the ideas presented in the previous section.

The first step of the algorithm is to initialise a graph with a root initial state (Line 1). With an initialised graph, the agent starts interacting with the environment episodically (Lines 2-12). For each episode, its history (i.e. sequence of actions, observations, and rewards) is initialised as the empty string ε (Line 3). For every history there is a corresponding node in the graph. If a history does not map to a node in the current graph, nodes are created on demand.[3] For

[3] Note that, in classic MCTS, values are not saved and nodes are not created after the expanded region (fringe) of the tree. Instead, we create nodes that



(a) Graph without merges (a tree).

(b) Statistical test.

(c) Merging a candidate state to a safe state.

(d) Performing a cyclic merge.

Figure 2: Overview of Monte Carlo tree search with state merging. For presentation purposes, we omit transition symbols $\sigma \in \Sigma$, given $\Sigma = AOR$.

the case of the empty history, the state returned by the graph is the initial state that is the root node (Line 4). While the episode is not over (Line 5), the agent chooses actions based on the current state given by the history so far. The agent chooses actions using the selection policy π_s if the current state is safe, i.e. the history maps to a node in the expanded part of the tree, otherwise according to the rollout policy π_r (Line 6). Applying actions to the environment will return an immediate observation and reward (Line 7). The observation and reward are appended to the history (Line 8), and the agent updates its current state given the current history (Line 9). When an episode is over, we perform statistical tests over the new graph and return new merges and promotions (Line 10), which are then applied to the structure of the graph (Line 11), if any. The final step of the algorithm is to backpropagate the total return of the collected history to all nodes traversed in that episode (Line 12).

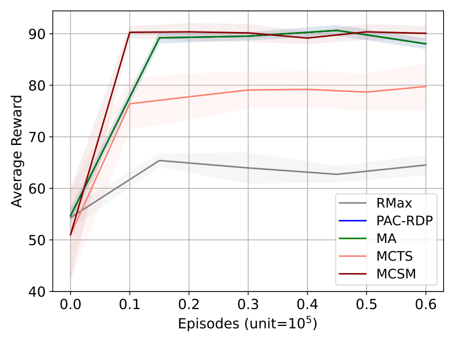
5 Experiments

We carry out an empirical evaluation to understand the capabilities of MCSM. We instantiate the algorithm employing a UCB1 policy [2] for the selection phase and a uniform policy in the rollout phase, standard in MCTS algorithms [13, 8]. For the statistical tests, we employ the similarity test based on [4] that ensures PAC guarantees. The hyperparameters used in the statistical tests, such as distinguishability, confidence, and accuracy, are detailed in appendix.

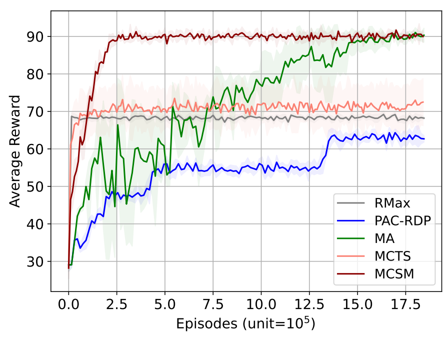
We compare the performance of MCSM against PAC-RDP [20] and Markov Abstractions (MA) [21], which are the current state-of-the-art algorithms for reinforcement learning in Regular Decision Processes. As a baseline, we consider the RMax [7] algorithm and a classic MCTS implementation [13, 8]. The RMax algorithm is a baseline to reflect the performance of an agent that assumes observations to be Markovian, i.e. acting without taking the history in consideration, while classic MCTS is a baseline that relies on complete histories in order to take decisions. Other algorithms in the literature of RDPs such as S3M [1] and RNN2RDP [22] are not in the scope of this empirical evaluation as they are algorithms that do not provide guarantees[4].

keep track of values and nodes for all histories collected throughout the interactions with the environment, as such data is essential for performing the statistical tests.

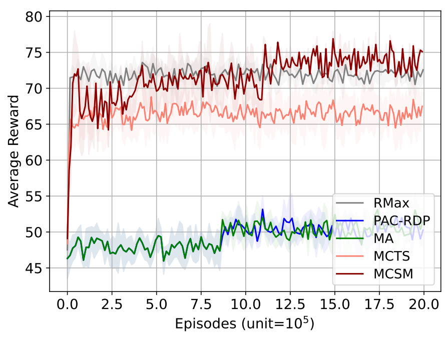
[4] Analysing the empirical results in their respective original papers, the reported performance of the S3M algorithm shows that it outperforms our algorithm in the Malfunction MAB domain, it performs comparably in the Rotating MAB, and it has poorer performance than ours in the remaining domains. The reported performance of RNN2RDP, on the other hand, shows that it outperforms our algorithm in the Malfunction MAB domain, while it performs comparably in the remaining domains.



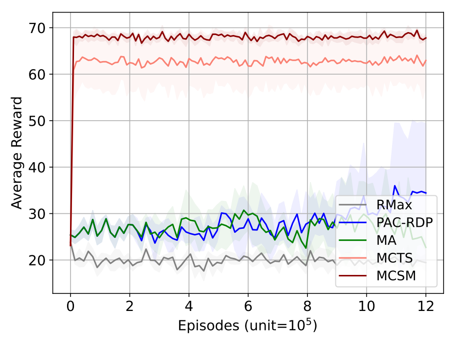
(a) Rotating MAB $k = 2$



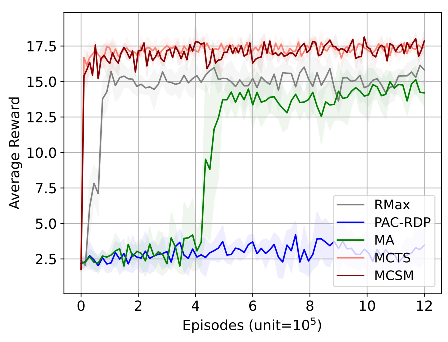
(b) Reset-Rotating MAB $k = 8$



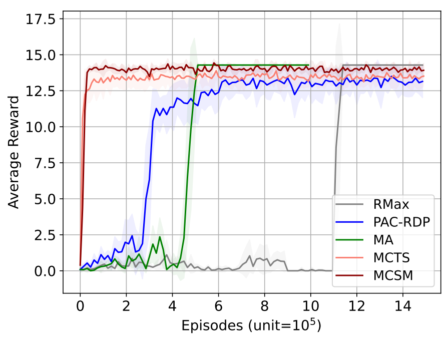
(c) Malfunction MAB $k = 5$



(d) Cheat MAB $k = 4$



(e) Rotating Maze $k = 3$



(f) Flickering Grid $k = 8$

Figure 3: Empirical evaluation of MCSM.

We consider six different domains from the literature of RDPs [1, 21]. The domains comprise Multi-Armed Bandits (MAB) problems and grid environments that exhibit different temporal aspects that have to be considered by an agent in order to achieve optimal behaviour. We now briefly explain each domain. Note that domains are defined in terms of a parameter k, which makes the domains more complex as it is assigned larger values, but it has a different meaning in each domain.

Rotating MAB. [1] It is a MAB with k arms where reward probabilities depend on the history of past rewards: they are shifted $(+1 \bmod k)$ every time the agent obtains a reward. The optimal behaviour is not only to pull the arm that has the highest reward probability, but to consider the shifted probabilities to the next arm once the agent is rewarded, and therefore pull the next arm.

Reset-Rotating MAB. [21] Similar to the definition above, however reward probabilities are reset to their initial state every time the agent does not get rewarded.

Malfunction MAB. [1] It is a MAB in which one arm has the highest probability of reward, but yields reward zero for one turn every time it is pulled k times. Once this arm is pulled k times, it has zero probability of reward in the next step. The optimal behaviour in this domain is not only to pull the arm with highest reward probability, but to take into consideration the number of times that the arm was pulled, such that once the arm is broken the optimal arm to pull in the next step is any other arm with nonzero probability of reward.

Cheat MAB. [1] It is a standard MAB. However, there is a *cheat* sequence of k actions that, once performed, allows maximum reward at every subsequent step. The cheat is a specific sequence $[a_i, a_{i+1}, \ldots, a_k]$ of arms that have to be pulled. Observations in this domain are the arms pulled by the agent. The optimal behaviour is to perform the actions that compose the cheat, and thereafter any action returns maximum reward.

Rotating Maze. [1] It is a grid domain with a fixed goal position that is 5 steps away from the initial position. The agent is able to move in any direction (up/left/down/right) and the actions have success probability 0.9, with the agent moving into the opposite direction when an action fails. Every k actions, the orientation of the agent is changed by 90 degrees counter-clockwise. Observations in this domain are limited to the coordinates of the grid.

Flickering Grid. [21] An 8x8 grid domain ($k = 8$) with goal cell $(3, 4)$, where at each step the agent observes a flicker (i.e. a blank observation) with 0.2 probability. Episodes in this domain are set to 15 steps.

5.1 *Empirical evaluation*

Figure 3 shows the performance of our algorithm in comparison to PAC-RDP [20] and MA [21], in addition to the RMax [7] and MCTS [13, 8] baselines. Each line represents an average over 5 runs with its standard deviation. At every 10k training episodes, we evaluate the current greedy policy of each agent. Note that the greedy policy is the best policy available to the agent at a given moment as it takes actions aiming to maximise the total return, and it is better than the policies used while training that include an exploration bias. Each evaluation is an average over 50 episodes, where for each episode we measure the accumulated reward divided by the number of steps taken. In the MAB domains, episodes are of length 10. The Maze and Flickering Grid domains have a maximum episode length of 15, with episodes terminating whenever the goal position is reached.

Our algorithm outperforms the state-of-the-art algorithms for RDPs, i.e. PAC-RDP and MA. MCSM has better performance in the Reset-Rotating MAB, Malfunction MAB, Cheat MAB, and Flickering Grid domains. It has comparable performance with MA in the Rotating MAB. Regarding the baselines MCTS and RMax, MCSM outperforms both over time. Interestingly, RMax outperforms all algorithms earlier in the Malfunction MAB, an example domain in which assuming observations as Markovian results in collecting high returns immediately. Finally, the MCSM performance is orders of magnitude higher in the Reset-Rotating MAB, the Cheat MAB, and the Flickering Grid domains, converging millions of episodes ahead of other algorithms and baselines.

A key general takeaway is that, although we are interested in a model-based approach for learning the underlying dynamics of the RDP and solving the decision process with the learned model, sometimes working directly on histories is beneficial. A clear example is shown in the Rotating Maze domain, where MCTS and MCSM perform comparably since reaching the goal takes only 7-8 steps on average, and the domain stochasticity is very low (0.1 probability of action failure), therefore not producing a tree of substantial depth before finding the optimal policy. A counterexample is the Malfunction MAB domain, which has higher stochasticity in actions' outcomes, which shows that merging states gradually improves the policy performance over time, while classic MCTS struggles to compute the policy for all possible histories. Nonetheless, it is clear that merging states will reduce the search space and reduce the number of states for which an algorithm has to estimate the value function, and therefore allows for improved performance in the long run.

Reproducibility, source code, and experimental setup. The results shown in this paper are reproducible, and the definitions of our experiments are provided in the source code, along with the exact random seeds to reproduce the results. The source code contains instructions on how to run the reported experiments, as well as running experiments in general using our instantiation of MCSM. A virtual environment is available for running the code with its requirements. Our experiments were carried out in a server running Ubuntu 18.04.5 LTS, with 512GB RAM, and 80 cores model Intel Xeon E5-2698 2.20GHz. Each training run takes one core and the necessary amount of compute and time is empirically linear on the number of episodes and steps required for showing convergence. Find the complete code, parameters, and instructions to run the experiments at https://github.com/whitemech/mcts-state-merging-code-ecai24.

6 Conclusions

We introduced MCSM, a practical and novel algorithm for Monte Carlo tree search with state merging. The algorithm is built on concepts from automata learning algorithms that intuitively connect with Monte Carlo tree search. We apply statistical tests from probabilistic automata learning to decide when nodes on the tree should be merged, incrementally transforming the structure into a cyclic graph [5]. The merging process improves the efficiency and scalability of standard MCTS, as complete branches of the tree get merged and the overall tree size consequently gets reduced, resulting in a smaller search space. A cyclic graph ultimately results in a model in which the agent can exploit by generating infinite episodes, in contrary to the naturally episodic structure of a tree. Considering a candidate state for every history allows us to take complete advantage of the sampled data and compute value estimates for all histories, and therefore have a policy with higher average return earlier on. Ultimately, nodes in the graph are a representation of an equivalence class of histories, i.e. histories that correspond to the same underlying state instead of multiple nodes in a tree.

Our experimental evaluation shows that MCSM overperforms standard MCTS and learns better policies faster as a result of its merges. MCSM also overperforms state-of-the-art algorithms for Regular Decision Processes, except for one example in which the performances match. While we apply this algorithm to Regular Decision Processes, which have a one-to-one correspondence to PDFA, we believe it can also be applied to larger classes of non-Markov decision processes, since we expand the fringe of candidate states by considering a new candidate state for every visited history, and therefore we can assume it applies to several history-based decision processes.

Finally, we limited our implementation based on classic MCTS, which employs the UCB1 selection policy and a uniform rollout policy. However, we believe recent improvements in MCTS and policies can further benefit the performance of our algorithm with trivial adaptations in future work. A memory-specific analysis of the algorithm is a clear analysis to be made in future work, as the scope of this paper has only focused on the sample efficiency of the algorithm. In other directions, techniques to prune the search could be applied, including the study of safety and advice properties that can be used to direct the search when the agent visits histories that violate such properties [16, 10].

A Hyperparameters details

We now report and explain the parameters used in the experimental evaluation in Table 1. The hyperparameters regard the statistical tests from probabilistic automata learning, and are strictly domain-dependent. First, the $delay$ parameter [21], sets the length of histories that must be considered in order to distinguish states during the statistical tests. Second, the distinguishability parameter μ [3], a parameter that needs to be computed taking into consideration the probabilities that define the domain dynamics, such that it reflects the probability of sampling distinguishing histories. Finally, the parameter n is an upper bound on the number of states, i.e. on the actual number of safe states in the target automaton [4]. Though these parameters are domain-specific, if domain knowledge is not available one can employ a search strategy for finding approximate values [4].

Table 1: Domain-specific parameters.

Domain	$delay$	μ	n
Rotating MAB $k = 2$	1	0.35	10
Reset-Rotating MAB $k = 8$	1	0.0874	10
Malfunction MAB $k = 5$	6	0.0127	10
Cheat MAB $k = 4$	5	0.0254	10
Rotating Maze $k = 3$	3	0.15	200
Flickering Grid $k = 8$	1	0.224	70

In addition to the parameters reported in Table 1, all experiments use other domain-independent parameters that are fixed. More specifically, we highlight confidence δ and accuracy ϵ parameters that regard to the PAC guarantees of the statistical test, both set to value 0.2 on all experiments. Other parameters are fixed to their standard values according to [4].

[5] To the best of our knowledge, this is the only proposed algorithm based on MCTS that converges to a cyclic graph structure.

Acknowledgements

This work has been partially supported by the ERC-ADG White-Mech (No. 834228), the PRIN project RIPER (No. 20203FFYLK), the PNRR MUR project FAIR (No. PE0000013), and the Sapienza Project MARLeN.

References

[1] E. Abadi and R. I. Brafman. Learning and solving regular decision processes. In *IJCAI*, 2020.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3), 2002.

[3] B. Balle, J. Castro, and f Ricard Gavaldà. Learning probabilistic automata: A study in state distinguishability. *Theoretical Comput. Sci.*, 473:46–60, 2013.

[4] B. Balle, J. Castro, and R. Gavaldà. Adaptively learning probabilistic deterministic automata from data streams. *Mach. Learn.*, 96(1-2):99–127, 2014.

[5] R. Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[6] R. I. Brafman and G. De Giacomo. Regular decision processes: A model for non-Markovian domains. In *IJCAI*, 2019.

[7] R. I. Brafman and M. Tennenholtz. R-MAX: A general polynomial time algorithm for near-optimal reinforcement learning. *JMLR*, 3:213–231, 2002.

[8] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[9] A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *J. Mach. Learn. Res.*, 5:473–497, 2004.

[10] G. De Giacomo, M. Favorito, L. Iocchi, F. Patrizi, and A. Ronca. Temporal logic monitoring rewards via transducers. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 860–870, 9 2020. doi: 10.24963/kr.2020/89. URL https://doi.org/10.24963/kr.2020/89.

[11] J. Hostetler, A. Fern, and T. Dietterich. State aggregation in Monte Carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[12] N. Jiang, S. Singh, and R. Lewis. Improving UCT planning via approximate homomorphisms. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1289–1296, 2014.

[13] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46056-5.

[14] K. L. Kroeker. A new benchmark for artificial intelligence. *Communications of the ACM*, 54(8):13–15, 2011.

[15] E. Leurent and O.-A. Maillard. Monte-carlo graph search: the value of merging similar states. In *Asian Conference on Machine Learning*, pages 577–592. PMLR, 2020.

[16] D. Neider, J.-R. Gaglione, I. Gavran, U. Topcu, B. Wu, and Z. Xu. Advice-guided reinforcement learning in a non-Markovian environment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9073–9080, 2021.

[17] N. Palmer and P. W. Goldberg. PAC-learnability of probabilistic deterministic finite state automata in terms of variation distance. *Theoretical Comput. Sci.*, 387(1):18–31, 2007.

[18] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[19] D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. *J. Comput. Syst. Sci.*, 56(2):133–152, 1998.

[20] A. Ronca and G. De Giacomo. Efficient PAC reinforcement learning in regular decision processes. In Z.-H. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2026–2032. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/279. URL https://doi.org/10.24963/ijcai.2021/279. Main Track.

[21] A. Ronca, G. Paludo Licks, and G. De Giacomo. Markov abstractions for PAC reinforcement learning in non-Markov decision processes. In L. D. Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 3408–3415. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/473. URL https://doi.org/10.24963/ijcai.2022/473. Main Track.

[22] T. Shahar and R. I. Brafman. Reinforcement learning in RDPs by combining deep RL with automata learning. In *ECAI 2023*, pages 2097–2104. IOS Press, 2023.

[23] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[24] S. Sokota, C. Y. Ho, Z. Ahmad, and J. Z. Kolter. Monte Carlo tree search with iteratively refining state abstractions. *Advances in Neural Information Processing Systems*, 34:18698–18709, 2021.

[25] R. S. Sutton and A. G. Barto. *Monte Carlo Methods*, volume 2, pages 91–117. MIT Press, 2018.

[26] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[27] L. Xu, J. Hurtado-Grueso, D. Jeurissen, D. P. Liebana, and A. Dockhorn. Elastic Monte Carlo tree search with state abstraction for strategy game playing. In *2022 IEEE Conference on Games (CoG)*, pages 369–376. IEEE, 2022.