Learning and Solving Regular Decision Processes

Eden Abadi and **Ronen I. Brafman**
Department of Computer Science, Ben Gurion University, Israel
abadied@post.bgu.ac.il, brafman@cs.bgu.ac.il

Abstract

Regular Decision Processes (RDPs) are a recently introduced model that extends MDPs with non-Markovian dynamics and rewards. The non-Markovian behavior is restricted to depend on regular properties of the history. These can be specified using regular expressions or formulas in linear dynamic logic over finite traces. Fully specified RDPs can be solved by compiling them into an appropriate MDP. Learning RDPs from data is a challenging problem that has yet to be addressed, on which we focus in this paper. Our approach rests on a new representation for RDPs using Mealy Machines that emit a distribution and an expected reward for each state-action pair. Building on this representation, we combine automata learning techniques with history clustering to learn such a Mealy Machine and solve it by adapting MCTS to it. We empirically evaluate this approach, demonstrating its feasibility.

1 Introduction

In the emerging area of personal health tracking, one records one's pulse, blood pressure, glucose levels, activity levels, nutritional information and much more, in an attempt to learn how to improve one's physical and mental health. In this domain, the state of many variables of interest and the effects of various actions are most likely *not* Markovian functions of the value of the most recently measured variables. Hence, applying standard, MDP-based, RL algorithms [Sutton and Barto, 1998] to a state model consisting of the value of these observed variables will most likely lead to sub-optimal behavior.

Motivated by such application domains, [Brafman and De Giacomo, 2019] introduced *Regular Decision Processes (RDPs)*, a non-Markovian extension of MDPs that does not require knowing or hypothesizing a hidden state. An RDP is a fully observable, non-Markovian model in which the next state and reward are a stochastic function of the *entire* history of the system. This dependence on the past is restricted to regular functions, only. That is, the next-state distribution and reward depends on which *regular* expression the history satisfies.

An RDP can be transformed into an MDP by extending the state of the RDP with variables that track the satisfaction of the regular expression governing its dynamics. To learn an RDP, we need to learn these regular expressions. For example, in the context of personal health-tracking, one might learn which sequences of activities and measurements make a state of hyperglycaemia, hypoglycaemia, or depression, likely, and consequently, adapt behavior policies that prevent them.

An optimal policy for an RDP is a mapping from regular properties of history to actions. Thus, it provides users with clear, understandable guidelines, based on *observable* properties of the world, in contrast to, e.g., arbitrary hidden states in a learned POMDP, or unclear features in a neural network.

This paper makes two contributions to the emerging theory of RDPs. Our first contribution is the use of a deterministic Mealy Machine to specify the RDP. For each state and action, this Mealy Machine emits as its output, a class label. This class label is associated with a distribution over the underlying system states and a reward signal. This idea extends the use of Mealy Machines to specify non-Markovian rewards, introduced recently by [Camacho *et al.*, 2019], to RDPs. Our second, and main contribution is to use this idea to formulate the first algorithm for learning RDPs from data, and to evaluate it on two non-Markovian domains. Our algorithm identifies, through exploration, histories that have similar dynamics based on their empirical distributions. Then, it learns a Mealy Machine that outputs, for each history, an appropriate label. Finally, we solve the RDP represented by this Mealy Machine to obtain an optimal policy. Our algorithm was implemented and tested on two domains modelled as RDPs, demonstrating its ability to learn RDPs from observable data and to generate a near-optimal policy for these models.

2 Background

We assume familiarity with MDPs, recalling basic notation only. We briefly discuss NMDPS, RDPs, and Mealy Machine.

2.1 MDP and NMDPs

A Markov Decision Process (MDP) is a tuple $M = \langle S, A, Tr, R, s_0 \rangle$. S is the set of states, A a set of actions, $Tr : S \times A \rightarrow \Pi(S)$ is the transition function that returns for every state s and action a the distribution over the next state. $R : S \times A \rightarrow \mathcal{R}$ is the reward function that returns the real valued reward received by the agent after performing action a at state s, and $s_0 \in S$ is the initial state.

A solution to an MDP is a *policy* $\rho : S \rightarrow A$ that maps each state to an action. The *value* of ρ at state s, $v^\rho(s)$, is the

expected sum of discounted rewards when starting at state s and selecting actions based on ρ. An *optimal* policy, denoted ρ^*, maximizes the expected sum of discounted rewards for every starting state $s \in S$. Such a policy always exists for infinite horizon discounted reward problems.

A non-Markovian Decision Process (NMDP) is defined identically to an MDP, except that the domains of Tr and R are finite histories (=sequences) of states and actions instead of single states. For convenience and simpler notation, we will assume that the last action executed is *part* of the current state (with a special null action for the initial state), allowing us to represent histories as sequences of states. However, where clarity requires, we will explicitly mention the actions, too. Consequently, in an NMDP, $Tr : S^+ \times A \times S \to \Pi(S)$ and $R : S^+ \times A \to \mathbb{R}$. With this dependence on the history, to act optimally, a policy ρ must, in general, take the form: $\rho : S^+ \to A$. Here, ρ is a partial function defined on every sequence $h \in S^+$ reachable from s_0 under ρ. We define reachability under ρ inductively: s_0 is reachable; if $h \in S^+$ is reachable and $Tr(h, a, s) > 0$ then $h \cdot s$ is reachable.

The value of a history $(s_0, s_1, ..., s_n)$ (also often called a *trace*) is its discounted sum of rewards: $v(s_0 s_1, .., s_n) = \sum_{i=0}^{n} \gamma^n R(s_0, .., s_i)$. Because we assume the reward value is lower and upper bounded and $0 < \gamma < 1$, this discounted sum is always finite and bounded from above and below.

2.2 RDPs

A Regular Decision Processes (RDP) [Brafman and De Giacomo, 2019] is an NMDP in which the next-state distribution and reward depend on which one of a finite set of regular expressions is satisfied by the current history. It is also a *factored* model, i.e., each state is an assignment to a set of state variables (assumed in this paper to be Boolean, for convenience).

Histories are essentially strings over an alphabet of states and actions. Sets of histories are languages over this alphabet. We could use Regular Expressions (RE), an intuitive and much used formalism, to specify languages. However, REs lack the logical structure needed to exploit finer-grained properties of the internal assignments of a state, and do not efficiently support various useful operations on sets of strings. Instead, RDPs use Linear Dynamic Logic on finite traces (LDL_f) [De Giacomo and Vardi, 2013] to specify sets of histories. LDL_f has the same expressive power as RE, allows us to refer to properties of states, and supports simple specification of conjunction, disjunction, and negation.

In this paper we do not make explicit use of the structure, syntax or operations on LDL_f formulas, so readers unfamiliar with them can simply replace each mention of them by RE. For more details, see [Brafman and De Giacomo, 2019]. However, it is important to know that for each LDL_f formula φ, there exists a finite-state automaton M_φ over the alphabet of RDP states that accepts a history of states IFF this history satisfies φ, and that this formula can be effectively constructed [Baier *et al.*, 2008; De Giacomo and Vardi, 2013].

Formally, an RDP is a tuple $M_L = \langle P, A, S, Tr_L, R_L, s_0 \rangle$. P is a set of primitive propositions inducing a state-space S with s_0 as the initial state. A is the set of actions. Tr_L, the transition function, is represented by a finite set T of quadruples of the form: $(\varphi, a, P', \pi(P'))$. φ is an LDL_f formula over P, $a \in A$, $P' \subseteq P$ is the set of propositions affected by a when φ holds, and $\pi(P')$ is a joint-distribution over P' describing its post-action distribution.

Intuitively, if the current history satisfies the formula φ, then action a can only affect the propositions in P', and the distribution over the possible next-state values of these propositions in specified by $\pi(P')$. Formally, if $\{(\varphi_i, a, P'_i, \pi_i(P')| i \in I_a\}$ are all quadruples for a, we require the φ_i's to be mutually exclusive ($\varphi_i \wedge \varphi_j \equiv false$ if $i \neq j$) and exhaustive ($\bigvee_i \varphi_i \equiv true$), so that the transition model is well defined for all histories. Letting $s|_{P'}$ denote s' projected to P', Tr_L is defined as follows: $Tr_L((s_0, a_0, ..., s_k), a, s') = \pi(s'|_{P'})$ if quadruple $(\varphi, a, P', \pi(P'))$ is the (single) one s.t. $s_1, ..., s_k \models \varphi$ and s_k and s' agree on all variables in $P \setminus P'$.

Finally, R_L, the reward function, is specified by a finite set R of pairs (φ, r), where φ is an LDL_f formula over P, and $r \in \mathbb{R}$ is a real-valued reward. Given a trace $s_0, ..., s_k$, the agent receives the reward: $R_L(s_0, ..., s_k) = \sum_{(\varphi,r) \in R,\, s0,...,s_k \models \varphi} r$. By definition R_L is bounded above and below.

Example 1. *We model a 2-armed, Non-Markovian Multi-Armed Bandit (NM-MAB) using an RDP. In standard MAB there is a single state, and the reward depends (possibly stochastically) on the choice of action. Our NM-MAB makes the probability of receiving a (fixed-size) reward dependent on the entire history of actions. It is a two-state RDP, where the state indicates whether a reward was received or not in the last interaction. Let* $\pi = [0.9, 0.2]$ *be a vector that assigns the probability of winning the reward for each action. This probability shifts right (i.e.,* $+1$ *mod* 2*) every time the agent receives a reward. For example, if the agent won three times in the past, then the probability of winning with action 1 is 0.2.*

The RDP $M_L = \langle P, A, S, Tr_L, R_L, s_0 \rangle$ *is defined as follows:* $P = \{w(on)\}$, $S = \{w, \bar{w}\}$, *the initial state* $s_0 = \bar{w}$, *the actions are* $A = \{a_0, a_1\}$. Tr_L, *the transition function, is represented by quadruples* $T = \{(\varphi, a_0, w, \pi_0), (\neg\varphi, a_0, w, \pi_1), (\varphi, a_1, w, \pi_1),$ $(\neg\varphi, a_1, w_1, \pi_0)\}$, *where* $\varphi = \langle (\bar{w}^*; w; \bar{w}^*; w; \bar{w}^*)^* \rangle end$ *and* $\pi_0(w) = 0.9, \pi_0(\bar{w}) = 0.1, \pi_1(w) = 0.2, \pi_1(\bar{w}) = 0.8$. $R_L = \{(\langle true; w \rangle end, r_{won}), (\langle true; \bar{w} \rangle end, r_{lost})\}$

2.3 Mealy-Machine

A (deterministic) Mealy Machine (MM) is a deterministic finite-state transducer whose output value is determined by its current state and the current input. Formally, it is a tuple $M = \langle Q, q_0, \Sigma, \Lambda, T, G \rangle$. Q is the finite set of states, q_0 is the initial state. Σ is the input alphabet and Λ is the output alphabet. The deterministic transition function $T : Q \times \Sigma \to Q$ maps a state and an input symbol to the corresponding next state. The deterministic output function $G : Q \times \Sigma \to \Lambda$ maps a state and an input symbol to the corresponding output symbol. On each step, the machine consumes an input symbol $\sigma \in \Sigma$, transitions from the state $q \in Q$ it started the step in to state $T(q, \sigma) \in Q$, and outputs the symbol $G(q, \sigma) \in \Lambda$.

3 Representing and Solving RDPs

An RDP associates with each history h and action a a distribution over next states (the transition function) and a reward.

Building on the idea of using Mealy Machines to specify non-Markovian rewards [Camacho *et al.*, 2019], our key observation is that the information in the RDP model can also be specified using a Mealy Machine. Below, we show how to build an equivalent MM given an RDP and vice versa. Later, we will exploit this representational equivalence to learn an RDP by learning its equivalent MM.

Suppose we are given an RDP $M_L = \langle P, A, S, Tr_L, R_L, s_0\rangle$ such that $\varphi_1, \ldots, \varphi_n$ are the LDL_f formulas that specify Tr_L and R_L. Each such formula is associated with an action (e.g., φ_i appears in some quadruple $(\varphi_i, a, P', \pi(P'))$). As noted earlier, for each formula φ, there exists an automaton M_φ that accepts a history h IFF h satisfies φ. We use a non-standard construction where the automaton associates the accepting status with transitions, rather than states, and its input alphabet is $S \times A$, the product of the set of RDP states and RDP actions. Its initial state corresponds to the empty history, and each transition extend this history with a state and the following action. Thus, each state represents a history that ended in some action. (The standard construction's input language is $A\times S$, reflecting the last action and state, but the two are interchangeable. The same holds for accepting states vs. transitions.)

Let M_i denote the automaton tracking φ_i, and define $M = \otimes_{i=1}^{n} M_i$ to be the product automaton. By definition, for every history $h \cdot s$ and action a, exactly one formula is satisfied. In the automaton, this history is represented by the edge from the state reached with history h that is labeled by (s, a). Hence, given any state $q = (q_1, \ldots, q_n)$ of M and every transition (s, a), exactly one automaton M_k enters an accepting transition. In addition, any number of automata tracking reward formulas may accept on this transition in M.

The Mealy Machine $M_{Me} = \langle Q, q_0, \Sigma, \Lambda, T, G\rangle$ that describes an RDP is obtained by augmenting the above M with an output function G defined as follows: Let $q = (q_1, \ldots, q_n)$ be a state of M, and let (s, a) be the input symbol. Let M_k be the unique automaton tracking a transition formula that accepts on input (s, a). Let $(\varphi_k, a, P', \pi(P'))$ be the quadruple associated with φ_k and a in Tr_L. Let r be the sum of rewards associated with all the reward formula tracking automata that accept on input (s, a) from their current state in q. Define $G(q, (s, a)) = ((P', \pi(P')), r)$, i.e., $G(q, (s, a))$ returns the propositions and distribution associated with φ_k and reward r.

The converse also holds. Given $M_{Me} = \langle Q, q_0, \Sigma, \Lambda, T, G\rangle$ where $\Sigma = S \times A$ and Λ returns pairs of the form $((P', \pi(P')), r)$, we can (re)construct the RDP. Σ contains the information on S and A. Let q be the state M_{Me} reaches on input (=history ending with an action) h, then $G(q, (s, a))$ provides us with the information on the transition and reward functions given history $h \cdot s \cdot a$.

To solve an RDP $M_L = \langle P, A, S, Tr_L, R_L, s_0\rangle$, we can transform it into an MDP M_{MDP} as follows [Brafman and De Giacomo, 2019]: Let $M_{Me} = \langle Q, q_0, \Sigma, \Lambda, T, G\rangle$ be the MM representation of M_L. Define $M_{MDP} = \langle Q \times S, A, Tr, R\rangle$, where $Tr((q, s), a, (q', s')) = \pi(s'|_{P'}) \times \delta_{q'=T(q,(s,a))}$ and $R((q, s), a) = r$. Here, $G(q, (s, a)) = ((P', \pi(P')), r)$, s and s' must not differ in the value of the propositions not in P', and δ is the Dirac delta. In words, we take the product of M_{Me} with the original state space S of the RDP. Transitions in the MM state component q are deterministic. Transitions in the RDP state component s are stochastic and distributed according to the RDP's transition function which is captured by the MM's output. The latter depend on the history but only through the properties $\varphi_1, \ldots, \varphi_n$, whose satisfaction can be inferred from q, s and a. Similarly, the rewards depend on s, a and the relevant properties of the history, captured by q.

M_{MDP} can be solved using standard MDP solution techniques [Puterman, 2005]. We use UCT [Kocsis and Szepesvári, 2006], an MCTS algorithm, because MCTS can be applied to RDPs without generating M_{MDP} explicitly. We maintain the current RDP state, i.e., the most recent set of observations, and the state q of the Mealy Machine. From q, for each action and current RDP state, we can obtain as output the information needed to sample the next set of observations and rewards. The new observations replace the old ones, and are used (with the action) to update the Mealy Machine. The choice of which action to apply follows the standard UCB1 criterion [Auer *et al.*, 2002] $a = arg\max_a Q((q, s), a) + c \cdot \sqrt{\frac{\log n((q,s))}{n((q,s),a)}}$.

Example 2. *The minimal MM representing the RDP in Example 1 is defined as follows:* $M = \langle Q, q_0, \Sigma, \Lambda, T, G\rangle$. $Q = \{q_0, q_1\}$, q_0 *is the initial state.* $\Sigma = \{w{\cdot}a_0, w{\cdot}a_1, \bar{w}{\cdot}a_0, \bar{w}{\cdot}a_1\}$, $\Lambda = \{\pi_0, \pi_1\}$. $T(q_0, w{\cdot}a_i) = T(q_1, \bar{w}{\cdot}a_i) = q_1, T(q_1, w{\cdot}a_i) = T(q_0, \bar{w} \cdot a_i) = q_0$. $G(q_1, \bar{w} \cdot a_j) = G(q_0, w \cdot a_j) = \pi_{1-j}, G(q_0, \bar{w} \cdot a_j) = G(q_1, w \cdot a_j) = \pi_j$.

4 Learning RDPs

Most MDP learning algorithms rely on the Markov assumption and full observability of the state for their correctness. Such algorithms are not suitable for learning non-Markovian models, such as an RDP.[1] In this section, we propose to exploit the alternative, Mealy Machine representation of RDPs and use Mealy Machine learning algorithms to learn the RDP model. Then, we use MCTS to generate an optimal policy for the learned model. Thus, our algorithm can be characterized as a Model-based RL algorithm for non-Markovian domains.

The high-level pseudo-code of our algorithm, S3M (Sample, Merge, Mealy Machine), is given in Algorithm 1. We assume it has access to the RDP's state space (since the RDP state is observable) and can repeatedly sample sequences from the initial state. Its final output is (a factored) MDP model which is used to compute the policy.

The algorithm starts by generating sample trajectories (L. 3). For the purpose of trajectory sampling, it initially assumes that the RDP is Markovian and hence that its state space is identical to that of the RDP (L. 1). It then clusters histories with similar next-state distributions and uses the next-state distribution of the cluster as its transition model (L. 4). Next, a Mealy Machine learning algorithm is invoked (L. 5). This algorithm expects input of the form *(input sequence, output)*,

[1]Deterministic MDPs are an important exception in which, instead of the state, we observe a deterministic function of it, called its *label*, s.t. given any state and action, the labels of all next states are different. RDPs and DMDPs are different extension of MDPs with limited partial observability, s.t. some domains can be captured in both models. Algorithms for learning DMDPs [Mao *et al.*, 2016; Tappler *et al.*, 2019] share the use of variants of automata-learning algorithms that we use here to learn RDPs.

Algorithm 1 Sample Merge Mealy Model (S3M)

Input: $domain$
Output: M

1: Initialize state space RDP state space S_{RDP}
2: **repeat**
3: H=Sample Domain(S_{RDP})
4: C=Cluster(H)
5: *Me* = *mealy_generator*(C)
6: Set state space to $S_{RDP} \times S_{Me}$
7: **until** Max_Iterations
8: **return** MDP with state space $S_{RDP} \times S_{Me}$

where *output* can be the output symbol obtained after reading the entire input sequence. In our case, the input takes the form (π, α), where π is a trace (=history) and α is a distribution over the next observation (RDP state) and reward. We now update the state-space of the MDP to be the product of the RDP state space and that of the learned MM, and repeat the process. The next subsections describe each step in more detail.

4.1 Sampling

To learn the RDP model we need to generate sample traces. We considered two methods for doing this. One is purely exploratory and does not attempt to exploit during the learning process, while the other does.

The *Pure Exploration* approach uses a stochastic policy that is biased towards actions that were sampled less. More specifically, for every $a \in A$ and $s \in S_{RDP}$, where S_{RDP} is the RDP state space, define:

$$P(a|s) = \frac{f(a,s)}{\sum_a f(a,s)} \text{ where } f(a,s) = 1 - \frac{n(a,s)}{\sum_a n(a,s)} \quad (1)$$

Here, $n(a,s)$ stands for the number of times action a was performed in state s of the RDP. This distribution favours actions that were sampled fewer times in a state.

The *Smart Sampling* approach is essentially Q-learning [Watkins and Dayan, 1992] with some exploration using the above scheme. Specifically, Q values are maintained for each state-action pair, where states are defined and updated as above, starting with a single-state Mealy Machine. $Q(s,a)$ is initialize to 0 for all states and actions, and is updated following each sample of the form s, a, r, s' using $Q(s,a) = Q(s,a) + \alpha(R(s,a) + \gamma \max_{a' \in A} Q(s',a') - Q(s,a))$. With probability $1-\epsilon$, we select the greedy action in state s, and with probability ϵ we sample an action based on the distribution defined in Equation 1.

4.2 Trace Distributions

Next, we associate with every trace encountered a set of propositions and a distribution over their probability. (Note that each prefix of a trace is also a trace.) Let $h = (o_1 a_1, o_2 a_2, ..., o_n, a_n)$ be a given trace. The o_i's denote the fully *o*bservable RDP states. We define $P_{h,(o,a)}$, the set of propositions affected by action a given history $h \cdot o$, to be all propositions $p \in P$ such that there exists a trace $hoao' = (o_1 a_1, o_2 a_2, ..., o_n, a_n, o, a, o')$ in our sample where o and o' differ on the value of p. We expect $P_{h,(o,a)}$ to be small because action effects are usually local. Next, we compute the empirical post-action distribution over $P_{h,(o,a)}$ for history h, RDP state o and action a. That is, the frequency of each assignment to $P_{h,(o,a)}$ in the last RDP state over traces of the form $hoao'$ in our sample.

4.3 Merging Histories and Their Distributions

By modeling a domain as an RDP, our basic assumption is that what dictates the next state distribution of a history is the class of regular expressions it belongs to. Hence, many different histories are likely to display similar behavior because they are instances of the same regular expression. Of course, we do not know what these regular expressions are, and because of the noisy nature of our sample, we cannot expect two histories that belong to the same class to have the same empirical distribution. Moreover, many histories will be sampled rarely, in which case their empirical next-state distribution is likely to significantly differ from the true one. For this reason, we attempt to cluster similar histories together based on their empirical next-state distribution, using KL Divergence [Kullback and Leibler, 1951] as a distance measure. However, we consider merging only histories that affect the *same* set of propositions.

Our goal is to create clusters s.t. each cluster represents a certain distribution and each trace is assigned to a single cluster. We create the clusters bottom-up. First, we create a single cluster for each trace hoa, as described in 4.2, with weight w denoting the number of samples used to create it. Then, for every two clusters with distributions P_1 and P_2 affecting the same propositions with weights $w_1 \geq w_2 \geq$ min_samples: we merge them if (1) the support of P_2 contains the support of P_1, *and* (2) $D_{KL}(P_1 \| P_2) \leq \epsilon$. Condition 1 is required for D_{KL} to be well defined. The new cluster's weight is $w = w_1 + w_2$ and its distribution P is defined to be:

$$P(\cdot) = (1/w)[w_1 \cdot P_1(\cdot) + w_2 \cdot P_2(\cdot)] \quad (2)$$

If a cluster has multiple other clusters with which it can merge, then the one with the smallest KL divergence is selected. We repeat this procedure until no merges are possible.

Next, for each distribution P whose weight < min_samples, we find the distribution Q from the above clusters that affects the same set of propositions such that $D_{KL}(P \| Q)$ is well defined and minimal, and merge the two using Equation 2 to obtain the new distribution. Notice that such a merge implies that the support of P is a subset of the support of Q.

The above is repeated for different values of ϵ, resulting in different models. We now explain how we select the final model. Each model has the form (Π, Tr). Each $\pi \in \Pi$ is a distribution over the set of assignments to some subset P_π of the RDP's set of propositions, with one such distribution associated with each cluster. $Tr : H \to \Pi$ maps each history in the sample to the distribution associated with its cluster.

We compare the models using the following loss function:

$$loss = -\sum_{h \in H} log(P(h|Tr(h)) + \lambda \cdot log(\sum_{\pi \in \ Pi} |supp(\pi)|) \quad (3)$$

$$P(h|Tr(h)) = \prod_{i=1}^{n} P(o_i | o_1 a_1 \ldots o_{i-1} a_{i-1}; Tr(h)) \quad (4)$$

where $|supp(\pi)|$ is the size of the support of distribution π. Thus, our loss function is the log-likelihood of the data with a regulizer that penalizes models with many clusters, and models with "mega"-clusters with large support.

4.4 Generate a Mealy Machine and an MDP

We now have data mapping each encountered history to its cluster index (with which a distribution is associated). We use the *flexfringe* algorithm [Verwer and Hammerschmidt, 2017] to learn a Mealy Machine from this data The result is a MM, $M_{Me} = \langle S_{Me}, s_{0_{Me}}, \Sigma, \Lambda, T, G\rangle$ representing the RDP. From this MM we generate an MDP $M = \langle S_{RDP} \times S_{Me}, A, Tr; R; (s_0, s_{0_{Me}})\rangle$, as explained in Section 3. The optimal policy for this MDP is optimal for the RDP – associating actions with regular functions of the RDP history.

5 Empirical Evaluation

As this is the first paper to explore learning and solving RDPs, we focus on evaluating our algorithm's ability to learn good policies and to scale-up. For Mealy Machine learning we used the EDSM implementation from the FlexFringe library [Verwer and Hammerschmidt, 2017]. To solve the learned RDPs we use UCT [Kocsis and Szepesvári, 2006], extended to RDPs.

5.1 The Domains

We test the two variants of our algorithm on two new RDP domains: NM-MAB and Rotating Maze. As a baseline, we use $R_{\max}$ – a model-based MDP learning algorithm [Brafman and Tennenholtz, 2002], which essentially assumes (wrongly) that the RDP transitions are Markovian.

Multi-Armed Bandit Domain

We described our NM-MAB domain in Example 1. For the experiments we created three MAB-based RDP models:

1. RotatingMAB: The domain defined in Example 1. The win probability for each arm depends on the entire history, but via a regular function.
2. MalfunctionMAB: One of the arms is broken, s.t. after the corresponding action is performed k times, its probability of winning drops to zero for one turn.
3. CheatMAB: There exists a sequence of actions s.t. after performing that sequence, all actions lead to a reward with probability 1 from that point on.

All NM-MAB domains have 2 arms/actions. The winning probabilities of the machines of the RotatingMAB were $(0.9, 0.2)$, for CheatMAB $(0.2, 0.2)$ and for MalfunctionMAB $(0.8, 0.2)$.

Maze Domain

The Maze domain is an $N \times N$ grid, where the agent starts in a fixed position. The possible actions are *up/down/left/right*. They succeed 90% of the time, and in the rest 10% the effect is to move in the opposite direction. The goal is to get to the designated location where a final reward is received. This task would be easy with a normal MDP, however, in this maze, every three actions the agent's orientation changes by $90°$. Thus, the effects of the actions are a regular function of the history. In our experiment we used a 4×4 maze, where the goal is five steps away from the initial position.

5.2 Results

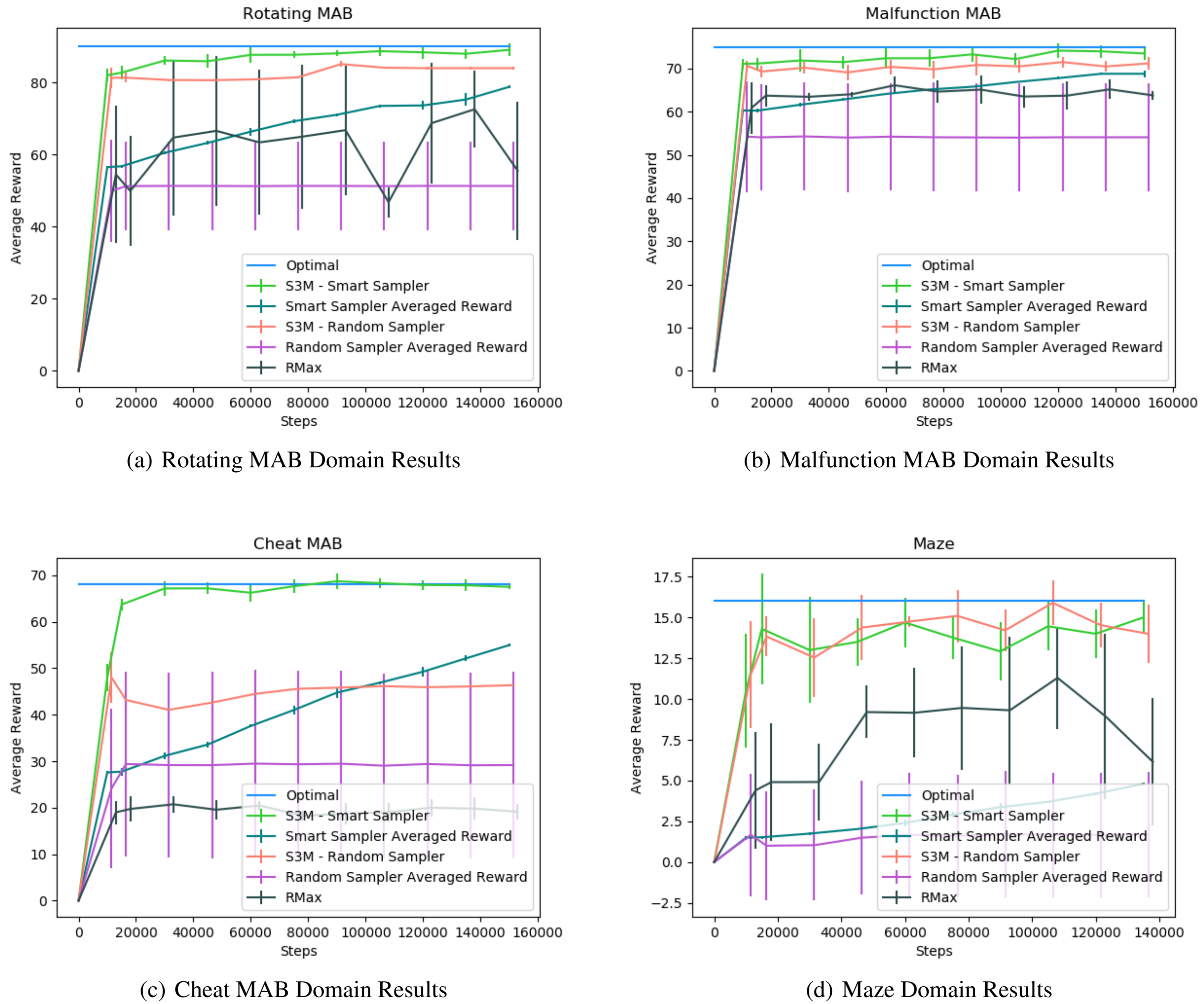
For each domain we used three configurations: (1) *Random Sampler:* S3M with *Pure Exploration* sampling; (2) *Smart Sampler:* S3M with *Smart Sampling*; (3) *RMax:* The model-based RL algorithm R_{Max} [Brafman and Tennenholtz, 2002] that uses the RDP states as its states, used as baseline.

The results are displayed in Figure 1. We show the value of the optimal policy (*Optimal*), the quality of the policies learned by the above three algorithms at each evaluation step, and the average reward accumulated during learning by the first two algorithms. Each experiment was repeated five times. We plot the average over these five repetitions with error bars representing the std. To evaluate the quality of the current policy during the learning process, the optimal policy for the current model was computed online using UCT. 50 trials were conducted using the currently learned Mealy Machine. MAB trials were 10 steps long, and Maze trials were terminated after 15 steps if the goal was not reached. The graph shows the average (over 50 trials) per-step rewards of these policies (averaged over 5 trials), and the average (over 5 trials) accumulated reward obtained as S3M was sampling traces.

Overall, S3M Smart Sampler does quite well, yielding optimal or near optimal policies, as well as accumulated rewards. Random Sampler does reasonably well, too, except on the CheatMAB, but its accumulated reward is typically much worse. R_{max}, which cannot capture the non-Markovian dynamics does much worse. Even in the Maze domain, the difference in its performance is exactly what we would expect when we ignore the non-Markovian behavior – this margin, in that case, is simply smaller than in the MAB domains.

It is interesting to compare the performance of the two sampling approaches. We expected that Smart Sampler will accumulate more reward, as it exploits more, and its exploitation is informed by the learned MM, and hence takes history into account. Indeed, in the MAB domain, we see such behavior: S3M with smart sampling converges to an optimal or near-optimal policy quickly, and the accumulated rewards increase steadily, while Random Sampler does worse. This is especially pronounced in the Cheat MAB domain, which is the most complex domain. We believe the reason for Random Sampler's weak performance in this domain is that too many of the samples are not along the more interesting traces that discover the "cheat" sequence. Therefore, it is more difficult for it to learn a good MM that can exploit it. Surprisingly, unlike in MAB, in the Maze domain there is no significant difference between the two S3M versions. We hypothesize that in Maze, there is more to learn about the general behavior of transitions because there are more states, and Random Sampler generates more diverse samples that provide more accurate statistics on various states. Smart Sampler, on the other hand, does not. Moreover, while the domain has more states, the regular expression that governs the dynamics is relatively simple, and so Random Sampler is still able to learn the corresponding automaton.

Generally speaking, we see a high correlation between the quality of the samples collected by the sampler and the quality of the learned model: when the average reward of the samples is monotonically increasing so does the averaged reward of the policy obtained by MCTS. An open question is what is



(a) Rotating MAB Domain Results

(b) Malfunction MAB Domain Results

(c) Cheat MAB Domain Results

(d) Maze Domain Results

Figure 1: Results

the exact relation: do samples that concentrate along desirable traces yield better Mealy Machines, or is it the case that because we learn a better Mealy Machine, the traces generated by the Smart Sampler have higher rewards (naturally).

Finally, we tried to evaluate the quality of the learned MM, directly. The MM models we learn contain many more states than the minimal MM. To evaluate them, we sampled 1000 histories for each domain, and for every prefix of every history we computed the $L2$ Euclidean distance between the true distribution and the one given by the MM learned after 150K steps. This value is bounded from below by zero and from above by $\sqrt{2}$, for distributions. The results are shown in the Table below. The values shown are the averaged Euclidean distance between the algorithm's generated distribution and the real one. We can see that for MAB domains the error is approximately 0.1. Roughly speaking, this means that, on average, the error for single transition ranges from 7% (when all error is concentrated on two states) to approximately 5.7% (for three states). In the Maze domain, on average, the maximum ranges from 13% to 3.3%. These values, while not perfect, are quite reasonable for estimating a near-optimal policy (as seen in Figure 1).

Domain	L_2	Domain	L_2
Rotating MAB	0.10	Malfunction MAB	0.11
Cheat MAB	0.11	Maze	0.19

Table 1: Model error

6 Summary

We presented the first algorithm for learning RDPs. By viewing the RDP specification as a Mealy Machine, we were able to combine Mealy Machine and RL algorithms to obtain an algorithm for learning RDPs that quickly learns a good Mealy Machine representation in our experiments. Naturally, there is much room for improvement, especially in methods for better sampling and better aggregation of histories.

References

[Auer *et al.*, 2002] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[Baier *et al.*, 2008] Jorge A. Baier, Christian Fritz, Meghyn Bienvenu, and Sheila McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI), Nectar Track*, pages 1509–1512, Chicago, Illinois, USA, July 13-17 2008.

[Brafman and De Giacomo, 2019] Ronen I. Brafman and Giuseppe De Giacomo. Planning for ltlf /ldlf goals in non-markovian fully observable nondeterministic domains. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1602–1608, 2019.

[Brafman and Tennenholtz, 2002] Ronen I. Brafman and Moshe Tennenholtz. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3:213–231, 2002.

[Camacho *et al.*, 2019] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[De Giacomo and Vardi, 2013] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI-13*, pages 854–860, 2013.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 282–293, 2006.

[Kullback and Leibler, 1951] Solomon Kullback and Richard Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[Mao *et al.*, 2016] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Mach. Learn.*, 105(2):255–299, 2016.

[Puterman, 2005] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[Tappler *et al.*, 2019] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. L^*-based learning of markov decision processes. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, pages 651–669, 2019.

[Verwer and Hammerschmidt, 2017] Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 638–642, 2017.

[Watkins and Dayan, 1992] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.