Regular Decision Processes

Reasoning Agents

Tania Sari Bonaventura, Christian Marinoni, Sveva Pepe and Simone Tedeschi

Master's Degree in Artificial Intelligence and Robotics

Sapienza University of Rome

September 2021

Contents

1 Introduction

Regular Decision Processes (RDPs) [Brafman and De Giacomo, 2019] have been recently introduced as 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. However, this dependence on the past is restricted to regular functions. That is, the next state distribution and reward depend 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 the RDP dynamics. Thus, essentially, to learn an RDP, we need to learn these regular expressions.

The works of Abadi and Brafman [Abadi and Brafman, 2020] and Ronca and De Giacomo [Ronca and De Giacomo, 2021] provide two significant contributions to the analysis and solution of RDPs. The first work focuses on using a deterministic Mealy Machine to specify the RDP where, for each state-action pair, a cluster is emitted. Each cluster has an associated distribution over the system states and reward signal. Then, they formulate the first algorithm to learn RDPs from data that merges histories with similar distributions to generate the best model from which the Mealy Machine is produced. Finally, the latter is exploited to provide the distributions needed for running MCTS [Kocsis, Levente and Szepesvári, Csaba, 2006, Silver and Veness, 2010]. Four domains (i.e., RotatingMAB, CheatMAB, MalfunctionMAB, RotatingMaze) are used for the evaluation.

On the other hand, the second work shows an algorithm that learns a probabilistic deterministic finite automaton (PDFA) that represents RDP. Then, it computes an intermediate policy by solving the MDP obtained from the PDFA. From such policy, the final one is derived by composing it with the transition function of the PDFA. The whole process, repeated multiple times, ensures to reach an ϵ-optimal policy in polynomial time.

Our project consists of the implementation of the algorithm provided by Abadi and Brafman [2020] without considering propositions while trying to reproduce the same results as the authors.

The following project report is organized as follows. Section 3 provides a theoretical description of the algorithm, and the relative steps; Section 4 describes the MCTS algorithm; Section 5 describe the parameters used in the different domains to evaluate the algorithm, respectively; Section 6 presents the obtained results; Section 7 examines the pros and cons of the two suggested algorithms based on the results obtained; and finally, Section 9 summarizes the main achievements of the project.

2 Preliminaries

2.1 MDP

A *Markov Decision Process* is a tuple that describes the interaction between the agent and the controlled environment. It is based on the Markov assumption that the outcome of an action is solely determined by the state in which it is performed. An MDP is defined as

$$< A, S, R, T, \gamma > \tag{1}$$

where:

- S is the state space, $s \in S$
- A is the set of actions, $a \in A$;
- $\mathbf{T} : S \times A \times S \rightarrow [0, 1]$ is a *transition function* which defines a probability distribution $\mathbf{T}(\cdot|s, a)$ over the last state and action a;
- $\mathbf{R}$ is the reward function $\mathbf{R} : S \times A \times S \rightarrow \mathbb{R}$, so the reward received when reaching a state s' after applying an action a on a state s.

- $\gamma \in (0, 1)$ is the discount factor, which defines the weight that future rewards have on the present. If $(\gamma = 0)$, that means the agent is short-sighted; in other words, it only cares about the first reward. If $(\gamma = 1)$, that means the agent is far-sighted, meaning it is interested in all future rewards.

2.2 NMDP

A Non-Markovian Decision Process is a tuple $< A, S, R, T, \gamma >$ similar to the MDP one, but that does not verify the Markov assumption, meaning that the outcome of the current action depends on the whole history (i.e., the sequence of all previous states). Thus, the transition function can be defined as $\mathbf{T} : S^* \times A \times S \rightarrow [0, 1]$ and the reward function as $\mathbf{R} : S^* \times A \times S \rightarrow \mathbb{R}$. Differently from the MDP, given two histories h_1 and h_2 $\mathbf{T}(\cdot|h_1 s, a) \neq \mathbf{T}(\cdot|h_2 s, a)$.

The transition and reward functions may be coupled to form the dynamics function $D : S^* A \times S \times R \rightarrow [0, 1]$, which provides a probability distribution $D(\cdot|h, a)$ over S R for each $h \in S^*$ and every $a \in A$. Namely, $D(s, r|h, a)$ is $T(s|h, a)$ if $r = R(h, a, s)$, and zero otherwise.

2.3 RDP

A Regular Decision Process is an $NMDP =< A, S, R, T, R, >$ with finite transducers representing the transition and reward functions. There is a finite transducer that outputs the function $T_h : A \times S \rightarrow [0, 1]$ induced by T when its first argument is h, and a finite transducer that outputs the function $R_h : A \times S \times R \rightarrow [0, 1]$ induced by R when its first argument is h. It is worth noting that the cross-product of such transducers gives a finite transducer for RDP's dynamics function D.

2.4 Mealy Machine

In this project, we plan to establish a (deterministic) Mealy Machine (MM), a finite-state deterministic transducer whose output value is determined by its present state and the current input, capable of modeling the dynamic function D. It is formally a tuple $M =< S, s_0, \Sigma, \Lambda, T, G >$. The finite set of states is denoted by S, and s_0 denotes the starting state. The input alphabet is Σ, and the output alphabet is Λ. $T : S \times \Sigma \rightarrow S$ is a deterministic transition function that maps a state and an input symbol to the next state. A state and an input symbol are mapped to the appropriate output symbol by the deterministic output function $G : S \times \Sigma \rightarrow \Lambda$. On each step, the machine consumes one input symbol, transitions from the state $s \in S$ in which it began the step to state $T(s, \sigma) \in S$, and produces the symbol $G(s, \sigma) \in \Lambda$.

3 Algorithm

The Abadi and Branfam algorithm, called *Sample Merge Mealy Model* (S3M), follows these steps:

- **Sampling**: it generates traces from the RDP by interacting with the environment (Section 3.1).
- **Base distribution**: it associates a distribution over the states to each history derived from the traces (Section 3.2).
- **Merger**: it merges the distributions of multiple histories based on the Kullback-Leibler divergence (Section 3.3).
- **Loss function**: it allows to select the best model, penalizing the ones with the few merges and too high support (Section 3.4).
- **Mealy Machine**: it generates the Mealy Machine from the computed model (Section 3.5).

For completeness, we report the S3M pseudo-code in Figure 1 as provided by the authors.

Algorithm 1 Sample Merge Mealy Model (S3M)

```
Input: $domain$
Parameter: $min\_samples$
Output: $M$
Initialize state space of $M$ to RDP state space $S_{RDP}$
repeat
    Set $S = sample(domain)$.
    Set $Tr = base\_distributions(S, min\_samples)$
    $best\_loss = \infty$
    for $\epsilon$ $in$ $possible\_epsilons$ do
        $Tr' = merger(Tr, \epsilon)$
        $loss = calc\_loss(Tr', S)$
        if $loss < best\_loss$ then
            $Tr = Tr'$
            $best\_loss = loss$
        end if
    end for
    $Me = mealy\_generator$
    Set $M = \langle P, A, S_{RDP} \times S_{Me}, Tr, R, (s_{0_M}, s_{0_{Me}})\rangle$
until Max_Iterations
return $M$
```

Figure 1: Pseudocode of the S3M algorithm

3.1 Sampling

To learn a Mealy Machine for the RDP we have to generate sample traces. We can produce these traces by either interacting with the environment through a purely exploratory approach or selecting actions based on Q-learning.

The *Pure Exploration* approach samples actions according to a distribution that is iteratively adjusted to encourage the exploration towards unseen regions of the environment. More formally, for every action $a \in A$ and state s in the RDP state space $s \in S_{RDP}$, an action is sampled according to the distribution

$$P(a|s) = \frac{f(a,s)}{\sum_a f(a,s)} \text{ with } f(a,s) = 1 - \frac{n(a,s)}{\sum_a n(a,s)} \tag{2}$$

where $n(a,s)$ represents the number of times action a has been performed in the state s.

Smart Sampling is an ϵ-greedy approach which selects the greedy action through Q-learning with probability 1-ϵ, and performs pure exploration as explained above with the remaining probability.

3.2 Base distribution

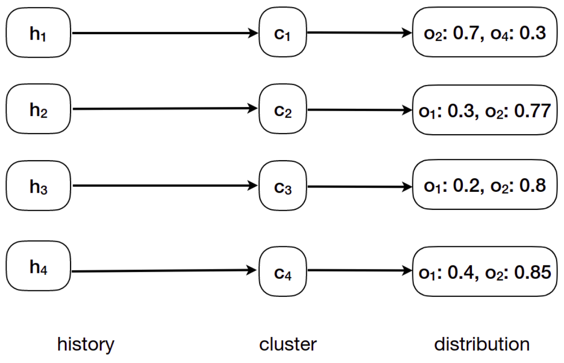
Given a history $h = (o_1a_1o_2a_2...o_na_n)$ derived from the trace ho', the authors define the set of propositions $P_{h,(o,a)}$ affected by action a. Then, they compute the empirical post-action distribution over such propositions for history h and observation o.

As aforementioned in the introduction, our work does not use any proposition. Therefore, given a trace, we just associate to its history a distribution over the observations $P(o|h)$.

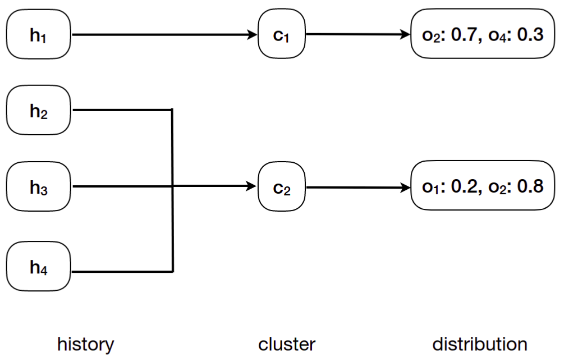
3.3 Merger

In this Section we want to merge histories with similar distributions and same support, i.e., we merge distributions over the same set of observations. For this reason, we attempt to cluster them by exploiting the Kullback-Leibler (KL) divergence as a distance measure.

Each history is, therefore, associated to a cluster which encapsulates its distribution, see Figure 2a. After the merging procedure, several clusters will be combined such that we will obtain different histories that refer to the same cluster, i.e., to the same distribution. Thus, the number of clusters will be lower than the initial one, see Figure 3.3.



(a) Clusters before merging



(b) Clusters after merging

Each cluster has a weight w that corresponds to the number of samples used to create it. Based on the weight, the merging procedure will follow two different paths. In the first one, considering two clusters – with distributions P_1 and P_2 and weights w_1 and w_2, respectively – if $w_1 \geq w_2 \geq min\_samples$[1] and the KL divergence $D_{KL}(P_1, P_2) \leq \epsilon$, they are merged. The resulting cluster will have weight $w = w_1 + w_2$ and the following associated distribution:

$$P(\cdot) = (1/w)[w_1 \cdot P_1(\cdot) + w_2 \cdot P_2(\cdot)] \tag{3}$$

If a cluster can be merged with several others, we choose the one with the smallest KL divergence.

On the other hand, once there are no more available merges, we consider the clusters with $w < min\_samples$. We try to incorporate them with those previously merged that have the same distribution support and the smallest KL divergence.

3.4 Loss function

To find the best model, the previous step is repeated for several values of ϵ. To select the model that has the best performance, we compare them with a loss function as follows:

$$loss = -\sum_{h \in H} log(P(h|Tr(h)) + \lambda \cdot log(\sum_{\pi \in \Pi} |supp(\pi)|) \tag{4}$$

$$P(h|Tr(h)) = \prod_{i=1}^{n} P(o_i|o_1 a_1 ... o_{i-1} a_{i-1}; Tr(h)) \tag{5}$$

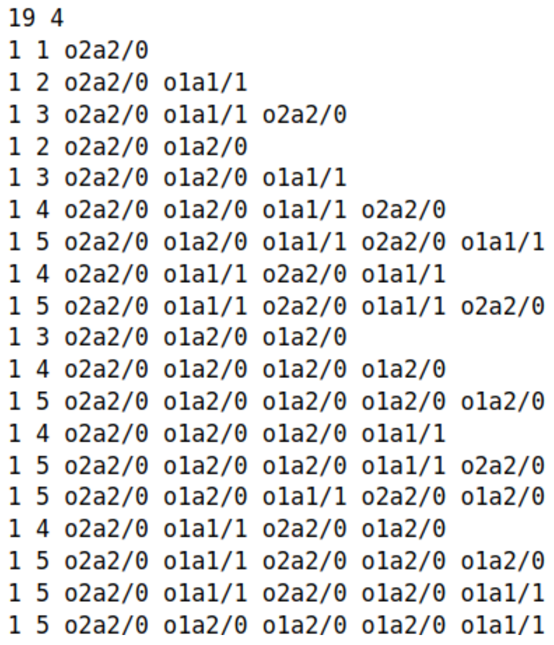
where the first loss term is the likelihood of the histories given the given model $Tr(h)$; while the second term depends on the support of the observations' distributions, $|supp(\pi)|$, that corresponds to the number of observations to which a non-zero probability is associated. The idea is to penalize models with few merges and too high support.

3.5 Mealy Machine

A Mealy Machine is a tuple $M = \langle S, s_0, \Sigma, \Lambda, T, G \rangle$ where S is the finite set of states, s_0 is the initial state, Σ is the input alphabet and Λ is the output alphabet. The transition function $T\colon S \times \Sigma \rightarrow S$ maps pairs of state and input symbol to the corresponding next state. The output function $G\colon S \times \Sigma \rightarrow \Lambda$ maps pairs of state and input symbol to the corresponding output symbol.

[1] $min\_samples$ is a given integer number

We use the *flexfringe* library [Verwer and Hammerschmidt, 2017] to learn the Mealy Machine from our data. Every history of our sample is associated with a cluster index to which it belongs. The following figure shows what we just described:



```
19 4
1 1 o2a2/0
1 2 o2a2/0 o1a1/1
1 3 o2a2/0 o1a1/1 o2a2/0
1 2 o2a2/0 o1a2/0
1 3 o2a2/0 o1a2/0 o1a1/1
1 4 o2a2/0 o1a2/0 o1a1/1 o2a2/0
1 5 o2a2/0 o1a2/0 o1a1/1 o2a2/0 o1a1/1
1 4 o2a2/0 o1a1/1 o2a2/0 o1a1/1
1 5 o2a2/0 o1a1/1 o2a2/0 o1a1/1 o2a2/0
1 3 o2a2/0 o1a2/0 o1a2/0
1 4 o2a2/0 o1a2/0 o1a2/0 o1a2/0
1 5 o2a2/0 o1a2/0 o1a2/0 o1a2/0 o1a2/0
1 4 o2a2/0 o1a2/0 o1a2/0 o1a1/1
1 5 o2a2/0 o1a2/0 o1a2/0 o1a1/1 o2a2/0
1 5 o2a2/0 o1a2/0 o1a1/1 o2a2/0 o1a2/0
1 4 o2a2/0 o1a1/1 o2a2/0 o1a2/0
1 5 o2a2/0 o1a1/1 o2a2/0 o1a2/0 o1a2/0
1 5 o2a2/0 o1a1/1 o2a2/0 o1a2/0 o1a1/1
1 5 o2a2/0 o1a2/0 o1a2/0 o1a2/0 o1a1/1
```

Figure 3: Example of Flexfringe input file.

As we can see from the above Figure, the first line contains the number of traces and the alphabet size, respectively. From the second row on, we have the following structure:

- the first digit represents whether the transducer should accept the sample or not;
- the second digit describes the length of the trace;
- the remaining elements are pairs of the format *input_symbol/output_symbol*, where an input symbol is given by the state-action pair and the output symbol is a cluster index.

By taking the product $S_{RDP} \times S_{Me}$ – where S_{RDP} represents the original state space of the RDP and S_{Me} the finite set of states of the obtained Mealy Machine – we achieve a MDP $M = \langle S_{RDP} \times S_{Me}, A, Tr, R, (s_0, s_{0_{Me}})\rangle$ that can be solved using UCT, an MCTS algorithm.

4 MCTS

Monte Carlo Tree Search (MCTS) uses Monte Carlo simulation to accumulate value estimates to guide towards highly rewarding paths in the search tree. In other words, MCTS pays more attention to nodes that are more promising, so it avoids having to brute force all possibilities which is impractical to do.

At its core, MCTS consists of 4 steps:

- **Selection**: the algorithm keeps selecting the best child nodes until a leaf node of the tree is reached.
- **Expansion**: then, from the leaf node the algorithm randomly picks an unexplored node.
- **Simulation**: after Expansion, the algorithm simulates an entire episode from the selected node by picking a sequence of random actions until the episode ends.

- **Backpropagation**: finally, it evaluates the reached state to figure out if the agent won or not. It traverses upwards to the root and updates the win score of each node based on the (winning or loosing) final state reached by the agent.

To produce sufficiently accurate estimated scores, the just described steps are repeated iteratively as shown in Figure 4.

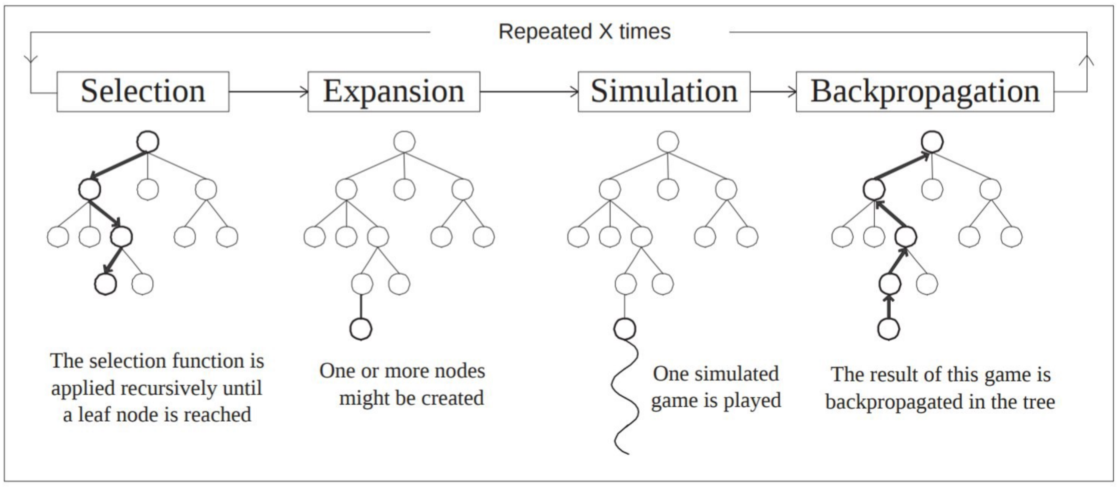

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 4: Steps of the MCTS algorithm

In particular, as done by authors, we use a particular instance of MCTS called *Upper Confidence bounds applied to Trees* (UCT) which uses the UCB1 criterion to select the best child, defined as follows:

$$UCB1_i = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{2 \cdot \log N}{n_i}} \tag{6}$$

where w_i identify the number of wins of node i, n_i is the number of visit of node i, c is the exploration parameter and N is the visits parent.

The first component of the UCB1 formula above corresponds to exploitation, as it is high for moves with high average win ratio. The second component corresponds to exploration, since it is high for moves with few simulations. MCTS estimates will typically be unreliable at the start of a search but converge to more reliable estimates given sufficient time and to perfect estimates given infinite time.

5 Setup Environments

We implemented our code in Python and we relied on the FlexFringe[2] library to generate the Mealy Machine. We evaluated our algorithm on four different domains, belonging to 2 families: Non Markovian Multi-Arm Bandit (MAB) and Rotating Maze.

MAB The Multi-Arm Bandit is the simplest class of Reinforcement Learning (RL) domains. Standard MAB is state-less – hence there are no transition function to learn. However, Abadi and Brafman [Abadi and Brafman, 2020] extends Non-Markovian MAB by making the probability of receiving a (fixed-size) reward – after the execution of one of the n available actions – depend on the entire history of previous actions. When $n = 2$, we essentially get a two-state RDP, where each state tells whether a reward was received or not. They introduced three MAB-based RDP models:

[2]`https://bitbucket.org/chrshmmmr/dfasat/src/master/`

- **RotatingMAB**: each action has an associated probability of winning a reward. When the agent receives a reward, this probability shifts right (i.e., +1 mod n). Hence, the winning probability for each arm depends on the entire history, via a regular function.
- **MalfunctionMAB**: after one action is executed k times, the corresponding arm becomes out of service, i.e., its winning probability lowers to zero for one turn.
- **CheatMAB**: there exists a sequence of actions that, once performed, will guarantee to reach the reward with probability 1, regardless of the chosen action.

We conducted experiments using 2 arms/actions for all of the three variations of the domain. The winning probabilities of the machines were (0.9, 0.2), (0.2, 0.2) and (0.8, 0.2), respectively.

Maze: the Maze domain is an N $\times$ N grid, where the agent starts in a fixed position and needs to reach a designated location to receive a final reward. The agent can do four actions (i.e., *up/down/left/right*) that succeed 90% of the time, while in the rest 10 % it moves in the opposite direction, if feasible. In a normal MDP this task would be quite easy to solve using conventional RL algorithms. However, the effects of the actions are made a regular function of the history by changing the agent's orientation by 90 degrees every three actions. In our experiment we used a 4 $\times$ 4 maze, where the goal is five steps away from the initial position.

The following table shows the configurations parameters that we use to produce the results in Section 6.

Domain	**min_sample**	**max_iterations**	$\boldsymbol{\lambda}$	ϵ	**UCT iterations**	**UCT trials**	**episodes**
RotatingMAB	100	500	100	[0.52, 1.04, 1.56, 2.08]	160000	30	10
MalfunctionMAB	10	100	100	[0.52, 1.04, 1.56, 2.08]	160000	30	10
CheatMAB	5	100	100	[0.52, 1.04, 1.56, 2.08]	160000	30	10
RotatingMaze	3	30	100	[0.52, 1.04, 1.56, 2.08]	140000	30	15

Table 1: Parameters used in the implementation

6 Results

In this Section, we compare two setups for each domain: i) only MCTS ii) S3M algoritm with pure exploration policy + MCTS, so to verify the effectiveness of the algorithm proposed by Abadi and Brafman. The outcomes are depicted in Figure 5. We illustrate the average reward collected by the first two algorithms during learning. Fifty trials were carried out using the currently learned Mealy Machine. MAB trials were ten steps (episode) long, while Maze trials terminated after 15 steps if the agent did not meet the goal. Except for the CheatMAB, Random Sampler performs pretty well, although its cumulative reward is generally significantly lower. A deeper examination of the Maze domain reveals that the performance difference is what we would anticipate if we disregard non-Markovian behaviour — the margin is less than in the MAB domains. In RL, proper exploration is crucial. In RDPs, exploring states is not enough; we also need to collect statistics on histories. It is fascinating to compare the results of the two techniques in this aspect. In all of the graphs, we can see how the S3M method, based solely on pure exploration, contributes more than the only MCTS approach. In particular, we can see how the difference between the two algorithms in the MAB domains is more pronounced, but the results in the MAZE are pretty similar, but this is due to the complexity of the domain itself. In the next section, we will look at why S3M can provide benefits and what we consider to be the most significant factors that allow S3M to outperform the single MCTS.

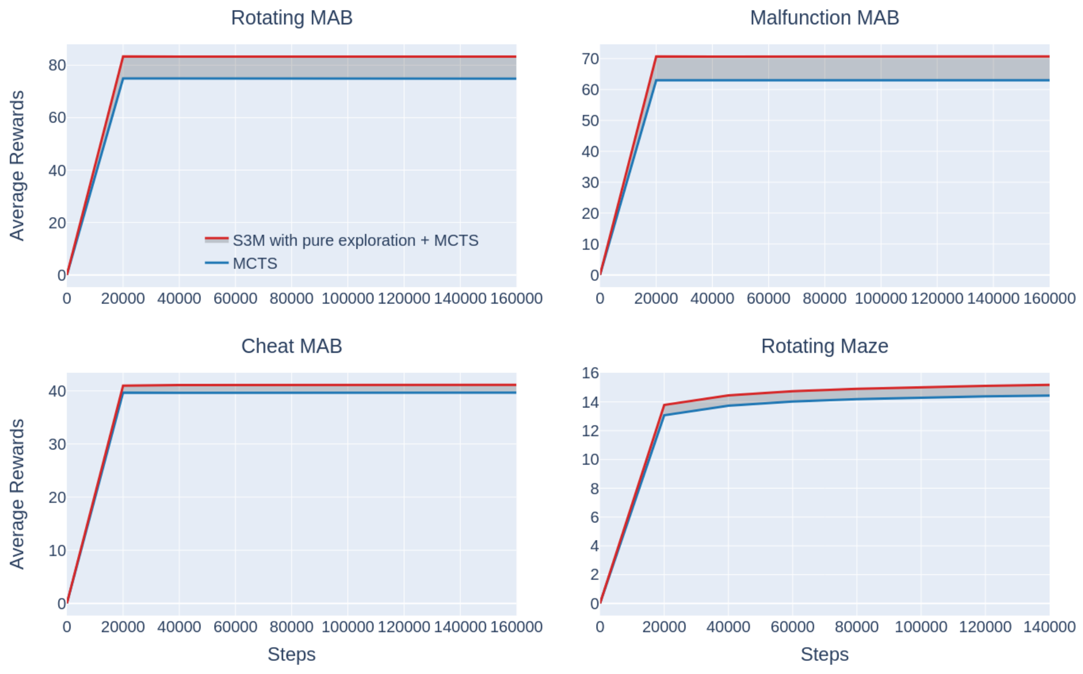


Figure 5: Results

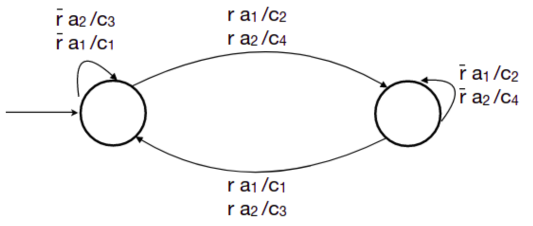
7 Analysis

Our S3M performance analysis mainly focused on the RotatingMAB domain, which is the simplest of those proposed. In actuality, it is defined by only two states - losing (0,) and winning (1,) - for which the agent receives a zero or full reward. A Mealy Machine should theoretically get closer to the ideal with each iteration of S3M, as explained in the Section 3. For this reason, it is crucial to know which Mealy Machine we should get.

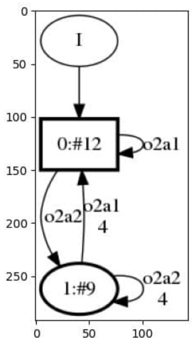
As shown in Figure 6a, it is distinguished by only two states connected by edges, each characterized by an observation, the corresponding action carried out, and the desired output (cluster). $\bar{r}$ denotes the statement "Agent did not receive a reward", whereas r represents the statement "Agent did receive a reward", and the latter implies a movement to the other state.

In RotatingMAB we have four clusters:

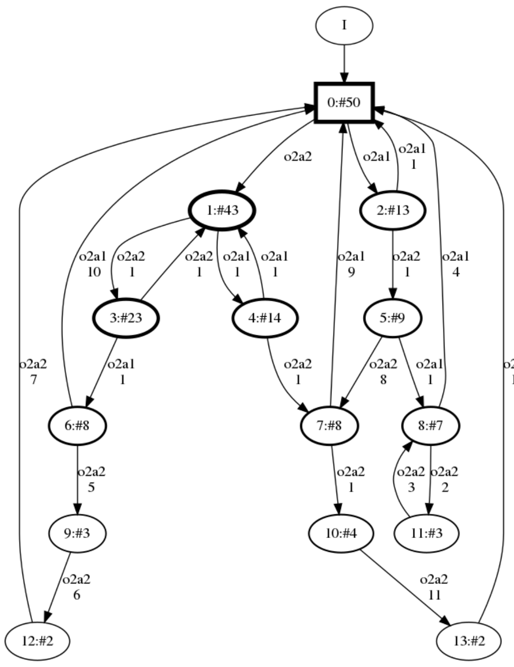
- c_1: the probability of observing a reward is 0.9
- c_2: the probability of observing a reward is 0.1 (the complement of 0.9)
- c_3: the probability of observing a reward is 0.2
- c_4: the probability of observing a reward is 0.8



(a) Expected Mealy Machine in Rotating MAB



(b) One example of the obtained Mealy Machine in Rotating MAB



(c) Second example of the obtained Mealy Machine in Rotating MAB

The algorithm shows rather large variability in the Mealy Machines' structure, mainly due to the merging between clusters that S3M performs. For example, Figure 6c shows a Mealy Machine very different from the ideal one. Contrary to what one might suppose, the contribution of a non-ideal Mealy Machine is generally positive if compared to the use of MCTS alone. Indeed, MCTS learns better if the generated MDP, having as states those obtained from the Cartesian product of the states of the RDP and the Mealy Machine, is used. Figure 8 shows the results obtained with the Mealy Machine displayed in Figure 6b (closer to the ideal one) and those obtained with the Mealy Machine in Figure 6c (furthest from ideal). In both cases, there is an improvement compared to the baseline (MCTS only). However, it remains rather complex to understand - without an analysis of the functioning of Flexfringe - which are the same factors that give rise to this variability.

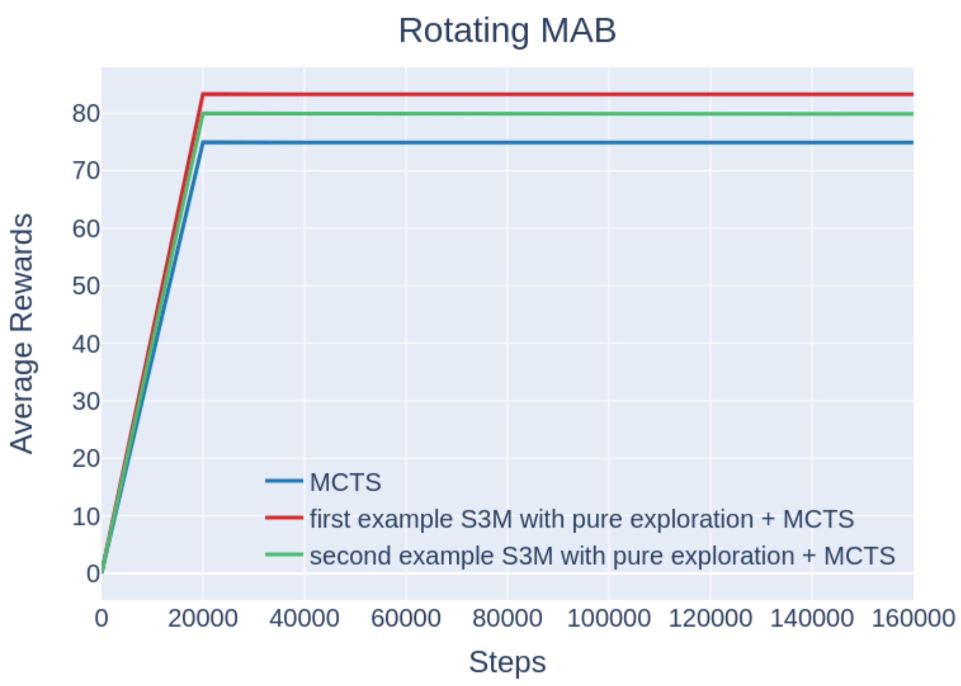


Figure 7: Results all examples

The choice of parameters also appears to have an impact on the generated results. In particular, the chosen ϵ values have a significant impact on the merging results and, as a consequence, on the structure of the mealy machine. The same thing happens with the parameter *min_sample*: larger values will often lead to larger Mealy Machines (with more states). Finally, the *episode's length* is revealed to be an essential parameter in the construction of the mealy machine since shorter episodes produce similar traces and tend to merge more.

8 Complex domains

Before delving into the specifics of our domain, we need to review some key ideas of the PAC-RL algorithm described in Ronca and De Giacomo [2021]'s work.

8.1 Overview of PAC-RL

The authors define reinforcement learning (RL) as the issue of an agent learning an optimum policy for an unknown RDP $\mathcal{P}$ by acting and obtaining observation states and rewards based on $\mathcal{P}$'s transition and reward functions. The agent conducts a series of actions $a_1, ..., a_n$, with each action a_i resulting in an observation state s_i and a reward r_i. The agent has the option to pause and perhaps restart. This procedure produces strings of the type $a_1 s_1 r_1, ..., a_n s_n r_n$, which we refer to as episodes. They contribute to the agent's experience, which serves as the foundation for learning an optimum policy. The authors' objective is to make the agent discover near-optimal policies in the shortest amount of time, without strictly maximizing the reward gathered during learning. As a result, they expect the agent to return policies $\pi_1, \pi_2, ...$ that improve with time.

These policies can be used to define the behaviour of an agent living in such an environment, which will then improve as it gets better policies. The fundamental question thus is how quickly an agent may reach a point from which it exclusively produces favorable policies. The number of steps that an agent requires is taken into account to assess its learning capabilities. In particular, an action step entails completing an action and recording the observation state and the reward; thus, a *step* is simply an action step.

The proposed algorithm is based on the PAC framework, which is relies on the fact that exact learning is not possible. Therefore, it presents two parameters, $\epsilon > 0$ and $\delta \in (0, 1)$, which characterize the required accuracy and confidence of success, respectively. In the RL context, this means searching for policies that are ϵ-optimal with a probability of at least $1 - \delta$. As a result, RL is deemed viable if these policies can be discovered in a number of steps which is polynomial in $1/\epsilon$ and $\ln(1/\delta)$, and in other RDP-related factors.

The parameters used to define an RDP $=< A, S, R, \mathbf{T}, \mathbf{R}, \gamma >$, which are presented below, are the characteristic element of this work:

$$\mathbf{d}_{\mathcal{P}} = (|A|, \frac{1}{1-\gamma}, R_{max}, n, \frac{1}{\rho}, \frac{1}{\mu}, \frac{1}{\eta}) \tag{7}$$

The first four parameters for the dynamics of $\mathcal{P}$ make use of the number of actions $|A|$, the discount factor γ, the maximum reward value R_{max}, and the number of states n of the minimal transducer. The reachability ρ of an RDP then assesses how simple it is to reach states of the dynamics transducer T when performed evenly and at random. When actions are selected uniformly at random, the distinguishability μ of an RDP measures how easy it is to distinguish states of the dynamics transducer T. The degree of determinism, η, indicates how straightforward it is to find transitions. If η is tiny, there are transitions that, although possible, are unlikely to be detected. When the RDP is deterministic, this parameter takes value one.

8.2 Weaknesses of the algorithms

To conduct a comprehensive analysis to develop a complex domain for each of the previously mentioned approaches, we identify positive and negative aspects of each work. In particular, when it comes to the S3M algorithm, we rely heavily on the analysis of the results provided by the authors, with special attention to the Rotating Maze domain. Here, contrarily to the other domains based on the MAB – where there are just few states and actions, and where Smart Sampling proves to be quite effective – the Pure Exploration procedure appears to be more effective than Smart Sampling.

The effectiveness of Pure exploration in the Rotating Maze domain can be attributed to two factors:

1. there are more states to visit and actions to choose from in each state. As a result, in this case, pure exploration succeeds in ensuring a better exploration of the domain by prioritizing the choice of the action selected fewer times so far. This means that the produced traces are more varied and, due to the structure of the algorithm, this allows for a better generated Mealy Machine. Consequently, in this way, one can learn the regular function at the core of that domain, which explains how to relate the stories with the results that the actions must produce;

2. the underlying regular function, in the end, is relatively simple, a more selective and wise approach is not required to learn the automaton.

While it is true that a grid domain like this seems to work better with pure exploration, it is also true that its effectiveness can be saturated by increasing the size of the grid and making the regular function that governs the dynamics more complex. These two conditions could make it difficult to learn useful automata starting from a set of traces which, although very varied, are too general to produce sufficiently informative clusters. Nevertheless, Smart Sampling does not overcome the difficulties stated above since, as implemented by the authors, it is based on a basic Q-learning where the Q-value associated with each action is solely determined by the reward received. The issue is that the algorithm, supposed to have a sufficiently big grid and a not oversized episode length, will not reach the reward every time. Moreover, assuming the use of a domain-dependent antagonist or some moving obstacles, the agent will reach the goal even more rarely. Thus, when doing Q-learning it will be biased in the choice of pretty limited sequences of actions and, even when randomly exploring the environment with the ϵ-greedy procedure, it will still not be able to reach the goal often enough to produce a varied set of good-quality traces and clusters.

We may infer that both Smart Sampling and Pure Exploration have flaws and that the algorithm would perform poorly in such complicated environments. The following figure summarizes the fundamental points described above:

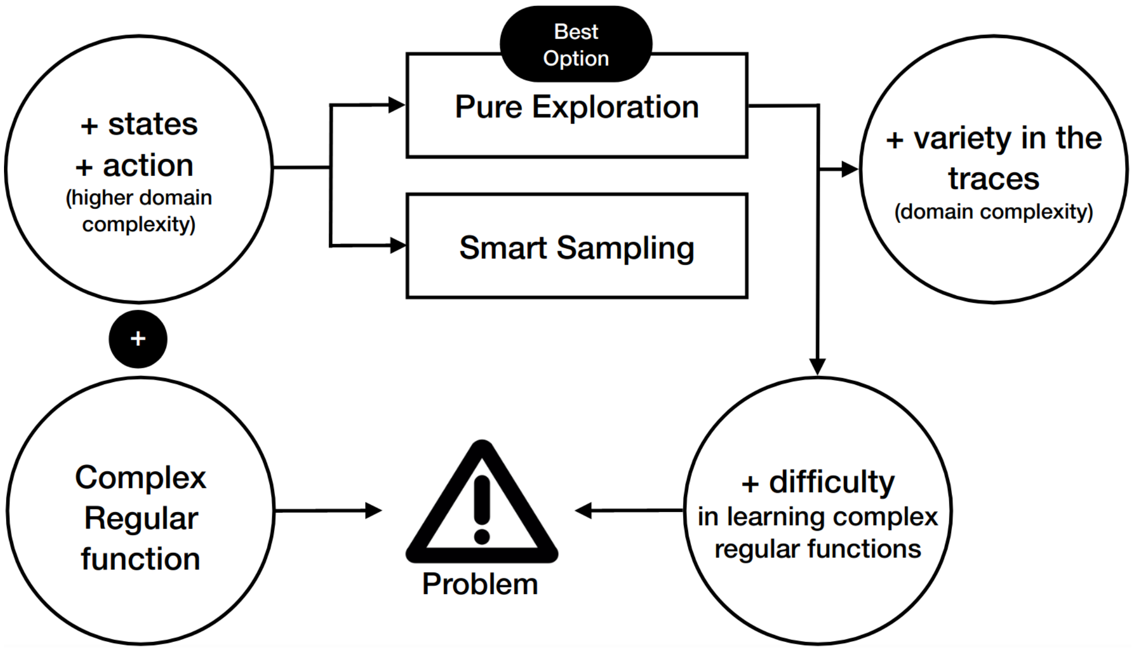


Figure 8: Problem analysis of s3m for domains similar to Rotating Maze

At the same time, this domain would disadvantage the PAC-RL algorithm too. Indeed, by including complex mechanisms, such as the antagonistic "agent" (still not another RL agent) blocking the actual agent for reward achievement or by including non-viable blocks that are moved according to some distributions, the reachability of a grid is likely to be significantly reduced. Meaning that both the latter and all other parameters that describe the Ronca and De Giacomo [2021] method will be reasonably high (unfavorable), preventing the algorithm – which is based on a random exploration – from converging in polynomial time. As a result, this algorithm may be equally ineffective for such class of domains.

From these assumptions stems our idea to put these concepts into a practical domain.

8.2.1 Grid domain

We created a domain similar to Rotating Maze but proved exceedingly challenging for both the S3M and PAC-RL algorithms. It is made out of a grid domain that contains the agent, his "opponent", and the goal that the agent must achieve. The Figure below shows our dominance in the initial situation.

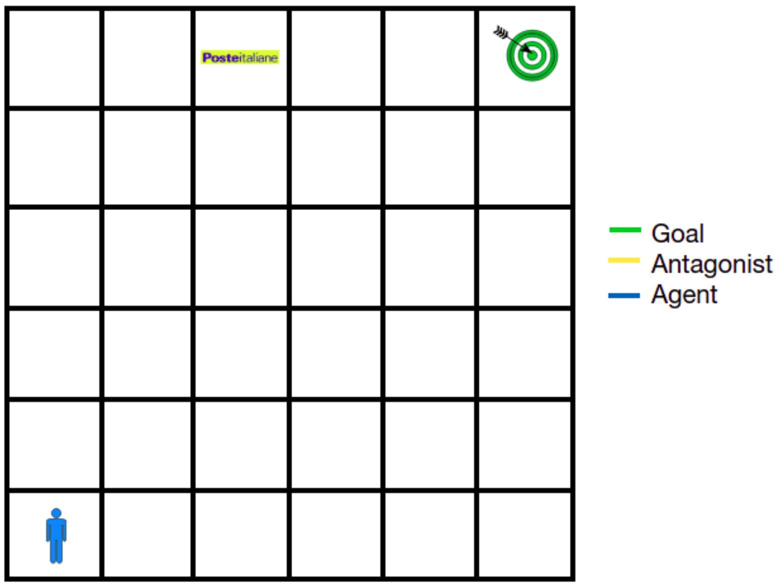


Figure 9: Initial setting of the domain

The antagonist moves at each agent's action toward one of the four surrounding cells according to a distribution. The rotation of the grid depends on the moves of both the agent and its antagonist. More precisely, the grid rotates every n chosen actions, with $n \geq 5$, just as it does in the Rotating Maze. If the agent goes in the same cell of its opponent, as in Figure 10, the algorithm will start over and the agent will be forced to try again to reach the goal.

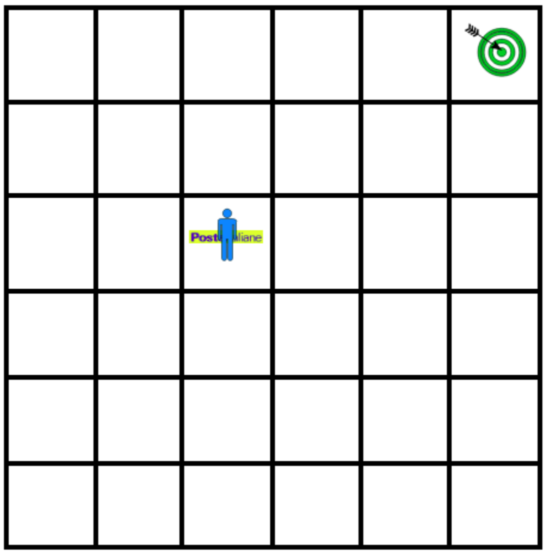


Figure 10: When both the agent and the antagonist are in the same cell the episode stops and the domain starts over

This family of domains is potentially difficult for the PAC-RL algorithm since, by carefully setting the distributions, one can make the aforementioned parameters unfavourable. Even with S3M, it turns out to be challenging, since the domain has a considerably bigger grid than Rotating Maze, further increasing the computing cost, which would rise, at least, to quadratic levels.

Furthermore, in the presence of an antagonist (or moving obstacles), the regular function becomes much more complicated. As a result, the Pure Exploration approach may not lead to effective learning since, as previously stated, the production of a large variety of traces may not be sufficient to learn a good quality automata. Even the Smart Sampling procedure would be ineffective because we will end up preferring to explore a limited set of traces.

9 Conclusions

For this project, we analyzed Regular Decision Processes – a recent non-Markovian extension of MDPs– working with the state-of-the-art papers of Abadi and Brafman [2020] and Ronca and De Giacomo [2021]. In particular, we focused on implementing S3M, an algorithm to learn the Mealy Machine underlying a specific domain. The advantage of this algorithm is that it enables to provide MCTS with a richer representation for its states.

After a deep analysis of S3M applied to several families of domains, it turned out to be very effective on their resolution. Furthermore, the performance are better than the baseline that uses MCTS only. To conclude the analysis, we tried to identify the cons of the two papers. To achieve this, we attempted to create a domain capable of underlining each paper's difficulties. We thought that an NxN grid domain (with a sufficiently big N, e.g., $N >= 5$) with the presence of an adversarial agent could be a good option. In particular, one of the main problems of S3M is the high computational cost that makes it inefficient in the above mentioned domain, i.e., a large domain where the final state could be challenging to reach. On the other hand, since the PAC-RL algorithm mainly depends on the parameters of the equation 7, such domain would not guarantee convergence since the values of the parameters would grow extremely.

In conclusion, even though the results of S3M seem to be promising, we believe that there still is room for improvement since, as aforementioned, the algorithm has a high computational cost that limits its application to a narrow set of domains.

References

E. Abadi and R. I. Brafman. Learning and solving regular decision processes. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1948–1954. International Joint Conferences on Artificial Intelligence Organization, 7 2020. doi: 10.24963/ijcai.2020/270. URL `https://doi.org/10.24963/ijcai.2020/270`. Main track.

R. I. Brafman and G. 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-19*, pages 1602–1608. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/222. URL `https://doi.org/10.24963/ijcai.2019/222`.

Kocsis, Levente and Szepesvári, Csaba. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

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.

D. Silver and J. Veness. Monte-carlo planning in large pomdps. Neural Information Processing Systems, 2010.

S. Verwer and C. A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017. doi: 10.1109/ICSME.2017.58.