Elective in AI - Reasoning Agents

Reinforcement Learning in Regular Decision Processes

Lorenzo Guercio	guercio.1913660@studenti.uniroma1.it	1913660
Francesco Starna	starna.1613660@studenti.uniroma1.it	1613660
Kevin Munda	munda.1905663@studenti.uniroma1.it	1905663
Graziano Specchi	specchi.1620431@studenti.uniroma1.it	1620431

2020/2021 Course

DIAG - Sapienza
Master in Artificial Intelligence and Robotics

Abstract

The introduction of *regular decision processes*, as a form of a non-Markov decision process, has been recently proposed to study reinforcement learning problems when the transition function depends on the whole history, rather than only on the current state. We implemented a *probably approximately correct* algorithm for learning regular decision processes, based on probabilistic automata learning. We finally did some experiments on three different domains in order to show which are the best characteristics that most fit the algorithm.

1 Introduction

Reinforcement learning (RL) has been using the Markov assumption to solve many domains. However, there are some domains in which it is not possible to neglect the past and only consider the current process state, and in those cases the assumption represents more of a limitation. *Regular decision processes* (RDP) have been recently proposed by [Brafman and De Giacomo, 2019] [1] to overcome this issue. The knowledge of a RPD allows one to compute an equivalent Markov decision process (MDP) and so to use standard solution techniques such as value iteration or policy iteration. The transformation process is not that immediate, in fact one needs to learn simultaneously a model of the dependencies on the past, and how observations and rewards evolve. Work on learning RDPs has been done in [Abadi and Brafman, 2020] [2], where the possibility for an RL agent to learn such a model in the form of an automaton has been exploited. The work provides empirical evidence that algorithms based on automata learning can find near-optimal policies for certain RDPs, however the proposed technique requires exponential time to converge. Subsequently [Alessandro Ronca and Giuseppe De Giacomo, 2021] [3] have introduced an algorithm which shows that RL in RDPs is feasible in polynomial time with respect to the probably approximately correct (PAC) learning criterion [Valiant, 1984; Kearns and Vazirani, 1994] [4] [5], and computes a near-optimal policy with high confidence, in a number of steps that is polynomial in the required accuracy and confidence. The algorithm first learns a probabilistic deterministic finite automaton (PDFA) that represents the underlying RDP, then it maps it to an MDP, which is solved to compute an intermediate stationary policy, which is then turned into a finite-state transducer (FST) by composing it with the transition function of the learned PDFA. We have implemented the proposed RL algorithm for solving RDPs, and in the following we show our solution and the experimental results on some remarkable domains.

2 Preliminaries

2.1 Finite-state Transducer

[Moore, 1956] [6] A finite-state transducer (FST) is a tuple $< Q, q_0, \Sigma, \emptyset, \Gamma, \theta >$ where:

- Σ is the finite input alphabet;
- Γ is the finite output alphabet;
- Q is a finite set of states;
- $q_0 \in$ Q is the initial state;
- τ : Q $\times$ $\Sigma \rightarrow$ Q is the deterministic transition function;
- θ : Q $\rightarrow \Gamma$ is the output function.

In this case the output function depends only on the state, so the FST is called *Moore Machine*. A transducer can be seen as a machine that maps every string $\omega \in \Sigma$ to the string $\theta(s_0, \omega)$. We use the transducer as output of the RL algorithm, which provides the transition function τ projected to the optimal policy learnt.

2.2 Probabilistic Deterministic Finite Automata

[Balle et al., 2014] [7] A Probabilistic Deterministic Finite Automaton (PDFA) is a tuple $< Q, \Sigma, \tau, \lambda, \xi, q_0 >$ where:

- Q is a finite set of states;
- Σ is an arbitrary finite alphabet;
- $\tau : Q \times \Sigma \rightarrow Q$ is the transition function;
- $\lambda : Q \times (\Sigma \cup \{\xi\}) \rightarrow [0, 1]$ 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);
- ξ is a special symbol not in Σ reserved to mark the end of a string;
- $q_0 \in Q$ is the initial state.

A PDFA can be seen as a graph; each node corresponds to a state $q \in$ in Q, and τ defines the transitions $Q \times \Sigma$ between them. We represent a RDP as a PDFA which is then learnt using the AdaCT algorithm.

2.2.1 Prefixes

A string *p* is a prefix of a string *s* if there exists a string *t* such that *s = pt*. The string itself is a prefix too. The empty string is a prefix of every string. For example the list of prefixes of the string "banana" is: *['', 'b', 'ba', 'ban', 'bana', 'banan', 'banana']*.

2.2.2 Supremum Distance

The supremum distance $L_\infty(D_1, D_2) = max_{x \in X} |D_1(x) - D_2(x)|$ is the maximum magnitude of the differences between two probability distributions. If the distributions are multisets of strings, we define the empirical distribution $\hat{S} = S(x)/|S|$, where $S(x)$ is the multiplicity of the string *x* in the multiset, and $|S|$ is the length of the multiset.

2.2.3 Prefixes Supremum Distance

The supremum distance on prefixes $prefL_\infty(D_1, D_2) = max_{x \in \Sigma*} |D_1(x\Sigma*) - D_2(x\Sigma*)|$, is the maximum magnitude of the differences between the prefixes of two probability distributions, where $D(x\Sigma*)$ is the probability under *D* of having *x* as a prefix. If the distributions are multisets of strings, we use the empirical distribution $\hat{S}$.

2.2.4 Distinguishability

We say distributions D_1 and D_2 are μ-distinguishable when $\mu \leq L_\infty(D_1, D_2)$. A PDFA is μ-distinguishable when, for each pair of states q_1 and q_2, their corresponding distributions $D(q_1)$ and $D(q_2)$ are μ-distinguishable. The distinguishability of a PDFA is defined as the supremum over all μ for which the PDFA is μ-distinguishable.

2.2.5 Prefix-distinguishability

We say distributions D_1 and D_2 are μ'-prefix-distinguishable when $\mu' \leq max\{L_\infty(D_1, D_2), prefL_\infty(D_1, D_2)\}$. A PDFA is μ'-prefix-distinguishable when, for each pair of states q_1 and q_2, their corresponding distributions $D(q_1)$ and $D(q_2)$ are μ-prefix-distinguishable. The prefix-distinguishability of a PDFA is defined as the supremum over all μ for which the PDFA is μ'-prefix-distinguishable.

2.2.6 Hyphothesis Graph

An algorithm for learning the relevant part of the structure of a target PDFA uses a data structure called a *hypothesis graph*: a directed graph G with a distinguished node called the initial state, where each arc is labelled using a symbol from Σ, and where each node is equipped with a multiset of strings from Σ^*. The size $|G|$ of a hypothesis graph is defined as the sum of the sizes of all the multisets assigned to its nodes. When the hypothesis graph is completed, its nodes and connections represent respectively the states and the transitions of a PDFA.

2.3 Markov Decision Process

[Bellman, 1957; Puterman, 1994] [8] [9] A Markov Decision Process (MDP) is a tuple M = <S, A, T, R, s_0, γ> where:

- S is a finite set of observation states;
- A is a finite set of actions;
- T : S* × A × S → [0, 1] is the transition function that returns for every state *s* and action *a* the distribution over the next state.
- R : S* × A × S → R is the reward function;
- s_0 is the initial state;
- $\gamma \in (0, 1)$ is the discount factor.

The transition and reward functions can be combined into the dynamics function D : S* × A × S × R → [0, 1] which defines a probability distribution $D(\cdot|s, a)$ over S × R for every s ∈ S* and every a ∈ A. A solution to an MDP is a policy π : S → A that maps each state to an action. An optimal policy, denoted π*, maximizes the expected sum of discounted rewards for every starting state s ∈ S.

We build a MDP with the PDFA learnt, in order to compute the optimal policy through *Value Iteration*. For *m* iterations, and for each state q ∈ Q, the algorithm calculates a state value, knowing the dynamics function. Maximizing the values of every transition we can build an optimal policy π.

2.4 non-Markov Decision Process

[Brafman and De Giacomo, 2019] [1] A non-Markovian Decision Process (NMDP) is defined identically to an MDP, except that the domains of *T* and *R* are finite sequences of states instead of single state. Every element *h* of S* is called a history. A policy is a function π : S* × A → [0, 1] that, for every history *h*, defines a probability distribution $\pi(\cdot|h)$ over the actions A. Every element of $(ASR)*$ is called a trace.

2.5 Regular Decision Process

[Alessandro Ronca and Giuseppe De Giacomo, 2021] [3] A Regular Decision Process (RDP) is a NMDP P = <S, A T, R, s_0, γ> whose transition and reward functions can be represented by finite-state transducers. Specifically, there is a finite transducer that, on every history *h*, outputs the function T_h: A × S → [0, 1] induced by *T* when its first argument is *h*, and there is a finite transducer that, on every history *h*, outputs the function R_h: A × S × R → [0, 1] induced by *R* when its first argument is *h*. In RDPs, the next state distribution and the reward function are conditioned on logical formulas based on LDLf – linear dynamic logic over finite traces [Giuseppe De Giacomo and Moshe Y. Vardi] [10], and refer to the entire history.

3 Reinforcement Learning in RDPs

We consider reinforcement learning (RL) as the problem of an agent that has to learn an optimal policy for an unknown RDP P, by acting and receiving observation states and rewards according to the transition and reward functions of P. The agent performs a sequence of actions a_1 ... a_n, receiving an observation state s_i and a reward r_i after each action a_i. The agent has the opportunity to stop and possibly start over. This process generates strings of the form $a_i s_i r_i$ ... $a_n s_n r_n$ which we call episodes. They form the experience of the agent, which is the basis to learn an optimal policy. In the following we explain our solution of the RL algorithm introduced by [Alessandro Ronca and Giuseppe De Giacomo, 2021] [3], which first goes through the learning process of a PDFA to represent a RDP.

3.1 Learning a PDFA - AdaCT

The algorithm used in this project to learn a probabilistic deterministic finite automata (PDFA) is the AdaCT algorithm defined in [Learning probabilistic automata, Balle, Castro, Gavaldà, 2013] [11]. The learning algorithm takes as inputs the alphabet size $|\Sigma|$, an upper bound n on the number of states of the target, the confidence parameter δ and a sample S containing N examples drawn from the target PDFA T with at most n states. AdaCT builds a hypothesis graph G incrementally: the algorithm performs at most $n|\Sigma|$ learning stages, each one making a pass over all training examples and guaranteed to add one transition to G. We can identify two periods in the learning process: the first one is called the "important period", during which the graph that AdaCT is building has theoretical guarantees of being a good representation of the target; during the second period, the transitions added to the graph don't have the same property and may be wrong. At the beginning of each stage, AdaCT has a graph G that summarizes the current knowledge of the target T; in particular nodes and edges in G represent states and transitions of the target T. The nodes of G are called "safe nodes", as during the important period they are guaranteed to stand for distinct states of T. Safe nodes are denoted by strings in $\Sigma*$. To each safe node v is attached a multiset S_{v}. If v was added to G during the important period, S_{v} keeps information about the distribution on the target state represented by v. At the start, the graph G consists of a single node v_0, that represents the initial state of the target, whose attached multiset equals the sample S received as input. At the beginning of each stage the algorithm creates a candidate node $u = v_{\mathrm{u}}\sigma$ for each safe node v_{u} of G and each $\sigma \in \Sigma$ such that $\tau_{\mathrm{G}}(v_{\mathrm{u}}, \sigma)$ is undefined. Attached to each candidate node u there is also a multiset S_{u}, initially empty. For each training example $x\xi = \sigma_0 ... \sigma_1 ... \sigma_{\mathrm{k}}\xi$ in the sample, AdaCT traverses the graph matching each observation σ_{i} to a state until either all observations in x have been exhausted and the example is discarded, or a transition to a candidate node is reached. This occurs when all transitions up to $\sigma_{\mathrm{i\text{-}1}}$ are defined and lead to a safe node w, but there is no edge out of w labeled by σ_{i}. Then the suffix $\sigma_{\mathrm{i+1}} ... \sigma_{\mathrm{k}}$ is added to the multiset of candidate $u = w\sigma_{\mathrm{i}}$. After that the following example is considered. When all training examples have been processed, the algorithm chooses the candidate $u = v_{\mathrm{u}}\sigma$ whose attached multiset S_{u} has maximum cardinality. Then it checks if this candidate corresponds to a target state that has not yet a representative in G. This is done by repeatedly calling the function $Test\_Distinct(u, v)$ described in Algorithm 4, varying v over the set of safe nodes, until some call returns 'Not Clear' or all safe nodes have been tested. We assume that, if $Test\_Distinct(u, v)$ returns 'Distinct', u and v correspond to distinct states in the target. Once all tests have been done, if all safe nodes have been found to be distinct from u, candidate u is promoted to a new safe node, so G gets a new node labelled with u and an edge from v_{u} to u labeled by σ is added to G. Multiset S_{u} is attached to the new safe node. Otherwise the algorithm chooses a safe node v that has not been distinguished from u and identifies nodes $v_{\mathrm{u}}\sigma$ and v by adding to G an edge from v_{u} to v labeled by σ; $v_{\mathrm{u}}\sigma$ is not anymore a candidate node. If there are no remaining candidates or G contains already $n|\Sigma|$ arcs, AdaCT returns G and stops. Otherwise, the current stage ends by erasing all candidates and a new stage begins.

Algorithm 1: AdaCT

Inputs : alphabet size Σ, upper bound of the number of states of the target n, confidence parameter δ, sample S drawn from the target PDFA T

Output: Hypothesis graph H

$H \longleftarrow$ init hypothesis graph with a node (initial state, multiset S);
$M \longleftarrow n * \Sigma$, maximum steps;
$SN \longleftarrow initialstate$, safe nodes array;
for $i \leq M$ **do**
  $CN \longleftarrow$ createCandidateNodesArray;
  if *length(CN)=0* **then**
    return H
  end
  $CN \longleftarrow$ populateMultisets(initial node);
  $N \longleftarrow$ select the node with maximum cardinality;
  for $j \leq$ *length(SN)* **do**
    $res \longleftarrow$ TestDistinct(N, SN_i, H, CN,δ, n, Σ);
    if *res = 'Not Clear'* **then**
      Add a transition from the parent of the candidate to the unclear node
    end
  end
  if *res = 'Distinct'* **then**
    Take the previous safe node;
    Add the link between the previous safe node and the new safe node;
    Add the new node to the graph;
  end
end
return H

Algorithm 2: createCandidateNodesArray

For each pair (n, a) (safe node, symbol) we check if a transition (an edge from n to another safe node through a) already exists in the hypothesis graph. If the transition exists, n is not a candidate node, otherwise we add (n, a) to the array

Algorithm 3: populateMultisets

Input: candidate node n
for *sample in* S **do**
  for *symbol in sample* **do**
    if *transition from* n *to another safe node through* $symbol$ *already exists* **then**
      $n \longleftarrow$ node reached by executing $symbol$ in n;
      $s \longleftarrow$ remove $symbol$ from $sample$;
    else
      Add the $sample$ to the multiset of n;
      break;

Algorithm 4: Test Distinct

Input : a candidate node u and a safe node v
Output: $Distinct$ or $NotClear$
$m_u \longleftarrow |S_u|$; $s_u \longleftarrow |prefixes(S_u)|$
$m_v \longleftarrow |S_v|$; $s_v \longleftarrow |prefixes(S_u)|$
$t_{u,v} \longleftarrow \sqrt{\frac{2}{min(m_u, m_v)} \cdot ln \frac{8(s_u + s_v)}{\delta_0}}$
$d \longleftarrow max(L_\infty(\hat{S}_u, \hat{S}_v), prefL_\infty(\hat{S}_u, \hat{S}_v)$
if $d > t_{u,v}$ **then**
  return $Distinct$
else
  return $NotClear$

3.2 PAC-RL Algorithm

3.2.1 Representing RDP as a PDFA

The learning algorithm of RDP relies on learning probabilistic automata. This can be done by representing a RDP as a PDFA, by introducing an extra action for the agent, the stop action; this is a special action ξ that allows the agent to terminate the current episode and start a new one. The exploration policy π_{p} with stop probability $p > 0$, is the policy that selects the stop action ξ with probability p and each action from A with probability $(1 - p)/|A|$. Under an exploration policy, a RDP determines a probability distribution on traces, and hence can be seen as a PDFA.

We consider a RDP P = <A, S, R, T, R, γ_{i}>, its dynamics function D, and the minimum transducer T = <Q, q_0, S, τ, Γ, θ_{i}> that represents D in the sense that $\theta(h)(a, s, r) = D(s, r|h, a)$; namely the output function of the transducer given a triplette asr corresponds to the dynamic function of the RDP given a history h and an action a. Specifically, the dynamics of a RDP under π_{p} are captured by the PDFA A = <Q, Σ, τ', λ, ξ, q_0> where:

- alphabet $\Sigma = \{asr \in ASR \mid \exists h.\ D(s, r|h, a) > 0\}$;
- transitions τ'(q, asr) = τ (q, s);
- probability function:
  – $\lambda(q, asr) = ((1 - p)/|A|)\theta(q)(a, s, r)$,
  – $\lambda(q, \xi) = p$.

3.2.2 Probably Approximately Correct

The goal is to understand how fast an agent can learn a near-optimal policy, giving more importance to time rather than maximizing the rewards. For this reason the notion of *Probably Approximately Correct* (PAC) framework [Valiant, 1984; Kearns and Vazirani, 1994] [4] [5], is introduced to support learning. The framework is based on the observation that exact learning is not feasible. Thus, it introduces two parameters $\varepsilon > 0$ and $\delta \in (0, 1)$ that describe the required accuracy and the confidence of success, respectively. In the RL setting it translates into looking for policies that are ε-optimal with probability at least $1 - \delta$.

3.2.3 Algorithm

Algorithm 5 shows the pseudo-code of the RL algorithm. It takes as input the action space A of the environment, a discount factor γ, the required accuracy ε, the confidence of success δ, and the upper bound on transducer states $\hat{n}$. Here, for simplification, we already know the number of states of the transducer, given by the environment specification. The algorithm first initializes the stop probability p and the set of episodes X, generated under an exploration policy $ß_{\text{p}}$. It reads the set Σ of action-state-reward symbols, and the maximum reward $\hat{R}_{\text{max}}$ occurring in X.

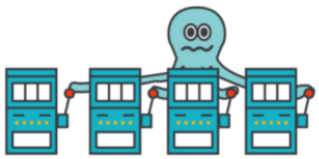

Figure 1: RotatingMab Environment

Figure 2: RotatingMaze Environment

It then learns the PDFA $\hat{A} =< \hat{Q}, \hat{\Sigma}, \tau', \lambda, \xi, \hat{q}_0 >$ via the AdaCT algorithm with inputs alphabet $\hat{\Sigma}$, confidence parameter δ, upper bound on transducer state n̂, and samples X. From Â it computes the MDP M̂ = <Q̂, A, $D^{\widehat{M}}$, γ, q̂$_0$>, where the dynamics function is computed as:

$D^{\widehat{M}}(q_2, r|q_1, a) = (|A|/(1-p)) * \sum_{s:\hat{\tau}'(q_1,asr)=q_2} \hat{\lambda}(q1, asr)$

This equation sums the probability function for each asr symbol for which the transition from a state q_1 with triplette asr to a state q_2 is defined. The probability function is defined as the empirical distribution of the prefixes of the multiset u attached to the node corresponding to the state q_1, hence, the multiplicity of the triplette asr in the multiset of the prefixes of u. Given the MDP, the algorithm solves it via m iterations of value iteration, finding an optimal policy $\pi*$. The policy function obtained is then composed with the transition function $\hat{\tau}$ defined as $\hat{\tau}(q, s) = \hat{\tau}'(q, asr)$ for an arbitrary choice of a and r such that $\hat{\lambda}(q, asr) > 0$. The algorithm returns a transducer $< \hat{Q}, \hat{q_0}', \hat{S}, \tau, A, \theta' >$, where $\theta'(\hat{q}) = \pi(\hat{q})$ and $\hat{S}$ consists of each observation-state occurring as the second component of an element of $\hat{\Sigma}$.

Algorithm 5: Reinforcement Learning RL

Input : Actions A, discount factor γ, required precision ϵ, confidence parameter δ, upper bound $\hat{n}$ on transducer states
Output: transducer of policy π
$p \longleftarrow 1/(10 \cdot \hat{n} + 1)$
$X \longleftarrow$ generate episodes under exploration policy π_p
$\hat{\Sigma} \longleftarrow$ symbols in X
$\hat{R}_{max} \longleftarrow$ max reward in X
$\hat{\mathcal{A}} \longleftarrow$ learn PDFA by calling $AdaCT(\hat{n}, |\hat{\Sigma}|, \delta, X)$
$\hat{M} \longleftarrow$ compute MDP induced by $\hat{\mathcal{A}}$ and γ
$m \longleftarrow \lceil \frac{1}{1-\gamma} \cdot ln\left(\frac{2 \cdot \hat{R}_{max}}{\epsilon \cdot (1-\gamma)^2}\right) \rceil$
$\pi \longleftarrow$ solve $\hat{M}$ by calling $ValueIteration(\hat{M}, m)$
return transducer composition of π and transition function τ' of $\hat{\mathcal{A}}$

4 Experiments

4.1 Environments

We tested the RL algorithm on three different domains, two of which are well known and already established. These are Rotating Multi-Armed-Bandit (MAB) and Rotating Maze. The MAB problem is a classic reinforcement learning example where we are given a slot machine with *n* arms (bandits) with each arm having its own rigged probability distribution of success. Pulling any one of the arms gives a stochastic reward of either '100' for success, or 0 for failure. The rotating variant simply means shifting the winning probabilities when the agent gains the reward. The objective is to pull the arms one-by-one in sequence maximizing the total reward collected in the long run.

The second domain has a maze platform, in which the objective is to minimize the number of steps to reach the goal states, namely exit the maze. The agent receives a unique reward '100' when one of the goal states has been reached. Fig.1 and Fig.2 show a graphical representation of the two domains, for a clearer understanding.

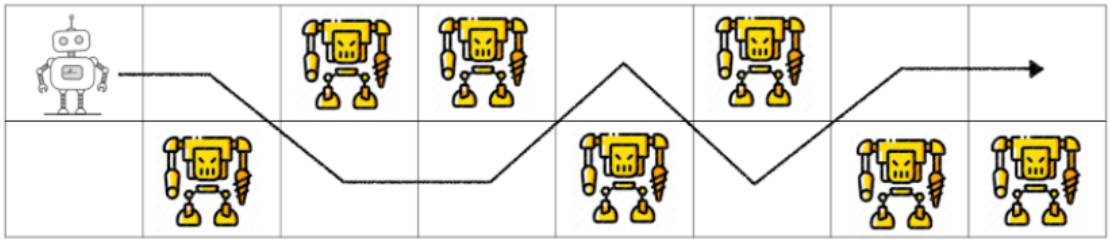

Figure 3: EnemyCorridor Environment

The third domain on which we tested the algorithm is a new one called Enemy Corridor. The agent has to cross a $2m$ grid while avoiding enemies. Every enemy guards a two-cell column: enemy i guards cells $(0, i)$ and $(1, i)$, and it is found in cell $(0, i)$ with probability $p_0 i$ or $p_1 i$. Initially the probability is $p_0 i$, and the two probabilities are swapped every time the agent hits an enemy. At every step, say i, the agent has to decide whether to go through cell $(0, i)$ or $(1, i)$, taking action a_0 or a_1, respectively. Specifically, when in cell $(0, i)$ or $(1, i)$, action a_0 leads to $(0, i + 1)$ and action a_1 to $(1, i + 1)$. From the last column, the agent is brought back to the first one. The agent receives a reward '100' every time it avoids an enemy. To maximise rewards, the agent has to learn to predict the enemies' positions, which depend on the history of observation states. Fig.3 shows a graphical representation of the Enemy Corridor domain.

Parameters	γ	ϵ	δ	$\lvert X \rvert$	$\lvert \pi \rvert$	l
Description	discount factor	required accuracy	confidence of success	number of samples	number of learned policies	expected episode length
Values	0.1	0.01	0.1	50000-500000	5-10	10-15

Table 1: Parameters used in RL algorithm

4.2 Results

We tested the three domains according to various settings. The parameters we tuned are reported in Table 1. The stop probability p is computed according to the episode length l as $p = 1/(l+1)$. The algorithm that we developed is highly dependent on the number of samples used to run the AdaCT algorithm. We tested the three domains progressively, from the simplest one (RotatingMab) to the most complex (RotatingMaze). The difference resides in the time taken to build the PDFA with the AdaCT algorithm and in the number of samples needed to reach the optimum level of reward for the environment. Since AdaCT cannot be parallelized, it is difficult to optimize the algorithm and the major problem that we faced is the long time (even several hours) needed to perform the tests. Now, we are going to present the results obtained for each environment. The subsequent plots have the number of samples as the x axis and the average reward on a single step on the y axis. We will see that, for a certain number of samples which is different for each environment, the reward reaches a plateau representing the maximum. For this number of samples, the obtained PDFA from the AdaCT algorithm represents the environment in the best way possible.

4.3 Rotating Mab

This environment is also used in the experiments of [Abadi and Brafman, 2020] [2]. For this reason, we compared our results with the already existing ones. We tried to recreate the same experiment conditions in order to be able to draw conclusions: the expected episode length is 10, and 10 is also the number of policies that are averaged at each step in order to evaluate the algorithm precisely. What we found out is that our method outperformed, albeit slightly, the method of Abadi and Brafman. Even if the two algorithms have a substantial difference in the fact that ours depends on the number of samples used, while theirs depend on the total number of steps, the yardstick, which is the averaged reward, is the same. As we can see in the two graphs below, at the same conditions our method outperforms the Abadi and Brafman one, even using the best sampling solution. Approximately at the 50k samples mark, our reward stabilizes at the level of 90, which means that a single step in the algorithm gives the reward '100' nine times out of ten.

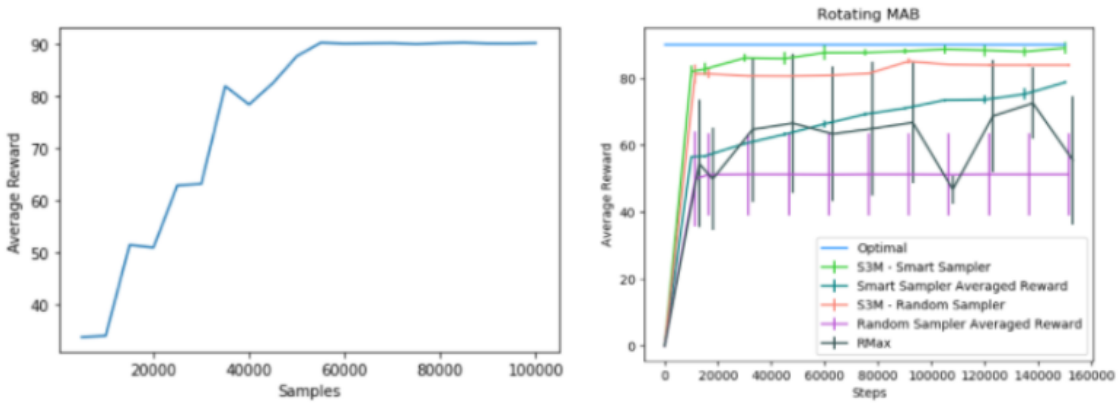

Figure 4: Comparison of the RotatingMab results with [Abadi and Brafman, 2020] [2]

It's also interesting to notice that, if we choose to run our algorithm with a higher episode length, the results are much more stable and the maximum is reached many samples before. On the other hand, the running time of the algorithm is much longer, and for this reason a middle way could be ideal for our study case. In the image below, the *expected episode length* parameter is set to 40, and the resulting graph is very robust to anomalies. We can notice that the *steps* in the graph correspond to the adding of a state in the PDFA found by the AdaCT algorithm: each time a node is added to the PDFA, a sudden increase in the average reward takes place.

4.4 Enemy Corridor

This second environment, which we showed before in detail, is original and has been created in order to test the algorithm in a more complete way. The tests have been performed with the same structure of the RotatingMab experiments in order to have a common yardstick, even if we cannot compare the results with [Abadi and Brafman, 2020] [2]. In the images below, we can see the plot of this environment. As in the RotatingMab environment, the steps represent the addition of a state in the PDFA returned by the AdaCT algorithm. The two plots only differ in the value of the expected episode length. In the left image, the parameter is set to 20, while in the right one it is set to 80, obtained with the operation $x = 10 * nodes$. This value guarantees the success of the creation of the PDFA from a certain number of samples going on. As in the RotatingMab environment, we can notice that, increasing the expected length of the episode, the number of samples needed to reach the maximum is inferior even if the time taken to learn the PDFA is superior. The best average reward obtained is around 75, which means that for each action taken, the reward '100' is obtained three times out of four. For this reason, we can say that the player has learned to a certain amount how to predict the enemies' positions, operation that depends on the history of observation states. In the image below, we can see the PDFA learned by the AdaCT algorithm. In this case, the graph is clearly more complex and of difficult comprehension.

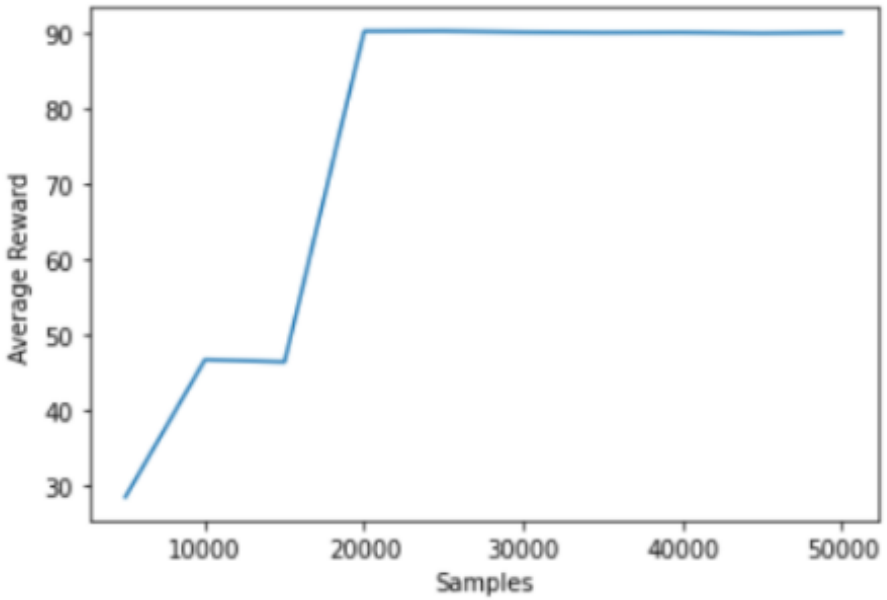

Figure 5: RotatingMab Plot with Expected Episode Length of 40

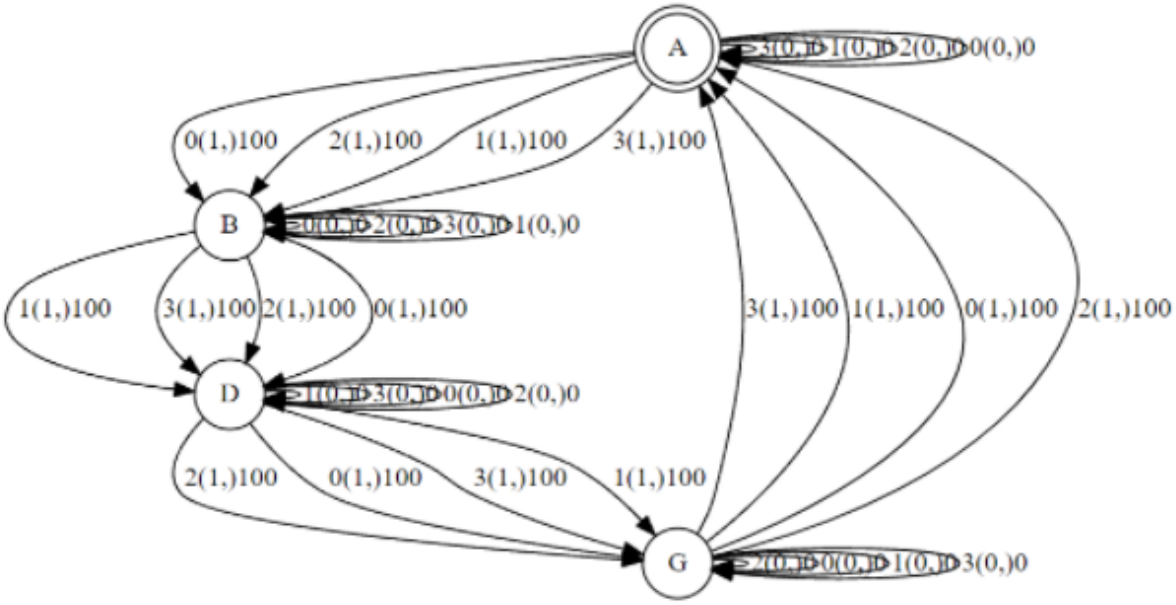

Figure 6: Graph of the RotatingMab Environment

However, it is correct since our corridor has length 4: the graph has 8 nodes (as the 8 cells in the environment's grid) and four transitions in and four out from each node. The transitions represent the action of going in the two successive cells on the right: for example, from the cell $(0, 2)$ we can go to $(0, 3)$ or $(1, 3)$, but for both of these two actions we can have a failure or a success. Hence, four transitions in total.

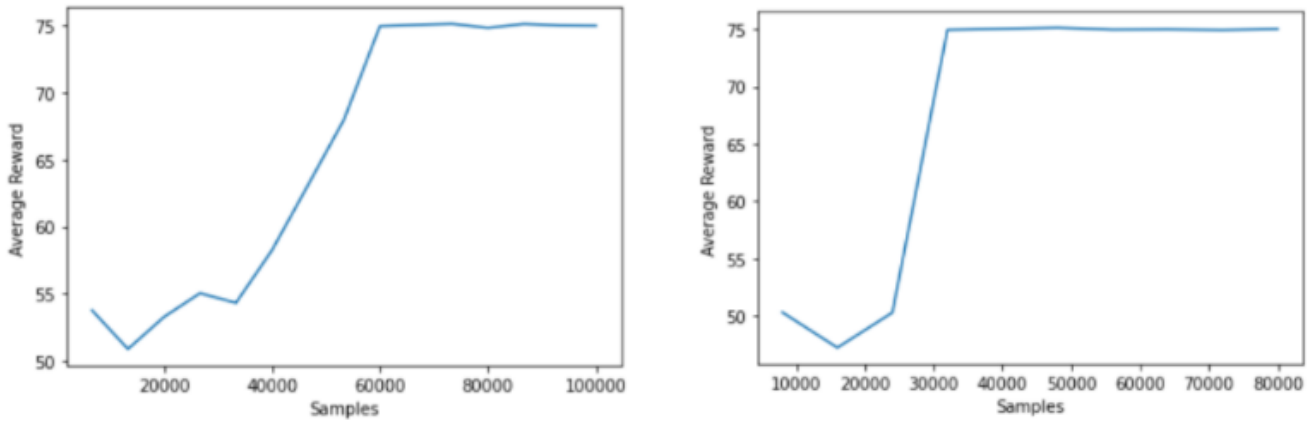

Figure 7: Left: episode length of 20 - Right: episode length of 80

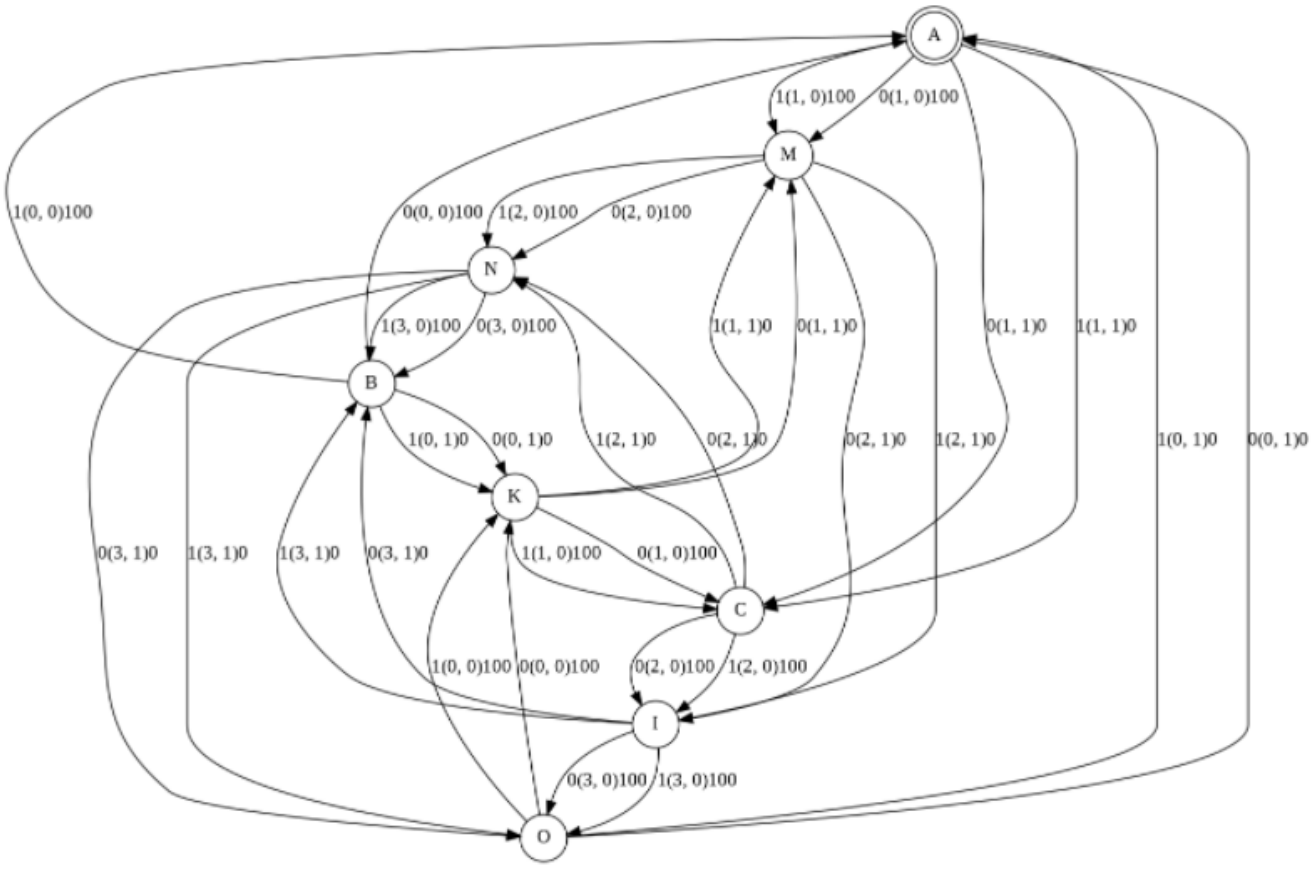

Figure 8: Graph of the EnemyCorridor environment

4.5 Rotating Maze

The last environment that we studied is by far the most complex. As explained before it's a 4*4 grid, where The possible actions of the agent are *up/down/left/right*. These actions succeed 90% of the time, and in the rest 10% the agent moves in the opposite direction. Our goal is to get to the designated location where a final reward '100' is given. An additional difficulty is added: in this maze, every three actions the agent's orientation changes by 90 degrees.
In our experiment, the goal is five steps away from the initial position. This environment is a little bit different from the previous ones, because the reward is only given at the target. For this reason, the average reward will be much lower than before, and in any case never greater than 20, because the minimum number of steps to reach the goal is five. In this border case, we have one action on five in total that gives the reward '100', and for this reason the average reward on all the actions of this episode is 20 $(0 * 4 + 100 * 1)/5$. In the images on the left, we can see the results obtained by our method and on the right the results obtained by [Abadi and Brafman, 2020] [2]. We notice that the results obtained with our method are a little bit lower than the other ones. The value of expected episode length used is 15, in order to make a perfect comparison with the Abadi and Brafman results. The number of samples needed to obtain a good result is very high: we were able to obtain a decent result only with more than 200.000 samples.

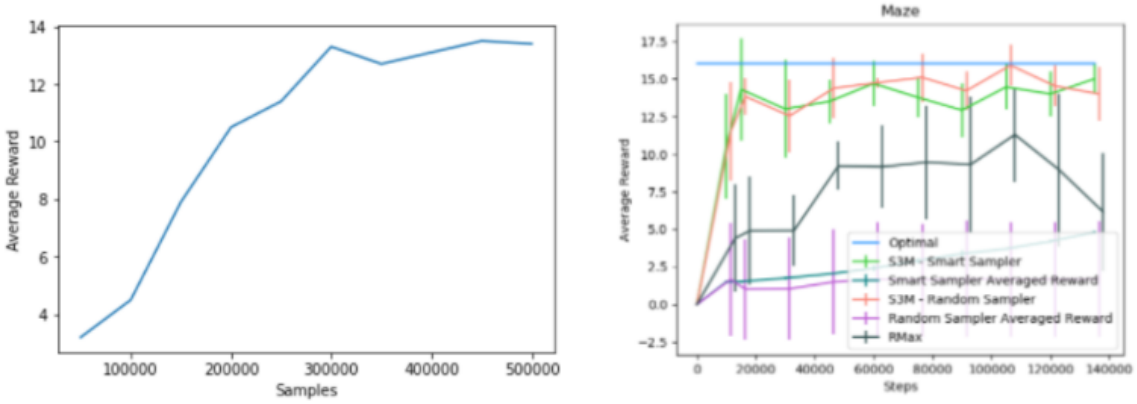

Figure 9: Comparison of the RotatingMaze results with [Abadi and Brafman, 2020] [2]

5 Conclusions

As a final discussion, after many tries on different settings for testing the three domains on learning RDPs, we came up with the most suitable characteristics which a domain should have in order to efficiently exploit the RL algorithm. First we present a suitable environment we analyzed. We named it *Minefield*, it is represented in Fig.10, and it is formalized as follows:

- *States*. A N x M grid where each state (n, m) is a grid cell.
- *Actions*. The agent can perform 3 actions: *Right*, *Left*, *Down*.
- *Observations*. Every time the agent performs an action, it receives back an observation $(n, m, bomb)$, specifically the cell position and whether the cell contains a bomb.
- *Probabilities*. Every row contains exactly one bomb, so the probability to find one is $1/m$. Every time the agent hits one, the bombs are shifted down of one row (the last bomb goes to the first row). There are two exceptions to this rule, concerning the first and last row of the environment: for the first row, the bomb cannot be in the initial cell $(0, 0)$ of the grid, while in the last row the bomb cannot be in the cell (N, M), the terminal state and called by us the *flower's cell*. In those two cases, the probability to find the bomb is $1/(m - 1)$.

- *Rewards*. Every time the agent runs into a cell with a bomb receives a negative reward -20. The only reward $+20 \cdot n$, with n as the number of rows in the environment, is achieved when the agent reaches the goal in the *flower's cell*.
- *Initial State*. The cell $(0, 0)$.
- *Terminal State*. The cell (N, M).

The goal of the agent is to find the bomb-free path and obtain the maximum reward, learning to predict bombs' positions, which depend on the history of observations. This environment summarizes the characteristic that better exploits the RL algorithm:

- *A relatively small number of states*. AdaCT takes a long time to learn the Rotating Maze domain, moreover not sufficiently, because the path combinations are too much. Also a grid can be very huge, so we suppose at maximum a 5x5 dimension. For this reason, we decided to opt for a total number of 3 actions, omitting the *Up* move in order to decrease the total number of states and the complexity of the environment.
- *Allowing the agent to explore the entire state domain*. Again referring to the Rotating Maze domain, the agent stops many times because of the presence of many terminal states. This prevents the graph from being built completely, missing many transitions. So we put only one terminal state which gives the only reward. Another possibility would be to have no terminal states, and obtain a positive reward if the agents misses the bomb, and a negative otherwise; the goal in this case is to maximize the reward taking a potentially infinite path without hitting any bomb (preventing the agent from coming back to the previous cell at each step).
- *No variables about the agent's orientation*. Another difference from the Rotating Maze domain is that in our environment we don't have any variable referring to the orientation of the agent. In the Maze environment, we have a change in the orientation of the agent every 3 actions that puts AdaCT in trouble trying to learn every possible combination. For this reason, we never refer to orientation in our domain, helping the algorithm to find the correct PDFA.

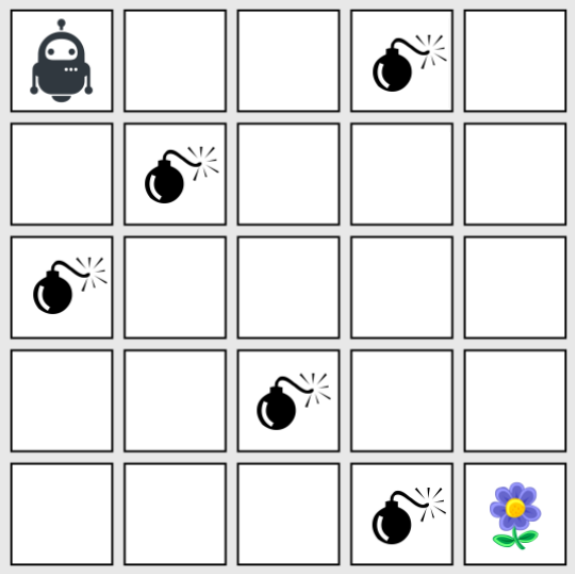

Figure 10: Minefield environment with 5x5 grid states.

6 Bibliography

[1] *Regular Decision Processes: A Model for Non-Markovian Domains*, Brafman, Ronen I. and De Giacomo, Giuseppe, 2019, https://doi.org/10.24963/ijcai.2019/766
[2] *Learning and Solving Regular Decision Processes*, Abadi, Eden and Brafman, Ronen I., 2020, https://doi.org/10.24963/ijcai.2020/270
[3] *Efficient PAC Reinforcement Learning in Regular Decision Processes*, Ronca, Alessandro and De Giacomo, Giuseppe, 2021, https://www.researchgate.net/publication/351623733_Efficient_PAC_Reinforcement_Learning_in_Regular_Decision_Processes
[4] *A theory of the learnable*, Valiant, L. G., 1984
[5] *An Introduction to Computational Learning Theory*, Kearns, Michael and Vazirani, Umesh, 1994, https://mitpress.mit.edu/books/introduction-computational-learning-theory
[6] *Gedanken-experiments on Sequential Machines*, Moore, Edward F., 1956, https://philpapers.org/rec/CHUMEF
[7] *Adaptively learning probabilistic deterministic automata from data streams*, Balle et al., 2014, https://doi.org/10.24963/ijcai.2020/270
[8] *Dynamic Programming*, Bellman., R. E., 1957, https://press.princeton.edu/books/paperback/9780691146683/dynamic-programming
[9] *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Puterman, M. L. , 1994, https://onlinelibrary.wiley.com/doi/book/10.1002/9780470316887
[10] *Linear Temporal Logic and Linear Dynamic Logic on Finite Traces*, De Giacomo, Giuseppe and Vardi, Moshe Y., 2013, https://www.cs.rice.edu/~vardi/papers/ijcai13.pdf
[11] *Learning probabilistic automata: A study in state distinguishability*, Balle, B., Castro, J. and Gavaldà, R., 2013, https://dl.acm.org/doi/abs/10.1016/j.tcs.2012.10.009