Learning Reward Machines: A Study in Partially Observable Reinforcement Learning[*]

Rodrigo Toro Icarte[a,c], Ethan Waldie[b], Toryn Q. Klassen[b,c], Richard Valenzano[d],
Margarita P. Castro[a], Sheila A. McIlraith[b,c]

[a] *Pontificia Universidad Católica de Chile, Vicuña Mackenna 4860, Macul, RM, Chile*
[b] *University of Toronto, 214 College St, Toronto, ON, Canada*
[c] *Vector Institute, 661 University, Toronto, ON, Canada*
[d] *Ryerson University, 350 Victoria St, Toronto, ON, Canada*

Abstract

Reinforcement learning (RL) is a central problem in artificial intelligence. This problem consists of defining artificial agents that can learn optimal behaviour by interacting with an environment – where the optimal behaviour is defined with respect to a reward signal that the agent seeks to maximize. Reward machines (RMs) provide a structured, automata-based representation of a reward function that enables an RL agent to decompose an RL problem into structured subproblems that can be efficiently learned via off-policy learning. Here we show that RMs can be learned from experience, instead of being specified by the user, and that the resulting problem decomposition can be used to effectively solve partially observable RL problems. We pose the task of learning RMs as a discrete optimization problem where the objective is to find an RM that decomposes the problem into a set of subproblems such that the combination of their optimal memoryless policies is an optimal policy for the original problem. We show the effectiveness of this approach on three partially observable domains, where it significantly outperforms A3C, PPO, and ACER, and discuss its advantages, limitations, and broader potential.[1]

Keywords: reinforcement learning, reward machines, partial observability, automata learning, abstractions, non-Markovian environments

1. Introduction

A fundamental component of human intelligence is our ability to make decisions – to decide how to act. Indeed, decision making is essential not only to individuals, but to companies, to governments, and to computer-controlled systems that ensure the safe and effective operation of much of our modern infrastructure. Unfortunately, making good decisions can be hard. It can depend on complex inter-relationships between diverse factors, not all of them observable, or well understood. *Reinforcement learning (RL)*

---

[1] Our code is available at `https://bitbucket.org/RToroIcarte/lrm`.
[*] This work is an extended version of our previous NeurIPS19 publication (Toro Icarte et al., 2019).
   Email address: `rntoro@uc.cl` (Rodrigo Toro Icarte)

endeavours to solve sequential decision-making problems using minimal supervision and minimal prior knowledge. Its goal is to define artificial agents that learn optimal behaviour by interacting with an environment, which may take the form of a simulator or the real world (Sutton and Barto, 2018). Every interaction with the environment delivers a reward signal. An RL agent seeks to learn a *policy* (a mapping from observations to actions) that maximizes its expected cumulative reward, improving its policy over time by learning from its past experiences.

The use of neural networks for function approximation has led to many recent advances in RL. Such deep RL methods have allowed agents to learn effective policies in many complex environment including board games (Silver et al., 2017), video games (Mnih et al., 2015), and robotic systems (Andrychowicz et al., 2018). However, RL methods (including deep RL methods) often struggle when the environment is *partially observable*. Indeed, partial observability is one of the main challenges towards applying RL in real-world problems (Dulac-Arnold et al., 2019, 2021). This is because agents in such environments usually require some form of memory to learn optimal behaviour (Singh et al., 1994). Recent approaches for giving memory to an RL agent either rely on recurrent neural networks (e.g., Hausknecht and Stone, 2015; Mnih et al., 2016; Jaderberg et al., 2016; Wang et al., 2016; Schulman et al., 2017), memory-augmented neural networks (e.g., Oh et al., 2016; Khan et al., 2017; Hung et al., 2018), or external memories that the agent can control using primitive actions (e.g., Littman, 1993; Peshkin et al., 1999; Zhang et al., 2016; Toro Icarte et al., 2020b).

In this work, we show that *reward machines (RMs)* are another useful tool for providing memory in a partially observable environment. RMs were originally conceived to provide a structured, automata-based representation of a reward function (Toro Icarte et al., 2018, 2020a; Camacho et al., 2019; De Giacomo et al., 2020). Exposed structure can be exploited by the *Q-learning for reward machines (QRM)* algorithm (Toro Icarte et al., 2018), which simultaneously learns a separate policy for each state in the RM. QRM has been shown to outperform standard and hierarchical deep RL over a variety of discrete and continuous domains. However, QRM was only defined for fully observable environments. Furthermore, the RMs were handcrafted by a user and then given to the RL agent, thus allowing the agent to exploit the exposed problem substructure. Here, we propose a method for learning an RM directly from experience in a partially observable environment, in a manner that allows the RM to serve as memory for an RL algorithm.

There are three main contributions of this work. The first is to propose a discrete optimization problem for learning reward machines from experience in a partially observable environment, where the objective is to find a reward machine that makes the problem *as Markovian as possible*. A requirement is that the RM learning method be given a finite set of detectors for properties that serve as the vocabulary for the RM. The model is also fed with traces collected by the agent while exploring the environment. Then, the optimization problem's objective function ranks the reward machines according to how well they predict future observations given the current RM state.

Our second contribution is to study different methodologies to solve the resulting optimization problem for learning reward machines. In particular, we propose a *mixed integer linear programming (MILP)* model, a *constrained programming (CP)* model, and two *local search (LS)* methods. In our experiments, the best performance was obtained by the local search methods.

Finally, we show how to integrate our models for learning reward machines into the

agent-environment interaction loop and show the effectiveness of doing so. By simultaneously learning an RM and a policy for the environment, we are able to significantly outperform several deep RL baselines that use recurrent neural networks as memory in three partially observable domains. We also extend the RM-tailored algorithm *Q-learning for reward machines (QRM)* to the case of partial observability where we see further gains when combined with our RM learning method.

This paper builds upon Toro Icarte et al. (2019) – where we originally proposed to formulate the problem of learning an RM as a discrete optimization problem and solved it using tabu search. In this work, we provide further details about this learning pipeline and propose three novel formulations to learn reward machines. These new formulations include a MILP, CP, and LS model. We compare the performance of these models relative to tabu search and found that a local search approach with restarts is consistently better at finding high-quality RMs than tabu search. As a result, the performance of our method improves considerably with respect to the performance reported in our previous publication.

2. Preliminaries

RL agents learn policies from experience. When the problem is fully-observable, the underlying environment model is typically assumed to be a *Markov decision process (MDP)*. An MDP is a tuple $\mathcal{M} = \langle S, A, r, p, \gamma, \mu \rangle$, where S is a finite set of *states*, A is a finite set of *actions*, $r : S \times A \to \mathbb{R}$ is the *reward function*, $p(s, a, s')$ is the *transition probability distribution*, γ is the *discount factor*, and μ is the *initial state distribution* where $\mu(s_0)$ is the probability that the agent starts in state $s_0 \in S$. In addition, a subset of the states might be labelled as *terminal states*.

At the beginning of an *episode*, the environment is set to an initial state s_0, sampled from μ. Then, at time step t, the agent observes the current state $s_t \in S$ and executes an action $a_t \in A$. In response, the environment returns the next state $s_{t+1} \sim p(\cdot|s_t, a_t)$ and the immediate reward $r_{t+1} = r(s_t, a_t, s_{t+1})$. The process then repeats from s_{t+1} until potentially reaching a terminal state, when a new episode will begin.

The agent's goal is to collect as much reward from the environment as possible. To do so, it learns a *policy* $\pi(a|s)$, which is a probability distribution over the actions $a \in A$ given a state $s \in S$. As the agent interacts with the environment, it also improves its policy until (ideally) finding an *optimal policy* π^*. An optimal policy is a policy that maximizes the expected return received by the agent, which is formally defined as follows:

$$\pi^* = \arg\max_{\pi} \sum_{s \in S} \mu(s) \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \bigg| s_0 = s \right] \tag{1}$$

Q-learning (Watkins and Dayan, 1992) is a well-known RL algorithm that uses samples of experience of the form (s_t, a_t, r_t, s_{t+1}) to estimate the optimal Q-function $q^*(s, a)$. Here, $q^*(s, a)$ is the expected return of selecting action a in state s and following an optimal policy π^* thereafter. During execution, Q-learning maintains the current estimate of the optimal Q-function (i.e., *Q-value*) for each state s and action a, $Q(s, a)$. Given a sampled experience (s_t, a_t, r_t, s_{t+1}), where s_{t+1} is the state reached after executing action a_t in state s_t and receiving a reward r_t, Q-learning updates $Q(s_t, a_t)$ towards

$r_t + \gamma \max_{a' \in A} Q(s_{t+1}, a')$. Given enough experience, the Q-value estimates will converge to the optimal Q-function, and so an optimal policy π^* can be computed by always selecting the action a with the highest value of $Q(s, a)$ for each state $s \in S$.

Unfortunately, Q-learning is impractical when solving problems with large state spaces. In such cases, *function approximation* methods are often used. Instead of storing a Q-value for each state-action pair in a table, deep RL methods like DQN (Mnih et al., 2015) and DDQN (Van Hasselt et al., 2016) represent the Q-function as $Q_\theta(s, a)$, where Q_θ is a neural network whose inputs are features of the state and the outputs are the Q-value estimates for each action $a \in A$. To train the network, mini-batches of experiences (s, a, r, s') are randomly sampled from an *experience replay* buffer and used to minimize the Bellman error. In the case of DQN, this is accomplished by minimizing the square error between $Q_\theta(s, a)$ and the Bellman estimate $r + \gamma \max_{a'} Q_{\theta'}(s', a')$. Note that the updates are made with respect to a *target network* with parameters θ'. The parameters θ' are held fixed when minimizing the square error, but updated to θ after a certain number of training updates. The role of the target network is to stabilize learning. DDQN follows a similar approach, but the Bellman estimate is computed by selecting the next action a' using Q_θ instead of the target network. This is, $r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_\theta(s', a'))$.

In partially observable problems, the underlying environment model is typically assumed to be a *Partially Observable Markov Decision Process (POMDP)*. A POMDP is a tuple $\mathcal{P}_\mathcal{O} = \langle S, O, A, r, p, \omega, \gamma, \mu \rangle$, where S, A, r, p, γ, and μ are defined as in an MDP, O is a finite set of *observations*, and $\omega(o|s)$ is the *observation probability distribution*. Interacting with a POMDP is similar to interacting with an MDP. The environment starts from a sampled initial state $s_0 \sim \mu$. At time step t, the agent is in state $s_t \in S$, executes an action $a_t \in A$, receives an immediate reward $r_{t+1} = r(s_t, a_t, s_{t+1})$, and moves to s_{t+1} according to $p(s_{t+1}|s_t, a_t)$. However, the agent does not observe s_t directly. Instead, the agent observes $o_t \in O$, which is linked to s_t via ω, where $\omega(o_t|s_t)$ is the probability of observing o_t from state s_t (Cassandra et al., 1994).

RL methods cannot be immediately applied to POMDPs because the transition probabilities and reward function are not necessarily Markovian w.r.t. O (though by definition they are w.r.t. S). As such, optimal policies may need to consider the complete history $(o_0, a_0, \ldots, a_{t-1}, o_t)$ of observations and actions when selecting the next action.[2] Several partially observable RL methods use a recurrent neural network to compactly represent the history, and then use a policy gradient method to train it. However, when we do have access to a full POMDP model $\mathcal{P}_\mathcal{O}$, then the history can be summarized into a *belief state*. A belief state is a probability distribution $b_t : S \to [0, 1]$ over S, such that $b_t(s)$ is the probability that the agent is in state $s \in S$ given the history up to time t. The initial belief state is computed using the initial observation o_0: $b_0(s) \propto \omega(s, o_0)$ for all $s \in S$. The belief state b_{t+1} is then determined from the previous belief state b_t, the executed action a_t, and the resulting observation o_{t+1} as follows:

$$b_{t+1}(s') \propto \omega(s', o_{t+1}) \sum_{s \in S} p(s, a_t, s') b_t(s) \quad \text{for all } s' \in S. \tag{2}$$

---

[2] Technically, the history of interactions should also include the immediate rewards (Izadi and Precup, 2005), this is $h_t = (o_0, a_0, r_1, \ldots, a_{t-1}, r_t, o_t)$. However, we can remove the immediate rewards from the history without loss of generality because it is always possible to include the immediate reward r_t as part of the observation o_t.
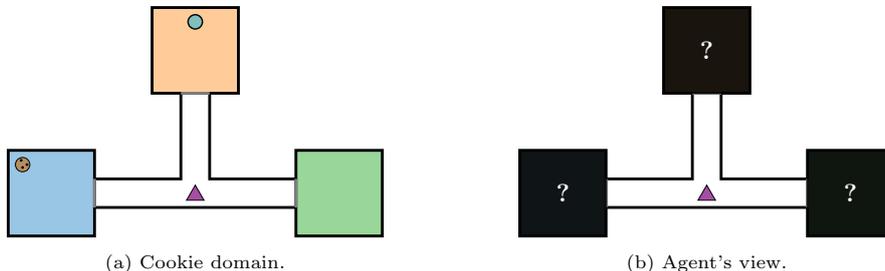
(a) Cookie domain.　　　　　　　　　　(b) Agent's view.

Figure 1: In the cookie domain, the agent can only see what is in the current room.

Since the state transitions and reward function are Markovian w.r.t. b_t, the set of all belief states B can be used to construct the belief MDP $\mathcal{M}_B$, where the states of $\mathcal{M}_B$ are B, A are the actions, the transition probabilities are computed using equation (2), and the reward function r_b is as follows:

$$r_b(b, a) = \sum_{s \in S} r(s, a)b(s). \tag{3}$$

Any optimal policies for $\mathcal{M}_B$ is also optimal for the POMDP (Cassandra et al., 1994).

3. Reward Machines for Partially Observable Environments

In this section, we define RMs for the case of partial observability. We use the following problem as a running example to help explain various concepts.

Example 1 (The cookie domain). *The* cookie domain, *shown in Figure 1a, has three rooms connected by a hallway. The agent (purple triangle) can move in the four cardinal directions. There is a button in the orange room that, when pressed, causes a cookie to randomly appear in the green or blue room. The agent receives a reward of $+1$ for reaching (and thus eating) the cookie and may then go and press the button again. Pressing the button before reaching a cookie will remove the existing cookie and cause a new cookie to randomly appear. There is no cookie at the beginning of the episode. This domain is partially observable since the agent can only see what it is in the room that it currently occupies, as shown in Figure 1b.*

RMs are finite state machines that are used to encode a reward function (Toro Icarte et al., 2018, 2020a). They are defined over a set of propositional symbols $\mathcal{P}$ that correspond to a set of high-level features that the agent can detect using a *labelling function* $L : O_\emptyset \times A_\emptyset \times O \to 2^\mathcal{P}$ where (for any set X) $X_\emptyset \triangleq X \cup \{\emptyset\}$. L assigns truth values to symbols in $\mathcal{P}$ given an environment experience $e = (o, a, o')$ where o' is the observation seen after executing action a when observing o. We use $L(\emptyset, \emptyset, o)$ to assign truth values to the initial observation. We call a truth value assignment of $\mathcal{P}$ a *high-level observation* because it provides a high-level view of the low-level environment observations via the labelling function L. A formal definition of an RM follows:

Definition 2 (reward machine). *Given a set of propositional symbols $\mathcal{P}$, a reward machine is a tuple $\mathcal{R}_\mathcal{P} = \langle U, u_0, \delta_u, \delta_r \rangle$ where U is a finite set of states, $u_0 \in U$ is an initial*

5

state, δ_u is the state-transition function, $\delta_u : U \times 2^{\mathcal{P}} \to U$, and δ_r is the reward-transition function, $\delta_r : U \times 2^{\mathcal{P}} \to \mathbb{R}$.

One way of thinking about RMs is that they decompose problems into a set of high-level states U and define transitions using if-like conditions defined by δ_u. These conditions are over a set of binary properties $\mathcal{P}$ that the agent can detect using L. For example, in the cookie domain, $\mathcal{P} = \{⬤,☺,◯,■,□,■,■\}$. These properties are true in the following situations: ■, □, ■, or ■ is true if the agent is in a room of that color; ⬤ is true if the agent is in the same room as a cookie; ◯ is true if the agent pushed the button with its last action; and ☺ is true if the agent ate a cookie with its last action.

Figure 2 shows three possible reward machines for the cookie domain. We note that these three machines define the same reward signal, 1 for eating a cookie and 0 otherwise, but differ in their states and transitions. As a result, they differ with respect to the amount of information about the current POMDP state that can be inferred from the RM state, as we will see below.

Each RM starts in the initial state u_0. Edge labels in the figures provide a visual representation of the functions δ_u and δ_r. For example, label $\langle ■ ☺, 1 \rangle$ between state u_2 and u_0 in Figure 2b represents $\delta_u(u_2, \{■, ☺\}) = u_0$ and $\delta_r(u_2, \{■, ☺\}) = 1$. Intuitively, this means that if the RM is in state u_2 and the agent just ate a cookie ☺ in room ■, then the agent will receive a reward of 1 and the RM will transition to u_0. Notice that any properties not listed in the label are false (e.g. ⬤ must be false to take the transition labelled $\langle ■ ☺, 1 \rangle$). We also use multiple labels separated by a semicolon (e.g., "$\langle ■, 0 \rangle; \langle ■ ⬤, 0 \rangle$") to describe different conditions for transitioning between the RM states, each with their own associated reward. The label $\langle \text{o/w}, r \rangle$ ("o/w" for "otherwise") on an edge from u_i to u_j means that transition will be made (and reward r received) if none of the other transitions from u_i can be taken.

Let us illustrate the behaviour of an RM using the one shown in Figure 2c. The RM will stay in u_0 until the agent presses the button (causing a cookie to appear), whereupon the RM moves to u_1. From u_1 the RM may move to u_2 or u_3 depending on whether the agent finds a cookie when it enters another room. Finally, the RM moves back to u_0 from u_2 (or u_3) when the agent eats a cookie. Note that it is possible to associate meaning with being in RM states. In the example, u_0 means that there is no cookie available, u_1 means that there is a cookie in some room (either blue or green), u_2 means that the cookie is in the green room, and u_3 means that the cookie is in the blue room.

When learning a policy for a given RM, one simple technique is to learn a policy $\pi(a|o, u)$ that considers the current observation $o \in O$ and the current RM state $u \in U$ to select action $a \in A$. Interestingly, a partially observable problem might be non-Markovian over O, but Markovian over $O \times U$ for some RM $\mathcal{R}_{\mathcal{P}}$. This is the case for the cookie domain with the RM from Figure 2c, for example.

Q-learning for RMs (QRM) is another way to learn a policy by exploiting a given RM (Toro Icarte et al., 2018). QRM learns one Q-function Q_u (i.e., policy) per RM state $u \in U$. Then, given any sample experience, the RM can be used to emulate how much reward would have been received had the RM been in any one of its states. Formally, experience $e = (o, a, o')$ can be transformed into a valid experience $(\langle o, u \rangle, a, \langle o', u' \rangle, r)$ used for updating Q_u for each $u \in U$, where $u' = \delta_u(u, L(e))$ and $r = \delta_r(u, L(e))$. Hence, any off-policy learning method can take advantage of these "synthetically" generated experiences to update all subpolicies simultaneously.
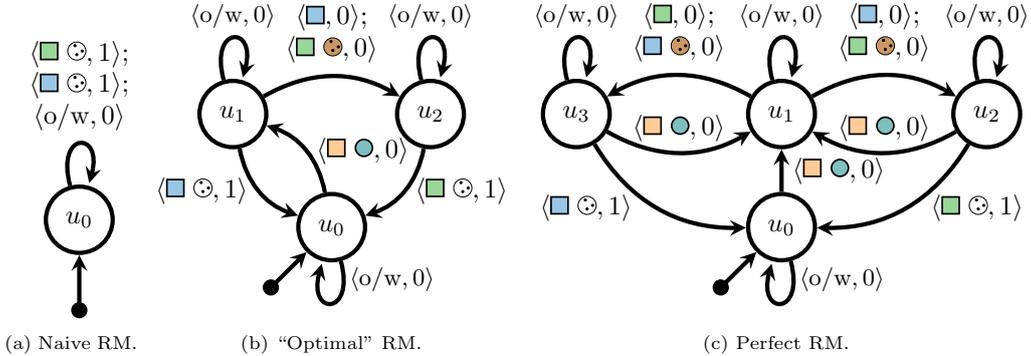
Figure 2: Three possible Reward Machines for the Cookie domain.

When tabular Q-learning is used, QRM is guaranteed to converge to an optimal policy on fully-observable problems (Toro Icarte et al., 2018). However, in a partially observable environment, an experience e might be more or less likely depending on the RM state that the agent was in when the experience was collected. For example, experience e might be possible in one RM state u_i but not in RM state u_j. Thus, updating the policy for u_j using e as QRM does, would introduce an unwanted bias to Q_{u_j}. We will discuss how to (partially) address this problem in Section 6.

4. Learning Reward Machines from Traces

To learn RMs, our overall idea is to search for an RM that can be used as external memory by an agent for solving a partially-observable problem. As input, our method will take a set of high-level propositional symbols $\mathcal{P}$ and a labelling function L that can detect them. Then, the key question is what properties should such an RM have.

Three proposals naturally emerge from the literature. The first comes from the work on learning *finite state machines (FSMs)* (Angluin and Smith, 1983; Zeng et al., 1993; Shvo et al., 2021), which suggests learning the smallest RM that correctly mimics the external reward signal given by the environment, as in Giantamidis and Tripakis' method for learning Moore machines (Giantamidis and Tripakis, 2016). Unfortunately, such approaches would learn RMs of limited utility, like the one in Figure 2a. This naive RM correctly predicts reward in the cookie domain (i.e., +1 for eating a cookie ☺, zero otherwise) but provides no memory in support of solving the task.

The second proposal comes from the literature on learning *finite state controllers (FSC)* (Meuleau et al., 1999) and on *model-free RL* methods (Sutton and Barto, 2018). This work suggests looking for the RM whose optimal policy receives the most reward. For instance, the RM from Figure 2b is "optimal" in this sense. It decomposes the problem into three states. The optimal policy for u_0 goes directly to press the button, the optimal policy for u_1 goes to the blue room and eats the cookie if present, and the optimal policy for u_2 goes to the green room and eats the cookie. Together, these three policies give rise to an optimal policy for the complete problem. This is a desirable property for RMs, but requires computing optimal policies in order to compare the

relative quality of RMs, which seems prohibitively expensive. However, we believe that finding ways to efficiently learn "optimal" RMs is a promising future work direction.

Finally, the third proposal comes from the literature on *predictive state representations (PSRs)* (Littman et al., 2002), *deterministic Markov models (DMMs)* (Mahmud, 2010), and *model-based RL* (Kaelbling et al., 1996). This line of research suggests learning the RM that remembers sufficient information about the history to make accurate Markovian predictions about the next observation. For instance, the cookie domain RM shown in Figure 2c is *perfect* w.r.t. this criterion. Intuitively, every transition in the cookie environment is already Markovian except for transitioning from one room to another. Depending on different factors, when entering into the green room there could be a cookie there (or not). The perfect RM is able to encode such information using 4 states: when at u_0 the agent knows that there is no cookie, at u_1 the agent knows that there is a cookie in the blue or the green room, at u_2 the agent knows that there is a cookie in the green room, and at u_3 the agent knows that there is a cookie in the blue room. Since keeping track of more information will not result in better predictions, this RM is *perfect*. Below, we develop a theory about perfect RMs and describe an approach for learning them from experience.

4.1. Perfect Reward Machines: Formal Definition and Properties

The key insight behind perfect RMs is to use their states U and transitions δ_u to keep track of relevant past information such that the partially observable environment $\mathcal{P}_{\mathcal{O}}$ becomes Markovian with respect to $O \times U$. This is ensured by the following definition.

Definition 3 (perfect reward machine). *An RM $\mathcal{R}_{\mathcal{P}} = \langle U, u_0, \delta_u, \delta_r \rangle$ is considered perfect for a POMDP $\mathcal{P}_{\mathcal{O}} = \langle S, O, A, r, p, \omega, \gamma, \mu \rangle$ w.r.t. a labelling function L iff for every trace $(o_0, a_0, \ldots, o_t, a_t)$ generated by any policy over $\mathcal{P}_{\mathcal{O}}$, the following holds:*

$$\Pr(o_{t+1}, r_t | o_0, a_0, \ldots, o_t, a_t) = \Pr(o_{t+1}, r_t | o_t, x_t, a_t),$$

where $x_0 = u_0$ and $x_t = \delta_u(x_{t-1}, L(o_{t-1}, a_{t-1}, o_t))$.[3]

Two important properties follow from Definition 3. First, if the set of reachable belief states B for the POMDP $\mathcal{P}_{\mathcal{O}}$ is finite, then there exists a perfect RM for $\mathcal{P}_{\mathcal{O}}$. Recall that a belief state models the probability of being at any POMDP state given the previous interactions with the environment. When the model of $\mathcal{P}_{\mathcal{O}}$ is known, we can compute the current belief state using the Bayes rule, as discussed in Section 2. This first property states that if there is a finite set of belief states reachable from the initial state, then there exists a perfect RM for that environment.

Theorem 4. *For any POMDP $\mathcal{P}_{\mathcal{O}}$ with a finite reachable belief space, there exists a perfect RM for $\mathcal{P}_{\mathcal{O}}$.*

Proof. If the reachable belief space B is finite, we can construct an RM that keeps track of the current belief state using one RM state per belief state and emulating their progression using δ_u, and one propositional symbol for every action-observation pair. Thus, the current belief state b_t can be inferred from the last observation, last action, and the current RM state. Hence, the equality from Definition 3 holds. □

---

[3] Note that x_t is the RM state that the agent is in at time t. We will make use of this notation below.

The second property of perfect RMs is that their optimal policies are also optimal for the POMDP. This means that summarizing the history $h_t = (o_0, a_0, \ldots, a_{t-1}, o_t)$ as $\phi(h_t) = \langle u_t, o_t \rangle$, where u_t is the current RM state and o_t is the current observation, results in *globally* optimal policies – i.e., $\pi^*(a_t|h_t) = \pi^*(a_t|u_t, o_t)$.

Theorem 5. *Let $\mathcal{R}_\mathcal{P}$ be a perfect RM for a POMDP $\mathcal{P}_\mathcal{O}$, then any optimal policy for $\mathcal{R}_\mathcal{P}$ w.r.t. the environmental reward is also optimal for $\mathcal{P}_\mathcal{O}$.*

Proof. As the next observation and immediate reward probabilities can be predicted from $O \times U \times A$, a perfect RM for $\mathcal{P}_\mathcal{O}$ also models the belief MDP $\mathcal{M}_B$ for $\mathcal{P}_\mathcal{O}$. As such, optimal policies over $O \times U$ are equivalent to optimal policies over $\mathcal{M}_B$, which are optimal over $\mathcal{P}_\mathcal{O}$ (Cassandra et al., 1994). $\square$

4.2. Perfect Reward Machines: How to Learn Them

We now consider the problem of learning a perfect RM from traces, assuming one exists w.r.t. the given labelling function L. Recall that a perfect RM transforms the original problem into a Markovian problem over $O \times U$. Hence, we should prefer RMs that accurately predict the next observation o' and the immediate reward r from the current observation o, RM state u, and action a. This might be achieved by collecting a training set of traces from the environment, fitting a predictive model for $\Pr(o', r|o, u, a)$, and picking the RM that makes better predictions. However, this approach can be very expensive, especially considering that the observations might be images.

Instead, we propose an alternative that focuses on a necessary condition for a perfect RM: the RM must predict what is *possible* and *impossible* in the environment at the abstract level given by the labelling function. E.g., it is impossible to be at u_3 in the RM from Figure 2c and make the high-level observation $\{\blacksquare, \bullet\}$, because the RM reaches u_3 only if the cookie was seen in the blue room (or not to be in the green room) and it leaves u_3 as soon as the agent eats the cookie (or presses the button).

This idea is formalized in the optimization model LRM. Let $\mathcal{T} = \{\mathcal{T}_0, \ldots, \mathcal{T}_n\}$ be a set of traces, where each trace $\mathcal{T}_i$ is a sequence of observations, actions, and rewards:

$$\mathcal{T}_i = (o_{i,0}, a_{i,0}, r_{i,1}, \ldots, a_{i,t_i-1}, r_{i,t_i}, o_{i,t_i}). \tag{4}$$

We now look for an RM $\langle U, u_0, \delta_u, \delta_r \rangle$ that can be used to predict $L(e_{i,t+1})$ from $L(e_{i,t})$ and the current RM state $x_{i,t}$, where $e_{i,t+1}$ is the experience $(o_{i,t}, a_{i,t}, o_{i,t+1})$ and we define $e_{i,0}$ as $(\emptyset, \emptyset, o_{i,0})$. The model parameters are the set of traces $\mathcal{T}$, the set of propositional symbols $\mathcal{P}$, the labelling function L, and a maximum number of states in the RM $u_{\max}$. The model also uses the sets $I = \{0 \ldots n\}$, $T_i = \{0 \ldots t_i-1\}$, and Σ, where I contains the indices of the traces, T_i their time steps, and Σ contains all the high-level observations that appear in $\mathcal{T}$. The model has two auxiliary variables $x_{i,t}$ and $N_{u,\sigma}$. Variable $x_{i,t} \in U$ represents the state of the RM after observing trace $\mathcal{T}_i$ up to time t. Variable $N_{u,\sigma} \subseteq 2^\Sigma$ is the set of all the next high-level observations seen from the RM state u and the high-level observations σ in $\mathcal{T}$. In other words, $\sigma' \in N_{u,\sigma}$ iff $u = x_{i,t}$, $\sigma = L(e_{i,t})$, and $\sigma' = L(e_{i,t+1})$ for some trace $\mathcal{T}_i$ and time t.

$$\underset{\langle U, u_0, \delta_u, \delta_r \rangle}{\text{minimize}} \sum_{i \in I} \sum_{t \in T_i} \log(|N_{x_{i,t}, L(e_{i,t})}|) \tag{LRM}$$

$$s.t. \ \langle U, u_0, \delta_u, \delta_r \rangle \in \mathcal{R}_\mathcal{P} \tag{5}$$

$$|U| \leq u_{\max} \tag{6}$$

$$x_{i,t} \in U \qquad \forall i \in I, t \in T_i \cup \{t_i\} \tag{7}$$

$$x_{i,0} = u_0 \qquad \forall i \in I \tag{8}$$

$$x_{i,t+1} = \delta_u(x_{i,t}, L(e_{i,t+1})) \qquad \forall i \in I, t \in T_i \tag{9}$$

$$N_{u,\sigma} \subseteq 2^\Sigma \qquad \forall u \in U, \sigma \in \Sigma \tag{10}$$

$$L(e_{i,t+1}) \in N_{x_{i,t}, L(e_{i,t})} \qquad \forall i \in I, t \in T_i \tag{11}$$

Constraints (5) and (6) ensure that we find a well-formed RM over $\mathcal{P}$ with at most $u_{\max}$ states. Constraints (7), (8), and (9) ensure that $x_{i,t}$ is equal to the current state of the RM, starting from u_0 and following δ_u. Constraints (10) and (11) ensure that the sets $N_{u,\sigma}$ contain every $L(e_{i,t+1})$ that has been seen right after σ and u in $\mathcal{T}$. The objective function comes from maximizing the log-likelihood for predicting $L(e_{i,t+1})$ using a uniform distribution over all the possible options given by $N_{u,\sigma}$.

A key property of this formulation is that any perfect RM is optimal w.r.t. the objective function in LRM when the number of traces tends to infinity:

Theorem 6. *When the set of training traces (and their lengths) tends to infinity and is collected by a policy such that $\pi(a|o) > \epsilon$ for all $o \in O$ and $a \in A$, any perfect RM with respect to L and at most u_{max} states will be an optimal solution to the formulation LRM.*

Proof. In the limit, $\sigma' \in N_{u,\sigma}$ if and only if the probability of observing σ' after executing an action from the RM state u while observing σ is non-zero. In particular, for all $i \in I$ and $t \in T$, the cardinality of $N_{x_{i,t}, L(e_{i,t})}$ will be minimal for a perfect RM. This property follows from the fact that perfect RMs make perfect predictions for the next observation o' given o, u, and a. Therefore, as we minimize the sum over $\log(|N_{x_{i,t}, L(e_{i,t})}|)$, the objective value for any perfect RM must be minimal. $\qquad \square$

Finally, note that the definition of a perfect RM does not impose conditions over the rewards associated with the RM (i.e., δ_r). This is why δ_r is a free variable in the model LRM. However, in order to apply methods that exploit RM structure (such as QRM), we still need δ_r to model the external reward signals given by the environment. To do so, we estimate $\delta_r(u, \sigma)$ using its empirical expectation over $\mathcal{T}$ – as commonly done when constructing belief MDPs (Cassandra et al., 1994). Formally,

$$\delta_r(u, \sigma) = \frac{\sum_{i \in I, t \in T_i} r_{i,t+1} 1_{u=x_{i,t} \wedge \sigma = L(e_{t+1})}}{\sum_{i \in I, t \in T_i} 1_{u=x_{i,t} \wedge \sigma = L(e_{t+1})} + \epsilon} \quad \text{for all } u \in U \text{ and } l \in 2^\mathcal{P}, \tag{12}$$

where $1_c = 1$ if condition c holds (zero otherwise) and ϵ is a small constant.

5. Searching for a Perfect Reward Machine

We now describe different approaches to solve LRM. These include a *mixed integer linear programming (MILP)* model, a *constrained programming (CP)* model, and two *local search (LS)* models. All our models are guaranteed to find optimal solutions given sufficient resources. But first, let us introduce two preprocessing steps over the training traces that our models use: *trace compression* and *prefix trees (PTs)*.

5.1. Trace Compression

Recall that LRM works over the high-level observations σ_t given by the labelling function $L(e_t)$, where e_t represents the experience (o_t, a_t, o_{t+1}) at time t. Thus, the first step is to transform each trace $\mathcal{T}_i$ from being a sequence of interactions with the environment into a sequence of truth value assignments of $\mathcal{P}$ via L:

$$\mathcal{T}_i = (o_{i,0}, a_{i,0}, r_{i,1}, \ldots, a_{i,t_i-1}, r_{i,t_i}, o_{i,t_i}) \xRightarrow{\sigma_t = L(e_t)} \tau_i = (\sigma_{i,0}, \sigma_{i,1}, \ldots, \sigma_{i,t_i}).$$

In the abstract space given by $\mathcal{P}$ and L, it is usually the case that the same high-level observation is seen many times in a row. For instance, a typical high-level trace in the cookie domain would look as follows:

$$\tau = (\square, \square, \square, \square, \blacksquare, \blacksquare, \blacksquare\bigcirc, \blacksquare, \blacksquare, \square, \square, \square, \square, \square, \square, \square, \blacksquare, \square, \square, \square, \square, \square, \square, \square, \square, \blacksquare\circledast).$$

This trace indicates that the agent was first at the hallway ($\square$), and then it moved to the orange room ($\blacksquare$) and pressed the button ($\blacksquare\bigcirc$). After that, it returned to the hallway ($\square$), noticed that there was no cookie in the blue room ($\blacksquare$), came back to the hallway ($\square$), and finally found a cookie in the green room ($\blacksquare\circledast$). Let us assume that the trace ended there for simplicity. As you can see, the most informative moments are when the high-level observations change. Indeed, knowing that the agent stayed at the hallway during 4, 7, or 8 steps is not particularly useful in this case. This suggests that we could potentially compress the training trace (without losing relevant information) by removing duplicated high-level observations that appear consecutively in the trace. For instance, the previous trace will look as follows after being compressed:

$$\tau_c = (\square, \blacksquare, \blacksquare\bigcirc, \blacksquare, \square, \blacksquare, \square, \blacksquare\circledast).$$

Compressing the traces is an optional preprocessing step that has advantages and disadvantages. The advantage is that it considerably improves the quality of the RMs that the models find given a fixed computational budget. Intuitively, any model has to go over the traces to evaluate how good an RM is. Compressing the traces reduces the computational cost of doing so. The disadvantage is that, by compressing the traces, we are assuming that observing two or more times the same high-level observation consecutively does not provide further information about the current POMDP state. If that is the case, then we can compress the traces and the model will still find optimal solutions for LRM. If that is not the case, then compressing the traces might help the models to find high-quality RMs faster but they will not find optimal solutions.

Two inconsistencies might arise if we compress the traces. First, note that we are learning RMs using compressed traces but then testing them over uncompressed traces. This is a problem because the optimal solution for LRM will exploit the fact that no training trace has the same high-level observation twice in a row. However, when the agent uses the learned RM in the environment (i.e., at test time), it might encounter the same high-level observation many times in a row and, thus, update the RM state in ways that were unintended by LRM. For that reason, if we do compress the traces, we have to include the following additional constraint to LRM:

$$[\delta_u(u', \sigma) = u] \Rightarrow [\delta_u(u, \sigma) = u] \qquad \forall u, u' \in U, \sigma \in \Sigma \qquad (13)$$

Figure 3: A PT for $\{(b), (ab), (ac)\}$. We took this example from Giantamidis and Tripakis (2016).

This constraint ensures that the learned reward machine will not enter and leave a state u using the same high-level observation σ.

The second inconsistency relates to constraint (9). According to that constraint, the second high-level observation of the trace is used to progress the initial state in the reward machine (and not the first one). Indeed, LRM dictates that $x_0 = u_0$ and $x_1 = \delta_u(u_0, \square)$ for τ_c. However, if we were processing the uncompressed version of τ_c (i.e., τ), then the initial RM state would have been updated using $x_1 = \delta_u(u_0, \square)$ instead of $x_1 = \delta_u(u_0, \square)$ because $\square$ appears many times in a row at the beginning of τ. We can solve this inconsistency by not compressing the first high-level observation of the trace. This is, to always include the first high-level observation and only compress from the second high-level observation forward. In our example, the proper manner of compressing τ would be as follows:

$$\tau'_c = (\square, \square, \square, \square\bullet, \square, \square, \square, \square, \square\odot).$$

5.2. Prefix Trees (PTs)

Regardless of whether we compress or not the training traces, from now on we will be working with a set of $n + 1$ traces $\mathcal{T} = \{\tau_0, \ldots, \tau_n\}$, where each trace is composed of a sequence of high-level observations: $\tau_i = (\sigma_{i,0}, \ldots, \sigma_{i,t_i})$. Then, LRM defines an independent variable $x_{i,t}$ which models the current RM state at time step t given the trace τ_i for all $i \in I$ and $t \in T_i$. However, doing so does not exploit the fact that there exist large sets of $x_{i,t}$ variables whose values are equivalent for any reward machine. These are cases where the prefix of two traces τ_i and τ_j are identical up to some time step t. Indeed, we know that variables $x_{i,t} = x_{j,t}$ if $\sigma_{i,t'} = \sigma_{j,t'}$ for all $t' < t$ because the transitions of a reward machine are deterministic. We can compactly capture this information using *prefix trees (PTs)* (De la Higuera, 2010).

PTs merge all the training traces into one large tree, where each trace becomes a branch in this tree. As an example, Figure 3 shows the PT for the following set of traces: $\tau_1 = (b)$, $\tau_2 = (a, a)$, and $\tau_3 = (a, b)$. Each node in a PT represents a prefix that appears in one or more training traces. In the example, there are 5 nodes corresponding to the 5 possible prefixes in $\{\tau_1, \tau_2, \tau_3\}$: ϵ, (a), (b), (a, b), and (a, c), where ϵ represents the empty trace. Thus, instead of assigning RM states to each variable $x_{i,t}$, our models assign RM states to each node in the PT. That way, these models are forced to assign the same sequence of RM states to all traces as long as their prefixes are identical. For instance,

12

Algorithm 1 Converting Training Data into Prefix Trees

1: **Input:** $\{\tau_0, \ldots, \tau_n\}$
2: root_node $\leftarrow$ create_root_node()
3: **for** $i = 0$ **to** n **do**
4:     node $\leftarrow$ root_node
5:     **for** $t = 0$ **to** $|\tau_i| - 1$ **do**
6:         node.increase_trace_counter_by_one()
7:         **if not** node.has_child($\sigma_{i,t}$) **then**
8:             node.add_child($\sigma_{i,t}$)
9:         node $\leftarrow$ node.get_child($\sigma_{i,t}$)
10: **return** root_node
---

assigning an RM state to node n_a in the PT will automatically assign the same RM state to $x_{2,1}$ and $x_{3,1}$ because n_a represents the current RM state after any trace observes a at the initial time step and, in this case, 'a' is the first observation of τ_2 and τ_3.

Algorithm 1 shows the pseudo-code to constructing a PT from a given set of traces $\{\tau_0, \ldots, \tau_n\}$. Starting at the root node, the code goes over all the training traces and adds them as branches of the tree (lines 7-8). We also count how many training traces pass through each node in the tree (line 6). This information helps us defining the right weights to penalize predictions in the objective function when reformulating LRM from assigning RM states to time steps to assigning RM states to nodes in the PT.

5.3. Solving LRM via Mixed Integer Linear Programming (MILP)

In this section, we present a MILP model for LRM. MILP solvers are able to solve optimization problems with linear constraints and objective functions. They can also handle continuous and discrete variables. MILP solvers are the state of the art for solving a wide range of discrete optimization problems and they are guaranteed to find optimal solutions given sufficient resources (Jünger et al., 2009).

Our MILP model receives as input $u_{\max}$ and the PT built using the training traces $\mathcal{T} = \{\tau_0, \ldots, \tau_n\}$. Note that the traces in $\mathcal{T}$ might or might not be compressed. We use the following notation to refer to the different elements in the PT: n_{root} is its root node, S is a set containing all the nodes in PT except for n_{root}, $S_{\mathrm{in}} \subset S$ is a subset of S that only contains the inner nodes of PT, w_n is the number of training traces that pass through node n, $p(n)$ is the parent of node n, $o(n)$ is the high-level observation associated with the edge between nodes $p(n)$ and n, and $C(n)$ is the set of children of node n. Also, this model assumes that the set Σ contains all the high-level observations that appear in $\mathcal{T}$, $|U| = u_{\max}$, and $K = 2^{|\Sigma|}$.

The idea behind our MILP model is to assign RM states to each node in the PT. Then, we look for an assignment that (i) can be produced by a deterministic machine and (ii) optimizes the same objective as LRM. To achieve this, we use the following decision variables. Variable $x_{n,u} \in \{0, 1\}$ indicates if node $n \in S \cup \{n_{\mathrm{root}}\}$ is assigned the RM state $u \in U$. Variable $d_{u,\sigma,u'} \in \{0, 1\}$ represents the possible transition from state $u \in U$ to state $u' \in U$ given observation $\sigma \in \Sigma$ in the RM. Formally, this means that $d_{u,\sigma,u'} = 1$ iff $\delta_u(u, \sigma) = u'$. Variable $p_{u,\sigma,\sigma'} \in \{0, 1\}$ indicates if $\sigma' \in \Sigma$ is a possible next observation at RM state $u \in U$ when observing $\sigma \in \Sigma$, where $p_{u,\sigma,\sigma'} = 1$ iff $\sigma' \in N_{u,\sigma}$. Variable

$y_{u,\sigma,m} \in \{0,1\}$ represents the cardinality of $N_{u,\sigma}$, meaning that $y_{u,\sigma,m} = 1$ iff $|N_{u,\sigma}| = m$. Lastly, variables z_n represent the log-likelihood cost for the predictions associated with node $n \in S$. The full model is then as follows:

$$\min \sum_{n \in S} w_n \cdot z_n \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(MILP)}$$

$$s.t. \ z_n \geq \sum_{m=1}^{K} y_{u,\sigma,m} \log(m) - (1 - x_{n,u}) \log(K) \qquad \forall n \in S, u \in U, \sigma = o(n) \quad (14)$$

$$\sum_{m=1}^{K} y_{u,\sigma,m} = 1 \qquad\qquad\qquad\qquad\qquad \forall u \in U, \sigma \in \Sigma \quad (15)$$

$$\sum_{\sigma' \in \Sigma} p_{u,\sigma,\sigma'} = \sum_{m=1}^{K} y_{u,\sigma,m} \cdot m \qquad\qquad\qquad \forall u \in U, \sigma \in \Sigma \quad (16)$$

$$p_{u,o(n),o(n')} \geq x_{n,u} \qquad\qquad\qquad \forall u \in U, n \in S_{\text{in}}, n' \in C(n) \quad (17)$$

$$\sum_{u' \in U} d_{u,\sigma,u'} = 1 \qquad\qquad\qquad\qquad \forall u \in U, \sigma \in \Sigma \quad (18)$$

$$\sum_{u \in U} x_{n,u} = 1 \qquad\qquad\qquad\qquad\qquad \forall n \in S, u \in U \quad (19)$$

$$x_{n,u_0} = 1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad n = n_{\text{root}} \quad (20)$$

$$x_{p(n),u} + x_{n,u'} - 1 \leq d_{u,o(n),u'} \qquad\qquad \forall u, u' \in U, n \in S \quad (21)$$

$$d_{u,\sigma,u'} \leq d_{u',\sigma,u'} \qquad\qquad \forall u, u' \in U, u \neq u', \sigma = \Sigma \quad (22)$$

$$x_{n,u} \in \{0,1\} \qquad\qquad\qquad \forall n \in S \cup \{n_{\text{root}}\}, u \in U \quad (23)$$

$$d_{u,\sigma,u'} \in \{0,1\} \qquad\qquad\qquad \forall u, u' \in U, \sigma \in \Sigma \quad (24)$$

$$p_{u,\sigma,\sigma'} \in \{0,1\} \qquad\qquad\qquad \forall u \in U, \sigma, \sigma' \in \Sigma \quad (25)$$

$$y_{u,\sigma,m} \in \{0,1\} \qquad\qquad \forall u \in U, \sigma \in \Sigma, m \in \{1..K\} \quad (26)$$

$$z_n \geq 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall n \in S \quad (27)$$

The objective function of MILP is a sum over the prediction costs at each node in the tree weighted by how many traces pass through that node. Constraint (14) models the log-likelihood cost for each node in the tree. Constraints (15) and (16) compute the cardinality of $N_{u,l}$. Constraint (17) defines the possible predictions given an RM state and high-level observation. Constraint (18) enforces that for each RM state a high-level observation can lead to exactly one other RM state. Constraint (19) enforces that exactly one RM state is assigned to each node in the tree. Constraint (20) assigns the initial RM state to the root node, and constraint (21) enforces that there exists a deterministic δ_u that can produce the assignment of RM states to tree nodes. Constraints (23)-(27) correspond to the variables' domains. Finally, we note that constraint (22) is an optional constraint that should be included only if the training traces were compressed. This constraint enforces that the RM state does not change after observing the same high-level observation two times consecutively.

5.4. Solving LRM via Constrained Programming (CP)

CP is another technique for solving discrete optimization problems. CP is less restrictive than MILP in the type of variables and constraints that it can handle. For instance, our MILP model had to include many auxiliary decision variables (e.g., $x_{n,u}$, $p_{u,\sigma,\sigma'}$, $y_{u,\sigma,m}$, and z_n) in order to linearize different aspects of LRM. In contrast, our CP model only uses one set of decision variables $d_{u,\sigma}$ to model $\delta_u(u,\sigma)$ and all other elements from LRM are defined w.r.t. those variables. In addition, CP solvers are also guaranteed to find optimal solutions given enough resources (Rossi et al., 2006).

Our CP model receives the same inputs as our MILP model, including the PT constructed using the (potentially compressed) training traces $\mathcal{T}$. Recall that the different elements in the PT are referred as follows: n_{root} is the root node, S is the set containing all nodes but n_{root}, w_n is the number of training traces that pass through node n, $p(n)$ is the parent of node n, $o(n)$ is the high-level observation between $p(n)$ and n, and $C(n)$ is the set of children of node n. The model also uses the set $U = \{0...u_{\max} - 1\}$, the set Σ with all the high-level observations in $\mathcal{T}$, and the set $S(\sigma, \sigma')$ containing all the nodes where σ' is observed immediately after observing σ:

$$S(\sigma, \sigma') = \{n \mid n \in S, n' \in C(n), \sigma = o(n), \sigma' = o(n')\}.$$

As we previously mentioned, the only decision variables are $d_{u,\sigma} \in U$ for all $u \in U$ and $\sigma \in \Sigma$. Note that $d_{u,\sigma}$ is an integer variable that goes from 0 to $u_{\max} - 1$ and it models the output of $\delta_u(u,\sigma)$ – i.e., if the RM state is u and the agent observes σ, then the next RM state will be $d_{u,\sigma}$. With that, the complete model is as follows:

$$\min \sum_{n \in S} w_n \cdot \texttt{log}(y_{x_n, o(n)}) \tag{CP}$$

$$s.t.\ y_{u,\sigma} \doteq \sum_{\sigma' \in \Sigma} p_{u,\sigma,\sigma'} \qquad\qquad \forall u \in U, \sigma \in \Sigma \tag{28}$$

$$p_{u,\sigma,\sigma'} \doteq \texttt{logical\_or}\left(\{x_n = u \mid n \in S(\sigma, \sigma')\}\right) \qquad \forall u \in U, \sigma, \sigma' \in \Sigma \tag{29}$$

$$x_n \doteq d_{u,o(n)} \qquad\qquad \forall n \in S, u = x_{p(n)} \tag{30}$$

$$x_n \doteq 0 \qquad\qquad n = n_{\text{root}} \tag{31}$$

$$\texttt{if\_then}(d_{u,\sigma} = u', d_{u',\sigma} = u') \qquad\qquad \forall u, u \in U, \sigma \in \Sigma \tag{32}$$

$$d_{u,\sigma} \in U \qquad\qquad \forall u \in U, \sigma \in \Sigma \tag{33}$$

This model uses the formalism and global constraints available in IBM ILOG CP Optimizer (IBM, 2018). Its only decision variables are $d_{u,\sigma}$, defined in constraint (33). Note that the domain of $d_{u,\sigma} \in U$ forces the RM to be deterministic, since there is exactly one possible transition for each $u \in U$ and $\sigma \in \Sigma$. Using $d_{u,\sigma}$, the model defines three auxiliary *CP expressions* in equations (28)-(31). Expression $y_{u,\sigma}$ represents the cardinality of the prediction set $N_{u,\sigma}$ after observing $\sigma \in \Sigma$ from the RM state $u \in U$. Expression $p_{u,\sigma,\sigma'}$ is one if and only if it is possible to observe $\sigma' \in \Sigma$ from the RM state $u \in U$ after observing $\sigma \in \Sigma$ (and zero otherwise). This expression uses a logical OR constraint, $\texttt{logical\_or}(Z)$ which returns 1 iff at least one element $z \in Z$ is true. Expression $x_n \in U$ indicates that the RM state assigned to the tree node n. Finally, the objective function is a weighted sum over the prediction errors and constraint (32) is an optional constraint

15

Figure 4: Local search approaches start from some feasible solution and iteratively move to the best solution within its neighbourhood. This process repeats until reaching a locally optimal solution.

used if the traces were compressed. This constraint enforces a self loop $\delta_u(u', \sigma) = u'$ if $\delta_u(u, \sigma) = u'$ for some $u \in U$.

5.5. Solving LRM via Local Search (LS) and Tabu Search (TS)

Finally, here we present our local search methods. We note that MILP and CP are known as *exact methods*. They incrementally construct a search tree where each branch represents a feasible solution to the problem and use different relaxation and propagation rules to prune this tree as much as possible. This approach allows them to find optimal solutions and prove that those solutions are optimal for small to medium size problems. However, they struggle when facing large problems because the size of the tree grows exponentially with the number of variables and computing relaxations and propagation rules become more expensive with the number of constraints.

When solving large scale problems, the best results are often obtained by *heuristic* methods. Heuristic methods propose polynomial-time approximations to solve NP-hard problems. They favor finding good solutions over providing strong optimality guarantees. Here, we explore two local search methods (Aarts et al., 2003).

Figure 4 shows how local search works. In the figure, each point inside the rectangle represents a feasible solution to the problem. Local search starts from a feasible solution and evaluates its objective function. Then, it evaluates the objective function of all the solutions near the current solution and then moves to the best solution within that region. This step is represented by a violet arrow in the figure. The process then repeats until a locally optimal solution is reached, where no neighbouring solution is better than the current solution.

Unfortunately, local search can converge to locally optimal solutions that might be far from a globally optimal solution. To deal with this issue, one option is to *restart* local search when it finds a locally optimal solution and start over from a different initial solution. Another option is to use *tabu search* (Glover and Laguna, 1998). Tabu search is a local search approach that saves the last n solutions in a *tabu list* and always moves to the best neighbour that is not in the list. This allows tabu search to escape locally optimal solutions. Here, we explore these two options for solving LRM.

Algorithm 2 shows a local search approach with restarts to learn an RM. This algorithm receives the set of high-level observations Σ, the training traces $\mathcal{T}$, the maximum number of RM state $u_{\max}$, and some termination criteria such as a time limit or a maximum number of steps $t_{\max}$. The algorithm starts from a randomly generated RM (line

16

Algorithm 2 A local search approach with restarts to solve LRM
___
1: **Input:** Σ, $\mathcal{T}$, $u_{\max}$, $t_{\max}$
2: $t \leftarrow 0$, $c^* \leftarrow \infty$, $\mathcal{R}^* \leftarrow$ None
3: **while** $t \leq t_{\max}$ **do**
4:     $c_p \leftarrow \infty$
5:     $\mathcal{R} \leftarrow$ sample_a_reward_machine($\mathcal{T}$, Σ, $u_{\max}$)
6:     $c \leftarrow$ evaluate_reward_machine($\mathcal{R}$, $\mathcal{T}$)
7:     **if** $c < c^*$ **then**
8:         $c^* \leftarrow c$, $\mathcal{R}^* \leftarrow \mathcal{R}$
9:     **while** $t \leq t_{\max}$ **and** $c < c_p$ **do**
10:         $t \leftarrow t + 1$, $c_p \leftarrow c$
11:         $\mathcal{N} \leftarrow$ get_neighbours($\mathcal{R}$, Σ, $u_{\max}$)
12:         **for** $\mathcal{R}_n \in \mathcal{N}$ **do**
13:             $c_n \leftarrow$ evaluate_reward_machine($\mathcal{R}_n$, $\mathcal{T}$)
14:             **if** $c_n < c$ **then**
15:                 $c \leftarrow c_n$, $\mathcal{R} \leftarrow \mathcal{R}_n$
16:         **if** $c < c^*$ **then**
17:             $c^* \leftarrow c$, $\mathcal{R}^* \leftarrow \mathcal{R}$
18: **return** $\mathcal{R}^*$
___

5). The initial RM is sampled from a uniform distribution. That is, every RM with at most $u_{\max}$ states can be selected with equal probability. On each iteration, the algorithm evaluates all neighbouring RMs (lines 9-17). We define the neighbourhood of an RM as the set of RMs that differ by exactly one transition (i.e., removing/adding a transition, or changing its value) and evaluate RMs using the objective function of LRM. When all neighbouring RMs are evaluated, the algorithm moves to the neighbouring RM with the lowest objective value (line 15), and the process repeats. If at any point a locally optimal solution is reached, then the algorithm starts over from another randomly generated RM (line 9). Finally, the best RM seen so far is returned when the terminal condition is met (line 18).

Algorithm 3 shows a tabu search approach to learn a reward machine. It has the same inputs as local search, but it also receives the size of the tabu list τ_{size}. Our tabu search method is identical to Algorithm 2 except that tabu search initializes the tabu list in line 2, adds the current RM to the tabu list in line 11, and moves to the best solution that is not in the tabu list in lines 12-18.

We note that, technically, these two methods will eventually find an optimal solution. The reason is that both methods restart the search when they get stuck: Local search restarts when it reaches a locally optimal solution and tabu search restarts when it reaches a neighbourhood where all the RMs are in the tabu list. Thus, these methods either find an optimal solution during the search or restart the search.[4] Since every RM can be sampled with equal probability when restarting, both methods will eventually sample an optimal solution (or find one). That said, the space of possible RMs is so vast that we cannot expect our implementations of local search and tabu search to necessarily find optimal solutions in practice.

___
[4]In the case of tabu search, we are assuming that the tabu list is large enough.

Algorithm 3 A tabu search approach to solve `LRM`

1: **Input:** Σ, $\mathcal{T}$, $u_{\max}$, $t_{\max}$, τ_{size}
2: $\tau \leftarrow$ initialize_tabu_list(τ_{size})
3: $t \leftarrow 0$, $c^* \leftarrow \infty$, $\mathcal{R}^* \leftarrow$ None
4: **while** $t \leq t_{\max}$ **do**
5: $\quad \mathcal{R} \leftarrow$ sample_a_reward_machine$(\mathcal{T}, \Sigma, u_{\max})$
6: $\quad c \leftarrow$ evaluate_reward_machine$(\mathcal{R}, \mathcal{T})$
7: $\quad$ **if** $c < c^*$ **then**
8: $\quad\quad c^* \leftarrow c$, $\mathcal{R}^* \leftarrow \mathcal{R}$
9: $\quad$ **while** $t \leq t_{\max}$ **and** $\mathcal{R} \notin \tau$ **do**
10: $\quad\quad t \leftarrow t + 1$, $c \leftarrow \infty$
11: $\quad\quad \tau \leftarrow$ add_reward_machine_to_tabu_list$(\tau, \mathcal{R})$
12: $\quad\quad \mathcal{N} \leftarrow$ get_neighbours$(\mathcal{R}, \Sigma, u_{\max})$
13: $\quad\quad$ **for** $\mathcal{R}_n \in \mathcal{N} \setminus \tau$ **do**
14: $\quad\quad\quad c_n \leftarrow$ evaluate_reward_machine$(\mathcal{R}_n, \mathcal{T})$
15: $\quad\quad\quad$ **if** $c_n < c$ **then**
16: $\quad\quad\quad\quad c \leftarrow c_n$, $\mathcal{R} \leftarrow \mathcal{R}_n$
17: $\quad\quad$ **if** $c < c^*$ **then**
18: $\quad\quad\quad c^* \leftarrow c$, $\mathcal{R}^* \leftarrow \mathcal{R}$
19: **return** $\mathcal{R}^*$

---

6. Simultaneously Learning a Reward Machine and a Policy

We now describe our overall approach to simultaneously finding an RM and exploiting that RM to learn a policy. Algorithm 4 shows the complete pseudo-code. Our approach starts by collecting a training set of traces $\mathcal{T}$ generated by following a random policy during t_{w} "warmup" steps (line 2). This set of traces is used to find an initial RM $\mathcal{R}$ using one of our discrete optimization models (line 3). The algorithm then sets the RM state to u_0, sets the current high-level observation σ to $L(\emptyset, \emptyset, o)$, and initializes the policy π (lines 4-5). The standard RL loop is then followed (lines 6-19): an action a is selected according to $\pi(a|o, u)$ and the agent receives the next observation o' and the immediate reward r. The RM state is then updated to $u' = \delta_u(u, L(o, a, o'))$ and the last experience (o, u, a, r, o', u') is used to update π. Finally, the environment and RM are reset if a terminal state is reached (lines 17-18).

If on any step, there is evidence that the current RM might not be perfect, our approach will attempt to find a new one (lines 11-16). Recall that the RM $\mathcal{R}$ was selected using the cardinality of its prediction sets N, where $N_{u,\sigma}$ is the set of high-level observations seen from the RM state u immediately after observing σ in the training data. If the current high-level observation σ' is not in $N_{u,\sigma}$, then adding the current trace to $\mathcal{T}$ will increase the size of $N_{u,\sigma}$ for $\mathcal{R}$ and, in consequence, $\mathcal{R}$ may no longer be the best RM. Therefore, if $\sigma' \notin N_{u,\sigma}$, we add the current trace to $\mathcal{T}$ and learn a new RM. Our method only uses the new RM if its cost is lower than $\mathcal{R}$'s and, if the RM is updated, a new policy is learned from scratch (lines 14-16).

Given the current RM, we can use any RL algorithm to learn a policy $\pi(a|o, u)$, by treating the combination of o and u as the current state. If the RM is perfect, then the optimal policy $\pi^*(a|o, u)$ will also be optimal for the original POMDP (as stated in

---
Algorithm 4 Algorithm to simultaneously learn a reward machine and a policy
---
1: **Input:** $\mathcal{P}$, L, $u_{\max}$, t_{w}
2: $\mathcal{T} \leftarrow$ collect_traces(t_{w})
3: $\mathcal{R}, N \leftarrow$ learn_rm($\mathcal{P}$, L, $\mathcal{T}$, $u_{\max}$)
4: $o \leftarrow$ env_get_initial_state(), $u \leftarrow u_0$, $\sigma \leftarrow L(\emptyset, \emptyset, o)$
5: $\pi \leftarrow$ initialize_policy()
6: **for** $t = 1$ **to** t_{train} **do**
7:     $a \leftarrow$ select_action(π, o, u)
8:     $o', r,$ done $\leftarrow$ env_execute_action(a)
9:     $u', \sigma' \leftarrow \delta_u(u, L(o, a, o')), L(o, a, o')$
10:     $\pi \leftarrow$ improve(π, o, u, σ, a, r, o', u', σ', done, N)
11:     **if** $\sigma' \notin N_{u,\sigma}$ **then**
12:         $\mathcal{T} \leftarrow \mathcal{T} \cup$ get_current_trace()
13:         $\mathcal{R}', N \leftarrow$ relearn_rm($\mathcal{R}$, $\mathcal{P}$, L, $\mathcal{T}$, $u_{\max}$)
14:         **if** $\mathcal{R} \neq \mathcal{R}'$ **then**
15:             done $\leftarrow$ **true**, $\mathcal{R} \leftarrow \mathcal{R}'$
16:             $\pi \leftarrow$ initialize_policy()
17:     **if** done **then**
18:         $o' \leftarrow$ env_get_initial_state(), $u' \leftarrow u_0$, $\sigma' \leftarrow L(\emptyset, \emptyset, o)$
19:     $o \leftarrow o'$, $u \leftarrow u'$, $\sigma \leftarrow \sigma'$
20: **return** π

Theorem 5). In this case, we can ignore the reward δ_r that comes from the RM and only consider the reward received directly from the environment. However, to further exploit the problem structure exposed by the RM (such as with QRM), we need to set δ_r. We do so using the empirical average, as described in Section 4.2.

Let us now explain how we incorporate QRM into this process. As explained in Section 3, standard QRM under partial observability can introduce a bias because an experience $e = (o, a, o')$ might be more or less likely depending on the RM state that the agent was in when the experience was collected. We partially address this issue by updating Q_u using (o, a, o') iff $L(o, a, o') \in N_{u,\sigma}$, where σ was the current high-level observation that generated the experience (o, a, o'). Hence, we do not transfer experiences from u_i to u_j if the current RM does not believe that (o, a, o') is possible in u_j. For example, consider the cookie domain and the perfect RM from Figure 2c. If some experience consists of entering to the green room and seeing a cookie, then this experience will not be used by states u_0 and u_3 as it is impossible to observe a cookie at the green room from those states. While adding this rule works in many cases, it does not fully address the problem. We further discuss this issue in Section 8.

7. Experimental Evaluation

In this section, we provide an empirical evaluation of our method in three partially observable environments. Our evaluation consists of two parts. First, we compare the effectiveness of our mixed integer linear programming model (MILP), our constrained programming model (CP), our local search with restart algorithm (LS), and our tabu

search method (TS) to solve `LRM`. We then show how the combination of learning an RM and a policy using double DQN (LRM+DDQN) and deep QRM (LRM+DQRM) compares to different baselines. As a brief summary, our results show the following:

1. MILP and CP find optimal solutions for small instances of `LRM`.
2. LS and TS find better solutions than MILP and CP for large instances of `LRM`.
3. LS consistently finds better solutions than TS.
4. Our LRM-based methods can outperform A3C, ACER, PPO, and DDQN.
5. LRM+DQRM learns faster then LRM+DDQN, but it is less stable.

7.1. Domains

We tested our approach on three partially observable domains, shown in Figure 5. These environments consist of three rooms connected by a hallway. The agent can move in the four cardinal directions but its actions fail with a 5% probability. The agent can only see what it is in the room that it currently occupies, as shown in Figure 5a. What makes these tasks difficult is the hallway. The hallway forces the agent to observe long sequences of identical observations multiple times to solve a task. However, depending on previous observations, the optimal actions and expected returns will be completely different when the agent is in the hallway.

The first environment is the *cookie domain* (Figure 5b) described in Section 3. Each episode is $5,000$ steps long, during which the agent should attempt to get as many cookies as possible. To do so, it has to press the button in the orange room and then look for the cookie that is delivered to the blue or green room.

The second environment is the *symbol domain* (Figure 5c). This domain has three symbols ♣, ♠, and ♦ in the blue and green rooms. At the beginning of an episode, one symbol from {♣, ♠, ♦} and possibly a right or left arrow are randomly placed at the orange room. Intuitively, that symbol and arrow will tell the agent where to go, for example, ♣ and → tell the agent to go to ♣ in the east room. If there is no arrow, the agent can go to the target symbol in either room. An episode ends when the agent reaches any symbol in the blue or green room, at which point it receives a reward of $+1$ if it reached the correct symbol and -1 otherwise. All other steps in the environment provide no reward.

The third environment is the *2-keys domain* (Figure 5d). The agent receives a reward of $+1$ when it reaches the coffee in the orange room. To do so, it must open the two doors, shown in brown. Each door requires a different key to open it, and the agent can only carry one key at a time. At the beginning of each episode, the two keys are randomly located in either the green room, the blue room, or split between them. To solve this problem, the agent must keep track of the locations of the keys.

7.2. Comparisons Between the Discrete Optimization Models

We first compare the performance of our four models for solving different instances of `LRM`. The objective of this experiment is to compare how well each model scales as we increase the size of the training data and the size of the reward machine. To that end, we generated 20 training sets per domain, where each training set consists of 10^3, 10^4, 10^5, or 10^6 experiences collected by following a uniformly random policy. We sampled five training sets per each possible size and learned reward machines with at most 5 or 10 states. This gave a total of 120 problems instances of `LRM`.

(a) Agent's view. (b) Cookie domain.

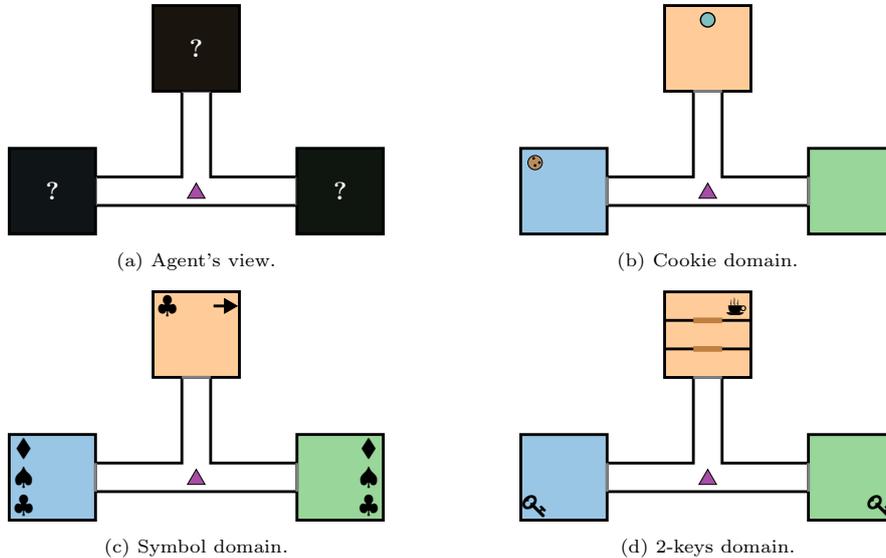(c) Symbol domain. (d) 2-keys domain.

Figure 5: Partially observable domains where the agent can only see what is in the current room.

Each approach was run with a 10-minute time limit using 62 cores on a Threadripper 2990WX processor with 124GB of RAM. We used Gurobi 9.1 (Gurobi Optimization, LLC, 2018) to solve the MILP model and IBM ILOG CP Optimizer 12.8 (IBM, 2018) for the CP model. These are sophisticated state-of-the-art solvers. In contrast, we used a simple Python implementation of local search and tabu search in our experiments. We set $\tau_{\text{size}} = 100$ for tabu search. We note that local search and tabu search are stochastic approaches that, in contrast to MILP and CP, might find a different solution on each run. For that reason, we ran local search and tabu search 5 times per problem instance and report the average cost across those runs.

Table 1 shows the final results. Each row shows the aggregated results over five problem instances that share the same domain (i.e., cookie, symbol, or 2-keys), maximum number of RM states (i.e., $u_{\max} \in \{5, 10\}$), and size of the training set (i.e., $|\mathcal{T}| \in \{10^3, 10^4, 10^5, 10^6\}$). Each row also shows the average size of the training set $|\mathcal{T}_c|$ after the traces are compressed (as described in Section 5). The table reports the average objective function of each model, where lower is better, and the number of instances where each model found the best solution among all others.

For training sets with less than $10,000$ experiences, our CP model tends to find the best solutions. However, for larger instance, local search methods dominate. Note that continuously restarting local search is a better strategy for learning RMs than using a tabu list in these domains. Still, the performance of TS is not too far from LS and, hence, we test both approaches for learning RMs in our next experiments.

7.3. Reinforcement Learning Experiments

We tested two versions of our *learned reward machine (LRM)* method: LRM+DDQN and LRM+DQRM. Both learn RMs from experience but LRM+DDQN learns a policy using DDQN (Van Hasselt et al., 2016) while LRM+DQRM uses the modified version

Table 1: Comparing different models for solving LRM in problem instances with training sets varying from 10^3 to 10^6 experiences and $u_{\max} \in \{5, 10\}$. Each row includes five problem instances.

Configuration			Avg. objective				No. best			
Dataset	$|\mathcal{T}|$	$|\mathcal{T}_c|$	MILP	CP	LS	TS	MILP	CP	LS	TS
	10^3	59	**12.6**	**12.6**	14.2	13.8	**5**	**5**	1	1
Cookie	10^4	487	237.3	**226.7**	229.1	230.2	1	**5**	2	0
($u_{\max} = 5$)	10^5	4943	3097.0	2700.4	**2699.7**	2719.7	0	**3**	2	0
	10^6	48663	31075.1	28226.1	**26462.6**	26833.3	0	0	**5**	0
	10^3	59	**6.5**	6.8	9.6	10.5	**5**	4	0	0
Cookie	10^4	487	233.3	204.6	206.0	**204.5**	0	1	0	**4**
($u_{\max} = 10$)	10^5	4943	3197.0	2713.7	**2658.9**	2696.8	0	0	**5**	0
	10^6	48663	30709.5	28366.8	**26461.7**	27092.0	0	0	**5**	0
	10^3	41	**21.0**	**21.0**	**21.0**	**21.0**	**5**	**5**	**5**	**5**
Symbol	10^4	268	**218.8**	**218.8**	**218.8**	220.0	**5**	**5**	**5**	3
($u_{\max} = 5$)	10^5	2597	3423.7	2897.9	**2896.7**	2902.3	0	3	**5**	2
	10^6	25875	36705.4	29689.9	**29687.7**	29688.8	0	4	**5**	3
	10^3	41	**16.2**	**16.2**	16.5	16.4	**5**	**5**	2	2
Symbol	10^4	268	185.8	**181.2**	181.5	185.0	1	**5**	3	0
($u_{\max} = 10$)	10^5	2597	3416.1	2620.7	**2583.5**	2620.4	0	0	**5**	0
	10^6	25875	36216.6	27992.9	**27050.4**	**27050.4**	0	0	**5**	**5**
	10^3	42	**6.9**	**6.9**	7.6	7.5	**5**	**5**	2	2
2-Keys	10^4	378	196.5	**176.7**	176.9	180.7	1	**5**	3	1
($u_{\max} = 5$)	10^5	3690	3713.2	2364.7	**2349.6**	2391.4	0	0	**5**	0
	10^6	37923	38875.3	29379.9	**24397.0**	24762.1	0	0	**5**	0
	10^3	42	**3.5**	**3.5**	5.4	5.1	**5**	**5**	0	0
2-Keys	10^4	378	184.4	151.6	**145.4**	157.2	0	0	**5**	0
($u_{\max} = 10$)	10^5	3690	3746.1	2363.8	**2210.6**	2237.6	0	0	**5**	0
	10^6	37923	38087.0	29065.0	**23352.9**	23558.8	0	0	**5**	0
Average/Total			9732.7	7900.3	**7251.8**	7325.2	38	60	**85**	28

of QRM described in Section 6. To learn the reward machine, these approaches solve LRM using local search with restarts or tabu search. In all domains, we used $u_{\max} = 10$, $t_{\max} = 100$, $\tau_{\text{size}} = 100$, $t_{\text{w}} = 200,000$, an epsilon greedy policy with $\epsilon = 0.1$, and a discount factor $\gamma = 0.9$. We compared against 4 baselines: DDQN (Van Hasselt et al., 2016), A3C (Mnih et al., 2016), ACER (Wang et al., 2016), and PPO (Schulman et al., 2017). DDQN uses the concatenation of the last 10 observations as input which gives DDQN a limited memory to better handle the domains. A3C, ACER, and PPO use an LSTM to summarize the history. Note that the output of the labelling function was also given to the baselines, as described below.

7.3.1. Hyperparameters and Features

For LRM+DDQN and LRM+DQRM, the neural network used has 5 fully connected layers with 64 neurons per layer. On every step, we trained the network using 32 sampled experiences from a replay buffer of size 100,000 and a learning rate of $5 \cdot 10^{-5}$. The target
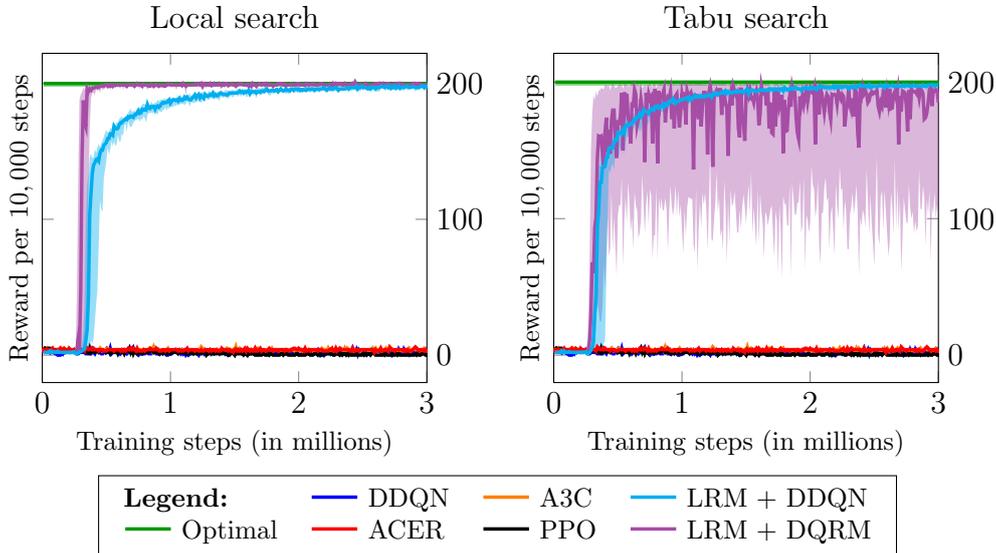
Figure 6: Results on the cookie domain. LRM is solved using local search or tabu search.

networks were updated every 100 steps.

The DDQN baseline uses the same parameters and network architecture as our LRM methods, but its input is the concatenation of the last 10 observations, as commonly done by Atari playing agents (Mnih et al., 2015). This gives DDQN a limited memory to better handle partially observable domains. We note that since the optimal path from any one room to another is less than 10 steps, giving the agent the last 10 observations means that the agent has enough information to perfectly summarize its history if it can figure out how to do so. The rest of the baselines, namely A3C, ACER, and PPO, use an LSTM to summarize the history.

To select hyperparameters for A3C, ACER, and PPO, we followed the same methodology that was used in their original publications. We ran each approach at least 30 times per domain, and on every run, we randomly selected the number of hidden neurons for the LSTM from $\{64, 128, 256, 512\}$ and a learning rate from (1e-3, 1e-5). We also sampled δ from $\{0, 1, 2\}$ for ACER and the clip range from (0.1, 0.3) for PPO. Other parameters were fixed to their default values.

While interacting with the environment, the agents were given a "top-down" view of the world represented as a set of binary matrices. One matrix had a 1 in the current location of the agent, one had a 1 in only those locations that are currently observable, and the remaining matrices each corresponded to an object in the environment and had a 1 at only those locations that were both currently observable and contained that object (i.e., locations in other rooms are "blacked out"). The agent also had access to features indicating if they were carrying a key, which color room they were in, and the current status of the events detected by the labelling function.

Figure 7: Results on the symbol domain. LRM is solved using local search or tabu search.

7.3.2. Results

Figures 6, 7, and 8 show the total cumulative rewards that each approach gets every 10,000 training steps and compares it to the optimal policy. For the LRM algorithms, the figures show the median performance over 30 runs per domain, and percentile 25 to 75 in the shadowed area. For the DDQN baseline, we show the maximum performance seen for each time period over 5 runs per problem. Similarly, we also show the maximum performance over the 30 runs of A3C, ACER, and PPO per period. All the baselines outperformed a random policy, but none make much progress on any of the domains. Each figure shows two settings. In the left, it shows the performance when LRM is solved using local search with restarts. In the right, it shows the case where LRM is solved using tabu search. Note that this only affects the LRM methods. The baselines' performance is identical in the left and right figures.

As the results show, LRM-based methods largely outperform all the baselines in these domains, reaching an optimal policy in the cookie domain (Figure 6) and a close-to-optimal policy in the symbol domain (Figure 7). We also note that LRM+DQRM learns faster than LRM+DDQN. In particular, LRM+DQRM converged to considerably better policies in the 2-keys domain (Figure 8). However, LRM+DQRM is more unstable than LRM+DDQN when solving LRM via tabu search. We believe this behaviour is due to two factors. First, tabu search is likely finding worse solutions than local search, as suggested by Table 1. Second, QRM exploits the structure of the learned RM. Thus, it is reasonable to expect that converging to a suboptimal RM would hurt the performance of DQRM more than the performance of DDQN.

Figure 8: Results on the 2-keys domain. LRM is solved using local search or tabu search.

8. Discussion

Solving partially observable RL problems is challenging and LRM was able to solve three problems that were conceptually simple but presented a major challenge to A3C, ACER, and PPO with LSTM-based memories. A key idea behind these results was to optimize over a necessary condition for perfect RMs. This objective favors RMs that are able to predict possible and impossible future observations at the abstract level given by the labelling function L. In this section, we discuss the advantages and current limitations of such an approach.

We begin by considering the performance of local search methods in our domains. Given a training set composed of one million transitions, our simple Python implementation of local search takes less than 2.5 minutes to learn an RM across all our environments, when using 62 workers on a Threadripper 2990WX processor and $t_{max} = 100$. Note that local search's main bottleneck is evaluating the neighbourhood around the current RM solution. As the size of the neighbourhood depends on the size of the set of propositional symbols $\mathcal{P}$, exhaustively evaluating the neighbourhood may sometimes become impractical. To handle such problem, we might import ideas from the *large neighborhood search* literature (Pisinger and Ropke, 2010).

Regarding limitations, learning the RM at the abstract level is efficient but requires ignoring (possibly relevant) low-level information. For instance, Figure 9a shows an adversarial example for LRM. The agent receives reward for eating the cookie (☺). There is an external force pulling the agent down – i.e., the outcome of the "`move-up`" action is actually a downward movement with high probability. The agent can press a button (◉) to turn off (or back on) the external force. Hence, the optimal policy is to press the button and then eat the cookie. Given $\mathcal{P} = \{☺, ◉\}$, a perfect RM for this environment is fairly simple (see Figure 9b) but LRM might not find it, even if the traces are not

25

(a) The gravity domain.   (b) A perfect RM for the gravity domain.

Figure 9: A partially observable environment where the agent cannot see the external force.

compressed. The reason is that pressing the button changes the low-level probabilities in the environment but does not change what is possible or impossible at the abstract level. In other words, while the LRM objective optimizes over necessary conditions for finding a perfect RM, those conditions are not sufficient to ensure that an optimal solution will be a perfect RM. In addition, if a perfect RM is found, our heuristic approach to share experiences in QRM would not work as intended because the experiences collected when the force is on (at u_0) would be incorrectly used to update the policy for the case where the force is off (at u_1).

Other current limitations include that it is unclear how to handle noise over the high-level detectors L and how to transfer learning from previously learned policies when a new RM is learned. Finally, defining a set of proper high-level detectors for a given environment might be a challenge to deploying LRM. Hence, looking for ways to automate that step is an important direction for future work.

9. Related Work

State-of-the-art approaches to partially observable RL use Recurrent Neural Networks (RNNs) as memory in combination with policy gradient (e.g., Mnih et al., 2016; Wang et al., 2016; Schulman et al., 2017; Jaderberg et al., 2016) or use external neural-based memories (e.g., Oh et al., 2016; Khan et al., 2017; Hung et al., 2018). Other approaches include extensions to Model-Based Bayesian RL that work under partial observability (e.g., Poupart and Vlassis, 2008; Doshi-Velez et al., 2013; Ghavamzadeh et al., 2015) or provide a small binary memory to the agent and a special set of actions to modify it (Peshkin et al., 1999). While our experiments highlight the merits of our approach with respect to RNN-based approaches, we rely on ideas that are largely orthogonal. As such, there is significant potential in mixing these approaches to get the benefit of memory at both the high- and the low-level.

The effectiveness of automata-based memory has long been recognized in the POMDP literature (Cassandra et al., 1994), where the objective is to find policies given a complete specification of the environment. The idea is to encode policies using *finite state controllers (FSCs)* which are *finite state machines (FSMs)* where the transitions are defined in terms of low-level observations from the environment and each state in the FSM is associated with one primitive action. When interacting with the environment, the agent always selects the action associated with the current state in the controller. Meuleau et al. (1999) adapted this idea to work in the RL setting by exploiting policy gradient

26

to learn policies encoded as FSCs. RMs can be considered as a generalization of FSC as they allow for transitions using conditions over high-level events and associate complete policies (instead of just one primitive action) to each state. This property allows our approach to easily leverage existing deep RL methods to learn policies from low-level inputs, such as images – which is not achievable by Meuleau et al. (1999). That said, further investigating using ideas for learning FSMs (e.g., Angluin and Smith, 1983; Zeng et al., 1993; Giantamidis and Tripakis, 2016; Shvo et al., 2021) in learning RMs is a promising direction for future work.

Our approach to learn RMs is greatly influenced by *predictive state representations (PSRs)* (Littman et al., 2002). The idea behind PSRs is to find a set of core tests (i.e., sequences of actions and observations) such that if the agent can predict the probabilities of these occurring, given any history H, then those probabilities can be used to compute the probability of any other test given H. The insight is that state representations that are good for predicting the next observation are good for solving partially observable environments. We adapted this idea to the context of RM learning.

Finally, we note that different approaches to learn RMs were proposed simultaneously, or shortly after, our original publication (e.g., Xu et al., 2020a,b; Furelos-Blanco et al., 2020; Rens et al., 2020; Gaon and Brafman, 2020; Memarian et al., 2020; Neider et al., 2021; Hasanbeig et al., 2021). They all learn reward machines in fully observable domains. Their goal is to learn the smallest RM that is consistent with the reward function – which makes sense for fully observable domains, but would have limited utility under partial observability (as discussed in Section 4).

Since they stay in the fully-observable setting, they can use off-the-shelf automata learning approaches to learn the RM. These include methods that learn reward machines using a SAT solver (Xu et al., 2020a; Neider et al., 2021), use inductive logic programming (Furelos-Blanco et al., 2020), and by using program synthesis (Hasanbeig et al., 2021). There has also been work on adapting the L^* algorithm (Angluin, 1987) to learn RMs given the model of the MDP (Rens et al., 2020), expert demonstrations (Memarian et al., 2020), or in a pure RL setting (Gaon and Brafman, 2020; Xu et al., 2020b).

Besides proposing approaches to learn reward machines for fully-observable problems, these works also make additional contributions that may be useful in the context of partial observability. For instance, Furelos-Blanco et al. (2020) and Hasanbeig et al. (2021) add a reward shaping procedure to encourage exploration. Xu et al. (2020a) propose a simple mechanism to transfer some of the previously learned Q-value estimates when a new reward machine is learned. Neider et al. (2021) show how to incorporate domain knowledge when learning a reward machine. Finally, Gaon and Brafman (2020) and Xu et al. (2020b) allow, in some cases, driving the agent's exploration towards finding *bugs* in the reward machine. Further study into how to use these in the case of partial observability is left as future work.

10. Concluding Remarks

We have presented a method for learning reward machines in partially observable environments and demonstrated the effectiveness of doing so to tackle partially observable RL problems that are unsolvable by the state-of-the art deep RL methods A3C, ACER and PPO. Informed by criteria from the POMDP, FSC, and PSR literature, we proposed a set of RM properties that support tackling RL in partially observable environments.

We used these properties to formulate RM learning as a discrete optimization problem. We experimented with several optimization methods, finding local search methods to be the most effective. We then combined this RM learning with policy learning for solving partially observable RL problems. Our combined approach outperformed a set of strong LSTM-based approaches on different domains.

We believe this work represents an important building block for creating RL agents that can solve cognitively challenging partially observable tasks. Not only did our approach solve problems that were unsolvable by A3C, ACER and PPO, but it did so in a relatively small number of training steps. RM learning provided the agent with memory, but more importantly the combination of RM learning and policy learning provided it with discrete reasoning capabilities that operated at a higher level of abstraction, while leveraging deep RL's ability to learn policies from low-level inputs. This work leaves open many interesting questions relating to abstraction, observability, and properties of the language over which RMs are constructed. We believe that addressing these questions will push the boundary of partially observable RL problems that can be solved.

Acknowledgements

References

Aarts, E., Aarts, E.H., Lenstra, J.K., 2003. Local search in combinatorial optimization. Princeton University Press.

Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al., 2018. Learning dexterous in-hand manipulation. CoRR abs/1808.00177. URL: http://arxiv.org/abs/1808.00177.

Angluin, D., 1987. Learning regular sets from queries and counterexamples. Information and computation 75, 87–106.

Angluin, D., Smith, C.H., 1983. Inductive inference: Theory and methods. ACM Computing Surveys (CSUR) 15, 237–269.

Camacho, A., Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A., 2019. LTL and beyond: Formal languages for reward function specification in reinforcement learning, in: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), pp. 6065–6073.

Cassandra, A.R., Kaelbling, L.P., Littman, M.L., 1994. Acting optimally in partially observable stochastic domains, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI), pp. 1023–1028.

De Giacomo, G., Favorito, M., Iocchi, L., Patrizi, F., Ronca, A., 2020. Temporal logic monitoring rewards via transducers, in: Proceedings of the 17th International Conference on Knowledge Representation and Reasoning (KR), pp. 860–870.

Doshi-Velez, F., Pfau, D., Wood, F., Roy, N., 2013. Bayesian nonparametric methods for partially-observable reinforcement learning. IEEE transactions on pattern analysis and machine intelligence 37, 394–407.

Dulac-Arnold, G., Levine, N., Mankowitz, D.J., Li, J., Paduraru, C., Gowal, S., Hester, T., 2021. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. Machine Learning 110, 1–50.

Dulac-Arnold, G., Mankowitz, D., Hester, T., 2019. Challenges of real-world reinforcement learning. CoRR abs/1904.12901. URL: http://arxiv.org/abs/1904.12901.

Furelos-Blanco, D., Law, M., Russo, A., Broda, K., Jonsson, A., 2020. Induction of subgoal automata for reinforcement learning., in: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 3890–3897.

Gaon, M., Brafman, R., 2020. Reinforcement learning with non-Markovian rewards, in: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 3980–3987.

Ghavamzadeh, M., Mannor, S., Pineau, J., Tamar, A., et al., 2015. Bayesian reinforcement learning: A survey. Foundations and Trends in Machine Learning 8, 359–483.

Giantamidis, G., Tripakis, S., 2016. Learning Moore machines from input-output traces, in: Proceedings of the 21st International Symposium on Formal Methods (FM), pp. 291–309.

Glover, F., Laguna, M., 1998. Tabu search, in: Handbook of combinatorial optimization. Springer, pp. 2093–2229.

Gurobi Optimization, LLC, 2018. Gurobi Optimizer Reference Manual. URL: http://www.gurobi.com.

Hasanbeig, M., Jeppu, N.Y., Abate, A., Melham, T., Kroening, D., 2021. Deepsynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. CoRR abs/1911.10244. URL: http://arxiv.org/abs/1911.10244.

Hausknecht, M., Stone, P., 2015. Deep recurrent q-learning for partially observable MDPs, in: AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15).

De la Higuera, C., 2010. Grammatical inference: learning automata and grammars. Cambridge University Press.

Hung, C.C., Lillicrap, T., Abramson, J., Wu, Y., Mirza, M., Carnevale, F., Ahuja, A., Wayne, G., 2018. Optimizing agent behavior over long time scales by transporting value. CoRR abs/1810.06721. URL: http://arxiv.org/abs/1810.06721.

IBM, 2018. ILOG CP Optimizer 12.8 Manual.

Izadi, M.T., Precup, D., 2005. Using rewards for belief state updates in partially observable Markov decision processes, in: Proceedings of the 16th European Conference on Machine Learning (ECML), pp. 593–600.

Jaderberg, M., Mnih, V., Czarnecki, W.M., Schaul, T., Leibo, J.Z., Silver, D., Kavukcuoglu, K., 2016. Reinforcement learning with unsupervised auxiliary tasks. CoRR abs/1611.05397. URL: http://arxiv.org/abs/1611.05397.

Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A., 2009. 50 Years of integer programming 1958-2008: From the early years to the state-of-the-art. Springer Science & Business Media.

Kaelbling, L.P., Littman, M.L., Moore, A.W., 1996. Reinforcement learning: A survey. Journal of artificial intelligence research 4, 237–285.

Khan, A., Zhang, C., Atanasov, N., Karydis, K., Kumar, V., Lee, D.D., 2017. Memory augmented control networks. CoRR abs/1709.05706. URL: http://arxiv.org/abs/1709.05706.

Littman, M.L., 1993. An optimization-based categorization of reinforcement learning environments, in: From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior, pp. 262–270.

Littman, M.L., Sutton, R.S., Singh, S., 2002. Predictive representations of state, in: Proceedings of the 15th Conference on Advances in Neural Information Processing Systems (NIPS), pp. 1555–1561.

Mahmud, M., 2010. Constructing states for reinforcement learning, in: Proceedings of the 27th International Conference on Machine Learning (ICML), pp. 727–734.

Memarian, F., Xu, Z., Wu, B., Wen, M., Topcu, U., 2020. Active task-inference-guided deep inverse reinforcement learning, in: Proceedings of the 59th IEEE Conference on on Decision and Control (CDC), pp. 1932–1938.

Meuleau, N., Peshkin, L., Kim, K.E., Kaelbling, L.P., 1999. Learning finite-state controllers for partially observable environments, in: Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI), pp. 427–436.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning, in: Proceedings of the 33rd International Conference on Machine Learning (ICML), pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al., 2015. Human-level control through deep reinforcement

learning. Nature 518, 529–533.

Neider, D., Gaglione, J.R., Gavran, I., Topcu, U., Wu, B., Xu, Z., 2021. Advice-guided reinforcement learning in a non-Markovian environment, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI), pp. 9073–9080.

Oh, J., Chockalingam, V., Singh, S., Lee, H., 2016. Control of memory, active perception, and action in minecraft, in: Proceedings of the 33rd International Conference on Machine Learning (ICML), pp. 2790–2799.

Peshkin, L., Meuleau, N., Kaelbling, L.P., 1999. Learning policies with external memory, in: Proceedings of the 16th International Conference on Machine Learning (ICML), pp. 307–314.

Pisinger, D., Ropke, S., 2010. Large neighborhood search, in: Handbook of metaheuristics. Springer, pp. 399–419.

Poupart, P., Vlassis, N., 2008. Model-based Bayesian reinforcement learning in partially observable domains, in: Proceedings of the 10th International Symposium on Artificial Intelligence and Mathematics (ISAIM), pp. 1–2.

Rens, G., Raskin, J.F., Reynouad, R., Marra, G., 2020. Online learning of non-Markovian reward models. CoRR abs/2009.12600. URL: `https://arxiv.org/abs/2009.12600`.

Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of constraint programming. Elsevier.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. CoRR abs/1707.06347. URL: `http://arxiv.org/abs/1707.06347`.

Shvo, M., Li, A.C., Toro Icarte, R., McIlraith, S.A., 2021. Interpretable sequence classification via discrete optimization, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI), pp. 9647–9656.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al., 2017. Mastering the game of Go without human knowledge. Nature 550, 354.

Singh, S.P., Jaakkola, T., Jordan, M.I., 1994. Learning without state-estimation in partially observable Markovian decision processes, in: Machine Learning Proceedings 1994. Elsevier, pp. 284–292.

Sutton, R.S., Barto, A.G., 2018. Reinforcement learning: An introduction. MIT press.

Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A., 2018. Using reward machines for high-level task specification and decomposition in reinforcement learning, in: Proceedings of the 35th International Conference on Machine Learning (ICML), pp. 2112–2121.

Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A., 2020a. Reward machines: Exploiting reward function structure in reinforcement learning. CoRR abs/2010.03950. URL: `https://arxiv.org/abs/2010.03950`.

Toro Icarte, R., Valenzano, R., Klassen, T.Q., Christoffersen, P., massoud Farahmand, A., McIlraith, S.A., 2020b. The act of remembering: a study in partially observable reinforcement learning. CoRR abs/2010.01753. URL: `http://arxiv.org/abs/2010.01753`.

Toro Icarte, R., Waldie, E., Klassen, T.Q., Valenzano, R., Castro, M.P., McIlraith, S.A., 2019. Learning reward machines for partially observable reinforcement learning, in: Proceedings of the 32nd Conference on Advances in Neural Information Processing Systems (NeurIPS), pp. 15497–15508.

Van Hasselt, H., Guez, A., Silver, D., 2016. Deep reinforcement learning with Double Q-learning, in: Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI), pp. 2094–2100.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., de Freitas, N., 2016. Sample efficient actor-critic with experience replay. CoRR abs/1611.01224. URL: `http://arxiv.org/abs/1611.01224`.

Watkins, C.J.C.H., Dayan, P., 1992. Q-learning. Machine learning 8, 279–292.

Xu, Z., Gavran, I., Ahmad, Y., Majumdar, R., Neider, D., Topcu, U., Wu, B., 2020a. Joint inference of reward machines and policies for reinforcement learning, in: Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS), pp. 590–598.

Xu, Z., Wu, B., Neider, D., Topcu, U., 2020b. Active finite reward automaton inference and reinforcement learning using queries and counterexamples. CoRR abs/2006.15714. URL: `https://arxiv.org/abs/2006.15714`.

Zeng, Z., Goodman, R.M., Smyth, P., 1993. Learning finite state machines with self-clustering recurrent networks. Neural Computation 5, 976–990.

Zhang, M., McCarthy, Z., Finn, C., Levine, S., Abbeel, P., 2016. Learning deep neural network policies with continuous memory states, in: Proceedings of the 2016 IEEE International Conference on Robotics and Automation (ICRA), pp. 520–527.