Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Pushdown Reward Machines for Reinforcement Learning

Giovanni Varricchione[1] , **Toryn Q. Klassen**[2,3] , **Natasha Alechina**[4,1] ,
Mehdi Dastani[1] , **Brian Logan**[5,1] and **Sheila A. McIlraith**[2,3]
[1]Utrecht Universiteit, Utrecht, The Netherlands
[2]University of Toronto, Toronto, Canada
[3]Vector Institute, Toronto, Canada
[4]Open Universiteit, Heerlen, The Netherlands
[5]University of Aberdeen, Aberdeen, United Kingdom
{g.varricchione, n.a.alechina, m.m.dastani, b.s.logan}@uu.nl, {toryn, sheila}@cs.toronto.edu

Abstract

Reward machines (RMs) are automata structures that encode (non-Markovian) reward functions for reinforcement learning (RL). RMs can reward any behaviour representable in regular languages and, when paired with RL algorithms that exploit RM structure, have been shown to significantly improve sample efficiency in many domains. In this work, we present *pushdown reward machines* (pdRMs), an extension of reward machines based on deterministic pushdown automata. pdRMs can recognise and reward temporally extended behaviours representable in deterministic context-free languages, making them more expressive than reward machines. We introduce two variants of pdRM-based policies, one which has access to the entire stack of the pdRM, and one which can only access the top k symbols (for a given constant k) of the stack. We propose a procedure to check when the two kinds of policies (for a given environment, pdRM, and constant k) achieve the same optimal state values. We then provide theoretical results establishing the expressive power of pdRMs, and space complexity results for the proposed learning problems. Lastly, we propose an approach for off-policy RL algorithms that exploits counterfactual experiences with pdRMs. We conclude by providing experimental results showing how agents can be trained to perform tasks representable in deterministic context-free languages using pdRMs.

1 Introduction

Reward machines (RMs) (Toro Icarte et al. 2018; Toro Icarte et al. 2022) are automata structures that are used to represent (non-Markovian) reward functions for reinforcement learning (RL). Among their merits, they enable RL algorithms to exploit the compositional structure of RMs in learning, resulting in significant sample efficiency gains. By virtue of their correspondence to deterministic finite state automata (DFAs), any reward-worthy behaviour expressible by a regular language, as well as variants of other formal languages such as variants of linear temporal logic (LTL), can be encoded by an RM. This means that a human can write their non-Markovian reward function, or reward-worthy (temporally extended) behaviour in a diversity of programming/formal languages, compile them to an RM, and take advantage of the sample efficiency gains of these RM-tailored learning algorithms (Camacho et al. 2019).

A restriction of RMs is that reward-worthy behaviour must be representable in a DFA-like structure—a regular or Type-3 language, according to Chomsky's hierarchy of languages (Chomsky and Schützenberger 1959). However, a number of interesting RL problems require the expressiveness of a context-free language. To enhance the expressiveness of RMs, Bester et al. (2024) introduced *counting reward automata* (CRAs) which augment RMs with counters. CRAs have the expressive power of counter machines. As a counter machine with two or more counters has the same expressive power as a Turing machine (Minsky 1967), CRAs with two or more counters can express behaviours that are representable by recursively enumerable languages, the largest class in the Chomsky hierarchy. Unfortunately, the expressive power of CRAs can come at a significant computational cost for RL. The resulting product MDP and policy incur a blowup depending on the maximum values that the counters can assume. This blowup can severely hinder training by slowing the convergence speed: as the MDP and policy state spaces grow, the time required to explore and learn increases.

In this paper, following Chomsky's hierarchy, we propose a more modest enhancement to the expressiveness of RMs by augmenting RMs with a single stack. We call these enhanced RMs *pushdown reward machines* (pdRMs). Our enhancement is based on deterministic pushdown automata (DPDAs), which allow us to encode reward-worthy behaviours that are representable, for example, in LR grammars (Knuth 1965), or, precisely, deterministic context-free languages. pdRMs can recognize a wide range of practical behaviours, such as modelling recursive calls in programming, collecting and delivering arbitrary numbers of parcels to specific locations, or search and rescue tasks where an agent must return to its starting point by remembering and retracing its (safe) route.

The main contributions of this paper are as follows:

- We define reward machines based on deterministic pushdown automata. Given their structure, pdRMs can encode tasks representable in deterministic context-free languages;
- We define two variants of pdRM-based policies. In the first, the policy has access to the entire pdRM stack, in the second it can only access the top k symbols (for a given k) of the stack;
- We provide a procedure to check whether optimal policies

with access to the top k stack symbols achieve the same state values as optimal policies with full-stack access;
- We analyse the expressive power of pdRMs and compare it to RMs and CRAs. We also evaluate the space blowup for pdRM- and CRA-based policies. We show that pdRM-policies accessing only the top k symbols of the stack are more compact than pdRM-policies accessing the entire stack and CRA-based policies;
- We propose an approach that exploits counterfactual experiences by generating synthetic experiences based on the states of the pdRM and the stack strings observed during training;
- We use pdRMs to train RL agents in several domains. We compare them to CRAs in a domain from (Bester et al. 2024). Then, we show the practical effects of the space complexity results we establish. We show how counterfactual and hierarchical approaches for pdRMs can be used to increase sample efficiency. Finally, we use pdRMs in a continuous domain, and compare against a deep learning algorithm using recurrent neural networks.

2 Preliminaries

In reinforcement learning (RL), the environment in which agents act and learn is modelled as a *Markov decision process* (MDPs) (Puterman 2014). A Markov decision process is a tuple $M = \langle S, A, p, r, \gamma \rangle$ where S is the non-empty set of states, A is the non-empty set of actions, $p : S \times A \to \Delta(S)$ is the state transition function where $\Delta(S)$ is the set of all probability distributions defined over S, $r : S \times A \times S \to \mathbb{R}$ is the reward function, and $\gamma \in (0, 1)$ is the discount factor. We write $p(s' \mid s, a)$ to denote the probability of transitioning from state s to state s' when action a is performed in s. A *policy* is a function $\pi : S \to \Delta(A)$ mapping any state of the MDP to a probability distribution over the set of actions. We denote by $\pi(a \mid s)$ the probability that an agent following policy π performs action a in state s.

At each timestep t, the MDP is in some state s_t. The agent takes an action $a_t \sim \pi(\cdot \mid s_t)$ using its policy π, after which the environment state is updated to $s_{t+1} \sim p(\cdot \mid s_t, a_t)$ and the agent is rewarded with $r_t = r(s_t, a_t, s_{t+1})$. The goal of the agent is to learn an *optimal* policy π^*, i.e., one that maximizes the expected discounted reward $\mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k r_k \mid S_0 = s\right]$ from any MDP state s.

Reward machines were introduced by Toro Icarte et al. (2018; 2022) to encode non-Markovian reward functions. A reward machine (RM) is a tuple $\mathcal{R} = \langle U, u_0, F, \Sigma, \delta_u, \delta_r \rangle$, where U is a finite non-empty set of states, u_0 is the initial reward machine state, $F \not\subseteq U$ is the set of final states, Σ is the input alphabet, $\delta_u : U \times \Sigma \to (U \cup F)$ is the transition function, and $\delta_r : U \times \Sigma \to \mathbb{R}$ is the output reward function. The transitions in the reward machine are related to transitions in the MDP by a *labelling function* $L : S \times A \times S \to \Sigma$ which labels each state-action-state triple of the MDP with an input symbol of the reward machine. When the reward function is specified by a reward machine, the agent's policy is defined over the Cartesian product of the set of MDP states and the set of states of the reward machine.

3 Pushdown Reward Machines

In this section, we define pushdown Reward Machines (pdRMs). Like RMs, pdRMs are used to express (non-Markovian) reward functions, however their automaton structure is enhanced with a stack, which serves as additional memory. In so doing, they enable the expression of reward-worthy temporally-extended behaviours that correspond to deterministic context-free languages.

Definition 1 (Pushdown Reward Machine)**.** A *pushdown reward machine* (pdRM) is a tuple $\mathcal{R} = \langle U, u_0, F, \Sigma, \Gamma, Z, \delta_u, \delta_r \rangle$, where:
- U is the finite set of states;
- $u_0 \in U$ is the initial state;
- $F \not\subseteq U$ is the set of final states;
- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ where Σ is the input alphabet;
- $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$ where Γ is the stack alphabet;
- $Z \in \Gamma$ is the initial stack symbol;
- $\delta_u : U \times \Sigma_\epsilon \times \Gamma_\epsilon \to (U \cup F) \times \Gamma_\epsilon^*$ is the transition function; and
- $\delta_r : U \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathbb{R}$ is the reward function.

where ϵ is the empty string. Below, we denote by $z \in \Gamma$ individual symbols of the stack alphabet, and by $\zeta \in \Gamma^*$ (possibly empty) stack strings. The transition function δ_u takes as input the current pdRM state u, the current input symbol $\sigma \in \Sigma_\epsilon$ and the topmost symbol of the stack $z \in \Gamma_\epsilon$, and returns a pair (u', ζ') where $u' \in (U \cup F)$ is the next state and $\zeta' \in \Gamma_\epsilon^*$ is a string which replaces z as the topmost symbol(s) of the stack. Note that σ, z and ζ' may be the empty string ϵ. Whenever $\sigma = \epsilon$, the pdRM does not read any symbol from the input, making a so-called "silent" transition. If $z = \epsilon$, no symbol is popped from the stack. Finally, if $\zeta' = \epsilon$, no symbol is pushed to the stack. In what follows, we consider only *deterministic* pdRMs where, for any pdRM state u and stack symbol z, if $\delta_u(u, \epsilon, z)$ is defined, then $\delta_u(u, \sigma, z)$ is not defined for any $\sigma \in \Sigma$. The reward function δ_r takes as input the current pdRM state u, the current input symbol σ and the topmost symbol z of the stack and returns a reward.

At each timestep, a pdRM is in a *configuration* $\langle u, z\zeta \rangle \in U \times \Gamma_\epsilon^*$, where u is the state of the pushdown reward machine and $z\zeta$ is the current string on the stack with z as the topmost symbol on the stack. The pdRM reads the current input symbol $\sigma \in \Sigma_\epsilon$, transitions to a new configuration $\langle u', \zeta'\zeta \rangle$ where $\langle u', \zeta' \rangle = \delta_u(u, \sigma, z)$, and outputs a reward $r = \delta_r(u, \sigma, z)$. We write $\langle u, \zeta \rangle \vdash_\sigma \langle u', \zeta' \rangle$ to denote that the pdRM moves from $\langle u, \zeta \rangle$ to $\langle u', \zeta' \rangle$ upon reading the symbol σ.

We illustrate the expressive power of a pdRM using a simple task expressible in a context-free language, which we call "Maze" task. In the task, the agent has to navigate a (gridlike) maze from a starting location to find a treasure and return to the starting point by following the path it traversed to reach the treasure in the reverse direction. The actions available to the agent are $A = \{\mathsf{u}, \mathsf{d}, \mathsf{l}, \mathsf{r}\}$, denoting up, down, left, and right respectively, and the task can be defined using the following (deterministic) context-free grammar:

$$S \to P\,\mathsf{x}$$
$$P \to \mathsf{u}\,P\,\mathsf{d} \mid \mathsf{d}\,P\,\mathsf{u} \mid \mathsf{l}\,P\,\mathsf{r} \mid \mathsf{r}\,P\,\mathsf{l} \mid \mathsf{t}$$

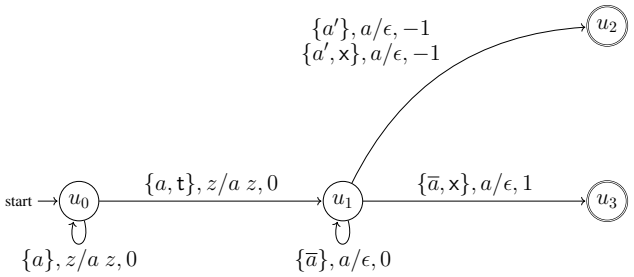

Figure 1: Pushdown reward machine for the Maze task. Each transition is labelled with a tuple $\ell, z/\zeta, r$, where ℓ is the input observation, z the top symbol on the stack, ζ the string of symbols pushed onto the stack (with the new top symbol leftmost in ζ), and r is the output reward. The symbol "z" indicates an arbitrary symbol in Γ. In the transition from u_1 to u_2, a' represents any direction except for $\overline{a}$, the opposite direction to a.

where t denotes the treasure and x the starting point.

The corresponding pdRM is shown in Figure 1. The states in the pdRM are $U = \{u_0, u_1, u_2, u_3\}$, the stack alphabet is $\Gamma = \{\mathsf{u}, \mathsf{d}, \mathsf{l}, \mathsf{r}\}$, the input alphabet $\Sigma = 2^{\Gamma \cup \{\mathsf{t}, \mathsf{x}\}}$ and the labelling L is defined as follows:

- $a \in L(s, a, s')$;
- $\mathsf{t} \in L(s, a, s')$ if $s \neq s'$ and s' is the location of the treasure;
- $\mathsf{x} \in L(s, a, s')$ if $s \neq s'$ and s' is the starting point.

As with standard reward machines, pdRMs can be used to reward the agent in MDPs. The product of an MDP and a pdRM gives rise to an "MDP-pdRM", defined as follows.

Definition 2. A *Markov decision process with a pushdown reward machine* (MDP-pdRM) is a tuple $\mathcal{T} = \langle S, A, p, \gamma, L, \mathcal{R} \rangle$, where S, A, p, γ are defined as in an MDP, $L : S \times A \times S \to \Sigma$ is a labelling function, and $\mathcal{R} = \langle U, u_0, F, \Sigma, \Gamma, Z, \delta_u, \delta_r \rangle$ is a pdRM.

Markov decision processes with a pushdown reward machine are analogous to the *Markov decision processes with a reward machine* introduced in (Toro Icarte et al. 2018), but with rewards specified by a pdRM rather than an RM.

An MDP-pdRM induces a *product MDP* $M_{\mathcal{T}} = \langle S_{\mathcal{T}}, A_{\mathcal{T}}, p_{\mathcal{T}}, r_{\mathcal{T}}, \gamma_{\mathcal{T}} \rangle$ where:

- $S_{\mathcal{T}} := S \times (U \cup F) \times \Gamma^*$;
- $A_{\mathcal{T}} := A$;
- $p_{\mathcal{T}}(\langle s', u', \zeta' \rangle \mid \langle s, u, \zeta \rangle, a) :=$

$$\begin{cases} p(s' \mid s, a) & \text{if } u \in F, u = u' \text{ and } \zeta = \zeta' \\ p(s' \mid s, a) & \text{if } u \notin F \text{ and} \\ & \langle u, \zeta \rangle \vdash_{L(s,a,s')} \langle u', \zeta' \rangle \\ 0 & \text{otherwise} \end{cases}$$

- $r_{\mathcal{T}}\left(\langle s, u, z\zeta \rangle, a, \langle s', u', \zeta' \rangle\right) := \delta_r(u, L(s, a, s'), z)$;
- $\gamma_{\mathcal{T}} := \gamma$.

Although not specified, the starting state of the pdRM in the product MDP is always u_0.

We now define the possible policies for MDP-pdRMs. The first type of policy has full access to the current state of the product MDP, and thus to all of the stack's contents.

Definition 3 (Policy)**.** A *policy* in an MDP-pdRM is a function $\pi : S \times U \times \Gamma^* \to \Delta(A)$.

A policy has, at each timestep, access to the current MDP state, pdRM state, and the entire contents of the pdRM stack. Since the stack is potentially unbounded, the state space of policies (hence their size) is potentially unbounded. As such, policies are not well-suited to many RL algorithms, e.g., tabular algorithms, especially in non-episodic tasks (i.e., a task in which the agent must act indefinitely in the environment). We therefore define a bounded version of policies which have access only to a portion of the stack. In this way, we bound the size of the policy, making it finite regardless of whether the task is episodic or not.

Definition 4 (k-policy)**.** Given a constant $k \geq 0$, a *k-policy* is a function $\pi : S \times U \times \left(\bigcup_{j \leq k} \Gamma^j\right) \to \Delta(A)$.

While k-policies can be more suited to learning in RL, we note that their limited observability of the product MDP's state can lead to suboptimal behaviours. In the next section, we will provide a procedure to verify in which cases a k-policy has access to enough information to achieve behaviours with the same state values as optimal policies.

4 When Are k-Policies Optimal?

As we have just observed, k-policies can achieve suboptimal behaviours compared to policies, since they have limited access to the product MDP's state. For example, imagine a task in which the agent is given a sequence of rooms (which is saved on the pdRM's stack) that it must clean. Depending on the room, the agent needs different supplies, which are stored in a storage room, to clean it. Gathering all the needed supplies before cleaning the rooms is more efficient as the agent will not have to return to the storage room, and for a policy to anticipate which supplies are needed, it needs access to the entire stack. However, having access to the entire stack is not always necessary to achieve an optimal behaviour. Indeed, in the Maze task it is possible to learn an optimal policy given access to only the top symbol on the stack, as the choice of direction when returning to the starting point is determined only by the top symbol. In this section, we characterise when, for each state, the state value for optimal k-policies is the same as the state value for optimal policies.

First, we define the notion of *k-equivalence* for states in the product MDP induced by an MDP-pdRM.

Definition 5 (k-equivalence $\sim_k$)**.** For a product MDP obtained from an MDP-pdRM, two states $\langle s, u, \zeta \rangle, \langle s, u, \zeta' \rangle$, are k-equivalent, denoted by $\langle s, u, \zeta \rangle \sim_k \langle s, u, \zeta' \rangle$, if and only if the top k symbols of ζ and ζ' are the same.

k-equivalence is an equivalence relation which partitions the state space of the product MDP into equivalence classes, where the members of each class share the MDP state, pdRM state, and top k symbols of the pdRM stack. In k-policies, the agent's policy is defined over the set of equivalence classes of the $\sim_k$ relation. We can therefore check whether optimal k-policies have the same value as optimal policies.

The size of the product MDP state space is potentially countably infinite since the states of the product contain the stack, and its size is unbounded. However the rewards are

bounded for all states and actions, and depend only on the the pdRM state, the current input symbol and the topmost symbol on the stack. As a result, the potentially unbounded growth in the size of the stack does not affect the immediate rewards. The Bellman operator therefore remains a contraction mapping and value iteration converges to the optimal value function as in the case of finite state spaces (see, e.g., (Feinberg 2011)).

Proposition 1. *If after performing value iteration on the product MDP, all $\sim_k$-equivalent states have the same value and the same sets of optimal actions, then optimal k-policies achieve the same state values as optimal policies.*

Proof. If the condition in the proposition holds, there exists an optimal policy where action selection depends only on the top k symbols on the stack. Hence, there exists a k-policy that results in the same state values as an optimal policy. □

In practice, it is not possible to check whether the condition in Proposition 1 holds for an infinite MDP. However, by bounding the maximal stack size to, e.g., the maximal length of a stack string given the episode length, we can check whether the condition holds for the resulting finite MDP.

5 Comparison With CRAs

In this section, we analyse the relative expressive power of pdRMs and counting reward automata (CRA) (Bester et al. 2024) and evaluate the space blowup for pdRM- and CRA-based policies. CRAs are based on counter machines (CM), i.e., finite automata augmented with k counters (for given k) that may be incremented, decremented and tested for zero.

First, we consider the expressive power of pdRMs and CRAs. As both pdRMs and CRAs are automata, the reward functions that they can encode correspond to the languages they accept. Deterministic pushdown automata are strictly more expressive than counter machines with one counter (Fischer 1966). On the other hand, counter machines with 2 or more counters can simulate a Turing machine (Minsky 1967), and as such they are strictly more expressive than pdRMs with one stack.

Next, we consider the space complexity of pdRMs and CRAs, where by "space complexity" we mean the bounds on the sizes of policies, i.e., the size of the table representing the polices that are learnt with pdRMs and CRAs. As pdRMs and CRAs only augment the state space of the underlying MDP, we analyse the blowup they cause in the state space to determine the blowup in the size of the policies. To compare space complexity, we focus on tasks representable in deterministic context-free languages and consider only episodic tasks of length n, as, in the case of non-episodic tasks (i.e., tasks where the agent acts in the environment indefinitely), both stacks and counters can assume infinitely many values and policies learnt with both pdRMs and CRAs may be infinite.

Policies As policies have access to the entire stack, they incur a blowup depending on the number of possible strings that can appear on the stack (i.e., the cardinality of the *stack language*). Let m be the maximum number of symbols that can be added to the stack at any transition of the pdRM. We define an ϵ-sequence as a sequence of ϵ-transitions in the pdRM which are taken before the pdRM must read a symbol to advance. Let e be the maximum number of ϵ-transitions pushing symbols to the stack in a single ϵ-sequence of the pdRM. At each step, the pdRM can add at most $m(e+1)$ symbols to the stack, and, for input words of length n, the maximum length of the stack string is bounded by $nm(e+1)$. Hence, the cardinality of the stack language is $|\Gamma|^0 + |\Gamma|^1 + \ldots + |\Gamma|^{nm(e+1)}$. If $|\Gamma| > 1$, then the number of stack strings is exactly $\frac{|\Gamma|^{nm(e+1)+1}-1}{|\Gamma|-1} \in O\left(|\Gamma|^{nm(e+1)}\right)$; for $|\Gamma| = 1$ we have that the number of stack strings is $nm(e+1)+1 \in O(nm(e+1))$.

Thus, we have the following:

Theorem 1. *If $|\Gamma| \geq 2$, policies incur a blowup in $O\left(|\Gamma|^{nm(e+1)}\right)$. If $|\Gamma| = 1$, policies incur a blowup in $O\left(nm(e+1)\right)$.*

k-policies Unlike for policies, for k-policies we just have to evaluate the number of strings of length at most k made of symbols from the stack alphabet, regardless of the task encoded by the pdRM. Clearly, the number of stack strings of length up to k is $|\Gamma|^0 + |\Gamma|^1 + \ldots + |\Gamma|^k$. Then, similarly to policies, if $|\Gamma| > 1$, the number of stack strings is exactly $\frac{|\Gamma|^{k+1}-1}{|\Gamma|-1} \in O(|\Gamma|^k)$, and for $|\Gamma| = 1$ the number of stack strings is $k+1 \in O(k)$. Note that for $k = 1$, this implies that the blowup is linear in $|\Gamma|$. Hence, we get the following.

Theorem 2. *If $|\Gamma| \geq 2$, k-policies incur a blowup in $O\left(|\Gamma|^k\right)$. If $|\Gamma| = 1$, k-policies incur a blowup in $O(k)$.*

Counting Reward Automata policies A policy trained with access to a CRA (*CRA policy*) has access to the values stored in all counters at each timestep. To evaluate the blowup in the size of the CRA policy, we analyse the number of possible combinations of counter values, which we call "counter configurations".

First of all, for any DPDA recognising some language L, it is possible to define a CM that recognises the same language by simulating the DPDA. The set of states and the state-transition function of the CM are the same as those of the DPDA. In order to simulate the stack, we need to encode each possible stack string in a counter configuration. As we noted in the discussion of MDP-pdRM policies above, the cardinality of the stack language, and thus the number of counter configurations needed, is in $O(|\Gamma|^{nm(e+1)})$ (where m and e are as in the argument for Theorem 1). This implies the following.

Theorem 3. *For any task representable as a DCFL, there always exists a CRA which encodes it and incurs a blowup of $O\left(|\Gamma|^{nm(e+1)}\right)$ in the size of the CRA policy.*

The above result gives an upper bound on the number of possible counter combinations. We note that there is no general tight lower bound on the number of counter configurations for counter machines with k counters (k-CM) to recognise arbitrary DCFLs. Below, we give an example of a DCFL for which any k-CM recognising it requires at least

exponentially many (in the input's length) counter configurations.

Let $L_p = \{\sigma x \sigma^R \mid \sigma \in \{0,1\}^*\}$ be a "marked palindrome" language, where σ^R is the string σ but reversed. We show that, for any k-CM that recognises L_p, the number of counter configurations is at least exponential in the length of the input word. We recall an argument given in the proof of Theorem 1.3 in (Fischer, Meyer, and Rosenberg 1968). For any k-CM to recognise L_p, it must be the case that after reading the first half of the word (i.e., before the "mark" x), the k-CM must be in different counter configurations after reading two distinct strings σ_1, σ_2 of length m. This can happen if and only if the number of possible counter configurations (after reading words of length m) is in $\Omega(2^m)$, as we need to encode each string in a tuple of natural numbers. However, in the (worst) case that the string does not contain the mark x, the CM must encode the entire string of length n in a combination of counter values. Therefore, in this second case the number of counter configurations is in $\Omega(2^n)$. Thus, we obtain the following result on the size of CRA policies.

Theorem 4. *There is a task representable by a DCFL for which any CRA that encodes it incurs a blowup of $\Omega(2^n)$ in the size of the CRA policy.*

In summary, pdRMs can incur an exponential, with respect to the episode's length, blowup in the size of policies. On the other hand, for k-policies the blowup is polynomial (and linear when $k = 1$) with respect to the cardinality of the stack alphabet. For CRAs, we can always obtain an upper bound for any task expressible as a DCFL by simulating a pdRM for the task (Theorem 3).

We have seen a case in which a CRA policy incurs a blowup which is at least exponential in the length of the episode (Theorem 4). However, in principle there is no tight lower bound on the blowup for policies based on CRAs, and pdRMs. For example, it is easy to see that the number of strings in the stack language of a pdRM which recognises the language $\{0^n 1^n \mid n \in \mathbb{N}\}$ is linear in the length of the input word. Similarly, one can prove the same bound for CRAs. Therefore, in this case, both policies and CRA policies would incur only a linear (in the episode's length) blowup in the policies' size. Because of this, whether CRAs or pdRMs hould be used to encode a given task should be decided on a case-by-case basis. However, if k-policies (for a given k) are sufficient to learn optimal behaviours with respect to the task (see Section 4), then k-policies should be preferred to policies if the policy size is to be minimised.

6 Exploiting Counterfactual Experiences

RMs were designed, primarily, with two objectives in mind: to provide a normal-form representation for temporally extended reward functions specified natively in RMs or translated from various languages, and to improve sample efficiency of RL via exploitation of the structure of that normal-form representation. The latter was achieved through the development of off-policy RL algorithms that performed counterfactual reasoning over the automata structure to consider experiences in different automata states. This approach was born out in algorithms such as QRM and CRM which operated in tabular and deep learning settings (Toro Icarte et al. 2018; Toro Icarte et al. 2022), with identical behaviour in tabular domains. At each timestep, these algorithms augment training with the *real* experience with a set of additional synthetic *counterfactual* experiences that correspond to the same (s, a, s') transition experienced in a counterfactual (different) state of the RM. A modified version of this algorithm and approach was proposed by Bester et al. (2024) for use with CRAs.

In this section, we introduce an extension of CRM, which we dub *CpRM*, that can be used with pdRMs. To exploit pdRMs, in addition to considering a counterfactual state of the pdRM, CpRM also changes the string on the stack to produce the counterfactual experiences. However, as we noted in Section 5, the number of stack strings can be exponential in the length of the episode. In practice, this can make CpRM extremely slow given the number of counterfactual updates it would make at each timestep. Moreover, observe that in some cases it is even possible that some stack strings are never observed. Because of this, we designed CpRM so that it uses only the set of observed stack strings to produce the counterfactual experiences.

A version of CpRM for policies is shown in Algorithm 1 (the approach can be adapted to k-policies by modifying action sampling and policy updates; see Appendix D in (Varricchione et al. 2025)). Compared to CRM, the main changes in CpRM lie in lines 2, 6, and 10. In line 2, we initialise the set of observed stack strings $\mathcal{O}$; in line 6 we add the current stack string ζ to $\mathcal{O}$ (obviously, if $\zeta \in \mathcal{O}$ already then this operation changes nothing); finally, in line 10 we cycle through the observed stack strings, and use these to generate the counterfactual experiences. The rest of the algorithm is effectively identical to the original CRM, making it usable with any off-policy RL algorithm. In Section 7 we provide results for agents trained with Q-learning augmented with CpRM.

7 Experimental Evaluation

In this section we provide empirical results we have obtained from various experiments in five domains.[1] One of the five domains is taken from (Bester et al. 2024) to allow comparison of pdRMs with CRAs. All domains were implemented using the Gymnasium framework (Towers et al. 2024). For discrete domains, we have trained agents using Q-learning (Watkins and Dayan 1992), both in its vanilla form and with counterfactual updates using CpRM. For the continuous domain we have trained agents with Proximal Policy Optimization (PPO) (Schulman et al. 2017) and its recurrent variant, using, respectively, the STABLE-BASELINES3 (Raffin et al. 2021) and SB3 CONTRIB implementations.

During experiments, agents were trained and then evaluated by periodically running 10 test episodes after a domain-specific number of training episodes. For all experiments we plot the rewards obtained by the agents in the test episodes, normalised between 1 and -1. Specifically, we plot the median rewards with lines, and the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles of

[1]The code is available at https://github.com/giovannivarr/pushdown-reward-machines.

Algorithm 1 Q-learning with CpRM (policy)

Input: MDP-pdRM $\mathcal{T}$, num_episodes

Initialise $\tilde{q}(s, u, \zeta, a)$ arbitrarily for each $s \in S, u \in U, \zeta \in \Gamma^*, a \in A$
$\mathcal{O} \leftarrow \{\}$ {Set of observed stack strings}
for $\ell \leftarrow 0$ **to** num_episodes **do**
  $s \leftarrow$ EnvInitialState(), $u \leftarrow u_0$ and $\zeta \leftarrow Z$
  while s is not terminal **and** $u \notin F$ **do**
    $\mathcal{O} \leftarrow \mathcal{O} \cup \{\zeta\}$
    Sample action a using policy derived from $\tilde{q}$ (e.g., ϵ-greedy) given current state $\langle s, \zeta, u \rangle$
    Take action a and observe the next state s'
    for pdRM state $u_c \in U$ **do**
      for stack string $z_c\zeta_c \in \mathcal{O}$ **do**
        $\langle u'_c, \zeta'_c \rangle \leftarrow \delta_u(u_c, L(s, a, s'), z_c)$
        $r_c \leftarrow \delta_r(u_c, L(s, a, s'), z_c)$
        if s' is terminal **or** $u'_c \in F$ **then**
          $\tilde{q}(s, u_c, z_c\zeta_c, a) \xleftarrow{\alpha} r_c$
        else
          $\tilde{q}(s, u_c, z_c\zeta_c, a) \xleftarrow{\alpha} r_c + \gamma \max_{a' \in A} \tilde{q}(s, u'_c, \zeta'_c\zeta_c, a)$
    Update pdRM configuration to $\langle u', \zeta' \rangle \dashv_{L(s,a,s')} \langle u, \zeta \rangle$

the rewards as shadowed areas. For details on the experimental setup, specifications of the machines used, and the pdRMs we have used, we refer the reader to Section E of the Appendix (Varricchione et al. 2025).

7.1 LETTERENV

The LETTERENV domain is a domain introduced in the original paper on counting reward automata (Bester et al. 2024). The environment is a gridworld where the agent can observe three different events A, B, and C in specific cells. Initially, only the events A and C can be observed. Every time that the agent observes the event A by visiting the corresponding cell, there is a probability of $\frac{1}{2}$ that the cell becomes labelled with the event B from that timestep onwards. The task consists in repeatedly observing the event A until the agent observes the event B, and then observing the event C for the same number of times that the event A was observed. We include this domain to show that pdRMs can be used to obtain comparable results to CRAs when encoding tasks representable in deterministic context-free languages (DCFLs).

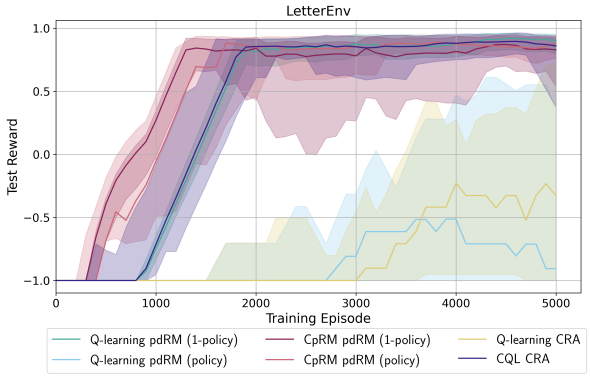

Figure 2: LETTERENV results, comparing agents trained with a pdRM and with a CRA. This shows how pdRMs can be used to encode part of the tasks encodable by CRAs.

For this experiment, we have used the implementation and experimental setup from Bester et al.'s Github repository as of May 12th 2025.[2] Additional documentation is also available online.[3] We trained six agents. Two were trained with the CRA from the repository of Bester et al., one using Q-learning and the other using CQL, a variant of the original QRM algorithm (Toro Icarte et al. 2022) that uses counterfactual experiences to improve sample efficiency, adapted by Bester et al. (2024) for CRAs. We refer to the two CRA-based agents as the *Q-CRA agent* and *CQL-CRA agent* respectively. For the pdRM-based agents, we trained a 1-policy and a policy agent with vanilla Q-learning, and a 1-policy and a policy agent with CpRM Q-learning. We refer to these agents as the *1-pdRM agent*, *pdRM agent*, *1-CpRM* agent, and *CpRM agent* respectively, and write *-pdRM* and *-CpRM agents* to refer to all agents trained with the respective variant of Q-learning.

The results of the experiment are shown in Fig. 2. As can be seen, the agents that achieve the best performance are the -CpRM agents, the 1-pdRM agent, and the CQL-CRA agent. In contrast, the Q-CRA and the pdRM agents manage to increase their accrued rewards only towards the end of the training episodes; however, they both cannot match the performance of the other agents (although, given enough time, they will eventually reach the same performance). For the agents trained with the pdRM, the results suggest that the smaller state space of the 1-pdRM agent allows it to more easily learn to achieve the task compared to the pdRM agent; moreover, the smaller state space allow the 1-pdRM agent to converge as fast as the CQL-CRA agent, which is trained with counterfactual experiences. Note how the 1-CpRM and the CpRM agents converged faster than the rest, and how the latter agent achieved the task as opposed to the pdRM agent which did not. This shows that CpRM can increase sample efficiency. The results suggest that pdRMs can be used in place of CRAs when tasks can be encoded in DCFLs.

7.2 1-TREASUREMAZE

In the 1-TREASUREMAZE environment, the goal of the agent is to navigate a maze to find a treasure, retrieve it, and then return to the initial location by following the reversed path it took to find the treasure. For a detailed explanation of the environment, we refer the reader to the example in Section 3. The aim of this experiment is to show there are cases where pdRMs are more sampe efficient than CRAs.

For this experiment, we trained the same agents as in the LETTERENV experiment. The CRA encodes the task by encoding the path the agent is following to reach the treasure,

[2]https://github.com/TristanBester/counting-reward-machines
[3]https://crm-74a68705.mintlify.app

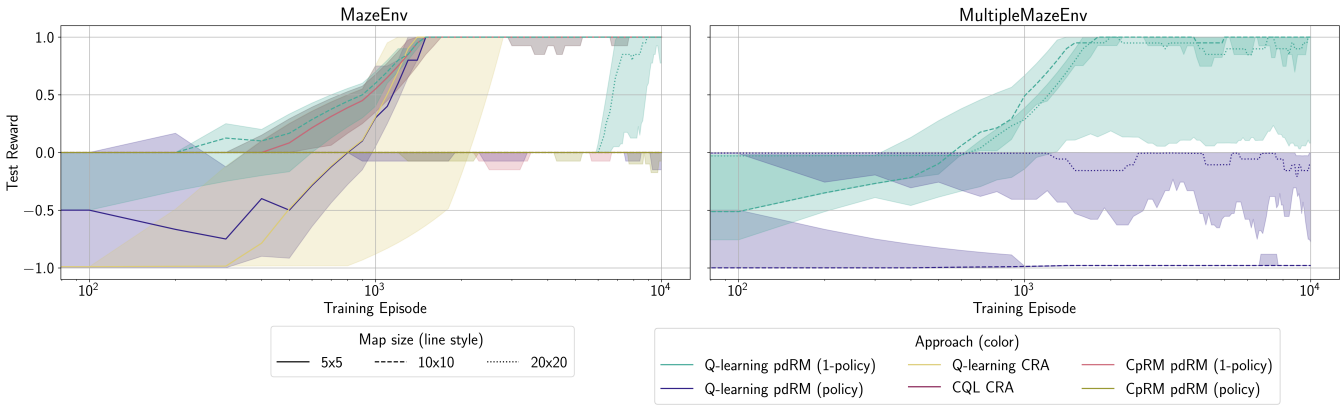

Figure 3: 1-TREASUREMAZE (left) and MULTIPLETREASUREMAZE (right) results. We provide individual plots for each maze in Section E of the Appendix in (Varricchione et al. 2025). In both plots, each maze is identified by a line style, and each agent by a colour. The 1-TREASUREMAZE plot shows how the 1-pdRM agent could achieve the task on all mazes, whereas the pdRM and the Q-learning CRA agents only on the smallest maze. The -CpRM agents only managed to achieve the task in the smallest maze; in the other cases training timed out due to the time required to perform the policy updates. The CQL-CRA agent never managed to achieve the task. For the MULTIPLETREASUREMAZE experiment, we include only results from the 1-pdRM and pdRM agents trained with vanilla Q-learning. As can be seen, on mazes the 1-pdRM agent learnt to achieve the task, whereas the pdRM agent did not. We believe this is due to the smaller size of the 1-policy.

similarly to the CM for the marked palindrome language in Section 5. At each step, the counters of the CRA are updated so that their configuration correctly encodes the path taken so far. As there are four possible directions identifying each step in the path, the number encoding a path is the translation in base 10 of a number in base 4. We define 0, 1, 2, and 3 to be respectively the directions $\mathsf{u}, \mathsf{d}, \mathsf{l}$, and r. Thus, at step i, we add to the current encoding of the path the value 4^i times the direction's value. In order to add this value, note how the CRA must repeatedly add 4's, as it can only add constant values in each of its transitions.

Three different mazes were used: a 5×5 maze, a 10×10 maze, and a 20×20. In this experiment, we end episodes when either the maximum number of timesteps has elapsed, or a limit on wall clock time has been reached. In the left plot in Fig. 3, we show the results obtained on all mazes for the 1-TREASUREMAZE experiment. We identify each agent with a colour, and each maze with a line style. In Section E of the Appendix (Varricchione et al. 2025), we provide further details on the maximum timesteps and wall clock limit per episode, and three further plots, each containing the results of the agents on each of the three mazes.

As can be seen (left plot in Fig. 3), all the -pdRM agents managed to achieve the task in the 5×5 maze. However, only the 1-pdRM agent managed to learn to also achieve the task in the 10×10 and 20×20 mazes. We believe this is due to the fact that the state space for the 1-pdRM agent is much smaller than that for the pdRM agent. The -CpRM agents could not learn to achieve the task as the training episodes ended prematurely due to the wall clock time limit. This is because of the number of counterfactual updates, which in this scenario is large given the size of the stack language. Note that it can be shown using the approach in Section 4 that optimal 1-policies learnt by the 1-pdRM agent have the same state values as optimal pdRM and -CpRM policies.

The Q-CRA agent learned to achieve the task in the 5×5 maze, but not in the larger mazes. We believe the agent could not learn to achieve the task in the larger mazes because the counter values incur an exponential blowup with respect to the number of elapsed timesteps, resulting in an exponential number of operations per timestep and the episodes to end prematurely due to the wall clock time limit. Finally, the CQL-CRA agent never learnt to achieve the task. In addition to having the same issue with respect to the exponential blowup in the number of CRA operations that we observed in the Q-CRA agent, the CQL-CRA agent's training was further slowed down by the counterfactual policy updates. There are exponentially many counterfactual policy updates as the number of possible combinations of counter values is exponential in the maximum number of timesteps (due to an argument similar to the one we have given to establish Theorem 4).

In summary, the experiment shows that there are scenarios where pdRMs are more suited to encode tasks and train agents than CRAs. Note that we do not claim this to always be the case, and believe that it should be decided on a case-by-case basis which of the two machines is more appropriate.

7.3 MULTIPLETREASUREMAZE

The MULTIPLETREASUREMAZE environment is a variant of the previous environment where the agent has to retrieve multiple treasures. Before retrieving the treasures, the agent has to first find an intermediate "safe" location. After finding a safe location, the agent starts looking for treasures: as soon as it finds one, it must return to the safe location by following the reversed path it took to find such treasure. This is repeated until the agent finds all treasures, after which the agent observes a special event informing it that it has found all treasures. The agent must then return from the safe location to the initial exit location, by following the reversed path it took to reach the safe location. We include this experiment to show how pdRMs can enable learning of

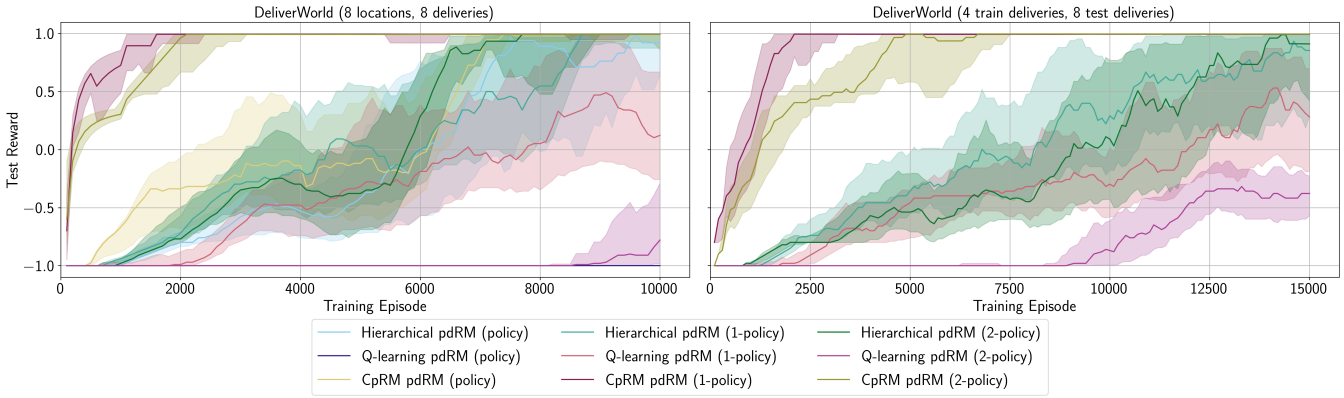

Figure 4: DELIVERWORLD results. Left plot: agents performed 8 deliveries during training and testing (DELIVERWORLD-8). Right plot: agents performed 4 deliveries during training episodes and 8 during testing episodes (DELIVERWORLD-4-8). For DELIVERWORLD-8, the only agents that consistently achieve the task at the end of training are the -CpRM agents, the Q-learning 1-pdRM agent and the hierarchical 1-pdRM and 2-pdRM agents. The hierarchical pdRM agent manages to achieve the task but not as consistently. The other Q-learning agents show the worse performance out of all agents, with the pdRM agent flatlining at a reward of -1. On the other hand, for DELIVERWORLD-4-8, only the -CpRM agents and the hierarchical agents eventually learn to consistently achieve the task. This shows how pdRMs can help in obtaining agents that complete longer tasks than the ones they were trained in.

more complex tasks.

In this experiment we have trained only the -pdRM agents. We have excluded the -CpRM and CRA-based agents because the wall clock time limit would have stopped the training episodes prematurely as in the 1-TREASUREMAZE environment. Only a 10×10 maze and a 20×20 one were used. Note that, for this experiment, the maximum number of timesteps per episode for each maze is larger than in the 1-TREASUREMAZE experiments.

As can be seen (right plot in Fig. 3), the 1-pdRM agent managed to eventually achieve the task in both mazes. Given the longer episodes compared to the 1-TREASUREMAZE domain, the agents were also able to learn to consistently achieve the task in a relatively small number of episodes. This shows that a pdRM can also be used in tasks where more complex operations with the stack are required. On the other hand, the pdRM agent did not manage to learn to complete the task by the end of training in either maze. We believe that this is mainly due to the size of the policy, which seems to be too large for the agent to learn in this more complex task.

The results show that, thanks to the flexible amount of stack information that pdRMs allow in defining the agent's policy, pdRMs allow us to train agents in more complex tasks.

7.4 DELIVERWORLD

In DELIVERWORLD, the agent that is supposed to deliver packages to locations. Each location is assigned a "type" (e.g., shopping mall, clothing store, etc.), and there can be multiple locations of the same type. At the start of each episode, the agent observes an event which determines the sequence of delivery location types it needs to visit to achieve the task. This sequence is chosen randomly from a set. The aim of this experiment is to illustrate that there are scenarios where it is beneficial for the agent to have access to more than the top symbol on the stack. In this experiment, we show how CpRM and a hierarchical approach can improve sample efficiency. Moreover, in a variant of the experiment, we show how pdRMs allow training of agents that can perform a larger number of subtasks in testing than in training episodes.

In this experiment we have compared three sets of agents, all trained with access to the same pdRM: one was trained with vanilla Q-learning, one with CpRM Q-learning, and the last with a hierarchical approach.

For the hierarchical approach, we have adapted the hierarchical algorithm proposed in (Toro Icarte et al. 2022). In our pdRM-based version, the meta-policy has access to the current MDP state, pdRM state and the stack of the pdRM. The options' policies have access to the current MDP state, pdRM state, and only the topmost symbol on the pdRM stack.

We have used a 20×20 grid, with two possible setups for the number of deliveries during training and the number of deliveries during testing. In the first, which we call DELIVERWORLD-8, the agents perform 8 deliveries during training and testing episodes. In the second, which we dub DELIVERWORLD-4-8, the agents perform 4 deliveries during training episodes and 8 during testing episodes. In DELIVERWORLD-8, we trained three agents per approach: a 1-policy agent, a 2-policy agent (denoted *2-pdRM* agent and *2-CpRM* agent for the Q-learning approaches), and a policy agent. In DELIVERWORLD-4-8, we have trained two agents per approach: a 1-policy and a 2-policy agent.

Figure 4 shows the plots for both experiments. In both plots, we can clearly see that the most sample efficient agents are the -CpRM ones, followed by the hierarchical ones, and with the vanilla Q-learning ones coming last.

In both scenarios, the -CpRM agents clearly outperform all other agents in speed of convergence. Notice how CpRM and the hierarchical approach improve the performance compared to the pdRM agent in the DELIVERWORLD-8 task: when trained with vanilla Q-learning, the pdRM agent never learns to achieve the task; however the CpRM and hierarchical

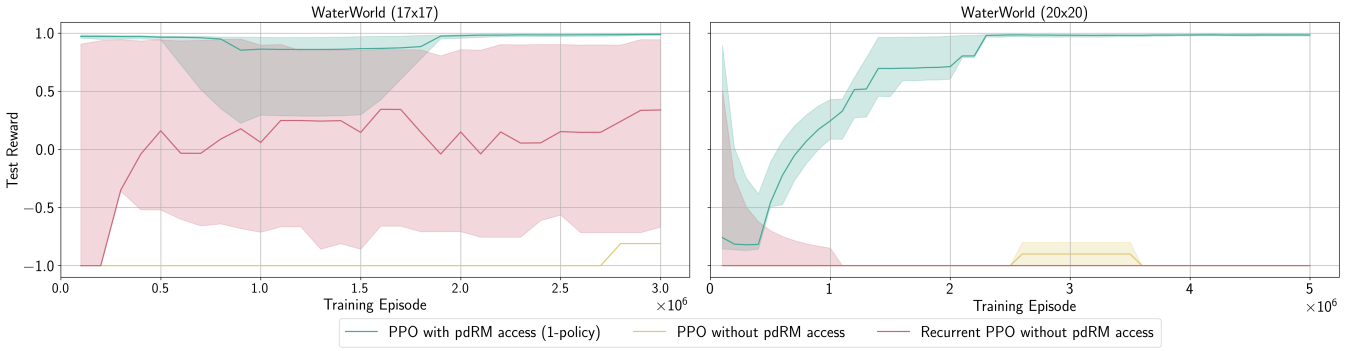

Figure 5: WATERWORLD results, 17×17 map (left) and 20×20 map (right). We compare the performance of a 1-pdRM agent trained with PPO against that of an agent trained with PPO and one of an agent trained with recurrent PPO. In the 17×17 map, the 1-pdRM agent is able to achieve the task very quickly. Only the recurrent PPO agent manages to considerably improve its performance, but does not match that of the 1-pdRM agent. Similarly to the results of the 17×17 map, in the 20×20 map the 1-pdRM agent is the only one that consistently achieves the task; significanly outperforming the other agents which do not improve their performance. These two plots show how the pdRM is crucial in training agents to achieve this task.

agent both do. This shows how the hierarchical approach and, when the stack language is not too large, CpRM can greatly increase sample efficiency.

In the hierarchical approach, we can see that in the plot of DELIVERWORLD-8 (left in Fig. 4) the 2-pdRM converges faster. We believe that this is due to the fact that, by having access to the top two symbols, the agent learns to correctly plan which delivery location type to visit first. On the other hand, in the plot of DELIVERWORLD-4-8 (right in Fig. 4), the 1-pdRM and the 2-pdRM seem to converge at the same pace. For this setup, we think that the fact that the agent needs to perform fewer deliveries during training penalizes agents with larger policy state spaces. Shorter training episodes (compared to the testing ones) imply that the agents can explore fewer states during training, thus leading to worse performance for the 2-pdRM agent compared to the results it obtained in DELIVERWORLD-8.

Finally, for the Q-learning agents, the 1-pdRM agent has the best performance in both plots. In principle, the other two Q-learning agents should be able to learn a policy that is at least as good as that of the 1-pdRM agent. Note that of the Q-learning agents, the only one which is guaranteed to achieve an optimal policy in the limit is the pdRM agent (see Section 4). However, given the state-space complexity of the policies and the limited number of episodes, the 2-pdRM and pdRM agents do not learn to achieve the task.

Interestingly, from the plot of DELIVERWORLD-4-8, we can see that agents trained with pdRMs learn policies that, even though trained in smaller instances, performed adequately in larger testing instances. The -CpRM agents are able to achieve the test task relatively quickly, whereas the hierarchical agents eventually manage to achieve the test task by the end of training. Moreover, although the Q-learning agents do not achieve the test task, their testing rewards increase over time. We conjecture that with a longer training they should also converge to policies that achieve the test task. This suggests that pdRMs can, to a certain degree, help agents to learn policies that can achieve longer tasks than the ones that they are trained in.

7.5 WATERWORLD

The last domain is based on the WATERWORLD domain (Karpathy 2015). The task consists of two steps. In the first, the agent must touch 8 balls. Each of these balls is labelled with a (unique) number from 2 to 9: whenever the agent touches the ball identified with i, the pdRM pushes the parity of i to the stack and the ball disappears. Once all 8 balls are touched, the pdRM moves to a new state and the second phase of the task begins. In this phase, the agent has to touch one of two other balls, numbered 0 and 1 determined by the topmost parity on the stack. When the agent correctly touches the ball, it is moved to a new random location and the topmost symbol of the stack is popped. The task is considered achieved when the stack has been emptied.

The aim of this experiment is to evaluate whether pdRMs can provide an advantage in continuous domains when using a deep algorithm. We trained three agents using Proximal Policy Optimization (PPO) (Schulman et al. 2017): the first has access to the pdRM that encodes the task (and is trained with a 1-policy), the second does not, and the third also does not have access to the pdRM, but is trained with *recurrent* PPO, i.e., an implementation of PPO where both the actor and critic networks employ *long short-term memory* (LSTM) units (Hochreiter and Schmidhuber 1997). LSTMs are a type of recurrent neural network, meaning they can capture sequential dependencies in data. Intuitively, using an LSTM should help the agent "remember" important events that have happened during the episode. Note that, as PPO is an on-policy algorithm, CpRM cannot be used. Instead, we used Soft Actor Critic (SAC) in conjunction with CpRM; however, the training of the agents did not terminate, as the server we used to run the WATERWORLD experiments had a time limit on each job. We therefore do not report the results of agents trained with the counterfactual approach for this experiment.

We have trained the agents in two different maps, a 17×17 one and a 20×20 one, and the results are shown in the left

and right plot of Fig. 5 respectively.

As can be seen, in the 17×17 map (left in Fig. 5) the agent trained with the pdRM was able to achieve the task very quickly, and while it showed a small decrease in performance between the 5^{th} and 20^{th} episodes, it recovered and managed to outperform its initial results. The agent trained with recurrent PPO managed to improve its policy, but by the end of training its performance still showed high variance. Finally, the vanilla PPO agent only showed an increase in performance at the end of training, however, its resulting performance was not comparable to that of the 1-pdRM agent or the recurrent PPO agent. In the 20×20 map (right in Fig. 5) neither the recurrent PPO nor vanilla PPO agents learnt a policy that achieved the task in the test episodes. However, the agent with access to the pdRM could learn a policy that consistently achieved the task in the test episodes from the 20^{th} training episode. The results suggest that, in this scenario, using a pdRM is crucial in training and in increasing sample efficiency, performing better than deep algorithms employing recurrent networks.

8 Related Work

The literature on reward machines is now quite large. We focus specifically on work in which reward machines are modified in a way that is similar to our approach. The approach of Bester et al. (2024) is closest to ours and is discussed in Section 5. The *task monitors* of Jothimurugan, Alur, and Bastani (2019) are automata with numeric registers (and so are similar to Bester et al.'s counting reward automata), but the purpose of the registers was not to define more complicated temporal patterns but to keep track of the quantitative *degrees* to which subtasks had been completed and constraints satisfied (so as to provide shaped rewards). Furelos-Blanco et al. (2023) augment reward machines by introducing a *hierarchy*. In such hierarchies, RMs are able to call other RMs during execution. However, as they assume that there is always a *leaf* RM that cannot call other RMs, and that each RM cannot be called by itself, even via recursive calls, they cannot express all tasks representable in DCFLs. Another modification of reward machines, *First-Order Reward Machines* (FORMs), is proposed by Ardon et al. (2025), where transitions are labelled by first-order logic formulas. However, FORMs only increase the expressivity of the events labelling transitions in RMs. On the other hand, our increased expressivity lies in the set of tasks that can be encoded by pdRMs, which is strictly larger than the set of tasks encodable by RMs.

Other work has proposed approaches where RL agents are trained to achieve tasks that can be represented as deterministic context-free languages. In particular, Hahn et al. (2022) introduce recursive reinforcement learning, where *recursive* MDPs (RMDPs) model the environment in which agents act. RMDPs generalise MDPs in that they consist of a set of MDPs where each MDP can "call" other MDPs. By keeping a stack of calls, RMDPs can encode tasks representable as DCFLs. Indeed, the authors specifically mention *context-free reward machines* as a possible application of recursive RL; however they do not provide a formal argument. While it would be interesting to formally connect recursive RL to pdRMs, this lies outside of the scope of this work and we leave it to future research.

9 Conclusions

In this paper, we have presented *pushdown reward machines*, an extension of reward machines which can encode non-Markovian tasks representable as deterministic context-free languages. Compared to reward machines, pdRMs are thus able to encode a strictly larger set of tasks. We have proposed two policy types for pdRMs, one where the agent has access to the full stack (*policies*), and one where it can access only its top k symbols (k-*policies*). In general, the state values of an optimal k-policy might not be as high as those of a policy. We described a procedure to check whether an MDP and a pdRM are such that the two policy types have the same optimal state values. We have also compared pdRMs to counting reward automata (Bester et al. 2024), another extension of reward machines capable of encoding tasks representable as any recursively enumerable language. We showed that, when an agent trained with a pdRM has access only to the top symbol of the pdRM's stack, the size of a 1-policy can be exponentially smaller (with respect to the episode's maximum length) than the size of a policy an agent trained with a CRA for the same task. Finally, in the experimental evaluation, we have shown how pdRMs can be used in practice. We have seen how in certain scenarios it is more convenient to use pdRMs than CRAs. We have also provided counterfactual and hierarchical algorithms specifically tailored for pdRMs, and saw how they can increase convergence speed. Finally, we have used pdRMs in a continuous domain, and showed how they can outperform state-of-the-art algorithms employing recurrent neural networks.

There are several directions for future work. First, we plan to investigate the mixed performance of CpRM in the 1-TREASUREMAZE and WATERWORLD tasks. It would also be interesting to automatically synthesise pdRMs, as was done with reward machines via, e.g., logic formalisms (Camacho et al. 2019; Varricchione et al. 2023), search and learning (Toro Icarte et al. 2019; Toro Icarte et al. 2023; Furelos-Blanco et al. 2023; Hasanbeig et al. 2024) or planning (Illanes et al. 2019; Varricchione et al. 2024). As deterministic pushdown automata are the underlying structure of pdRMs, we think LR(k) grammars (Knuth 1965) could be good candidates to synthesise pdRMs, as they can be easily translated into DPDAs (Aho and Ullman 1973). Finally, as the stack can provide the agent with even further memory, pdRMs can be an interesting alternative approach to RMs (Toro Icarte et al. 2019; Toro Icarte et al. 2023) in dealing with partially observable environments.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. The second and final authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canada CIFAR AI Chairs Program. Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies

sponsoring the Vector Institute for Artificial Intelligence (www.vectorinstitute.ai/partners). Finally, the second and final authors thank the Schwartz Reisman Institute for Technology and Society for providing a rich multi-disciplinary research environment.

References

Aho, A. V., and Ullman, J. D. 1973. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall Englewood Cliffs, NJ.

Ardon, L.; Furelos-Blanco, D.; Parac, R.; and Russo, A. 2025. FORM: Learning expressive and transferable first-order logic reward machines. *arXiv preprint arXiv:2501.00364*.

Bester, T.; Rosman, B.; James, S.; and Tasse, G. N. 2024. Counting reward automata: Sample efficient reinforcement learning through the exploitation of reward function structure. *arXiv preprint arXiv:2312.11364*.

Camacho, A.; Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and 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-19*, 6065–6073.

Chomsky, N., and Schützenberger, M. P. 1959. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26. Elsevier. 118–161.

Feinberg, E. A. 2011. Total expected discounted reward MDPs: Existence of optimal policies. In Cochran, J. J., ed., *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Hoboken, NJ.

Fischer, P. C.; Meyer, A. R.; and Rosenberg, A. L. 1968. Counter machines and counter languages. *Mathematical Systems Theory* 2(3):265–283.

Fischer, P. C. 1966. Turing machines with restricted memory access. *Information and Control* 9(4):364–379.

Furelos-Blanco, D.; Law, M.; Jonsson, A.; Broda, K.; and Russo, A. 2023. Hierarchies of reward machines. In *International Conference on Machine Learning*, 10494–10541. PMLR.

Hahn, E. M.; Perez, M.; Schewe, S.; Somenzi, F.; Trivedi, A.; and Wojtczak, D. 2022. Recursive reinforcement learning. *Advances in Neural Information Processing Systems* 35:35519–35532.

Hasanbeig, H.; Jeppu, N. Y.; Abate, A.; Melham, T.; and Kroening, D. 2024. Symbolic task inference in deep reinforcement learning. *J. Artif. Intell. Res.* 80:1099–1137.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Illanes, L.; Yan, X.; Toro Icarte, R.; and McIlraith, S. A. 2019. Symbolic planning and model-free reinforcement learning: Training taskable agents. In *4th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM-19)*, 191–195.

Jothimurugan, K.; Alur, R.; and Bastani, O. 2019. A composable specification language for reinforcement learning tasks. In *Advances in Neural Information Processing Systems 32*, 13021–13030.

Karpathy, A. 2015. REINFORCEjs: Waterworld demo. https://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html.

Knuth, D. E. 1965. On the translation of languages from left to right. *Information and Control* 8(6):607–639.

Minsky, M. L. 1967. *Computation*. Prentice-Hall Englewood Cliffs.

Puterman, M. L. 2014. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M.; and Dormann, N. 2021. Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research* 22(268):1–8.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and 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 2018)*, 2112–2121.

Toro Icarte, R.; Waldie, E.; Klassen, T. Q.; Valenzano, R. A.; Castro, M. P.; and McIlraith, S. A. 2019. Learning reward machines for partially observable reinforcement learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019*, 15497–15508.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and McIlraith, S. A. 2022. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research* 73:173–208.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; Castro, M. P.; Waldie, E.; and McIlraith, S. A. 2023. Learning reward machines: A study in partially observable reinforcement learning. *Artificial Intelligence* 323:103989.

Towers, M.; Kwiatkowski, A.; Terry, J.; Balis, J. U.; De Cola, G.; Deleu, T.; Goulao, M.; Kallinteris, A.; Krimmel, M.; KG, A.; et al. 2024. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*.

Varricchione, G.; Alechina, N.; Dastani, M.; and Logan, B. 2023. Synthesising reward machines for cooperative multi-agent reinforcement learning. In Malvone, V., and Murano, A., eds., *Proceedings of the 20th European Conference on Multi-Agent Systems (EUMAS 2023)*, 328–344. Springer Nature Switzerland.

Varricchione, G.; Alechina, N.; Dastani, M.; and Logan, B. 2024. Maximally permissive reward machines. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI 2024)*, 1181–1188. IOS Press.

Varricchione, G.; Klassen, T. Q.; Alechina, N.; Dastani, M.; Logan, B.; and McIlraith, S. A. 2025. Pushdown reward machines for reinforcement learning. *arXiv preprint arXiv:2508.06894*.

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