

https://dspace.org/

Counting Reward Automata: Exploiting Structure in Reward Functions Expressible in Decidable Formal Languages

Bester, Tristan

2024-07
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/45658
Bester, Tristan. (2024). Counting Reward Automata: Exploiting Structure in Reward Functions Expressible in Decidable Formal Languages. [Master's dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/45658

Downloaded from a DSpace repository. https://dspace.org/

Counting Reward Automata: Exploiting Structure in Reward Functions Expressible in Decidable Formal Languages

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Tristan Bester
2090454

Supervised by:
Prof. Benjamin Rosman
Dr. Steven James
Mr. Geraud Nangue Tasse

July 29, 2024



A thesis submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

Abstract

In general, reinforcement learning agents are restricted from directly accessing the environment model. This restricts the agent's access to the environmental dynamics and reward models, which are only accessible through repeated environmental interactions. As reinforcement learning is well suited for use in complex environments, which are challenging to model, the general assumption that the transition probabilities associated with the environment are unknown is justified. However, as agents cannot discern rewards directly from the environment, reward functions must be designed and implemented for both simulated and real-world environments. As a result, the assumption that the reward model must remain hidden from the agent is unnecessary and detrimental to learning. Previously, methods have been developed that utilise the structure of the reward function to enable more sample-efficient learning.

These methods employ a finite state machine variant to facilitate reward specification in a manner that exposes the internal structure of the reward function. This approach is particularly effective when solving long-horizon tasks as it enables the use of counterfactual reasoning with off-policy learning which significantly improves sample efficiency. However, as these approaches are dependent on finite-state machines, they are only able to express a small number of reward functions. This severely limits the applicability of these approaches as they cannot model simple tasks such as **"fetch a coffee for each person in the office"** which involves counting – one of the numerous properties finite state machines cannot model. This work addresses the limited expressiveness of current state machine-based approaches to reward modelling. Specifically, we introduce a novel approach compatible with any reward function which can be expressed as a well-defined algorithm

We present the counting reward automaton – an abstract machine capable of modelling reward functions expressible in any decidable formal language. Unlike previous approaches to state machine-based reward modelling, which are limited to the expression of tasks as regular languages, our framework allows for tasks described by decidable formal languages. It follows that our framework is an extremely general approach to reward modelling – compatible with any task specification expressible as a well-defined algorithm. This is a significant contribution as it greatly extends the class of problems which can benefit from the improved learning techniques facilitated by state machine-based reward modelling. We prove that an agent equipped with such an abstract machine is able to solve an extended set of tasks. We show that this increase in expressive power does not come at the cost of increased automaton complexity. This is followed by the introduction of several learning algorithms designed to increase sample efficiency through the exploitation of automaton structure. These algorithms are based on counterfactual reasoning with off-policy RL and use techniques from the fields of HRL and reward shaping. Finally, we evaluate our approach in several domains requiring long-horizon plans. Empirical results demonstrate that our method outperforms competing approaches in terms of automaton complexity, sample efficiency, and task completion.

Declaration

I, Tristan Bester, hereby declare the contents of this thesis to be my own work. This thesis is submitted for the degree of Master of Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.



Signature

2024-07-29

Date

Acknowledgements

I would like to thank my supervisors Benjamin Rosman, Steven James and Geraud Nangue Tasse for their continued support throughout this project. Thank you for inspiring me!

To my family and Racquel.

Contents

Preface
Abstract i
Declaration ii
Acknowledgements iii
Table of Contents v

1 Introduction **2**
1.1 Overview 2
1.2 Research Problem 4
1.3 Main Contributions 5
1.4 Thesis Structure 6

2 Background **8**
2.1 Introduction 8
2.2 Reinforcement Learning 9
2.2.1 Markov Decision Processes 9
2.2.2 Non-Markovian Reward Decision Processes 9
2.2.3 Policies and Value Functions 10
2.2.4 Function Approximation 12
2.2.5 Hierarchical Reinforcement Learning 13
2.2.6 Reward Shaping 14
2.3 Formal Language and Automata Theory 14
2.3.1 Chomsky Hierarchy 15
2.3.2 Automata 16
2.3.3 Decidability in Formal Languages 20
2.3.4 Counter Machines 21
2.3.5 Temporal Logic, Automata and Product MDPs 23
2.4 Reward Machines 24
2.4.1 Understanding the Formulation 24
2.4.2 Reward Machine Specification 26
2.5 Conclusion 27

3 Modelling Non-Markovian Reward Functions **28**
3.1 Introduction 28
3.2 The Hierarchy of Non-Markovian Reward 29

3.3 Insufficiency of Reward Machines . . . 30
3.4 Beyond Finite State Machines . . . 31
3.5 Related Work . . . 32
3.5.1 Reward Machines . . . 32
3.5.2 Specification of Reward Models . . . 32
3.5.3 Knowledge Exploitation . . . 33
3.6 Conclusion . . . 34

4 Augmenting Agents with Counter Machines **35**
4.1 Introduction . . . 35
4.2 Counting Reward Automata . . . 36
4.2.1 Formulation . . . 36
4.2.2 Counting Reward Automaton Operation . . . 36
4.2.3 Solving the Example Task . . . 39
4.3 Learning through Counterfactual Experience . . . 40
4.3.1 Product MDP Baseline . . . 40
4.3.2 Counterfactual Experiences . . . 42
4.4 Relationship to Reward Machines . . . 43
4.4.1 Reward Machines – A Special Case of the CRA Framework . . . 44
4.4.2 Episodic Reinforcement Learning . . . 45
4.5 Investigating Practical Considerations . . . 49
4.5.1 Empirical Design . . . 49
4.5.2 Sample Efficiency . . . 50
4.5.3 State Machine Complexity . . . 52
4.6 Function Approximation . . . 54
4.6.1 Existing Approaches . . . 55
4.6.2 Exploiting Counterfactual Experiences . . . 57
4.6.3 Experimental Evaluation . . . 60
4.7 Conclusion . . . 64

5 Hierarchical Reinforcement Learning for Counting Reward Automata **66**
5.1 Introduction . . . 66
5.2 Hierarchical Counting Reward Automata . . . 67
5.2.1 Episodic Reinforcement Learning . . . 71
5.3 Experimental Evaluation . . . 72
5.3.1 *LetterEnv* Results . . . 72
5.3.2 *OfficeWorld* Results . . . 73
5.4 Function Approximation Case . . . 74
5.4.1 Extending the Formulation . . . 74
5.4.2 Experimental Evaluation . . . 75
5.5 Conclusion . . . 77

6 Reward Shaping for Counting Reward Automata **79**
6.1 Introduction . . . 79
6.2 Automated Reward Shaping . . . 80
6.2.1 Formulation . . . 80

6.2.2 Illustrative Example . . . 81
6.2.3 Limitations . . . 83
6.3 Reward Shaping for Counting Reward Automata . . . 84
6.3.1 Formulation . . . 85
6.3.2 Illustrative Example . . . 88
6.4 Experimental Evaluation . . . 91
6.4.1 Tabular Case . . . 92
6.4.2 Function Approximation . . . 94
6.5 Conclusion . . . 95

7 Discussion **97**
7.1 Introduction . . . 97
7.2 Tabular Reinforcement Learning . . . 97
7.2.1 *LetterWorld* . . . 98
7.2.2 *OfficeWorld* . . . 99
7.2.3 Summary . . . 100
7.3 Function Approximation . . . 100
7.3.1 Sparse Reward Model . . . 101
7.3.2 Dense Reward Model . . . 102
7.3.3 Summary . . . 103
7.4 Conclusion . . . 103

8 Conclusion and Future Work **104**
8.1 Future Work . . . 104
8.2 Conclusion . . . 105

A Model Hyperparameters **107**
A.1 *LetterWorld* . . . 107
A.2 *OfficeWorld* . . . 107
A.3 *PuckWorld* Sparse Reward Model . . . 108
A.4 *PuckWorld* Dense Reward Model . . . 109

B Performance Estimation **110**

C Product MDP State Encoding **111**

References **119**

Chapter 1

Introduction

1.1 Overview

Problems involving sequences of tasks are crucial in modern society as they underpin various domains such as autonomous driving, healthcare, finance, and manufacturing. These tasks involve sequences of actions unfolding over time, demanding sustained and coherent behaviour from agents. Consider the simple task of preparing a coffee. This task involves a series of steps, each requiring precise coordination to achieve the desired outcome. Initially, one must measure the coffee grounds and add them to the filter, followed by pouring water into the reservoir and switching on the coffee maker. Each step in this process depends on the successful completion of preceding subtasks, illustrating the intricate nature of temporally extended tasks.

When solving temporally extended tasks, a desirable property of any generally intelligent agent is the ability to learn subtasks in parallel. When presented with environmental experience, we would expect a generally intelligent agent to use the given information to learn about all subtasks of interest, as opposed to only the subtask currently being executed. Consider the case in which the agent switches on the coffee maker while attempting to fill the reservoir with water. In this instance, we would expect a generally intelligent agent to simultaneously use the acquired information to learn about many subtasks. More specifically, it should not only learn that the executed sequence of actions does not lead to the reservoir being filled with water but also that this sequence of actions results in the coffee maker being switched on.

Reinforcement learning (RL) has proven to be a promising approach for developing generally intelligent agents. Despite recent successes in the field, ranging from applications in video games [Badia *et al.* 2020] to complex robotic manipulation tasks [Kalashnikov *et al.* 2018; Kumar *et al.* 2016], several shortcomings continue to hinder the applicability of RL to real-world problems. A key issue is sample efficiency – these methods often require millions of samples to learn optimal behaviours. While it is conceivable that such an approach can be carried out in a simulated environment, it is simply not feasible in the real world. This inefficiency is further exacerbated when agents are required to carry out long-horizon task specifications containing sequences of subtasks

(as would be expected of any generally intelligent agent) [Konidaris and Barto 2007]. Sparse reward models infrequently provide agents with reward signals, resulting in long periods without feedback. When applied to temporally extended tasks, sparse reward models have been shown to significantly reduce the ability of agents. This is a consequence of the long sequences of actions which must be executed before learning signals are received [Arjona-Medina *et al.* 2019]. This makes it difficult for agents to learn as regardless of the amount of data they collect, access to learning signals remains limited.

In recent years, a state machine-based approach known as *Reward Machines* has been introduced [Icarte *et al.* 2018 2022]. This framework facilitates the type of learning characteristic of generally intelligent agents when solving temporally extended tasks. The defining feature of this formulation is that it specifies reward functions through the use of state machines. The information encoded within the structure of these automata enables agents to simultaneously learn about all of the subtasks in a specification while interacting with the environment. This strongly contrasts conventional RL approaches in which agents can only learn about the active subtask being executed. As a result, state machine-based approaches are significantly more sample-efficient for long-horizon tasks than the conventional techniques used in RL. Unfortunately, only a small number of task specifications are compatible with the reward machine formulation. For example, the task **"Go to A the same number of times as B"** cannot be expressed as a reward machine. This is a consequence of the automaton variant used by the formulation for reward modelling. This limitation has significantly inhibited the application of reward machines to real-world tasks of interest.

In this work, we are interested in modelling the learning patterns characteristic of generally intelligent agents when solving temporally extended tasks. As the reward machine formulation offers a restricted framework compatible with this goal, we generalise the approach to support any reward function that can be expressed as a well-defined algorithm [Rogers Jr 1987]. As a result of the increased expressiveness of our formulation, we extend the set of compatible task specifications from those which can be represented as finite state machines to include all tasks which can be implemented by a Turing machine [Yu 1997]. To overcome the aforementioned limitations of reward machines, we replace the automaton at the core of the formulation. Specifically, we define a novel abstract machine known as a *Counting Reward Automaton* capable of modelling any reward function expressible as a decidable formal language. This result is achieved by modelling reward functions through the use of k*-counter automata* as opposed to *finite state machines*, which are used in the reward machine formulation. We show that the state machines produced by our approach are not only more intuitive to specify than reward machines but also significantly simpler both in terms of theoretical and practical measures. Furthermore, we prove that our approach converges to globally as opposed to hierarchically optimal policies.

In addition, we present several learning algorithms designed to increase sample efficiency through the exploitation of counting reward automaton structure. Each of the presented algorithms decomposes task specifications into a number of subproblems by analysing the structure of the automaton. In our first algorithm, this information is

exploited to shape the original reward function automatically. This allows our formulation to overcome the challenges associated with sparse reward models. Furthermore, this decomposition facilitates the introduction of techniques from the field of *Hierarchical Reinforcement Learning*. In our second algorithm, we make use of *options* to model the subproblems associated with a task specification. The structure of the automaton allows these options to be learnt in parallel, which offers the potential for significant improvements in sample efficiency.

We evaluate the effectiveness and applicability of the presented approaches across a diverse array of complex domains, encompassing various intricacies and challenges inherent to each domain. We begin by demonstrating the aforementioned approaches in several tabular domains, each of increasing complexity. Agents are evaluated based on their ability to solve a variety of task specifications only expressible as *non-regular* formal languages including properties which cannot be modelled by current approaches. For example, we solve the task **"fetch a coffee for each person in the office"** which involves counting – one of the numerous properties finite state machines cannot model. Following this demonstration, we show that each of our approaches can solve temporally extended high-dimensional control problems through function approximation. The results show that our approach is not only the most intuitive for task specification but also the most sample-efficient state machine-based approach.

1.2 Research Problem

In the previous section, we discussed the importance of long-horizon tasks, showing that they commonly form part of the decision-making process followed by generally intelligent agents. As the solutions of many significant problems can be expressed naturally as sequences of subtasks, this class of problems is of great practical interest. We described how generally intelligent agents can decouple their learning from the subtask being executed when faced with long-horizon problems. This property of the learning process is desirable as it allows agents to learn about all subtasks of interest in parallel, vastly increasing sample efficiency. If artificially intelligent agents are to reach the ultimate goal of being able to solve long-horizon tasks in reasonable amounts of time, it is necessary to equip them with learning algorithms that facilitate a similar approach to learning. This is an open problem in the field of artificial intelligence.

This work decomposes this problem into three components, which are solved in order. The first component is focused on understanding the limitations of current-state machine-based approaches. As these formulations provide a framework that facilitates the type of learning we are interested in, their limitations must be formally characterised to facilitate further research. Without such a rigorous characterisation, we are unable to study and improve upon these approaches.

After the completion of the first component, the question remains: what type of state machine can be used to overcome these limitations? This is an important problem as it is not necessarily true that finite state machines are sufficient to solve any temporally extended task. We have already shown in Section 1.1 that these automata are insuffi-

cient for simple task specifications such as **"Go to A the same number of times as B"**. This is the main problem we aim to address.

Finally, we consider how the developed automaton variant can facilitate the type of learning we are interested in. More specifically, this part of the problem is focused on creating learning algorithms that can learn subtasks in parallel. This is an essential aspect of the problem as it enables artificially intelligent agents to learn similarly to generally intelligent agents on a much larger set of problems than is currently possible. This component of the problem is focused on extending the usefulness of the developed automaton beyond reward specification into the domain of learning applications.

1.3 Main Contributions

We now describe the main contributions of this work:

1. *Hierarchy of Non-Markovian Reward (Chapter 3)*: We define a novel containment hierarchy for non-Markovian reward models based on the Chomsky hierarchy. This is an important contribution as it presents a principled approach for partitioning the set of all reward models based on their complexity. The introduction of such a classification enables the limitations associated with current state machine-based approaches to be expressed precisely.

2. *Limitation Characterisation (Chapter 3)*: We formalise the role of automata in reward modelling. Specifically, we describe how state machine-based approaches rely on the idea of membership[1] to perform reward assignment. This is used to support the main result of the analysis in which we show that reward machines are a variation of the *Moore machine* formulation. We show that as a direct consequence of this result, reward machines are only compatible with the weakest class of reward models in the previously defined *Hierarchy of Non-Markovian Reward*.

3. *Counting Reward Automata (Chapter 4)*: To overcome the limitations of current state machine-based approaches, we introduce a novel abstract machine known as a *Counting Reward Automaton* capable of modelling any reward function that can be expressed as a decidable formal language. This is a significant contribution as it greatly extends the set of tasks for which artificially intelligent agents can emulate the learning processes followed by generally intelligent agents. To be precise, our formulation extends the set of compatible tasks from the restricted class of regular tasks[2] to the extremely general class of decidable task specifications[3]. In addition, we thoroughly analyse the relationship between counting reward automata and reward machines. Several properties of the relationship, including sample efficiency and ease of specification, are formalised through proofs that illustrate our approach's advantages.

[1] The membership problem is the problem of deciding whether a string belongs to a formal language.
[2] Tasks only expressible as regular languages.
[3] Tasks only expressible as recursive languages.

4. *Hierarchical Reinforcement Learning for Counting Reward Automata (Chapter 5)*: We introduce an approach to increase sample efficiency that is based on hierarchical reinforcement learning. Specifically, this approach exploits the structure of counting reward automata to decompose the problem into several subtasks. These subproblems are modelled as options, which are learnt in parallel as the agent interacts with the environment. This offers the potential for significant increases in sample efficiency based on the structure of the overall problem.

5. *Reward Shaping for Counting Reward Automata (Chapter 6)*: We propose a novel approach to potential-based reward shaping that relies on the internal structure of counting reward automata. This approach exploits the structure of a counting reward automaton to derive a shaping reward function that augments the original reward model. This is an important contribution as it allows our formulation to overcome the challenges associated with sparse reward models without relying on domain knowledge or complex reward engineering.

1.4 Thesis Structure

We begin in Chapter 2 by providing a brief synopsis of the main concepts and formulations used throughout this research. As we are interested in solving tasks in the context of reinforcement learning, we provide a brief overview of the topic. The conventional approach followed in reinforcement learning is to directly define the reward function as a mapping from environmental interactions to real numbers. However, state machine-based approaches modify this formalism by replacing the reward function with a state machine that generates a richer reward signal. With the introduction of this alternative representation for reward functions, it has become natural to express tasks using formal languages. Consequently, we define and describe the necessary concepts and structures from formal language and automata theory. Finally, we conclude the chapter by discussing the *reward machine* formulation, which is currently the state-of-the-art approach in the field of state machine-based reward modelling.

The focus of Chapter 3 is to provide an overview of the field of non-Markovian reward modelling before formally characterising the limitations of current approaches. We begin by showing that non-Markovian reward models fall into a spectrum based on the extent to which they rely on historical information. We formalise this notion by introducing the *Hierarchy of Non-Markovian Reward*. This structure enables us to express the set of reward models solvable by current techniques in a principled manner. Specifically, we prove that current state machine-based approaches for non-Markovian reward modelling are only able to solve the weakest class of models in the hierarchy. Finally, we conclude the chapter by proving that the limitations associated with current approaches are a direct consequence of the automaton variants chosen for reward modelling.

In Chapter 4, we introduce a novel abstract machine designed to overcome the limitations of current state machine-based approaches. In addition, we formally characterise several important properties of the relationship between our method

and existing approaches.[4] We begin this chapter by defining *Counting Reward Automata* as a novel abstract machine variant capable of modelling any reward function expressible as a decidable formal language. We demonstrate that our formulation extends the set of solvable tasks from the least to the most general class of reward models[5] compatible with formal languages. This is followed by a thorough analysis in which we compare our approach to the current state-of-the-art techniques in the field of state machine-based reward modelling. We conclude the section by proving several results relating to automaton complexity and sample efficiency.

Chapter 5 introduces our first additional approach to increase sample efficiency when using counting reward automata, which is based on hierarchical reinforcement learning. We introduce *Hierarchical Reinforcement Learning for Counting Reward Automata* as an approach to increase sample efficiency. This approach exploits the structure of automata to decompose tasks into subtasks. These subproblems are solved in parallel using techniques from the field of hierarchical reinforcement learning, namely options. With a definition of the learning algorithm in place, we conduct several experiments, evaluating the approach in both tubular and continuous domains, which require function approximation.

In Chapter 6, we introduce our second additional approach to increase sample efficiency when using counting reward automata, which incorporates potential-based reward shaping into the formulation. We propose a novel approach to potential-based reward shaping known as *Reward Shaping for Counting Reward Automata*. This approach augments the original reward model with a shaping reward function derived from the automaton's structure. Several experiments are conducted to evaluate our approach. Specifically, we perform a number of experiments in both tabular and function approximation domains, considering both sparse and dense reward models.

In Chapter 7, we provide a comprehensive performance evaluation including all of the approaches presented in this research. Through critical analysis and comparison, we aim to discern the strengths, weaknesses and overall effectiveness of each approach presented in this work. Our evaluation covers various dimensions, including tabular domains, function approximation and, finally, both sparse and dense reward models. Furthermore, we contextualise our results within the broader landscape of existing literature and methodologies by rigorously evaluating our methods against the latest advancements in the field..

[4] This chapter is based on work that has been published in Bester *et al.* [2024] at the *Workshop on Neuro-Symbolic Learning and Reasoning in the Era of Large Language Models* at AAAI.

[5] As defined by the *Hierarchy of Non-Markovian Reward*, defined in Section 3.2.

Chapter 2

Background

2.1 Introduction

In this chapter, we provide a brief overview of the main concepts and formulations used throughout this research.

As we are interested in solving tasks in the context of reinforcement learning, we provide an introduction to the topic in Section 2.2. We begin in Section 2.2.1 by discussing *Markov Decision Processes (MDPs)*, the mathematical framework used to model decision-making problems in which an agent interacts with an environment. Next, in Section 2.2.2, we describe an extension to the MDP formulation known as *Non-Markovian Reward Decision Processes (NMRDPs)*. In this formulation, the reward structure is allowed to depend on past states, actions or observations which would violate the *Markov* property associated with MDPs. This is a significant extension to the MDP formulation as non-Markovian reward structures allow for more expressive and complex representations of reward, which can be useful in modelling certain real-world scenarios where past events influence future rewards. In Section 2.2.3, we describe *policies* and *value functions*. Polices and value functions are essential concepts in RL, used to make decisions as well as evaluate the quality of the resulting actions. This is followed by a discussion around parameterised representations of policies and value functions in Section 2.2.4. Section 2.2.5 describes *Hierarchical Reinforcement Learning* – an approach to decomposing complex tasks into a hierarchy of simpler tasks to improve sample efficiency. Finally, in Section 2.2.6, we discuss a technique known as *Reward Shaping*, which modifies the reward signal to facilitate more sample-efficient learning.

The conventional approach followed in reinforcement learning is to directly define the reward function as a mapping from environmental interactions to real numbers. However, state machine-based approaches to reward modelling modify this formalism by replacing the reward function with a state machine that generates a richer reward signal. This shift prompts a natural inclination towards expressing tasks using formal languages. Consequently, we define and describe the necessary concepts and structures from formal language and automata theory in Section 2.3. We begin by discussing the *Chomsky hierarchy* in Section 2.3.1 as the canonical classification system for formal lan-

guages. Next, we discuss the common state machine variants used for recognition in Section 2.3.2. This is followed by a description of *counter automata* in Section 2.3.4, a state machine variant with equivalent recognition power to that of a *Turing machine*. Finally, in Section 2.3.5, we discuss the relationship between *temporal logic, automata* and *product MDPs*.

Finally, we conclude the chapter in Section 2.4 by discussing the *Reward Machine* formulation, which is currently the state-of-the-art approach in the field of state machine-based reward modelling. The reward machine formulation aims to model complex reward structures that depend on sequences of events or states. Unlike traditional reward functions, which map individual state-action pairs to immediate rewards, reward machines encode reward signals as state machines that emit rewards based on sequences of observations and actions. The formulation is discussed in detail in Section 2.4.1. Finally, in Section 2.4.2, we describe existing approaches for reward machine specification.

2.2 Reinforcement Learning

In this section, we provide a brief introduction to *Reinforcement Learning (RL)*. Reinforcement learning is focused on finding control strategies that allow agents to maximise some notion of cumulative reward. These strategies are learned through trial and error with an environment. In this section, we discuss the theory central to reinforcement learning as well as several solution strategies.

2.2.1 Markov Decision Processes

The most common formulation used to model reinforcement learning problems is the *Markov Decision Process (MDP)* [Bellman 1957]. An MDP $\mathcal{M} = \langle S, A, P, \gamma, R \rangle$ is a sequential decision-making model in which S is a countable state space, A is a countable action space, $R : S \times A \times S \mapsto [r_{min}, r_{max}]$ is a bounded reward function, $P(s'|s,a) \doteq \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$ specifies the action-conditioned transition probabilities between states and finally, $\gamma \in [0,1]$ is the discount factor. The states in S are truth assignments to a set $\mathcal{P}$ of primitive propositions. This set of propositions is used to capture the high-level events taking place in an environment when the agent occupies a given state. It follows that if ϕ is a propositional formula and s a state, we can test if the condition specified by ϕ is satisfied in the state s, which is denoted by $s \models \phi$ [Brafman *et al.* 2018]. A function known as the labelling function $\mathcal{L}$ is used to perform the mapping of states onto truth assignments $\mathcal{L} : S \mapsto 2^{\mathcal{P}}$.

2.2.2 Non-Markovian Reward Decision Processes

Bacchus *et al.* [1996] introduced a generalisation of the MDP model known as a *Non-Markovian Reward Decision Process (NMRDP)*, a subset of the more general class of *Non-Markovian Decision Processes (NMDPs)* [Lenaers and van Otterlo 2022]. In contrast to MDPs, the reward models used in NMRDPs are specified as real-valued functions over

finite state-action sequences known as *traces*. A trace τ captures the history of state-action pairs experienced by the agent up until a given time step, $\tau = \langle s_0, a_0, ..., s_t, a_t \rangle$, where t is the last timestep included in the trace [Chatterji *et al.* 2021]. We use the notation $(S \times A)^*$ to denote the set of all traces. Formally, an NMRDP is defined as a tuple $\mathcal{M} = \langle S, A, P, \gamma, \bar{R} \rangle$ where S, A, P and γ are defined as in an MDP, and the history-dependent reward function is defined as $\bar{R} : (S \times A)^* \mapsto [r_{min}, r_{max}]$ [Camacho *et al.* 2017].

As the reward functions utilised in NMRDPs depend on entire traces, explicitly defining a non-Markovian reward function is a non-trivial task. Typically, we would like to reward behaviours corresponding to various patterns in traces, making reward definition difficult. Fortunately, temporal logics provide a convenient language for reward specification using a set of pairs $\{(\varphi_i, r_i)_{i=1}^m\}$. For a partial trace $\tau = \langle s_0, a_0, ..., s_n, a_n \rangle$, a reward is assigned based on the formulas φ_i satisfied by τ. Formally, the reward assigned to the trace τ is given by:

$$\bar{R}(\tau) = \sum_{i \in A} r_i$$

where $A = \{i : \tau \models \varphi_i \wedge 1 \leq i \leq m\}$ [Brafman *et al.* 2018].

2.2.3 Policies and Value Functions

Policies

Agents in reinforcement learning are usually assumed to be very simple, consisting of only an action selection strategy known as a *policy*. The policy determines how the agent behaves based on its perceived environmental state. Guiding actions based on perceived representations of the environment, a policy π directs the response to be taken in those states. Both deterministic and stochastic policies exist. Deterministic policies produce a single action for given a state, denoted as $\pi(s_t) = a_t$. Given a state representation, stochastic policies yield a probability distribution over actions denoted as $\pi(a_t|s_t)$, where $\pi(s_t, a_t)$ signifies the likelihood of the agent selecting action a in state s at time t [Van Hasselt 2011].

Value Functions

The value of a state or action plays an important role in reinforcement learning. In most cases, this idea of value is quantified in terms of expected cumulative reward. In this section, we formalise the notion of value with the introduction of precise definitions for the value of a state, action and policy.

For a given MDP and policy π, it is possible to determine the *state-value* $V^\pi(s)$ of following this policy when starting in state s. This value is defined as the expected cumulative discounted reward starting from state s and following policy π thereafter. The value of a state s under policy π is denoted as $V^\pi(s)$ and is formally defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s \right] \tag{2.1}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. The function V^π is known as the *state-value function for policy* π.

In reinforcement learning, we are often more interested in maximising as opposed to computing the value of a policy. The optimal value of a state is the maximal return that can be obtained with any policy starting from that state. The optimal value is denoted by V^* and is defined as follows:

$$V^*(s) = \max_\pi V^\pi(s)$$

It follows that the objective is to find the optimal policy π^* that maximises the value for each state. By definition:

$$V^{\pi^*} = V^*$$

The value defined in Equation 2.1 can be defined recursively as follows:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\big[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s\big] \\ &= \mathbb{E}_\pi\big[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots) \mid s_t = s\big] \\ &= \mathbb{E}_\pi\big[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\big] \\ &= \sum_a \pi(s,a) \sum_{s'} P(s'|s,a)\big(R(s,a) + \gamma V^\pi(s')\big) \end{aligned}$$

This specifies a linear system of $|S|$ equations, which can be solved to find the value of each state for finite state and action spaces. Similarly, the optimal value function can be described with a recursive definition:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)\big(R(s,a) + \gamma V^*(s')\big) \tag{2.2}$$

Unfortunately, this system of equations is difficult to solve analytically due to the non-linearity introduced by the max operator. Equation 2.2 was initially introduced in Bellman [1954] and is known as the *Bellman optimality equation*.

Similarly to state values, it is possible to describe the value of taking a specific action in a given state. An *action-value* is defined as the expected cumulative reward when performing an action a in state s and following a policy π thereafter, which is denoted by $Q^\pi(s,a)$ and formally defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a\right]$$

The function Q^π is known as the *action-value function for policy* π. The action-value function can also be defined recursively:

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a)\big(R(s) + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')\big)$$

The optimal action-value function Q^* can be defined recursively:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\big(R(s) + \gamma \max_{a'} Q^*(s',a')\big)$$

As there are more action values ($|S \times A|$) than state values ($|S|$), more memory is required to store action-value functions than state-value functions. However, action values have an important advantage over state values. That is, the optimal policy can be determined by acting greedily with respect to the optimal action-value function. In contrast, when only state values are known, Equation 2.2 must be solved for each state to find the optimal action. This is especially important for model-free algorithms in which Equation 2.2 cannot be solved as P and R are unknown.

2.2.4 Function Approximation

For many problems in which we would like to apply reinforcement learning, the state space is often extremely large or infinite [Yarats *et al.* 2021]. For example, consider an agent learning to play a video game from images. The state space of this problem is vast, with the number of possible images greatly surpassing the number of atoms in the universe. As discussed in Sutton and Barto [2018], finding an optimal policy or value function in such a setting is intractable, even in the limit of infinite time and data. Consequently, when working in such a domain, we shift our focus from finding optimal solutions towards computing good approximations using limited computational resources.

In many tasks with large state spaces, almost every state encountered by an agent will never have been seen before. This illustrates one of the two main problems with tabular representations of value functions. That is, not only is it often impossible to store these data structures in memory, but the amount of time and data required to fill these representations accurately makes the approach computationally intractable. We rely on generalisation to act as a solution to this problem. Such an approach allows the agent to use its experience with a limited subset of the state space to produce a good approximation over a much larger subset. The concept of generalisation has been extensively studied in several fields other than reinforcement learning [Ferrari and Stengel 2005]. This has resulted in several approaches combining existing methods with reinforcement learning [Sutton *et al.* 1999a]. In this context, the required generalisation type is often referred to as *function approximation*. These approaches use samples from a desired function (such as the value function) to construct an approximate representation of the entire function. Function approximation belongs to the broader field of *supervised learning*, which is a major area of machine learning [Clarke *et al.* 2009]. As a result, several algorithms have been studied within these fields that may be used in the role of a function approximator within a reinforcement learning algorithm. In practice, however, certain formulations perform significantly better than others.

When applying function approximation in reinforcement learning, the value function is represented as a parameterised function with a weight vector $\boldsymbol{w} \in \mathbb{R}^d$ instead of a table. The approximate value of state s given weight vector $\boldsymbol{w}$ is written as $\hat{V}(s;\boldsymbol{w}) \approx V^\pi(s)$.

The function $\hat{V}$ may be a linear function in features of the state, with $\boldsymbol{w}$ representing the vector of feature weights. In the more general case, $\hat{V}$ may be represented as an *artificial neural network*, with $\boldsymbol{w}$ as the vector of connection weights between the neurons in each layer [Haykin 1994]. Typically, the dimensionality of the weight vector is much smaller than the cardinality of the state space ($d \ll |S|$). A consequence of using such a parameterisation is that updating the value of a single state will affect the values of many other states. This generalisation has the potential to make learning more powerful. However, it can also result in a training process that is difficult to manage and understand.

2.2.5 Hierarchical Reinforcement Learning

As discussed in Barto and Mahadevan [2003], one of the main factors inhibiting reinforcement learning from being applied to temporally extended tasks is the *curse of dimensionality*. That is the exponential growth in the number of parameters which need to be learned when using any compact encoding of the system state. Several methods belonging to the field of *Hierarchical Reinforcement Learning (HRL)* have been proposed to combat this issue by introducing temporal abstraction. These principled approaches use temporally extended activities, each of which follows its own policy until termination. Both hierarchical control architectures as well as learning algorithms have been shown to follow naturally from these ideas [Botvinick *et al.* 2009].

The theory of *Semi-Markov Decision Processes (SMDPs)* provides a formal basis for several hierarchical reinforcement learning algorithms. These decision processes are a generalisation of MDPs in which the amount of time between successive decisions is modelled as a random variable [Puterman 2014; Mahadevan *et al.* 1997]. When faced with such a decision problem, often the idea of what is known as a *macro* is introduced. Macros are temporal abstractions which allow sequences of actions to be invoked as if they were primitive operators or actions. These abstractions form the basis of most hierarchical specifications as macros can, by definition, make use of other macros in their definitions. In general, hierarchical approaches generalise the idea of a macro to closed-loop policies. These policies are often referred to as *temporally-extended actions*, *options* [Sutton *et al.* 1999b], *skills* [Thrun and Schwartz 1994] or *behaviours* [Brooks 1986].

Sutton *et al.* [1999b] formalise the notion of a temporally extended action resulting in what is known as an *option*. An option consists of three components: a policy $\pi_o : S \times A \mapsto [0, 1]$, an initiation set $I_o \subseteq S$ specifying the states in which the option can be initiated and finally, a termination condition $\beta_o : S \mapsto [0, 1]$ specifying the probability of the option terminating in a particular state. As a result, an option is formally defined as a tuple (I_o, π_o, β_o). Once an option is executed, actions are selected according to π_o until the option stochastically terminates based on β_o. For example, if the agent is executing an option in state s, the probability of the following action being a is given by $\pi_o(a|s)$. After performing this action, the environment transitions into state s', and the option terminates with probability $\beta_o(s')$. In the event that the option terminates, the agent is able to select another option before continuing the process. Each action available in the environment can be seen as a special case of the options framework. These options are

available in the environment whenever the action is available. When executed, these options last precisely one step and select a specific action in all states. Such options are known as *primitive options*. Without the inclusion of these options in the available option set, there is no guarantee that the method will converge to an optimal policy. This follows from the fact that the selected set of options restricts policy space which may result in optimal policies being pruned. In such a case, the resulting policy is said to be *hierarchically optimal* as opposed to globally optimal [Brunskill and Li 2014].

2.2.6 Reward Shaping

One of the fundamental challenges in reinforcement learning is the *credit assignment problem*. To be specific, this problem refers to the difficulty of measuring the influence of an agent's actions on future rewards [Lansdell *et al.* 2019]. In most cases, the information contained within the reward signal is insufficient to specify the long-term consequences of individual actions. This makes the learning problem more difficult for reinforcement learning algorithms, decreasing sample efficiency and slowing convergence.

Reward shaping is a convenient approach for dealing with the credit assignment problem. This technique simplifies the relationship between individual actions and their long-term consequences through the introduction of an alternative reward function [Grzes 2017]. Notably, these techniques are often constrained, requiring that the policy learned with reward shaping is equivalent to the policy that would have been learned using the original reward function [Badnava *et al.* 2023].

In order to preserve the set of optimal policies, it has been shown that the alternative reward function must be a potential function [Ng *et al.* 1999]. That is, for a reward transformation $R'(s, a, s') = R(s, a, s') + F(s, a, s')$, where R is the original reward function and F is the shaping reward function, F must be chosen from a restricted class of potential-based reward shaping functions. Potential-based reward-shaping functions have the form:

$$F(s, a, s') = \gamma\phi(s') - \phi(s)$$

where $\phi : S \mapsto \mathbb{R}$ is an arbitrary real-valued function and γ is the discount factor. Reward shaping through the use of potential functions is known as *potential-based reward shaping*. This approach has the desirable property of preserving optimal and near-optimal policies with respect to the original MDP.

2.3 Formal Language and Automata Theory

While the conventional approach followed in reinforcement learning is to directly define the reward function as a mapping from environmental interactions to real numbers, Icarte *et al.* [2022] modifies the formalism by replacing the reward function with a finite automaton that is used to generate a richer reward signal.

The topic of *automata theory* is contained within the field of theoretical computer science and is focused on the study of formulations known as *abstract machines*. An

automaton defines an abstract model of computation, able to perform a predefined set of operations in response to a set of inputs [Hopcroft *et al.* 2001]. An automaton which has been restricted to contain a finite number of machine states defines a formulation known as a *finite state machine (FSM)*. The FSM formulation is designed to model simplified computations in which the abstract machines transition between states in response to inputs.

Automata theory is closely related to the field of *formal language theory*. This field focuses on the study of sets of strings known as *formal languages*. Each *string* belonging to a formal language consists of a sequence of symbols taken from a finite set known as the language's *alphabet*. This sequence is well-formed according to a specific set of rules known as a *formal grammar* [Moll *et al.* 2012]. The grammar does not specify the meaning of strings belonging to the generated language; it only imposes constraints on their structure. Automata play an essential role in formal language theory as finite representations of possibly infinite formal languages.

Formal grammars consist of two components: a set of rules for rewriting strings known as *production rules* as well as a *start-symbol* from which rewriting begins [Meduna 2014]. The function of a grammar is twofold. Firstly, it is used to generate the strings of a language, acting as a language generator. Secondly, the rules within the grammar are used to form the basis of a *recogniser* - an automaton used to produce a binary output indicating whether a given string belongs to the language. Each production rule is defined by two components: the left-hand side and the right-hand side. The first component specifies the particular string to be replaced by the second component, which is the replacement string. Production rules may be applied to any string that contains the first component of the rule to produce a string in which an occurrence of the first component has been replaced with the second component. Grammars distinguish between two classes of symbols: *terminal* and *non-terminal* symbols. All strings that can be generated using the production rules associated with a grammar form the language generated by that grammar. These strings are required to contain only terminal symbols. The strings in this set are created through the repeated application of production rules, beginning from a string consisting of a single start symbol.

2.3.1 Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars [Chomsky 1956]. Four different classes of formal grammars exist within the Chomsky hierarchy. Each class not only generates increasingly complex languages but can also completely generate the languages of all inferior classes. Formal grammars are classified into the following four classes:

Type-3 Grammars

Regular languages are generated by type-3 grammars. These grammars impose the following constraints on the structure of production rules. For any production rule, the first component is restricted to contain one non-terminal symbol. The second compo-

nent is constrained to contain one terminal symbol, which a single non-terminal symbol may optionally follow. Finite state machines recognise these languages.

Type-2 Grammars

Context-free languages are generated by type-2 grammars. These grammars impose the following constraints on the structure of production rules. Any production rule associated with grammars of this type must have the structure $A \rightarrow \alpha$. In these rules, the symbol A is used to denote a non-terminal symbol. The symbol α is used to denote any string of non-terminal and terminal symbols. *Non-deterministic pushdown automata* recognise these languages.

Type-1 Grammars

Context-sensitive languages are generated by type-1 grammars. These grammars impose the following constraints on the structure of production rules. Any production rule associated with grammars of this type must have the structure $\alpha A \beta \rightarrow \alpha \gamma \beta$. The symbol A is used to denote a non-terminal symbol. The symbols α, β and γ are used to denote any string of non-terminal and terminal symbols. The string γ is required to be nonempty. *Linear bounded automata* recognise these languages.

Type-0 Grammars

Recursively enumerable or *Turing-recognisable* languages are generated by type-0 grammars. Any formal grammar is a type-0 grammar. *Turing machines* recognise these languages.

2.3.2 Automata

The classes of automata are related to the set of formal languages that the machines recognise. This section defines the four main classes of automata and the languages they recognise. A summary of the classes automata and the properties of the languages they recognise is given in Table 2.1.

Finite State Machines

Finite state machines (FSMs) form the simplest computational model out of the set of automata that is able to recognise the class of regular languages. Both deterministic and non-deterministic variants of finite state machines exist. The addition of non-determinism does not increase the set of languages recognisable by the machine. There are two classes of finite state machines depending on whether or not the automata produce output. We begin by defining the simpler case, FSM which do not produce output. Figure 2.1 illustrates an example of such an FSM.

Formally, a finite state machine is defined as a tuple $\langle Q, \Sigma, q_0, F, \delta \rangle$ where:

- Q denotes the finite set of machine states.

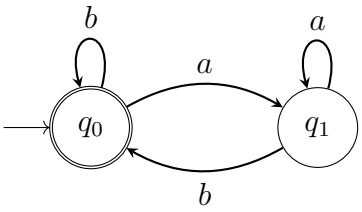


Figure 2.1: Finite state machine illustration. The machine begins in the initial state q_0. At each timestep, the machine receives a symbol $\sigma \in \Sigma$ as input, which may be a or b in this case. The machine then transitions states in response to the input as defined by δ. The machine continues reading input symbols until the input string is exhausted. If the machine ends in an accepting state (q_0 is accepting as illustrated by the double circle), then the given string is said to be a member of the language recognised by the machine.

- Σ denotes the input alphabet (a finite set of symbols).
- $q_0 \in Q$ denotes the state the machine begins in.
- $F \subseteq Q$ denotes the set of accepting states for the machine.
- $\delta : Q \times \Sigma \mapsto Q$ denotes the transition function of the machine.

Next, we provide definitions for the class of FSM which produce output. The corresponding set of automata is comprised of two automaton variants, namely, Mealy and Moore machines which differ in the structure of their output functions. More specifically, after reading an input symbol, Moore machines produce an output based on the active machine state whereas the output of a Mealy machine is dependent on the current machine state as well as the input symbol. Examples of Moore and Mealy machines are illustrated in Figures 2.2 and 2.3 respectively.

A Moore machine is formally defined as a tuple $\langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$ where:

- Q denotes the finite set of machine states.
- Σ denotes the input alphabet (a finite set of symbols).
- Δ denotes the output alphabet (a finite set of symbols).
- $\delta : Q \times \Sigma \mapsto Q$ denotes the transition function of the machine.
- $\lambda : Q \mapsto \Delta$ denotes the output function of the machine.
- $q_0 \in Q$ denotes the state the machine begins in.

A Mealy machine is formally defined as a tuple $\langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$ where:

- Q denotes the finite set of machine states.
- Σ denotes the input alphabet (a finite set of symbols).
- Δ denotes the output alphabet (a finite set of symbols).
- $\delta : Q \times \Sigma \mapsto Q$ denotes the transition function of the machine.

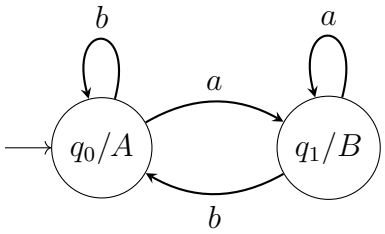


Figure 2.2: Moore machine illustration. The machine begins in the initial state q_0. At each timestep, the machine receives a symbol $\sigma \in \Sigma$ as input, which may be a or b in this case. The machine then transitions states in response to the input as defined by δ as well as produces an output symbol as defined by λ, which may be A or B in this example. Each machine state is labelled using two components, firstly the name of the machine state before the forward slash and then the output symbol associated with the state. The machine continues reading input symbols and producing outputs until the input string is exhausted.

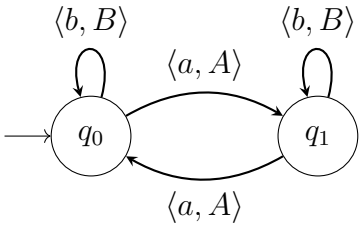


Figure 2.3: Mealy machine illustration. The machine begins in the initial state q_0. At each timestep, the machine receives a symbol $\sigma \in \Sigma$ as input, which may be a or b in this case. The machine then transitions states in response to the input as defined by δ as well as produces an output symbol as defined by λ, which may be A or B in this example. Each transition is labelled with a tuple in which the first component is the input symbol associated with the transition and the second component is the output symbol produced by the automaton. The machine continues reading input symbols and producing outputs until the input string is exhausted.

- $\lambda : Q \times \Sigma \mapsto \Delta$ denotes the output function of the machine.
- $q_0 \in Q$ denotes the state the machine begins in.

Pushdown Automata

A pushdown automaton (PDA) is a type of automaton that incorporates an infinite stack as memory. As a result of the additional memory accessible to this machine, pushdown automata are more capable than finite-state machines. Both deterministic and non-deterministic PDA variants exist. Non-determinism is required by the formulation to recognise the set of context-free languages. An example PDA is illustrated in Figure 2.4.

A PDA is defined as a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, z_0, F \rangle$ where:

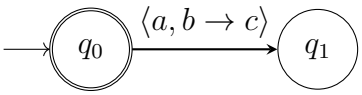


Figure 2.4: Pushdown automaton illustration. The machine functions similarly to an FSM. The machines differ in their transition behaviour. Each PDA transition is based on the current machine state, the top stack symbol and the input symbol. When transitioning, machine states are changed in addition to the stack contents being modified. This is illustrated by the tuples associated with each edge. The input symbol is denoted by a. The symbol required on the top of the stack is given as b. The set of symbols c is pushed to the stack after the transition.

- Q denotes the finite set of machine states.
- Σ denotes the finite set of input symbols.
- Γ denotes the finite set of stack symbols.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \mapsto Q \times \Gamma^*$ denotes the transition function of the machine.
- $q_0 \in Q$ denotes the state the machine begins in.
- $z_0 \in \Gamma$ denotes the symbol initially on the stack.
- $F \subseteq Q$ denotes accepting states.

Turing Machines and Linear Bounded Automata

Turing machines are abstract machines that are mathematical models of computation. Turing machines function by manipulating symbols on an infinite tape. These symbols are modified based on some set of predetermined operations. Although the model is simple, it can be used to implement any algorithm that can be executed on a computer. Turing machines are able to recognise the set of Turing-recognisable formal languages. Figure 2.5 illustrates an example Turing machine.

Turing machines are formally defined as tuples $\langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where:

- Q denotes the finite set of machine states.
- Γ denotes the finite set of tape symbols.
- $b \in \Gamma$ denotes the blank symbol.
- Σ denotes the finite set of input symbols.
- $q_0 \in Q$ denotes the state the machine begins in.
- $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ denotes the transition function. This is a partial function. Left and right movements on the tape are denoted using the symbols L and R. The machine halts if δ is not defined.
- $F \subseteq Q$ denotes the set of accepting states for the machine.

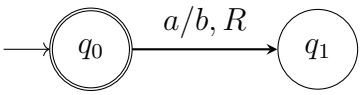


Figure 2.5: Turing machine illustration. The machine functions similarly to an FSM. The machines differ in their transition behaviour. In state q_0 with a as the tape symbol, the machine will write the symbol b to the tape. After writing b to the tape, the machine will move one cell to the right on the tape which is indicated by R.

Linear bounded automata (LBA) are defined as restricted Turing machines. An LBA is equivalent to a non-deterministic Turing machine that satisfies the following three conditions:

1. The input alphabet includes two special symbols which function as end markers on the tape.
2. Other symbols may not be written over the end markers.
3. Transitions must remain within the bounds on the tape defined by the end markers.

Regardless of the infinite tape, the machine's computation is restricted. The machine can only access the section of the tape containing the input string and the two cells on either side of the input string containing the end markers. As a result of these constraints, linear bounded automata can only recognise the set of context-sensitive languages.

Grammar	**Languages**	**Automaton**	**Production Rules Constraints**
Type-3	Regular	Finite state machine	$A \to a$, $A \to aB$
Type-2	Context-free	Non-deterministic pushdown automaton	$\alpha A \beta \to \alpha \gamma \beta$
Type-1	Context-sensitive	Linear bounded automaton	$A \to \alpha$
Type-0	Recursively enumerable	Turing machine	$\gamma \to \alpha$ (γ is non-empty)

Table 2.1: The four types of grammars in Chomsky's hierarchy along with the production rule structure, languages generated and the type of automaton that recognises the corresponding class of languages.

2.3.3 Decidability in Formal Languages

A fundamental concept in the theory of computation is that of decidability which deals with the question of whether or not a given problem can be solved algorithmically. That is, whether or not there exists a finite procedure, or well-defined algorithm, that is able

to provide a yes or no answer for every instance of a problem within a finite amount of time.

In the context of formal languages, this translates to determining whether there is an algorithm that can decide if any given string belongs to a particular language. A formal language is said to be decidable (or recursive) if there exists a Turing machine that can determine whether any given string belongs to the language or not, always halting with a yes or no answer. The set of all formal languages is partitioned into decidable (recursive), semi-decidable (recursively enumerable) and non-recursively enumerable languages [Linz and Rodger 2022].

Decidable (Recursive) Languages

A language L is said to be decidable if there exists a Turing machine that halts and correctly decides whether any given string ω belongs to L or not. In other words, the machine accepts ω if $\omega \in L$ and rejects if $\omega \notin L$, always halting in a finite amount of time. These languages are also known as recursive languages because the membership decision process can be expressed as a total recursive function.

Semi-Decidable (Recursively Enumerable) Languages

A language L is said to be semi-decidable if there exists a Turing machine that halts and accepts any string ω that belongs to L, but it may either reject or run indefinitely if ω does not belong to L. Every decidable language is also recursively enumerable, but the converse is not true. A recursively enumerable language may have an algorithm to enumerate its members but not to definitively decide non-membership.

Non-Recursively Enumerable Languages

A language L is said to be non-recursively enumerable if there does not exist a Turing machine that can enumerate all the strings in L, which implies that there is no Turing machine that can accept exactly the strings in L and possibly run indefinitely on strings not in L. Similarly, there is no Turing machine that can enumerate all the strings not in L. These languages cannot be described by Turing machines and represent problems which are inherently unsolvable by any algorithmic process.

2.3.4 Counter Machines

Counter automata, also known as *counter machines*, are a finite state machine variant which have been extended to include a finite set of integer-valued variables known as counters. These automata update the values of their counters while processing strings, influencing how the abstract machines transition between states [Valiant and Paterson 1975]. Notably, a key result from theoretical computer science established that a 2-counter machine with unbounded computation time is Turing-complete [Fischer 1966]. As a result, such a counter machine is able to recognise any recursively enumerable formal language as the abstract machine can simulate any Turing machine.

Formulation

In this section, we introduce the *k-counter machine* formulation as defined in Merrill [2020]. For $m \in \mathbb{Z}$, let $\pm m$ denote the function $\lambda x.x \pm m$ as defined in *lambda calculus* [Barendregt 1984]. A k-counter machine is formally defined as a tuple $\langle \Sigma, Q, q_0, u, \delta, F \rangle$ where

- Σ denotes the input alphabet (a finite set of symbols).
- Q denotes a finite set of machine states.
- $q_0 \in Q$ denotes the state the machine begins in.
- u denotes the function used to update counters:

$$u : Q \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto \{+m : m \in \mathbb{Z}\}^k$$

- δ denotes the transition function for machine states:

$$\delta : Q \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto Q$$

- $F \subset Q \times \{0,1\}^k$ denotes the set of accepting automaton configurations.

where k is the number of counter variables and ε is used to denote an empty string, allowing the machine to transition without reading an input symbol.

Input strings are processed by the machine one input symbol at a time. Upon reading an input symbol, the machine transitions between states and updates the values of its counters. The exact updates are based on the current machine state, counter values and input symbol. More precisely, we will denote a counter machine configuration as a tuple $\langle q, \boldsymbol{c} \rangle \in Q \times \mathbb{Z}^k$. After the input symbol $x \in \Sigma$ has been read by the machine, the counter machine configuration is updated as follows:

$$\langle q, \boldsymbol{c} \rangle \underset{x}{\rightarrow} \langle \delta(x, q, Z(\boldsymbol{c})), u(x, q, Z(\boldsymbol{c})) \rangle$$

where Z is used to denote the *broadcasted zero-test* function:

$$Z(\boldsymbol{v})_i = \begin{cases} 0 & \text{if } v_i = 0 \\ 1 & \text{otherwise.} \end{cases}$$

for vector $\boldsymbol{v} \in \mathbb{Z}^k$.

Example

In this section, we provide an example illustrating how counter machines operate. We describe how a counter machine can be used to recognise strings belonging to the context-free language $L = \{A^N B^N : N \in \mathbb{N}\}$. The counter machine is illustrated in Figure 2.6.

The machine is illustrated as a directed graph. Each vertex in the graph is a machine

state, and each edge in the graph is a transition. Associated with each edge is a label $\langle \sigma, \boldsymbol{\omega}, \boldsymbol{\mu} \rangle$ where $\sigma \in \Sigma$ is an input symbol, $\boldsymbol{\omega} \in \{0, 1\}^k$ is the *zero-tested* counter state and $\boldsymbol{\mu} \in \{+m : m \in \mathbb{Z}\}^k$ is the counter modifier.

We use an example string $AABB$ to demonstrate how the machine functions. The machine begins in the initial state u_0 with a counter value of 0. The input string is processed one symbol at a time. Firstly, the symbol A is read. At this point, the machine's counter state is $[0]$ as the counter value is zero. Upon reading this symbol, the machine transitions, remaining in state u_0 and incrementing its counter value by 1. Next, the machine reads the symbol A. At this point, the counter state is $[1]$ as the counter value is not equal to zero. This causes the machine to transition, remaining in state u_0 and incrementing its counter value to 2. The machine then reads the input symbol B. The machine transitions to state u_1 and decrements the counter value by 1. Finally, the machine reads the input symbol B. The machine transitions and decrements the counter value to zero. The input string is accepted as the automaton is in an accepting state with a counter value of zero after the input string is exhausted.

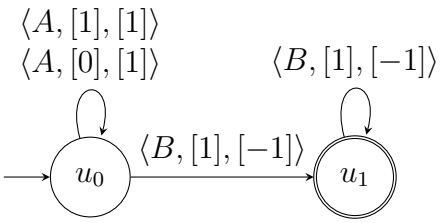


Figure 2.6: Illustration of a counter machine that recognises the language $L = \{A^N B^N : N \in \mathbb{N}\}$.

2.3.5 Temporal Logic, Automata and Product MDPs

Historically, temporal logics have seen widespread adoption in the expression of non-Markovian reward functions [Bacchus *et al.* 1996; Pnueli 1977]. These logics are used to define propositional formulae over environment traces, which specify the conditions under which the agent should be rewarded. Such temporal formulae, defined using techniques such as *Linear Temporal Logic (LTL)*, can be compiled into equivalent deterministic finite automata. Formally, an FSM for formula φ is denoted $A_\varphi = \langle Q, 2^{\mathcal{P}}, q_0, F, \delta \rangle$, where $2^{\mathcal{P}}$ is the input alphabet containing all truth assignments to the propositions in $\mathcal{P}$, Q is the set of machine states, δ is the state transition function, F is the set of accepting states and q_0 is the initial machine state [Brafman *et al.* 2018].

Solving an NMRDP in which reward specification is performed through a set of pairs $\{(\varphi_i, r_i)_{i=1}^m\}$ (discussed in section 2.2.2) can be achieved through the formulation of an extended MDP. This MDP is equivalent to the original NMRDP in the sense that a mapping exists between the states of the two models, yielding identical transition probabilities. Each formula φ_i is compiled into an equivalent automaton, and what is known as the *cross-product* between the original NMRDP and these automata is computed. The resulting decision process has a larger state space than the NMRDP; however, the Markov property is satisfied. Hence, this decision process is called the *extended MDP*.

A solution to the original model can be obtained by solving the extended model, which is convenient as it is compatible with conventional reinforcement learning algorithms [Lenaers and van Otterlo 2022].

2.4 Reward Machines

Reinforcement learning is particularly well suited for use in complex environments that are difficult to model, which prevents the use of other sequential decision-making techniques. As a result, imposing the restriction on the agent that the transition probabilities of the environment are unknown is justified. Agents cannot directly discern rewards when interacting with an environment. This requires that reward functions be designed and implemented for both simulated and real-world environments. As all environments necessitate manual development of reward functions, the assumption that the reward model must remain hidden from the agent is unnecessary and detrimental to learning.

It has previously been shown that giving the agent access to the reward function specification can be used to learn optimal policies faster. To this end, two main approaches have been developed to utilise knowledge of the reward function structure to facilitate more sample-efficient learning. In the first approach, reward functions are generated to satisfy a predefined task specification language. Such a task specification language is usually expressed through temporal logic or makes use of the concept of sub-goal sequences [Singh 1992; Toro Icarte *et al.* 2018b; Li *et al.* 2017]. The second approach employs a finite state machine variant, known as a *Reward Machine*, in order to define the reward function [Icarte *et al.* 2022]. Such a formulation is advantageous as it allows reward functions to be composed temporally in a number of different ways, including loops and conditionals. This approach is particularly promising as it is able to model reward structures that cannot be expressed by the methods described in the previous approach. The purpose of these automata is to specify rewards in a manner that exposes the internal structure of the reward function. Allowing agents to access this information has been shown to enable faster learning when compared to conventional reward specification techniques.

2.4.1 Understanding the Formulation

In this section, we examine the *OfficeWorld* environment presented in Figure 2.7 as an example. Using this environment, we demonstrate how reward machines can be used for reward specification.

As an illustrative example, we consider how a reward function for the task of collecting coffee and delivering it to the office location can be specified through the use of a reward machine. A reward machine is a finite state machine which has been augmented with an output function. Consequently, reward machines belong to a more general class of automata known as *Moore machines* [Eilenberg 1974]. The operation of a reward machine is controlled by the set of high-level events taking place in the agent's environment. More precisely, propositional symbols are used to encode outcomes of interest which the agent can detect. These propositions are contained in the set $\mathcal{P}$. The

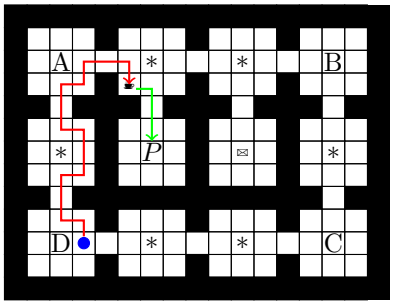


Figure 2.7: Illustration of the *OfficeWorld* presented in Icarte *et al.* [2022]. The agent, represented as a blue circle, begins in a fixed location. The agent can move in any of the four cardinal directions, and its observations are restricted to its current position in the environment. The symbols $*$ represent decorations, which are broken if the agent collides with them. Mail can be collected from the location ✉ and coffee can be made at the location ☕. Several people are located in the office located at P. Finally, there are four locations of interest denoted by A, B, C and D. The illustrated trajectory shows coffee being delivered to the office by the agent.

subset of events taking place at each time step is used as input by the machine. After receiving an input, the automaton transitions between states and outputs a reward function, which is used for reward assignment at each step. As discussed in section 2.2.1, the labelling function $\mathcal{L} : S \mapsto 2^{\mathcal{P}}$ is used to assign truth values to the propositions in $\mathcal{P}$ given the agent's state.

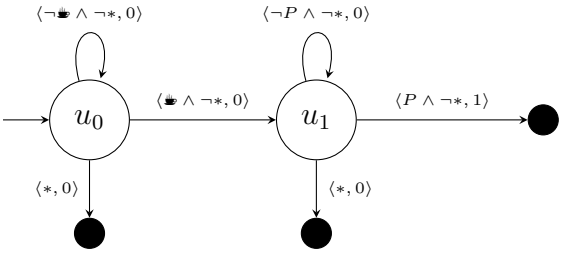


Figure 2.8: Simple reward machine used to solve the example task in the *OfficeWorld* environment.

Reward machines that specify tasks in the *OfficeWorld* are defined over the set of propositional variables $\mathcal{P} = \{☕, ✉, *, P, A, B, C, D\}$. The propositions in $\mathcal{P}$ are satisfied when the agent is at their respective locations. A reward machine defined in an environment with states S and actions A is a tuple $R_{\mathcal{P}SA} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ where U is a finite set of machine states, $u_0 \in U$ is the initial state, $F \cap U = \emptyset$ is a set of terminal states, $\delta_u : U \times 2^{\mathcal{P}} \mapsto U \cup F$ is the state-transition function and $\delta_r : U \mapsto [S \times A \times S \mapsto \mathbb{R}]$ is the state-reward function.

A reward machine for the example task is illustrated in Figure 2.8. This machine receives an input at each timestep containing the propositions that are satisfied in the environment. The states of the reward machine are illustrated as vertices in the directed graph. Associated with each edge is a tuple $\langle \varphi, c \rangle$. The first element in the tuple

φ denotes a logical formula over the set of propositions $\mathcal{P}$. The second element in the tuple c is a reward function. This reward function may be replaced by a constant to produce a special case of the formulation known as a *simple reward machine*. These tuples signify that when the current truth assignment satisfies the formula φ within the given machine state, the agent receives a reward value of c, and the machine transitions to the specified state via the directed edge.

The reward machine formulation obtains solutions using a similar approach to that discussed in Section 2.3.5 with the formation of an extended MDP. That is, the *cross-product* between the original NMRDP and the reward machine is computed to produce an MDP. As discussed, this MDP has a larger state space than that of the original NMRDP; however, the Markov property is satisfied. In the reward machine formulation, this MDP is known as the *cross-product MDP*. The *cross-product MDP* associated with a reward machine $\mathcal{R}_{\mathcal{PSA}} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ is formally defined as a tuple $\mathcal{M} = \langle S', A', P', \gamma', R' \rangle$ where

$$
\begin{aligned}
S' &= S \times U \cup F \\
A' &= A \\
\gamma' &= \gamma \\
P'(\langle s', u' \rangle | \langle s, u \rangle, a) &= \begin{cases} P(s'|s,a) & \text{if } u \in F \text{ and } u' = u \\ P(s'|s,a) & \text{if } u \in U \text{ and } u' = \delta_u(u, L(s,a,s)) \\ 0 & \text{otherwise.} \end{cases} \\
R'(\langle s, u \rangle, a, \langle s', u' \rangle) &= \begin{cases} \delta_r(u)(s,a,s) & \text{if } u \notin F \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

where S, A, P and γ are defined as in the NMRDP. As the cross-product MDP is fundamentally an MDP, a solution to the original model can be obtained by solving this extended model, which is convenient as it is compatible with conventional reinforcement learning algorithms [Lenaers and van Otterlo 2022].

2.4.2 Reward Machine Specification

Currently, two main approaches for reward machine specification exist [Camacho *et al.* 2019]. In the first approach, a reward machine is defined for the task directly. This requires that a state machine corresponding to the task definition be defined, specifying both the subtask dependencies and dynamics. With the definition of such an automaton in place, the state-reward function must be defined for each edge in the resulting directed graph. While this approach is able to produce the desired results, it is tedious and error-prone when specifying complex temporally extended behaviours consisting of multiple interrelated stages. Alternatively, the second approach allows reward machines to be specified using temporal logics such as LTL. Regardless of the specification language, the same process is followed for reward machine generation. Firstly, a deterministic finite automaton is constructed, which recognises the language of environment traces that satisfy the original goal specification. This automaton is then used to construct a reward machine through the addition of an appropriate output function.

2.5 Conclusion

In this chapter, we provided a brief overview of the main concepts and formulations used throughout this research. As we focus on addressing challenges within reinforcement learning, we offer a concise overview of the subject. Traditionally, reinforcement learning involves directly defining the reward function as a mapping from interactions in the environment to real numbers. However, state machine-based approaches alter this framework by substituting the reward function with a state machine. When using this alternative representation for reward functions, it becomes intuitive to represent tasks using formal languages. Consequently, we describe the necessary concepts and structures from formal language and automata theory. Finally, we conclude the chapter by examining the reward machine formulation.

Chapter 3

Modelling Non-Markovian Reward Functions

3.1 Introduction

Reward models play an important role in decision processes, assigning rewards to agents based on their interactions with an environment. These models function as mappings from trajectories onto scalar reward values. We review the field of non-Markovian reward modelling before formally characterising the limitations of current approaches. Finally, we propose counter automata as a novel solution to these limitations.

The primary contributions outlined in this chapter are given below:

- A novel containment hierarchy is defined for non-Markovian reward models. This is a significant contribution as it allows reward functions to be partitioned principally based on complexity.
- We use this hierarchy to express the limitations of current state machine-based approaches to reward modelling in a precise manner.
- We formalise the role of automata in reward modelling, describing how state machine-based approaches rely on the idea of membership to perform reward assignments.
- We prove that reward machines are only compatible with the weakest class of reward models in the *Hierarchy of Non-Markovian Reward*.
- We propose counter automata as a solution to the limitations of current approaches before strongly motivating why this automaton variant is well-suited for the application.

The structure of this chapter is as follows. We begin by showing that a spectrum of non-Markovian reward models exists based on the extent to which they rely on historical information. This notion is formalised through the introduction of the *Hierarchy of Non-Markovian Reward* in Section 3.2. Next, in Section 3.3, we prove that current state machine-based approaches for reward modelling are only compatible with the

weakest class of reward models in the hierarchy. In Section 3.4, we propose counter machines as a solution to the limitations of current state machine-based approaches for reward modelling. Finally, we conclude the chapter by discussing related work for non-Markovian reward modelling in Section 3.5.

3.2 The Hierarchy of Non-Markovian Reward

The conventional approach followed in reinforcement learning is to directly define the reward function as a mapping from environmental interactions to real numbers. However, state machine-based approaches to reward modelling modify this formalism by replacing the reward function with a state machine that generates a richer reward signal. The purpose of these state machines is to model complex reward structures which rely on sequences of events or states. As state machines are at the core of the reward assignment process, this set of reward models relies on the idea of membership[1] to compute rewards. Specifically, the sequences of state-action pairs from which trajectories are composed are viewed as strings belonging to formal languages[2]. These reward models define formal languages whose strings represent sequences of state-action pairs which are deemed desirable. Finally, reward values are computed based on whether or not the reward model's state machine recognises the agent's trajectory string. The design of this state machine is such that it recognises exactly the set of trajectory strings which satisfy the given task specification.

As discussed, state machine-based reward modelling techniques rely on the idea of membership in order to perform reward assignments. As a result, the class of formal languages and, consequently, the automaton variant used for reward modelling plays an important role in the expressiveness of the model. For example, consider the set of *Regular Non-Markovian Reward Models* as defined in Lenaers and van Otterlo [2022]. These models treat reward assignment as a membership problem for regular languages. However, as the concept of counting cannot be expressed in regular languages, these reward models cannot utilise this property for reward assignment. This illustrates the relationship between the complexity of reward models and the types of formal languages required for their expression. Similarly to the Chomsky hierarchy, a containment hierarchy of reward models exists based on the class of formal languages required for their expression. Each class not only models increasingly complex reward functions but can also completely model all functions belonging to any of the inferior classes. We refer to this hierarchy as the *Hierarchy of Non-Markovian Reward*.

An illustration of the hierarchy of non-Markovian reward is provided in Figure 3.1. This figure illustrates the set of all reward functions. A subset of these functions are expressible by regular non-Markovian reward models. We refer to this subset as *type-3 non-Markovian reward models* in reference to the Chomsky hierarchy of formal gram-

[1]The membership problem for formal languages is defined as follows: given a string s and a language L, determine if $s \in L$.

[2]Formal languages defined over the alphabet $\Sigma = S \times A \times S$.

mars. Next, we consider the subset of all reward models which treat reward assignment as a membership problem for context-free languages. Similarly, we refer to this subset as *type-2 non-Markovian reward models*. Notably, this class of reward models is able to completely model all functions belonging to the inferior class of type-3 non-Markovian reward models. Finally, we define *type-1* and *type-0 non-Markovian reward models* as the subset of all reward models which treat reward assignment as a membership problem for context-sensitive and recursive languages, respectively. The set of type-0 non-Markovian reward models is an extremely general class of reward models, equal to all reward models expressible as well-defined algorithms [Rogers Jr 1987].

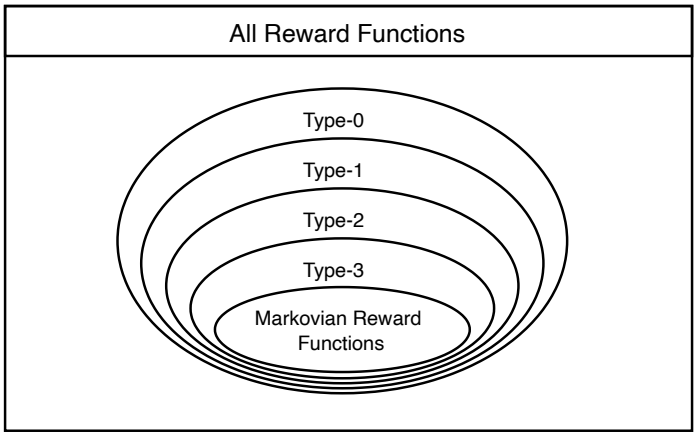


Figure 3.1: Illustration of the Hierarchy of Non-Markovian Reward.

3.3 Insufficiency of Reward Machines

In this section, we formalise the limitations of the reward machine formulation. Specifically, we focus on the ability of the framework to express arbitrary reward functions. As discussed in Section 2.4.1, the mechanism used in the reward machine formulation to map environment traces onto rewards is a *Deterministic Finite Automaton (DFA)*. To facilitate the usage of a state machine for reward assignment, the sequences of state-action pairs from which trajectories are composed are treated as strings belonging to a formal language. Specifically, these formal languages are defined over the alphabet $\Sigma = S \times A \times S$. Consequently, reward machines define an ordering on the set Σ^* with certain traces receiving more reward than others[3]. The set of strings that induce equal reward from the automaton can be defined by some formal grammar. Herein lies the key limitation of the reward machine formulation. The finite state machine (DFA) responsible for the expression of the reward function is only able to recognise regular languages. That is, languages generated by type-3 grammars in the Chomsky hierarchy. As a result, reward functions only specifiable in higher-level grammars in the Chomsky hierarchy cannot be expressed by reward machines. This result is formalised in Theo-

[3]In this context $*$ is used to represent the Kleene star [Fletcher *et al.* 1991].

rem 1. An example task specification which cannot be expressed by reward machines is as follows: *For each person in the office, collect and deliver exactly one coffee.* As this task requires counting, a property not expressible in regular languages, it cannot be expressed as a reward machine [Eilenberg 1974].

Theorem 1. *Non-Markovian reward functions that distinguish between histories via properties not expressible as regular expressions over the set $S \times A \times S$ cannot be expressed as reward machines.*

Proof. To prove the result, we show that reward machines are an instance of the Moore machine formulation which is restricted to recognising the class of regular languages.

Let $\mathcal{R}_{\mathcal{PSA}} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ be a reward machine. The reward machine $\mathcal{R}_{\mathcal{PSA}}$ can be defined as a Moore machine $\mathcal{A}_{\mathcal{M}} = \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$:

$$
\begin{aligned}
Q &= U \cup F \\
\Sigma &= 2^{\mathcal{P}} \\
\Delta &= \text{Im}(\delta_r) = \{f : S \times A \times S \mapsto [r_{min}, r_{max}] \mid \delta_r(u) = f \text{ for some } u \in U \cup F\} \\
\delta &= \delta_u : Q \times \Sigma \mapsto Q \\
\lambda &= \delta_r : Q \mapsto \Delta \\
q_0 &= u_0
\end{aligned}
$$

where $\text{Im}(f)$ is used to denote the image of the mapping f, that is, the set of all output values it may produce.

Although Moore machines cannot be used for language recognition directly, the approach defined in Riaz and Zafar [2008] can be used to construct a deterministic finite automaton $\mathcal{A}_{\mathcal{D}}$ with equivalent recognition power. The result follows from the fact that the DFA $\mathcal{A}_{\mathcal{D}}$ is only able to distinguish between histories via properties expressible as regular expressions over the set $S \times A \times S$. □

Corollary 1.1. *The set of reward models expressible by the reward machine formulation is exactly equal to the class of type-3 non-Markovian reward models.*

Proof. Reward machines are only able to express non-Markovian reward functions that treat reward assignment as a membership problem for regular languages (Theorem 1). It follows that the set of reward functions expressible by the reward machine formulation is, by definition, the class of type-3 non-Markovian reward models. □

3.4 Beyond Finite State Machines

As the reward machine formulation is based on finite state machines, the core automaton used for reward expression is limited to regular languages. Consequently, the formulation is constrained to type-3 non-Markovian reward models (see Corollary 1.1). In order to express tasks corresponding to more complex (non-regular) formal languages, more powerful automata are required. We propose counter machines as

a natural choice for this extension for several reasons. Firstly, the k-counter machine formulation is Turing-complete for $k \geq 2$, and as a result, it is able to express tasks corresponding to any decidable formal language [Fischer 1966]. Secondly, counter machines support a synchronous model of computation in which the machine is updated after each agent-environment interaction. This greatly simplifies the reward assignment process. Finally, these automata unify the stack and tape components associated with pushdown automata and Turing machines into a single set of counter variables. Not only is this representation simpler to implement in practice, but it functions seamlessly in the aforementioned synchronous model of computation, something which is not necessarily true for stack or tape representations of memory.

3.5 Related Work

3.5.1 Reward Machines

The concept of reward machines was initially proposed in Icarte *et al.* [2018]. During that period, significant research utilised LTL to reward agents in Markov Decision Processes (MDPs) [Brafman *et al.* 2018; Lacerda *et al.* 2014; Bacchus *et al.* 1996; Camacho *et al.* 2017]. A prevalent approach involved translating LTL specifications into finite state machines that reward agents upon reaching accepting states. However, this method operated solely on the product-MDP and did not incorporate counterfactual experience generation. Subsequently, a novel RL approach emerged, leveraging LTL structure to reward agents, known as *LPOPL* [Toro Icarte *et al.* 2018a]. This method served as the precursor to a technique known as *QRM*, which led to *CRM* grounded in the same learning principles of counterfactual experience generation and off-policy learning. Finally, the reward machine formulation reached maturity in Icarte *et al.* [2022].

3.5.2 Specification of Reward Models

In recent years, extensive research has investigated the utilisation of formal languages for task specification.[Cai *et al.* 2021; Vaezipoor *et al.* 2021; Li and Belta 2019; Hasanbeig *et al.* 2018; Toro Icarte *et al.* 2018b; Littman *et al.* 2017; Li *et al.* 2017]. This stems from two advantageous properties inherent in formal languages. Firstly, formal languages provide a natural means to specify reward functions for complex systems. Secondly, the compositional structure of formal languages exposes task information to the agent, facilitating accelerated learning processes. Nevertheless, crafting learning approaches customised for the wide array of languages is a labour-intensive endeavour. Camacho *et al.* [2019] suggested a strategy to overcome this challenge wherein developers specify reward functions in their preferred language. These functions are then converted to reward machines for learning. This idea was developed further in Middleton *et al.* [2020] with the creation of a conversion framework.

However, it is important to recognise that formulating reward functions that consistently induce the desired behaviour can pose challenges. Reward machines generally do not alleviate this issue (although they may simplify the specification of temporally

extended behaviours). For example, a designer may fail to incorporate penalties for specific undesirable states, leading to optimal policies with adverse side effects. Moreover, there exists a potential danger wherein the agent may uncover unintended methods to maximise reward, as noted by Amodei *et al.* [2016]. It has been argued that handcrafted reward functions should be regarded solely as indications of designers' intentions [Hadfield-Menell *et al.* 2017]. There are several alternatives to manually designing reward functions including trajectory preferences [Christiano *et al.* 2017; Akrour *et al.* 2012], feedback strategies [MacGlashan *et al.* 2017; Knox and Stone 2008; Thomaz *et al.* 2006] and demonstrations [Argall *et al.* 2009; Ziebart *et al.* 2008; Abbeel and Ng 2004; Ng and Russell 2000].

3.5.3 Knowledge Exploitation

The idea to increase sample efficiency by exploiting reward machine structure was motivated by analogous methods for leveraging prior knowledge in reinforcement learning. Specifically, several approaches have been developed within the RL domain to exploit prior knowledge, addressing challenges such as reward shaping [Camacho *et al.* 2018; Ng *et al.* 1999], problem decomposition [Mann *et al.* 2015] and data augmentation [Pitis *et al.* 2020; Andrychowicz *et al.* 2017].

In the field of reinforcement learning, HRL is widely regarded as the most promising approach for effectively leveraging decomposition. Foundational works in this domain encompass methods such as *Hierarchical Abstract Machines (HAMs)* [Parr and Russell 1997], the *options framework* [Sutton *et al.* 1999b] and *MAXQ* [Dietterich 2000]. The core concept of hierarchical reinforcement learning revolves around decomposing tasks into simpler problems with reusable solutions. A notable drawback of HRL methods is that despite their efficacy in decomposition, these methods do not ensure convergence to optimal policies due to the inherent constraint imposed by hierarchies on the policy space, which potentially leads to the pruning of optimal policies. An alternative approach to hierarchical reinforcement learning, proposed in Singh [1992], delineates tasks as *sub-goal sequences*. In this paradigm, individual tasks are solved independently, followed by the utilisation of a gating function to facilitate transitions between policies. This concept is similar to the idea of *policy sketches* introduced in Andreas *et al.* [2017]. Notably, while reward machines are limited to regular languages, they still offer greater expressiveness in comparison to sub-goal sequences and policy sketches.

The approach utilised by reward machines to learn from counterfactual experiences shares a number of important similarities with techniques such as *Hindsight Experience Replay (HER)* [Andrychowicz *et al.* 2017] and *Counterfactual Data Augmentation (CoDA)* [Pitis *et al.* 2020]. Hindsight experience replay operates by relabelling experiences to speed up learning, mirroring the strategy employed by reward machines to enhance sample efficiency. However, while HER utilises goal states for relabelling, the reward machine formulation relies on the structure of an automaton. An advantage of the reward machine formulation lies in its capability to encode temporally extended behaviours, which cannot be achieved through HER. Additionally, the approach utilised by reward machines to learn from counterfactual experiences shares similarities with CoDA. CoDA entails generating counterfactual experiences by merging two existing

experiences to create novel counterfactual scenarios, leveraging locally independent causal factors. This differs from reward machines which utilise finite state machines to produce multiple counterfactual experiences based on a single environmental encounter.

3.6 Conclusion

In this chapter, we provided an overview of the field of non-Markovian reward modelling before formally characterising the limitations of current approaches. We began by introducing the hierarchy of non-Markovian reward, allowing reward models to be classified based on the extent to which they rely on historical information. Next, we proved that reward machines are only able to model type-3 non-Markovian reward functions. This was followed by a discussion about why the constrained expression of reward machines stems from the specific state machine variant employed for reward representation. Finally, we proposed counter automata as a solution to the limitations of reward machines. In the next chapter, we introduce a novel abstract machine based on the counter machine formulation that is capable of modelling type-0 non-Markovian reward functions.

Chapter 4

Augmenting Agents with Counter Machines

4.1 Introduction

To address the limited expressive power of reward machines, we introduce a novel abstract machine variant known as a *Counting Reward Automaton (CRA)*. These abstract machines depend on the Turing-complete k-counter machine formulation, allowing them to model the class of type-0 non-Markovian reward functions. As a result, any reward function can be expressed in the CRA framework as long as the reward assignment process is a well-defined algorithm. It follows that the CRA framework is an extremely general approach to modelling reward, compatible with a much larger set of problems than current state machine-based approaches.

The main contributions of this chapter are as follows:

- We introduce *Counting Reward Automata* – A novel abstract machine variant capable of modelling the full class of type-0 non-Markovian reward functions.
- We show that the CRA framework greatly extends the set of tasks for which artificially intelligent agents can emulate the learning patterns characteristic of generally intelligent agents.
- We present the *Counterfactual Q-Learning* algorithm as an illustrative example, demonstrating how automaton-enabled counterfactual reasoning can be used to increase the sample efficiency of any off-policy learning algorithm.
- We prove that *Counterfactual Q-Learning* converges to the optimal policy in the tabular case under the same conditions as *Q-Learning*.
- We prove that reward machines are a special case of the CRA framework.
- We prove that, in all cases, reward machines are more difficult to specify than CRA in terms of the conventionally accepted complexity metrics utilised in the field of automata theory.

- We prove that, in the context of episodic reinforcement learning, the *Counterfactual Q-Learning* and *Counterfactual Reward Machines* algorithms are equally sample efficient.

This chapter is structured as follows. In Section 4.2, we present the CRA framework. We begin by introducing the formulation in Section 4.2.1 before demonstrating how the automaton can be used for reward modelling in Section 4.2.2. Next, in Section 4.2.3, we discuss the relationship between CRA and product MDPs in the context of reinforcement learning. In Section 4.3, we introduce a set of learning algorithms compatible with the CRA framework. This is followed by a thorough analysis of the relationship between CRA and reward machines in Section 4.4. With the necessary formalisms in place, we proceed to evaluate our approach in a number of domains. In Section 4.5, we conduct several experiments in tabular domains, each of increasing complexity. Finally, in Section 4.6, we evaluate our approach on a temporally extended high-dimensional control problem requiring function approximation.

4.2 Counting Reward Automata

4.2.1 Formulation

We now introduce the CRA framework as an approach to modelling reward in both MDPs as well as NMRDPs. A CRA is a k-counter machine that has been augmented with an output function. The output function returns a reward function after each machine transition. Every time the agent interacts with the environment, an abstract description of the interaction is passed to the machine as input. This causes the machine to transition, producing a reward function which is used to reward the agent for its previous interaction.

Definition 1 (Counting Reward Automaton)**.** *Given a set of environment states S, a set of actions A and a set of propositional symbols modelling high-level events $\mathcal{P}$, a k-counter counting reward automaton is defined by a tuple $\langle U, F, u_0, \delta_u, \delta_c, \delta_r, \rangle$, where U is a finite set of non-terminal states, F is a finite set of terminal states ($U \cap F = \emptyset$), u_0 is the initial state, δ_u is the state-transition function, δ_c is the counter-update function and δ_r is the state-reward function:*

$$\delta_u : U \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto U \cup F$$
$$\delta_c : U \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto \{+m : m \in \mathbb{Z}\}^k$$
$$\delta_r : U \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto [S \times A \times S \mapsto [r_{min}, r_{max}]]$$

where $\Sigma = 2^{\mathcal{P}}$ is the input alphabet and ε is used to denote an empty string.

4.2.2 Counting Reward Automaton Operation

We now illustrate how a CRA can be used to model reward through the use of a running example. In this example, we consider the *LetterWorld* environment presented in Figure 4.1. In this environment, a symbol (letter) is associated with certain positions in the

environment. Symbols may either be observed infinitely often or replaced by another symbol after a set number of observations. The agent is able to observe two aspects of the environment. That is, its current xy-location as well as any symbols associated with the position in the environment. The agent is able to move in any of the four cardinal directions. In this example, the following environmental configuration is used. In each episode, the symbol A may be *observed* a fixed number of times before being replaced by the symbol B. A symbol is said to be observed each time the agent transitions into the corresponding location in the environment. The symbols B and C may be observed infinitely often. Specifically, the symbol A may be observed N times within an episode, where N is a discrete random variable following a geometric distribution with support $\mathbb{N}$ [Steyer and Nagel 2017]. The agent is required to observe a sequence of environment symbols which corresponds to a string in the *context-free language (CFL)* described by the set $L = \{A^N BC^N : N \in \mathbb{N}\}$.

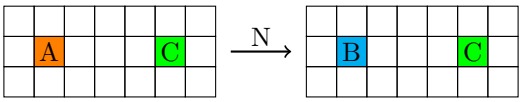


Figure 4.1: Illustration of a *LetterWorld* environment configuration. The symbol A is replaced by B after it has been observed N times by the agent. The agent is required to observe a sequence of environment symbols which corresponds to a string in the context-free language $L = \{A^N BC^N : N \in \mathbb{N}\}$.

The reward model for this task cannot be expressed as a reward machine. This is a consequence of the manner in which the reward model distinguishes between histories of environmental interactions. More precisely, any reward model that distinguishes between histories based on properties not expressible as regular expressions over the elements of the set $S \times A \times S$ cannot be expressed as a reward machine (see Theorem 1). In this example, the reward model counts how many times a state has been reached when assigning rewards. As counting is a property which cannot be expressed in regular languages, reward machines are not able to express this reward model.

Three propositions are used to model the presence of each symbol in the agent's current position $\mathcal{P} = \{P_A, P_B, P_C\}$. After each interaction of the agent with the environment, the labelling function is used to compute the abstract representation, which is used as input to the machine $L(s, a, s') = \sigma \in 2^{\mathcal{P}}$. Upon reading the input symbol σ, the machine transitions states according to δ_u, updates its counters according to δ_c and emits a reward function based on δ_r. This process continues until the machine enters a terminal state.

We now describe how a CRA can be used to model the reward function for the example task. For convenience, we make use of a special case of the CRA formulation known as a *Constant Counting Reward Automaton (CCRA)*. After each transition, a CCRA returns a reward value directly as opposed to a reward function, which is output by a CRA. As a result of the underlying equivalence, throughout this work, we refer to CCRA as CRA.

Definition 2 (Constant Counting Reward Automaton)**.** *Given a set of propositional symbols modelling high-level events $\mathcal{P}$, a constant counting reward automaton is defined as*

a tuple $\langle U, F, u_0, \delta_u, \delta_c, \delta_r \rangle$ *where* U, F, u_0, δ_u *and* δ_c *are defined as in a counting reward automaton; however, the state-reward function* δ_r *is defined as*

$$\delta_r : U \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k \mapsto [r_{min}, r_{max}]$$

where $\Sigma = 2^{\mathcal{P}}$ *is the input alphabet and* ε *is used to denote an empty string.*

Theorem 2. *There exists an equivalent counting reward automaton for each constant counting reward automaton.*

Proof. Let $\mathcal{A}_{\mathcal{P}} = \langle U, F, u_0, \delta_u, \delta_c, \delta_r^c \rangle$ be a CCRA.

The equivalent CRA is defined as $\mathcal{A}_{\mathcal{PSA}} = \langle U, F, u_0, \delta_u, \delta_c, \delta_r^f \rangle$ where U, F, u_0, δ_u and δ_c are defined as in the CCRA.

A machine transition is defined for each triple $\langle u, \sigma, \boldsymbol{\omega} \rangle \in U \times \{\Sigma \cup \{\varepsilon\}\} \times \{0,1\}^k$. Upon reading an input symbol, the machine transitions producing $\delta_r^c(u, \sigma, \boldsymbol{\omega}) = x$ for some $x \in [r_{min}, r_{max}]$. To emulate δ_r^c, the reward function returned by δ_r^f for the transition is required to return x for all inputs in the set $S \times A \times S$. As a result, δ_r^f may return the function defined as

$$f_x : S \times A \times S \to \{x\}$$

$$f_x(s, a, s') = x \;\; \forall \langle s, a, s' \rangle \in S \times A \times S$$

In general,

$$\delta_r^f(u, \sigma, \boldsymbol{\omega}) = f : S \times A \times S \to \{\delta_r^c(u, \sigma, \boldsymbol{\omega})\}$$

□

A graphical representation of the example CCRA is illustrated in Figure 4.2. The machine contains two non-terminal states, one for each subtask in the overall task specification. In the state u_0, the objective of the agent is to observe the symbol A repeatedly. However, in the state u_1, the agent is required to observe the symbol C repeatedly. Intuitively, the automaton stores the number of A symbols it has seen in its counter. After observing the symbol B, the machine decrements its counter each time the symbol C is observed. Once the counter has been decremented to zero, the machine transitions into a terminal state and emits a reward of one.

The automaton is represented as a directed graph. Each vertex in the graph represents a state in the machine, with u_0 being the initial state. Terminal states are represented by filled circles. Associated with each edge in the graph is a tuple $\langle \varphi, \boldsymbol{\omega}, \boldsymbol{\mu}, r \rangle$, in which φ is a propositional logic formula over $\mathcal{P}$, the vector $\boldsymbol{\omega}$ contains the zero-tested counter states, $\boldsymbol{\mu}$ contains the counter modifiers and r is the reward value associated with the transition.

A directed edge between states u_i and u_j labelled by the tuple $\langle \varphi, \boldsymbol{\omega}, \boldsymbol{\mu}, r \rangle$ means that the machine is in state u_i, the truth assignment $L(s, a, s') = \sigma$ satisfies φ, that is $\sigma \models \varphi$, and $Z(\boldsymbol{c}) = \boldsymbol{\omega}$, then the next machine state is equal to u_j, the counter values are updated to $\boldsymbol{c} + \boldsymbol{\mu}$ and the agent is given a reward of r. For example, the edge between u_0 and u_1 labelled by $\langle P_B, [1], [0], 0 \rangle$ means that the machine will transition from state u_0 to state u_1 without modifying the counter value and emit a reward value of zero if the machine

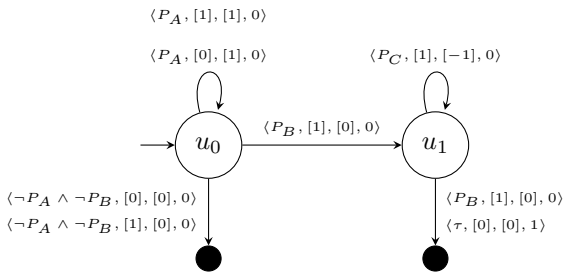


Figure 4.2: Illustration of a CCRA used to solve the example CFL task in the $LetterWorld$ environment. The τ symbol is used to represent a tautology (a propositional formula that is always true), which conditions the corresponding transition only on the states of the counters.

is in state u_0 with a non-zero counter value when the proposition P_B becomes true. Finally, the automaton is updated after every agent-environment interaction. That is, if the agent takes action a in state s resulting in the following state s', the machine configuration is updated to

$$u' = \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$$
$$\boldsymbol{c}' = \delta_c(u, L(s, a, s'), Z(\boldsymbol{c}))$$

and the agent receives a reward of $r = \delta_r(u, L(s, a, s'), Z(\boldsymbol{c}))(s, a, s')$.

4.2.3 Solving the Example Task

With the CCRA in place, we proceed to discuss how a solution to the example task can be obtained. As described, the automaton specifies the reward function for the task. The reward is non-Markovian with respect to the ground environment state. That is, the agent's observation of the environment does not contain a sufficient amount of information to determine the reward. For example, consider the case in which the agent observes that it is in the position associated with the symbol B. As this observation does not contain information about the sequence of symbols previously observed, it is insufficient to determine the reward. However, the machine configuration can be combined with the ground environment state to produce a Markov state. The machine configuration is defined as the current machine state and the values of the counter variables $\langle u, \boldsymbol{c} \rangle$.

Definition 3 (Automaton-Augmented Markov Decision Process)**.** *An automaton-augmented Markov decision process is an MDP in which the reward function is modelled through the use of a counting reward automaton. An AAMDP is a tuple $\langle S, A, P, \gamma, U, F, u_0, \delta_u, \delta_c, \delta_r \rangle$ in which S, A, P and γ are defined based on an environment representation, and $U, F, u_0, \delta_u, \delta_c$ and δ_r are defined as in a counting reward automaton. Each AAMDP induces and equiva-*

lent MDP $\langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, R^{\mathcal{A}}, \gamma^{\mathcal{A}} \rangle$ *in which:*

$$
\begin{aligned}
A^{\mathcal{A}} &= A \\
S^{\mathcal{A}} &= S \times U \cup F \times \{1, ..., \Gamma\}^k \\
\gamma^{\mathcal{A}} &= \gamma \\
P^{\mathcal{A}} &= P(\langle s', u', \boldsymbol{c}' \rangle | \langle s, u, \boldsymbol{c} \rangle, a) \\
&= \begin{cases} P(s'|s,a) & \textit{if } u \in F, u' = u, \boldsymbol{c}' = \boldsymbol{c} \\ P(s'|s,a) & \textit{if } u \notin F, u' = \delta_c(u, \sigma, \boldsymbol{\omega}), \boldsymbol{c}' = \delta_c(u, \sigma, \boldsymbol{\omega}) \\ 0 & \textit{otherwise.} \end{cases} \\
R^{\mathcal{A}} &= \delta_r(u, \sigma, \boldsymbol{c})(s, a, s')
\end{aligned}
$$

where $\sigma = L(s, a, s')$*,* $\boldsymbol{\omega} = Z(\boldsymbol{c})$ *and* Γ *is an upper bound on the value of counters derived from the maximum episode length.*

The combination of the machine configuration with the ground environment state produces a decision process which has a larger state space than that of the NMRDP and belongs to the class of product MDPs as discussed in Section 2.3.5. As in the reward machine formulation, the automaton can be thought of as defining a reward function on the product MDP. Each state in this MDP is formed by combining an automaton configuration with a ground environment state. We refer to the product MDP created by combining CRA configurations with ground environment states as the *Automaton Augmented Markov Decision Process (AAMDP)*. While the general CRA formulation is not limited, we focus on finite-horizon MDPs. That is, we enforce a fixed upper bound on trajectory length. We use the symbol $\mathcal{H}$ to denote the maximum length of any trajectory. This value can be used to compute an upper bound on the value of any machine counter. Specifically, $\mathcal{H}$ is multiplied by the maximum counter increment defined in δ_c to produce the upper bound denoted Γ. As the AAMDP is fundamentally an MDP, conventional reinforcement learning algorithms such as Q-learning can be used to obtain a solution in the form of an optimal policy.

4.3 Learning through Counterfactual Experience

In this section, we discuss learning algorithms for use with the AAMDP formulation. As AAMDPs are fundamentally MDPs, we begin by discussing compatibility with conventional reinforcement learning techniques. This is followed by the introduction of a novel learning algorithm for AAMDPs. This algorithm exploits task information encoded in the CRA of the AAMDP to increase sample efficiency. We provide pseudo-code implementations for both the tabular and function approximation cases as well as discuss each of the algorithm's convergence guarantees.

4.3.1 Product MDP Baseline

Each AAMDP is an MDP which takes into account the state of the underlying CRA at each timestep. As a result, the AAMDP agent considers not only the underlying ground

environment state when selecting actions but also the current CRA configuration $\langle u, \boldsymbol{c} \rangle$. Regardless of this distinction, the construction satisfies the properties of an MDP, and as a result, any learning algorithm compatible with MDPs can be used in the formulation. Any convergence guarantees associated with such algorithms in MDPs hold by extension in AAMDPs. However, as these algorithms do not incorporate automaton information into the learning process, they will not be more sample efficient simply as a result of being applied to an AAMDP.

Algorithm 1 Tabular Q-Learning for AAMDPs

Input: AAMDP $\mathcal{M} = \langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, \gamma^{\mathcal{A}}, R^{\mathcal{A}} \rangle$, learning rate α
Output: Optimal Q-value function Q^*

1: Initialise $Q(s, u, \boldsymbol{c}, a) \leftarrow 0 \; \forall \; \langle s, u, \boldsymbol{c}, a \rangle \in S^{\mathcal{A}} \times A^{\mathcal{A}}$
2: **repeat**
3: Initialise $u \leftarrow u_0$
4: Initialise $\boldsymbol{c} \leftarrow \mathbf{0}$
5: Initialise s as initial environment state
6: **while** $u \notin F$ and s non-terminal **do**
7: Sample a from $\langle s, u, \boldsymbol{c} \rangle$ using policy derived from Q (e.g. ϵ-greedy)
8: Execute action a, observe s'
9: Compute $u' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
10: Compute $\boldsymbol{c}' \leftarrow \delta_c(u, L(s, a, s'), Z(\boldsymbol{c}))$
11: Compute $r \leftarrow \delta_r(u, L(s, a, s'), Z(\boldsymbol{c}))(s, a, s')$
12: **if** $u' \in F$ or s' is terminal **then**
13: $Q(s, u, \boldsymbol{c}, a) \overset{\alpha}{\leftarrow} r$
14: **else**
15: $Q(s, u, \boldsymbol{c}, a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in A} Q(s', u', \boldsymbol{c}', a')$
16: **end if**
17: Set $s \leftarrow s'$
18: Set $u \leftarrow u'$
19: Set $\boldsymbol{c} \leftarrow \boldsymbol{c}'$
20: **end while**
21: **until** end.

To illustrate how a conventional reinforcement learning algorithm can be used with an AAMDP, we provide an adapted version of the tabular Q-learning algorithm as described in Watkins and Dayan [1992]. The pseudocode for this algorithm is provided in Algorithm 1. Firstly, the Q-values are initialised to zero (line 1). After initialisation, the training process is started. At the beginning of each episode, the counter machine configuration is initialised, and the initial ground environment state is sampled (lines 3-5). An episode is then carried out until completion (lines 6-19). At each timestep, the agent samples an action before executing it in the ground environment and observing the next ground environment state (lines 7-8). This information is used to compute the next CRA configuration (lines 9-10) as well as the reward associated with the transition (line 11). Finally, the corresponding Q-value is then updated based on the observed agent-environment interaction (lines 12-16).

4.3.2 Counterfactual Experiences

We now describe how task information encoded with the structure of a CRA can be used to enable more sample-efficient learning. The learning algorithm we propose to exploit CRA task information is based on the *Counterfactual experiences for Reward Machines (CRM)* algorithm [Icarte *et al.* 2022]. However, our algorithm incorporates the required extensions and modifications to support the k-counter machine at the core of the CRA formulation. A notable property of this algorithm is that the generated counterfactual experiences can be used to increase the sample efficiency of any off-policy algorithm. This will be demonstrated when we extend our results to the function approximation domain.

We begin by providing an overview of the intuition behind counterfactual experiences in reinforcement learning. More specifically, we describe how task information encoded within the structure of an automaton can be used to enable more sample-efficient learning. Suppose an agent in a ground environment state s takes action a, resulting in a subsequent state of s'. If the CRA configuration was $\langle u, \boldsymbol{c} \rangle$ before the transition, then the reward signal emitted by the automaton would be:

$$r = \delta_r(u, L(s, a, s'), Z(\boldsymbol{c}))(s, a, s')$$

However, we are able to determine the outcome if the machine had been in any of its possible configurations before the transition. This form of counterfactual reasoning generates a set of counterfactual transitions, each of which may be used for learning. This form of synthetic experience generation is only possible due to the task information encoded within the automaton structure.

In order to illustrate how CRA counterfactual experience generation can be incorporated into an off-policy learning algorithm, we present *Counterfactual Q-Learning (CQL)* in Algorithm 2. From the pseudocode implementation, it can be seen that the only aspect of the algorithm which requires modification is the Q-function update. That is, rather than only updating the Q-function once based on the observed AAMDP transition, it is updated multiple times, considering every possible automaton configuration prior to the transition. The process begins with the agent executing an action and observing the next ground environment state (line 8). This is followed by counterfactual experience generation. That is, we simulate the outcome of the transition had the machine been in any of its non-terminal configurations when the transition took place (lines 9-10)[1]. For each experience, the corresponding next machine configuration and reward are computed (lines 11-13). This information is used to update the Q-function (lines 14-18). Finally, after the process of counterfactual experience generation is complete, the machine configuration is updated based on the transition which was observed in reality.

Theorem 3. *Given an AAMDP $\mathcal{M}$, tabular CQL converges to an optimal policy for $\mathcal{M}$ in the limit (under the same conditions required for convergence of Q-learning in MDPs).*

[1]The loop on line 10 can be optimised in practice by maintaining a cache of observed counter states. This prevents learning over counter values, which are not required to solve the task.

Algorithm 2 Counterfactual Q-Learning (CQL)

Input: AAMDP $\mathcal{M} = \langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, \gamma^{\mathcal{A}}, R^{\mathcal{A}} \rangle$, learning rate α, upper bound on counter values Γ
Output: Optimal Q-value function Q^*

Initialise $Q(s, u, \boldsymbol{c}, a) \leftarrow 0 \; \forall \; \langle s, u, \boldsymbol{c}, a \rangle \in S^{\mathcal{A}} \times A^{\mathcal{A}}$
repeat
  Initialise $u \leftarrow u_0$
  Initialise $\boldsymbol{c} \leftarrow \mathbf{0}$
  Initialise s as initial environment state
  while $u \notin F$ and s non-terminal **do**
    Sample a from $\langle s, u, \boldsymbol{c} \rangle$ using policy derived from Q (e.g. ϵ-greedy)
    Execute action a, observe s'
    for $u_i \in U$ **do**
      for $\boldsymbol{c_i} \in \{1, ..., \Gamma\}^k$ **do**
        $u_j \leftarrow \delta_u(u_i, L(s, a, s'), Z(\boldsymbol{c_i}))$
        $\boldsymbol{c_j} \leftarrow \delta_c(u_i, L(s, a, s'), Z(\boldsymbol{c_i}))$
        $r_j \leftarrow \delta_r(u_i, L(s, a, s'), Z(\boldsymbol{c_i}))$
        if $u_j \in F$ or s' is terminal **then**
          $Q(s, u_i, \boldsymbol{c_i}, a) \overset{\alpha}{\leftarrow} r_j$
        else
          $Q(s, u_i, \boldsymbol{c_i}, a) \overset{\alpha}{\leftarrow} r_j + \gamma \max_{a' \in A} Q(s', u_j, \boldsymbol{c_j}, a')$
        end if
      end for
    end for
    Compute $u' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Compute $\boldsymbol{c'} \leftarrow \delta_c(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Set $s \leftarrow s'$
    Set $u \leftarrow u'$
    Set $\boldsymbol{c} \leftarrow \boldsymbol{c'}$
  end while
until end

Proof. As the counterfactual experiences generated in CQL are sampled according to the transition probability distribution $P(\langle s', u', \boldsymbol{c'} \rangle \mid \langle s, u, \boldsymbol{c} \rangle, a) = P(s'|s, a)$, the convergence proof provided by Watkins and Dayan [1992] for tabular Q-learning applies directly to the case of CQL. □

4.4 Relationship to Reward Machines

In this section, we explore the connection between counting reward automata and reward machines. We begin by showing that reward machines are a special case of the CRA framework. Following this, we examine the relationship between the formulations in the context of episodic reinforcement learning, which is significantly more nuanced.

The primary findings of this examination are twofold. Firstly, reward machines are significantly more challenging to apply in this setting than counting reward automata. Secondly, in this context, both formulations are equally sample-efficient. This comprehensive analysis demonstrates that the CRA framework equals or surpasses reward machines across all key metrics, including state machine complexity, sample efficiency, and overall expressiveness.

4.4.1 Reward Machines – A Special Case of the CRA Framework

As discussed, counter machines can informally be thought of as extended finite-state machines that include integer-valued variables known as counters. It follows that the counter machine formulation can be used to emulate[2] any finite-state machine. Specifically, an arbitrary finite state machine can be emulated by a counter machine which utilises the same automaton structure and does not modify its counters.

The idea of emulation illustrates an important aspect of the relationship between the CRA and RM formulations. That is, reward machines are a special case of the CRA formulation. This is a consequence of the fact that an equivalent CRA exists for any reward machine. For any sequence of inputs, such a CRA will produce an identical sequence of outputs to that of the RM. These notions are formalised in Theorem 4.

Theorem 4. *Any reward machine can be emulated by a CRA with an equivalent number of machine states and transitions.*

Proof. Let $\mathcal{R}_{\mathcal{PSA}} = \langle U^{\mathcal{R}}, u_0^{\mathcal{R}}, F^{\mathcal{R}}, \delta_u^{\mathcal{R}}, \delta_r^{\mathcal{R}} \rangle$ be a reward machine.

The number of RM states is equal to $|U^{\mathcal{R}} \cup F^{\mathcal{R}}|$. By definition, an RM transition exists for each pair $\langle u, \sigma \rangle \in (U^{\mathcal{R}} \cup F^{\mathcal{R}}) \times 2^{\mathcal{P}}$. It follows that the number of RM transitions is equal to $|(U^{\mathcal{R}} \cup F^{\mathcal{R}}) \times 2^{\mathcal{P}}|$.

We now show how to define a CRA capable of emulating the RM. Firstly, to emulate the RM, the CRA does not require the use of its counters. Therefore, the counter values remain $\mathbf{0}$ at all times, as they are not modified in any transitions. Secondly, the CRA only transitions when the RM transitions, that is, after reading an input symbol. The CRA is defined as $\mathcal{A}_{\mathcal{PSA}} = \langle U^{\mathcal{A}}, F^{\mathcal{A}}, u_0^{\mathcal{A}}, \delta_u^{\mathcal{A}}, \delta_c^{\mathcal{A}}, \delta_r^{\mathcal{A}} \rangle$ where $U^{\mathcal{A}} = U^{\mathcal{R}}$, $F^{\mathcal{A}} = F^{\mathcal{R}}$, $u_0^{\mathcal{A}} = u_0^{\mathcal{R}}$,

$$\begin{aligned}
\delta_c^{\mathcal{A}}(u, \sigma, \mathbf{0}) &= +\mathbf{0} \;\; \forall \, \langle u, \sigma \rangle \in (U^{\mathcal{A}} \cup F^{\mathcal{A}}) \times 2^{\mathcal{P}} \\
\delta_u^{\mathcal{A}}(u, \sigma, \mathbf{0}) &= \delta_u^{\mathcal{R}}(u, \sigma) \;\; \forall \, \langle u, \sigma \rangle \in (U^{\mathcal{A}} \cup F^{\mathcal{A}}) \times 2^{\mathcal{P}} \\
\delta_r^{\mathcal{A}}(u, \sigma, \mathbf{0}) &= \delta_r^{\mathcal{R}}(u) \;\; \forall \, \langle u, \sigma \rangle \in (U^{\mathcal{A}} \cup F^{\mathcal{A}}) \times 2^{\mathcal{P}}
\end{aligned}$$

From $U^{\mathcal{A}} = U^{\mathcal{R}}$ and $F^{\mathcal{A}} = F^{\mathcal{R}}$, it follows that the machines have the same number of states. As the cardinality of the domain of the CRA transition functions is equal to that of the RM transition function $|(U^{\mathcal{A}} \cup F^{\mathcal{A}}) \times 2^{\mathcal{P}} \times \{\mathbf{0}\}| = |(U^{\mathcal{R}} \cup F^{\mathcal{R}}) \times 2^{\mathcal{P}}|$ it follows that the machines have the same number of transitions. □

[2] Reproduce the function of another system.

4.4.2 Episodic Reinforcement Learning

The Emulating Reward Machine

As discussed, reward machines would require an infinite number of states in order to express a task that is only expressible in non-regular languages. Thus, by definition, reward machines cannot be applied in such a setting as they are **finite** state machines. However, in the context of episodic reinforcement learning, a finite upper bound exists on episode length. Consequently, any task can be represented as a finite sequence of steps and, as a result, can be represented as a finite state machine. This illustrates an important property of the relationship between the CRA and RM formulations in the context of episodic reinforcement learning. That is, for any CRA applied within this context, there exists an equivalent RM, which we refer to as the *emulating reward machine*. This result is formalised in Theorem 5.

Theorem 5. *In the context of episodic reinforcement learning, any CRA can be emulated by a reward machine.*

Proof. As CRA are Mealy machines, we define the emulating reward machine through the use of the equivalent Mealy machine formulation of reward machines [Icarte *et al.* 2018; Camacho *et al.* 2019; Icarte *et al.* 2022].

Let $\mathcal{A}_\mathcal{C} = \langle U^c, F^c, u_0^c, \delta_u^c, \delta_c^c, \delta_r^c \rangle$ be a CRA and $\mathcal{H}$ denote the maximum episode length. Use Γ to denote the upper bound on counter values, which is obtained by multiplying $\mathcal{H}$ by the largest counter increment defined in δ_c^c.

Define $\Omega = \{1, ..., \Gamma\}$ as the set of values which may be taken on by a counter variable and use $C = \{\boldsymbol{c} : \boldsymbol{c} \in \Omega^k\}$ to denote the set of all counter configurations.

Finally, define the mapping $\phi : U^c \times C \mapsto U^c \times \Omega^k$ where

$$\begin{aligned}\phi(\langle u, \boldsymbol{c} \rangle) &= \phi(\langle u, \langle c_1, c_2, ..., c_k \rangle \rangle) \\ &= \langle u, c_1, c_2, ..., c_k \rangle\end{aligned}$$

Then the CRA $\mathcal{A}_\mathcal{C}$ can be represented as the RM $\mathcal{A}_\mathcal{R} = \langle U^r, u_0^r, F^r, \delta_u^r, \delta_r^r \rangle$ with

$$\begin{aligned}U^r &= \{\phi(\langle u, \boldsymbol{c} \rangle) : \langle u, \boldsymbol{c} \rangle \in U^c \times C\} \\ F^r &= \{\phi(\langle f, \boldsymbol{c} \rangle) : \langle f, \boldsymbol{c} \rangle \in F^c \times C\} \\ u_0^r &= \langle u_0^c, \mathbf{0} \rangle \\ \delta_u^r(\phi(\langle u, \boldsymbol{c} \rangle), \sigma) &= \phi(\langle \delta_u^c(u, \sigma, Z(\boldsymbol{c})), \delta_c^c(u, \sigma, Z(\boldsymbol{c})) \rangle) \\ \delta_r^r(\phi(\langle u, \boldsymbol{c} \rangle), \sigma) &= \delta_r^c(u, \sigma, Z(\boldsymbol{c}))\end{aligned}$$

□

While the existence of the emulating reward machine may seem to suggest sufficiency of the formulation for non-Markovian reward modelling[3], this is not the case. As a result of the complexity of the reward machines required for CRA emulation, the approach quickly becomes infeasible for any tasks of practical interest. More specifically, the emulation process produces exceedingly complex automata when applied to non-regular tasks[4]. Not only are these automata difficult to specify, but they are nearly impossible to verify for correctness due to their size and complexity. An important property in the field of reinforcement learning where an incorrectly defined reward model can lead to significant wastage of computational resources and time [Patterson *et al.* 2020]. The aforementioned drawbacks associated with emulating reward machines have significant theoretical as well as practical implications.

These automata are significantly more complex than their CRA counterparts in terms of the conventionally accepted complexity metrics utilised in the field of automata theory. That is the number of states and transitions required to define an automaton [Gruber and Holzer 2007]. These metrics are designed to quantify the difficulty of the automaton design and verification processes [Mitchell 2009]. For example, larger automata with more states and transitions are more difficult to specify and verify for correctness than smaller automata. The relationship between the number of RM and CRA states and transitions is formalised in Theorems 6 and 7. That is, in all cases, the emulating RM is more difficult to specify than the original CRA as it requires a larger number of machine states (Theorem 6) and, in the best case, an equal number of transitions (Theorem 7).

Theorem 6. *The number of states in an emulating reward machine is greater than that of the original CRA.*

Proof. Let $\mathcal{A}_\mathcal{C} = \langle U^c, F^c, u_0^c, \delta_u^c, \delta_c^c, \delta_r^c \rangle$ be a CRA. From the definition of a CRA, it follows that the number of states in the CRA is equal to $|U^c \cup F^c|$.

From Theorem 5, it follows that the number of states in the emulating reward machine is equal to:

$$\begin{aligned} |U^r \cup F^r| &= |(U^c \cup F^c) \times C| \\ &= |(U^c \cup F^c) \times \{0, ..., \Gamma\}^k| \\ &= |U^c \cup F^c| \cdot |\{0, ..., \Gamma\}^k| && (|A \times B| = |A| \cdot |B|) \\ &= |U^c \cup F^c| \cdot (\Gamma + 1)^k \end{aligned}$$

As the minimum length of an episode is one, it follows that $\Gamma \geq 1$. Therefore, in all cases:

$$\begin{aligned} |U^r \cup F^r| &= |U^c \cup F^c| \cdot (\Gamma + 1)^k \\ &> |U^c \cup F^c| && (\Gamma \geq 1, k \geq 1) \end{aligned}$$

[3]Modelling the reward function of any task expressible in a decidable formal language in the context of episodic reinforcement learning.

[4]Task specifications which cannot be expressed in regular languages.

□

Theorem 7. *The number of transitions in an emulating reward machine is greater than or equal to that of the original CRA.*

Proof. Let $\mathcal{A}_\mathcal{C} = \langle U^c, F^c, u_0^c, \delta_u^c, \delta_c^c, \delta_r^c \rangle$ be a CRA. From the definition of a CRA, it follows that the number of transitions is equal to $|\text{dom}(\delta_u^c)|$, where $\text{dom}(f)$ is used to denote the domain of a given function f. As a result, the number of transitions defined for the automaton is equal to:

$$\begin{aligned} |\text{dom}(\delta_u^c)| &= |U^c \times \{0,1\}^k \times \Sigma| \\ &= |U^c| \cdot |\{0,1\}^k| \cdot |\Sigma| && (|A \times B| = |A| \cdot |B|) \\ &= |U^c| \cdot 2^k \cdot |\Sigma| \end{aligned}$$

Similarly, the number of transitions in the emulating reward machine is equal to $|\text{dom}(\delta_u^r)|$. From Theorem 5, it follows that the number of transitions defined for this automaton is equal to:

$$\begin{aligned} |\text{dom}(\delta_u^r)| &= |U^c \times \{0, ..., \Gamma\}^k \times \Sigma| \\ &= |U^c| \cdot |\{0, ..., \Gamma\}^k| \cdot |\Sigma| && (|A \times B| = |A| \cdot |B|) \\ &= |U^c| \cdot (\Gamma + 1)^k \cdot |\Sigma| \end{aligned}$$

As the minimum length of an episode is one, it follows that $\Gamma \geq 1$. Therefore, in all cases:

$$\begin{aligned} |\text{dom}(\delta_u^r)| &= |U^c| \cdot (\Gamma + 1)^k \cdot |\Sigma| \\ &\geq |U^c| \cdot 2^k \cdot |\Sigma| && (\Gamma \geq 1, k \geq 1) \\ &= |\text{dom}(\delta_u^c)| \end{aligned}$$

□

Not only are emulating reward machines undesirable from the perspective of automata theory, but several practical considerations further illustrate the limitations of this approach. The transition function of an automaton operates based on the machine configuration and input symbol. The configuration of a CRA is equal to the current machine state as well as the value of the counter variables $\langle u, \boldsymbol{c} \rangle$. In contrast, the configuration of a reward machine is simply equal to the current machine state $\langle u \rangle$. This representation complicates the implementation process. For example, when implementing the transition functions of a CRA, there is a natural relationship between function inputs and outputs. Consider the case where the machine has a configuration of $\langle u_0^c, [0] \rangle$, and the machine observes a proposition causing it to move to state u_1^c. Naturally, the next machine configuration is equal to $\langle u_1^c, [0] \rangle$. Understanding the properties of the transition function of reward machines proves considerably more challenging. Consider the case in which the emulating RM has a configuration u_0^r corresponding to the CRA configuration $\langle u_0^c, [0] \rangle$. Once again, the machine observes a proposition which would

cause the CRA to move to state u_1^c. There is no natural way to reason about the value of the next RM state. For example, u_1^r may be used to denote either of the following two CRA configurations $\langle u_0^c, [1]\rangle$ or $\langle u_1^c, [0]\rangle$. There is simply not enough information in the RM representation to deduce the following machine configuration. While the use of atomic representations for machine configurations does not impose any theoretical limitations on the RM formulation, it complicates the implementation process. As a result, more complex specification algorithms are required when implementing emulating reward machines in comparison to CRA, which make use of a more natural factored representation.

Counterfactual Experience Generation

In this section, we compare the CRA counterfactual experience generation process to that of emulating reward machines. We specifically focus on the amount of information contained within the sets of counterfactual experiences produced by either of the approaches. A particular focus is placed on this aspect of the algorithms as it has important implications for sample efficiency.

Both approaches follow a similar process for counterfactual experience generation. That is, for a given ground environment transition $\langle s, a, s'\rangle$, they consider a set of outcomes had the machine been in any of its configurations prior to the transition. As the set of configurations of an emulating reward machine is simply an alternative representation to that of the original CRA, it follows that both sets of counterfactual experience contain an equivalent amount of information. This result is formalised in Theorem 8.

Theorem 8. *The set of counterfactual experiences generated by an emulating reward machine is an alternative representation to that of the original CRA.*

Proof. A counterfactual experience is defined as a tuple $\langle \mu, a, \mu', r\rangle$ where μ, μ' are machine configurations, a is an action and r is a reward value.

Let $E^{\mathcal{A}}$ denote the set of counterfactual experiences generated by the original CRA. Similarly, let $E^{\mathcal{R}}$ denote the set of counterfactual experiences generated by the emulating reward machine.

To prove the result, we show that there exists a mapping $\varphi : E^{\mathcal{A}} \mapsto E^{\mathcal{R}}$ which is a bijection, where φ is defined as:

$$\varphi(\langle\langle u, \boldsymbol{c}\rangle, a, \langle u', \boldsymbol{c}'\rangle, r\rangle) = \langle \phi(\langle u, \boldsymbol{c}\rangle), a, \phi(\langle u', \boldsymbol{c}'\rangle), r\rangle$$

where ϕ is defined as in Theorem 5.

(injective) Let $x, y \in E^{\mathcal{A}}$ be defined as:

$$x = \langle\langle u_1, \boldsymbol{c_1}\rangle, a_1, \langle u_1', \boldsymbol{c_1}'\rangle, r_1\rangle$$
$$y = \langle\langle u_2, \boldsymbol{c_2}\rangle, a_2, \langle u_2', \boldsymbol{c_2}'\rangle, r_2\rangle$$

Assume $\varphi(x) = \varphi(y)$, then:

$$\langle \phi(\langle u_1, \boldsymbol{c_1}\rangle), a_1, \phi(\langle u_1', \boldsymbol{c_1}'\rangle), r_1\rangle = \langle \phi(\langle u_2, \boldsymbol{c_2}\rangle), a_2, \phi(\langle u_2', \boldsymbol{c_2}'\rangle), r_2\rangle$$
$$\implies \langle\langle u_1, c_1^1, ..., c_1^k\rangle, a_1, \langle u_1', c_1'^1, ..., c_1'^k\rangle, r_1\rangle = \langle\langle u_2, c_2^1, ..., c_2^k\rangle, a_2, \langle u_2', c_2'^1, ..., c_2'^k\rangle, r_2\rangle$$

write the values $c_x^1, ..., c_x^n$ using a vector representation $\boldsymbol{c_x} = [c_x^1, ..., c_x^n]^T$

$$\begin{aligned} &\implies \langle\langle u_1, \boldsymbol{c_1}\rangle, a_1, \langle u_1', \boldsymbol{c_1}'\rangle, r_1\rangle = \langle\langle u_2, \boldsymbol{c_2}\rangle, a_2, \langle u_2', \boldsymbol{c_2}'\rangle, r_2\rangle \\ &\implies x = y \end{aligned}$$

(surjective) Let $y = \langle\langle u, c^1, ..., c^k\rangle, a, \langle u', c'^1, ..., c'^k\rangle, r\rangle \in E^{\mathcal{R}}$.

Let $\boldsymbol{c}$ denote a vector representation of the values $c^1, ..., c^k$ in y, that is $\boldsymbol{c} = [c^1, ..., c^k]^T$. Similarly, let $\boldsymbol{c}'$ denote a vector representation of the values $c'^1, ..., c'^k$ in y, that is $\boldsymbol{c}' = [c'^1, ..., c'^k]^T$.

For $x = \langle\langle u, \boldsymbol{c}\rangle, a, \langle u', \boldsymbol{c}'\rangle, r\rangle \in E^{\mathcal{A}}$ we have that:

$$\begin{aligned} \varphi(x) &= \langle\phi(\langle u, \boldsymbol{c}\rangle), a, \phi(\langle u', \boldsymbol{c}'\rangle), r\rangle \\ &= \langle\langle u, c^1, ..., c^k\rangle, a, \langle u', c'^1, ..., c'^k\rangle, r\rangle \\ &= y \end{aligned}$$

□

Corollary 8.1. *In the context of episodic reinforcement learning, the CQL and CRM algorithms are equally sample-efficient.*

Proof. The CQL and CRM algorithms utilise the introduction of counterfactual experiences as a mechanism to increase sample efficiency. However, in the context of episodic reinforcement learning, the sets of counterfactual experiences generated by either of the approaches contain equivalent amounts of information (Theorem 8). As a result, the CQL and CRM algorithms make use of the same information when updating the underlying policy during the learning process. It follows that, in this context, the CQL and CRM algorithms are equally sample-efficient. □

4.5 Investigating Practical Considerations

We begin this section by discussing our approach to experimental design. Next, we cover a series of carefully constructed experiments designed to evaluate the sample efficiency of our approach. Finally, we compare and contrast the complexity of the state machines produced by the CRA framework against that of the reward machine formulation. By delving into these characteristics, we aim to shed light on the effectiveness of our approach and its implications for the broader landscape of reinforcement learning methodologies.

4.5.1 Empirical Design

Empirical design in reinforcement learning is a complex task. The development of useful experiments requires both attention to detail as well as significant amounts of computational resources. As the complexity and scale of experiments increase, there

is often a conflict with the need for adequate statistical evidence when comparing algorithms. As a result, we follow the set of best practices described in Patterson *et al.* [2020] for experimental design, utilising the prescribed approach for performance estimation.

Reporting the mean performance of an agent across multiple independent evaluations is a well-known convention in reinforcement learning. We follow this convention. However, quantifying the uncertainty in these performance estimates is a non-trivial task. The performance distribution not being Gaussian, is a common scenario in many cases. In this context, employing Student's-t confidence intervals to quantify uncertainty is not suitable. To derive confidence intervals, we resort to bootstrap-based statistics, employing the following bootstrap procedure. We begin by evaluating each algorithm n times to obtain n measures of performance. Each evaluation makes use of a unique random seed. Subsequently, a new dataset of performance measures is generated by resampling n values from the original dataset with replacement. The resampling procedure is iterated a total of m times, with m set to a very large value (e.g. $m = 10,000$). The statistic of interest (e.g. the mean) is computed for each resulting dataset, followed by assessing the variability of that statistic across all m datasets. Generating a 95% confidence interval involves reporting the 0.025 percentile of the estimated statistic across all m estimates as the lower bound and the 0.975 percentile as the upper bound. All results in this research are presented with an associated 95% confidence interval.

4.5.2 Sample Efficiency

In this section, we evaluate the sample efficiency of several algorithms when applied to tasks of varying complexity in tabular domains. Specifically, we compare the CQL, CRM and Q-learning algorithms in two tabular domains. In the first experiment, the agent is required to solve a context-free task specification[5] in the *LetterWorld* environment. The following experiment is performed in a more difficult domain and makes use of a more complex context-sensitive[6] task specification, requiring additional agent memory.

LetterWorld

In this experiment, we evaluate the performance of the aforementioned algorithms when applied to the CFL task used as an illustrative example in Section 4.2.2. This experiment takes place in the *LetterWorld* environment illustrated in Figure 4.1. As discussed, the agent is required to observe a sequence of environment symbols which corresponds to a string in the context-free language $L = \{A^N BC^N : N \in \mathbb{N}\}$. Specifically, the value of N is sampled from the discrete uniform distribution $\mathcal{U}\{1, 3\}$ to ensure the task can be completed within each episode. A sparse reward model is used in which the agent is given a reward value of one upon task completion and zero otherwise. Each agent is trained for 100,000 environmental interactions. The success rate, that is, the

[5] A task specification only expressible in a context-free language.
[6] A task specification only expressible in a context-sensitive language.

ability of the agent to complete the task specification, is recorded throughout the training process. The results are illustrated in Figure 4.3.

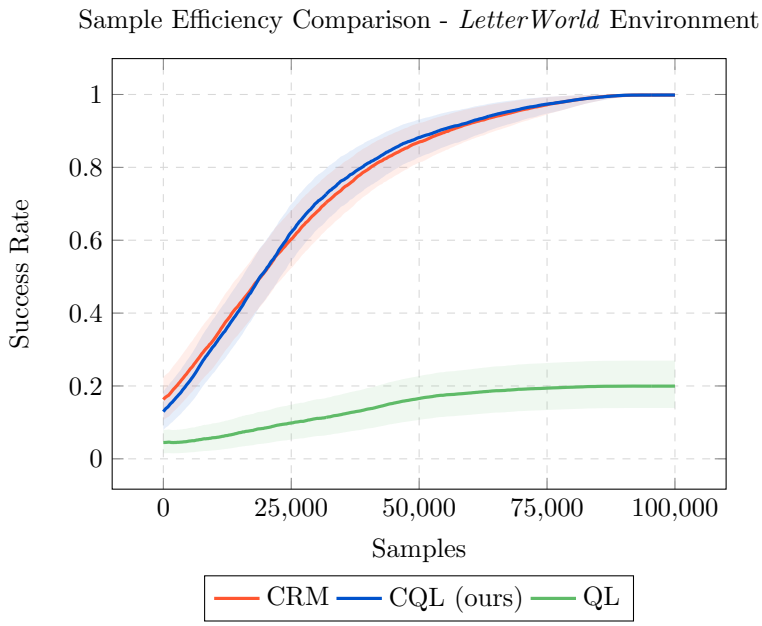


Figure 4.3: Sample efficiency comparison for the *LetterWorld* environment. Results show mean agent performance over several independent runs. Uncertainty is represented using a 95% confidence interval around the mean. Model hyperparameters are detailed in Appendix A.1, while parameters for performance estimation can be found in Appendix B.

The results demonstrate a significant increase in sample efficiency with the introduction of counterfactual experiences. The CQL and CRM algorithms are shown to measurably outperform Q-learning well beyond the associated uncertainty bounds. As expected from Corollary 8.1, the CQL and CRM algorithms exhibit nearly identical results.

OfficeWorld

The well-known *OfficeWorld* environment, illustrated in Figure 2.7, is used to evaluate our method on a more complex task specification. Specifically, we consider a task specification that is only expressible as a context-sensitive language. In order to complete the task, the agent is required to perform a precise sequence of operations. Firstly, the agent is required to navigate to the mail room and collect all available mail. The amount of mail present in the environment at the start of each episode is sampled from the discrete uniform distribution $\mathcal{U}\{1, 3\}$. The agent is then required to prepare a single coffee for the owner of each item of mail. Finally, the collected mail and coffee must be delivered to the employees located in the office. The task is immediately failed if the agent breaks a decoration, prepares an incorrect number of coffees or tries to deliver an incorrect number of items. An illustration of an example agent trajectory is provided in Figure 4.4.

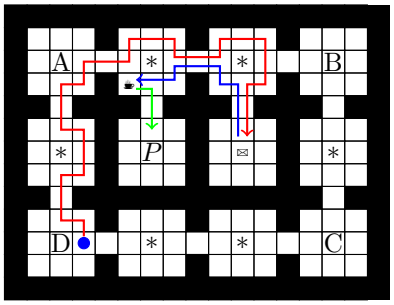


Figure 4.4: Illustration of an example agent trajectory for the context-sensitive task specification in the *OfficeWorld* environment. The agent begins by navigating to the mail room and collecting all available mail as shown in red. Next, the agent prepares a coffee for the owner of each item of mail as shown in blue. Finally, the agent delivers the coffee and mail to the employees located in the office as shown in green.

This task requires the agent to remember both the amount of mail it has collected and the number of coffees that have been prepared. As a result of this property, the task can only be represented as a context-sensitive language. The memory requirements associated with the task specification require the use of two counter variables in the CRA formulation. Once again, we compare the sample efficiency of the CQL, CRM and Q-Learning algorithms using a sparse reward model. Specifically, the agent is given a reward value of one upon task completion and zero otherwise. The results of the comparison are illustrated in Figure 4.5.

We observe similar results to the previous experiment with CQL and CRM significantly outperforming the Q-Learning algorithm. An increase in the relative performance difference between CQL/CRM and Q-learning in comparison to the previous experiment is evident from the results. This is a consequence of the increase in task complexity between the two experiments. The likelihood of task completion with random action selection decreases with an increase in task complexity. As a result, the Q-learning algorithm is less frequently provided with useful learning signals from the sparse reward model. The CQL and CRM algorithms are able to combat this issue through the introduction of counterfactual experience generation, which enables learning even in the event of task failure.

4.5.3 State Machine Complexity

With a sample efficiency comparison in place, we proceed to demonstrate one of the key advantages of the CRA formulation. That is, we compare the complexity of the state machines produced by either of the approaches. As discussed in Theorem 5, in the context of episodic reinforcement learning, counting reward automata can be emulated by reward machines. However, the RM formulation produces more complex automata than the CRA formulation. These automata quickly become exceedingly complex when

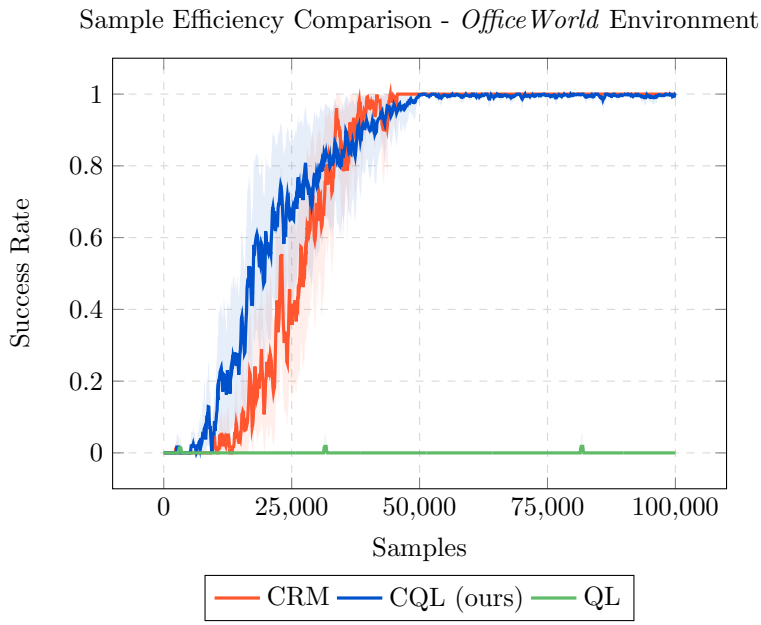


Figure 4.5: Sample efficiency comparison for the *OfficeWorld* environment. Results show mean agent performance over several independent runs. Uncertainty is represented using a 95% confidence interval around the mean. Model hyperparameters are detailed in Appendix A.2, while parameters for performance estimation can be found in Appendix B.

applied to non-regular tasks[7]. As discussed in Section 4.4.2, this property of the RM formulation has both theoretical as well as practical implications.

We begin with a theoretical comparison of the complexity of the state machines required by either of the approaches to solve the *OfficeWorld* task described in Section 4.5.2. That is, we produce a reward model for the task using either of the approaches before comparing the number of machine states and transitions in the resulting automata. The results are shown in Figure 4.6.

From the results, two important aspects of the relationship between CRA and emulating reward machines are apparent. Firstly, the complexity of the CRA is not dependent on the maximum length of a task string. Consequently, a single automaton can be used regardless of the maximum task string length. This is not true for emulating reward machines where a unique automaton exists for each upper bound on task string length. Secondly, even for a relatively simple task specification, the number of states and transitions in the emulating reward machine quickly explodes. As a result, verification of these reward models quickly becomes infeasible without the introduction of complex verification algorithms, which are not required for CRA. This shows that for any non-trivial task specification, emulating reward machines are significantly more complex to specify and verify than CRA.

[7]Task specifications which cannot be expressed in regular languages.

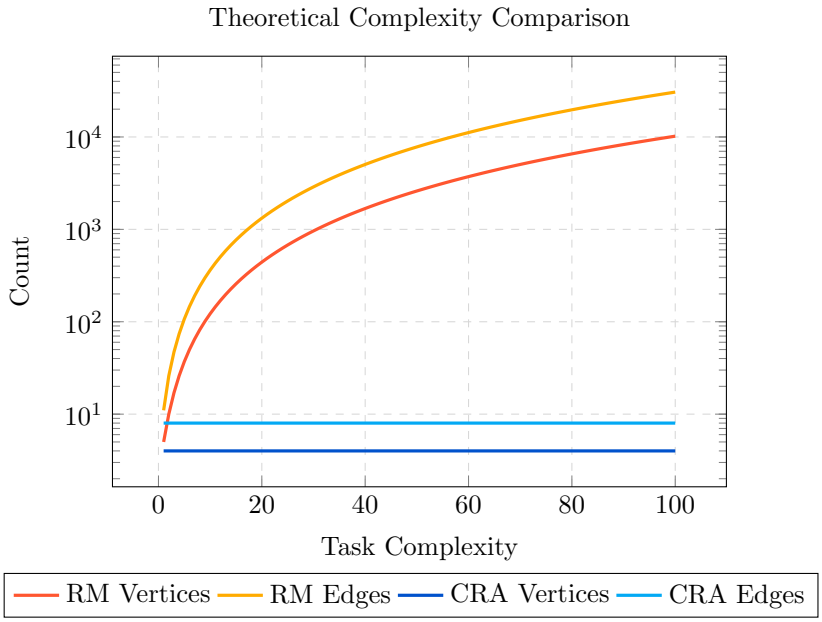


Figure 4.6: Automaton complexity comparison between the CRA and RM formulations. Task complexity represents the upper bound on the length of a task string.

Emulating reward machines are not only more complex in a theoretical sense, as quantified by the field of automata theory, but also require significantly more complex specification algorithms for implementation in high-level programming languages. This is a consequence of the limitations discussed in Section 4.4.2.

We now analyse the complexity of the algorithms used to specify the state machines required by either of the approaches to solve the *OfficeWorld* task described in Section 4.5.2. The complexity of the specification algorithms is quantified by computing the *cyclomatic complexity* of the implementations. Cyclomatic complexity is a software metric used to indicate the complexity of a program [Albrecht and Gaffney 1983]. This metric is computed based on the number of linearly independent paths through the control-flow graph of a program. In this experiment, each automaton is implemented as a directed graph. The introduction of this standardised representation ensures that the implementation comparison is fair. The results of the comparison are shown in Figure 4.7. From the results, it is clear that emulating reward machines are significantly more complex to implement in high-level programming languages.

4.6 Function Approximation

In this section, we discuss how the CRA formulation can be extended for use with function approximation algorithms. We begin with an analysis of existing approaches, describing how reward machines can be used with function approximation. Next, we discuss a number of key limitations in existing approaches and propose a corresponding set of solutions. Specifically, we show that current techniques incorrectly modify function approximation algorithms for use with counterfactual experience, adversely

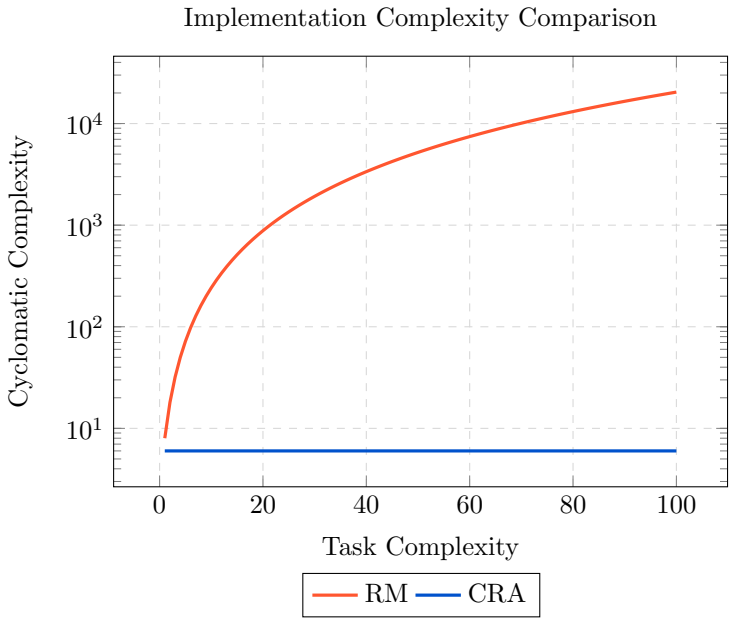


Figure 4.7: Automaton complexity comparison between the CRA and RM formulations. Task complexity represents the upper bound on the length of a task string.

affecting sample efficiency. Finally, we perform an empirical evaluation of the proposed techniques on a complex high-dimensional control problem.

4.6.1 Existing Approaches

A common property of real-world problems in which we would like to apply reinforcement learning is that the state space is often extremely large or infinite. As discussed in Section 2.2.4, it is intractable to find optimal policies or value functions in these settings. Consequently, we focus on computing good approximations using limited computational resources. Regardless of the off-policy algorithm, the reward machine formulation follows the same approach when being applied in the context of function approximation. Specifically, Q-learning is replaced with a modified off-policy function approximation algorithm. Two aspects of this algorithm are modified. That is the model hyperparameters as well as the *replay memory*. To illustrate how reward machines can be applied in conjunction with function approximation, we make use of the *Deep Q-learning (DQN)* algorithm [Mnih *et al.* 2013]. The learning process is similar to that of the tabular case. At each time step, the agent executes an action in the ground environment and observes the next state. The labelling function is then used to compute an abstract representation of the transition, which is used as input by the machine. Upon reading the input symbol, the machine updates its configuration and emits a reward. This information is used to compute a set of counterfactual experiences which are added to the replay memory. A single gradient update is then performed before proceeding to the next time step. To facilitate learning with counterfactual experiences, the RM formulation modifies the size of the replay memory as well as the number of samples used to compute gradients during optimisation. Specifically, the number of

samples used to compute gradients is equal to the default value multiplied by the number of automaton states. The size of the replay memory is increased in a similar manner. A pseudocode implementation of the modified DQN algorithm is given in Algorithm 3.

Algorithm 3 CRM With Function Approximation

Input: Cross-product MDP $\mathcal{M} = \langle S^{\mathcal{R}}, A^{\mathcal{R}}, P^{\mathcal{R}}, \gamma^{\mathcal{R}}, R^{\mathcal{R}} \rangle$, batch size N_b, replay memory size N_m, target network update frequency C.

1: Initialise replay memory $\mathcal{D}$ with capacity $N_m \cdot |U|$
2: Initialise Q-network $Q_\theta : S^{\mathcal{R}} \times A^{\mathcal{R}} \mapsto \mathbb{R}$ with random weights θ
3: Initialise target network $\bar{Q}_{\bar{\theta}} : S^{\mathcal{R}} \times A^{\mathcal{R}} \mapsto \mathbb{R}$ with random weights $\bar{\theta}$
4: **repeat**
5: Initialise $u \leftarrow u_0$
6: Initialise s as initial environment state
7: **while** $u \notin F$ and s non-terminal **do**
8: Sample a from $\langle s, u \rangle$ using policy derived from Q_θ (e.g. ϵ-greedy)
9: Execute action a, observe s'
10: **for** $u_i \in U$ **do**
11: $u_j \leftarrow \delta_u(u_i, L(s, a, s'))$
12: $r_j \leftarrow \delta_r(u_i, L(s, a, s'))$
13: Store transition $\langle \langle s, u_i \rangle, a, \langle s', u_j \rangle, r_j \rangle$ in $\mathcal{D}$
14: **end for**
15: Sample random batch of $N_b \cdot |U|$ transitions $\langle \langle s, u_i \rangle, a, \langle s', u_j \rangle, r_j \rangle$ from $\mathcal{D}$
16: Set $y_j = \begin{cases} r_j & \text{if } u_j \in F \text{ or } s' \text{ terminal} \\ r_j + \gamma \max\limits_{a'} \bar{Q}_{\bar{\theta}}(\langle s', u_j \rangle, a') & \text{otherwise.} \end{cases}$
17: Perform gradient descent step on $(y_j - Q_\theta(\langle s, u_i \rangle, a_j))^2$
18: Every C steps update target network $\bar{\theta} \leftarrow \theta$
19: Compute $u' \leftarrow \delta_u(u, L(s, a, s'))$
20: Set $s \leftarrow s'$
21: Set $u \leftarrow u'$
22: **end while**
23: **until** end

The algorithm begins with the initialisation of the replay memory. The capacity of the buffer is set to the default DQN value multiplied by the number of RM states (line 1). Next, both the Q-network and target network are initialised with random weights (lines 2-3). The training process is then started (lines 4-23). At the beginning of each episode, the active machine state and environment state are initialised (lines 5-6). The agent interacts with the environment until the end of the episode (lines 7-22). At each time step, the agent selects an action before executing it in the environment and observing the next state (lines 8-9). This information is then used to generate counterfactual experiences, each of which is stored in the replay memory (lines 10-14). A batch of transitions is then randomly sampled from the replay memory before being used to perform a gradient descent step. The number of transitions in the batch is equal to the default DQN value multiplied by the number of RM states (lines 15-17).

The parameters of the target network are then updated based on the defined update frequency (line 18), and finally, the active ground environment and machine state are updated (lines 19-21).

4.6.2 Exploiting Counterfactual Experiences

In this section, we analyse how the reward machine formulation modifies off-policy function approximation algorithms to increase sample efficiency. Specifically, we perform an in-depth analysis of the manner in which counterfactual experiences are integrated into the learning process. This analysis is carried out in the *LetterWorld* environment using the CFL task specification described in Section 4.2.2. We begin with a comparison between the sample efficiency of the CRM and DQN algorithms. This is followed by a discussion of the results, explaining how the RM formulation is able to increase sample efficiency. Finally, we discuss how these modifications can be improved before introducing *Counterfactual-DQN (CDQN)*, a version of the DQN algorithm which has been adapted for use with the CRA formulation.

Sample Efficiency Comparison

In this experiment, we evaluate the performance of the aforementioned algorithms when applied to the CFL task used as an illustrative example in Section 4.2.2. As discussed, the agent is required to observe a sequence of environment symbols which corresponds to a string in the context-free language $L = \{A^N BC^N : N \in \mathbb{N}\}$. As the value of N is varied, we track the number of samples required for each algorithm to obtain a solution. That is the number of samples required to solve the task when performing greedy action selection. The results of the experiment are illustrated in Figure 4.8.

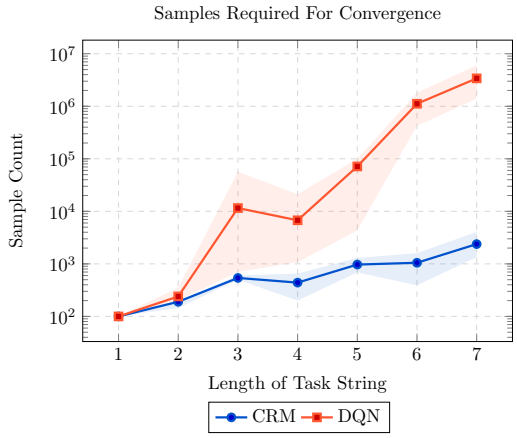


Figure 4.8: Number of samples required to obtain a solution as a function of task complexity (note: the figure makes use of a logarithmic scale). Parameters for performance estimation can be found in Appendix B.

As expected, the results illustrate that CRM is significantly more sample-efficient than DQN. The results illustrate that as the value of N is increased, both algorithms require

more environmental interactions before obtaining a solution. Specifically, the number of samples required by the DQN algorithm increases exponentially at a much faster rate than CRM. These results are in agreement with previous experiments. That is when using a sparse reward model, algorithms without counterfactual experiences perform significantly worse as the complexity of the task specification increases. This relationship is evident in the relative performance difference of CQL/CRM and Q-learning between the two experiments in Section 4.5.2.

Probability of Task Completion

The results of the sample efficiency comparison are an expected consequence of the relationship between the complexity of a task and the probability of it being completed by an agent. As the complexity of a task specification increases, the probability of it being completed through random action selection decreases. This relationship is illustrated in Figure 4.9. The probability of task completion is computed through the use of a Monte Carlo simulation using $10,000$ samples.

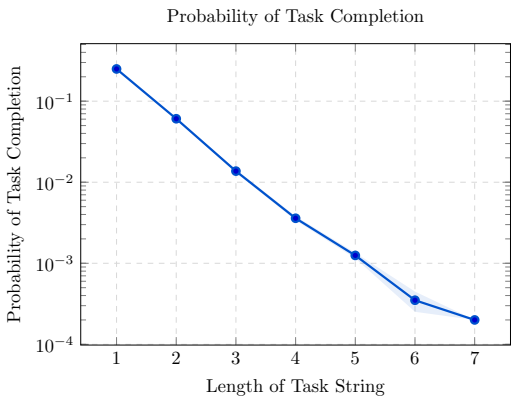


Figure 4.9: Probability of task completion for a random agent as a function of task complexity. Parameters for performance estimation can be found in Appendix B.

When using a sparse reward model, learning algorithms without counterfactual experiences are only able to obtain useful reward signals upon task completion. As a result, a decrease in the probability of task completion results in the algorithms requiring significantly more samples to obtain a solution. This explains why the sample efficiency of the DQN algorithm scales poorly with an increase in task complexity. The CRM algorithm is able to combat this issue through the introduction of counterfactual experience generation, which enables learning even in the event of task failure.

Access to Learning Signal

As discussed, the introduction of counterfactual experiences allows CRM agents to learn even if the task has failed. This mechanism increases sample efficiency in comparison to DQN which is only able to learn upon task completion. To illustrate the amount of learning signal accessible to each algorithm, we track the total reward that passes through each agent's replay memory throughout the process of training, normalised

by the number of training steps. We use the term *reward flux* when referring to this quantity. In Figure 4.10, we plot the reward flux of each algorithm as a function of task complexity.

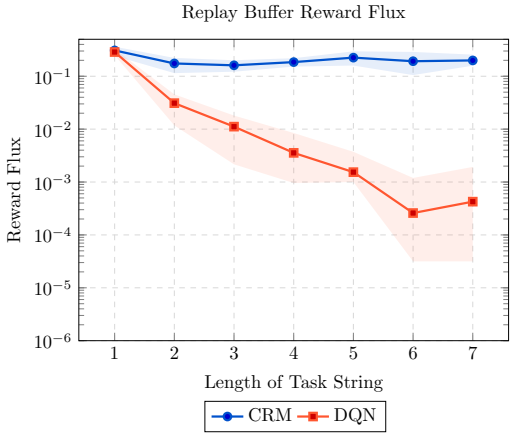


Figure 4.10: Replay buffer reward flux as a function of task complexity. Parameters for performance estimation can be found in Appendix B.

From the results, it is clear that as the complexity of the task specification increases, the number of useful transitions in the DQN replay memory decreases. As a result, the likelihood of sampling these transitions for gradient updates is lower, impeding the learning process. Notably, the introduction of counterfactual experiences has almost completely alleviated this problem. As the complexity of the task specification is increased, it is clear that the reward flux remains almost constant. This ensures that the agent always has access to meaningful transitions in the replay memory, which can be used when learning, resulting in significant performance gains.

Increasing Sample Efficiency

As the speed at which DQN learns is dependent on the rate of task completion, the algorithm prioritises collecting more samples from the environment with the most recent policy. The hope is that this improved policy will collect "higher-quality" transitions from the environment and, thereby, speed up the learning process. As a result, the DQN algorithm performs one step of gradient descent per environmental interaction. As counterfactual methods produce much higher quality transitions from the start of the training process, there is less incentive to interact with the environment as much as possible. Consequently, we suggest that counterfactual methods should focus on performing a much larger number of optimisation steps per environmental interaction. This idea is formalised in the *Counterfactual Deep Q-learning (CDQN)* algorithm in which we introduce a hyperparameter $\beta \in \mathbb{N}$, which scales the number of optimisation steps performed per environmental interaction. The CDQN algorithm is defined in Algorithm 4.

Algorithm 4 Counterfactual-DQN (CDQN)

Input: AAMDP $\mathcal{M} = \langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, \gamma^{\mathcal{A}}, R^{\mathcal{A}} \rangle$, optimisation hyperparameter β, batch size N_b, replay memory size N_m, target network update frequency C, upper bound on counter values Γ.

Initialise replay memory $\mathcal{D}$ with capacity $N_m \cdot |U|$
Initialise Q-network $Q_\theta : S^{\mathcal{A}} \times A^{\mathcal{A}} \mapsto \mathbb{R}$ with random weights θ
Initialise target network $\bar{Q}_{\bar{\theta}} : S^{\mathcal{A}} \times A^{\mathcal{A}} \mapsto \mathbb{R}$ with random weights $\bar{\theta}$
repeat
  Initialise $u \leftarrow u_0$
  Initialise $\boldsymbol{c} \leftarrow \mathbf{0}$
  Initialise s as initial environment state
  while $u \notin F$ and s non-terminal **do**
    Sample a from $\langle s, u, \boldsymbol{c} \rangle$ using policy derived from Q_θ (e.g. ϵ-greedy)
    Execute action a, observe s'
    for $u_i \in U$ **do**
      for $\boldsymbol{c}_i \in \{0, ..., \Gamma\}^k$ **do**
        $u_j \leftarrow \delta_u(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
        $\boldsymbol{c}_j \leftarrow \delta_c(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
        $r_j \leftarrow \delta_r(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
        Store transition $\langle \langle s, u_i, \boldsymbol{c}_i \rangle, a, \langle s', u_j, \boldsymbol{c}_j \rangle, r_j \rangle$ in $\mathcal{D}$
      end for
    end for
    for $n \in \{1, ..., \beta\}$ **do**
      Sample batch of $N_b \cdot |U|$ transitions $\langle \langle s, u_i, \boldsymbol{c}_i \rangle, a, \langle s', u_j, \boldsymbol{c}_j \rangle, r_j \rangle$ from $\mathcal{D}$
      Set $y_j = \begin{cases} r_j & \text{if } u_j \in F \text{ or } s' \text{ terminal} \\ r_j + \gamma \max_{a'} \bar{Q}_{\bar{\theta}}(\langle s', u_j, \boldsymbol{c}_j \rangle, a') & \text{otherwise.} \end{cases}$
      Perform gradient descent step on $(y_j - Q_\theta(\langle s, u_i, \boldsymbol{c}_i \rangle, a_j))^2$
      Every C steps update target network $\bar{\theta} \leftarrow \theta$
    end for
    Compute $u' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Compute $\boldsymbol{c}' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Set $s \leftarrow s'$
    Set $u \leftarrow u'$
    Set $\boldsymbol{c} \leftarrow \boldsymbol{c}'$
  end while
until end

4.6.3 Experimental Evaluation

In this section, we evaluate the performance of the aforementioned approaches when solving temporally extended high-dimensional control problems. For this evaluation, we make use of a more difficult version of the *PuckWorld* environment: a well-known reinforcement learning domain with a continuous state space and a discrete action space. As many descriptions of this environment exist throughout the literature, we focus on the specification provided in Haider *et al.* [2021]. We begin with a description

of the environment and task specification. Next, we discuss several important implementation details. Finally, we present the results of the performance evaluation.

Environment

Conventionally, the *PuckWorld* environment consists of a puck agent, which is required to track a moving target. The state space contains six continuous variables representing the position of the target as well as the position and velocity of the agent. Five actions are available to the agent for control. Specifically, it can modify its velocity by a fixed value in any of the cardinal directions. The task makes use of a dense reward model in which the agent receives a reward based on its distance from the target. The lower the distance, the higher the reward.

To facilitate more difficult task specifications, we introduce a number of modifications to the original *PuckWorld* environment. We begin by increasing the number of target positions from one to three, allowing each target to move throughout the environment as an independent entity. Next, we introduce an adversary which continually moves towards the agent. If the enemy is able to get sufficiently close to the agent, the task is immediately failed. Finally, we introduce a temporally extended task specification. The agent is required to *track* each of the targets in a specific sequence before the end of the episode. The agent is said to be *tracking* a target if it remains sufficiently close to it for a specific number of time steps. The state space of the problem consists of 12 continuous dimensions representing the position and velocity of the agent as well as the positions of the enemy and targets. We utilise the original action space of the problem. The extended environment is illustrated in Figure 4.11. The agent is illustrated as a green circle. Each of the targets T_1, T_2 and T_3 are illustrated as orange, yellow and blue circles, each with its own unique velocities, and finally, the enemy is illustrated in red. If the agent falls within the red region surrounding the enemy, the task is immediately failed.

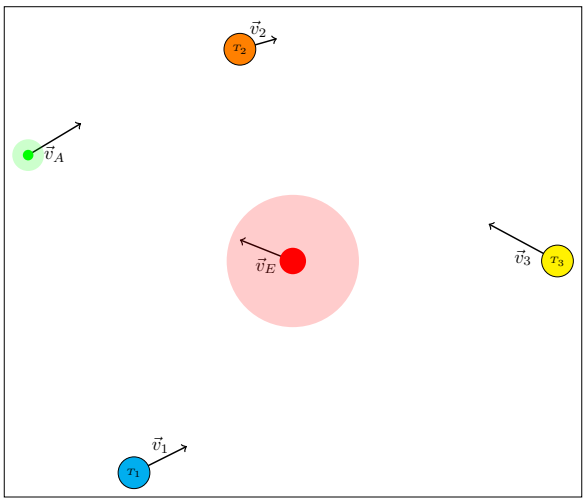


Figure 4.11: Illustration of the extended *PuckWorld* environment.

We make use of the following set of propositions $\mathcal{P} = \{T_1, T_2, T_3, E\}$ to model high-level events of interest in the environment. Each of the propositions T_i is satisfied when the agent is sufficiently close to the respective target. The proposition E is satisfied when the agent is within the red region surrounding the enemy. In this experiment, we make use of a complex temporally extended task specification. The agent is required to track each of the targets for N time steps while simultaneously avoiding the enemy. The targets must be tracked in the exact sequence T_1, T_2 and finally, T_3. The task is immediately failed if the agent tracks any target other than the active target (i.e. the agent gets too close to a non-active target) or enters the red region surrounding the enemy. Trajectories which satisfy this task specification can be represented using the context-sensitive formal language $L = \{T_1^N T_2^N T_3^N : N \in \mathbb{N}\}$.

Reward Model

We now describe the structure of the counting reward automaton used as a reward model for this problem. An illustration of a simplified representation of the automaton is given in Figure 4.12. As a result of the memory requirements associated with the task specification, three counter variables are used in the CRA formulation. These values represent the remaining number of tracking time steps associated with each target. Intuitively, the automaton consists of three states, one corresponding to each subtask in the specification. When in state u_0, the agent is required to track the target T_1 for the required number of time steps. For each time step in which the agent is sufficiently close to the target, the first counter variable is decremented by one (see transition 1). The automaton remains in this state until either the agent fails the task (see transition 3) or completes the tracking objective, causing the automaton to transition to state u_1 (see transition 2). The automaton states u_1 and u_2 function similarly; however, the agent tracks targets T_2 and T_3, respectively.

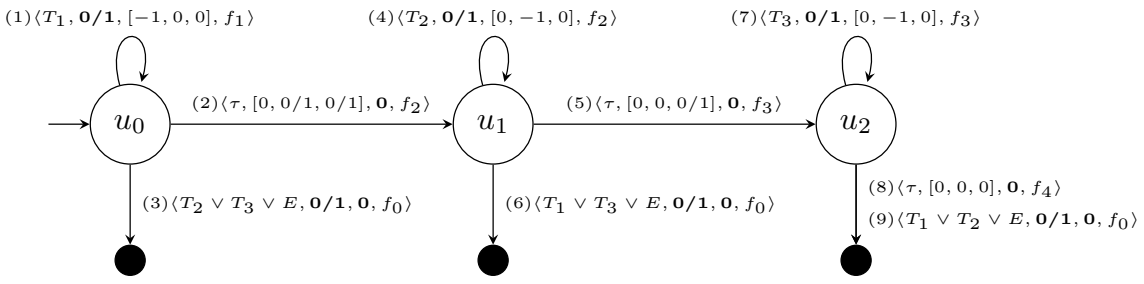


Figure 4.12: Simplified representation of the CRA used in the *PuckWorld* environment. The symbols $0/1$ and $\mathbf{0/1}$ are used to represent that the transition does not depend on the state of the counter variable. The symbol τ is used to represent a tautology.

In this experiment, we evaluate the performance of both dense and sparse reward models. When using the sparse reward model, the following definitions are assigned to the

reward functions of the CRA (see Figure 4.12):

$$
\begin{aligned}
f_0(s, a, s') &= 0 \\
f_1(s, a, s') &= 0 \\
f_2(s, a, s') &= 0 \\
f_3(s, a, s') &= 0 \\
f_4(s, a, s') &= 1
\end{aligned}
$$

for all $\langle s, a, s' \rangle \in S \times A \times S$. The dense reward model is more complex. Let $\psi_a(s)$ return the position vector of the agent given a state s. Similarly, let $\psi_t^i(s)$ return the position vector of target T_i given a state s. The reward functions used in the dense model are defined as:

$$
\begin{aligned}
f_0(s, a, s') &= -10 \\
f_1(s, a, s') &= \left\| \psi_a(s') - \psi_t^1(s') \right\|^2 \\
f_2(s, a, s') &= \left\| \psi_a(s') - \psi_t^2(s') \right\|^2 \\
f_3(s, a, s') &= \left\| \psi_a(s') - \psi_t^3(s') \right\|^2 \\
f_4(s, a, s') &= 100
\end{aligned}
$$

for all $\langle s, a, s' \rangle \in S \times A \times S$ where $\|\cdot\|^2$ is the ℓ^2-norm.

Implementation Details

The CDQN and CRM algorithms only require access to the replay memory of an algorithm in order to perform counterfactual experience generation. Consequently, counterfactual experiences can be incorporated into any off-policy learning algorithm without modification to the core learning implementation. To ensure our results are reliable, we make use of the Stable-Baselines3 DQN implementation [Raffin *et al.* 2021]. The CDQN and CRM algorithms are implemented by modifying the replay memory of this implementation. Specifically, we are only required to alter the manner in which transitions are inserted into the buffer. This result is important for two reasons. Firstly, it ensures our results are reliable as the core learning algorithm is identical across all three implementations. Secondly, it demonstrates that counterfactual experiences can easily be added to other off-policy learning algorithms. Algorithm hyperparameters are given in Appendix A.3 and Appendix A.4.

Results

The results for the sparse reward model are illustrated in Figure 4.13. It is clear that both the CDQN and CRM algorithms are significantly more sample-efficient than the DQN algorithm in this context. This is an expected consequence of the inability of the DQN algorithm to effectively extract learning signals from the sparse reward model. As the CDQN and CRM algorithms make use of counterfactual experience, they are able to alleviate this issue. Notably, the CDQN is significantly more sample-efficient than

the CRM algorithm. This is a consequence of the introduced β hyperparameter, allowing CDQN to fully exploit the set of counterfactual experiences available in the replay memory, speeding up learning. As a result, we can see CDQN quickly obtains a solution while CRM requires significantly more samples to obtain comparable performance.

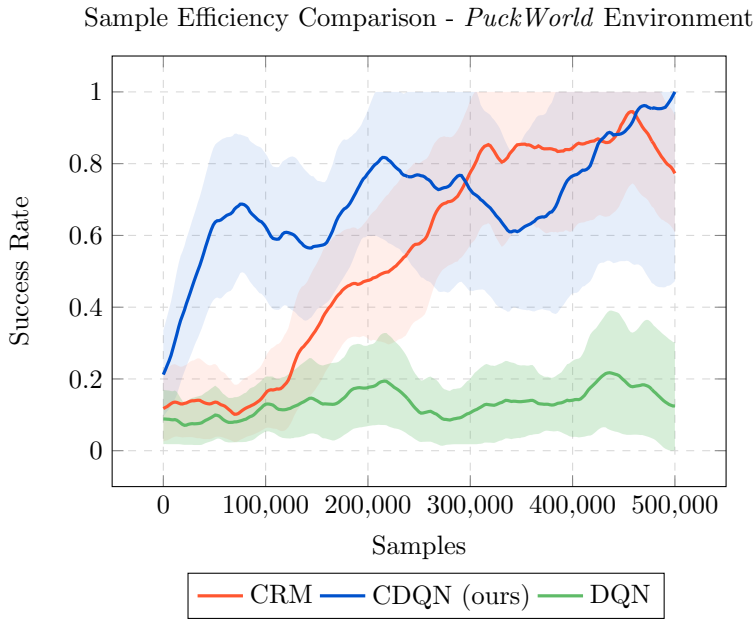


Figure 4.13: Sample efficiency comparison for the *PuckWorld* environment using sparse rewards. Model hyperparameters are detailed in Appendix A.3, while parameters for performance estimation can be found in Appendix B.

Figure 4.14 illustrates the results for the dense reward model. The introduction of a dense reward model illustrates several important properties of the formulations. Firstly, only relying on dense reward models is not a competitive strategy in comparison to counterfactual methods. This experiment makes use of a carefully defined dense reward model which incorporates domain knowledge to simplify the learning process. Regardless, the DQN algorithm is significantly outperformed by counterfactual methods in terms of sample efficiency. This is a consequence of the fact that approaches without counterfactual reasoning must solve each subtask in the sequence before being able to learn about the next. Secondly, the use of a dense reward model increases the amount of information available in the replay buffer. As a result, the CDQN algorithm significantly outperforms the CRM algorithm by focusing on the exploitation of this information. This is clear from the results, which show that the performance of CDQN after 100,000 training samples is comparable to that of CRM after 400,000 samples: an expected result as the CDQN algorithm β hyperparameter is set to 4 in this experiment.

4.7 Conclusion

In this chapter, we introduced counting reward automata: a novel abstract machine variant capable of overcoming the limitations of current state machine-based approaches.

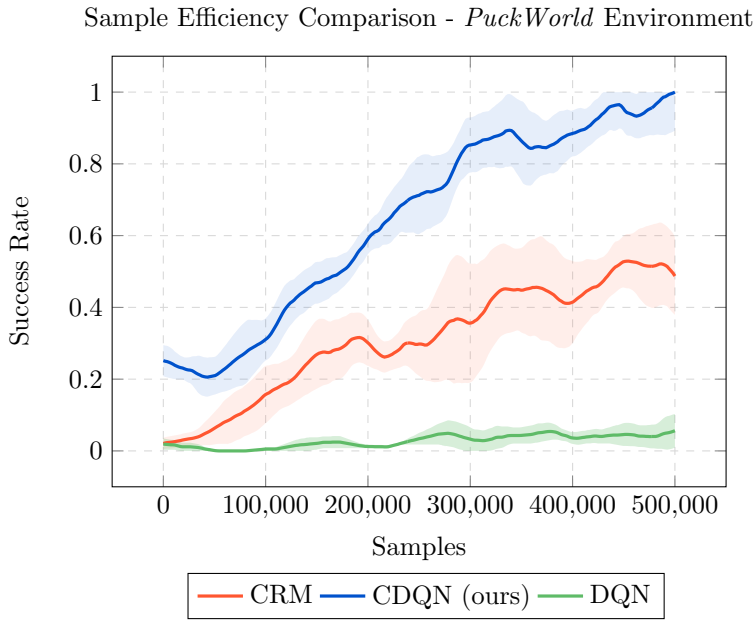


Figure 4.14: Sample efficiency comparison for the *PuckWorld* environment using dense rewards. Model hyperparameters are detailed in Appendix A.4, while parameters for performance estimation can be found in Appendix B.

This is a significant contribution as it greatly extends the set of tasks for which artificially intelligent agents can emulate the learning processes followed by generally intelligent agents. In addition, we thoroughly analysed the relationship between counting reward automata and reward machines. Several properties of the relationship, including sample efficiency and ease of specification, are formalised through proofs, which illustrate our approach's advantages. Finally, we conducted several experiments, evaluating our approach in both tabular and function approximation domains.

In conclusion, the CRA framework demonstrates promising capabilities in enhancing reinforcement learning algorithms, particularly in addressing the challenge of sample efficiency. However, as with any approach, avenues remain for further improvement and extension. In the next chapter, we delve into the augmentation of our proposed method through the integration of techniques from hierarchical reinforcement learning. By incorporating hierarchical structures into our framework, we aim to leverage the inherent hierarchical nature of many real-world tasks, thus potentially improving the efficiency, scalability, and robustness of the learning process. We explore how hierarchical representations can facilitate the acquisition of temporally abstracted knowledge to enable more efficient learning. Through this extension, we seek to advance the capabilities of our method and push the boundaries of reinforcement learning towards more complex and challenging domains.

Chapter 5

Hierarchical Reinforcement Learning for Counting Reward Automata

5.1 Introduction

In this chapter, we extend the CRA framework by integrating techniques from the field of hierarchical reinforcement learning, specifically *options*. HRL offers a structured approach to handling complex tasks with long time horizons. By incorporating options, we aim to enhance the scalability and efficiency of our method, enabling agents to learn more efficiently in environments with hierarchical structures. This integration opens avenues for addressing challenges in real-world tasks, leading to more autonomous and adaptive intelligent systems.

We present a learning algorithm tailored to harness the structure of a counting reward automaton to improve sample efficiency. This approach is based on hierarchical reinforcement learning and makes use of the options framework. The solution consists of two components. Firstly, a set of temporally extended actions or options which are learnt through counterfactual experience, allowing the agent to change machine configurations on command. Secondly, the agent learns to select options from this set in a manner that maximises the cumulative reward. These two components function together, allowing agents to carry out tasks in an environment.

The main contributions of the chapter are as follows:

- We present *Hierarchical Counting Reward Automata (HRCA)* – an approach based on hierarchical reinforcement learning which makes use of the options framework to increase sample efficiency.
- We show that CRA structure can be used to decompose tasks into subtasks, which can be modelled as options and solved in parallel.
- We prove that, in the context of episodic reinforcement learning, the set of options learnt by *Hierarchical Reinforcement Learning for Reward Machines (HRM)* is an alternative representation to that of HCRA.

- We prove that, in the context of episodic reinforcement learning, the HCRA and HRM algorithms are equally sample efficient.

We begin in Section 5.2 by describing our method. Specifically, we discuss the formulation of our approach as well as a number of important implementation details. Next, we carry out an experimental evaluation of our approach in Section 5.3. Specifically, we perform a number of experiments in tabular domains, each of increasing complexity. Finally, in Section 5.4, we describe how our approach can be extended to support function approximation. In this section, we discuss several implementation details before carrying out an experimental evaluation of the approach for both sparse and dense reward models.

5.2 Hierarchical Counting Reward Automata

In this section, we introduce the *Hierarchical Counting Reward Automata (HCRA)* algorithm, which is based on the *Hierarchical Reinforcement Learning for Reward Machines (HRM)* algorithm. The intuition behind this algorithm is simple. The overall task specification is decomposed into a number of subproblems using the counting reward automaton structure. These subproblems are then solved in parallel through the use of counterfactual experience generation. Finally, a solution to the task specification is obtained through the intelligent orchestration of subproblem solutions. From this description, it is clear that the HCRA algorithm consists of two important components. That is learning a set of subproblem solutions as well as an orchestration mechanism.

As discussed, the HCRA algorithm uses the structure of the CRA for subproblem discovery. Any solution to a task specification modelled by a CRA can be viewed as a sequence of machine configurations $\langle u, \boldsymbol{c} \rangle$. As a result, the HCRA algorithm treats moving between successive[1] machine configurations as subproblems. For a given machine configuration, the HCRA algorithm maintains a set of temporally extended actions. These actions are modelled through the use of options, allowing the automaton to transition to any of the next possible configurations. In general, the HCRA algorithm learns one option for each transition between CRA configurations. This set of transitions between machine configurations is given by:

$$\mathcal{T} = \{\langle u, \boldsymbol{c}, \delta_u(u, \sigma, Z(\boldsymbol{c})), \delta_c(u, \sigma, Z(\boldsymbol{c}))\rangle : u \in U, \boldsymbol{c} \in \{0, ..., \Gamma\}^k, \sigma \in 2^{\mathcal{P}}\}$$

[1] In this context, successive machine configurations are those which can be reached within one automaton transition.

As a result, the full set of options is defined as $\mathcal{O} = \{\langle \mathcal{I}_\tau, \pi_\tau, \beta_\tau \rangle : \tau \in \mathcal{T}\}$ in which the elements of each tuple are defined as:

$$\mathcal{I}_\tau = \{\langle s, u, \boldsymbol{c} \rangle : s \in S, u = \tau_1, \boldsymbol{c} = \tau_2\}$$
$$\beta_\tau(s, u', \boldsymbol{c}') = \begin{cases} 1 & \text{if } u' \neq \tau_1 \text{ or } \boldsymbol{c}' \neq \tau_2 \text{ or } s' \text{ is terminal} \\ 0 & \text{otherwise.} \end{cases}$$
$$\pi_\tau : S \times A \mapsto [0, 1]$$

where τ_i is used to denote the i^{th} element of the tuple τ. The initiation set $\mathcal{I}_\tau$ of the option labelled τ is defined as the set of all states in the AAMDP in which the machine configuration is equal to $\langle \tau_1, \tau_2 \rangle$. This follows from the fact that the automaton must first be in a given configuration before being able to change out of that configuration. Let $\mathcal{O}(u, \boldsymbol{c}) \subseteq \mathcal{O}$ denote the set of options which can be initiated from the machine configuration $\langle u, \boldsymbol{c} \rangle$. The termination condition of the option τ is defined as the set of all AAMDP states in which the machine configuration is not equal to $\langle \tau_1, \tau_2 \rangle$. The option terminates immediately after the automaton changes configurations. Finally, as the option can only be executed given the automaton is in a specific configuration, the option policy π_τ can be defined as a mapping from ground environment states to actions.

When executing option $o_\tau \in \mathcal{O}$, the objective is to change the automaton configuration from $\langle \tau_1, \tau_2 \rangle$ to $\langle \tau_3, \tau_4 \rangle$. As a result, the policy π_τ is trained using the following reward function:

$$r_{\text{base}} = \delta_r(\tau_1, \sigma, Z(\tau_2))(s, a, s')$$
$$r_\tau(s, a, s') = \begin{cases} r_{\text{base}} + r^+ & \text{if } \langle u_t, \boldsymbol{c}_t \rangle \neq \langle \tau_1, \tau_2 \rangle, \langle u_t, \boldsymbol{c}_t \rangle = \langle \tau_3, \tau_4 \rangle \\ r_{\text{base}} + r^- & \text{if } \langle u_t, \boldsymbol{c}_t \rangle \neq \langle \tau_1, \tau_2 \rangle, \langle u_t, \boldsymbol{c}_t \rangle \neq \langle \tau_3, \tau_4 \rangle \\ r_{\text{base}} & \text{otherwise.} \end{cases}$$

where r^+, r^- are hyperparameters with $r^+ > r^-$ and $\sigma = L(s, a, s')$. The same reward is assigned to each transition by this reward function as in the underlying CRA; this reward is denoted by r_{base}. However, in the event that the automaton changes configurations, an additional reward of r^+ or r^- is added to the transition based on whether or not the configuration changed as desired. For example, if the transition $\langle s, a, s' \rangle$ occurs and the automaton changes configurations from $\langle \tau_1, \tau_2 \rangle$ to $\langle \tau_3, \tau_4 \rangle$, the agent is given an additional reward of r^+ as the desired configuration change took place. The additional reward would be equal to r^- for any other configuration change. As a result, each option policy is incentivised to transition between automaton configurations as desired while respecting the underlying reward definitions of the CRA.

A high-level policy is introduced as an orchestration mechanism, selecting which option to execute at each stage in the process. This policy solves a simplified control problem, executing options to move between machine configurations in a manner which maximises cumulative reward. Given the active ground environment state and machine configuration, this policy determines which option is executed. The set of options which may be executed is determined based on the initiation sets of the individual options.

This high-level policy is trained using the approach defined in Sutton *et al.* [1999b] for training *policies over options*.

The primary benefit of the HCRA algorithm lies in its capacity to simultaneously learn policies for all options in $\mathcal{O}$. This is achieved by incorporating counterfactual experience generation into off-policy reinforcement learning. Suppose an agent in ground state s takes action a, resulting in a subsequent state of s'. If the machine configuration was $\langle u, \boldsymbol{c} \rangle$ before the transition and the option $\tau = \langle u, \boldsymbol{c}, u', \boldsymbol{c}' \rangle$ was being executed, then the reward is equal to $r_\tau(s, a, s')$. This value can then be used to update the option policy π_τ. However, the automaton structure allows us to compute reward values for any prior machine configuration and active option. Through this process of counterfactual reasoning, we are able to make use of the information within a single environmental interaction to update all of the option policies. This property of the HCRA algorithm allows us to simultaneously learn option policies.

A pseudocode implementation of the HCRA algorithm is provided in Algorithm 5. We make use of tabular Q-learning as the off-policy algorithm. However, the approach is compatible with any other off-policy reinforcement learning algorithm. The algorithm begins by defining the set of possible transitions between machine configurations (line 1). The set is defined as described above. Next, both the high-level and option Q-value functions are initialised to zero for all state-action pairs (lines 2-3). The training process is then carried out (lines 4-43). At the beginning of each episode, the counter machine configuration is initialised, and the initial state is sampled from the environment (lines 5-7). The agent then interacts with the environment until the end of the episode (lines 8-42). At each time step, the agent assesses whether a new option must be chosen (line 9) and, if needed, samples one based on the high-level Q-value function $\bar{Q}$ (lines 10-11). The agent then executes this option until termination. An action is then selected using the active option and executed in the environment (lines 13-14). The resulting experience is then used to compute the following machine configuration and reward (lines 15-17). Counterfactual experience generation is then performed, updating each of the option policies based on the most recent environmental interaction (lines 18-29). If the active option terminates (line 31), it is updated based on the approach for training policies over options defined in Sutton *et al.* [1999b] (lines 32-36) and the active option is set to indicate that a new option should be selected (line 37). Finally, the active ground environment state and machine configuration are updated in addition to the active option reward and step count (lines 39-41).

The main advantage of the HCRA algorithm is its ability to quickly learn good policies for AAMDPs compared to the CQL algorithm. This is achieved by learning a set of options which allow the machine to move between configurations on demand – simplifying the control problem for the high-level policy. As discussed, this advantage is a consequence of the algorithm's capability to simultaneously learn policies for all options in parallel. A notable drawback of the formulation is that even in the case of tabular reinforcement learning, it may produce sub-optimal solutions. This limitation arises as policies over options are not necessarily optimal. This is a consequence of the fact that the options are incentivised to change machine configurations as quickly as possible without any consideration for overall performance after the transition occurs.

Algorithm 5 Hierarchical Counting Reward Automata (HCRA)

Input: AAMDP $\mathcal{M} = \langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, \gamma^{\mathcal{A}}, R^{\mathcal{A}} \rangle$, learning rate α, upper bound on counter values Γ

Define a set of transitions between machine configurations $\mathcal{T}$
Initialise high-level $\bar{Q}(s, \tau) \leftarrow 0 \ \ \forall \langle s, \tau \rangle \in S \times \mathcal{T}$
For all options $\tau \in \mathcal{T}$, initialise option $Q_\tau(s, a) \leftarrow 0 \ \ \forall \langle s, a \rangle \in S \times A$
repeat
  Initialise $u \leftarrow u_0$
  Initialise $c \leftarrow \mathbf{0}$
  Initialise s as initial environment state
  while $u \notin F$ and s non-terminal **do**
    if τ_t is undefined **then**
      Sample option $\tau_t \in \mathcal{O}(u, \boldsymbol{c})$ using policy derived from $\bar{Q}$ (e.g. ϵ-greedy)
      Set $s_t \leftarrow s, r_t \leftarrow 0, t \leftarrow 0$
    end if
    Sample a from s using policy derived from Q_{τ_t} (e.g. ϵ-greedy)
    Execute action a, observe s'
    Compute $r \leftarrow \delta_r(u, L(s, a, s'), Z(\boldsymbol{c}))(s, a, s')$
    Compute $u' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Compute $\boldsymbol{c}' \leftarrow \delta_c(u, L(s, a, s'), Z(\boldsymbol{c}))$
    for $u_i \in U$ **do**
      for $\boldsymbol{c}_i \in \{0, ..., \Gamma\}^k$ **do**
        for $\tau \in \mathcal{O}(u_i, \boldsymbol{c}_i)$ **do**
          $u_j \leftarrow \delta_u(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
          $\boldsymbol{c}_j \leftarrow \delta_c(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
          if $\langle u_j, \boldsymbol{c}_j \rangle \neq \langle u_i, \boldsymbol{c}_i \rangle$ or s terminal **then**
            $Q_\tau(s, a) \overset{\alpha}{\leftarrow} r_\tau(s, a, s')$
          else
            $Q_\tau(s, a) \overset{\alpha}{\leftarrow} r_\tau(s, a, s') + \gamma \max_{a'} Q_\tau(s', a')$
          end if
        end for
      end for
    end for
    if s' is terminal or $\langle u, \boldsymbol{c} \rangle \neq \langle u', \boldsymbol{c}' \rangle$ **then**
      if s' is terminal or $u' \in F$ **then**
        $\bar{Q}(s_t, \tau_t) \overset{\alpha}{\leftarrow} r_t + \gamma^t r$
      else
        $\bar{Q}(s_t, \tau_t) \overset{\alpha}{\leftarrow} r_t + \gamma^t r + \gamma^{t+1} \max_{\tau' \in \mathcal{O}(u', \boldsymbol{c}')} \bar{Q}(s', \tau')$
      end if
      Set τ_t to undefined
    end if
    Set $s \leftarrow s', u \leftarrow u', \boldsymbol{c} \leftarrow \boldsymbol{c}'$
    Set $r_t \leftarrow r_t + \gamma^t r$
    Set $t \leftarrow t + 1$
  end while
until end

5.2.1 Episodic Reinforcement Learning

In this section, we compare the HCRA and HRM algorithms in episodic reinforcement learning. Specifically, we focus on the set of options learnt in either approach and the information used for training. We focus particularly on these aspects of the algorithms as they have significant implications for sample efficiency.

The HCRA algorithm learns a single option for every possible transition between machine configurations $\langle u, \boldsymbol{c} \rangle$. Similarly, the HRM algorithm learns an option for every possible transition between machine configurations $\langle u \rangle$. For each transition between machine configurations in a CRA, a corresponding transition exists in the emulating reward machine. It follows that the HCRA and HRM algorithms make use of different representations of the same set of options. This result is formalised in Theorem 9.

Theorem 9. *In the context of episodic reinforcement learning, the set of options learnt by HRM is an alternative representation to that of HCRA.*

Proof. Given a CRA $\mathcal{A}_\mathcal{C} = \langle U^c, F^c, u_0^c, \delta_u^c, \delta_c^c, \delta_r^c \rangle$ and an emulating reward machine $\mathcal{A}_\mathcal{R} = \langle U^r, u_0^r, F^r, \delta_u^r, \delta_r^r \rangle$.

The set of transitions defined for the CRA is equal to:

$$T^c = \{\langle \langle u, \boldsymbol{c} \rangle, \sigma, \langle u', \boldsymbol{c}' \rangle \rangle : \langle u, \boldsymbol{c}, \sigma \rangle \in U^c \times \{0, ..., \Gamma\}^k \times 2^\mathcal{P} \text{ and } u' = \delta_u^c(u, \sigma, Z(\boldsymbol{c})), \boldsymbol{c}' = \delta_c^c(u, \sigma, Z(\boldsymbol{c}))\}$$

Similarly, the set of transitions defined for the emulating RM is equal to:

$$T^r = \{\langle u, \sigma, u' \rangle : \langle u, \sigma \rangle \in U^r \times 2^\mathcal{P} \text{ and } u' = \delta_u^r(u, \sigma)\}$$

To prove the result, we show that there exists a mapping $\varphi : T^c \mapsto T^r$, which is a bijection, where φ is defined as follows:

$$\varphi(\langle \langle u, \boldsymbol{c} \rangle, \sigma, \langle u', \boldsymbol{c}' \rangle \rangle) = \langle \phi(\langle u, \boldsymbol{c} \rangle), \sigma, \phi(\langle u', \boldsymbol{c}' \rangle) \rangle$$

where ϕ is defined as in Theorem 5.

(injective) Let $x, y \in T^c$ be defined as:

$$x = \langle \langle u_1, \boldsymbol{c_1} \rangle, \sigma_1, \langle u_1', \boldsymbol{c_1}' \rangle \rangle$$
$$y = \langle \langle u_2, \boldsymbol{c_2} \rangle, \sigma_2, \langle u_2', \boldsymbol{c_2}' \rangle \rangle$$

Assume $\varphi(x) = \varphi(y)$, then:

$$\langle \phi(\langle u_1, \boldsymbol{c_1} \rangle), \sigma_1, \phi(\langle u_1', \boldsymbol{c_1}' \rangle) \rangle = \langle \phi(\langle u_2, \boldsymbol{c_2} \rangle), \sigma_2, \phi(\langle u_2', \boldsymbol{c_2}' \rangle) \rangle$$
$$\implies \langle \langle u_1, c_1^1, ..., c_1^k \rangle, \sigma_1, \langle u_1', c_1'^1, ..., c_1'^k \rangle \rangle = \langle \langle u_2, c_2^1, ..., c_2^k \rangle, \sigma_2, \langle u_2', c_2'^1, ..., c_2'^k \rangle \rangle$$

write the values $c_x^1, ..., c_x^k$ using a vector representation $\boldsymbol{c_x} = [c_x^1, ..., c_x^k]^T$

$$\implies \langle \langle u_1, \boldsymbol{c_1} \rangle, \sigma_1, \langle u_1', \boldsymbol{c_1}' \rangle \rangle = \langle \langle u_2, \boldsymbol{c_2} \rangle, \sigma_2, \langle u_2', \boldsymbol{c_2}' \rangle \rangle$$
$$\implies x = y$$

(surjective) Let $y = \langle\langle u, c^1, ..., c^k\rangle, \sigma, \langle u', c'^1, ..., c'^k\rangle\rangle \in T^r$.

Let $\boldsymbol{c}$ denote a vector representation of the values $c^1, ..., c^k$ in y, that is, $\boldsymbol{c} = [c^1, ..., c^k]^T$. Similarly, let $\boldsymbol{c}'$ denote a vector representation of the values $c'^1, ..., c'^k$ in y, that is, $\boldsymbol{c}' = [c'^1, ..., c'^k]^T$.

Then, for $x = \langle\langle u, \boldsymbol{c}\rangle, \sigma, \langle u', \boldsymbol{c}'\rangle\rangle \in T^c$ we have that:

$$\begin{aligned}\varphi(x) &= \langle\phi(\langle u, \boldsymbol{c}\rangle), \sigma, \phi(\langle u', \boldsymbol{c}'\rangle)\rangle \\ &= \langle\langle u, c^1, ..., c^k\rangle, \sigma, \langle u', c'^1, ..., c'^k\rangle\rangle \\ &= y\end{aligned}$$

□

The HCRA and HRM algorithms increase sample efficiency through the introduction of counterfactual experiences. This allows all of the options to be learnt simultaneously. However, in the context of episodic reinforcement learning, the sets of counterfactual experiences generated by either of the approaches contain equivalent amounts of information (Theorem 8). As a result, the HCRA and HRM algorithms utilise the same information when updating the underlying option policies during the learning process. Consequently, at any point in time, the corresponding options associated with either of the approaches have the same effect when executed in the environment. Thus, throughout the training process, the high-level controllers associated with each of the algorithms select from the same set of temporally extended actions (different representations used between the algorithms), which bring about the same effects in the environment. It follows that the algorithms are equally sample-efficient.

5.3 Experimental Evaluation

In this experiment, we evaluate the sample efficiency of several algorithms when applied to tasks of varying complexity in tabular domains. Specifically, we compare the CQL, CRM, HCRA, HRM and Q-learning algorithms in two tabular domains. In the first experiment, the agent is required to solve a context-free task specification in the *LetterWorld* environment. In the following experiment, a more difficult domain is introduced, requiring the agent to solve a context-sensitive task specification in the *OfficeWorld* environment.

5.3.1 *LetterEnv* Results

In this experiment, the sample efficiency of the aforementioned algorithms is evaluated in the *LetterWorld* environment illustrated in Figure 4.1. Specifically, each algorithm is required to solve the CFL task, which has been used as an illustrative example in Section 4.2.2. Once again, we make use of the environmental configuration described in Section 4.5.2. A sparse reward model is used in which the agent is given a reward of one upon task completion and zero otherwise. Finally, each agent is trained for 100,000 environmental interactions. The success rate, that is, the ability of the agent

to complete the task specification, is recorded throughout the training process. The results are illustrated in Figure 5.1.

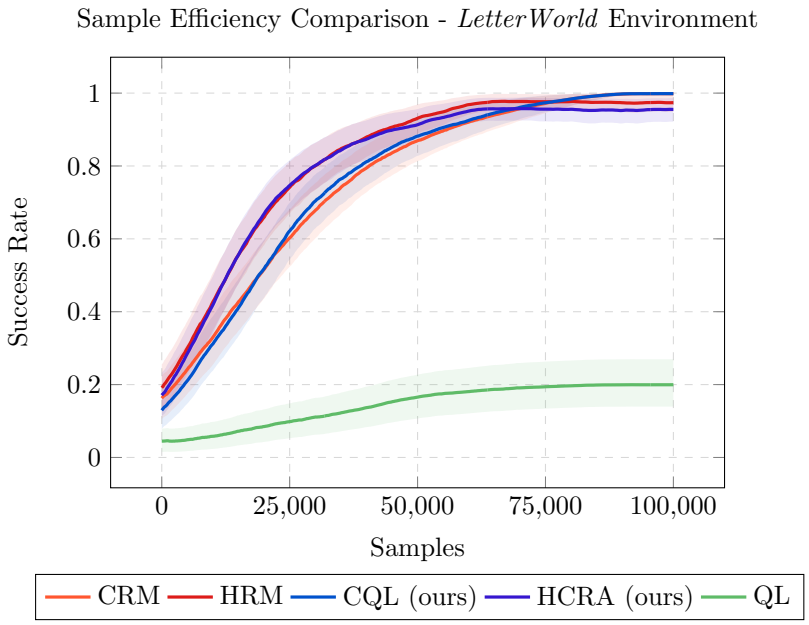


Figure 5.1: Sample efficiency comparison for the *LetterWorld* environment. Model hyperparameters are detailed in Appendix A.1, while parameters for performance estimation can be found in Appendix B.

The results demonstrate a measurable increase in sample efficiency between the conventional counterfactual methods and those based on hierarchical reinforcement learning. Specifically, we can see HCRA and HRM outperform the CDQN and CRM algorithms, all of which significantly outperform Q-learning. As expected, the HCRA and HRM algorithms exhibit nearly identical behaviour.

5.3.2 *OfficeWorld* Results

To evaluate our approach on a more challenging problem, we make use of a context-sensitive task in the *OfficeWorld* environment illustrated in Figure 4.4. Specifically, we make use of the task specification discussed in Section 4.5.2, and the environmental configuration described in Section 4.5.2 for performance evaluation. We compare the sample efficiency of the CQL, CRM, HCRA, HRM and Q-learning algorithms using a sparse reward model in which the agent is given a reward value of one upon task completion and zero otherwise. The results are illustrated in Figure 5.2.

We observe similar results to the previous experiment with HCRA and HRM measurably outperforming CQL and CRM in terms of sample efficiency. Notably, there is a slight increase in the relative performance difference between HCRA/HRM and the CQL/CRM algorithms in comparison to the previous experiment. This is likely a consequence of the increase in task complexity, enabling HCRA/HRM to simplify the control problem to the orchestration of options rather than primitive actions. Once again, the HCRA and

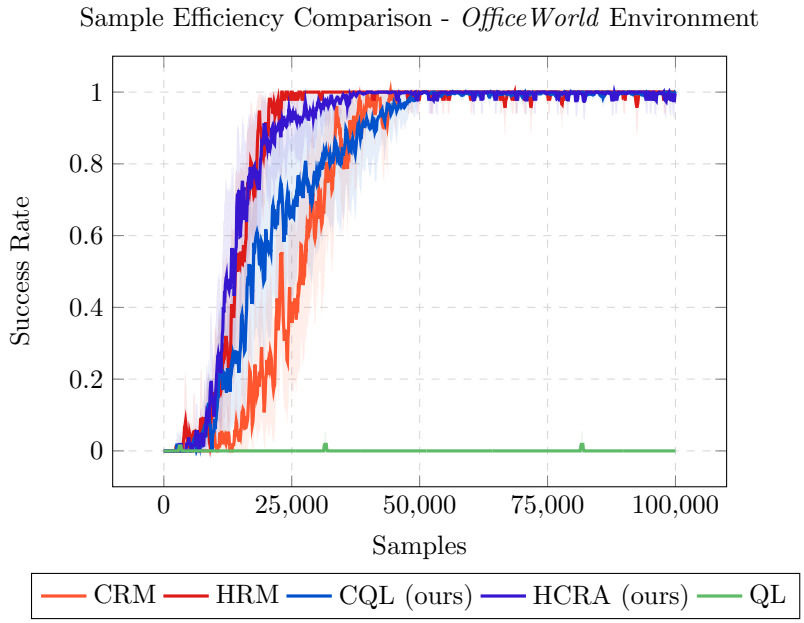


Figure 5.2: Sample efficiency comparison for the *OfficeWorld* environment. Model hyperparameters are detailed in Appendix A.2, while parameters for performance estimation can be found in Appendix B.

HRM algorithms exhibit nearly identical behaviour as expected. Finally, the Q-learning algorithm is shown to exhibit the worst performance, rarely completing the task as a result of its complexity.

5.4 Function Approximation Case

We begin this section by discussing how the HCRA algorithm can be extended for use with function approximation algorithms. This is followed by a discussion of several important implementation details. Next, we perform a detailed experimental analysis. Specifically, we evaluate the performance of several algorithms when solving temporally extended high-dimensional control problems. For this evaluation, we make use of the previously discussed modified *PuckWorld* environment. Finally, we discuss the results of the evaluation.

5.4.1 Extending the Formulation

Several extensions are required to facilitate the usage of function approximation algorithms within the HCRA formulation. These extensions are based on the HRM algorithm for reward machines. Firstly, both the high-level and option Q-value functions must be modelled through the use of suitable function approximators. In addition, appropriate off-policy learning algorithms which are compatible with function approximation must be used to facilitate the learning process. Any off-policy learning algorithm can be used within the HCRA formulation as it follows the same approach regardless. To

illustrate how HCRA can be applied in the context of function approximation, we use the Deep Q-learning algorithm as an example. The learning process is similar to that of the tabular case. However, all of the option's Q-value functions are modelled using the same function approximator. That is, two function approximators are used, one for the high-level controller and another for all of the options.

A pseudocode implementation of the HCRA algorithm for function approximation is provided in Algorithm 6. The HCRA algorithm begins by defining the set of possible transitions between machine configurations (line 1). The set is defined as described above. The high-level/option's replay memories are then initialised (line 2). Next, the high-level/option's Q-networks and target networks are initialised with random weights (lines 3-4). The training process is then carried out (lines 5-41). At the beginning of each episode, the counter machine configuration and environment state are initialised (lines 6-7). The agent then interacts with the environment until the end of the episode (lines 8-40). At each time step, the agent assesses whether a new option must be chosen (line 9) and, if needed, samples one based on the high-level Q-network $\bar{Q}_{\bar{\theta}}$ (lines 10-11). The agent then executes this option until termination. An action is selected using the active option and executed in the environment (lines 13-14). The resulting experience is then used to perform counterfactual experience generation (lines 15-20). The generated set of experiences is added to the option's replay memory (line 19). The option's Q-network Q_θ is then updated for a total of β optimisation steps (lines 21-26). If the active option terminates, the associated transition is stored in the high-level replay memory (line 28), and a flag value is assigned to the active option, indicating that a new option should be selected (line 29). The high-level Q-network $\bar{Q}_{\bar{\theta}}$ is then updated for a total of β optimisation steps (lines 31-36). Finally, the active ground environment state and machine configuration are updated in addition to the active option reward and step count (lines 37-39).

5.4.2 Experimental Evaluation

In this section, we evaluate the performance of the aforementioned approaches when solving temporally extended high-dimensional control problems. For this evaluation, we make use of the extended version of the *PuckWorld* environment described in Section 4.6.3. Each agent is required to solve the context-sensitive task specification described in Section 4.6.3. As a result, we make use of the previously defined CRA for the task specification. This automaton supports sparse and dense reward models, both of which are evaluated in this experiment. We begin by analysing the results associated with the sparse reward model, illustrated in Figure 5.3. From the results, it is clear that both HCRA and CDQN can quickly learn good solutions to the problem. However, the HCRA model continues to steadily improve throughout training, eventually obtaining the best performance out of any of the evaluated methods. As HCRA is based on the CDQN algorithm, it significantly outperforms HRM in terms of sample efficiency. This is a consequence of the fact that CDQN incorporates a number of improvements to the CRM optimisation procedure on which HRM is based. In general, the hierarchical methods are shown to outperform the unmodified counterfactual methods.

Algorithm 6 HCRA For Function Approximation (using DQN)

Input: AAMDP $\mathcal{M} = \langle S^{\mathcal{A}}, A^{\mathcal{A}}, P^{\mathcal{A}}, \gamma^{\mathcal{A}}, R^{\mathcal{A}} \rangle$,
CDQN hyperparameters $\beta, C, N_m, N_b, \alpha, \Gamma$

Define a set of transitions between machine configurations $\mathcal{T}$
Initialise high-level/options replay memories $\mathcal{D}_{\mathcal{H}}/\mathcal{D}_{\mathcal{O}}$ with capacity $N_m \cdot |\mathcal{O}|$
Initialise high-level Q-network/target network $\bar{Q}_{\bar{\theta}}/\bar{Q}'_{\bar{\theta}'} : S \times \mathcal{T} \mapsto \mathbb{R}$ randomly
Initialise options Q-network/target network $Q_\theta/Q'_{\theta'} : S \times A \times \mathcal{T} \mapsto \mathbb{R}$ randomly
repeat
  Initialise $u \leftarrow u_0, \boldsymbol{c} \leftarrow \mathbf{0}$
  Initialise s as initial environment state
  while $u \notin F$ and s non-terminal **do**
    if τ_t is undefined **then**
      Sample option $\tau_t \in \mathcal{O}(u, \boldsymbol{c})$ using policy derived from $\bar{Q}_{\bar{\theta}}$ (e.g. ϵ-greedy)
      Set $s_t \leftarrow s, r_t \leftarrow 0, t \leftarrow 0$
    end if
    Sample a from $\langle s, \tau_t \rangle$ using policy derived from Q_θ (e.g. ϵ-greedy)
    Execute action a, observe s'
    for $u_i \in U, \boldsymbol{c}_i \in \{0, ..., \Gamma\}^k, \tau \in \mathcal{O}(u_i, \boldsymbol{c}_i)$ **do**
      $u_j \leftarrow \delta_u(u_i, L(s, a, s'), Z(\boldsymbol{c}_i)), \boldsymbol{c}_j \leftarrow \delta_c(u_i, L(s, a, s'), Z(\boldsymbol{c}_i))$
      $r_j \leftarrow r_\tau(s, a, s')$
      Store transition $\langle \langle s, u_i, \boldsymbol{c}_i \rangle, a, \langle s', u_j, \boldsymbol{c}_j \rangle, r_j, \tau \rangle$ in $\mathcal{D}_{\mathcal{O}}$
    end for
    for $n \in \{1, ..., \beta\}$ **do**
      Sample batch of $N_b \cdot |\mathcal{O}|$ transitions $\langle \langle s, u_i, \boldsymbol{c}_i \rangle, a, \langle s', u_j, \boldsymbol{c}_j \rangle, r_j, \tau \rangle$ from $\mathcal{D}_{\mathcal{O}}$.
      Set $y_j = \begin{cases} r_j & \text{if } \langle u_j, \boldsymbol{c}_j \rangle \neq \langle u_i, \boldsymbol{c}_i \rangle \text{ or } s' \text{ terminal} \\ r_j + \gamma \max\limits_{a'} Q'_{\theta'}(s', a', \tau) & \text{otherwise.} \end{cases}$
      Perform gradient descent step on $(y_j - Q_\theta(s, a, \tau))^2$
      Every C steps update target network $\theta' \leftarrow \theta$.
    end for
    if s' is terminal or $\langle u, \boldsymbol{c} \rangle \neq \langle u', \boldsymbol{c}' \rangle$ **then**
      Store transition $\langle s_t, \tau_t, s, r_t + \gamma^t r, u', t \rangle$ in $\mathcal{D}_{\mathcal{H}}$
      Set τ_t to undefined
    end if
    for $n \in \{1, ..., \beta\}$ **do**
      Sample batch of $N_b \cdot |\mathcal{O}|$ transitions $\langle s_i, \tau, s_j, r_j, u_j, t_j \rangle$ from $\mathcal{D}_{\mathcal{H}}$.
      Set $y_j = \begin{cases} r_j & \text{if } u_j \in F \text{ or } s' \text{ terminal} \\ r_j + \gamma^{t_j+1} \max\limits_{\tau' \in \mathcal{O}(\tau_1, \tau_2)} \bar{Q}'_{\bar{\theta}'}(s_j, \tau') & \text{otherwise.} \end{cases}$
      Perform gradient descent step on $(y_j - \bar{Q}_{\bar{\theta}}(s, \tau))^2$
      Every C steps update target network $\bar{\theta}' \leftarrow \bar{\theta}$.
    end for
    Compute $r \leftarrow \delta_r(u, L(s, a, s'), Z(\boldsymbol{c}))(s, a, s')$
    Compute $u' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c})), \boldsymbol{c}' \leftarrow \delta_u(u, L(s, a, s'), Z(\boldsymbol{c}))$
    Set $s \leftarrow s', u \leftarrow u', \boldsymbol{c} \leftarrow \boldsymbol{c}', r_t \leftarrow r_t + \gamma^t r, t \leftarrow t + 1$
  end while
until end

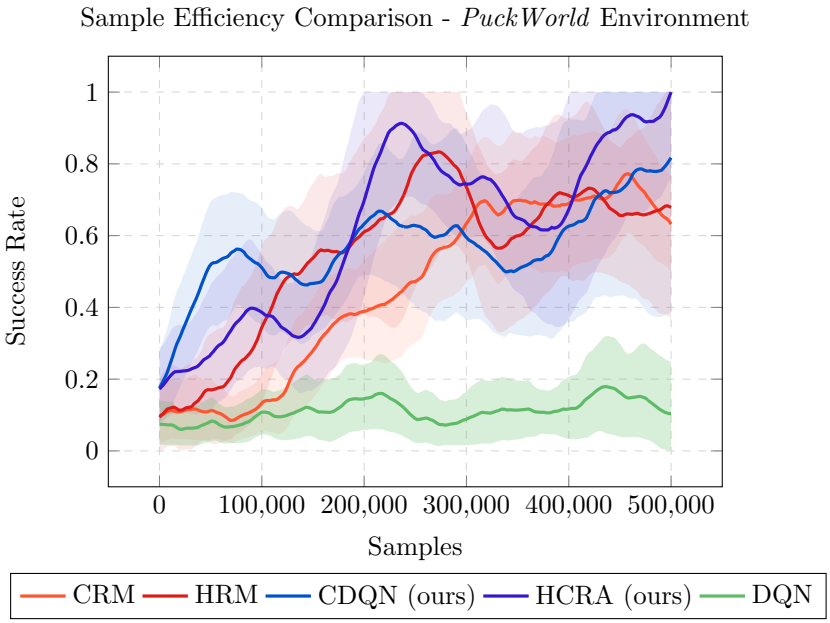


Figure 5.3: Sample efficiency comparison for the *PuckWorld* environment using sparse rewards. Model hyperparameters are detailed in Appendix A.3, while parameters for performance estimation can be found in Appendix B.

Next, we evaluate the results associated with the dense reward model, illustrated in Figure 5.4. It can be seen that the relative performance difference between the hierarchical and purely counterfactual approaches, for both HCRA/CDQN and HRM/CRM, has increased significantly in comparison to the results associated with a sparse reward model. As a result, it is clear that the introduction of a dense reward model benefits hierarchical methods significantly more than purely counterfactual methods in this experiment.

The hierarchical approach to learning and control is significantly more sample-efficient than purely counterfactual methods for some problems. These methods have been shown to consistently provide measurable increases in performance when applied to the problems evaluated in this research. As a result, this class of models should always be evaluated alongside purely counterfactual methods as they offer the potential for significant performance gains. However, it is important to note that these models may converge to sub-optimal solutions.

5.5 Conclusion

In this chapter, we have presented the hierarchical counting reward automata formulation. We began by describing the formulation and how it works. Next, we discussed the advantages of the HCRA algorithm before providing a pseudocode implementation of the approach for the tabular case. This was followed by a detailed comparison between the HCRA and HRM algorithms in the context of episodic reinforcement learning, showing that both algorithms are equally sample-efficient. An experimental analysis

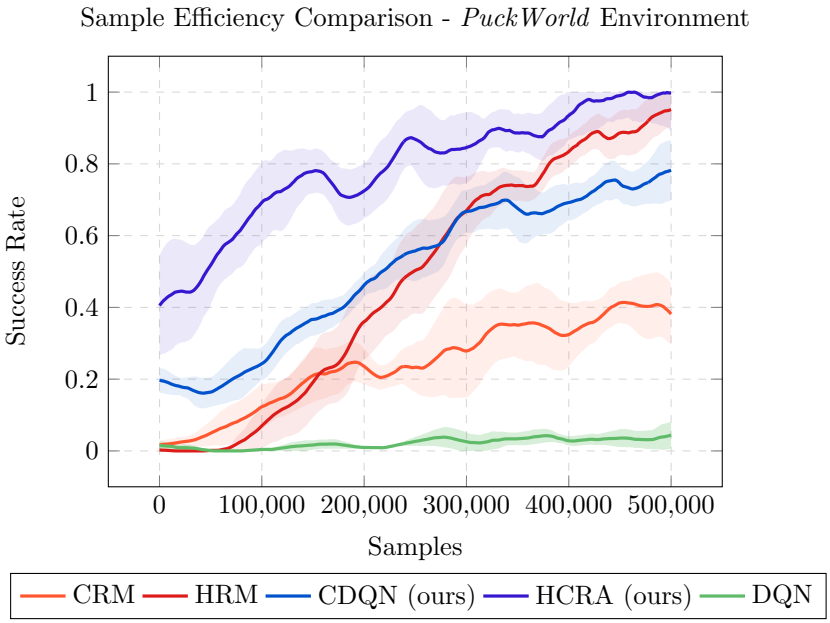


Figure 5.4: Sample efficiency comparison for the *PuckWorld* environment using dense rewards. Model hyperparameters are detailed in Appendix A.4, while parameters for performance estimation can be found in Appendix B.

was then conducted in several tabular domains. Finally, we discussed how the HCRA algorithm can be used with function approximation before evaluating the approach. Both sparse and dense reward models were evaluated in our experiments. In general, the results illustrated that the hierarchical approach to learning and control is significantly more sample-efficient than purely counterfactual methods for some problems. The hierarchical models were shown to consistently provide measurable increases in performance for the problems evaluated in this research.

In conclusion, the HCRA algorithm presented in this chapter showcases promising advancements in reinforcement learning, particularly in addressing challenges related to sample efficiency. However, to further enhance the applicability and performance of our approach, we turn our attention to the integration of reward-shaping techniques in the subsequent chapter. Reward shaping offers a mechanism to provide additional guidance to the learning agent by shaping the reward signal, thereby accelerating learning and potentially improving the final performance. By automatically designing auxiliary rewards, we aim to steer the learning process towards more desirable behaviours, expedite convergence, and speed up the discovery of effective policies. In the next chapter, we delve into the incorporation of reward shaping into our method, exploring its impact on learning dynamics and sample efficiency. Through this extension, we aspire to push the boundaries of our method and contribute to the advancement of reinforcement learning algorithms in handling complex and diverse real-world environments.

Chapter 6

Reward Shaping for Counting Reward Automata

6.1 Introduction

We now extend our methodology by integrating potential-based reward-shaping. Reward shaping provides a means to accelerate learning by modifying the original reward signal with additional incentives. We focus specifically on potential-based reward shaping, leveraging potential functions to design auxiliary rewards that encourage desirable state visitation patterns. This chapter explores the integration of potential-based reward shaping into the CRA framework to improve sample efficiency. Through this extension, we aim to advance the capabilities of our approach in handling complex real-world environments more effectively.

We present an approach to reward shaping designed to increase sample efficiency through the exploitation of CRA structure. This approach incorporates potential-based reward shaping, using the structure of a counting reward automaton to derive a shaping reward function. Reward shaping frequently entails introducing intermediate rewards that serve to indicate to agents their progress towards solving the task. The additional learning signal embedded within these rewards frequently increases sample efficiency.

The main contributions of this chapter are as follows:

- We formalise the limitations of *Automated Reward Shaping*, the current state-of-the-art approach for reward shaping with reward machines.
- We present *Reward Shaping for Counting Reward Automata* – a novel approach to potential-based reward shaping that utilises the structure of counting reward automata to derive potential functions.
- We show that *Reward Shaping for Counting Reward Automata* addresses each of the limitations associated with *Automated Reward Shaping*:
  - Incentivisation of task failure.
  - Unclear auxiliary reward signals.

- Rewarding agents while not progressing towards task completion.
- Compatibility is restricted to simple reward machines – a strict subset of all reward machines.

This chapter is structured as follows. We begin in Section 6.2 with a discussion of the *Automated Reward Shaping (RS)* formulation for reward machines. Specifically, we describe the formulation before examining its limitations. We then introduce *Reward Shaping for Counting Reward Automata (RS-CRA)* in Section 6.3. A novel approach to potential-based reward shaping which relies on counting reward automata. We begin this section by describing our formulation before providing an in-depth illustrative example, demonstrating how the approach can be used for reward shaping. Finally, in Section 6.4, we perform an experimental evaluation of the approaches covered in this chapter. Agents are evaluated in both tabular as well as function approximation domains on several task specifications.

6.2 Automated Reward Shaping

A technique known as automated reward shaping was introduced in Icarte *et al.* [2022] as an approach to perform reward shaping when using the reward machine formulation. The utility of the approach lies in its ability to automatically derive a shaping reward function from the structure of a reward machine, eliminating the need for domain knowledge or complex reward engineering. As the technique makes use of potential-based reward shaping, it has the desirable property that the set of optimal policies for the shaped reward are also optimal in the original MDP.

6.2.1 Formulation

Potential-based reward shaping is a technique used in reinforcement learning to increase sample efficiency by providing additional rewards to the agent during training. As discussed, the main idea behind the approach is to introduce auxiliary rewards which guide the agent towards completing the task more efficiently: a key component of any potential-based reward-shaping algorithm is the potential function. A function which returns the desirability of each state in a given MDP. As a result, the first step in the automated reward-shaping process is to define a potential function for the cross-product MDP, which can be used for reward-shaping. Specifically, the potential of MDP states is defined based on the desirability of the corresponding reward machine state. That is, all cross-product MDP states associated with a given RM state share the same potential value. This value is computed by considering an estimate of return when starting from a given reward machine state. Such a return can be calculated by treating the reward machine itself as an MDP.

In order to compute expected returns for each RM state, the automaton is represented as an MDP and value iteration is performed on the resulting decision process. The intuition behind the formation of this alternative representation is straightforward. A simple reward machine $\mathcal{A} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ can be represented as a MDP

$\mathcal{M} = \langle S, A, R, P, \gamma \rangle$, where

$$
\begin{aligned}
S &= U \cup F \\
A &= 2^{\mathcal{P}} \\
R(u, \sigma, u') &= \begin{cases} \delta_r(u, \sigma) & \text{if } u \in U \\ 0 & \text{otherwise.} \end{cases} \\
P(u'|u, \sigma) &= \begin{cases} 1 & \text{if } u \in F \text{ and } u' = u \\ 1 & \text{if } u \in U \text{ and } u' = \delta_u(u, \sigma) \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

and $\gamma \in [0, 1]$. Intuitively, each action in this MDP corresponds to a deterministic transition in the corresponding reward machine. The optimal value function of this MDP defines the expected discounted return for each reward machine state. With the alternative representation in place, the optimal value function can be computed using value iteration. Once the optimal value function has been computed, it can be used to define a potential function for the cross-product MDP, which can be used for reward shaping:

$$
\begin{aligned}
\Phi : S \times U \cup F &\mapsto \mathbb{R} \\
\Phi(s, u) &= -V^*(u)
\end{aligned}
$$

where V^* is the optimal value function of the MDP.

A pseudocode implementation is provided in Algorithm 7, showing how to perform value iteration using the MDP representation of a simple reward machine. The algorithm begins by initialising the value of all states to zero (lines 1-3). Next, the parameter ε is initialised to one. This parameter measures the magnitude of value function updates between iterations. The optimal value function is then calculated (lines 5-12). At the start of each iteration, the parameter ε is set to zero (line 6). Then, for each non-terminal state in the MDP, an updated value is calculated (line 8). The value of ε is then updated, and the new state value is stored (lines 9-10). Finally, the optimal value function is returned (line 13).

6.2.2 Illustrative Example

We now provide an illustrative example to demonstrate how automated reward shaping works. In this example, we consider the context-free task specification described in Section 4.2.2. This task is defined for the *LetterWorld* environment and requires agents to observe specific sequences of environment symbols. More specifically, each agent is required to observe a sequence of environment symbols which corresponds to a string in the context-free language $L = \{A^N B C^N : N \in \mathbb{N}\}$. To aid the explanation, we simplify the problem by restricting the value of N to three.

The reward machine required to solve this task is illustrated in Figure 6.1. This automaton implements a sparse reward model in which the agent is given a reward of one for completing the task and zero otherwise. Such a reward model was selected for this illustration as it is an ideal use case for reward shaping. The first step in the automated

Algorithm 7 Value Iteration for Automated Reward Shaping

Input: MDP $\mathcal{M} = \langle U, u_0, F, \delta_u, \delta_r \rangle$

for $u \in U \cup F$ **do**
  Initialise $V(u) \leftarrow 0$
end for
Initialise $\varepsilon \leftarrow 1$
while $\varepsilon > 0$ **do**
  $\varepsilon \leftarrow 0$
  for $u \in U$ **do**
    $V' \leftarrow \max\{\delta_r(u, \sigma) + \gamma V(\delta_u(u, \sigma)) \mid \forall \sigma \in 2^{\mathcal{P}}\}$
    $\varepsilon \leftarrow \max\{\varepsilon, |V(u) - V'|\}$
    $V(u) \leftarrow V'$
  end for
end while
return V

reward-shaping process is to define a potential function for the cross-product MDP. As discussed, the definition of this potential function is based on the expected return associated with being in a given reward machine state. Such returns can be calculated by treating the automaton as an MDP, using the alternative representation previously discussed. The optimal value function of this MDP defines the total expected discounted return for each RM state. This value function can be computed using value iteration. With the optimal value function in place, the potential function can be defined as above.

The optimal value[1] of each reward machine state in Figure 6.1 is illustrated in red. Using the previously defined potential function, we can compute the shaping reward associated with each reward machine transition. For example, consider the transition from the reward machine state u_0 into the terminal state, which is illustrated as a filled circle. The shaping reward associated with the transition is equal to:

$$\begin{aligned} r_s(\langle s, u \rangle, a, \langle s', u' \rangle) &= \gamma \Phi(\langle s', u' \rangle) - \Phi(\langle s, u \rangle) \\ &= \gamma(-V^*(u')) - (-V^*(u)) \\ &= 0.9(-0) - (-0.53) \\ &= 0.53 \end{aligned}$$

The full reward for the transition is obtained by adding the shaping reward to the base reward value from the reward machine:

$$\begin{aligned} r(\langle s, u \rangle, a, \langle s', u' \rangle) &= \delta_r(u, L(s, a, s')) + r_s(\langle s, u \rangle, a, \langle s', u' \rangle) \\ &= 0 + 0.53 \\ &= 0.53 \end{aligned}$$

[1] For a discount factor of $\gamma = 0.9$.

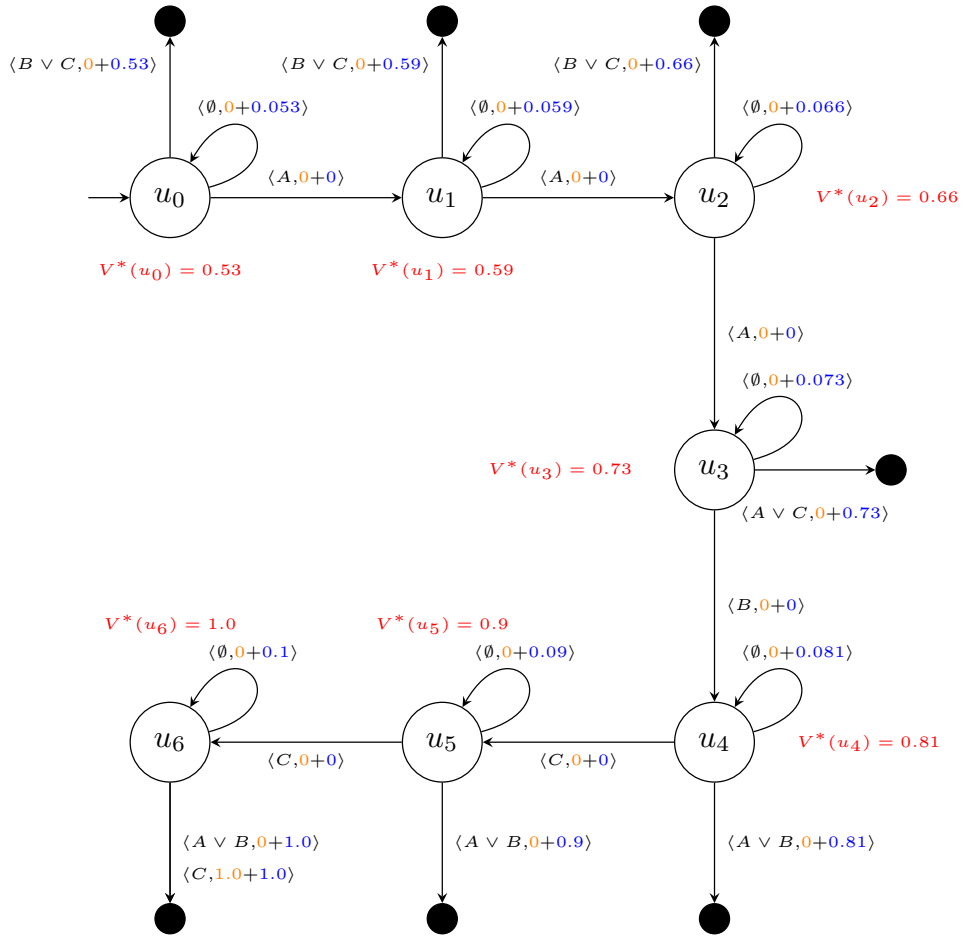


Figure 6.1: Illustration of the RM used to solve the CFL task. Terminal states are represented as filled circles. Automaton state values are shown in red, the base reward of the RM is shown in orange, and the shaping reward is given in blue. Values are rounded off to two decimal places for illustration; original calculations were performed at higher precision.

A similar process can be used to compute the reward for the remaining transitions. In the figure, the automaton's base reward is illustrated in orange, and the shaping reward is illustrated in blue.

6.2.3 Limitations

This example illustrates several limitations of the automated reward-shaping formulation. Firstly, the formulation is only applicable to simple reward machines, a strict subset of all reward machines. Consequently, automated reward shaping cannot be used for tasks which require information about ground environment states and actions when assigning rewards. This restriction is limiting as these properties are often used in robotics control problems where agents are penalised based on the magnitude of torques applied to joints.

Secondly, agents are incentivised to fail. As all of the terminal states in the reward machine have a value of zero, transitioning into any of these states will result in the agent being given a positive reward (if the reward machine has only non-negative rewards). Different potentials cannot be defined to differentiate between "desirable" and "undesirable" terminal states as potential-based reward shaping assumes that all terminal states have the same potential [Ng *et al.* 1999]. This property of automated reward shaping is particularly detrimental when using large automata with sparse reward models. Notably, such instances are ideal use cases for reward shaping as they are difficult to solve without the introduction of additional reward signals. However, when applied in such a setting, automated reward shaping produces reward functions with several undesirable properties. Consider the previous example. Not only are agents strongly incentivised to fail the task, but progression towards task completion remains unrewarded. Specifically, agents are incentivised to fail the task as large rewards are assigned to undesirable transitions into terminal states, which result in task failure. This is illustrated in the example where the agent is given a reward of 0.53 for transitioning from machine state u_0 into the terminal state. In addition, the lowest reward assigned to any transition is associated with progression between successive machine states, a necessary requirement for task completion. The transitions between each pair of successive machine states illustrate this property of the reward model. For example, the agent is given a reward of 0 for transitioning from machine state u_0 to u_1. This is a clear indication of progress towards task completion; however, it remains unrewarded in the modified reward model.

Finally, agents are rewarded for remaining in the same automaton state. This property of automated reward shaping is undesirable as it rewards the agent for behaviour which does not move it closer towards task completion. This is a consequence of the fact that the reward machine formulation defines automated reward shaping to make use of a discount factor $\gamma \in [0, 1)$ which is not a necessary requirement of potential-based reward shaping (the assumptions around the value of the discount factor are discussed in more detail in Section 6.3.1). This property of the reward model is illustrated in the example as a positive reward value is assigned to the self-loop transition associated with each automaton state.

Together, these limitations result in a formulation which is restrictive in terms of applications as well as produces reward models with undesirable performance characteristics. These challenges are clearly illustrated in the original results presented in Icarte *et al.* [2022], where the agent with reward shaping performs significantly worse than the counterfactual agent for large automata. However, automated reward shaping has been shown to increase sample efficiency in some applications that use small automata.

6.3 Reward Shaping for Counting Reward Automata

In this section, we introduce a novel approach to reward shaping known as *Reward Shaping for Counting Reward Automata (RS-CRA)*. This approach incorporates potential-based reward shaping, utilising the structure of a counting reward automaton to derive a shaping reward function. Such an approach to reward shaping facilitates the intro-

duction of additional learning signals without the need for domain knowledge or complex reward engineering. The design of reward shaping for counting reward automata intentionally addresses the limitations associated with automated reward shaping.

6.3.1 Formulation

The overall objective of the RS-CRA formulation is to compute a modified reward function which introduces auxiliary rewards that guide agents towards completing tasks more efficiently. As RS-CRA is a potential-based reward shaping technique, the modified reward model is derived from a potential function. The purpose of this potential function is to quantify the desirability of each state in the MDP, where higher potentials indicate increased desirability. As discussed, the potential function at the core of the RS-CRA formulation is derived from the structure of the task CRA.

When computing potential functions, it is important to note that potential-based reward shaping assumes that all terminal states have the same potential. As a result, different potentials cannot be defined to differentiate between "desirable" and "undesirable" terminal states. If this property is not accounted for when designing potential functions, the modified reward model may incentivise agents to fail the task. This is a key limitation of automated reward shaping. As the potentials of terminal states cannot be modified, the RS-CRA formulation follows an alternative approach to circumvent this issue. The CRA is modified by replacing undesirable transitions into terminal states with an alternative set of transitions. Each of the transitions in this set results in the automaton entering a non-terminal state from which task completion is impossible. Specifically, the automaton is transitioned into a *trap state*. A trap state is a non-terminal state which the automaton cannot transition out of once it has been entered, thereby preventing task completion. In order to perform this transformation, the RS-CRA algorithm makes use of a construction known as a *trap state CRA*. This is an alternative representation of the original CRA in which undesirable transitions[2] have been replaced with transitions into a single trap state. Once entered, the automaton remains in the trap state, emitting a constant reward value at each time step. The value of this reward is specified through the use of a penalty hyperparameter p. Specifically, p is chosen such that entering the trap state is, in all cases, less desirable than completing the task. The formal definition of a trap state CRA is provided in Definition 4.

Definition 4 (Trap State Counting Reward Automaton)**.** *Given a CRA* $\langle U, F, u_0, \delta_u, \delta_c, \delta_r \rangle$*, a penalty* p*, and a set of undesirable transitions between automaton configurations* W *where*

$$W = \{\langle \langle u_i, \boldsymbol{c}_i \rangle, \langle u_j, \boldsymbol{c}_j \rangle \rangle : u_i \in U \text{ and } u_j \in F\}$$

Define the function $\xi : U \times 2^{\mathcal{P}} \times \{0, ..., 1\}^k \mapsto U \times \{0, .., \Gamma\}^k$ *as*

$$\xi(u, \sigma, Z(\boldsymbol{c})) = \langle \langle u, \boldsymbol{c} \rangle, \langle \delta_u(u, \sigma, Z(\boldsymbol{c})), \delta_c(u, \sigma, Z(\boldsymbol{c})) \rangle \rangle$$

[2]Automaton transitions resulting in task failure.

which returns transitions between automaton configurations for a given machine state and input value. Then the corresponding trap state CRA is defined as $\langle U', F', u'_0, \delta'_u, \delta'_c, \delta'_r \rangle$ *where:*

$$
\begin{aligned}
U' &= U \cup \{u_T\} \\
F' &= F \\
u'_0 &= u_0 \\
\delta'_u(u, L(s,a,s'), Z(\boldsymbol{c})) &= \begin{cases} \delta_u(u, L(s,a,s'), Z(\boldsymbol{c})) & \textit{if } \xi(u, L(s,a,s'), Z(\boldsymbol{c})) \notin W \\ u_T & \textit{if } u = u_T \\ u_T & \textit{otherwise.} \end{cases} \\
\delta'_c(u, L(s,a,s'), Z(\boldsymbol{c})) &= \begin{cases} \delta_c(u, L(s,a,s'), Z(\boldsymbol{c})) & \textit{if } \xi(u, L(s,a,s'), Z(\boldsymbol{c})) \notin W \\ \mathbf{0} & \textit{if } u = u_T \\ \mathbf{0} & \textit{otherwise.} \end{cases} \\
\delta'_r(u, L(s,a,s'), Z(\boldsymbol{c})) &= \begin{cases} \delta_r(u, L(s,a,s'), Z(\boldsymbol{c})) & \textit{if } u \neq u_T \\ f_p & \textit{otherwise.} \end{cases}
\end{aligned}
$$

where f_p *is a function that returns* p *for all inputs and* u_T *is the trap state.*

For a given CRA, an intuitive process is followed to define the corresponding trap state CRA. Firstly, the set of transitions which result in task failure are added to a set W. Next, an alternative CRA is defined with an equivalent set of terminal and non-terminal states. However, a single non-terminal trap state u_T is introduced. The dynamics of the automaton are then modified such that any transition in W is replaced with a transition into the trap state. Only one transition exists for the trap state. That is a self-loop transition which does not modify the counter values. Finally, the reward function remains unchanged for all transitions other than the trap state transition. For this transition, a constant penalty function is returned.

With a trap state CRA in place, any notions of "desirable" or "undesirable" terminal states have been removed. This enables the definition of potential functions which are guaranteed not to incentivise the failure of task specifications. As discussed, the purpose of a potential function in reward shaping is to quantify the desirability of states in an MDP. Similarly to automated reward shaping, the RS-CRA formulation assigns potentials to AAMDP states based on the desirability of the associated CRA configurations. When quantifying desirability, the approach makes use of a graph representation of the automaton. This follows from the fact that any solution to a task modelled by a CRA can be viewed as a sequence of automaton configurations. Consequently, progress towards task completion can be approximated using the distance of configurations from the goal. Such distances are natural to compute when representing the CRA as a weighted directed graph. Each vertex in this graph represents an automaton configuration, while the associated transitions are modelled as edges. The weight of each edge is equal to the minimum value of the bounded reward function associated with the corresponding transition in the automaton. A formal definition for this graph representation is provided in Definition 5.

Definition 5 (Graph Representation of a CRA)**.** *Given a CRA* $\langle U, F, u_0, \delta_u, \delta_c, \delta_r \rangle$ *and an upper bound on counter values* Γ *derived from the maximum episode length, the graph representation of the automaton is defined as* $G = \langle V, E, w \rangle$*, where*

$$V = U \times \{0, ..., \Gamma\}^k$$
$$E = \{\langle v_i, v_j \rangle : v_i = \langle u_j, \boldsymbol{c}_i \rangle \in V; \exists\, \sigma \in 2^{\mathcal{P}}, v_j = \langle \delta_u(u_i, \sigma, Z(\boldsymbol{c}_i)), \delta_c(u_i, \sigma, Z(\boldsymbol{c}_i)) \rangle\}$$
$$w(v_i, v_j) = \min_{x \in X} R(x)$$

where R *is the reward function associated with the transition in the CRA and* $X = S \times A \times S$*.*

The vertices in this graph represent configurations $\langle u, \boldsymbol{c} \rangle$ of the counting reward automaton. Each edge corresponds to a transition between successive automaton configurations as defined by δ_u and δ_c. The weights of these edges are equal to the minimum values of the bounded reward functions returned by δ_r for the associated CRA transitions.

As discussed, the RS-CRA formulation makes use of the graph representation of a CRA to compute the desirability of automaton configurations. Desirability is quantified using an intuitive metric. That is, the maximum distance from each non-terminal vertex to a terminal vertex[3]. More precisely, the approach makes use of the length of the longest simple path[4] between the two vertices. This quantity has been selected as it provides an indication of the total cumulative reward which can be obtained from a given configuration, enabling the computation of potentials. The RS-CRA formulation follows the conventional approach to solving the longest simple path problem. That is, a transformation is applied to the original graph, allowing the shortest path problem to be solved. The required transformation is simple, consisting of only two elementary steps. Firstly, the signs of all of the weights in the graph are inverted. Secondly, the absolute value of the minimum weight in the resulting graph is added to each edge. This ensures that all of the weights in the graph are greater than or equal to zero. The shortest path problem is then solved in the resulting graph using Dijkstra's algorithm [Dijkstra 1959]. A pseudocode implementation of this algorithm is given in Algorithm 8.

Once the longest simple path problem has been solved for each vertex in the graph, the length of these paths can be computed. These values are used to define potentials for each automaton configuration. Specifically, the potential of an automaton configuration is defined based on the length of the path associated with the corresponding vertex. That is if the path length is ℓ, the configuration potential is defined as $-\ell$. With the set of configuration potentials in place, a full potential function for the AAMDP can be

[3] A terminal (non-terminal) vertex is a vertex representing a terminal (non-terminal) automaton configuration.

[4] A simple path is a path in a graph which does not have repeating vertices.

Algorithm 8 Dijkstra's Algorithm [Dijkstra 1959]

Input: Weighted Graph = $G = \langle V, E, w \rangle$

Initialise $s \leftarrow$ source vertex
Initialise distance mapping $d[s] \leftarrow 0$
Initialise $d[v] \leftarrow 0 \ \ \forall \, v \in V \setminus \{s\}$
Initialise sets $Q \leftarrow V, S \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
    $v \leftarrow \min_{v' \in Q} d[v']$
    $Q \leftarrow Q \setminus \{v\}$
    $S \leftarrow S \cup \{v\}$
    for neighbour u of v **do**
        if $u \in S$ **then**
            continue
        end if
        $x \leftarrow d[v] + w(v, u)$
        if $x < d[u]$ **then**
            $d[u] \leftarrow x$
        end if
    end for
end while
return d

defined as:

$$\Phi : S \times U \cup F \times \{0, ..., \Gamma\}^k \mapsto \mathbb{R}$$
$$\Phi(s, u, \boldsymbol{c}) = \omega \cdot \phi(u, \boldsymbol{c})$$

where $\phi(u, \boldsymbol{c})$ is the potential of the automaton configuration $\langle u, \boldsymbol{c} \rangle$ and $\omega \in \mathbb{R}$ is a hyperparameter used to control reward scaling.

Finally, we conclude by discussing the conditions associated with the discount factor γ. For the undiscounted case ($\gamma = 1$), the ground environment is required to satisfy the two standard regularity conditions for MDPs with finite state spaces [Sutton and Barto 2018]. Firstly, the ground environment state space is required to have a distinguished state known as an *absorbing* state such that the episode "stops" after a transition into such a state, with no further rewards thereafter. Secondly, it is assumed that the ground environment transition probabilities are *proper*. This means that upon the execution of any policy starting from any state, it will eventually transition into an absorbing state with a probability of 1. The situation is much simpler for the discounted case ($\gamma < 1$) as no regularity conditions are required to hold.

6.3.2 Illustrative Example

We now provide an illustrative example to demonstrate the operation of the RS-CRA formulation. Similarly to the previous section, we consider the example task described in Section 4.2.2 in which the value of N has been restricted to three. The CRA used to

model the reward for this problem is illustrated in Section 4.2.2. We assume that the ground environment satisfies the regularity conditions discussed in Section 6.3.1 and set $\gamma = 1$. Note that the RS-CRA formulation is not limited in that it requires these regularity conditions to be satisfied. As discussed in the previous section, in the event that these conditions are not satisfied, we would make use of a discount factor of less than one. In this example, we opt to use a discount factor of one for ease of explanation. The first step in the RS-CRA process is to replace undesirable transitions into terminal states with transitions into a single trap state. This is achieved by defining the trap state CRA. In this automaton, undesirable transitions into terminal states are replaced with transitions into a single trap state. For this example, we consider all transitions into a terminal state with a reward of zero to be undesirable.

The trap state CRA is then transformed into a graph representation. An illustration of this graph representation is shown in Figure 6.2. The illustrated graph structure resembles an emulating reward machine as additional information has been added to the diagram to aid explanation. Firstly, a label has been added to each vertex in the graph to show the corresponding automaton configuration. Secondly, associated with each edge in the graph is a tuple containing two elements. The first element in the tuple is the set of propositions required for the CRA to execute the transition[5]. The second element in the tuple is the minimum value of the bounded reward function associated with the corresponding CRA transition.

With the construction of the graph representation in place, we proceed to the computation of potential values for each automaton configuration. As discussed, we compute the maximum distance from each non-terminal vertex to a terminal vertex. These values are used to compute potentials. The process is illustrated in Figure 6.3. Three numbers are associated with each edge in the graph. The first number, in black, is the minimum value of the bounded reward function associated with transition in the original CRA. The second number, in blue, is the negation of the first number. Finally, the third number, in red, is the value of the second number after the addition of the absolute value of the minimum edge weight. This value is the final edge weight in the transformed graph. Each vertex in the graph is labelled with two numbers. The first number, in orange, is the length of the shortest path to a terminal vertex, computed using Dijkstra's algorithm. The second number, in purple, is the final potential value for the machine configuration associated with the vertex. This value is obtained by negating the first number.

Once the potential of each automaton configuration has been calculated, we can proceed to define the potential function. As discussed, the potential function introduces auxiliary rewards which guide the agent towards task completion. The resulting modified reward model consists of two components: a base reward from the CRA and a shaping reward value. A representation of the modified reward model produced by the RS-CRA formulation for this example is given in Figure 6.4. This illustration shows the reward associated with each of the possible transitions between CRA configurations.

[5]Transitions depend only on propositions as counter values are fixed in each vertex.

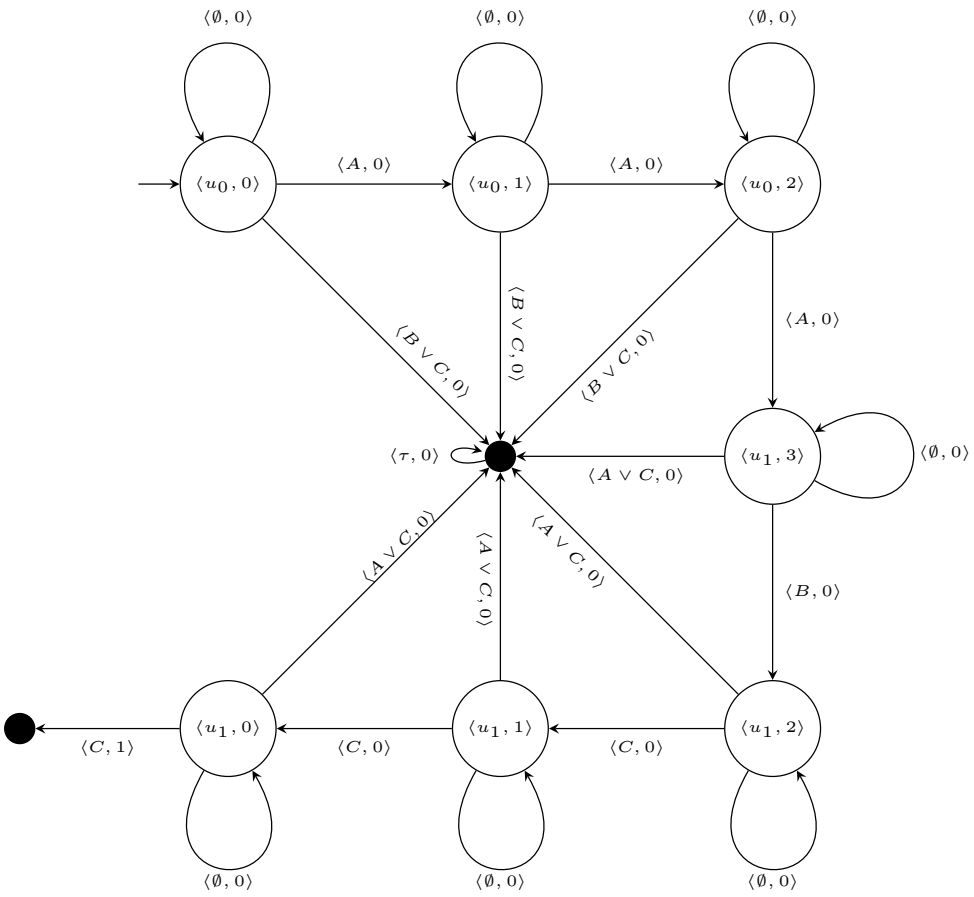


Figure 6.2: Graph representation of trap state CRA. Edge labels show propositions required for CRA transitions as well as the minimum value of the associated reward function. τ is used to represent tautology.

The reward associated with each transition is illustrated as a sum of two values. The first value, in black, is the base reward from the original CRA. The second value, in blue, is the shaping reward value. This quantity is calculated based on the change in potential between successive automaton configurations, as required by potential-based reward shaping. Finally, the potential of each automaton configuration is illustrated in red.

This example clearly illustrates how the RS-CRA formulation addresses the limitations of automated reward shaping. Firstly, the modified reward model does not incentivise failure of the task specification. This is achieved through the introduction of the trap state CRA. Secondly, the potentials of automaton configurations are computed based on the distance to terminal states. This ensures that the signals provided by auxiliary rewards clearly indicate how agents should progress towards task completion. Thirdly, agents only receive additional rewards for transitions which alter the automaton configuration. As a result, agents are only provided with additional rewards while progressing towards task completion. Finally, the RS-CRA formulation is compatible with the full CRA formulation as opposed to just CCRA. This enables the formulation to be applied

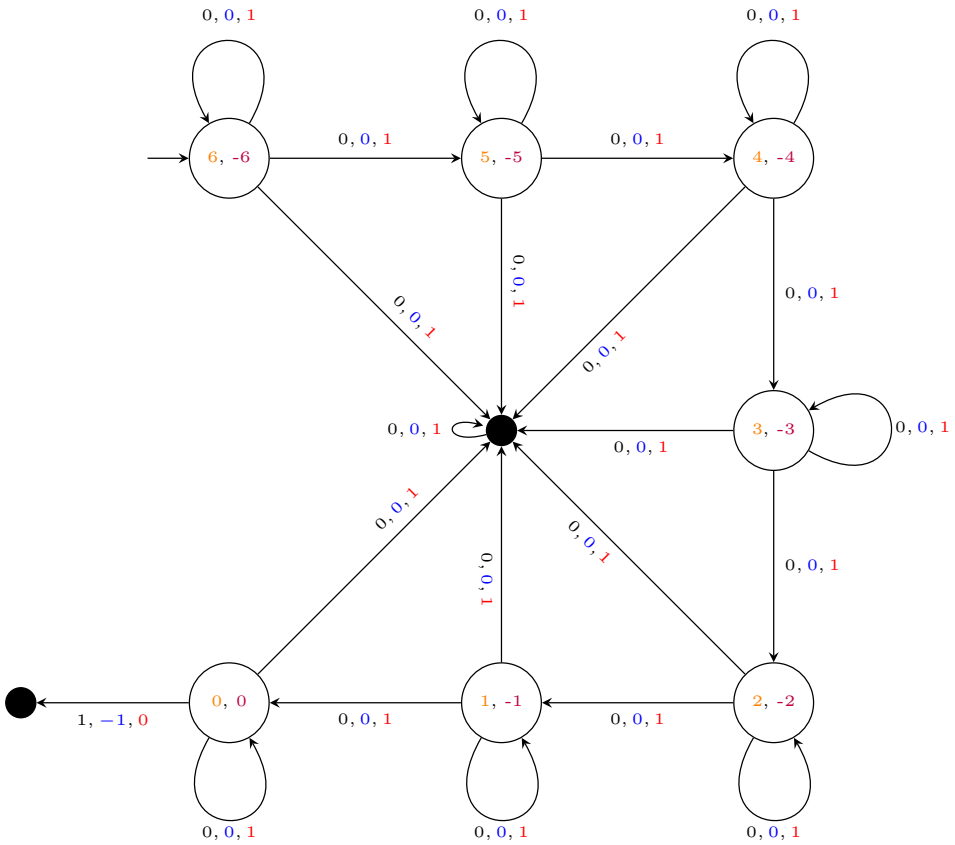


Figure 6.3: Illustration of the calculation of state potentials. For each edge, the original reward for the transition is shown in black. The negation of this value is in blue, and the shifted value is in red. For each vertex, the length of the longest simple path to a desirable terminal state is orange, and the final potential value is purple.

to a much larger set of problems than automated reward shaping. For example, RS-CRA can be used for tasks which require information about ground environment states and actions when assigning rewards, properties are often used in robotics control problems where agents are penalised based on the magnitude of torques applied to joints. Automated reward shaping cannot be applied in these settings.

6.4 Experimental Evaluation

In this section, we evaluate the sample efficiency of the aforementioned algorithms when applied to tasks of varying complexity in both tabular and function approximation domains. In the first experiment, the agent is required to solve a context-free task specification in the *LetterWorld* environment. Next, a more difficult domain is introduced, requiring the agent to solve a context-sensitive task specification in the *OfficeWorld* environment. Finally, agents are required to solve a complex temporally extended high-dimensional control problem in the extended *PuckWorld* environment.

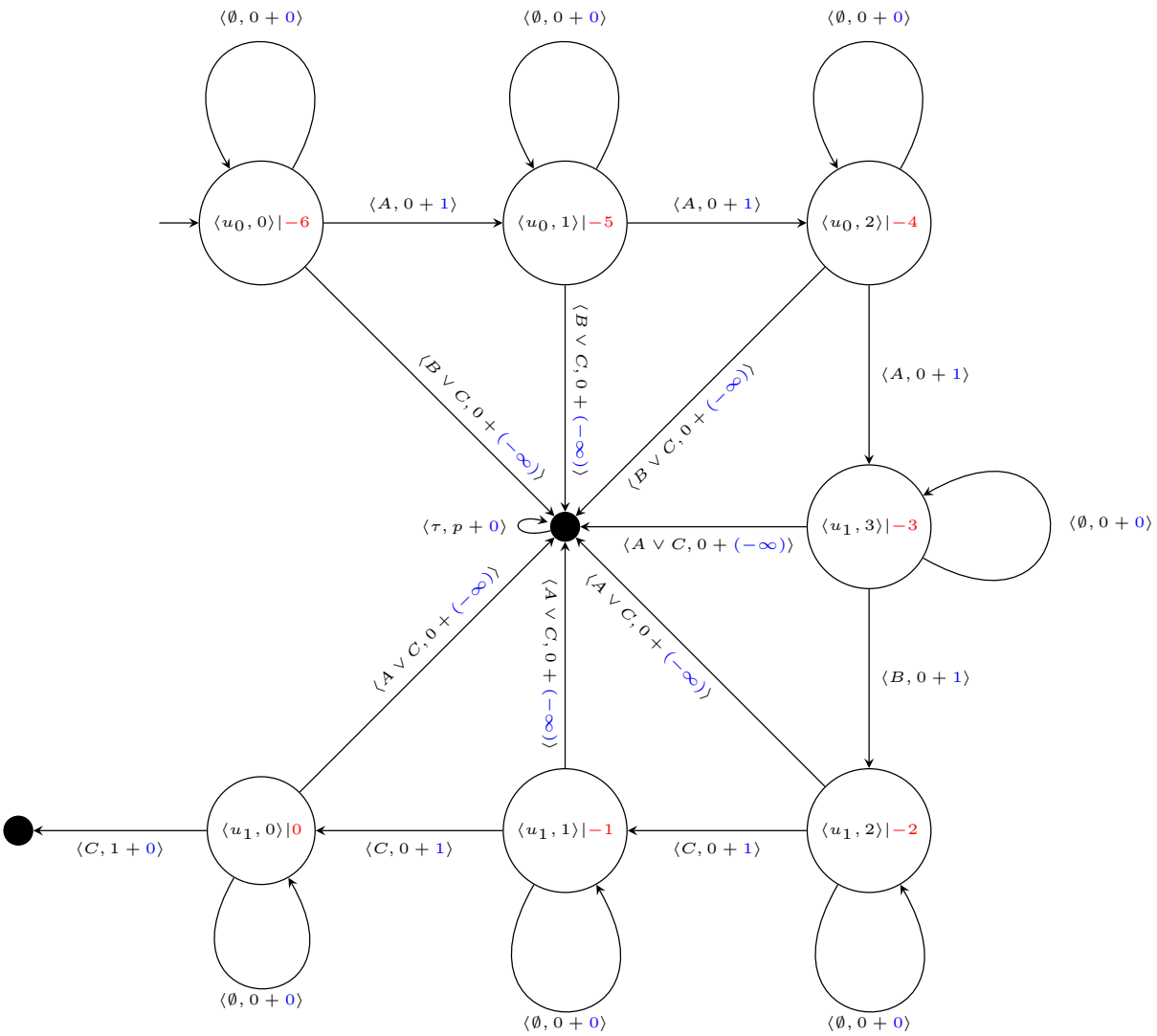


Figure 6.4: Illustration of the modified reward function produced by the RS-CRA formulation. The potential of each automaton configuration is illustrated in red. For each edge, the original reward of the CRA is shown in black, and the shaping reward is shown in blue. Calculations performed using $\gamma = 1, \alpha = 1$ and $p = -1$.

In this experiment, we evaluate the performance of both sparse and dense reward models.

6.4.1 Tabular Case

We now evaluate several algorithms when applied to tasks of varying complexity in tabular domains. Specifically, we compare the CQL, CRM and Q-learning algorithms to CQL+RS and CRM+RS. In this context, CRM+RS is used to denote CRM with automated reward shaping, and similarly, CQL+RS is used to denote CQL using a reward model defined by the RS-CRA formulation. In the first experiment, the agent is required to solve a context-free task specification in the *LetterWorld* environment. In

the following experiment, agents are evaluated using a more complex context-sensitive task specification in a more difficult domain.

LetterWorld

This experiment is performed in the *LetterWorld* environment illustrated in Figure 4.1. Each algorithm is required to solve the CFL task, which has been used as an illustrative example in Section 4.2.2. The environment is configured as discussed in Section 4.5.2. The task makes use of a sparse reward model in which the agent is given a reward of one upon task completion and zero otherwise. Finally, each agent is trained for 100,000 environmental interactions. The success rate, which is the ability of the agent to complete the task specification, is recorded throughout the training process. The results are illustrated in Figure 6.5.

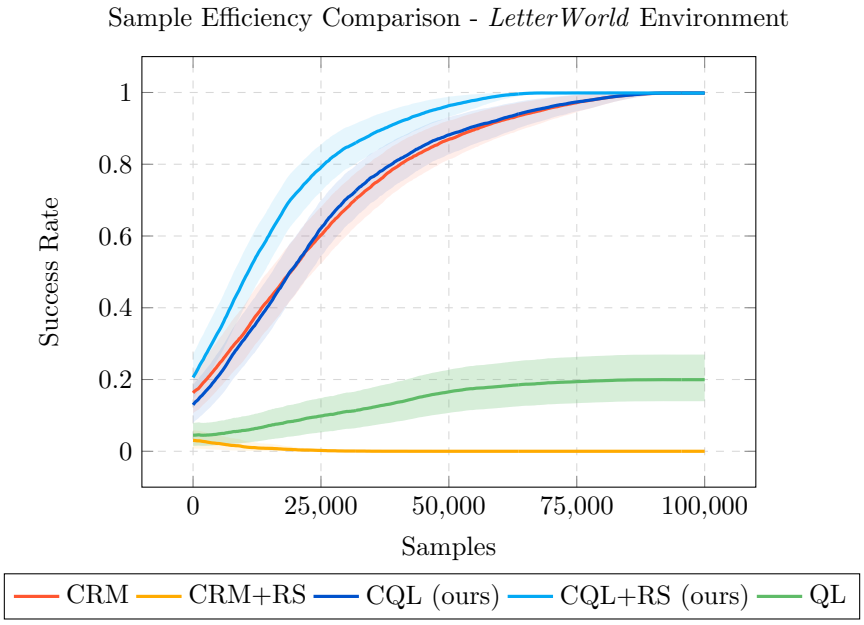


Figure 6.5: Sample efficiency comparison for the *LetterWorld* environment. Model hyperparameters are detailed in Appendix A.1, while parameters for performance estimation can be found in Appendix B.

The results clearly show two important observations. Firstly, the RS-CRA algorithm significantly increases the sample efficiency of CQL. As a result, CQL+RS exhibits the best performance out of any of the evaluated algorithms, outperforming both CQL and CRM by a significant margin. Secondly, the CRM+RS algorithm performs very poorly when applied to the large automata required for this experiment. This result is in agreement with that of the experiments performed in the original proposal of the approach [Icarte *et al.* 2022]. This follows from the fact that, as discussed, agents are incentivised to fail the task many times before learning the optimal solution, which results in poor performance.

OfficeWorld

We make use of a context-sensitive task specification in the *OfficeWorld* environment to evaluate our approach on a more challenging problem. Specifically, we make use of the task specification and environmental configuration discussed in Section 4.5.2. An example agent trajectory which solves the task is illustrated in Figure 4.4. Once again, the same set of algorithms are evaluated using the previously described sparse reward model.

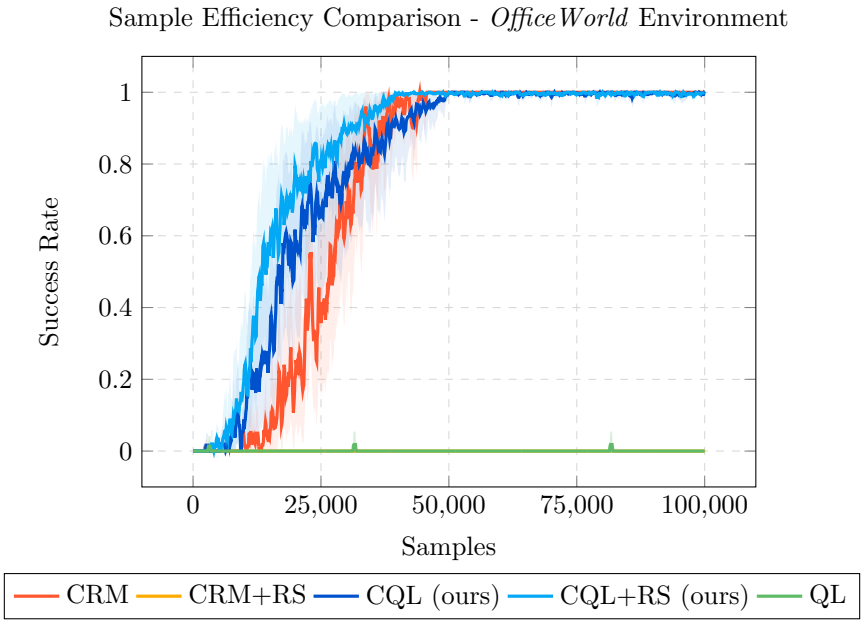


Figure 6.6: Sample efficiency comparison for the *OfficeWorld* environment. The results for CRM+RS are not visible as they are covered by that of QL. Model hyperparameters are detailed in Appendix A.2, while parameters for performance estimation can be found in Appendix B.

The results are illustrated in Figure 6.6. We observe similar behaviour to the previous experiment, with CQL+RS outperforming all of the other approaches. Similarly, CRM+RS is once again shown to exhibit poor performance. This result is expected as the complexity of the task specification has increased, requiring larger automata, which are known to be detrimental to the performance of CRM+RS.

6.4.2 Function Approximation

The RS-CRA algorithm does not require any extensions to be used in the context of function approximation. The function approximation algorithm simply makes use of the shaped reward rather than the original. In this section, we evaluate the performance of the aforementioned approaches when solving temporally extended high-dimensional control problems. Agents are evaluated in the extended version of the *PuckWorld* environment described in Section 4.6.3. Each agent is required to solve the context-sensitive task specification, which is also discussed in Section 4.6.3. Using the previously defined automata, we evaluate both sparse and dense reward models.

We begin by analysing the results associated with the sparse reward model, illustrated in Figure 6.7. From the results, it is clear that the RS-CRA algorithm is able to effectively simplify the learning process for temporally extended control problems through the introduction of additional learning signals. As a result, RS-CRA is shown to increase the sample efficiency of the CDQN algorithm significantly. Once again, the CRM+RS algorithm continues to exhibit poor performance when applied to the large automata required for complex tasks.

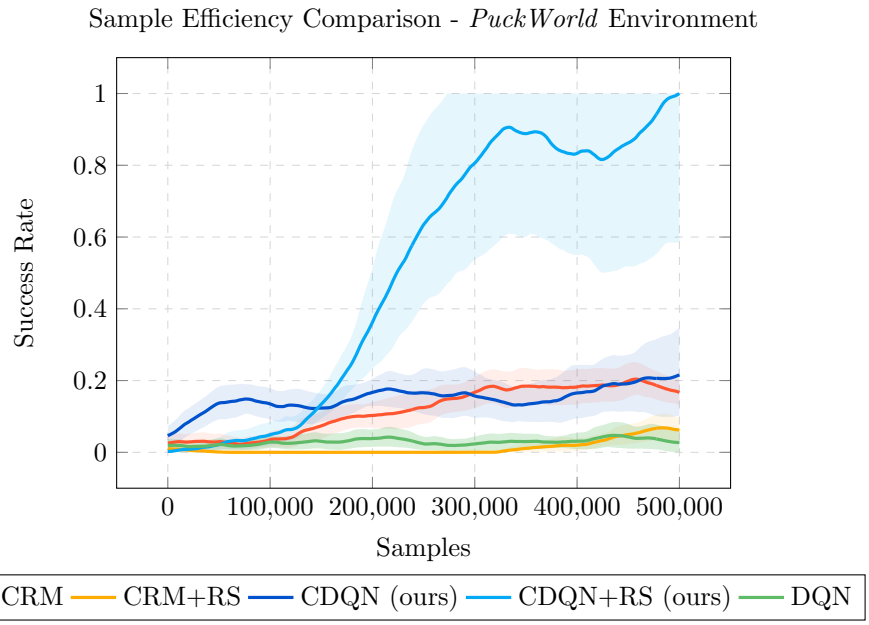


Figure 6.7: Sample efficiency comparison for the *PuckWorld* environment using sparse rewards. Model hyperparameters are detailed in Appendix A.3, while parameters for performance estimation can be found in Appendix B.

The results for dense reward models are illustrated in Figure 6.8. Once again, it is clear that the introduction of additional learning signals from the RS-CRA algorithm is beneficial to CDQN. However, the results are less dramatic as the base reward model provides significantly more information than that of the previous experiment. Regardless, the RS-CRA algorithm is once again shown to consistently provide measurable increases in sample efficiency. The CRM+RS algorithm performs slightly better than DQN in this experiment for two reasons. The agent now has access to additional useful reward signals from the dense reward model as well as counterfactual experience generation to speed up learning. However, this algorithm still performs significantly worse than CRM without reward shaping.

6.5 Conclusion

In this chapter, we have presented a novel approach to reward shaping known as reward shaping for counting reward automata. The formulation incorporates potential-based reward shaping, using the structure of a counting reward automaton to derive

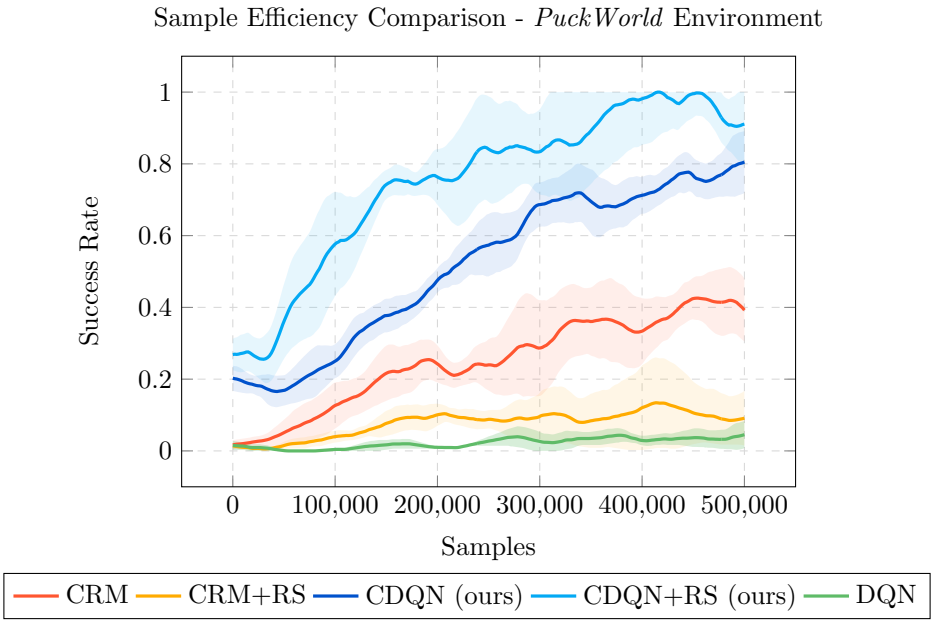


Figure 6.8: Sample efficiency comparison for the *PuckWorld* environment using dense rewards. Model hyperparameters are detailed in Appendix A.4, while parameters for performance estimation can be found in Appendix B.

a shaping reward function. Initially, we began by discussing the automated reward-shaping formulation. This discussion was followed by an illustrative example in which we demonstrated how the formulation can be used for reward shaping. An analysis was then conducted on the resulting reward model, clearly illustrating its limitations. Next, we proposed reward shaping for counting reward automata as a solution to the limitations associated with automated reward shaping. Specifically, we described the formulation before proceeding to an in-depth illustrative example. Finally, an experimental analysis was carried out in both tabular and function approximation domains. Throughout the analysis, reward shaping for counting reward automata was shown to consistently provide measurable increases in sample efficiency in comparison to competing approaches.

In conclusion, this chapter has presented a comprehensive examination of reward shaping for counting reward automata, showcasing its efficacy in addressing key challenges faced by current approaches. However, to gain a deeper understanding of its performance and compare it against existing approaches, we proceed to the next chapter, where we undertake a thorough analysis of the various methodologies presented in this research.

Chapter 7

Discussion

7.1 Introduction

In the preceding chapters, we introduced a number of novel approaches aimed at addressing the limitations of long-horizon reinforcement learning. Specifically, our focus has been the development of effective strategies to overcome the challenges faced by conventional RL techniques when applied in these domains. The design of our methods has been tailored towards a single overarching goal – allowing artificially intelligent agents to emulate the learning processes used by generally intelligent agents to solve long-horizon tasks.

In this chapter, we provide a comprehensive performance evaluation, including all of the approaches presented in this research. Through critical analysis and comparison, we aim to discern the strengths, weaknesses and overall effectiveness of each approach. Our evaluation covers various dimensions, including tabular domains, function approximation and, finally, both sparse and dense reward models. Furthermore, we contextualise our results within the broader landscape of existing literature and methodologies within the field.

This chapter is organised as follows. We begin in Section 7.2 by examining the performance of several algorithms across tasks of differing complexities within tabular domains. We evaluate these approaches in two tabular domains, each progressively more intricate than the last. Subsequently, in Section 7.3, we analyse the effectiveness of the same set of approaches in addressing temporally extended high-dimensional control problems using function approximation. We evaluate performance using sparse and dense reward models in this setting.

7.2 Tabular Reinforcement Learning

This section contains a performance evaluation for several algorithms applied to tasks of varying complexity in tabular domains. Specifically, we compare all of the approaches evaluated in previous chapters, in addition to the HCRA+RS and HRM+RS algorithms.

In this context, HRM+RS is used to denote HRM with automated reward shaping, and similarly, HCRA+RS is used to denote HCRA using a reward model defined by the RS-CRA formulation. In the first experiment, the agent is required to solve a context-free task specification in the *LetterWorld* environment. In the following experiment, agents are evaluated using a more complex context-sensitive task specification in a more difficult domain, namely *OfficeWorld*.

An overview of the results is as follows:

- Our approaches to counterfactual learning (CQL, HCRA, CRA+RS) consistently outperform the Q-Learning baseline in all experiments.
- Hierarchical methods (HCRA, HRM) consistently outperform purely counterfactual methods and exhibit the best performance in certain cases.
- Hierarchical methods (HCRA, HRM) tend to learn faster than purely counterfactual methods (CQL, CRM). However, they converge to suboptimal policies in some cases.
- Our approach to reward shaping (CQL+RS) consistently yields improved performance compared to purely counterfactual methods (CQL, CRM).
- Purely counterfactual methods (CQL, CRM) exhibit robust performance while remaining relatively simple in comparison to competing approaches,
- Reward machine-based approaches utilising automated reward shaping consistently fail to obtain solutions.

7.2.1 *LetterWorld*

This experiment is performed in the *LetterWorld* environment illustrated in Figure 4.1. Each algorithm is required to solve the CFL task, which has been used as an illustrative example in Section 4.2.2. The environment is configured as discussed in Section 4.5.2. The task makes use of a sparse reward model in which the agent is given a reward of one upon task completion and zero otherwise. Finally, each agent is trained for 100,000 environmental interactions. The success rate, which is the ability of the agent to complete the task specification, is recorded throughout the training process. The results are illustrated in Figure 7.1.

In this experiment, CQL+RS is shown to exhibit the best performance among the evaluated approaches. Following closely are the hierarchical methods, HRA and HRM. While these hierarchical techniques demonstrate accelerated learning compared to the purely counterfactual methods, CQL and CRM, they eventually converge towards suboptimal policies. Interestingly, the incorporation of reward shaping into the hierarchical approaches does not seem to enhance the learning process. Specifically, HCRA is shown to converge towards lower-quality solutions while HRM diverges completely.

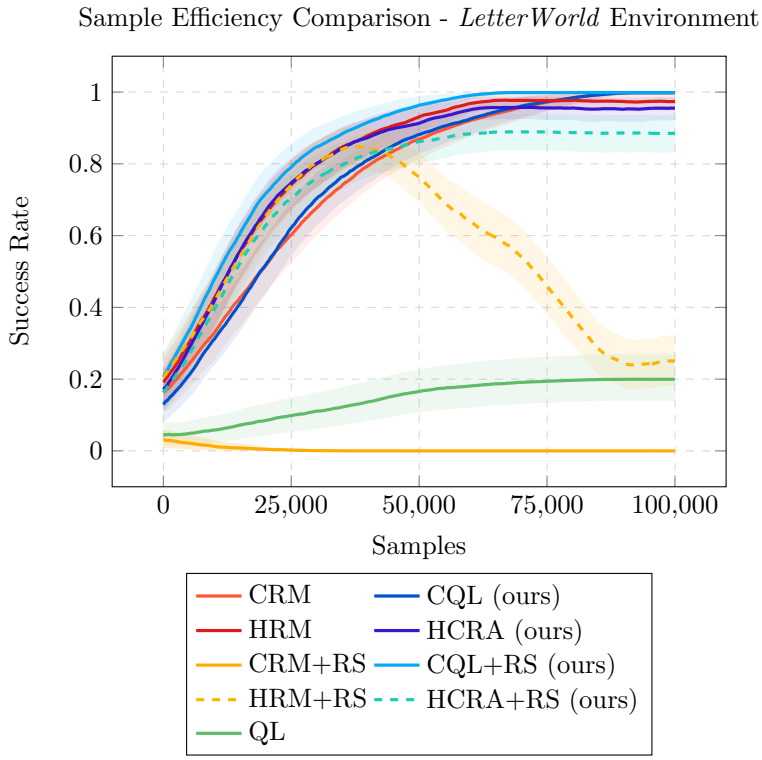


Figure 7.1: Sample efficiency comparison for the *LetterWorld* environment. Model hyperparameters are detailed in Appendix A.1, while parameters for performance estimation can be found in Appendix B.

7.2.2 *OfficeWorld*

To evaluate our approach on a more challenging problem, we make use of a context-sensitive task in the *OfficeWorld* environment illustrated in 4.4. Specifically, we make use of both the task specification as well as the environmental configuration described in Section 4.5.2 for performance evaluation. Once again, we compare the aforementioned set of algorithms using a sparse reward model in which the agent is given a reward value of one upon task completion and zero otherwise. The results are illustrated in Figure 7.2.

Among the methods we evaluated, the hierarchical learning approaches, HCRA and HRM, stand out for their superior performance. Following closely is our approach to counterfactual learning with reward shaping, the CQL+RS algorithm. Notably, incorporating reward shaping into HCRA does not seem to enhance learning speed. As anticipated, HRM with automated reward shaping fails to solve the task specification. Finally, the purely counterfactual methods, CQL and CRM, showcase robust performance, only slightly surpassed by that of the extended approaches, while offering a significantly simpler solution.

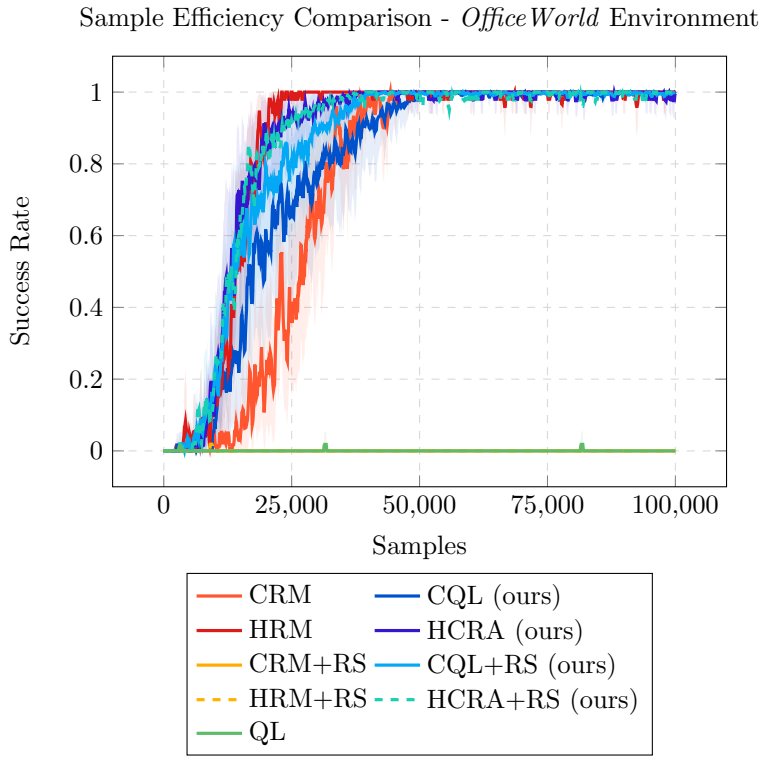


Figure 7.2: Sample efficiency comparison for the *OfficeWorld* environment. Due to occlusion caused by results that did not converge, not all lines on the graph are visible. Model hyperparameters are detailed in Appendix A.2, while parameters for performance estimation can be found in Appendix B.

7.2.3 Summary

In tabular reinforcement learning, we recommend starting with CQL because of its relative simplicity and robust performance. If better sample efficiency is desired, consider employing the RS-CRA algorithm for reward shaping, as it has been shown to consistently improve performance. For further performance enhancements, consider experimenting with the hierarchical approach HCRA, which offers the potential for significantly better performance as we have shown, albeit with a risk of converging to a suboptimal solution.

7.3 Function Approximation

In this section, we evaluate the performance of the aforementioned approaches when solving temporally extended high-dimensional control problems. For this evaluation, we make use of the extended version of the *PuckWorld* environment described in Section 4.6.3. Each agent is required to solve the context-sensitive task specification described in Section 4.6.3. As a result, we make use of the previously defined CRA for the task specification. This automaton supports both sparse and dense reward models, both of which are evaluated in the following experiments.

An overview of the results is as follows:

- Our approach to reward shaping is highly effective for both sparse and dense reward models.
- Our approach to reward shaping provides significant increases in sample efficiency when applied to sparse reward models.
- Hierarchical methods exhibit accelerated learning and are highly effective with both dense and sparse reward models.
- Our modifications to function approximation are highly effective when using either dense or sparse rewards.
- Reward shaping is only useful for hierarchical methods when using sparse reward models.

7.3.1 Sparse Reward Model

We begin our analysis by examining the results associated with training agents in the *PuckWorld* environment using a sparse reward model. This initial investigation aims to provide insights into the performance and learning dynamics of agents in a challenging setting where the learning signal is sparse.

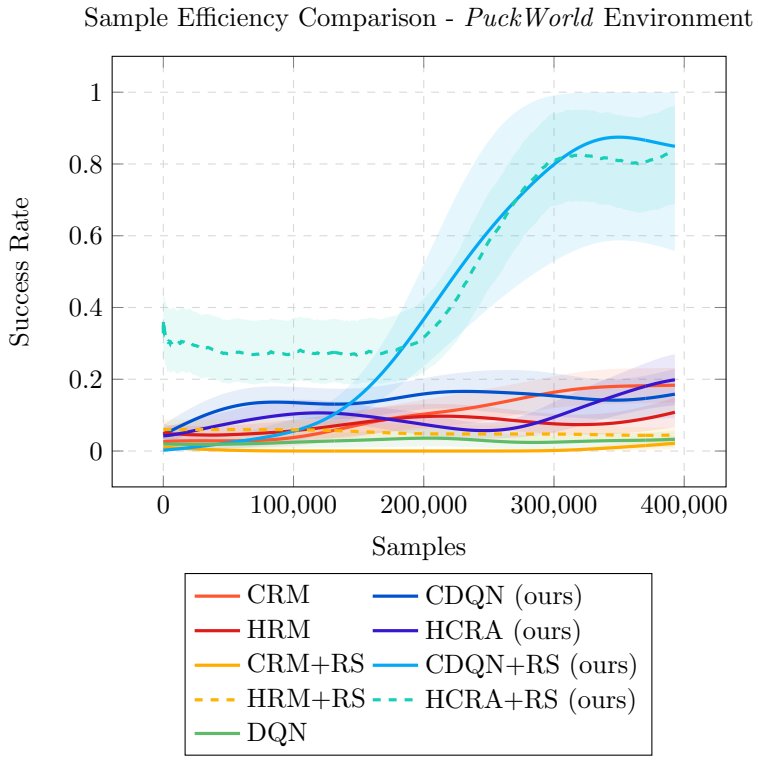


Figure 7.3: Sample efficiency comparison for the *PuckWorld* environment using sparse rewards. Model hyperparameters are detailed in Appendix A.3, while parameters for performance estimation can be found in Appendix B.

From the results illustrated in Figure 7.3, it is evident that our approach to reward shaping with counterfactual experience generation significantly improves performance, outperforming all other approaches by a significant margin. Specifically, our algorithms CDQN+RS and HCRA+RS demonstrate notably superior performance in comparison to all other methods. While the hierarchical algorithm HCRA+RS initially exhibits accelerated learning, CDQN+RS eventually catches up during training, yielding comparable results by the end of the training process. Our modified function approximation algorithms (CDQN, HCRA) are shown to be significantly more sample-efficient than the existing state-of-the-art approaches (CRM, HRM). Notably, the purely counterfactual methods outperform the hierarchical approaches in this experiment. Finally, automated reward shaping once again displays poor performance in this environment.

7.3.2 Dense Reward Model

We will now shift our focus to analysing the results obtained from training agents in the *PuckWorld* environment using a dense reward model. This analysis aims to provide insights into the agent's performance and learning dynamics under conditions where the reward signal is more abundant.

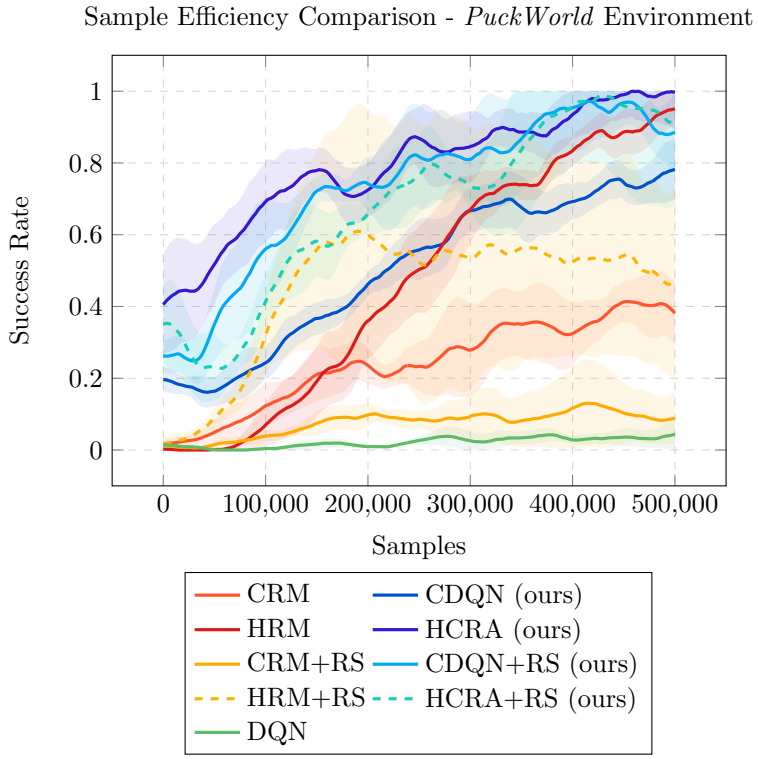


Figure 7.4: Sample efficiency comparison for the *PuckWorld* environment using dense rewards. Model hyperparameters are detailed in Appendix A.3, while parameters for performance estimation can be found in Appendix B.

The results of this experiment are presented in Figure 7.4. Hierarchical approaches, particularly HCRA, demonstrate the highest performance in this experiment. Addition-

ally, our approach to reward shaping yields impressive results, with CDQN+RS performing only slightly worse than HCRA. However, incorporating reward shaping into hierarchical methods does not appear to enhance learning. Despite their simplicity, the counterfactual approaches perform well. Furthermore, the modifications we introduced for function approximation are proven to be highly effective, resulting in all of our approaches significantly outperforming the current state-of-the-art techniques.

7.3.3 Summary

In the context of function approximation, we suggest the following approach: If the reward model is sparse, begin with our approach to reward shaping CDQN+RS. Should you require rapid learning to a base level of competency, utilise our hierarchical approach with reward shaping, HCRA+RS; otherwise, simply reward shaping should suffice. If the reward model is dense, start with our hierarchical reinforcement learning approach, HCRA, as it offers significant performance improvements over purely counterfactual approaches. However, for a straightforward and effective starting point, consider employing CDQN, which offers simplicity coupled with good performance.

7.4 Conclusion

In this chapter, we conducted a comprehensive performance evaluation encompassing all approaches presented in this research. Our evaluation covered various factors, including tabular domains, function approximation, and both sparse and dense reward models. Additionally, we contextualised our findings within the broader landscape of existing literature and methodologies, seeking to understand the effectiveness of our contributions against established benchmarks and state-of-the-art techniques. Ultimately, this discussion serves as a reflection on the efficacy of the methods proposed in this thesis, paving the way for future research endeavours within the field, a topic which is discussed further in the next chapter.

Chapter 8

Conclusion and Future Work

8.1 Future Work

This research has focused on understanding the learning patterns characteristic of generally intelligent agents when solving temporally extended tasks. One key aspect of such agents is the ability to learn subtasks in parallel, a highly desirable trait in achieving sample-efficient learning. As the reward machine formulation offers a restricted framework compatible with this goal, we have generalised the approach with the introduction of the CRA framework to support any reward function that can be expressed as a well-defined algorithm. However, despite these advancements, the full potential of the CRA framework remains underutilised. As a result, we now consider potential avenues for future research.

Sample efficiency

Gathering experience from the real world presents challenges due to the associated costs as well as potential risks. Therefore, a crucial aspect of reinforcement learning revolves around achieving sample efficiency, aiming to extract maximum information from each interaction with the environment. This efficiency is essential for learning the best policy when access to training data is limited. While we have introduced several approaches to enhance sample efficiency, primarily relying on off-policy learning methods, we have yet to explore how CRA can improve on-policy algorithms. However, these approaches are constrained in that they are only able to learn from experiences created by the policy which is being improved. This constraint poses challenges in utilising counterfactual CRA experience for policy updates. As an initial step towards addressing this issue, we propose leveraging importance sampling in order to account for policy differences across various CRA configurations.

Concurrent Composition

The capacity to compose acquired skills to solve novel tasks is a necessary characteristic of any lifelong learning agent. This aspect of learning, known as concurrent compo-

sition, involves agents reusing existing knowledge to solve new tasks. In contrast, the main focus of this work has been on temporal composition, which involves the sequential combination of behaviours to solve problems. Concurrent composition gains particular significance in multitask or lifelong learning scenarios, where learning to solve intricate tasks from scratch proves impractical. The work done by Nangue Tasse *et al.* [2020] has contributed to formalising the logical composition of tasks through Boolean algebra, enabling the formulation of new tasks based on the negation, disjunction, and conjunction of foundational tasks.

Efforts have been made to enable agents to solve a diverse array of problems specified through language within the same environment. One prevalent approach entails reusing skills acquired from prior tasks and generalising them compositionally to solve new problems. However, this approach poses challenges due to the curse of dimensionality arising from the myriad of ways in which high-level goals can be combined logically and temporally in language. To address this challenge, the *Skill Machines* framework has been introduced in Nangue Tasse *et al.* [2023], wherein an agent initially learns a sufficient set of skill primitives to accomplish all high-level goals within its environment. Subsequently, the agent can flexibly compose these skills both logically and temporally to reliably fulfil temporal logic specifications in any regular language, such as regular fragments of linear temporal logic. However, this framework is confined to regular languages. As a result, replacing the state machine at the core of this approach with a CRA is a promising avenue for future work, extending the set of problems compatible with the formulation.

Safe Reinforcement Learning

Lastly, we turn our attention to safety, a critical consideration in reinforcement learning. Guaranteeing the safety of agents' behaviour under all conditions is essential before reinforcement learning can be widely adopted in everyday life. Typically, agents can be trained to adhere to safety protocols by imposing significant negative rewards for constraint violations. However, this presents challenges, as agents optimise for expected discounted returns and may prioritise immediate rewards over adhering to safety constraints. CRA offer a solution by modelling non-Markovian rewards, which simplifies the safety problem. Specifically, they can provide distinct rewards after a safety constraint has been violated. For instance, a CRA may offer a positive reward for task completion but no reward if safety rules are violated during an episode. Consequently, the optimal policy would prioritise avoiding rule violations or minimising the probability of doing so when unavoidable. This aspect presents an intriguing avenue for future research that remains largely unexplored.

8.2 Conclusion

We have introduced counting reward automata as a method to enable artificial agents to mimic the learning patterns characteristic of generally intelligent agents when solving temporally extended tasks. Specifically, we have defined a novel abstract machine,

known as a counting reward automaton, capable of modelling any reward function expressible as a decidable formal language. This achievement is realised by representing reward functions through k-counter automata, diverging from the use of finite state machines in the reward machine formulation. We show that the state machines produced by our approach are not only more intuitive to specify than reward machines but also significantly simpler both in terms of theoretical and practical measures. In addition, we establish that our approach converges to globally optimal policies, as opposed to hierarchically optimal ones. Furthermore, we propose a set of reinforcement learning algorithms that leverage the structure of counting reward automata to enhance sample efficiency. We introduce one approach based on hierarchical reinforcement learning alongside another that builds upon the concept of potential-based reward shaping.

The contributions of this thesis, including counting reward automata and their associated learning algorithms, play a pivotal role in advancing the field of reinforcement learning. These innovations offer promising avenues for addressing complex decision-making tasks. By enhancing our understanding and capabilities for handling temporally extended behaviours, these contributions extend the frontier of reinforcement learning research and applications, offering potential solutions to real-world challenges across various domains.

Appendix A

Model Hyperparameters

A.1 *LetterWorld*

Model	γ	α	ε_{start}	ε_{end}	ω	r^+	r^-
QL	0.99	0.001	1.0	0.01	N/A	N/A	N/A
CRM	0.99	0.001	1.0	0.01	N/A	N/A	N/A
CRM+RS	0.99	0.001	1.0	0.01	N/A	N/A	N/A
CQL	0.99	0.001	1.0	0.01	N/A	N/A	N/A
CQL+RS	1.0	0.001	1.0	0.01	1.0	N/A	N/A
HRM	0.99	0.01	1.0	0.01	N/A	1.0	-1.0
HRM + RS	0.99	0.01	1.0	0.01	N/A	1.0	-1.0
HCRA	0.99	0.01	1.0	0.01	N/A	1.0	-1.0
HCRA + RS	1.0	0.01	1.0	0.01	1.0	1.0	-1.0

Table A.1: Model hyperparameters for the experiments conducted in the *LetterWorld* environment.

A.2 *OfficeWorld*

Model	γ	α	ε_{start}	ε_{end}	ω	r^+	r^-
QL	0.99	0.0001	1.0	0.01	N/A	N/A	N/A
CRM	0.99	0.0001	1.0	0.01	N/A	N/A	N/A
CRM+RS	0.99	0.0001	1.0	0.01	N/A	N/A	N/A
CQL	0.99	0.0001	1.0	0.01	N/A	N/A	N/A
CQL+RS	1.0	0.0001	1.0	0.01	1.0	N/A	N/A
HRM	0.99	0.0001	1.0	0.01	N/A	1.0	-1.0
HRM+RS	0.99	0.0001	1.0	0.01	N/A	1.0	-1.0
HCRA	0.99	0.0001	1.0	0.01	N/A	1.0	-1.0
HCRA+RS	1.0	0.0001	1.0	0.01	1.0	1.0	-1.0

Table A.2: Model hyperparameters for the experiments conducted in the *OfficeWorld* environment.

A.3 *PuckWorld* Sparse Reward Model

Base Model Parameters	
Policy	"mlpolicy"
Learning Starts	1000
Learning Rate (α)	0.0001
Exploration Final Eps	0.01
Exploration Fraction	0.1
Batch Size	32
Buffer Size	1000000
Gradient Steps	1
Target Update Interval	10000
Train Frequency	4
Tau	1.0
CQL Modifiers	
Gradient Steps Multiplier (β)	4
HRL & HCRA	
Options Reward (r^+)	1.0
Options Reward (r^-)	-1.0
H-CRA	
Reward Modifier (ω)	1.0

Table A.3: Model hyperparameters for the experiments conducted in the *PuckWorld* environment, which make use of a sparse reward model. Hyperparameter names are given as defined in the Stable Baselines3 implementation [Raffin *et al.* 2021]. DQN base hyperparameters taken from Mnih *et al.* [2015].

A.4 *PuckWorld* Dense Reward Model

Base Model Parameters	
Policy	"mlpolicy"
Learning Starts	1000
Learning Rate (α)	0.0001
Exploration Final Eps	0.01
Exploration Fraction	0.1
Batch Size	32
Buffer Size	1000000
Gradient Steps	1
Target Update Interval	10000
Train Frequency	4
Tau	1.0
CQL Modifiers	
Gradient Steps Multiplier (β)	4
HRL & HCRA	
Options Reward (r^+)	1.0
Options Reward (r^-)	-1.0
H-CRA	
Reward Modifier (ω)	1.0

Table A.4: Model hyperparameters for the experiments conducted in the *PuckWorld* environment, which make use of a dense reward model. Hyperparameter names are given as defined in the Stable Baselines3 implementation [Raffin *et al.* 2021]. DQN base hyperparameters taken from Mnih *et al.* [2015].

Appendix B

Performance Estimation

Environment	n	m
LetterWorld	100	10,000
OfficeWorld	100	10,000
PuckWorld (Sparse)	10	10,000
PuckWorld (Dense)	10	10,000

Table B.1: Bootstrap parameters used for performance estimation in each environment.

Experiment	n	m
Sample Efficiency	10	1
Task Completion	10	1
Reward Flux	10	1

Table B.2: Bootstrap parameters used for performance estimation in the experiments in Section 4.6.

Appendix C

Product MDP State Encoding

For use with function approximation, we follow the approach described in Icarte *et al.* [2022] for encoding the cross-product MDP states. That is, the active reward machine state is represented using a one-hot vector which is concatenated onto the representation of the ground environment state. We follow a similar approach when encoding AAMDP states. Firstly, the active automaton state is encoded using a one-hot vector. Secondly, the values of the counter variables are represented as a vector. These two vectors are concatenated onto the representation of the ground environment state to produce a representation of the active AAMDP state.

References

[Abbeel and Ng 2004] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[Akrour *et al.* 2012] Riad Akrour, Marc Schoenauer, and Michèle Sebag. April: Active preference learning-based reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part II 23*, pages 116–131. Springer, 2012.

[Albrecht and Gaffney 1983] Allan J. Albrecht and John E Gaffney. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE transactions on software engineering*, (6):639–648, 1983.

[Amodei *et al.* 2016] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[Andreas *et al.* 2017] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International conference on machine learning*, pages 166–175. PMLR, 2017.

[Andrychowicz *et al.* 2017] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[Argall *et al.* 2009] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[Arjona-Medina *et al.* 2019] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

[Bacchus *et al.* 1996] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1160–1167, 1996.

[Badia *et al.* 2020] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *International conference on machine learning*, pages 507–517. PMLR, 2020.

[Badnava *et al.* 2023] Babak Badnava, Mona Esmaeili, Nasser Mozayani, and Payman Zarkesh-Ha. A new potential-based reward shaping for reinforcement learning agent. In *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 01–06. IEEE, 2023.

[Barendregt 1984] Hendrik Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[Barto and Mahadevan 2003] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

[Bellman 1954] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[Bellman 1957] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[Bester *et al.* 2024] Tristan Bester, Benjamin Rosman, Steven James, and Geraud Nangue Tasse. Counting reward automata: Sample efficient reinforcement learning through the exploitation of reward function structure. In *Workshop on Neuro-Symbolic Learning and Reasoning in the Era of Large Language Models at AAAI*, 2024.

[Botvinick *et al.* 2009] Matthew M Botvinick, Yael Niv, and Andew G Barto. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *cognition*, 113(3):262–280, 2009.

[Brafman *et al.* 2018] Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. Ltlf/ldlf non-markovian rewards. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[Brooks 1986] Rodney A Brooks. Achieving artificial intelligence through building robots. 1986.

[Brunskill and Li 2014] Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International conference on machine learning*, pages 316–324. PMLR, 2014.

[Cai *et al.* 2021] Mingyu Cai, Mohammadhosein Hasanbeig, Shaoping Xiao, Alessandro Abate, and Zhen Kan. Modular deep reinforcement learning for continuous motion planning with temporal logic. *IEEE robotics and automation letters*, 6(4):7973–7980, 2021.

[Camacho *et al.* 2017] Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A McIlraith. *Decision-Making with Non-Markovian Rewards: Guiding search via automata-*

based reward shaping. Technical report, Technical Report CSRG-632, Department of Computer Science, University of Toronto, 2017.

[Camacho *et al.* 2018] Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A McIlraith. Non-markovian rewards expressed in ltl: Guiding search via reward shaping (extended version). In *GoalsRL, a workshop collocated with ICML/IJCAI/AAMAS*, 2018.

[Camacho *et al.* 2019] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q Klassen, Richard Anthony Valenzano, and Sheila A McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, volume 19, pages 6065–6073, 2019.

[Chatterji *et al.* 2021] Niladri Chatterji, Aldo Pacchiano, Peter Bartlett, and Michael Jordan. On the theory of reinforcement learning with once-per-episode feedback. *Advances in Neural Information Processing Systems*, 34:3401–3412, 2021.

[Chomsky 1956] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[Christiano *et al.* 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[Clarke *et al.* 2009] Bertrand Clarke, Ernest Fokoue, and Hao Helen Zhang. *Principles and theory for data mining and machine learning*. Springer, 2009.

[Dietterich 2000] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[Dijkstra 1959] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[Eilenberg 1974] Samuel Eilenberg. *Automata, languages, and machines*. Academic press, 1974.

[Ferrari and Stengel 2005] Silvia Ferrari and Robert F Stengel. Smooth function approximation using neural networks. *IEEE Transactions on Neural Networks*, 16(1):24–38, 2005.

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

[Fletcher *et al.* 1991] Peter Fletcher, Hughes Hoyle, and C Wayne Patty. Foundations of discrete mathematics. *(No Title)*, 1991.

[Gruber and Holzer 2007] Hermann Gruber and Markus Holzer. Computational complexity of nfa minimization for finite and unary languages. *LATA*, 8:261–272, 2007.

[Grzes 2017] Marek Grzes. Reward shaping in episodic reinforcement learning. 2017.

[Hadfield-Menell *et al.* 2017] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. *Advances in neural information processing systems*, 30, 2017.

[Haider *et al.* 2021] Abbas Haider, Glenn Hawe, Hui Wang, and Bryan Scotney. Gaussian based non-linear function approximation for reinforcement learning. *SN Computer Science*, 2:1–12, 2021.

[Hasanbeig *et al.* 2018] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained reinforcement learning. *arXiv preprint arXiv:1801.08099*, 2018.

[Haykin 1994] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[Hopcroft *et al.* 2001] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[Icarte *et al.* 2018] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

[Icarte *et al.* 2022] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[Kalashnikov *et al.* 2018] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, and Vincent Vanhoucke. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018.

[Knox and Stone 2008] W Bradley Knox and Peter Stone. Tamer: Training an agent manually via evaluative reinforcement. In *2008 7th IEEE international conference on development and learning*, pages 292–297. IEEE, 2008.

[Konidaris and Barto 2007] George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *Ijcai*, volume 7, pages 895–900, 2007.

[Kumar *et al.* 2016] Vikash Kumar, Emanuel Todorov, and Sergey Levine. Optimal control with learned local models: Application to dexterous manipulation. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 378–383. IEEE, 2016.

[Lacerda *et al.* 2014] Bruno Lacerda, David Parker, and Nick Hawes. Optimal and dynamic planning for markov decision processes with co-safe ltl specifications. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1511–1516. IEEE, 2014.

[Lansdell *et al.* 2019] Benjamin James Lansdell, Prashanth Ravi Prakash, and Konrad Paul Kording. Learning to solve the credit assignment problem. *arXiv preprint arXiv:1906.00889*, 2019.

[Lenaers and van Otterlo 2022] Nicky Lenaers and Martijn van Otterlo. Regular decision processes for grid worlds. In *Artificial Intelligence and Machine Learning: 33rd Benelux Conference on Artificial Intelligence, BNAIC/Benelearn 2021, Esch-sur-Alzette, Luxembourg, November 10–12, 2021, Revised Selected Papers 33*, pages 218–238. Springer, 2022.

[Li and Belta 2019] Xiao Li and Calin Belta. Temporal logic guided safe reinforcement learning using control barrier functions. *arXiv preprint arXiv:1903.09885*, 2019.

[Li *et al.* 2017] Xiao Li, Cristian-Ioan Vasile, and Calin Belta. Reinforcement learning with temporal logic rewards. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3834–3839. IEEE, 2017.

[Linz and Rodger 2022] Peter Linz and Susan H Rodger. *An introduction to formal languages and automata*. Jones & Bartlett Learning, 2022.

[Littman *et al.* 2017] Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via gltl. *arXiv preprint arXiv:1704.04341*, 2017.

[MacGlashan *et al.* 2017] James MacGlashan, Mark K Ho, Robert Loftin, Bei Peng, Guan Wang, David L Roberts, Matthew E Taylor, and Michael L Littman. Interactive learning from policy-dependent human feedback. In *International conference on machine learning*, pages 2285–2294. PMLR, 2017.

[Mahadevan *et al.* 1997] Sridhar Mahadevan, Nicholas Marchalleck, Tapas K Das, and Abhijit Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 202–210. MORGAN KAUFMANN PUBLISHERS, INC., 1997.

[Mann *et al.* 2015] Timothy A Mann, Shie Mannor, and Doina Precup. Approximate value iteration with temporally extended actions. *Journal of Artificial Intelligence Research*, 53:375–438, 2015.

[Meduna 2014] Alexander Meduna. *Formal languages and computation: models and their applications*. CRC Press, 2014.

[Merrill 2020] William Merrill. On the linguistic capacity of real-time counter automata. *arXiv preprint arXiv:2004.06866*, 2020.

[Middleton *et al.* 2020] Jaime Middleton, Toryn Q Klassen, JA Baier, and Sheila A McIlraith. Fl-at: A formal language–automaton transmogrifier. In *System demonstration at The 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.

[Mitchell 2009] Melanie Mitchell. *Complexity: A guided tour*. Oxford university press, 2009.

[Mnih *et al.* 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Mnih *et al.* 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, and Georg Ostrovski. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[Moll *et al.* 2012] Robert N Moll, Michael A Arbib, and Assaf J Kfoury. *An introduction to formal language theory*. Springer Science & Business Media, 2012.

[Nangue Tasse *et al.* 2020] Geraud Nangue Tasse, Steven James, and Benjamin Rosman. A boolean task algebra for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:9497–9507, 2020.

[Nangue Tasse *et al.* 2023] Geraud Nangue Tasse, Devon Jarvis, Steven James, and Benjamin Rosman. Skill machines: Temporal logic skill composition in reinforcement learning. In *The Twelfth International Conference on Learning Representations*, 2023.

[Ng and Russell 2000] Andrew Y Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[Ng *et al.* 1999] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[Parr and Russell 1997] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

[Patterson *et al.* 2020] Andrew Patterson, Samuel Neumann, Martha White, and Adam White. Draft: Empirical design in reinforcement learning. *Journal of Artificial Intelligence Research*, 1, 2020.

[Pitis *et al.* 2020] Silviu Pitis, Elliot Creager, and Animesh Garg. Counterfactual data augmentation using locally factored dynamics. *Advances in Neural Information Processing Systems*, 33:3976–3990, 2020.

[Pnueli 1977] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[Puterman 2014] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[Raffin *et al.* 2021] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[Riaz and Zafar 2008] Shagufta Riaz and Nazir Ahmad Zafar. Constructive formal conversion of moore machine to deterministic finite automata. In *WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering*, number 10. WSEAS, 2008.

[Rogers Jr 1987] Hartley Rogers Jr. *Theory of recursive functions and effective computability*. MIT press, 1987.

[Singh 1992] Satinder P Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the National Conference on Artificial Intelligence*, number 10, page 202. Citeseer, 1992.

[Steyer and Nagel 2017] Rolf Steyer and Werner Nagel. *Probability and conditional expectation: Fundamentals for the empirical sciences*, volume 5. John Wiley & Sons, 2017.

[Sutton and Barto 2018] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sutton *et al.* 1999a] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[Sutton *et al.* 1999b] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[Thomaz *et al.* 2006] Andrea L Thomaz, Guy Hoffman, and Cynthia Breazeal. Reinforcement learning with human teachers: Understanding how people want to teach robots. In *ROMAN 2006-The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 352–357. IEEE, 2006.

[Thrun and Schwartz 1994] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. *Advances in neural information processing systems*, 7, 1994.

[Toro Icarte *et al.* 2018a] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Teaching multiple tasks to an rl agent using ltl. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 452–461, 2018.

[Toro Icarte *et al.* 2018b] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Anthony Valenzano, and Sheila A McIlraith. Advice-based exploration in model-based reinforcement learning. In *Advances in Artificial Intelligence: 31st Canadian Conference on Artificial Intelligence, Canadian AI 2018, Toronto, ON, Canada, May 8–11, 2018, Proceedings 31*, pages 72–83. Springer, 2018.

[Vaezipoor *et al.* 2021] Pashootan Vaezipoor, Andrew C Li, Rodrigo A Toro Icarte, and Sheila A Mcilraith. Ltl2action: Generalizing ltl instructions for multi-task rl. In *International Conference on Machine Learning*, pages 10497–10508. PMLR, 2021.

[Valiant and Paterson 1975] Leslie G Valiant and Michael S Paterson. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10(3):340–350, 1975.

[Van Hasselt 2011] Hado Philip Van Hasselt. *Insights in reinforcement learning*. Hado van Hasselt, 2011.

[Watkins and Dayan 1992] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[Yarats *et al.* 2021] Denis Yarats, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto. Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv preprint arXiv:2107.09645*, 2021.

[Yu 1997] Sheng Yu. *Regular languages*. Springer, 1997.

[Ziebart *et al.* 2008] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.