

LEARNING AND EXPLOITING REWARD MACHINES FOR
REINFORCEMENT LEARNING

Daniel Furelos Blanco

Department of Computing
Imperial College London

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

October 2023

Statement of Originality

I, Daniel Furelos Blanco, declare that the work in this thesis is my own. The work of others has been appropriately referenced. A full list of references is given in the bibliography.

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Licence (CC BY NC-SA). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that; you credit the author, do not use it for commercial purposes and share any derivative works under the same licence. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

Reinforcement learning (RL) with non-Markovian rewards requires agents to learn history-dependent policies, which is particularly challenging in long-horizon or sparse reward settings. Reward machines (RMs) are finite-state machines that represent non-Markovian reward functions in terms of high-level events. By compactly encoding high-level event histories, RMs thereby constitute an external memory that makes rewards Markovian and enables the applicability of standard RL algorithms in non-Markovian reward settings. The structure elucidated by RMs facilitates task decomposition, allowing policy learning to become more efficient when rewards are sparse. Nevertheless, the potential of RMs is limited by the complexity of handcrafting them and the lack of reusability within larger RMs. In this thesis, we address these problems.

In the first part of the thesis, we devise a method for learning minimal RMs from traces of high-level events observed by an RL agent. The learning is powered by an inductive logic programming system and is launched when the current RM does not correctly recognize a trace. To make RM learning more efficient, we conceive a symmetry breaking mechanism to shrink the search space whilst remaining complete. We empirically demonstrate that exploiting a learned RM leads to performance on par with a handcrafted one.

In the second part of the thesis, we build hierarchies of RMs (HRMs) by endowing RMs with the ability to call each other, enabling the reusability of the RMs' structures and policies. In particular, the HRMs are exploited by treating each call as an independently solvable subtask, and learned through a curriculum-based method extending our RM learning approach. Our experiments reveal that (i) exploiting a handcrafted HRM leads to faster convergence than with a flat HRM, and (ii) learning an HRM is feasible in cases where its equivalent flat representation is not.

To my family

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, without whom this thesis would not have been possible. Alessandra Russo and Krysia Broda provided me with encouragement, freedom, and unwavering support throughout all my studies. In October 2018, during our second meeting, I talked about an idea I had seen at ICML’18 that intrigued me called *reward machines*. Five years later, here we are. Having the chance to explore and make my own choices from the very beginning has been truly invaluable. Mark Law played a vital role during the early stages of my degree. He not only helped me get started in the world of answer set programming, but also spent countless hours with me finding strategies to make the learning of reward machines more efficient. His patience and serenity during our meetings were reassuring and kept me confident when things were not working out. Anders Jonsson has been an incredible mentor since I met him as an undergraduate at Universitat Pompeu Fabra eleven years ago. He introduced me to object-oriented programming, competitive programming and, ultimately, artificial intelligence research. Without his guidance, I would not have pursued a PhD.

I am sincerely thankful to Sheila McIlraith and Francesco Belardinelli for serving as my examiners and providing insightful feedback. The work of Sheila’s group has not only been motivating but has also been a standard for rigor and clarity. Francesco thoroughly assessed all my PhD milestones, ensuring that my research stayed on the right track.

I want to thank the members of the SPIKE research group for their kindness, constructive criticism, and companionship. I am particularly grateful to Alex F. Spies, Charles Pert, Daniel Pace, Davide Cavezza, Frederik Kelbel, Hadeel Al-Negheimish, Kexin Gu Baugh, Nuri Cingillioglu, Paweł Gomoluch, and Rachel Lee Mekhtieva for their friendship and keeping me (somewhat) sane throughout the degree. Huxley’s 558B has not only witnessed different research ideas grow, but also crazy solutions to fill gaps in the market, a coffee business, endless memes, rock bands, a fascination for pigeons, and weekly SuperTuxKart and MarioKart world championships. It has been a great privilege to share this journey with them.

I have been fortunate to be involved in parallel research projects during my studies. My internship at InstaDeep enabled me to explore topics and technologies I had no experience with. I sincerely appreciate the trust and positivity of their research team, which kept me going when I doubted myself. I am also grateful to my fellow PhD candidates Leo Ardon and Roko Parać, and my co-supervised students—Miroslav Lambrev, Rahil Shah, and Zitai Wang—who made me grow as a mentor and pushed me to think further about the possibilities of my work.

Beyond research, I also highly appreciate the work of people at the Department of Computing, including Amani El-Kholy, Hassan Patel, and the members of the Computing Support Group, who make the lives of PhD students much easier.

Leaving Barcelona for London is one of the hardest decisions I have ever made. However, despite the distance, my friends and family have always made me feel as if I had never left. I wholeheartedly thank my high-school friends—Roger, Oriol, Gian, Dani G., Ester, Laura, Laia, Xisco, Elena, Carla, Dani D., Maria, and Anna—for their encouragement and for taking the time to visit me or catch up whenever I am back home. I am also grateful to my childhood friends—Sawo, Sergio, and Andrés—who always lifted my spirits through thick and thin. Finally, I am eternally thankful to my parents, Marisol and Ramón, and my brother Víctor for their unconditional love and support. None of my accomplishments would have been possible without them.

Contents

List of Figures	xiii
List of Tables	xv
Acronyms	xvii
Notation	xix
1 Introduction	1
1.1 Learning and Exploiting Reward Machines	3
1.2 Learning and Exploiting Hierarchies of Reward Machines	4
1.3 Publications	4
1.4 Thesis Structure	5
2 Background	7
2.1 Reinforcement Learning	7
2.1.1 Fundamentals	7
2.1.2 Deep Reinforcement Learning	10
2.1.3 Hierarchical Reinforcement Learning	11
2.1.4 Non-Markovian Reward Decision Processes	14
2.1.5 Reward Machines	17
2.2 Inductive Learning of Answer Set Programs	20
2.2.1 Answer Set Programming	20
2.2.2 ILASP	22
I Reward Machines	25
3 Formalism of Reward Machines	27
3.1 Tasks	27
3.2 Reward Machines	29
3.3 Representation in Answer Set Programming	32
3.3.1 Traces	32
3.3.2 Reward Machines	32
3.3.3 Proof of Correctness	35
3.3.4 Determinism	37

3.4	Symmetry Breaking	37
3.4.1	Graph Indexing	39
3.4.2	SAT Encoding	42
3.4.3	Application to Reward Machines	47
3.5	Summary	53
4	Learning and Exploiting Reward Machines	55
4.1	Exploiting Reward Machines	55
4.1.1	Learning an Option for each Edge and a Metapolicy for each State (HRL) . .	56
4.1.2	Learning an Option for each Reward Machine State (QRM)	59
4.2	Learning Reward Machines from Traces	60
4.2.1	The Reward Machine Learning Task	60
4.2.2	Solving the Reward Machine Learning Task with ILASP	61
4.3	Interleaved Learning	64
4.3.1	Algorithm	64
4.3.2	Properties	66
4.3.3	Implementation	66
4.4	Summary	67
5	Evaluation of Reward Machines	69
5.1	Experimental Setup	69
5.1.1	Generalization	69
5.1.2	Restrictions	70
5.1.3	Reinforcement Learning Algorithms	72
5.1.4	Reporting Results	72
5.2	Experiments in OFFICEWORLD	73
5.2.1	Instance Generation	73
5.2.2	Tasks	73
5.2.3	Hyperparameters	73
5.2.4	Results	74
5.2.5	Ablations	76
5.3	Experiments in CRAFTWORLD	83
5.3.1	Instance Generation	84
5.3.2	Tasks	84
5.3.3	Hyperparameters	85
5.3.4	Results	85
5.4	Experiments in WATERWORLD	86
5.4.1	Instance Generation	86
5.4.2	Tasks	88
5.4.3	Hyperparameters	88
5.4.4	Results	89
5.5	Summary	90

II	Hierarchies of Reward Machines	93
6	Formalism of Hierarchies of Reward Machines	95
6.1	Hierarchies of Reward Machines	97
6.1.1	Structure	97
6.1.2	Traversal	98
6.2	Properties	102
6.2.1	Proof of Theorem 6.2.1	103
6.2.2	Proof of Theorem 6.2.2	109
6.3	Representation in Answer Set Programming	111
6.3.1	Traces	112
6.3.2	Hierarchies of Reward Machines	112
6.3.3	Proof of Correctness	117
6.3.4	Determinism	119
6.3.5	Symmetry Breaking	120
6.4	Summary	121
7	Learning and Exploiting Hierarchies of Reward Machines	123
7.1	Exploiting Hierarchies of Reward Machines	123
7.1.1	Options	123
7.1.2	Algorithm	126
7.1.3	Implementation	131
7.2	Learning Hierarchies of Reward Machines from Traces	134
7.2.1	The Hierarchy of Reward Machines Learning Task	134
7.2.2	Solving the Hierarchy of Reward Machines Learning Task with ILASP	135
7.3	Interleaved Learning	136
7.3.1	Curriculum Learning	137
7.3.2	Interleaving Algorithm	138
7.4	Summary	139
8	Evaluation of Hierarchies of Reward Machines	141
8.1	Experimental Setup	141
8.1.1	Domains	141
8.1.2	Hyperparameters and Restrictions	144
8.1.3	Reporting Results	146
8.2	Learning Non-Flat Hierarchies of Reward Machines	146
8.2.1	Experimental Setup	147
8.2.2	Results	147
8.3	Learning Flat Hierarchies of Reward Machines	149
8.3.1	Experimental Setup	150
8.3.2	Results	153
8.4	Exploiting Handcrafted Hierarchies of Reward Machines	153
8.4.1	Experimental Setup	154
8.4.2	Results	154

8.5	Summary	154
9	Related Work	157
9.1	Finite-State Machines in Reinforcement Learning	157
9.1.1	Reward Machines	157
9.1.2	Other Formalisms	161
9.2	Hierarchical Reinforcement Learning	162
9.3	Curriculum Learning	164
9.4	Symmetry Breaking	164
10	Conclusion	167
10.1	Summary of Contributions	167
10.2	Future Work	168
	Bibliography	171
A	Reward Machines	185
A.1	Examples	185
B	Hierarchies of Reward Machines	189
B.1	Examples	189
B.2	Learning Non-Flat Hierarchies of Reward Machines	189

List of Figures

2.1	The agent-environment interaction in an MDP	8
2.2	Illustrations of the different types of optimality	14
2.3	An instance of the OFFICEWORLD domain	15
2.4	The agent-environment interaction in a labeled MDP	16
2.5	Reward machine for the COFFEE task in the OFFICEWORLD domain	17
2.6	The agent-environment interaction in a labeled MDP with a reward machine	18
3.1	The agent-environment interaction in a labeled MDP with a termination function	28
3.2	Reward machine for the COFFEE task in the OFFICEWORLD domain using our formalism	31
3.3	Predicate dependencies in the program of Proposition 3.3.1	36
3.4	Minimal reward machines for two OFFICEWORLD tasks	38
4.1	Shaping rewards produced by two distance metrics with $\gamma = 0.99$	61
5.1	Example of an OFFICEWORLD grid whose traces lead to an overgeneral RM for COFFEE	70
5.2	Learning curves for different RL algorithms in the OFFICEWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM)	75
5.3	Example of the impact that interleaved RM learning has on the learning curves of the COFFEE task	76
5.4	Learning curves for different combinations of instance sets ($\mathbb{I}_1^{10}, \mathbb{I}_1^{50}, \mathbb{I}_1^{100}$) and maximum episode lengths (100, 250, 500)	78
5.5	Learning curves for different combinations of instance sets ($\mathbb{I}_2^{10}, \mathbb{I}_2^{50}, \mathbb{I}_2^{100}$) and maximum episode lengths (100, 250, 500)	80
5.6	Learning curves for different RL algorithms in the CRAFTWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM)	87
5.7	The WATERWORLD domain	88
5.8	Learning curves for different RL algorithms in the WATERWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM)	91
6.1	An instance of the CRAFTWORLD domain.	96
6.2	A standard RM and an HRM for BOOK	96
6.3	Example to justify the need for the preliminary transformation algorithm	104
6.4	Results of flattening the HRM in Figure 6.2b	108
6.5	Example of an HRM used in the proof of Theorem 6.2.2	110

6.6	Predicate dependencies in the program of Proposition 6.3.1	117
6.7	Answer sets for each of the partitions in the program $P = \mathbb{A}(H) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$, where H is an HRM, \mathcal{R} is the set of general rules and λ^* is a label trace.	118
7.1	The core procedures involved in the policy learning algorithm that exploits HRMs	129
7.2	Examples of formula trees for different sets of labels	132
7.3	Overview of the LHRM algorithm	138
8.1	An instance of the CRAFTWORLD grid in the FRL setting	142
8.2	LHRM learning curves for CRAFTWORLD (FRL) and WATERWORLD (WD)	148
8.3	Learning curves for DQN, DRQN and LHRM in CRAFTWORLD	149
8.4	Learning curves for three CRAFTWORLD tasks using handcrafted HRMs	155
8.5	Learning curves for three WATERWORLD tasks using handcrafted HRMs	156
A.1	Reward machines for the OFFICEWORLD tasks in the ablation experiments from Section 5.2.5	186
A.2	Reward machines for some of the CRAFTWORLD tasks in Chapter 5	186
A.3	Reward machines for the WATERWORLD tasks in Chapter 5	187
B.1	Root reward machines for each of the CRAFTWORLD tasks in Chapter 8	190
B.2	Root reward machines for each of the WATERWORLD tasks in Chapter 8	191

List of Tables

5.1	Hyperparameters used in the OFFICEWORLD experiments	74
5.2	RM learning metrics for the OFFICEWORLD tasks using HRL_G	77
5.3	Hyperparameters used in the OFFICEWORLD ablation experiments	77
5.4	Total RM learning time in seconds for different combinations of instance sets (\mathbb{I}_1^{10} , \mathbb{I}_1^{50} , \mathbb{I}_1^{100}) and maximum episode lengths (100, 250, 500)	79
5.5	Number of examples needed to learn the last RM for different combinations of instance sets (\mathbb{I}_1^{10} , \mathbb{I}_1^{50} , \mathbb{I}_1^{100}) and maximum episode lengths (100, 250, 500)	79
5.6	Example length of the goal, dead-end and incomplete examples used to learn the last RM in the \mathbb{I}_1^{50} setting	79
5.7	Total RM learning time in seconds for different combinations of instance sets (\mathbb{I}_2^{10} , \mathbb{I}_2^{50} , \mathbb{I}_2^{100}) and maximum episode lengths (100, 250, 500)	81
5.8	Number of examples needed to learn the last RM for different combinations of instance sets (\mathbb{I}_2^{10} , \mathbb{I}_2^{50} , \mathbb{I}_2^{100}) and maximum episode lengths (100, 250, 500)	81
5.9	Example length of the goal, dead-end and incomplete examples used to learn the last RM in the \mathbb{I}_2^{50} setting	81
5.10	Number of examples (total, goal, dead-end and incomplete) needed to learn the last RM in the \mathbb{I}_2^{50} and $N = 250$ setting	81
5.11	RM learning metrics when traces are compressed or uncompressed	81
5.12	RM learning metrics when the proposition set is unrestricted or restricted to a particular task	82
5.13	Comparison of different RM learning metrics for the cases where RMs must be acyclic and where RMs can have cycles	82
5.14	RM learning metrics for different maximum number of edges from one state to another	83
5.15	Total RM learning time when symmetry breaking is disabled and enabled	83
5.16	Hyperparameters used in the CRAFTWORLD experiments	85
5.17	RM learning metrics for the CRAFTWORLD tasks using HRL_G	86
5.18	Hyperparameters used in the WATERWORLD experiments	89
5.19	RM learning metrics for the WATERWORLD tasks using HRL_G	89
6.1	List of CRAFTWORLD tasks	96
8.1	List of WATERWORLD tasks	143
8.2	List of hyperparameters and their values	145
8.3	Results of learning non-flat and flat HRMs using different methods	152

B.1	Results of LHRM in CRAFTWORLD for the default case	192
B.2	Results of LHRM in WATERWORLD for the default case	192
B.3	Results of LHRM in CRAFTWORLD with a restricted set of callable RMs	193
B.4	Results of LHRM in WATERWORLD with a restricted set of callable RMs	193
B.5	Results of LHRM in CRAFTWORLD without exploration using options	194
B.6	Results of LHRM in WATERWORLD without exploration using options	194

Acronyms

AI Artificial Intelligence

ASP Answer Set Programming

BFS Breadth-First Search

CDPI Context-Dependent Partial Interpretation

CNN Convolutional Neural Network

CRM Counterfactual Experiences for Reward Machines

DDQN Double Deep Q-Network

DFA Deterministic Finite Automata

DFS Depth-First Search

DNF Disjunctive Normal Form

DQN Deep Q-Network

DRL Deep Reinforcement Learning

DRQN Deep Recurrent Q-Network

EDSM Evidence Driven State Merging

FSM Finite-State Machine

HAM Hierarchical Abstract Machine

HRL Hierarchical Reinforcement Learning

HRM Hierarchy of Reward Machines

ILASP Inductive Learning of Answer Set Programs

ISA Induction of Subgoal Automata for Reinforcement Learning

LSTM Long Short-Term Memory

MDP Markov Decision Process

MLP Multilayer Perceptron

NMRDP Non-Markovian Reward Decision Process

POMDP Partially Observable Markov Decision Process

PTA Prefix Tree Acceptor

QRM Q-learning for Reward Machines

RL Reinforcement Learning

RM Reward Machine

RNN Recurrent Neural Network

SMDP Semi-Markov Decision Process

Notation

The following list only includes the notation that is recurrently used throughout the thesis.

Answer Set Programming and Inductive Learning of Answer Set Programs

A	an answer set
$\mathbb{A}(X)$	ASP representation of X (e.g., a trace)
\mathcal{B}	background knowledge of an ILASP task
$\mathcal{S}_{\mathfrak{M}}$	hypothesis space of an ILASP task, usually characterized by a set of predicate schemas \mathfrak{M} (called mode declarations)
\mathcal{E}^+	set of positive examples of an ILASP task
\mathcal{E}^-	set of negative examples of an ILASP task
e^{inc}	set of inclusions of a CDPI
e^{exc}	set of exclusions of a CDPI
e^{ctx}	context of a CDPI
\mathcal{H}	hypothesis (i.e., inductive solution of an ILASP task)

Functions

$\mathbb{1}[A]$	indicator function of event A
$\mathbb{E}[X]$	expectation of a random variable X
$\arg \max_a f(a)$	value of a at which $f(a)$ is maximized
$\arg \min_a f(a)$	value of a at which $f(a)$ is minimized

Logic

\perp	truth value <i>false</i>
\top	truth value <i>true</i>
$\text{DNF}_{\mathcal{P}}$	set of all possible DNF formulas over a set of propositions \mathcal{P}
$\text{DNF}(\varphi)$	DNF representation of formula φ
$ \varphi $	number of disjuncts of a DNF formula φ
$\phi_i \in \varphi$	the i -th conjunction/disjunct of a DNF formula φ
$\phi \in \varphi$	a conjunction/disjunct of a DNF formula φ

$\varphi(\mathcal{L})$	DNF formula formed by the disjuncts of a DNF formula φ satisfied by a label \mathcal{L}
$\phi \subseteq \phi'$	all disjuncts of a DNF formula ϕ appear in a DNF formula ϕ'

Markov Decision Processes

t	a discrete timestep
\mathcal{S}	set of states
\mathcal{A}	set of actions
p	probability transition function
r	reward function
γ	discount factor
\mathcal{P}	set of propositions
l	labeling function
τ	termination function
h^*	state-action history such that $* = G$ denotes a goal history, $* = D$ denotes a dead-end history, and $* = I$ denotes an incomplete history
\mathbf{s}	tuple $\langle s, s^T, s^G \rangle$, where s is a state, s^T indicates whether the history observed so far is terminal, and s^G indicates whether the history observed so far is a goal history
\mathcal{L}	label (i.e., a set of propositions)
\mathbb{L}	set of labels
λ^*	label trace such that $* = G$ denotes a goal trace, $* = D$ denotes a dead-end trace, and $* = I$ denotes an incomplete trace
Λ^*	set of label traces such that $* = G$ denotes a set of goal traces, $* = D$ denotes a set of dead-end traces, and $* = I$ denotes a set of incomplete traces
π	policy
R	return
v	state-value function
q	action-value function

Options

\mathcal{I}_ω	initiation set of an option ω
π_ω	policy of an option ω
β_ω	termination condition of an option ω
Π	policy over options
Ω	set of options

Reinforcement Learning Algorithms

α	learning rate
ϵ	exploration factor

θ	parameters of a deep Q-network
θ^-	parameters of a target deep Q-network
\mathcal{D}	experience replay buffer

Reward Machines

M	reward machine
H	hierarchy of reward machines
\mathcal{M}	set of reward machines
\mathcal{U}	set of states
δ_M	state-transition function of a reward machine M
δ_H	hierarchical transition function of a hierarchy of reward machines H
φ	logical transition function
r	reward-transition function
u^0	initial state
u^A	accepting state
u^R	rejecting state
\mathcal{U}^A	set of accepting states
\mathcal{U}^R	set of rejecting states
$M(\lambda)$	traversal of trace λ in reward machine M
$H(\lambda)$	hierarchy traversal of trace λ in a hierarchy of reward machines H
κ	maximum number of directed edges from a state to another
Γ	call stack
Φ	accumulated context
$\xi_{i,u,\Phi}$	exit condition for hierarchy state $\langle M_i, u, \Phi, \Gamma \rangle$, where Γ is an arbitrary call stack
h	height of a reward machine in a hierarchy of reward machines

Sets and Sequences

\mathbb{N}	the set of natural numbers
\mathbb{R}	the set of real numbers
$ \mathcal{X} $	number of elements in a finite set \mathcal{X}
$2^{\mathcal{X}}$	the power set of a finite set \mathcal{X}
$\Delta(\mathcal{X})$	probability simplex over a finite set \mathcal{X} , $\Delta(\mathcal{X}) = \{\mu \in \mathbb{R}^{\mathcal{X}} \mid \sum_x \mu(x) = 1, \mu(x) \geq 0 \ (\forall x)\}$
$U(\mathcal{X})$	uniform distribution over a set \mathcal{X}
\mathcal{X}^*	(possibly empty) sequences of elements from a finite set \mathcal{X}
\mathcal{X}^+	sequences of elements from a finite set \mathcal{X}
$X \oplus Y$	concatenation of two sequences X and Y
$X[t]$	t -th element in a sequence X
$X[t:]$	subsequence of a sequence X starting from the t -th element

Chapter 1

Introduction

Reinforcement learning (RL; Sutton and Barto, 2018) is one of the most promising and rapidly advancing fields in artificial intelligence (AI) research. Fundamentally, RL involves training agents with the aim of maximizing the cumulative reward they receive through repeated trial-and-error interactions with an environment. The fact that many tasks can be seen through the lens of maximizing reward makes RL a highly versatile decision-making paradigm. Indeed, it is hypothesized *“that all what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)”* (Sutton, 2004; Littman, 2018); in a similar vein, von Neumann and Morgenstern (1947) state that *“rational”* agents maximize expected utility. Examples of reinforcement learning agents include a cleaning robot rewarded based on the percentage of area cleaned, or a Sudoku-solving agent receiving +1 reward for solving a board.

Unlike methods in the supervised learning paradigm, reinforcement learning methods do not learn from extensive labeled data but from the experience generated by interacting with the environment. However, learning from interaction introduces several key *challenges*, as summarized by Abel (2020):

- **Exploration vs. Exploitation.** Trading off between decisions known to be good (exploitation) and decisions that may lead to higher cumulative rewards (exploration).
- **Credit assignment.** Determining what decisions are crucial to maximizing cumulative reward.
- **Generalization.** Enabling knowledge gained from past experiences to inform decisions in new situations.

In spite of these challenges, the combination of RL and the recent advances in deep learning (Goodfellow et al., 2016), known as deep reinforcement learning (DRL), has led to impressive successes across different fields, including human-level game playing (Mnih et al., 2015; Silver et al., 2016; Vinyals et al., 2019; Wurman et al., 2022), flying balloons in the stratosphere (Bellemare et al., 2020), beating humans at curling (Won et al., 2020), border testing (Bastani et al., 2021) and nuclear fusion (Degraeve et al., 2022). Nevertheless, most current DRL approaches suffer from important *shortcomings* inherent to deep learning methods (Kaelbling, 2020; Shanahan and Mitchell, 2022): low sample efficiency (i.e., lots of training data is required), limited transferability (i.e., learned knowledge is hardly reusable), and poor out-of-distribution generalization. There is widespread consensus that these shortcomings can be mitigated through *abstractions* (Ho et al., 2019; Konidaris,

2019; Kaelbling, 2020; Shanahan and Mitchell, 2022); namely, they facilitate generalization, support transferability, increase sample efficiency and, if structured, also enable composability (i.e., combining existing pieces of knowledge).

Finite-state machines (FSMs) are a widely exploited abstraction in AI, with applications in robotics (Brooks, 1989), games (Buckland, 2004), and automated planning (Bonet et al., 2009; Hu and De Giacomo, 2013; Segovia-Aguas et al., 2018). In reinforcement learning, they have been applied in several ways, including representing decision hierarchies (Parr and Russell, 1997), encoding external memory in partially observable environments (Meuleau et al., 1999; Toro Icarte et al., 2019), and interpreting an agent’s decisions (Koul et al., 2019). One type of FSM abstraction that has gained significant attention in recent years is the *reward machine* (RM; Toro Icarte et al., 2018a, 2022), which represents a task’s reward function in terms of high-level events. The structure of an RM consists of states connected through edges labeled with (i) a condition representing a subgoal in terms of high-level events, and (ii) a reward given upon satisfying the condition. By revealing the structure of a task through an RM, agents can:

- **Learn policies over histories.** Generally, RL agents learn policies (i.e., mappings from observations to actions) based on the most recent observation; however, there are scenarios where reward depends on history (i.e., on the full interaction), such as in partially observable environments. In these cases, agents must learn policies over histories, which can be prohibitively expensive since histories can be arbitrarily long (Spaan, 2012). Reward machines constitute compact history representations in terms of high-level events; therefore, RL agents can learn policies over the states of the RM, which is far more efficient than capturing the full history. In the same vein, generalizing over compact histories represented by RMs is easier than over full histories.
- **Perform task decomposition.** Reward machines enable decomposing the captured task into subtasks. For instance, the task “deliver coffee to the office” can be broken down into two subtasks: going to the coffee machine for a coffee, and going to the office. In particular, Toro Icarte et al. (2018a) propose decomposing a task into one subtask per RM state; that is, learning a policy for each RM state. The policies depend on each other; hence, essentially, a single but decomposed policy is learned. The experience generated by a policy is leveraged to train all policies, improving sample efficiency compared to standard methods. Task decomposition is especially relevant in sparse reward tasks, where the reward is zero most of the time.

Overall, abstracting tasks through RMs enables tackling complex tasks efficiently and alleviates various shortcomings of DRL methods (i.e., sample efficiency, knowledge reusability, generalization). Nevertheless, the applicability and benefits of reward machines are limited by several factors:

1. **The complexity of handcrafting them.** Manually designing a reward machine for any given task is infeasible in practice.
2. **The exploitation at a single timescale.** Subtask policies learned by exploiting RMs at a single timescale have limited reusability since they depend on each other (i.e., all aim to maximize cumulative reward in the global task). Furthermore, these approaches are often less sample-efficient than those employing multiple timescales since rewards are more distant (Dietterich et al., 2008, Section 5.8).

3. **The lack of composability.** Reward machines cannot call each other; that is, an RM cannot be reused within other RMs. Therefore, designing or learning RMs from scratch is highly inefficient.

In this thesis, we devise methods to address these limitations, increasing the applicability of RMs and boosting their advantages. The thesis statement is as follows:

Thesis Statement. *The applicability of reward machines can be expanded and their benefits enhanced by learning them from traces, exploiting them at multiple timescales, and hierarchically composing them.*

In the remainder of this chapter, we detail our contributions (Sections 1.1–1.2), list the publications this thesis builds upon (Section 1.3), and outline the structure of the thesis (Section 1.4).

1.1 Learning and Exploiting Reward Machines

In the first part of the thesis, we describe methods for learning and exploiting RMs. In what follows, we summarize our contributions on both fronts and their joint evaluation.

On the *exploitation* side, we devise a method leveraging the fact that RMs are inherently amenable to learning at multiple timescales. Namely, the conditions labeling the edges constitute independently solvable subtasks (i.e., oblivious to the global task), and the agent learns to choose the best subtask to perform from each RM state; hence, decision-making is implemented at two timescales. Decomposing a task across multiple timescales enables reusability and induces denser rewards; consequently, sample efficiency is often increased (Dietterich et al., 2008, Section 5.8). In a similar vein, we propose methods for defining additional rewards based on the structure of the RM and apply them in the single-timescale approach by Toro Icarte et al. (2018a). We describe both approaches using the options framework (Sutton et al., 1999; Precup, 2001), a formalism for temporal abstraction in RL.

On the *learning* side, we represent RMs using a logic programming language and employ a state-of-the-art inductive logic programming system to learn these representations from traces of high-level events observed by the RL agent. To speed up learning, we devise a symmetry breaking mechanism that discards multiple equivalent RMs during the search for a solution. The learning of an RM is interleaved with the exploitation algorithm of choice: a new RM is learned when the currently exploited one does not cover a trace observed by the agent. The proposed interleaving scheme guarantees the RMs are minimal; that is, they have the fewest possible states required to cover the traces.

The proposed interleaving method is *evaluated* in several grid-world and continuous state space problems using different exploitation algorithms. We provide an in-depth empirical analysis of the RM learning performance in terms of the traces, the symmetry breaking, and specific restrictions imposed on the final learnable RM. For each class of RL problem, we show that exploiting learned RMs induces effective policies whose performance is comparable to that obtained by exploiting handcrafted RMs.

1.2 Learning and Exploiting Hierarchies of Reward Machines

In the second part of the thesis, we introduce a formalism for composing RMs hierarchically, along with methods for learning and exploiting these hierarchies. The core objectives are scaling up the learning of RMs and enabling exploitation at arbitrarily many timescales.

We enhance the abstraction power of RMs by defining *hierarchies of RMs (HRMs)*, where constituent RMs can call other RMs. By composing RMs hierarchically, the lack of usability of RMs within other RMs is addressed. Theoretically, we prove that any HRM can be transformed into an equivalent flat HRM that behaves exactly like the original RMs; besides, we show that under certain conditions, the equivalent flat HRM can have exponentially more states and edges.

To *exploit* HRMs, we extend our previously outlined two-timescale algorithm. This method treats each RM call as a subtask and thus exploits HRMs at arbitrarily many timescales, considering a richer range of increasingly abstract and reusable subtasks. Empirically, leveraging a handcrafted HRM enables faster convergence than an equivalent flat HRM, showing that hierarchically composing RMs improves sample efficiency.

To *learn* HRMs, we introduce a curriculum-based method that learns an HRM from traces for each task in a list. The HRMs are learned such that those for more complex tasks can reuse those for simpler tasks. The learning system and the interleaving scheme are analogous to those we propose for standard RMs. In line with the theory, our experiments reveal that decomposing an RM into several smaller RMs is crucial to make its learning feasible (i.e., the flat HRM cannot be efficiently learned from scratch) for two reasons:

1. The constituent RMs are simpler; that is, they have fewer states and edges since calls to other RMs are used.
2. The policies for previously learned RMs can be used to efficiently explore the environment in the search for traces in more complex tasks.

Moreover, we show that exploiting the learned HRMs is more effective than employing no external memory or neural memories to capture histories.

1.3 Publications

This thesis includes revised and expanded content from the following published works (listed in chronological order):

- **D. Furelos-Blanco**, M. Law, A. Russo, K. Broda, and A. Jonsson. Induction of Subgoal Automata for Reinforcement Learning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 3890–3897, 2020.
- **D. Furelos-Blanco**, M. Law, A. Jonsson, K. Broda, and A. Russo. Induction and Exploitation of Subgoal Automata for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 70:1031–1116, 2021.
- **D. Furelos-Blanco**, M. Law, A. Jonsson, K. Broda, and A. Russo. Hierarchies of Reward Machines. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 10494–10541, 2023.

- Early versions of this work were presented at the *5th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)* and the *Planning and Reinforcement Learning (PRL) Workshop at the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*.

The following publications, in chronological order, have been also authored during the PhD program but are not presented as part of this thesis:

- **D. Furelos-Blanco** and A. Jonsson. Solving Multiagent Planning Problems with Concurrent Conditional Effects. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7594–7601, 2019.
- L. Ardon, **D. Furelos-Blanco**, and A. Russo. Learning Reward Machines in Cooperative Multi-Agent Tasks. In *Proceedings of the Neuro-Symbolic AI for Agent and Multi-Agent Systems (NeSyMAS) Workshop at the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2023.
- N. Grinsztajn, **D. Furelos-Blanco**, S. Surana, C. Bonnet, and T. D. Barrett. Winner Takes It All: Training Performant RL Populations for Combinatorial Optimization. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.
- C. Bonnet, D. Luo, D. Byrne, S. Surana, P. Duckworth, V. Coyette, L. I. Midgley, S. Abramowitz, E. Tegegn, T. Kalloniatis, O. Mahjoub, M. Macfarlane, A. P. Smit, N. Grinsztajn, R. Boige, C. N. Waters, M. A. Mimouni, U. A. Mbou Sob, R. de Kock, S. Singh, **D. Furelos-Blanco**, V. Le, A. Pretorius, and A. Laterre. Jumanji: a Diverse Suite of Scalable Reinforcement Learning Environments in JAX. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

1.4 Thesis Structure

The thesis is organized as follows. We start by describing relevant background material on reinforcement learning and inductive learning of answer set programs in Chapter 2. Our contributions are split into two parts with a similar structure. Part I presents our work on learning and exploiting reward machines:

- Chapter 3 formalizes the tasks and reward machines considered throughout Part I. We introduce a representation for RMs using a logic programming language, and devise rules for enforcing determinism and a unique indexing of the states and edges in RMs.
- Chapter 4 introduces two RL algorithms for exploiting RMs and a method for learning RMs from traces that leverages the representation in the previous chapter. The exploitation and learning mechanisms are then assembled in an algorithm that interleaves them.
- Chapter 5 performs an experimental evaluation of the presented methods across different domains.

Part II presents our work on learning and exploiting hierarchies of reward machines:

- Chapter 6 develops the formalism for hierarchically composing RMs, the core building block of Part II. Theoretical results regarding the equivalence of these hierarchies with standard RMs are proven. Analogously to Chapter 3, we introduce a representation for the hierarchies using a logic programming language.
- Chapter 7 describes an algorithm that exploits the structure of a hierarchy of reward machines at multiple timescales, and a method for learning hierarchies from traces. These methods are combined into a curriculum-based algorithm that interleaves them.
- Chapter 8 evaluates the proposed learning and exploitation methods in several domains.

The following two chapters wrap up the thesis. Chapter 9 discusses relevant related work, and Chapter 10 concludes by summarizing our contributions and outlining possible future work. Appendices A and B respectively extend Parts I and II with illustrations of reward machines and additional experimental results.

Chapter 2

Background

In this chapter, we survey the key concepts of reinforcement learning (Section 2.1) and inductive learning of answer set programs (Section 2.2) that lay the foundations for this thesis.

2.1 Reinforcement Learning

Reinforcement learning (RL; Sutton and Barto, 2018) is a learning framework where an *agent* learns how to act in an *environment* to maximize some *reward* signal. In the following sections, we describe the fundamental concepts in RL (Section 2.1.1) and its combination with deep neural networks (Section 2.1.2), as well as settings where an agent acts at multiple levels of abstraction (Section 2.1.3) and learns from history-dependent rewards (Section 2.1.4). Finally, we introduce reward machines (Section 2.1.5), the component around which this thesis is built.

2.1.1 Fundamentals

In this section, we introduce the key concepts in RL on which the thesis builds upon.

The Agent-Environment Interaction and Markov Decision Processes

The environment is often modeled as a Markov decision process (MDP; Bellman, 1957; Puterman, 1994). Formally, a finite MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ where \mathcal{S} is the finite set of states, \mathcal{A} is the finite set of actions, $p : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$ is the transition probability function, $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. The transition probability and reward functions constitute the *dynamics* or *model* of the environment.

Figure 2.1 illustrates the agent-environment interaction in a Markov decision process. At time t , the agent observes the environment’s state $s_t \in \mathcal{S}$, and performs an action $a_t \in \mathcal{A}$. The environment then outputs the next state $s_{t+1} \sim p(\cdot \mid s_t, a_t)$ and a reward $r_{t+1} = r(s_t, a_t, s_{t+1})$. We denote the state-action *history* at time t by $h_t = \langle s_0, a_0, \dots, s_{t-1}, a_{t-1}, s_t \rangle \in (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$. The interaction is called *episodic* if it lasts for a finite number of steps (i.e., there is a distinguished terminal state), or *continuing* if the number of steps is unlimited. In this thesis, we focus on the episodic setting.

The MDP dynamics depend only on the current state and the performed action, and not on the full state-action history. When the current state retains all relevant history information, it is said

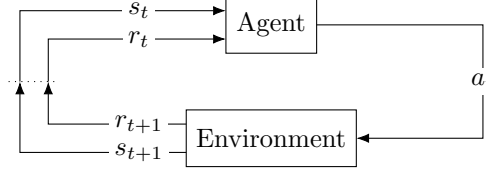


Figure 2.1: The agent-environment interaction in an MDP (Sutton and Barto, 2018).

to have the *Markov property*. Formally,

$$p(s_{t+1} \mid h_t, a_t) = p(s_{t+1} \mid s_t, a_t),$$

$$r(s_t, a_t, s_{t+1} \mid h_t, a_t) = r(s_t, a_t, s_{t+1}).$$

Policies and Value Functions

The objective of the agent is to learn a *policy* $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$, a mapping from states to probability distributions over actions, that maximizes the expected sum of discounted reward (or *return*), $R_t = \mathbb{E}[\sum_{k=t+1}^N \gamma^{k-t-1} r(s_{k-1}, a_{k-1}, s_k)]$, where N is a random variable representing the last step of an episode. The discount factor $\gamma \in [0, 1]$ determines the influence of future rewards. If $\gamma = 0$, the agent aims to maximize immediate rewards. As γ approaches 1, the agent considers future rewards more strongly. If $\gamma = 1$, all future rewards are considered equally.

Reinforcement learning algorithms often involve estimating *value functions*, which measure the quality of a given policy. There are two types of value functions:

- The *state-value function* $v^\pi : \mathcal{S} \rightarrow \mathbb{R}$ is the expected return from a given state under policy π . Formally,

$$v^\pi(s) = \mathbb{E}_\pi [R_t \mid s_t = s]$$

$$= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} p(s' \mid s, a) (r(s, a, s') + \gamma v^\pi(s')).$$

- The *action-value function* $q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the expected return from a given state-action pair and following policy π thereafter. Formally,

$$q^\pi(s, a) = \mathbb{E}_\pi [R_t \mid s_t = s, a_t = a]$$

$$= \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \left(r(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q^\pi(s', a') \right).$$

These recursively-defined equations are known as *Bellman equations*.

A policy π is better than another policy π' if and only if $v^\pi(s)$ is greater than $v^{\pi'}(s)$ for all states $s \in \mathcal{S}$. An *optimal policy* π^* is a policy that is better than or equal to all other policies. There might be several optimal policies, all of which share the same *optimal state-value function* v^* :

$$v^*(s) = \max_{\pi} v^\pi(s)$$

$$= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s' \mid s, a) (r(s, a, s') + \gamma v^*(s')).$$

Likewise, optimal policies also share the same *optimal action-value function* q^* , which is defined as:

$$\begin{aligned} q^*(s, a) &= \max_{\pi} q^{\pi}(s, a) \\ &= \sum_{s' \in \mathcal{S}} p(s' | s, a) \left(r(s, a, s') + \gamma \max_{a' \in \mathcal{A}} q^*(s', a') \right). \end{aligned}$$

An optimal policy can be induced from $q^*(s, a)$ by choosing the action with the highest value. These equations are known as *Bellman optimality equations*.

Algorithms

If the dynamics of the environment are known, an optimal policy can be computed using dynamic programming methods, such as value iteration (Bellman, 1957). However, in most practical settings, the dynamics are unknown; thus, the agent needs to interact with the environment to compute an optimal policy. There are two families of methods based on what the agent primarily estimates:¹

- If the model of the environment is estimated, the method is *model-based*. The learned model can then be used to estimate the optimal value function or the optimal policy by simulating the interaction in the MDP.
- If the model of the environment is not estimated, the method is *model-free*. There are two types of model-free methods:
 - If the optimal action-value function is estimated, the method is *value-based*.
 - If the optimal policy is estimated directly, the method is *policy-based*.

Algorithms are also categorized as either on-policy or off-policy based on their (i) *behavior policy*, which is used for acting, and (ii) *target policy*, which is the policy being learned. If the behavior and target policies are the same, the algorithm is *on-policy*; otherwise, the algorithm is *off-policy*.

In this thesis, we concentrate on Q-learning (Watkins and Dayan, 1992), a value-based method that learns an estimate \hat{q} of the optimal action-value q^* for state-action $\langle s, a \rangle \in \mathcal{S} \times \mathcal{A}$ pair, often initialized to 0. These estimates are updated using an experience $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ through the following rule:

$$\hat{q}(s_t, a_t) = \hat{q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s_{t+1}, a') - \hat{q}(s_t, a_t) \right),$$

where $\alpha \in [0, 1]$ is a learning rate, and the term $r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s_{t+1}, a')$ is called the *target*. Q-learning performs *bootstrapping* since the value function is updated from its own estimations. The algorithm is off-policy: the target policy is a *greedy policy*, which takes the action with the highest value at a given state, and the behavior policy is usually an ϵ -*greedy policy*, which acts greedily with probability $1 - \epsilon$ and selects an action uniformly at random with probability ϵ . The ϵ -greedy policy balances *exploitation* by acting greedily and *exploration* by acting randomly. If the agent always acts greedily, it will likely not visit important regions of the state space, and the resulting policy will be poor. Indeed, Q-learning requires the value for every state-action pair to be updated infinitely often for converging to the optimal action-value function; hence, balancing exploitation and exploration is crucial.

¹The families might be different in other works. For instance, Abel (2020) distinguishes between model-based, model-free and policy-based, and defines model-free methods as those that compute the action-value function.

2.1.2 Deep Reinforcement Learning

Tabular methods employ tables to store value estimates; for instance, Q-learning (see Section 2.1.1) keeps a table of size $\mathcal{S} \times \mathcal{A}$ to store the action-values. These methods suffer from two problems that strongly limit their applicability:

1. **Lack of scalability.** These methods cannot scale to large state and action spaces since the system’s memory inherently limits the table size, and learning a value for each state becomes too slow.
2. **Lack of generalization.** The estimations in observed states should generalize to similar unseen states.

To exemplify these limitations, let us consider an environment with a continuous state space. First, naively keeping a table of size $\mathcal{S} \times \mathcal{A}$ is unfeasible since \mathcal{S} is infinite. Second, it is unlikely that the agent will observe the same state several times, so it is important to generalize.

To address the previous problems, the optimal action-value function can be *approximated*. Function approximation with *deep learning* methods (Goodfellow et al., 2016) is the *de facto* standard given its success on complex tasks such as speech recognition (Hinton et al., 2012b), object recognition (Krizhevsky et al., 2012), and language translation (Sutskever et al., 2014). The combination of reinforcement learning and deep learning, known as *deep reinforcement learning* (DRL; Arulkumaran et al., 2017), endows agents with the ability to generalize and tackle high-dimensional states (e.g., images). In this thesis, we focus on a specific type of DRL method called *deep Q-networks*, which we describe (along with some extensions) in the following paragraphs.

Deep Q-networks (DQNs)

A deep Q-network (DQN; Mnih et al., 2015) employs a deep neural network with parameters θ to approximate the optimal action-value function $q^*(s, a; \theta)$. The updates are performed akin to Q-learning; however, learning these functions using non-linear function approximators, like deep neural networks, is unstable and may diverge since (i) states in successive experiences are strongly correlated, and (ii) the target is constantly changing. To address (i), DQNs employ a *replay buffer* (Lin, 1992), which stores the agent’s experiences $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$. The DQN is trained from minibatches of experiences sampled uniformly at random from the buffer, hence breaking the correlations. To address (ii), a separate DQN called the *target network* with parameters θ^- is used to compute the target. The target is prevented from continuous changes by periodically updating θ^- with the values of θ and keeping them fixed between updates. The Q-learning update uses the following loss function:

$$\mathbb{E}_{\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle \sim U(\mathcal{D})} \left[(y_t - q(s_t, a_t; \theta))^2 \right],$$

where $y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} q(s_{t+1}, a'; \theta^-)$ is the target and \mathcal{D} is the replay buffer.

The literature on extending DQNs to address some of their limitations is vast. In the following paragraphs, we describe those relevant to this thesis. We refer the reader to the work of Hessel et al. (2018), which describes many of these extensions and combines them into a single algorithm. However, some limitations of DQNs are better addressed using other DRL algorithms; for example, the DDPG algorithm (Lillicrap et al., 2016) is applicable in environments with continuous action spaces, whereas DQNs are not.

Double Deep Q-networks (DDQNs)

Methods based on Q-learning, including DQNs, can overestimate action-values due to the maximization used in the target (van Hasselt, 2010). A possible solution is to modify the target such that the action selection is decoupled from its evaluation. In particular, Double DQNs (DDQNs; van Hasselt et al., 2016) use the network with parameters θ for selection and the target network with parameters θ^- for evaluation, which results in the following target:

$$y_t = r_{t+1} + \gamma q(s_{t+1}, \arg \max_{a' \in \mathcal{A}} q(s_{t+1}, a'; \theta); \theta^-).$$

van Hasselt et al. (2016) experimentally show that DDQNs outperform DQNs.

Deep Recurrent Q-networks (DRQNs)

Different strategies have been proposed to apply DQNs in partially observable settings. The seminal work of Mnih et al. (2015) addressed partial observability in Atari games by stacking the last four frames. However, stacking an arbitrary number of frames (or any observations from the environment) is not practical in general. Alternatively, recurrent neural networks, such as LSTMs (Hochreiter and Schmidhuber, 1997), have been successfully used to summarize histories in complex videogames (Vinyals et al., 2019).

Deep Recurrent Q-networks (DRQNs; Hausknecht and Stone, 2015) extend DQNs by using LSTMs. LSTMs maintain a hidden state that summarizes the history of states perceived during an episode; thus, histories of experience must be sampled to compute the hidden state. Hausknecht and Stone (2015) propose two methods for updating DRQNs based on the length of the sampled histories:

- **Bootstrapped sequential updates.** The updates are performed from full episodes sampled uniformly at random. The hidden state is carried from the start of the episode, enabling a more accurate representation of it. However, the states in each episode are strongly correlated, which violates the DQN random sampling policy.
- **Bootstrapped random updates.** The updates are performed on fixed-length subsequences from episodes. Both episodes and subsequences are sampled uniformly at random. Although the violation of the DQN random sampling policy is less severe than in sequential updates, the updates might be imprecise since the hidden state may be initialized midway through an episode. Lample and Chaplot (2017) address the latter problem by only considering those states for which enough history has been provided in the update; that is, while the full subsequence is used to compute the hidden states, the updates are only applied for part of it. Alternatively, Kapturowski et al. (2019) propose to store the hidden state in the replay buffer and use it to initialize the network at training time, but the hidden state produced by an old network might differ from a newer one.

2.1.3 Hierarchical Reinforcement Learning

Hierarchical reinforcement learning (HRL; Barto and Mahadevan, 2003) methods enable agents to act at multiple levels of temporal abstraction by decomposing a task into subtasks. These methods have

several potential benefits. First, learning becomes simpler since learning policies for the subtasks should be easier than learning a single policy for the overall task. Second, the subtask policies can be reused across tasks. Third, exploration is more effective since the agent can move in the state space more efficiently by taking larger steps.

This thesis focuses on a specific HRL framework called options (Sutton et al., 1999; Precup, 2001), which we introduce in the following paragraphs. Other classic frameworks are hierarchical abstract machines (HAMs; Parr and Russell, 1997; Parr, 1998) and MAXQ (Dietterich, 2000). We also describe the different types of optimality that emerge in HRL.

Options

The *options* framework (Sutton et al., 1999; Precup, 2001) generalizes macro-actions (i.e., sequences of actions) to closed-loop policies. Formally, given an MDP $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, a (Markov) *option* is a tuple $\omega = \langle \mathcal{I}_\omega, \pi_\omega, \beta_\omega \rangle$, where $\mathcal{I}_\omega \subseteq \mathcal{S}$ is the option’s initiation set, $\pi_\omega : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ is the option’s policy, and $\beta_\omega : \mathcal{S} \rightarrow [0, 1]$ is the option’s termination condition. An option can be started in state $s \in \mathcal{S}$ if $s \in \mathcal{I}_\omega$, selects actions according to π_ω , and terminates in state $s' \in \mathcal{S}$ with probability $\beta_\omega(s')$. An action $a \in \mathcal{A}$ is an option that is applicable in any state ($\mathcal{I}_a = \mathcal{S}$), always performs a ($\pi_a(s) = a, \forall s \in \mathcal{S}$), and lasts one step ($\beta_a(s) = 1, \forall s \in \mathcal{S}$). Actions are often called *primitive actions* to distinguish them from other options.

Given a set of options Ω , several options could be started in a given state $s \in \mathcal{S}$. A *policy over options* (or *metapolicy*) $\Pi : \mathcal{S} \rightarrow \Delta(\Omega)$ determines what option to start in a state $s \in \mathcal{S}$. The resulting execution model is as follows. A policy over options selects an option ω in a given state $s \in \mathcal{S}$, which in turn selects primitive actions until it terminates. Decision-making hence happens at two timescales. The execution model can be extended to an arbitrary number of timescales by defining option policies on one level as policies over options on the next level.

The augmentation of an MDP with a given set of options Ω is a *semi-Markov decision process* (SMDP; Jewell, 1963; de Cani, 1964; Puterman, 1994). SMDPs model problems where actions can take variable amounts of time. Classical RL algorithms are easily extensible to SMDPs. SMDP Q-learning (Bradtke and Duff, 1994) extends Q-learning by interpreting the reward as the return accumulated during a temporally-extended action’s execution and appropriately discounting the target to reflect the execution time.² SMDP Q-learning can be used to learn a policy over options by treating options as indivisible units; that is, by following their policy until termination once they are selected. The update rule is:

$$q(s_t, \omega_t) = q(s_t, \omega_t) + \alpha \left(r + \gamma^k \max_{\omega' \in \Omega} q(s_{t+k}, \omega') - q(s_t, \omega_t) \right),$$

where k is the number of steps between s_t and s_{t+k} (i.e., the elapsed time between the initiation and termination of option ω_t), and $r = \sum_{j=1}^k \gamma^{j-1} r_{t+j}$ is the cumulative discounted reward over this time. SMDP Q-learning converges to an optimal policy over Ω under conditions similar to those for Q-learning (Parr, 1998). If the set of options includes the primitive actions (formally, $\mathcal{A} \subseteq \Omega$), the optimal policies over Ω are the same as the optimal policies over \mathcal{A} ; otherwise, the optimal policies over Ω can be suboptimal.

²Note that we use the term *temporally-extended action* instead of *option* since SMDP methods are also applicable in other HRL frameworks.

Intra-option learning methods (Sutton et al., 1998) enable learning multiple options from the experience of another. For instance, one-step intra-option Q-learning updates each option using an experience $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ generated by one of the options. Since experience accumulates faster, the convergence speed is often increased. Previously, Kaelbling (1993) proposed a similar method that updates multiple policies for achieving goals given an experience generated by one of them. These methods are off-policy, like standard Q-learning, since the learned policies differ from those generating the experiences.

Types of Optimality

Hierarchies can constrain the set of learnable policies and, therefore, may make learning an optimal policy impossible. For instance, in the options framework, an optimal policy is guaranteed to be in the set of learnable policies as long as any primitive action can be selected after each step. In cases where global optimality is not attainable, we aim to characterize the optimality of the policies consistent with the hierarchy. Dietterich (2000) identifies two types of optimality in hierarchical reinforcement learning: *hierarchical optimality* and *recursive optimality*. We base our descriptions on those by Dietterich (2000) and Ghavamzadeh (2005):

- A policy is *hierarchically optimal* if it achieves the highest cumulative reward among all policies consistent with the given hierarchy; that is, the policy for the entire hierarchy is a global optimum consistent with the given hierarchy, while the policies of the subtasks are not necessarily optimal.
- A policy is *recursively optimal* if it is optimal given the policies of its subtasks. The policy for each subtask is locally optimal and independent from higher-level tasks; thus, the policies are highly reusable across different tasks.

Example 2.1.1. *Dietterich (2000) illustrates the differences between the different types of optimality through the grids in Figure 2.2. The agent starts somewhere in the left room and must reach location G in the right room. There is a high-level subtask per room, each using the actions: up (\uparrow), right (\rightarrow) and down (\downarrow). The subtask for the right room is GO-TO-GOAL, which terminates when the agent reaches G. We consider two possible subtasks for the left room:*

- EXIT-ROOM, which terminates when the agent moves to the right room through either the upper or lower doors. Figure 2.2a displays a recursively optimal policy since the subtask policies are both locally optimal: the agent leaves the left room through the shortest path to a door, and reaches G in the right room through the shortest path to it. The policy is not globally optimal since the agent might exit the left room through the lower door, which is suboptimal from the global task perspective. The policy is not hierarchically optimal either: the EXIT-ROOM subtask allows the agent to leave the left room through any door, but using the upper door is the best choice since the agent needs fewer steps to reach G. Indeed, Figure 2.2b shows a hierarchically optimal policy where the agent always leaves through the upper room; in this case, the policy is also globally optimal since the agent completes the global task in the fewest steps, but it is not recursively optimal since the agent does not follow the shortest path to a door in the left room.
- EXIT-ROOM-THROUGH-LOWER-DOOR, which terminates when the agent moves to the right room through the lower door. Figure 2.2c illustrates a hierarchically optimal policy because it is an

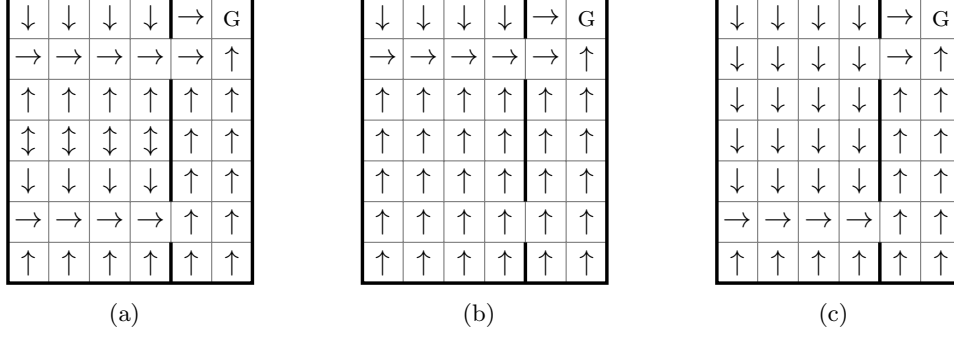


Figure 2.2: Illustrations of the different types of optimality. See Example 2.1.1 for details.

optimum consistent with the given hierarchy; that is, the hierarchy constrains the policy such that the agent can only leave using the lower door. The policy is also recursively optimal since the policies for both rooms are locally optimal. However, it is not globally optimal.

2.1.4 Non-Markovian Reward Decision Processes

The dynamics of an environment modeled as an MDP are assumed to be Markovian; that is, the dynamics solely depend on the current state and the performed action (see Section 2.1.1). However, making such an assumption is unrealistic (Whitehead and Lin, 1995): in real-world settings, the true state of the environment is often hidden (e.g., due to insufficient or faulty sensors). In addition, Markovian rewards cannot express certain types of tasks (Abel et al., 2021).

In this thesis, we focus on scenarios where the reward function is non-Markovian, also known as *non-Markovian reward decision processes* (NMRDPs; Bacchus et al., 1996). Formally, the non-Markovian reward function is defined as $r : (\mathcal{S} \times \mathcal{A})^+ \times \mathcal{S} \rightarrow \mathbb{R}$; that is, the reward obtained by the agent at each step depends on the state-action history. The transition probability function remains Markovian. We exemplify non-Markovian rewards through a simple domain.

Example 2.1.2. *The OFFICEWORLD (Toro Icarte et al., 2018a), illustrated in Figure 2.3, is a 12×9 grid containing different special locations. The state space $\mathcal{S} = \{0, \dots, 11\} \times \{0, \dots, 8\}$ is determined by the grid positions. The agent, depicted by \hat{x} , can move in the four cardinal directions; that is, the action space is $\mathcal{A} = \{\text{up, down, left, right}\}$. The agent always moves in the intended direction (i.e., actions are deterministic), and remains in the same location if it moves towards a wall. The tasks consist of visiting a sequence of locations while avoiding the decorations (*). A reward of 1 is given when the locations in the sequence have been visited; otherwise, the reward is 0. We consider the following three tasks:*

- **COFFEE:** *go to the coffee location (☕) followed by the office (o).*
- **COFFEEMAIL:** *go to the coffee location (☕) and the mail location (✉) in any order, followed by the office (o).*
- **VISITABCD:** *go to locations A, B, C and D in order.*

Rewards are non-Markovian since they are determined by the history of states rather than the current state alone. For instance, the state-action history $h = \langle \langle 4, 6 \rangle, \text{left}, \langle 3, 6 \rangle, \text{right}, \langle 4, 6 \rangle, \text{down}, \langle 4, 5 \rangle,$

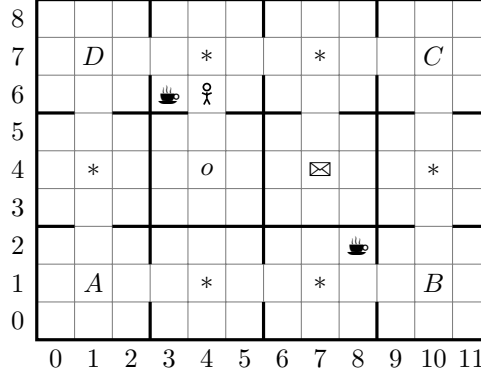





Figure 2.3: An instance of the OFFICEWORLD domain (Toro Icarte et al., 2018a).



down, $\langle 4, 4 \rangle$ yields a reward of 1 in COFFEE since location  is visited before location *o*.

Non-Markovian rewards are closely related to partial observability. Environments involving partial observability are formalized as *partially observable MDPs* (POMDPs; Åström, 1965; Kaelbling et al., 1998; Hasinoff, 2003; Spaan, 2012), which hide the state from the agent and instead provide an *observation* at each step. While the reward function of the POMDP is Markovian over the (hidden) states, the rewards become non-Markovian from the agent perspective since the observations are non-Markovian. Therefore, NMRDPs and POMDPs are similar in that an agent must learn a policy from non-Markovian reward signals; however, the transition function of NMRDPs is Markovian, whereas the transition function over observations in POMDPs might not be. In this thesis, we build upon the NMRDP formalism, but the methods introduced here could be framed in the context of POMDPs as long as the transitions over observations are Markovian; indeed, some of our original work used the latter (Furelos-Blanco et al., 2021). We refer the reader to the works by Toro Icarte et al. (2019, 2023) for POMDPs with non-Markovian transitions over observations.

There are different kinds of approaches to dealing with non-Markovian rewards. The straightforward option is to learn *memoryless policies* by applying standard RL algorithms for MDPs, such as Q-learning; however, these approaches are limited, as illustrated in the following example.

Example 2.1.3. *Given the COFFEE task in the OFFICEWORLD domain, the agent in Figure 2.3 cannot make an accurate decision in position $\langle 4, 6 \rangle$. The policy at this position will be the same regardless of whether the agent has been in location  or not; thus, it cannot determine at any point in time whether it should go to  or not since it does not retain knowledge about history.*

Given the limitations of memoryless policies, the alternative is to learn *history-dependent policies*. The naive approach consists of learning policies over the entire state-action history. However, this is not practical since histories can grow arbitrarily, and it is difficult to generalize from them (Spaan, 2012). Scalable approaches extend the state of the environment with variables or compact history representations that make reward Markovian (Whitehead and Lin, 1995). We depict an example of them below.

Example 2.1.4. *To make rewards Markovian from the agent’s perspective in the COFFEE task for OFFICEWORLD, the state can be extended with a Boolean variable that indicates whether the agent has been in location . Given the instance from Figure 2.3, the policy can determine to visit location  if the agent has not been there before, and visit location *o* otherwise.*

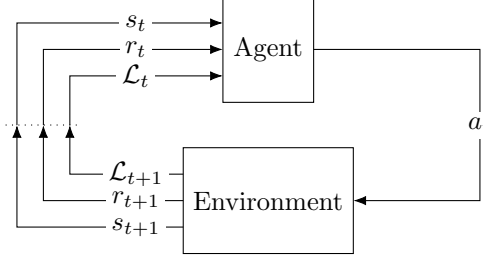


Figure 2.4: The agent-environment interaction in a labeled MDP.

In this thesis, we compactly encode history in terms of either (i) the states perceived by the agent, or (ii) high-level propositional events. For the former, we employ recurrent neural networks such as DRQNs (see Section 2.1.2). For the latter, we consider the environments to be modeled as *labeled MDPs* (Fu and Topcu, 2014; Xu et al., 2020), a type of NMRDP that provides agents with a set of high-level propositional events observed at the current state. Formally, a labeled MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma, \mathcal{P}, l \rangle$, where:

- \mathcal{S} , \mathcal{A} , p , r and γ are defined as for NMRDPs;
- \mathcal{P} is a finite set of *propositions* representing high-level events; and
- $l : \mathcal{S} \rightarrow 2^{\mathcal{P}}$ is a *labeling function* mapping states into propositions subsets (or *labels*).

Figure 2.4 illustrates the agent-environment interaction in labeled MDPs. Unlike MDPs, the agent observes a label $\mathcal{L} \in 2^{\mathcal{P}}$ generated by the labeling function l at each step.

The purpose of labeled MDPs is to represent history in terms of high-level propositional events. Given a history $h_t = \langle s_0, a_0, \dots, s_t \rangle \in (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$, a *label trace* (or *trace*, for short) $\lambda_t = \langle l(s_0), \dots, l(s_t) \rangle \in (2^{\mathcal{P}})^+$ assigns labels to all states in h_t . The goal is to find a *policy* $\pi : (2^{\mathcal{P}})^+ \times \mathcal{S} \rightarrow \Delta(\mathcal{A})$, a mapping from traces-states into probability distributions over actions that maximizes the expected return. However, this is only possible if label traces are faithful representations of history; that is, the reward can be written in terms of traces instead of history. Formally, $r(h_t, a_t, s_{t+1}) = r(h_{t+1}) = r(\lambda_{t+1}, s_{t+1})$.

Example 2.1.5. The OFFICEWORLD tasks in Example 2.1.2 are formalized as labeled MDPs by:

1. Defining the set of propositions as $\mathcal{P} = \{\text{☒}, \text{☒}, o, A, B, C, D, *\}$; that is, there is a proposition for each special location in the grid.
2. Defining the labeling function as a mapping from a location (i.e., a state in OFFICEWORLD) into the set of propositions found in that location.

Given the instance from Figure 2.3 and the history $h = \langle \langle 4, 6 \rangle, \text{left}, \langle 3, 6 \rangle, \text{right}, \langle 4, 6 \rangle, \text{down}, \langle 4, 5 \rangle, \text{down}, \langle 4, 4 \rangle \rangle$, the resulting trace is $\lambda = \langle \{\}, \{\text{☒}\}, \{\}, \{\}, \{o\} \rangle$.

Even though histories are expressed in terms of label traces in labeled MDPs, it is still impractical to learn policies over full traces. However, traces can be compactly represented using temporally abstract structures, such as linear temporal logic (e.g., Bacchus et al., 1996; Toro Icarte et al., 2018b), and finite-state machines (e.g., Toro Icarte et al., 2018a, 2022). This thesis focuses on a specific case of finite-state machines called reward machines, described in Section 2.1.5.

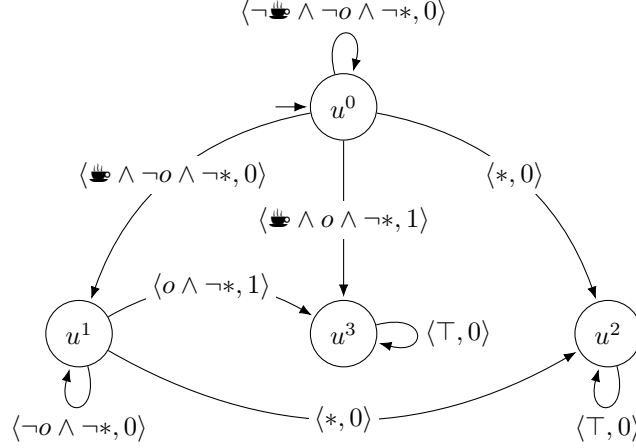


Figure 2.5: Reward machine for the COFFEE task in the OFFICEWORLD domain.

2.1.5 Reward Machines

Reward machines (RMs; Toro Icarte et al., 2018a, 2022) are finite-state machines that represent reward functions in terms of high-level propositional events. Formally, a reward machine is a tuple $M = \langle \mathcal{U}, \mathcal{P}, \delta, r, u^0 \rangle$, where:

- \mathcal{U} is a finite set of states;
- \mathcal{P} is a finite set of propositions that constitutes the alphabet of the reward machine;
- $\delta : \mathcal{U} \times 2^{\mathcal{P}} \rightarrow \mathcal{U}$ is the deterministic state-transition function, which takes an RM state and a subset of propositions (or *label*) and returns an RM state;
- $r : \mathcal{U} \times 2^{\mathcal{P}} \rightarrow \mathbb{R}$ is the reward-transition function, which outputs the reward associated with a state-label pair; and
- $u^0 \in \mathcal{U}$ is the initial state of the RM.

The definition allows for arbitrary propositional logic formulas over \mathcal{P} on the transitions. To verify if a formula is satisfied by a label $\mathcal{L} \in 2^{\mathcal{P}}$, \mathcal{L} is used as a truth assignment where propositions in \mathcal{L} are true, and false otherwise; for example, $\{\text{☿}, o\} \models \text{☿} \wedge o \wedge \neg *$.

Toro Icarte et al. (2018a, 2022) generalize these RMs by redefining the reward-transition function such that a reward function is returned instead of a scalar; formally, given an MDP, the reward-transition function becomes $r : \mathcal{U} \times 2^{\mathcal{P}} \rightarrow [\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}]$. In this thesis, we build upon the more specific version defined above.

Example 2.1.6. Figure 2.5 illustrates the RM for the COFFEE task in OFFICEWORLD. The edges are labeled by $\langle \varphi, r \rangle$ pairs, where φ is a propositional logic formula over $\mathcal{P} = \{\text{☿}, \boxtimes, o, A, B, C, D, *\}$ and r is a reward. The formula $\text{☿} \wedge \neg o \wedge \neg *$ accounts for 32 proposition sets, each the union of $\{\text{☿}\}$ with a subset of $\{\boxtimes, A, B, C, D\}$. The RM covers both (i) the case where ☿ and o share the same location (i.e. $\{\text{☿}, o\}$ is observable) through a direct path from u^0 to u^3 , and (ii) the case where ☿ and o are in different locations through the path via u^1 . The state-transition function is deterministic since no label can simultaneously satisfy transitions to two different states.

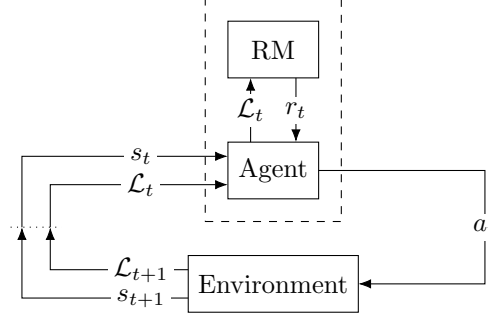


Figure 2.6: The agent-environment interaction in a labeled MDP with a reward machine.

Figure 2.6 shows the agent-environment interaction in labeled MDPs when an RM is used. The reward structure is revealed to the agent through the RM during the interaction. Starting from the RM's initial state, the agent moves through the RM according to the state-transition function and receives rewards according to the reward-transition function. Given an RM M and a trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$, an *RM traversal* $M(\lambda) = \langle v_0, v_1, \dots, v_{n+1} \rangle$ is a unique sequence of RM states such that:

1. $v_0 = u^0$, and
2. $\delta(v_i, \mathcal{L}_i) = v_{i+1}$ for $i = 0, \dots, n$.

Example 2.1.7. Given the RM in Figure 2.5 and the trace $\lambda = \langle \{\}, \{\text{☛}\}, \{\}, \{\}, \{o\} \rangle$, the resulting traversal is $\langle u^0, u^0, u^1, u^1, u^1, u^3 \rangle$.

Reward machines are compact representations of traces. The RM states correspond to different stages in completing a task; indeed, as shown in the RM traversal, a trace can be expressed as a sequence of RM states. For instance, the state u_1 in Figure 2.5 indicates that the agent has been to location ☛ and ignores everything else in the trace. Therefore, rewards become Markovian if defined over $\mathcal{S} \times \mathcal{U}$; that is, if the labeled MDP states \mathcal{S} are augmented with the task completion information provided by the RM states \mathcal{U} .

Given that the reward is Markovian over $\mathcal{S} \times \mathcal{U}$, an agent can simply learn a policy over $\mathcal{S} \times \mathcal{U}$ using standard RL methods, such as Q-learning. However, RMs provide agents with a task's reward structure, which enables task decomposition and, hence, increased sample-efficiency. In what follows, we describe some of these methods.

Q-learning for Reward Machines (QRM)

Q-learning for reward machines (QRM; Toro Icarte et al., 2018a) learns an action-value function over $\mathcal{S} \times \mathcal{U}$. The value function is distributed among the states in the RM; that is, QRM maintains an action-value function for each state in the RM. Given an experience $\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle$, QRM updates the action-value function $q_u : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ of an RM state $u \in \mathcal{U}$ using the following rule:

$$q_u(s_t, a_t) = q_u(s_t, a_t) + \alpha \left(r(u, \mathcal{L}_{t+1}) + \gamma \max_{a' \in \mathcal{A}} q_{u'}(s_{t+1}, a') - q_u(s_t, a_t) \right), \quad (2.1)$$

where $u' = \delta(u, \mathcal{L}_{t+1})$ is the next RM state. Even though the action-value functions are distributed, these are coupled through the target in the update rule; that is, the action-value function of an RM

state u depends on that of the next RM state u' , hence QRM effectively learns a value function over $\mathcal{S} \times \mathcal{U}$. QRM performs off-policy learning by applying the update above on all states in the RM given a single experience. In the tabular case, QRM converges to an optimal policy in the limit (Toro Icarte et al., 2018a, Theorem 4.1).

QRM is extensible to the function approximation case by approximating the action-value functions using DQNs. For each RM state $u \in \mathcal{U}$, there is a network with parameters θ_u and a target network with parameters θ_u^- . The DQN for a state $u \in \mathcal{U}$ is updated using the following loss function:

$$\mathbb{E}_{\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle \sim U(\mathcal{D})} \left[\left(r(u, \mathcal{L}_{t+1}) + \gamma \max_{a' \in \mathcal{A}} q_{u'}(s_{t+1}, a'; \theta_{u'}^-) - q_u(s_t, a_t; \theta_u) \right)^2 \right],$$

where $u' = \delta(u, \mathcal{L}_{t+1})$ is the next RM state, and \mathcal{D} stores the experiences $\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle$ observed during the agent-environment interaction (i.e., experiences are shared by the DQNs).

Counterfactual Experiences for Reward Machines (CRM)

Counterfactual experiences for reward machines (CRM; Toro Icarte et al., 2022) is a generalization of the idea underlying QRM: use the experiences generated by the policy on the current RM state to update the policy on other RM states. Formally, there is a single policy $\pi : \mathcal{S} \times \mathcal{U} \rightarrow \Delta(\mathcal{A})$ conditioned on the current environment state and the current RM state. Given an experience $\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle$, a set of synthetic experiences is created by counterfactual reasoning, i.e. simulating that \mathcal{L}_{t+1} was observed in each of the states of the RM:

$$\{ \langle s_t, u, a_t, r(u, \mathcal{L}_{t+1}), s_{t+1}, \delta(u, \mathcal{L}_{t+1}) \rangle \mid u \in \mathcal{U} \}.$$

These synthetic experiences can generally be used by off-policy learning methods. We describe how this is done in Q-learning and DQNs.

CRM with Q-learning maintains a single action-value function $q : \mathcal{S} \times \mathcal{A} \times \mathcal{U} \rightarrow \mathbb{R}$. Each synthetic experience $\langle s_t, u, a_t, r_{t+1}, s_{t+1}, u' \rangle$ is used to update the action-value function using the following rule:

$$q(s_t, a_t, u) = q(s_t, a_t, u) + \alpha \left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} q(s_{t+1}, a', u') - q(s_t, a_t, u) \right).$$

In the tabular case, like QRM, CRM with Q-learning converges to an optimal policy in the limit (Toro Icarte et al., 2022, Theorem 4.1).

Akin to QRM, CRM is extensible to the function approximation case by approximating the value function through a DQN with parameters θ and a target DQN with parameters θ^- . After each step, the set of experiences mentioned above is pushed to the replay buffer \mathcal{D} . The DQN is updated using the following loss function:

$$\mathbb{E}_{\langle s_t, u, a_t, r_{t+1}, s_{t+1}, u' \rangle \sim U(\mathcal{D})} \left[\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} q(s_{t+1}, a', u'; \theta^-) - q(s_t, a_t, u; \theta) \right)^2 \right].$$

CRM with Q-learning differs from QRM in that a single action-value function is maintained,

which makes the implementation easier and more efficient, specially in the function approximation case as noted by Toro Icarte et al. (2022).

2.2 Inductive Learning of Answer Set Programs

Inductive logic programming (Muggleton, 1991) is a learning paradigm where the goal is to learn a hypothesis in the form of a logic program, which together with a given background knowledge explains a set of examples. In the following sections, we describe answer set programming (Section 2.2.1), the logic programming language we employ, and the ILASP system for learning answer set programs (Section 2.2.2).

2.2.1 Answer Set Programming

Answer set programming (ASP; Gelfond and Lifschitz, 1988) is a declarative programming language for knowledge representation and reasoning. An ASP problem is expressed in a logical format and the models (called answer sets) of its representation provide the solutions to that problem. In the paragraphs below, we describe the main ASP concepts used in the thesis based on the materials by Law (2015).

Syntax

An *atom* is an expression of the form $p(t_1, \dots, t_n)$ where p is a *predicate* symbol of arity n and t_1, \dots, t_n are *terms*. If $n = 0$, we omit the parentheses. In this thesis, a term can be either a variable or a constant. By convention, variables are denoted using upper case (e.g., X or Y), while constants are written in lower case (e.g., `coffee` or `mail`). An atom is said to be *ground* if none of its terms is a variable. A *literal* is an atom a or its negation `not a`. The `not` symbol is called *negation as failure* (Clark, 1977).

An ASP *program* P is a set of rules. In this thesis, we assume that this set is formed by normal rules, choice rules, and constraints:

- A *normal rule* is of the form $h :- b_1, \dots, b_n$, where h is an atom constituting the *head* of the rule, and b_1, \dots, b_n are literals that constitute the *body* of the rule. A normal rule with an empty body is a *fact*.
- A *choice rule* is of the form $lb\{h_1, \dots, h_m\}ub :- b_1, \dots, b_n$. The *head* is $lb\{h_1, \dots, h_m\}ub$, where lb and ub are integers such that $lb \leq ub$ and h_1, \dots, h_m are atoms. If unspecified, lb and ub are 0 and ∞ , respectively. The *body* is constituted by literals b_1, \dots, b_n .
- A *constraint* is a normal rule with an empty head, i.e. of the form $:- b_1, \dots, b_n$.

Given a rule R , we denote by $body(R)$ the set of literals forming the body of R , $body^+(R)$ the set of all positive literals in the body of R , and $body^-(R)$ the set of all negative literals in the body of R .

Semantics

The *Herbrand base* of a program P is the set of all ground atoms that can be made from predicates and constants that appear in P . A *Herbrand interpretation* of a program P assigns every atom in

the Herbrand base of P to either true or false. It is often denoted as the set of atoms assigned to true. The *grounding of a program* P , denoted $ground(P)$, results from replacing each rule in the program with every ground instance of the rule. A Herbrand interpretation I satisfies a ground rule R if either the body is not satisfied by I or the head is satisfied by I . The body of a ground rule R is satisfied by a Herbrand interpretation I if and only if $body^+(R) \subseteq I$ and $body^-(R) \cap I = \emptyset$, whereas the head is satisfied differently depending on the type of rule:

- The head h of a ground normal rule is satisfied by I if $h \in I$.
- The head of a ground choice rule is satisfied by I if and only if the number of satisfied atoms in the head is between lb and ub (both included), i.e., $lb \leq |I \cap \{h_1, \dots, h_m\}| \leq ub$.
- The head of a ground constraint is unsatisfiable.

For any program P , a Herbrand interpretation I is a *Herbrand model* if every rule in $ground(P)$ is satisfied by I . A Herbrand model M of a program P is *minimal* if and only if there is no strict subset M' of M such that M' is also a Herbrand model of P .

Example 2.2.1. *The following shows an ASP program P consisting of two facts and two normal rules, and the resulting ground program $ground(P)$:*

$$P = \left\{ \begin{array}{l} p(X) :- \text{not } q(X), r(X). \\ q(X) :- \text{not } p(X), r(X). \\ r(1). \\ r(2). \end{array} \right\}, \quad ground(P) = \left\{ \begin{array}{l} p(1) :- \text{not } q(1), r(1). \\ p(2) :- \text{not } q(2), r(2). \\ q(1) :- \text{not } p(1), r(1). \\ q(2) :- \text{not } p(2), r(2). \\ r(1). \\ r(2). \end{array} \right\}.$$

The Herbrand base is $\{p(1), p(2), q(1), q(2), r(1), r(2)\}$. The Herbrand interpretation $I_1 = \{p(1), q(2), r(1), r(2)\}$ is a Herbrand model of P since all rules in $ground(P)$ are satisfied; in contrast, $I_2 = \{q(1), r(1), r(2)\}$ is not a model of P since the first rule in $ground(P)$ is not satisfied.

The *reduct* P^I of a ground program P with respect to a Herbrand interpretation I is built in four steps (Law et al., 2015b):³

1. Replace the heads of all constraints with \perp .
2. For each choice rule R :
 - if its head is not satisfied by I , replace its head with \perp , or
 - if its head is satisfied by I then remove R and for each atom h in the head of R such that $h \in I$, add the rule $h :- body(R)$.
3. Remove any rule R such that the body of R contains the negation of an atom in I .
4. Remove all negative literals from the body of any remaining rules.

³This is a non-standard form of building the reduct, but it is proven to be equivalent (Law et al., 2015b) to the standard definitions (e.g., Calimeri et al., 2020).

The reduct is always a ground program consisting only of *definite rules*, i.e. rules of the form $h :- b_1, \dots, b_n$ where b_1, \dots, b_n are atoms. For any program that consists only of definite rules, there is a *unique* minimal Herbrand model; hence, the reduct always has a unique minimal Herbrand model. A Herbrand interpretation I is an *answer set* of P if and only if I is the minimal Herbrand model of the reduct P^I .

Example 2.2.2. *Given the program P in Example 2.2.1, the Herbrand interpretation $I_1 = \{p(1), q(2), r(1), r(2)\}$ is an answer set of P . To obtain the answer set, the reduct of the ground program with respect to I_1 is first computed:*

$$\text{ground}(P)^{I_1} = \left\{ \begin{array}{l} p(1) :- r(1). \\ q(2) :- r(2). \\ r(1). \\ r(2). \end{array} \right\}.$$

The unique minimal model of the reduct is $\{p(1), q(2), r(1), r(2)\}$, which matches I_1 ; thus, I_1 is an answer set. The following three Herbrand interpretations are the other answer sets of P :

$$\{p(1), p(2), r(1), r(2)\}, \quad \{p(2), q(1), r(1), r(2)\}, \quad \{q(1), q(2), r(1), r(2)\}.$$

The Herbrand interpretation $I_2 = \{p(1), p(2), q(1), q(2), r(1), r(2)\}$ is not an answer set since the subset $\{r(1), r(2)\}$ is the minimal model of the reduct $\text{ground}(P)^{I_2} = \{r(1), r(2)\}$.

An ASP program P is *stratified* (Sergot, 2017) when there is a partition $P = P_0 \cup P_1 \cup \dots \cup P_n$ such that (i) P_i and P_j are disjoint for all $i \neq j$, (ii) the definition of every predicate p (all clauses with p in the head) is contained in one of the partitions P_i , and (iii) for each $1 \leq i \leq n$:

- if a predicate occurs positively in a clause of P_i then its definition is contained within $\bigcup_{j \leq i} P_j$, and
- if a predicate occurs negatively in a clause of P_i then its definition is contained within $\bigcup_{j < i} P_j$.

If an ASP program is stratified, then it has a *unique* answer set (Gelfond and Lifschitz, 1988, Corollary 1).

2.2.2 ILASP

ILASP (Inductive Learning of Answer Set Programs; Law et al., 2015a) is an inductive logic programming system for learning ASP programs from partial answer sets.

A *context-dependent partial interpretation* (CDPI; Law et al., 2016) is a pair $\langle \langle e^{inc}, e^{exc} \rangle, e^{ctx} \rangle$, where:

- $\langle e^{inc}, e^{exc} \rangle$ is a pair of sets of ground atoms, called a *partial interpretation*. We refer to e^{inc} and e^{exc} as the *inclusions* and *exclusions* respectively.
- e^{ctx} is an ASP program, called a *context*.

A program P *accepts* a CDPI $\langle \langle e^{inc}, e^{exc} \rangle, e^{ctx} \rangle$ if and only if there is an answer set A of $P \cup e^{ctx}$ such that $e^{inc} \subseteq A$ and $e^{exc} \cap A = \emptyset$.

An *ILASP task* (Law et al., 2016) is a tuple $T = \langle \mathcal{B}, \mathcal{S}_{\mathfrak{M}}, \langle \mathcal{E}^+, \mathcal{E}^- \rangle \rangle$ where:⁴

- \mathcal{B} is the ASP background knowledge, which describes a set of known concepts before learning;
- $\mathcal{S}_{\mathfrak{M}}$ is the set of ASP rules allowed in the hypotheses (i.e., learned programs), usually characterized by a set of predicate schemas \mathfrak{M} (called mode declarations); and
- \mathcal{E}^+ and \mathcal{E}^- are sets of CDPIs called, respectively, the positive and negative examples.

A hypothesis $\mathcal{H} \subseteq \mathcal{S}_{\mathfrak{M}}$ is an *inductive solution* of T if and only if:

1. $\forall e \in \mathcal{E}^+, \mathcal{B} \cup \mathcal{H}$ accepts e , and
2. $\forall e \in \mathcal{E}^-, \mathcal{B} \cup \mathcal{H}$ does not accept e .

Example 2.2.3. Let $T = \langle \mathcal{B}, \mathcal{S}_{\mathfrak{M}}, \langle \mathcal{E}^+, \mathcal{E}^- \rangle \rangle$ be an ILASP task where:

$$\begin{aligned} \mathcal{B} &= \{p : \text{not } q\}, \\ \mathcal{E}^+ &= \left\{ \begin{aligned} &\langle \{p\}, \{q\}, \{r\} \rangle, \\ &\langle \{q\}, \{p\}, \{r\} \rangle, \\ &\langle \{p\}, \{q\}, \emptyset \rangle \end{aligned} \right\}, \\ \mathcal{S}_{\mathfrak{M}} &= \left\{ \begin{aligned} &q. \quad q : \text{not } p. \\ &q : \text{not } p, r. \quad q : \text{not } p, r. \end{aligned} \right\}, \\ \mathcal{E}^- &= \left\{ \langle \{q\}, \{p\}, \emptyset \rangle \right\}. \end{aligned}$$

A candidate hypothesis is $\mathcal{H} = \{q : \text{not } p, r\}$. Now we have to check whether the positive examples are accepted, and the negative is not:

- For the positive examples $\langle \{p\}, \{q\}, \{r\} \rangle$ and $\langle \{q\}, \{p\}, \{r\} \rangle$, the program $\mathcal{B} \cup \mathcal{H} \cup \{r\}$ has two answer sets $A_1 = \{p, r\}$ and $A_2 = \{q, r\}$. Then, $\mathcal{B} \cup \mathcal{H} \cup \{r\}$ accepts the first example because $\{p\} \subseteq A_1$ and $\{q\} \cap A_1 = \emptyset$, and also the second one because $\{q\} \subseteq A_2$ and $\{p\} \cap A_2 = \emptyset$.
- For the positive example $\langle \{p\}, \{q\}, \emptyset \rangle$ and the negative example $\langle \{q\}, \{p\}, \emptyset \rangle$, the program $\mathcal{B} \cup \mathcal{H} \cup \emptyset$ has a single answer set $A'_1 = \{p\}$. Then, $\mathcal{B} \cup \mathcal{H} \cup \emptyset$ accepts the first example since $\{p\} \subseteq A'_1$ and $\{q\} \cap A'_1 = \emptyset$, and does not accept the second one since $\{q\} \not\subseteq A'_1$ (also, $\{p\} \cap A'_1 \neq \emptyset$).

Therefore, \mathcal{H} is an inductive solution of T . In contrast, $\mathcal{H}' = \{q\}$ is not an inductive solution. For instance, given the positive example $\langle \{p\}, \{q\}, \emptyset \rangle$, the program $\mathcal{B} \cup \mathcal{H}' \cup \emptyset$ has a single answer set, $A''_1 = \{q\}$. Then, this program does not accept the example because $\{p\} \not\subseteq A''_1$ (also, $\{q\} \cap A''_1 \neq \emptyset$), which causes \mathcal{H}' not to be an inductive solution.

ILASP, like other inductive logic programming systems, biases the search for inductive solutions towards those with the fewest literals. An inductive solution is called *optimal* if there is no inductive solution with fewer literals.

⁴The ILASP task definition given here captures the subset of ILASP used in this thesis.

Part I

Reward Machines

Chapter 3

Formalism of Reward Machines

Two aspects limit the applicability and benefits of reward machines. First, handcrafting a reward machine for any given task is often infeasible in practice. Second, despite exploiting task decomposition to improve sample efficiency, the QRM algorithm has two shortcomings: (i) it operates at a single timescale, making subtask policies hardly reusable since they are interdependent, and (ii) it starts updating action values only after a non-zero reward transition is satisfied, becoming less effective when the reward machine rarely emits non-zero rewards or high-level propositional events are infrequent.

In this part of the thesis, we propose mechanisms for addressing these previous limitations. We devise a method for learning RMs from traces, hence alleviating the need for human intervention. To speed up the learning process, we propose a symmetry breaking mechanism that removes equivalent RMs from the solution space. On the exploitation side, we introduce two methods for making the reward signal denser to improve sample efficiency. On the one hand, we present an HRL method for exploiting RMs at multiple timescales, i.e. dividing the task into simpler independently solvable subtasks. These subtasks are reusable, even within the same RM since they do not depend on the global task. On the other hand, we describe a reward shaping mechanism for QRM that provides additional rewards based on the RM’s structure. We ensemble learning and exploitation through an algorithm that interleaves them: a new RM is learned when the current one does not cover a trace observed by the agent. The algorithm is guaranteed to eventually find a minimal RM (i.e., with the fewest states) that captures the observable traces in the environment.

In this chapter, we formally define the type of *tasks* (Section 3.1) and *reward machines* (Section 3.2) considered throughout this part of the thesis. Next, we introduce an *ASP representation* for reward machines (Section 3.3). Finally, we devise a *symmetry breaking* mechanism (Section 3.4) that enforces a unique indexing of the states and edges in the reward machines.

3.1 Tasks

In this section, we formalize the tasks tackled in this thesis. We consider *episodic* tasks, where termination is determined by two possible situations: the achievement of the task’s goal, or the unfeasibility of achieving the goal (i.e., reaching a dead-end). In what follows, we provide a self-contained redefinition of the labeled MDPs introduced in Section 2.1.4 to allow for these two types

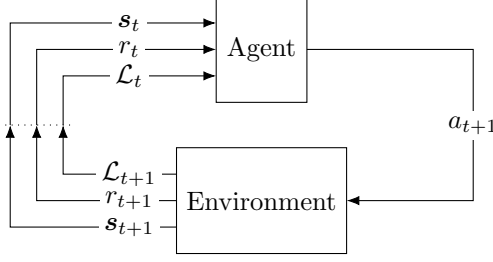


Figure 3.1: The agent-environment interaction in a labeled MDP with a termination function.

of termination.

Definition 3.1.1 (Labeled MDP). *A labeled MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma, \mathcal{P}, l, \tau \rangle$ where:*

- \mathcal{S} , \mathcal{A} , p , and γ are defined as for MDPs;
- $r : (\mathcal{S} \times \mathcal{A})^+ \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward function, which maps histories into rewards;
- \mathcal{P} is a finite set of propositions representing high-level events;
- $l : \mathcal{S} \rightarrow 2^{\mathcal{P}}$ is a labeling function mapping states into proposition subsets (or labels); and
- $\tau : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \rightarrow \{\perp, \top\} \times \{\perp, \top\}$ is the termination function, which maps histories into Boolean-pairs $\langle T, G \rangle$ where T indicates whether the history is terminal (i.e., the episode is completed) and G indicates whether the goal is achieved. If T is true and G is not, it means a dead-end has been reached.

Both the reward function and the termination function are non-Markovian since they are defined over histories. However, as shown in Section 2.1.4, these functions can be written in terms of label traces instead of histories. In the case of the termination function, given the history h_t , the corresponding trace λ_t and the state s_t at time t , the equivalence $\tau(h_t) = \tau(\lambda_t, s_t)$ holds.

Figure 3.1 illustrates the agent-environment interaction in the labeled MDPs considered in this thesis. At time t , the trace is $\lambda_t \in (2^{\mathcal{P}})^+$, and the agent observes a tuple $\mathbf{s}_t = \langle s_t, s_t^T, s_t^G \rangle$, where $s_t \in \mathcal{S}$ is the state and $\langle s_t^T, s_t^G \rangle = \tau(\lambda_t, s_t)$ is the termination information, with s_t^T and s_t^G indicating whether or not the history $\langle \lambda_t, s_t \rangle$ is terminal or a goal, respectively. The agent also observes a label $\mathcal{L}_t = l(s_t)$. If the history is non-terminal, the agent runs action $a_t \in \mathcal{A}$, and the environment transitions to state $s_{t+1} \sim p(\cdot \mid s_t, a_t)$. The agent then observes tuple \mathbf{s}_{t+1} and label \mathcal{L}_{t+1} , extends the trace as $\lambda_{t+1} = \lambda_t \oplus \mathcal{L}_{t+1}$, and receives reward $r_{t+1} = r(\lambda_{t+1}, s_{t+1})$.

Given the history h_t and the termination information $\langle s_t^T, s_t^G \rangle$ at time t , the history is one of the following:¹

1. A *goal history* h_t^G if $s_t^T = \top \wedge s_t^G = \top$ (i.e., the goal is achieved).
2. A *dead-end history* h_t^D if $s_t^T = \top \wedge s_t^G = \perp$ (i.e., a dead-end is reached).
3. An *incomplete history* h_t^I if $s_t^T = \perp$ (i.e., the history is not terminal).

¹With a slight abuse of notation, the symbols \perp (false) and \top (true) are used as both Boolean variables and Boolean formulas throughout the thesis.

Analogously, the label trace λ_t at time t is (i) a *goal trace* λ_t^G if h_t is a goal history, (ii) a *dead-end trace* λ_t^D if h_t is a dead-end history, and (iii) an *incomplete trace* λ_t^I if h_t is an incomplete history.

Example 3.1.1. *The termination in the OFFICEWORLD tasks (see Example 2.1.2) is defined such that histories ending on a decoration location are dead-end histories, whereas those completing the task specification are goal histories. The following table illustrates some example histories and traces for the COFFEE given the grid from Figure 2.3.*

Category	History	Trace
Goal	$\langle\langle 4, 6 \rangle, \text{left}, \langle 3, 6 \rangle, \text{right}, \langle 4, 6 \rangle, \text{down}, \langle 4, 5 \rangle, \text{down}, \langle 4, 4 \rangle\rangle$	$\langle\{\}, \{\text{☿}\}, \{\}, \{\}, \{o\}\rangle$
Dead-end	$\langle\langle 4, 6 \rangle, \text{up}, \langle 4, 7 \rangle\rangle$	$\langle\{\}, \{*\}\rangle$
Incomplete	$\langle\langle 4, 6 \rangle, \text{left}, \langle 3, 6 \rangle\rangle$	$\langle\{\}, \{\text{☿}\}\rangle$

For the remainder of the thesis, we assume the reward is 1 for goal histories and 0 otherwise; hence, our agents will learn from extremely sparse rewards. The task decomposition provided by reward machines, as shown throughout the thesis, enables such tasks to be tackled effectively. The assumption is mainly leveraged for the learning of the reward machines: we will only need to learn the conditions labeling the edges, but not the rewards. Further details are described in the following chapters.

Assumption 3.1.1. *The reward is 1 for goal histories and 0 otherwise.*

3.2 Reward Machines

The reward machines (RMs) in this thesis are different from the original (see Section 2.1.5) in three regards. First, the state transitions are determined by propositional logic formulas in disjunctive normal form (DNF). Second, there are two special states aligned with the redefinition of the labeled MDPs, which respectively denote the task’s goal achievement and the unfeasibility of achieving it (i.e., reaching a dead-end). Third, the reward-transition function is defined for state pairs instead of state-label pairs. We redefine the RMs to account for these differences as follows.

Definition 3.2.1 (Reward machine). *A reward machine is a tuple $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$, where:*

- \mathcal{U} is a finite set of states;
- \mathcal{P} is a finite set of propositions that constitutes the alphabet of the reward machine;
- $\varphi : \mathcal{U} \times \mathcal{U} \rightarrow \text{DNF}_{\mathcal{P}}$ is the logical transition function such that $\varphi(u, u')$ denotes the DNF formula over \mathcal{P} to be satisfied to transition from $u \in \mathcal{U}$ to $u' \in \mathcal{U}$;
- $r : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ is the reward-transition function, which outputs the reward associated with a state transition;
- $u^0 \in \mathcal{U}$ is the unique initial state;
- $u^A \in \mathcal{U}$ is the unique accepting state, which denotes the task’s goal achievement; and

- $u^R \in \mathcal{U}$ is the unique rejecting state, which denotes the unfeasibility of achieving the task's goal.

Expressing a transition condition $\varphi(u, u')$ as a DNF formula enables representing multiple edges between two states u and u' ; that is, each disjunct of $\varphi(u, u')$ labels a different edge between u and u' . We now introduce the *state-transition function* of an RM, which determines under what conditions a transition between two states takes place given a label.

Definition 3.2.2 (State-transition function). *The state-transition function $\delta_M : \mathcal{U} \times 2^{\mathcal{P}} \rightarrow \mathcal{U}$ of a reward machine M is defined in terms of its logical transition function φ . Formally,*

$$\delta_M(u, \mathcal{L}) = \begin{cases} u' & \text{if } \mathcal{L} \models \varphi(u, u'); \\ u & \text{if } \nexists u' \in \mathcal{U} \text{ such that } \mathcal{L} \models \varphi(u, u'). \end{cases}$$

That is, given a state $u \in \mathcal{U}$ and a label $\mathcal{L} \subseteq \mathcal{P}$, the next state is $u' \in \mathcal{U}$ if the associated formula $\varphi(u, u')$ is satisfied by \mathcal{L} ; otherwise, when no formula from u is satisfied by \mathcal{L} , the state remains unchanged.

The state-transition function must behave *deterministically*; that is, given a state $u \in \mathcal{U}$ and a label $\mathcal{L} \subseteq \mathcal{P}$, at most one formula is satisfied. Formally, there do not exist two states $u', u'' \in \mathcal{U}$ such that $\mathcal{L} \models \varphi(u, u'), \mathcal{L} \models \varphi(u, u'')$, and $u' \neq u''$. Determinism is guaranteed when all pairs of transitions from a given state to two different states are mutually exclusive; that is, a proposition appears positively on one edge and negatively on another. Next, we make some assumptions about the definition of the logical transition function and the reward-transition function.

Assumption 3.2.1. *The logical transition function φ of an RM is such that $\varphi(u, u) = \perp$ for all states $u \in \mathcal{U}$.*

Assumption 3.2.2. *The logical transition function φ of an RM is such that $\varphi(u, u') = \perp$ for $u \in \{u^A, u^R\}$ and $u' \in \mathcal{U}$.*

Assumption 3.2.3. *The reward-transition function is $r(u, u') = \mathbb{1}[u \neq u^A \wedge u' = u^A]$ as per Assumption 3.1.1; that is, a reward of 1 of one is given for transitions to the accepting state, and it is 0 otherwise.*

By Assumption 3.2.1, the state-transition function is such that the state remains unchanged if no formula to a *different* state is satisfied by a label; hence, self-loops are labeled ‘o.w.’ (otherwise) in any figures illustrating RMs. Likewise, the accepting and rejecting states are absorbing (i.e., cannot be left once reached) by Assumption 3.2.2. In addition, for simplicity, we omit rewards from the figures since all RMs share the same reward-transition function by Assumption 3.2.3.

Example 3.2.1. *Figure 3.2 shows the reward machine for the OFFICEWORLD’s COFFEE task using our formalism. The RM resembles that in Figure 2.5 except for (i) the states u^2 and u^3 , which respectively become the rejecting state u^R and the accepting state u^A , and (ii) the edges are labeled according to the following logical transition function φ :*

$$\begin{aligned} \varphi(u^0, u^1) &= \neg o \wedge \neg * & \varphi(u^0, u^A) &= o \wedge \neg * & \varphi(u^0, u^R) &= *, \\ \varphi(u^1, u^A) &= o \wedge \neg * & \varphi(u^1, u^R) &= *. \end{aligned}$$

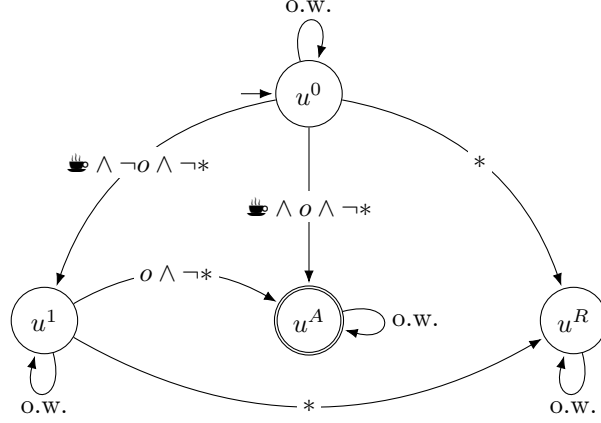


Figure 3.2: Reward machine for the COFFEE task in the OFFICEWORLD domain using our formalism.

For all absent pairs of states $\langle u, u' \rangle$, $\varphi(u, u') = \perp$. The transition function is deterministic because all pairs of outgoing transitions from a given state to two different states are mutually exclusive; for instance, the formulas $o \wedge \neg *$ and $*$ are mutually exclusive because $*$ appears negatively in the former and positively in the latter. In this case, there is not more than one edge between each pair of states; that is, $|\varphi(u, u')| = 1$ for all pairs of states.

The *RM traversal* is the same as that defined in Section 2.1.5 but using the state-transition function δ_M instead. We redefine the traversal to account for this change and define what it means for a trace to be accepted or rejected by an RM.

Definition 3.2.3 (RM traversal). *Given an RM M and a trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$, an RM traversal $M(\lambda) = \langle v_0, v_1, \dots, v_{n+1} \rangle$ is a unique sequence of RM states such that:*

1. $v_0 = u^0$, and
2. $\delta_M(v_i, \mathcal{L}_i) = v_{i+1}$ for $i = 0, \dots, n$.

An RM M accepts a trace λ if the traversal $M(\lambda) = \langle v_0, \dots, v_{n+1} \rangle$ is such that $v_{n+1} = u^A$ (i.e., the last state in the traversal is the accepting state). Analogously, M rejects λ if $v_{n+1} = u^R$ (i.e., the last state in the traversal is the rejecting state).

Example 3.2.2. The reward machine M in Figure 3.2 accepts $\lambda_1 = \langle \{\}, \{\text{coffee}\}, \{\}, \{\}, \{o\} \rangle$, rejects $\lambda_2 = \langle \{\}, \{*\} \rangle$, and does not accept or reject $\lambda_3 = \langle \{\}, \{\text{coffee}\} \rangle$ since the traversals are $M(\lambda_1) = \langle u^0, u^0, u^1, u^1, u^1, u^A \rangle$, $M(\lambda_2) = \langle u^0, u^0, u^R \rangle$ and $M(\lambda_3) = \langle u^0, u^0, u^1 \rangle$, respectively.

Having established how an RM evaluates (acceptance, rejection, or neither) a trace, we need to determine whether this evaluation complies with the trace type (goal, dead-end, or incomplete). For this purpose, we introduce the concept of *validity* with respect to a trace (i.e., whether the trace type matches the evaluation of the RM).

Definition 3.2.4 (Validity). *Given a trace λ^* , where $*$ $\in \{G, D, I\}$, a reward machine M is valid with respect to λ^* if one of the following holds:*

- M accepts λ^* and $*$ $= G$ (i.e., λ^* is a goal trace).

- M rejects λ^* and $*$ = D (i.e., λ^* is a dead-end trace).
- M does not accept nor reject λ^* and $*$ = I (i.e., λ^* is an incomplete trace).

Validity is crucial to prove the correctness of the ASP encoding of the RMs (see Section 3.3), as well as to learn RMs (see Chapter 4).

3.3 Representation in Answer Set Programming

In this section, we describe the representation in answer set programming (ASP) of traces (Section 3.3.1) and reward machines (Section 3.3.2), and prove its correctness (Section 3.3.3). We also introduce a rule set for verifying that the represented RM is deterministic (Section 3.3.4).

3.3.1 Traces

Traces are represented in terms of three atoms: $\mathbf{prop}(p, t)$ indicates that proposition $p \in \mathcal{P}$ is observed at step t , $\mathbf{step}(t)$ states that t is a step of the trace, and $\mathbf{last}(n)$ indicates that the trace ends at step n .

Definition 3.3.1 (ASP representation of a trace). *Given a trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$, $\mathbb{A}(\lambda)$ denotes the set of ASP facts that describe it:*

$$\begin{aligned} \mathbb{A}(\lambda) = & \{\mathbf{prop}(p, t). \mid 0 \leq t \leq n, p \in \mathcal{L}_t\} \cup \\ & \{\mathbf{step}(t). \mid 0 \leq t \leq n\} \cup \\ & \{\mathbf{last}(n). \}. \end{aligned}$$

Example 3.3.1. *The ASP representation of trace $\lambda = \langle \{a\}, \{\}, \{b, c\} \rangle$ is $\mathbb{A}(\lambda) = \{\mathbf{prop}(a, 0)., \mathbf{prop}(b, 2)., \mathbf{prop}(c, 2)., \mathbf{step}(0)., \mathbf{step}(1)., \mathbf{step}(2)., \mathbf{last}(2). \}$.*

3.3.2 Reward Machines

We here describe the ASP representation of reward machines. First, we explain the particular rules that characterize a given RM. Second, we introduce the rules representing how the traversal in any RM is performed for any trace.

Structure

We introduce two representations of the structure (i.e., states and logic transition function) of a specific reward machine, each with a different purpose:

- A *non-factual* representation, where transitions are expressed through rules (Definition 3.3.2), which is used by the traversal rules introduced later and learned in Chapter 4.
- A *factual* representation, where transitions are expressed through facts (Definition 3.3.3), which is used to verify whether the RM complies with structural properties such as determinism (Section 3.3.4).

We start defining and exemplifying the former, and continue by justifying the need for the latter.

Definition 3.3.2 (Non-factual ASP representation of a reward machine). *Given a reward machine $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$, $\mathbb{A}(M) = \mathbb{A}_{\mathcal{U}}(M) \cup \mathbb{A}_{\varphi}(M)$ denotes the set of ASP rules that describe it, where:*

$$\mathbb{A}_{\mathcal{U}}(M) = \{\text{state}(u) . \mid u \in \mathcal{U}\}$$

and

$$\mathbb{A}_{\varphi}(M) = \left\{ \begin{array}{l} \text{ed}(u, u', i). \\ \bar{\varphi}(u, u', i, T) :- \text{not prop}(p_1, T), \text{step}(T). \\ \vdots \\ \bar{\varphi}(u, u', i, T) :- \text{not prop}(p_n, T), \text{step}(T). \\ \bar{\varphi}(u, u', i, T) :- \text{prop}(p_{n+1}, T), \text{step}(T). \\ \vdots \\ \bar{\varphi}(u, u', i, T) :- \text{prop}(p_m, T), \text{step}(T). \end{array} \mid \begin{array}{l} u \in \mathcal{U} \setminus \{u^A, u^R\}, \\ u' \in \mathcal{U} \setminus \{u\}, \\ 1 \leq i \leq |\varphi(u, u')|, \\ \phi_i \in \varphi(u, u'), \\ \phi_i = p_1 \wedge \dots \wedge p_n \\ \wedge \neg p_{n+1} \wedge \dots \wedge \neg p_m \end{array} \right\}.$$

The rule set $\mathbb{A}(M)$ is formed by a set $\mathbb{A}_{\mathcal{U}}(M)$ representing the set of states of M , and a set $\mathbb{A}_{\varphi}(M)$ representing the logical transition function of M . The constituent rules are described as follows:

- Facts $\text{state}(u)$ indicate that u is an RM state.
- Facts $\text{ed}(u, u', i)$ indicate that there is a transition from state u to u' using edge i , where i is the i -th conjunction in the DNF formula $\varphi(u, u')$.²
- Normal rules whose head is $\bar{\varphi}(u, u', i, T)$ state that the transition from state u to state u' with edge i does not hold at step T . The body consists of a single (positive or negative) $\text{prop}(p, T)$ literal and an atom $\text{step}(T)$ indicating that T is a step.³

Note that $\bar{\varphi}$ represents the negation of the logical transition function φ . The reason for this choice comes from the fact that learning $\bar{\varphi}$ instead of φ makes the search space smaller, thus speeding up RM learning (see Chapter 4).

Example 3.3.2. *The non-factual ASP representation of the RM in Figure 3.2 is given by:*

$$\left\{ \begin{array}{ll} \text{state}(u^0). \text{state}(u^1). & \text{state}(u^A). \text{state}(u^R). \\ \text{ed}(u^0, u^1, 1). \text{ed}(u^0, u^A, 1). \text{ed}(u^0, u^R, 1). & \text{ed}(u^1, u^A, 1). \text{ed}(u^1, u^R, 1). \\ \bar{\varphi}(u^0, u^1, 1, T) :- \text{not prop}(\text{⚡}, T), \text{step}(T). & \bar{\varphi}(u^0, u^1, 1, T) :- \text{prop}(o, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^1, 1, T) :- \text{prop}(*, T), \text{step}(T). & \bar{\varphi}(u^0, u^A, 1, T) :- \text{not prop}(\text{⚡}, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^A, 1, T) :- \text{not prop}(o, T), \text{step}(T). & \bar{\varphi}(u^0, u^A, 1, T) :- \text{prop}(*, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^R, 1, T) :- \text{not prop}(*, T), \text{step}(T). & \bar{\varphi}(u^1, u^A, 1, T) :- \text{not prop}(o, T), \text{step}(T). \\ \bar{\varphi}(u^1, u^A, 1, T) :- \text{prop}(*, T), \text{step}(T). & \bar{\varphi}(u^1, u^R, 1, T) :- \text{not prop}(*, T), \text{step}(T). \end{array} \right\}.$$

²Remember each conjunction in the DNF formula $\varphi(u, u')$ represents a different edge between states u and u' .

³Variables are represented using upper case letters, which is the case of steps T here.

The non-factual representation introduced above represents the formulas labeling the edges as rules; however, consequently, we cannot easily verify structural properties (represented as constraints) over them. To address this problem, we introduce the factual representation, which results from mapping the $\bar{\varphi}$ rules into facts. Verifying structural properties of the RM through constraints over facts is more amenable than doing so over variable-dependent rules (e.g., the $\bar{\varphi}$ rules depend on the time variable T). We define the factual representation of an RM and exemplify it below.

Definition 3.3.3 (Factual ASP representation of a reward machine). *Given the ASP representation $\mathbb{A}(M)$ of a reward machine M , the factual ASP representation $\mathbb{A}^F(M)$ of M is the result of mapping the $\bar{\varphi}$ rules into $\text{pos}(u, u', i, p)$ and $\text{neg}(u, u', i, p)$ facts expressing that proposition p appears positively (resp. negatively) in the edge i from state u to state u' . Formally,*

$$\mathbb{A}^F(M) = \left\{ \begin{array}{l|l} \text{state}(u). & \text{state}(u). \\ \text{ed}(u, u', i). & \text{ed}(u, u', i). \\ \text{pos}(u, u', i, p_1). & \bar{\varphi}(u, u', i, T) :- \text{not prop}(p_1, T), \text{step}(T). \\ \vdots & \vdots \\ \text{pos}(u, u', i, p_n). & \bar{\varphi}(u, u', i, T) :- \text{not prop}(p_n, T), \text{step}(T). \\ \text{neg}(u, u', i, p_{n+1}). & \bar{\varphi}(u, u', i, T) :- \text{prop}(p_{n+1}, T), \text{step}(T). \\ \vdots & \vdots \\ \text{neg}(u, u', i, p_m). & \bar{\varphi}(u, u', i, T) :- \text{prop}(p_m, T), \text{step}(T). \end{array} \right\},$$

where the rules on the right hand side are those within $\mathbb{A}(M)$.

Example 3.3.3. *The following rules constitute the factual ASP representation built from the rule set in Example 3.3.2:*

$$\left\{ \begin{array}{l|l} \text{state}(u^0). \text{state}(u^1). & \text{state}(u^A). \text{state}(u^R). \\ \text{ed}(u^0, u^1, 1). \text{ed}(u^0, u^A, 1). \text{ed}(u^0, u^R, 1). & \text{ed}(u^1, u^A, 1). \text{ed}(u^1, u^R, 1). \\ \text{pos}(u^0, u^1, 1, \clubsuit). & \text{neg}(u^0, u^1, 1, o). \\ \text{neg}(u^0, u^1, 1, *). & \text{pos}(u^0, u^A, 1, \clubsuit). \\ \text{pos}(u^0, u^A, 1, o). & \text{neg}(u^0, u^A, 1, *). \\ \text{pos}(u^0, u^R, 1, *). & \text{pos}(u^1, u^A, 1, o). \\ \text{neg}(u^1, u^A, 1, *). & \text{pos}(u^1, u^R, 1, *). \end{array} \right\}.$$

The factual representation is convenient to verify structural properties, such as determinism (Section 3.3.4) and the compliance with a canonical indexing of states and edges (Section 3.4). Nevertheless, the non-factual representation is the one learned later (see Chapter 4) since it requires less grounding and sets the foundations for representing more complex FSMs in the future (see Chapter 10). As detailed later, our algorithm will learn a non-factual representation and map it to the factual one to verify the candidate RM properties. Since the non-factual representation is our choice for learning, the rules modeling RM traversals introduced next are defined over it.

General Rules

To check whether an RM accepts or rejects a trace, it is necessary to reason about how traces traverse RMs. For this purpose, we introduce a set of rules $\mathcal{R} = \mathcal{R}_\varphi \cup \mathcal{R}_\delta \cup \mathcal{R}_{\text{st}}$ defined over the

non-factual ASP representation of an RM. The subsets \mathcal{R}_φ and \mathcal{R}_δ define the rules related to the state-transition function:

- The first rule in \mathcal{R}_φ defines the logical transition function φ in terms of its negation $\bar{\varphi}$ and **ed** atoms. The second rule indicates that an outgoing transition from state **X** is taken at step **T**.

$$\mathcal{R}_\varphi = \left\{ \begin{array}{l} \varphi(\mathbf{X}, \mathbf{Y}, \mathbf{E}, \mathbf{T}) : \text{-not } \bar{\varphi}(\mathbf{X}, \mathbf{Y}, \mathbf{E}, \mathbf{T}), \mathbf{ed}(\mathbf{X}, \mathbf{Y}, \mathbf{E}), \mathbf{step}(\mathbf{T}). \\ \mathbf{out}\text{-}\varphi(\mathbf{X}, \mathbf{T}) : \text{-}\varphi(\mathbf{X}, \text{-}, \text{-}, \mathbf{T}). \end{array} \right\}$$

- The rules in \mathcal{R}_δ define the state-transition function δ in terms of φ as per Definition 3.2.2. The first rule states that the transition from **X** to **Y** at step **T** is active if the formula in an outgoing transition to **Y** is satisfied at that step. The second rule indicates that the transition from state **X** to itself at step **T** is active if no outgoing transition is active at that step.

$$\mathcal{R}_\delta = \left\{ \begin{array}{l} \delta(\mathbf{X}, \mathbf{Y}, \mathbf{T}) : \text{-}\varphi(\mathbf{X}, \mathbf{Y}, \text{-}, \mathbf{T}). \\ \delta(\mathbf{X}, \mathbf{X}, \mathbf{T}) : \text{-not out-}\varphi(\mathbf{X}, \mathbf{T}), \mathbf{state}(\mathbf{X}), \mathbf{step}(\mathbf{T}). \end{array} \right\}$$

The subset \mathcal{R}_{st} is used to define the RM traversal of the trace (that is, the sequence of visited RM states), and the criteria for accepting or rejecting a trace. The $\mathbf{st}(t, u)$ atoms indicate that a trace is in state u at step t . The first rule defines that the agent is in u^0 at step 0. The second rule determines that at step $\mathbf{T}+1$ the agent will be in state **Y** if it is in state **X** at step **T** and a transition between them is active at that step. The third (resp. fourth) rule indicates that the trace is accepted (resp. rejected) if the state at the trace's last step is u^A (resp. u^R).

$$\mathcal{R}_{\text{st}} = \left\{ \begin{array}{l} \mathbf{st}(0, u^0). \\ \mathbf{st}(\mathbf{T}+1, \mathbf{Y}) : \text{-}\mathbf{st}(\mathbf{T}, \mathbf{X}), \delta(\mathbf{X}, \mathbf{Y}, \mathbf{T}). \\ \mathbf{accept} : \text{-}\mathbf{last}(\mathbf{T}), \mathbf{st}(\mathbf{T}+1, u^A). \\ \mathbf{reject} : \text{-}\mathbf{last}(\mathbf{T}), \mathbf{st}(\mathbf{T}+1, u^R). \end{array} \right\}$$

3.3.3 Proof of Correctness

We prove the correctness of the ASP representation in the following lines. The non-factual representation of the RMs is employed; indeed, the previously introduced traversal rules are defined over this representation. The following result is crucial to prove the correctness of the RM learning task introduced in Chapter 4.

Proposition 3.3.1 (Correctness of the ASP encoding). *Given a finite trace λ^* , where $*$ $\in \{G, D, I\}$, and a reward machine M that is valid with respect to λ^* , the program $P = \mathbb{A}(M) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$ has a unique answer set A and (i) **accept** $\in A$ if and only if $* = G$, and (ii) **reject** $\in A$ if and only if $* = D$.*

Proof. First, we prove that the program $P = \mathbb{A}(M) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$, where $\mathcal{R} = \mathcal{R}_\varphi \cup \mathcal{R}_\delta \cup \mathcal{R}_{\text{st}}$, has a unique answer set. If P is stratified then it has a unique answer set (see Section 2.2.1); hence, we prove that P is stratified. The program can be partitioned as $P = P_0 \cup P_1 \cup P_2 \cup P_3$, where

$$P_0 = \mathbb{A}(\lambda^*), \quad P_1 = \mathbb{A}(M), \quad P_2 = \mathcal{R}_\varphi, \quad P_3 = \mathcal{R}_\delta \cup \mathcal{R}_{\text{st}}.$$

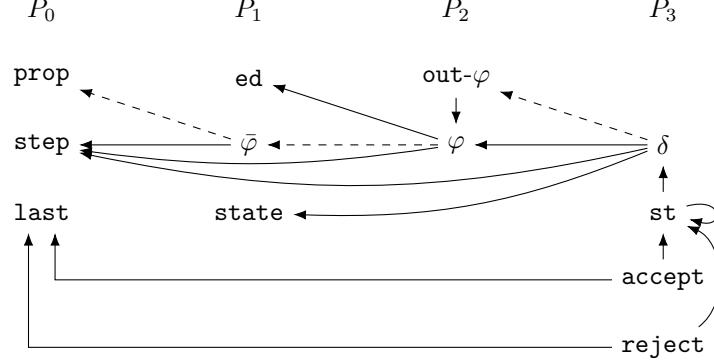


Figure 3.3: Predicate dependencies in the program of Proposition 3.3.1, where predicates are grouped by the partitions used in the proof. Dashed edges $x \rightarrow y$ denote that y occurs negatively in some definition of x , whereas solid edges denote that y only occurs positively in the definitions of x .

Figure 3.3 graphically proves that this partitioning complies with the definition of a stratified program. Predicates are grouped into disjoint partitions and connected such that an edge $x \rightarrow y$ denotes that either (i) y occurs negatively in some definition of x , or (ii) y only occurs positively in the definitions of x . Case (i) is illustrated with a dashed line, whereas (ii) is illustrated with a solid line. In compliance with the definition of a stratified program, dashed lines always point to predicates in partitions with a strictly lower index, while solid lines always point to predicates within the same partition or lower-indexed partitions.

The unique answer set is $A = A_0 \cup A_1 \cup A_2 \cup A_3$, where A_i corresponds to partition P_i :

$$A_0 = \{\mathbf{prop}(p, t). \mid p \in \lambda^*[t], 0 \leq t \leq n\} \cup \{\mathbf{step}(t). \mid 0 \leq t \leq n\} \cup \{\mathbf{last}(n). \},$$

$$\begin{aligned} & \{\mathbf{state}(u). \mid u \in \mathcal{U}\} \cup \\ A_1 = & \{\mathbf{ed}(u, u', i). \mid u, u' \in \mathcal{U}, 1 \leq i \leq |\varphi(u, u')|\} \cup \\ & \{\bar{\varphi}(u, u', i, t). \mid u, u' \in \mathcal{U}, \phi_i \in \varphi(u, u'), 0 \leq t \leq n, \lambda^*[t] \not\models \phi_i\}, \end{aligned}$$

$$A_2 = \{\varphi(u, u', i, t). \mid u, u' \in \mathcal{U}, \phi_i \in \varphi(u, u'), 0 \leq t \leq n, \lambda^*[t] \models \phi_i\} \cup \{\mathbf{out-}\varphi(u, t). \mid u \in \mathcal{U}, 0 \leq t \leq n, \exists u' \in \mathcal{U} \text{ s.t. } \lambda^*[t] \models \varphi(u, u')\},$$

$$\begin{aligned} & \{\delta(u, u', t). \mid u, u' \in \mathcal{U}, 0 \leq t \leq n, \lambda^*[t] \models \varphi(u, u')\} \cup \\ & \{\delta(u, u, t). \mid u \in \mathcal{U}, 0 \leq t \leq n, \nexists u' \in \mathcal{U} \text{ s.t. } \lambda^*[t] \models \varphi(u, u')\} \cup \\ & \{\mathbf{st}(0, u^0). \} \cup \\ A_3 = & \{\mathbf{st}(t, u). \mid 1 \leq t \leq n+1, u = M(\lambda^*)[t]\} \cup \\ & \{\mathbf{accept}. \mid M(\lambda^*)[n+1] = u^A\} \cup \\ & \{\mathbf{reject}. \mid M(\lambda^*)[n+1] = u^R\}. \end{aligned}$$

We now prove that $\mathbf{accept} \in A$ if and only if $* = G$ (i.e., the trace achieves the goal). If $* = G$ then, since the RM is valid with respect to λ^* (see Definition 3.2.4), the RM traversal $M(\lambda^*)$ finishes in the accepting state u^A ; that is, $M(\lambda^*)[n+1] = u^A$. This holds if and only if $\mathbf{accept} \in A$.

The proof showing that $\text{reject} \in A$ if and only if $* = D$ (i.e., the trace reaches a dead-end) is similar to the previous one. If $* = D$ then, since the RM is valid with respect to λ^* , the RM traversal $M(\lambda^*)$ finishes in the rejecting state u^R ; that is, $M(\lambda^*)[n+1] = u^R$. This holds if and only if $\text{reject} \in A$. \square

3.3.4 Determinism

The state-transition function δ_M of a reward machine M must be deterministic, as described in Section 3.2. In our framework, δ_M is deterministic if the logical transition function φ it builds upon is deterministic, which is guaranteed when the formulas labeling edges from a given state to two different states are *mutually exclusive* (i.e., a proposition appears positively in one formula and negatively in the other). Determinism ensures that each trace is mapped into a single traversal. Indeed, the algorithms we consider for exploiting RMs (see Section 2.1.5 and Chapter 4) do not model uncertainty over the current RM state; namely, they assume RMs to be perfect high-level abstractions of the state-action histories and, therefore, need to know the exact current RM state to determine the best action to perform.

The ASP representation of a reward machine (Section 3.3.2) encodes the logic transition function φ and is correct under the assumption that φ is deterministic (Section 3.3.3); otherwise, as mentioned above, there could be multiple traversals for a given trace, which could lead to simultaneous acceptance and rejection. It is thus imperative to have rules that verify whether an RM is deterministic or not.

The set of rules below verifies whether an RM is deterministic by encoding the mutual exclusivity condition described before. These rules are defined over the factual ASP representation of the RMs (see Definition 3.3.3) since it enables checking whether a proposition appears positively or negatively in a given edge. The $\text{mutex}(u, v, e, v', e')$ atoms indicate that the formula on the edge from u to v with index e is mutually exclusive with that on the edge from u to v' with index e' . The first and second rules specify that two outgoing edges from state X to two different states (Y and Z) are mutually exclusive if a proposition P appears positively in one edge and negatively in the other.⁴ The third rule enforces mutual exclusivity between the edges from a given state X to two different states Y and Z .

$$\left\{ \begin{array}{l} \text{mutex}(X, Y, EY, Z, EZ) : -\text{pos}(X, Y, EY, P), \text{neg}(X, Z, EZ, P), Y < Z. \\ \text{mutex}(X, Y, EY, Z, EZ) : -\text{neg}(X, Y, EY, P), \text{pos}(X, Z, EZ, P), Y < Z. \\ :-\text{not mutex}(X, Y, EY, Z, EZ), \text{ed}(X, Y, EY), \text{ed}(X, Z, EZ), Y < Z. \end{array} \right\}$$

These rules are enforced during the learning of RMs to guarantee that the resulting RMs are deterministic. Further details are provided in Chapter 4.

3.4 Symmetry Breaking

A reward machine can be easily transformed into equivalent (or *symmetric*) reward machines. There are different types of symmetries. First, any two states except for u^0 , u^A and u^R are interchangeable;

⁴The comparison $Y < Z$ is done instead of $Y \neq Z$ for efficiency. Both comparisons are equivalent in this context; however, the former imposes a lexicographical order to evaluate the rules and thus avoids reevaluating the expression when Y and Z are interchanged.

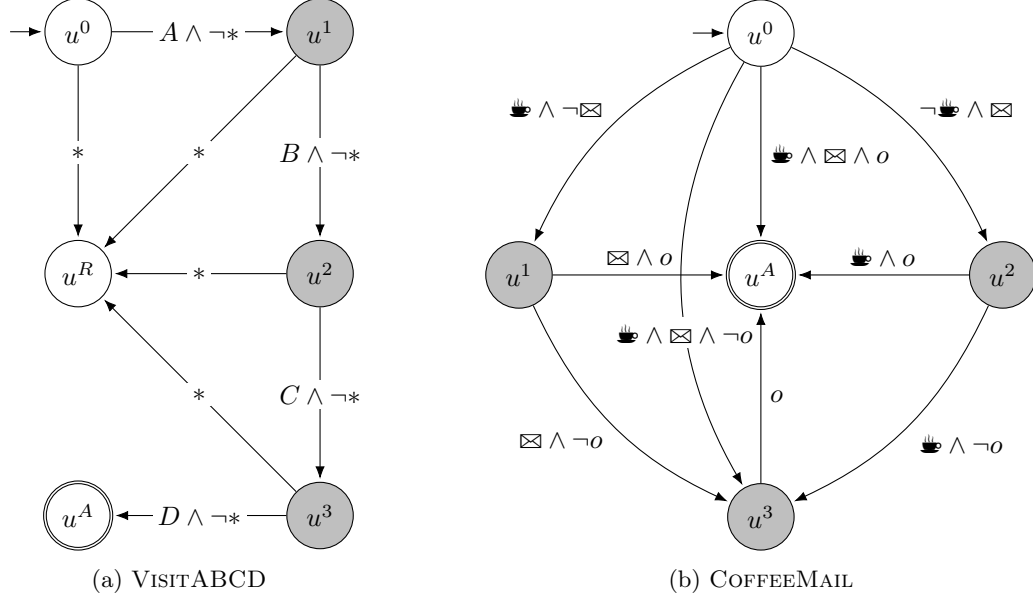


Figure 3.4: Minimal reward machines for two OFFICEWORLD tasks. Self-transitions, and transitions to the rejecting state u^R in (b) are omitted for simplicity. The shaded states can be interchanged in the absence of symmetry breaking.

for example, Figure 3.4 shows two RMs whose states u^1 , u^2 and u^3 can be used interchangeably. Second, there can also be symmetries in the indexing of the edges; namely, the edges in the ASP representation of an RM can be arbitrarily indexed using any integer number. For instance, if the edge indices can be within $\{1, 2\}$, an alternative representation to that in Example 3.3.2 of the RM for COFFEE (see Figure 3.2) is:

$$\left\{ \begin{array}{ll} \text{state}(u^0). \text{state}(u^1). & \text{state}(u^A). \text{state}(u^R). \\ \text{ed}(u^0, u^1, 2). \text{ed}(u^0, u^A, 1). \text{ed}(u^0, u^R, 2). & \text{ed}(u^1, u^A, 1). \text{ed}(u^1, u^R, 2). \\ \bar{\varphi}(u^0, u^1, 2, T) :- \text{not prop}(\text{☕}, T), \text{step}(T). & \bar{\varphi}(u^0, u^1, 2, T) :- \text{prop}(o, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^1, 2, T) :- \text{prop}(*, T), \text{step}(T). & \bar{\varphi}(u^0, u^A, 1, T) :- \text{not prop}(\text{☕}, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^A, 1, T) :- \text{not prop}(o, T), \text{step}(T). & \bar{\varphi}(u^0, u^A, 1, T) :- \text{prop}(*, T), \text{step}(T). \\ \bar{\varphi}(u^0, u^R, 2, T) :- \text{not prop}(*, T), \text{step}(T). & \bar{\varphi}(u^1, u^A, 1, T) :- \text{not prop}(o, T), \text{step}(T). \\ \bar{\varphi}(u^1, u^A, 1, T) :- \text{prop}(*, T), \text{step}(T). & \bar{\varphi}(u^1, u^R, 2, T) :- \text{not prop}(*, T), \text{step}(T). \end{array} \right\}.$$

Third, and finally, the indices of two edges between the same pair of states can also be interchanged.

We here describe a symmetry breaking mechanism that imposes a unique assignment of state and edge indices given a labeling of the RM edges. We formalize the mechanism using a class of labeled directed graphs that subsumes RMs (Section 3.4.1), and encode it as a satisfiability (SAT) formula and formally prove several of its properties (Section 3.4.2). Finally, we show how the mechanism is applicable to RMs and propose two ASP encodings (Section 3.4.3) to avoid considering multiple symmetric RMs during learning (see Chapter 4), hence speeding up the process.

3.4.1 Graph Indexing

We propose a symmetry breaking mechanism for a particular class of labeled directed graphs. Let $\mathcal{L} = \{l_1, \dots, l_k\}$ be a set of *labels*, and let $G = \langle \mathcal{V}, \mathcal{E} \rangle$ be a labeled directed graph with a set of *nodes* $\mathcal{V} = \{v_1, \dots, v_n\}$ and a set of *edges* \mathcal{E} . Each edge in \mathcal{E} is of the form $\langle u, v, L \rangle$, where $u, v \in \mathcal{V}$ are the two connected nodes, and $L \subseteq \mathcal{L}$ is a subset of labels. For each node $u \in \mathcal{V}$, let $\mathcal{E}^o(u) = \{\langle v, w, L \rangle \in \mathcal{E} \mid u = v\}$ be the set of outgoing labeled edges from u , and let $\mathcal{E}^i(u) = \{\langle v, w, L \rangle \in \mathcal{E} \mid u = w\}$ be the set of incoming labeled edges.

We define a class \mathcal{G} of labeled directed graphs by imposing the following three assumptions:

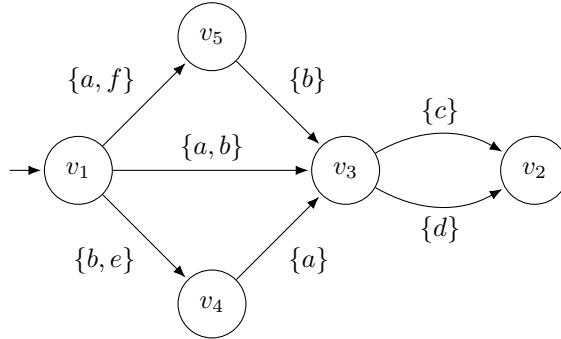
Assumption 3.4.1. *The node v_1 is a designated start node.*

Assumption 3.4.2. *Each node $u \in \mathcal{V} \setminus \{v_1\}$ is reachable on a directed path from v_1 .*

Assumption 3.4.3. *Outgoing label sets from each node are unique, i.e. for each $u \in \mathcal{V}$ and label set $L \subseteq \mathcal{L}$ there is at most one edge $\langle u, v, L \rangle \in \mathcal{E}^o(u)$.*

As a consequence of Assumption 3.4.2, it holds that $|\mathcal{E}^i(u)| \geq 1$ for each $u \in \mathcal{V} \setminus \{v_1\}$.

Example 3.4.1. *The following figure is a labeled directed graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ that belongs to class \mathcal{G} , where $\mathcal{V} = \{v_1, \dots, v_5\}$. The set of labels is $\mathcal{L} = \{a, b, c, d, e, f\}$.*



Label Set Ordering

Given an *ordered* set of labels $\mathcal{L} = \{l_1, \dots, l_k\}$, we impose a total order on label sets as follows.

Definition 3.4.1. *A label set $L \subseteq \mathcal{L}$ is lower than a label set $L' \subseteq \mathcal{L}$, denoted $L < L'$, if there exists a label $l_{1 \leq m \leq k} \in \mathcal{L}$ such that*

1. $l_m \notin L$ and $l_m \in L'$, and
2. *there is not a label $l_{m' \mid m' < m}$ such that $l_{m'} \in L$ and $l_{m'} \notin L'$.*

A label set $L \subseteq \mathcal{L}$ can be mapped into a binary string $B(L) \in \{0, 1\}^k$, where $B_m(L) = 1$ if $l_m \in L$ and 0 otherwise. Note that $B_m(L)$ denotes the m -th binary digit in $B(L)$. Then, a label set L is lower than another label set L' if its binary representation $B(L)$ is lexicographically lower than $B(L')$.

Example 3.4.2. *Given the label set $\mathcal{L} = \{a, b, c, d, e, f\}$, the following inequalities between some of its subsets hold. The second column contains the corresponding binary representations of the sets.*

$$\begin{array}{lll} \{\} & < & \{a, d, f\} \quad (000000 < 100101), \\ \{d\} & < & \{c\} \quad (000100 < 001000), \\ \{b, e\} & < & \{a, f\} \quad (010010 < 100001), \\ \{a, f\} & < & \{a, b\} \quad (100001 < 110000). \end{array}$$

Graph Indexing

Given a graph $G \in \mathcal{G}$, a *graph indexing* is a tuple of bijections $\mathcal{I}(G) = \langle f, \{\Gamma_u\}_{u \in \mathcal{V}} \rangle$, where f assigns unique integers to each node $u \in \mathcal{V}$ and, given a node $u \in \mathcal{V}$, Γ_u assigns unique integers to each outgoing edge in $\mathcal{E}^o(u)$. Formally, the bijections are defined as

$$\begin{aligned} f : \mathcal{V} &\rightarrow \{1, \dots, |\mathcal{V}|\} \text{ s.t. } f(v_1) = 1, \\ \Gamma_u : \mathcal{E}^o(u) &\rightarrow \{1, \dots, |\mathcal{E}^o(u)|\}, \quad \forall u \in \mathcal{V}. \end{aligned}$$

Hence a graph indexing always assigns 1 to the designated start node v_1 . Since outgoing label sets are unique due to Assumption 3.4.3, we use $\Gamma_u(L)$ as shorthand for $\Gamma_u(u, v, L)$.

Given a graph indexing $\mathcal{I}(G)$, we introduce an associated *parent function* $\Pi_{\mathcal{I}} : \mathcal{V} \setminus \{v_1\} \rightarrow \{1, \dots, |\mathcal{V}|\} \times \mathbb{N}$ from nodes (excluding the start node v_1) to pairs of integers, defined as

$$\Pi_{\mathcal{I}}(v) = \min_{\langle u, v, L \rangle \in \mathcal{E}^i(v)} \langle f(u), \Gamma_u(L) \rangle.$$

Here, the minimum is with respect to a lexicographical ordering of integer pairs. Hence $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$ is the smallest integer i assigned to any node on an incoming edge to v and, in the case of ties, the smallest integer e on such an edge. The parent function is well-defined since $|\mathcal{E}^i(v)| \geq 1$ for each $v \in \mathcal{V} \setminus \{v_1\}$ due to Assumption 3.4.2.

We consider the graph indexing that corresponds to a breadth-first search (BFS) traversal of the graph G , which we proceed to define.

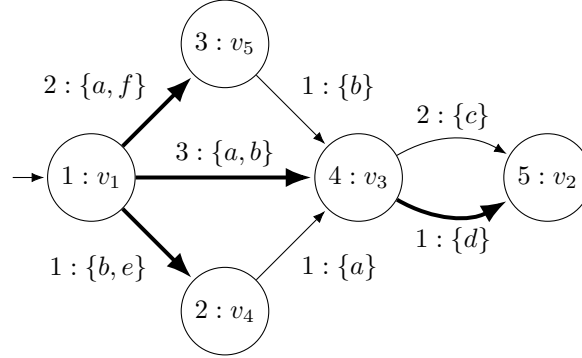
Definition 3.4.2 (BFS traversal). *A graph indexing $\mathcal{I}(G)$ is a BFS traversal if the following conditions hold:*

1. *For each pair of nodes u and v in $\mathcal{V} \setminus \{v_1\}$, $\Pi_{\mathcal{I}}(u) < \Pi_{\mathcal{I}}(v) \Leftrightarrow f(u) < f(v)$.*
2. *For each node $u \in \mathcal{V}$ and each pair of outgoing edges $\langle u, v, L \rangle$ and $\langle u, v', L' \rangle$ in $\mathcal{E}^o(u)$, $L < L' \Leftrightarrow \Gamma_u(L) < \Gamma_u(L')$.*

Due to the second condition in Definition 3.4.2, the bijection Γ_u of each node $u \in \mathcal{V}$ clearly orders outgoing edges by their label sets. Due to the first condition, the bijection f orders node u before node v if the parent function of u is smaller than that of v . In a BFS traversal from v_1 , nodes are processed in the order they are first visited. In this context, the parent function identifies the edge used to visit a node for the first time. Together, these facts imply that f assigns integers to nodes in the order they are visited by a BFS traversal from v_1 , given that the label set ordering is used to

break ties among edges. This BFS traversal can be characterized by a BFS subtree whose edges are defined by the parent function.

Example 3.4.3. The figure below shows a graph indexing for the graph in Example 3.4.1, with nodes and edges labeled by their assigned integer. This graph indexing is a BFS traversal since nodes are ordered according to their distance from the start node v_1 , and since the edge integers used to break ties are consistent with the label set ordering shown in Example 3.4.2. The parent function is given by $\Pi_{\mathcal{I}}(v_2) = \langle 4, 1 \rangle$, $\Pi_{\mathcal{I}}(v_3) = \langle 1, 3 \rangle$, $\Pi_{\mathcal{I}}(v_4) = \langle 1, 1 \rangle$, and $\Pi_{\mathcal{I}}(v_5) = \langle 1, 2 \rangle$, and the corresponding BFS subtree appears in bold.



The important property that we exploit about BFS traversals is that they are unique, which we prove in Lemma 3.4.1. To prove it, we use the result in Proposition 3.4.1 when BFS is applied to any kind of directed graph where all nodes are reachable.

Proposition 3.4.1. *If BFS visits the neighbors of each node in a fixed order, the resulting tree is unique.*

Proof. By contradiction. Assume that two different BFS trees, T and T' , are produced using the same visitation criteria. Then T contains an edge $\langle u, v \rangle$ that is not in T' , and T' contains an edge $\langle u', v \rangle$ that is not in T . In the case of T , this means that u was visited before u' . Analogously, u' was visited before u to produce T' . Therefore, the visitation criteria is different for each of the BFS trees. This is a contradiction. \square

Lemma 3.4.1. *Each graph $G \in \mathcal{G}$ has a unique associated BFS traversal $\mathcal{I}(G)$.*

Proof. Intuitively, the lemma holds because there is only one way to perform a BFS traversal from v_1 , given that we use the label set ordering to break ties among edges.

Formally, for each $u \in \mathcal{V}$, since outgoing label sets are unique by Assumption 3.4.3, there is a unique bijection Γ_u that satisfies the second condition in Definition 3.4.2. If Γ_u orders outgoing edges in any other way, there will always be two outgoing edges $\langle u, v, L \rangle$ and $\langle u, v', L' \rangle$ in $\mathcal{E}^o(u)$ such that $L < L'$ and $\Gamma_u(L) > \Gamma_u(L')$, thus violating the condition.

Then, if we fix the correct definition of Γ_u for each $u \in \mathcal{V}$, there is a unique bijection f that satisfies the first condition in Definition 3.4.2. To recover this bijection we can simply perform a BFS traversal from v_1 , using the bijections Γ_u , $u \in \mathcal{V}$, to break ties among edges. By Proposition 3.4.1, this results in a unique BFS tree. If f orders nodes in any other way, there will always be two nodes u and v in $\mathcal{V} \setminus \{v_1\}$ such that $\Pi_{\mathcal{I}}(u) < \Pi_{\mathcal{I}}(v)$ and $f(u) > f(v)$, thus violating the condition. \square

3.4.2 SAT Encoding

We define a SAT formula that encodes a BFS traversal $\mathcal{I}(G) = \langle f, \{\Gamma_u\}_{u \in \mathcal{V}} \rangle$ of a given graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ in the class \mathcal{G} , defined on a set of labels $\mathcal{L} = \{l_1, \dots, l_k\}$. Since a graph indexing $\mathcal{I}(G)$ assigns unique integers to nodes and edges, we use i to refer to a node u such that $f(u) = i$, $\langle i, e \rangle$ or $\langle i, e, L \rangle$ to refer to an edge $\langle u, v, L \rangle$ such that $f(u) = i$ and $\Gamma_u(L) = e$, and m to refer to a label $l_m \in \mathcal{L}$. We sometimes extend this notation in the natural way, e.g. by writing $\Gamma_i(L)$, $\mathcal{E}^o(i)$ and $\Pi_{\mathcal{I}}(i)$.

Variables

We first define a set \mathcal{X} of propositional SAT variables for all combinations of symbols (sometimes with restrictions as indicated):

1. $ed(i, j, e)$, [edge $\langle i, e \rangle$ ends in node j]
2. $label(i, e, m)$, [the label set on edge $\langle i, e \rangle$ includes $l_m \in \mathcal{L}$]
3. $pa(i, j)$, $i < j$, [node i is the parent of j in the BFS subtree]
4. $sm(i, j, e)$, $i < j$, [e is the smallest integer on a BFS edge from node i to j]
5. $lt(i, e - 1, e, m)$, $e > 1$, [there is a label $l_{m'} |_{m' \leq m}$ on $\langle i, e \rangle$ and not on $\langle i, e - 1 \rangle$]

Intuitively, variables $ed(i, j, e)$ and $label(i, e, m)$ are used to encode a graph G together with an associated graph indexing $\mathcal{I}(G)$, variables $pa(i, j)$ and $sm(i, j, e)$ are used to encode the parent function $\Pi_{\mathcal{I}}$, and variables $lt(i, e - 1, e, m)$ are used to encode the label set ordering.

Clauses

We next define a set \mathcal{C} of clauses on \mathcal{X} for all combinations of symbols (sometimes with restrictions as indicated). The first set of clauses (1–8) enforces the first condition in Definition 3.4.2: for any two nodes $i > 1$ and $j > 1$, $\Pi_{\mathcal{I}}(i) < \Pi_{\mathcal{I}}(j) \Leftrightarrow i < j$.

1. $\bigvee_{i < j} pa(i, j)$, $j > 1$, [node $j > 1$ has incoming BFS edge]
2. $pa(i, j) \Rightarrow \neg pa(i', j)$, $i < i' < j$, [incoming BFS edge is unique]
3. $pa(i, j) \Rightarrow \bigvee_e sm(i, j, e)$, $i < j$, [BFS edge implies smallest integer]
4. $pa(i, j) \Rightarrow \neg ed(i', j', e)$, $i' < i < j \leq j'$, [respect BFS order]
5. $sm(i, j, e) \Rightarrow pa(i, j)$, $i < j$, [smallest integer implies BFS edge]
6. $sm(i, j, e) \Rightarrow \neg sm(i, j, e')$, $i < j$, $e < e'$, [smallest integer is unique]
7. $sm(i, j, e) \Rightarrow ed(i, j, e)$, $i < j$, [smallest integer implies edge]
8. $sm(i, j, e) \Rightarrow \neg ed(i, j', e')$, $i < j \leq j'$, $e' < e$, [correctly break ties]

Intuitively, Clauses 1 and 2 state that each node $j > 1$ has a unique parent node i in the BFS subtree. Clauses 3, 5 and 6 state that each node $j > 1$ has a unique incoming edge $\langle i, e \rangle$ with smallest integer e from its parent i in the BFS subtree. Clause 7 ensures that the incoming edge $\langle i, e \rangle$ to j in the BFS subtree corresponds to an actual edge in the graph G .

Clauses 4 and 8 constitute the core of symmetry breaking by enforcing the condition that $\Pi_{\mathcal{I}}(i) < \Pi_{\mathcal{I}}(j)$ should imply $i < j$. By definition of $\Pi_{\mathcal{I}}$, the incoming edge $\langle i, e \rangle$ to j in the BFS subtree should be the lexicographically smallest such integer pair. Hence the graph G cannot contain any incoming edge $\langle i', e' \rangle$ to j from a node $i' < i$. In addition, no node $j' > j$ can have such an incoming

edge either, since otherwise its parent function would be smaller than that of j , thus violating the desired condition. These two facts are jointly encoded in Clause 4 by enforcing the restriction $j' \geq j$.

Likewise, if $\langle i, e \rangle$ is the incoming edge to j in the BFS subtree, the graph G cannot contain an incoming edge $\langle i, e' \rangle$ from the same node i with $e' < e$. Again, no node $j' > j$ can have such an incoming edge either, since otherwise its parent function would be smaller than that of j . These two facts are jointly encoded in Clause 8 by enforcing the restriction $j' \geq j$.

The second set of clauses (9–14) assigns edge integers to the outgoing edges from each node, enforcing the second condition in Definition 3.4.2: for each node i and pair of outgoing edges $\langle i, e, L \rangle$ and $\langle i, e', L' \rangle$, $L < L' \Leftrightarrow e < e'$. Due to the transitivity of the relation $<$, it is sufficient to check that the condition holds for all pairs of consecutive edge integers $\langle e - 1, e \rangle$. Clauses 9 and 10 enforce that edge integers are unique between 1 and $|\mathcal{E}^o(i)|$.

9. $ed(i, j, e) \Rightarrow \bigvee_{j'} ed(i, j', e - 1), e > 1$, [edge integers start at 1 and are contiguous]
10. $ed(i, j, e) \Rightarrow \neg ed(i, j', e), j < j'$, [edge integers cannot be duplicated]

Clauses 11–14 are used to enforce that two consecutive edges $\langle i, e - 1, L \rangle$ and $\langle i, e, L' \rangle$ satisfy $L < L'$. Formally, variable $lt(i, e - 1, e, m)$ is only true if there exists $m' \leq m$ such that $l_{m'} \notin L$ and $l_{m'} \in L'$. This is implemented using the following two clauses:

11. $lt(i, e - 1, e, m) \Rightarrow \neg label(i, e - 1, m) \vee lt(i, e - 1, e, m - 1), e > 1$,
12. $lt(i, e - 1, e, m) \Rightarrow label(i, e, m) \vee lt(i, e - 1, e, m - 1), e > 1$.

Hence if $lt(i, e - 1, e, m)$ holds, either $l_m \notin L$ and $l_m \in L'$, or $lt(i, e - 1, e, m - 1)$ holds for $m - 1$. The disjuncts mentioning $m - 1$ are only evaluated when $m > 1$. The next clause ensures that for each edge $\langle i, e \rangle$ with $e > 1$, $lt(i, e - 1, e, m)$ is true for at least one label $l_m \in \mathcal{L}$:

13. $ed(i, j, e) \Rightarrow \bigvee_m lt(i, e - 1, e, m), e > 1$.

Finally, the following clause encodes the second part of Definition 3.4.1, ensuring that the label set on edge $\langle i, e \rangle$ is *not* lower than that on $\langle i, e - 1 \rangle$:

14. $lt(i, e - 1, e, m) \vee \neg label(i, e - 1, m) \vee label(i, e, m), e > 1$.

Properties

We proceed to prove several properties about the SAT encoding. Concretely, we show that there is a one-to-one correspondence between the BFS traversal of a graph and a solution to the SAT encoding.

Definition 3.4.3. *Given a graph $G = \langle \mathcal{V}, \mathcal{E} \rangle \in \mathcal{G}$ defined on a set of labels $\mathcal{L} = \{l_1, \dots, l_k\}$ and an associated graph indexing $\mathcal{I}(G) = \langle f, \{\Gamma_u\}_{u \in \mathcal{V}} \rangle$, let $X(G, \mathcal{I})$ be an assignment to the SAT variables in \mathcal{X} , assigning false to all variables in \mathcal{X} except as follows:*

- For each edge $\langle u, v, L \rangle \in \mathcal{E}$, $ed(f(u), f(v), \Gamma_u(L))$ is true.
- For each edge $\langle u, v, L \rangle \in \mathcal{E}$ and each label $l_m \in L$, $label(f(u), \Gamma_u(L), m)$ is true.

- For each node $v \in \mathcal{V} \setminus \{v_1\}$ with $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$, $pa(i, f(v))$ and $sm(i, f(v), e)$ are true.
- For each node $u \in \mathcal{V}$, each pair of outgoing edges $\langle u, v, L \rangle$ and $\langle u, v', L' \rangle$ in $\mathcal{E}^o(u)$ such that $\Gamma_u(L) = \Gamma_u(L') - 1$, and each label $l_m \in \mathcal{L}$, $lt(f(u), \Gamma_u(L), \Gamma_u(L'), m)$ is true if there exists $m' \leq m$ such that $l_{m'} \notin L$ and $l_{m'} \in L'$.

Example 3.4.4. Given the graph G , graph indexing $\mathcal{I}(G)$ and set of labels $\mathcal{L} = \{a, b, c, d, e, f\}$ from Example 3.4.3, the assignment $X(G, \mathcal{I})$ makes the following SAT variables in \mathcal{X} true:

$$\left\{ \begin{array}{llllll} ed(1, 2, 1) & label(1, 1, 2) & label(1, 1, 5) & ed(1, 3, 2) & label(1, 2, 1) & label(1, 2, 6) \\ ed(1, 4, 3) & label(1, 3, 1) & label(1, 3, 2) & ed(2, 4, 1) & label(2, 1, 1) & \\ ed(3, 4, 1) & label(3, 1, 2) & ed(4, 5, 1) & label(4, 1, 4) & ed(4, 5, 2) & label(4, 2, 3) \\ pa(1, 2) & pa(1, 3) & pa(1, 4) & pa(4, 5) & & \\ sm(1, 2, 1) & sm(1, 3, 2) & sm(1, 4, 3) & sm(4, 5, 1) & & \\ lt(1, 1, 2, 1) & lt(1, 1, 2, 2) & lt(1, 1, 2, 3) & lt(1, 1, 2, 4) & lt(1, 1, 2, 5) & lt(1, 1, 2, 6) \\ lt(1, 2, 3, 2) & lt(1, 2, 3, 3) & lt(1, 2, 3, 4) & lt(1, 2, 3, 5) & lt(1, 2, 3, 6) & \\ lt(4, 1, 2, 3) & lt(4, 1, 2, 4) & lt(4, 1, 2, 5) & lt(4, 1, 2, 6) & & \end{array} \right\}.$$

Theorem 3.4.1. Given a graph G and a graph indexing $\mathcal{I}(G)$, the assignment $X(G, \mathcal{I})$ to the SAT variables in \mathcal{X} satisfies all SAT clauses in \mathcal{C} if and only if $\mathcal{I}(G)$ is a BFS traversal.

Proof. \Leftarrow : Assume that $\mathcal{I}(G)$ is a BFS traversal (i.e., it satisfies the conditions of Definition 3.4.2). We show that each clause in \mathcal{C} is satisfied:

1. $\bigvee_{i| i < j} pa(i, j)$ holds for v such that $f(v) = j > 1$ since $pa(i, j)$ is true for $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$.
2. $pa(i, j) \Rightarrow \neg pa(i', j)$ holds since $pa(i, j)$ is true for a single i .
3. $pa(i, j) \Rightarrow \bigvee_e sm(i, j, e)$ holds for v such that $f(v) = j$ since $pa(i, j)$ and $sm(i, j, e)$ are true for $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$.
4. $pa(i, j) \Rightarrow \neg ed(i', j', e)$ holds for v such that $f(v) = j$ and $i' < i < j = j'$ since $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$ is the lexicographically smallest integer pair on incoming edges to v , implying that G cannot contain an edge to v from a node u with $f(u) = i' < i$. Moreover, since $\mathcal{I}(G)$ is a BFS traversal, we cannot have $\Pi_{\mathcal{I}}(w) < \Pi_{\mathcal{I}}(v)$ for a node w with $f(w) = j' > j$, else the second condition in Definition 3.4.2 is violated. Hence G cannot contain an edge to w from a node u with $f(u) = i' < i$, so the clause also holds for $i' < i < j < j'$.
5. $sm(i, j, e) \Rightarrow pa(i, j)$ holds for v such that $f(v) = j$ since $pa(i, j)$ and $sm(i, j, e)$ are true for $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$.
6. $sm(i, j, e) \Rightarrow \neg sm(i, j, e')$ holds since $sm(i, j, e)$ is true for a single i and e .
7. $sm(i, j, e) \Rightarrow ed(i, j, e)$ holds for v such that $f(v) = j$ since $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$ implies that there exists an edge $\langle u, v, L \rangle \in \mathcal{E}$ with $f(u) = i$ and $\Gamma_u(L) = e$.
8. $sm(i, j, e) \Rightarrow \neg ed(i, j', e')$ holds for v such that $f(v) = j$ and $j = j'$ since $\Pi_{\mathcal{I}}(v) = \langle i, e \rangle$ is the lexicographically smallest integer pair on incoming edges to v , implying that G cannot contain an edge $\langle u, v, L \rangle$ with $f(u) = i$ and $\Gamma_u(L) < e$. For $j < j'$, the parent function of the node w

with $f(w) = j'$ cannot be smaller than that of v , else the second condition in Definition 3.4.2 is violated. Hence G cannot contain an edge $\langle u, w, L \rangle$ with $f(u) = i$ and $\Gamma_u(L) < e$. Thus the clause also holds for the case $j < j'$.

9. $ed(i, j, e) \Rightarrow \bigvee_{j'} ed(i, j', e - 1)$ holds for u with $f(u) = i$ since Γ_u is a bijection onto $\{1, \dots, |\mathcal{E}^o(u)|\}$.
10. $ed(i, j, e) \Rightarrow \neg ed(i, j', e)$ holds for u with $f(u) = i$ since Γ_u is a bijection.
11. $lt(i, e - 1, e, m) \Rightarrow \neg label(i, e - 1, m) \vee lt(i, e - 1, e, m - 1)$ holds for u , $f(u) = i$, and outgoing edges $\langle u, v, L \rangle$, $\langle u, v', L' \rangle$ with $\Gamma_u(L) = e - 1 = \Gamma_u(L') - 1$ since $lt(i, e - 1, e, m)$ implies that either $l_m \notin L$ or there exists $m' < m$ such that $l_{m'} \notin L$ and $l_{m'} \in L'$.
12. $lt(i, e - 1, e, m) \Rightarrow label(i, e, m) \vee lt(i, e - 1, e, m - 1)$ holds for the same setting since $lt(i, e - 1, e, m)$ implies that either $l_m \in L'$ or there exists $m' < m$ such that $l_{m'} \notin L$ and $l_{m'} \in L'$.
13. $ed(i, j, e) \Rightarrow \bigvee_{j'} ed(i, j', e - 1)$ holds since $\mathcal{I}(G)$ is a BFS traversal, implying that the bijection Γ_u for u with $f(u) = i$ satisfies $L < L'$ whenever $\Gamma_u(L) < \Gamma_u(L')$ (and in particular when $\Gamma_u(L) = e - 1 = \Gamma_u(L') - 1$). Since $L < L'$, there has to exist at least one $m, 1 \leq m \leq k$ such that $l_m \notin L$ and $l_m \in L'$ due to Definition 3.4.1.
14. $lt(i, e - 1, e, m) \vee \neg label(i, e - 1, m) \vee label(i, e, m)$ also holds for u with $f(u) = i$ since $\Gamma_u(L) = e - 1 = \Gamma_u(L') - 1$ implies $L < L'$. Hence for the given m , Definition 3.4.1 is satisfied either 1) by a label $l_{m' \leq m}$, implying that $lt(i, e - 1, e, m)$ is true; or 2) by a label $l_{m' > m}$, implying that $l_m \in L$ and $l_m \notin L'$ cannot both be true.

\Rightarrow : Assume that $X(G, \mathcal{I})$ satisfies all SAT clauses. We show that $\mathcal{I}(G)$ is a BFS traversal. First note from above that $X(G, \mathcal{I})$ satisfies all clauses except 4, 8, 13 and 14 even if $\mathcal{I}(G)$ is not a BFS traversal. Hence we can focus exclusively on these four clauses.

We first analyze Clauses 13 and 14. For Clause 13 to be true, any edge $\langle i, e \rangle$ induced from graph G with $e > 1$ has to satisfy $lt(i, e - 1, e, m)$ for at least one label $l_m \in \mathcal{L}$. Let u be the node with $f(u) = i$ and let $\langle u, v, L \rangle$ and $\langle u, v', L' \rangle$ be the two outgoing edges in $\mathcal{E}^o(u)$ such that $\Gamma_u(L) = e - 1 = \Gamma_u(L') - 1$. By definition of $X(G, \mathcal{I})$ there exists $m, 1 \leq m \leq k$ such that $l_m \notin L$ and $l_m \in L'$. For the smallest such m there cannot exist $m' < m$ such that $l_{m'} \in L$ and $l_{m'} \notin L'$, else Clause 14 would be violated for m' . Hence Definition 3.4.1 holds for label l_m , implying $L < L'$.

We next analyze Clauses 4 and 8. Let u be the node such that $f(u) = j$ and $\Pi_{\mathcal{I}}(u) = \langle i, e \rangle$, and let v be any node such that $f(v) = j' > j$. Since Clause 4 holds, graph G cannot contain an edge $\langle w, v, L \rangle$ such that $f(w) = i' < i$. Since Clause 8 holds, graph G cannot contain an edge $\langle w, v, L \rangle$ such that $f(w) = i$ and $\Gamma_w(L) < e$. Since $\Pi_{\mathcal{I}}(v)$ cannot equal $\langle i, e \rangle$, it has to be larger than $\langle i, e \rangle$, implying $\Pi_{\mathcal{I}}(u) < \Pi_{\mathcal{I}}(v)$.

We have shown that the two conditions in Definition 3.4.2 hold: for each pair of nodes u and v in $\mathcal{V} \setminus \{v_1\}$, $\Pi_{\mathcal{I}}(u) < \Pi_{\mathcal{I}}(v)$ implies $f(u) < f(v)$, and for each node $u \in \mathcal{V}$ and pair of outgoing edges $\langle u, v, L \rangle$ and $\langle u, v', L' \rangle$ in $\mathcal{E}^o(u)$, $L < L'$ implies $\Gamma_u(L) < \Gamma_u(L')$. Hence by definition $\mathcal{I}(G)$ is a BFS traversal. \square

Definition 3.4.4. Let X be an assignment to the SAT variables \mathcal{X} that satisfies the SAT clauses in \mathcal{C} . Given an edge $\langle i, e \rangle$, let $L(i, e) = \{l_m \mid label(i, e, m)\}$ be the label set induced by X . We

define a mapping $\mathcal{G}(X) = \langle G, \mathcal{I}(G) \rangle$ from X to a graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ and associated graph indexing $\mathcal{I}(G) = \langle f, \{\Gamma_u\}_{u \in \mathcal{V}} \rangle$ as follows:

- The set of nodes is $\mathcal{V} = \{v_1, \dots, v_n\}$, where n is the largest node index in the assignment, and $f(v_i) = i$ for each $v_i \in \mathcal{V}$.
- The set of edges is $\mathcal{E} = \{(v_i, v_j, L(i, e)) \mid \text{ed}(i, j, e)\}$ and $\Gamma_{v_i}(L(i, e)) = e$.
- The parent function of each $v_j \in \mathcal{V}$ equals $\Pi_{\mathcal{I}}(v_j) = \langle i, e \rangle$ if $\text{sm}(i, j, e)$ is true.

Theorem 3.4.2. *Given an assignment X to the SAT variables \mathcal{X} that satisfies all clauses in \mathcal{C} , the mapping $\mathcal{G}(X) = \langle G, \mathcal{I}(G) \rangle$ induces a graph G in the class \mathcal{G} and a well-defined graph indexing $\mathcal{I}(G)$.*

Proof. We first show that the induced graph indexing $\mathcal{I}(G)$ is well-defined. Clearly f is a bijection onto $\{1, \dots, |\mathcal{V}|\}$ by definition. We next show that Γ_{v_i} is a bijection onto $\{1, \dots, |\mathcal{E}^o(v_i)|\}$ for each node $v_i \in \mathcal{V}$. Clause 10 ensures that $\text{ed}(i, j, e)$ and $\text{ed}(i, j', e)$ cannot be true simultaneously for $j \neq j'$. Due to Clause 9, if edge $\langle i, e \rangle$ is defined for $e > 1$, then so is $\langle i, e - 1 \rangle$. Applying this argument recursively implies that $\langle i, e \rangle$ is uniquely defined for $e \in \{1, \dots, |\mathcal{E}^o(v_i)|\}$, where $|\mathcal{E}^o(v_i)|$ is the largest integer of an outgoing edge from i .

We next show that the induced label set $L(i, e)$ on each outgoing edge $\langle i, e \rangle$ from i is unique. Clause 13 implies that for each edge $\langle i, e \rangle$ with $e > 1$, $\text{lt}(i, e - 1, e, m)$ is true for at least one label $l_m \in \mathcal{L}$. Clauses 11 and 12 ensure that $\text{lt}(i, e - 1, e, m)$ is true only if there exists $m' \leq m$ such that $\neg \text{label}(i, e - 1, m')$ and $\text{label}(i, e, m')$ are true. For the smallest such m' there cannot exist $m'' < m'$ such that $\text{label}(i, e - 1, m'')$ and $\neg \text{label}(i, e, m'')$ are true, else Clause 14 would be violated for m'' . Hence label l_m satisfies the condition in Definition 3.4.1 with respect to the induced label sets $L(i, e - 1)$ and $L(i, e)$, implying $L(i, e - 1) < L(i, e)$.

In conclusion, we have shown that $\langle i, e \rangle$ is uniquely defined for $e \in \{1, \dots, |\mathcal{E}^o(v_i)|\}$, and that $L(i, e - 1) < L(i, e)$ holds for each pair of consecutive integers in $\{1, \dots, |\mathcal{E}^o(v_i)|\}$. Since Γ_{v_i} is defined as $\Gamma_{v_i}(L(i, e)) = e$ for each $e \in \{1, \dots, |\mathcal{E}^o(v_i)|\}$, this implies that Γ_{v_i} is a well-defined bijection from $\mathcal{E}^o(v_i)$ to $\{1, \dots, |\mathcal{E}^o(v_i)|\}$.

We also need to show that the induced parent function $\Pi_{\mathcal{I}}$ is well-defined, i.e. that for each $j > 1$, $\text{sm}(i, j, e)$ is true for a single i and e , and that $\Pi_{\mathcal{I}}(v_j) = \langle i, e \rangle$ is consistent with the definition of $\Pi_{\mathcal{I}}$. Clauses 1 and 2 imply that $\text{pa}(i, j)$ is true for a single i . Clauses 3, 5 and 6 imply that $\text{sm}(i, j, e)$ can only be true for the same i and j as $\text{pa}(i, j)$, and that $\text{sm}(i, j, e)$ is true for a single e . For $\Pi_{\mathcal{I}}(v_j) = \langle i, e \rangle$ to hold, G has to contain the edge $\langle v_i, v_j, L(i, e) \rangle$, which is guaranteed by Clause 7. Moreover, G cannot contain any edge $\langle v_{i'}, v_j, L(i', e') \rangle$ such that $\langle i', e' \rangle < \langle i, e \rangle$, which is guaranteed by Clauses 4 and 8.

We finally show that the induced graph G belongs to the class \mathcal{G} , i.e. all three assumptions on the graphs are satisfied. We satisfy Assumption 3.4.1 by designating v_1 as the start node. We have already shown above that the induced label set $L(i, e)$ on each outgoing edge $\langle i, e \rangle$ from i is unique, satisfying Assumption 3.4.3. It remains to show that Assumption 3.4.2 holds, i.e. that each node v_j , $j > 1$, is reachable from v_1 . Since $\text{sm}(i, j, e)$ is true for a single i and e such that $i < j$, G has to contain the edge $\langle v_i, v_j, L(i, e) \rangle$ due to Clause 7. Aggregating these incoming edges for all nodes different from v_1 results in a BFS subtree rooted in v_1 , and each node v_j , $j > 1$, is reachable from v_1 in this subtree. \square

We combine the previous theoretical results and show there is a one-to-one correspondence between the BFS traversal of a graph and a solution to the SAT encoding. By Theorem 3.4.2, the mapping $\mathcal{G}(X)$ of a satisfying assignment X to the SAT clauses in \mathcal{C} induces a graph $G \in \mathcal{G}$ and a well-defined graph indexing $\mathcal{I}(G)$. By Theorem 3.4.1, $\mathcal{I}(G)$ is a BFS traversal since X satisfies all clauses. Finally, since by Lemma 3.4.1 each graph $G \in \mathcal{G}$ has a unique BFS traversal, it follows that the SAT encoding cannot generate two permutations of node integers that represent the same graph G . Hence the SAT encoding breaks the symmetries in graphs such as those in Figure 3.4.

3.4.3 Application to Reward Machines

In this section, we show that reward machines are a particular case of the labeled directed graphs on which we have formalized our symmetry breaking method. We present two ASP encodings of the method for its application to RMs. Both encodings build on the factual ASP representation of RMs, and their respective descriptions are preceded by a section introducing commonalities between these. The first encoding is a direct translation from the SAT clauses in the previous section, whereas the second encoding results from leveraging ASP-specific aspects for improved efficiency. These rules are used to speed up the learning of the RMs described in Chapter 4. Crucially, since the proposed graph indexing is unique due to Lemma 3.4.1, the RM learner can only represent each graph in one way, precluding multiple symmetric variations; in fact, any unique graph indexing could have been used for this purpose.

Reward Machines as Labeled Directed Graphs

A reward machine $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$ is a special case of labeled directed graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ in the class \mathcal{G} . The set of RM states \mathcal{U} corresponds to the set of nodes \mathcal{V} , and the logical transition function φ corresponds to the set of edges \mathcal{E} . Besides, reward machines comply with all three assumptions we made about graphs in the class \mathcal{G} :

- Assumption 3.4.1 holds because reward machines have an initial state u^0 .
- Assumption 3.4.2 is enforced through the following rules. The first rule defines the initial state u^0 to be reachable, while the second rule indicates that a state is reachable if it has an incoming edge from a reachable state. The third rule enforces all states to be reachable.

$$\left\{ \begin{array}{l} \text{reachable}(u^0). \\ \text{reachable}(Y) :- \text{reachable}(X), \text{ed}(X, Y, _). \\ :- \text{not reachable}(X), \text{state}(X). \end{array} \right\}$$

- Assumption 3.4.3 holds for the following two reasons. First, since the logical transition function is deterministic, the formulas labeling two outgoing edges from a given state to two different states are mutually exclusive and, hence, different. Second, the DNF formula between two states never includes a disjunct twice or more; thus, there cannot be two edges between the same state-pair labeled by the same formula.

Even though reward machines comply with the three assumptions on labeled directed graphs, there are two differences between them:

1. The edges of an RM are labeled by propositional formulas over a set of propositions \mathcal{P} , whereas the edges of a labeled directed graph are defined over a set of labels \mathcal{L} .⁵
2. The edge indices in an indexed labeled directed graph differ from those in our ASP representation of an RM. The graph indexing presented for labeled directed graphs assigns a different index to each of the outgoing edges from a node; in contrast, indices are unique for each pair of states in the ASP representation of an RM (i.e., indices can be repeated for edges between other pairs).

To address these discrepancies, we map the representation for RMs into a representation for labeled directed graphs on top of which we define the symmetry breaking constraints. In what follows, we first describe a mapping from RMs to labeled directed graphs; next, we introduce two ASP encodings (a SAT-based one and an efficient alternative) of the symmetry breaking rules that build on the previous mapping.

Common Encoding

In the following paragraphs, we describe the rules shared by the two encodings presented later.

Mapping Propositions into Labels. We denote the set of labels characterizing labeled directed graphs by \mathcal{L}_{sb} . The proposition set \mathcal{P} used to label the edges of an RM is mapped into a set of labels \mathcal{L}_{sb} , which consists of integer values for easy comparison. Each of these integers encodes either a proposition or its negation. Formally, given a proposition set \mathcal{P} and a bijective function $f : \mathcal{P} \rightarrow \{1, \dots, |\mathcal{P}|\}$ mapping each proposition to a different integer between 1 and $|\mathcal{P}|$, the set of labels is

$$\mathcal{L}_{sb} = \{f(p), |\mathcal{P}| + f(p) \mid p \in \mathcal{P}\}, \quad (3.1)$$

where $f(p)$ is the label associated with p and $f(p) + |\mathcal{P}|$ is the label associated with $\neg p$; therefore, \mathcal{L}_{sb} consists of $2|\mathcal{P}|$ labels, where labels $1, \dots, |\mathcal{P}|$ correspond to the propositions, and labels $|\mathcal{P}| + 1, \dots, 2|\mathcal{P}|$ correspond to their respective negations. The mapping is encoded in ASP using the following atoms:

- `prop_id(p, l)` indicates that proposition $p \in \mathcal{P}$ is associated with label l .
- `num_props(i)` indicates that the proposition set has size i .
- `valid_sb_label(l)` indicates that l is a label.

We simply ground the above atoms according to their descriptions:

$$\{\text{prop_id}(p, f(p)). \mid p \in \mathcal{P}\} \cup \{\text{num_props}(|\mathcal{P}|).\} \cup \{\text{valid_sb_label}(l). \mid 1 \leq l \leq 2|\mathcal{P}|\}.$$

To complete the mapping, we map the formulas on the edges into label sets leveraging the factual representation from Definition 3.3.3. Labels are represented using facts `label` similar to the variables employed in the SAT encoding; however, these facts differ across the two proposed encodings, so we will describe them in the respective sections.

⁵The labels from the symmetry breaking formalization should not be confused with the labels observed during the interaction with a labeled MDP. The former could be anything comparable (e.g., integers, strings), while the latter are exclusively sets of propositions. We disambiguate these where needed in the rest of the section.

State Ordering. To enforce a BFS traversal on the RMs, the states are assigned an integer index for easy comparison akin to the SAT encoding (see Section 3.4.2). The atom `state_id(u, i)` denotes that the RM state u has index i and its ground instances are:

$$\{\text{state_id}(u^i, i) \mid u^i \in \mathcal{U} \setminus \{u^A, u^R\}\}.$$

Unlike the bijection f , we here (without loss of generality) assign indices starting from 0 instead of 1 since the first state is u^0 . Besides, the accepting and rejecting states are not given an index (i.e., they are excluded from the BFS ordering) as they are fixed and hence cannot be interchanged with other states, and they cannot be the parent of any other state since they do not have outgoing transitions by Assumption 3.2.2.

Both encodings use the rules below to compare the indices of two states easily. The first rule defines the atom `state_lt(u, u')`, which expresses that the index of state u is lower than that of u' ; likewise, the second rule defines the atom `state_leq(u, u')`, which expresses that the index of state u is lower or equal than that of u' .

$$\left\{ \begin{array}{l} \text{state_lt}(X, Y) :- \text{state_id}(X, \text{XID}), \text{state_id}(Y, \text{YID}), \text{XID} < \text{YID}. \\ \text{state_leq}(X, Y) :- \text{state_id}(X, \text{XID}), \text{state_id}(Y, \text{YID}), \text{XID} \leq \text{YID}. \end{array} \right\}$$

SAT-Based Encoding

This encoding is a direct translation from the SAT encoding introduced in Section 3.4.2, which includes three parts: (i) a mapping from the edge indices used by RMs into the edge indices used in the symmetry breaking method, (ii) a mapping from formulas over propositions into label sets using the previously introduced mapping from propositions into labels, and (iii) a set of ASP rules to represent the SAT clauses.

Edge Index Mapping. The range of possible edge indices in the target representation is given by the atom `edge_id(i)`, where i is an edge index, whose ground instances are:

$$\{\text{edge_id}(i) \mid 1 \leq i \leq (|\mathcal{U}| - 1)\kappa\},$$

where κ is the maximum number of edges from one state to another; hence, $(|\mathcal{U}| - 1)\kappa$ is the maximum number of outgoing edges from a state since each state can have edges to $|\mathcal{U}| - 1$ different states.

The mapping is represented through facts of the form `mapping(u, v, e, e')` indicating that edge e between u and v is mapped into e' , and enforced using the rules below. The first rule describes that an edge index E from X to Y is mapped into exactly one edge index EE in the range given by the `edge_id` facts.⁶ The second rule enforces two outgoing edges from a state X to two different states Y and Z to be mapped into different edge indices. The third rule enforces two edge indices E and EP between the same state-pair $\langle X, Y \rangle$ to be mapped into different edge indices. The fourth rule indicates that if there are two edge indices E and EP between states X and Y such that $E < EP$, then

⁶The head of the choice rule employs a *conditional literal* (Gebser et al., 2019) of the form $\text{l}_0 : \text{l}_1, \dots, \text{l}_n$. In this case, $\text{l}_0, \dots, \text{l}_n$ are atoms, and $\text{l}_1, \dots, \text{l}_n$ constitute the condition. This choice rule thus compactly expresses a choice over a set of atoms; indeed, the rule is equivalent to $1\{\text{mapping}(X, Y, E, 1); \text{mapping}(X, Y, E, 2)\}1 :- \text{ed}(X, Y, E)$. if `edge_id(i)` is true only for $i \in \{1, 2\}$.

the indices they are mapped into (EE and EEP) must preserve the ordering ($EE < EEP$).

$$\left\{ \begin{array}{l} 1\{\text{mapping}(X, Y, E, EE) : \text{edge_id}(EE)\} 1 : -\text{ed}(X, Y, E). \\ :-\text{mapping}(X, Y, -, EE), \text{mapping}(X, Z, -, EE), Y < Z. \\ :-\text{mapping}(X, Y, E, EE), \text{mapping}(X, Y, EP, EE), E < EP. \\ :-\text{ed}(X, Y, E), \text{ed}(X, Y, EP), E < EP, \text{mapping}(X, Y, E, EE), \text{mapping}(X, Y, EP, EEP), EE > EEP. \end{array} \right\}$$

Given the mapping above, we redefine the **ed**, **pos** and **neg** facts used in the factual representation of the RMs using the following rules:

$$\left\{ \begin{array}{l} \text{map_ed}(X, Y, EP) : -\text{ed}(X, Y, E), \text{mapping}(X, Y, E, EP). \\ \text{map_pos}(X, Y, EP, P) : -\text{pos}(X, Y, E, P), \text{mapping}(X, Y, E, EP). \\ \text{map_neg}(X, Y, EP, P) : -\text{neg}(X, Y, E, P), \text{mapping}(X, Y, E, EP). \end{array} \right\}.$$

The **map_ed**, **map_pos**, and **map_neg** predicates are used in the ASP encoding of the symmetry breaking constraints explained later.

Mapping Formulas into Label Sets. Formulas over a set of propositions are mapped into label sets by leveraging the previously described proposition-to-label and edge index mappings. The first rule sets PID as a label of edge E from X if the corresponding proposition P appears positively in that edge. The second rule sets $PID+N$ as a label of edge E from X if the corresponding proposition P appears negatively in that edge and N is the number of propositions.

$$\left\{ \begin{array}{l} \text{label}(X, E, PID) : -\text{map_pos}(X, Y, E, P), \text{prop_id}(P, PID). \\ \text{label}(X, E, PID+N) : -\text{map_neg}(X, Y, E, P), \text{prop_id}(P, PID), \text{num_props}(N). \end{array} \right\}$$

Symmetry Breaking Rules. Analogously to the SAT encoding, the ASP representation includes three parts. First, we introduce the rule set enforcing the indexing given by the BFS traversal on the RM. These rules are defined in terms of an auxiliary atom $\text{ed_sb}(u, u', i)$ equivalent to $\text{map_ed}(u, u', i)$ but only defined for those states u' that have a state index (i.e., all states except for the accepting and rejecting states):

$$\text{ed_sb}(X, Y, E) : -\text{map_ed}(X, Y, E), \text{state_id}(Y, -).$$

The resulting set of rules for Clauses 1–8 in the SAT encoding is:

$$\left\{ \begin{array}{l} 1\{\text{pa}(X, Y) : \text{state}(X), \text{state_lt}(X, Y)\} : -\text{state}(Y), \text{state_id}(Y, YID), YID > 0. \\ :-\text{pa}(X, Y), \text{pa}(XP, Y), \text{state_lt}(X, XP), \text{state_lt}(XP, Y). \\ 1\{\text{sm}(X, Y, E) : \text{edge_id}(E)\} : -\text{pa}(X, Y). \\ :-\text{pa}(X, Y), \text{ed_sb}(XP, YP, -), \text{state_lt}(XP, X), \text{state_leq}(Y, YP). \\ :-\text{sm}(X, Y, -), \text{not pa}(X, Y). \\ :-\text{sm}(X, Y, E), \text{sm}(X, Y, EP), E < EP. \\ :-\text{sm}(X, Y, E), \text{not ed_sb}(X, Y, E). \\ :-\text{sm}(X, Y, E), \text{ed_sb}(X, YP, EP), \text{state_lt}(X, Y), \text{state_leq}(Y, YP), EP < E. \end{array} \right\}.$$

The next rule set encodes Clauses 9 and 10, which enforces edge indices to be unique between 1

and the number of outgoing edges from a given state:

$$\left\{ \begin{array}{l} :- \text{map_ed}(X, Y, E), \text{not map_ed}(X, -, E-1), E > 1. \\ :- \text{map_ed}(X, Y, E), \text{map_ed}(X, Z, E), Y < Z. \end{array} \right\}.$$

The rule set below encodes Clauses 11–14 in ASP. Clauses 11 and 12 are divided into two rules respectively to cover the different label values, i.e. $L=1$ (there are no lower-valued labels to compare to) and $L>1$ (there are lower-valued labels to compare to).

$$\left\{ \begin{array}{l} :- \text{lt}(X, E-1, E, L), \text{label}(X, E-1, L), \text{not lt}(X, E-1, E, L-1), E > 1, L > 1. \\ :- \text{lt}(X, E-1, E, L), \text{label}(X, E-1, L), E > 1, L = 1. \\ :- \text{lt}(X, E-1, E, L), \text{not label}(X, E, L), \text{not lt}(X, E-1, E, L-1), E > 1, L > 1. \\ :- \text{lt}(X, E-1, E, L), \text{not label}(X, E, L), E > 1, L = 1. \\ 1\{\text{lt}(X, E-1, E, L) : \text{valid_sb_label}(L)\} :- \text{map_ed}(X, Y, E), E > 1. \\ :- \text{not lt}(X, E-1, E, L), \text{label}(X, E-1, L), \text{not label}(X, E, L), \text{map_ed}(X, -, E), E > 1. \end{array} \right\}$$

Efficient Alternative Encoding

The encoding we describe here is an alternative to the SAT-based encoding that performs symmetry breaking more efficiently. Like the SAT-based approach, this encoding addresses the fact that edge indexing required by the symmetry breaking method differs from that in the RM representation; however, we here do not use an intermediate mapping and directly operate on the edge indices from the RM representation. Remember that the edge indexing used in the symmetry breaking is such that each outgoing edge from a given state has a different index; that is, an integer number uniquely identifies each edge. Here, we preserve the same uniqueness principle by expressing edges as $(u, (v, e))$, meaning that there is an edge from u to v with edge index e . The tuple (v, e) uniquely identifies each outgoing edge from u .

The section is divided into two parts. First, like in the SAT-based approach, we map propositional formulas on the edges into label sets. Then, we describe the set of ASP rules for breaking symmetries.

Mapping Formulas into Label Sets. We use the atoms $\text{label}(u, (v, e), l)$ to express that label l appears in the edge from state u to state v with index e . The rule set below transforms the formulas over propositions into label sets. The first rule sets PID as a label of edge (Y, E) from X if the corresponding proposition P appears positively in that edge. Likewise, the second rule sets $\text{PID}+N$ as a label of edge (Y, E) from X if the corresponding proposition P appears negatively in that edge and N is the number of propositions.

$$\left\{ \begin{array}{l} \text{label}(X, (Y, E), \text{PID}) :- \text{pos}(X, Y, E, P), \text{prop_id}(P, \text{PID}). \\ \text{label}(X, (Y, E), \text{PID}+N) :- \text{neg}(X, Y, E, P), \text{prop_id}(P, \text{PID}), \text{num_props}(N). \end{array} \right\} \quad (3.2)$$

While the `label` predicate in the SAT-based encoding only used the edge index for referring to an outgoing edge, we here use a state-edge pair, as explained before.

Symmetry Breaking Rules. We start by describing the rules that enforce outgoing edges from a given state to be ordered by their respective label sets. The atoms $\text{ed_lt}(u, (v, e), (v', e'))$ indicate that the edge from u to v with edge index e is lower than the edge from u to v' with edge index e' .

The rule set below encodes how this ordering is determined and what constraints are imposed on it. The first rule determines that given two outgoing edges from X , (Y, E) and (YP, EP) , either (Y, E) is lower than (YP, EP) or vice versa. Now, the order between outgoing edges from a state X must respect two constraints:

- The second rule enforces transitivity; that is, if **Edge1** is lower than **Edge2**, and **Edge2** is lower than **Edge3**, then **Edge1** must be lower than **Edge3**.⁷
- The third rule enforces that two edges to the same state Y must be ordered according to their edge index; that is, given edges (Y, E) and (Y, EP) from X such that $E < EP$, edge (Y, E) must be lower than (Y, EP) .

$$\left\{ \begin{array}{l} 1\{\text{ed_lt}(X, (Y, E), (YP, EP)); \text{ed_lt}(X, (YP, EP), (Y, E))\} 1 :- \text{ed}(X, Y, E), \text{ed}(X, YP, EP), \\ \hspace{10em} (Y, E) < (YP, EP). \\ :- \text{ed_lt}(X, \text{Edge1}, \text{Edge2}), \text{ed_lt}(X, \text{Edge2}, \text{Edge3}), \text{not ed_lt}(X, \text{Edge1}, \text{Edge3}), \\ \hspace{2em} \text{Edge1} \neq \text{Edge3}. \\ :- \text{ed_lt}(X, (Y, E), (Y, EP)), \text{ed}(X, Y, E), \text{ed}(X, Y, EP), E > EP. \end{array} \right\} \quad (3.3)$$

The previous rule set guesses an ordering for the outgoing edges from a given state; however, this ordering must comply with that of the label sets given in Definition 3.4.1. We use the atoms $\text{label_lt}(u, (v, e), (v', e'), l)$ to indicate there is a label $l' \leq l$ that appears in edge (v', e') and does not appear in a lower edge (v, e) , both being outgoing edges from u . These atoms encode the first condition in Definition 3.4.1 up to a specific label. The rule set below prunes solutions where outgoing edges do not follow the established label ordering criteria. The first rule indicates that $\text{label_lt}(X, \text{Edge1}, \text{Edge2}, L)$ is true if **Edge1** is lower than **Edge2**, and the label L does not appear in **Edge1** and appears in **Edge2**. The second rule states that label_lt is true for a valid label $L+1$ if it is true for L . The third rule states that if **Edge1** is lower than **Edge2**, then the label set on **Edge1** must be lower than that on **Edge2**. The three last literals in the last rule enforce both conditions from Definition 3.4.1.

$$\left\{ \begin{array}{l} \text{label_lt}(X, \text{Edge1}, \text{Edge2}, L) :- \text{ed_lt}(X, \text{Edge1}, \text{Edge2}), \text{not label}(X, \text{Edge1}, L), \\ \hspace{10em} \text{label}(X, \text{Edge2}, L). \\ \text{label_lt}(X, \text{Edge1}, \text{Edge2}, L+1) :- \text{label_lt}(X, \text{Edge1}, \text{Edge2}, L), \text{valid_sb_label}(L+1). \\ :- \text{ed_lt}(X, \text{Edge1}, \text{Edge2}), \text{label}(X, \text{Edge1}, L), \text{not label}(X, \text{Edge2}, L), \\ \hspace{2em} \text{not label_lt}(X, \text{Edge1}, \text{Edge2}, L). \end{array} \right\}$$

The following set of rules imposes that lower edge indices cannot be left unused. First, we define a fact $\text{edge_id}(i)$ for each possible edge index i between 1 and κ , where κ is the maximum number of edges from one state to another. Second, the constraint indicates that if there is an edge from X to Y with index $E > 1$, there must be an edge between the same states but with index $E-1$ as well.

$$\left\{ \begin{array}{l} \text{edge_id}(1..\kappa). \\ :- \text{ed}(X, Y, E), \text{not ed}(X, Y, E-1), \text{edge_id}(E), E > 1. \end{array} \right\} \quad (3.4)$$

Finally, we describe the rules for enforcing the BFS traversal on the RM. Like in the SAT-based approach, we ground $\text{ed_sb}(u, u', e)$ atoms for all edges except for those directed to an unindexed

⁷For conciseness, state-edge variable pairs such as (Y, E) are represented using a single variable (e.g., **Edge1**).

state (i.e., the accepting and rejecting states):

$$\text{ed_sb}(\mathbf{X}, \mathbf{Y}, \mathbf{E}) :- \text{ed}(\mathbf{X}, \mathbf{Y}, \mathbf{E}), \text{state_id}(\mathbf{Y}, _). \quad (3.5)$$

Next, we introduce the $\text{pa}(u, v)$ atoms denoting that state u is the parent of v in the BFS subtree, which is equivalent to the variable $pa(i, j)$ from the SAT encoding and used in the rule set below to enforce the BFS ordering. The first rule defines that state \mathbf{X} is the parent of \mathbf{Y} if there is an edge from \mathbf{X} to \mathbf{Y} , \mathbf{X} has a lower index than \mathbf{Y} , and there is no state \mathbf{Z} whose index is lower than \mathbf{X} 's and has an edge to \mathbf{Y} .⁸ The second rule indicates that all indexed states except for the initial state must have a parent. The third rule imposes the BFS ordering similarly to Clause 4 in the SAT encoding. The first two rules encode Clauses 1 and 2 of the SAT encoding.

$$\left\{ \begin{array}{l} \text{pa}(\mathbf{X}, \mathbf{Y}) :- \text{ed_sb}(\mathbf{X}, \mathbf{Y}, _), \text{state_lt}(\mathbf{X}, \mathbf{Y}), \\ \quad \# \text{false} : \text{ed_sb}(\mathbf{Z}, \mathbf{Y}, _), \text{state_lt}(\mathbf{Z}, \mathbf{X}). \\ :- \text{state_id}(\mathbf{Y}, \text{YID}), \text{YID} > 0, \text{not pa}(_, \mathbf{Y}). \\ :- \text{pa}(\mathbf{X}, \mathbf{Y}), \text{ed_sb}(\mathbf{X}, \mathbf{Y}, _), \text{state_lt}(\mathbf{X}, \mathbf{Y}), \text{state_leq}(\mathbf{Y}, \mathbf{Y}). \end{array} \right\}$$

We now need to enforce that the BFS children from a given state are correctly ordered; that is, those children pointed by lower edges should be identified by lower state indices. The $\text{state_ord}(u)$ atoms indicate that state u is properly ordered with respect to its siblings (i.e., other states with the same parent state). The following rule set enforces this ordering. The first rule defines that a state \mathbf{Y} is correctly ordered with respect to its siblings if the edge from their parent \mathbf{X} to \mathbf{Y} , i.e. (\mathbf{Y}, \mathbf{E}) , is lower than the edge to another state $(\mathbf{Y}, \mathbf{EP})$ if $\mathbf{Y} < \mathbf{Y}, \mathbf{EP}$; that is, edges must be ordered according to the order of the state indices. The second rule enforces all states with a parent to be correctly ordered with respect to their siblings.

$$\left\{ \begin{array}{l} \text{state_ord}(\mathbf{Y}) :- \text{ed_sb}(\mathbf{X}, \mathbf{Y}, \mathbf{E}), \text{pa}(\mathbf{X}, \mathbf{Y}), \\ \quad \# \text{false} : \text{ed_sb}(\mathbf{X}, \mathbf{Y}, \mathbf{EP}), \text{state_lt}(\mathbf{Y}, \mathbf{Y}, \mathbf{EP}), \text{ed_lt}(\mathbf{X}, (\mathbf{Y}, \mathbf{EP}), (\mathbf{Y}, \mathbf{E})). \\ :- \text{pa}(_, \mathbf{Y}), \text{not state_ord}(\mathbf{Y}). \end{array} \right\}$$

3.5 Summary

In this chapter, we formalized the tasks considered throughout the thesis and the RMs capturing their reward functions. In the next chapter, we outline RL methods for exploiting the structure of these RMs. Furthermore, we propose an RM learning method that leverages the ASP representation and the symmetry breaking method presented here; crucially, the latter accelerates learning by discarding multiple equivalent RMs from the search space.

⁸What follows $\# \text{false}$ must not hold in order to make the body of the rule true.

Chapter 4

Learning and Exploiting Reward Machines

In this chapter, we introduce two RL algorithms for *exploiting* the structure of a given reward machine (Section 4.1), followed by a method for *learning* reward machines from traces (Section 4.2). The described exploitation and learning methods are then put together into an algorithm that *interleaves* them (Section 4.3).

4.1 Exploiting Reward Machines

We describe two policy learning methods that exploit the RM structure. Each method is characterized by a different way of using *options*:

1. Learning an option for each edge in the RM and a metapolicy to choose between options in each RM state (Section 4.1.1).
2. Learning an option for each RM state (Section 4.1.2).

In both cases, the structure revealed by a reward machine M is exploited as follows. First, the agent selects an option when it reaches an RM state. Note that in (1) there can be multiple options to choose from, whereas in (2) there is a single option. Once an option is chosen, the agent selects actions according to that option’s policy until its termination. An option terminates when either (i) the episode ends, or (ii) a formula labeling an edge from the current RM state is satisfied. After the agent experiences a tuple $\langle \mathbf{s}_t, a_t, \mathbf{s}_{t+1}, \mathcal{L}_{t+1} \rangle$ in RM state u at timestep t , it transitions to RM state $u' = \delta_M(u, \mathcal{L}_{t+1})$ following the state-transition function of the RM.

Reward machines, as described in Section 2.1.5, compactly represent traces. The combination of an environment state and an RM state makes rewards Markovian as long as the combination of an environment state and a trace also makes them Markovian. Because each option is only executed for a specific trace (represented by the current RM state), we can define an option’s policy over environment states only.

In what follows, we describe the features of the RL methods including: how options are modeled, how policies are learned, and which optimality guarantees they have.

4.1.1 Learning an Option for each Edge and a Metapolicy for each State (HRL)

The edges of a reward machine are labeled by propositional formulas over a set of propositions \mathcal{P} . Intuitively, each of these formulas represents a subgoal of the task represented by the RM. An intuitive approach for exploiting the RM structure consists of learning (i) an option that aims to reach an environment state whose label satisfies a given formula, and (ii) a metapolicy that learns which option to take at each RM state. Since decisions are taken at two hierarchical levels (i.e. two timescales), we refer to this approach as HRL (Hierarchical Reinforcement Learning).

Option Modeling

Given a reward machine $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$, the set of options in a *non-terminal* RM state $u \in \mathcal{U}$ (i.e., with outgoing transitions to other states) is

$$\Omega_u = \{\omega_{u,\phi} \mid \phi \in \varphi(u, u'), u' \in \mathcal{U}, \phi \neq \perp\},$$

where $\omega_{u,\phi}$ is the option that aims to satisfy a non-false disjunct ϕ in a DNF formula from RM state $u \in \mathcal{U}$. Formally, each option is a tuple $\omega_{u,\phi} = \langle \mathcal{I}_u, \pi_\phi, \beta_u \rangle$ where:¹

- The initiation set $\mathcal{I}_u = \mathcal{S}$ consists of all the states in the MDP; that is, any option available in u can be started in any state.
- The policy $\pi_\phi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ maps an environment state into a probability distribution over primitive actions with the aim of observing a label that satisfies ϕ .
- The termination condition $\beta_u : \mathcal{S} \rightarrow [0, 1]$ indicates that the option terminates if any formula on an outgoing edge from u holds (i.e., the formula does not necessarily have to be ϕ). Formally,

$$\beta_u(s) = \begin{cases} 1 & \text{if } l(s) \models \varphi(u, u') \text{ for some } u' \in \mathcal{U}; \\ 0 & \text{otherwise.} \end{cases}$$

During the interaction with the MDP, the agent determines termination using the label returned by the environment instead of evaluating the labeling function l itself. When the RM does not perfectly capture histories, options may also terminate when the history is terminal (i.e., the terminal indicator observed by the agent is true); for instance, this occurs when a goal (resp. dead-end) trace does not end in the accepting (resp. rejecting) state during the learning of an RM (see Section 4.3).

Note that even if two options share the same policy (i.e., they are associated with the same formula ϕ), the termination function may differ since it depends on the RM state u .

In the case of a *terminal* RM state u (e.g., the accepting and rejecting states), decision-making only occurs when the RM does not perfectly capture histories; for instance, when the traversal of an incomplete trace finishes in the accepting or the rejecting states. The set of available options $\Omega_u = \mathcal{A}$ is constituted by the primitive actions; hence, each option lasts one step. While this case

¹For simplicity, the option components are only subscripted with the parameters of the option they depend on.

may seem unlikely, it arises during RM learning (see Section 4.3) since a target RM is rarely learned after a single attempt.

Policy Learning

In this approach, decisions are taken at two levels by learning two types of policies: (i) metapolicies (i.e., policies over options) and (ii) option policies. We describe these policies for both the tabular and the function approximation cases in the following paragraphs.

Metapolicies. A *metapolicy* $\Pi_u : \mathcal{S} \rightarrow \Delta(\Omega_u)$ in RM state $u \in \mathcal{U}$ maps an environment state into a probability distribution over the options available at u . These policies are learned using SMDP Q-learning with ϵ -greedy exploration. Given an experience tuple $\langle \mathbf{s}_t, \omega_t, \mathbf{s}_{t+k} \rangle$, where $\omega_t \in \Omega_u$ is an option taken in RM state u at timestep t , the update rule is the following:

$$q_u(\mathbf{s}_t, \omega_t) = q_u(\mathbf{s}_t, \omega_t) + \alpha \left(r + \gamma^k \max_{\omega' \in \Omega_{u'}} q_{u'}(\mathbf{s}_{t+k}, \omega') - q_u(\mathbf{s}_t, \omega_t) \right), \quad (4.1)$$

where k is the number of steps between \mathbf{s}_t and \mathbf{s}_{t+k} , r is the cumulative discounted reward over this time, and u' is the RM state when the option terminates. The discounted term of the target depends on u' (i.e., the policy in u depends on that in u') and becomes 0 when s_{t+k}^T is true (i.e., the history is terminal) since there is no applicable action thereafter.² The reward is that emitted by the reward-transition function of the RM; therefore, by Assumption 3.2.3, the cumulative discounted reward r only has a non-zero value of γ^{k-1} when u' is the accepting state since a reward of +1 is given after $k-1$ steps in that situation.

In the function approximation case, the option-values are approximated through a DQN with parameters $\boldsymbol{\theta}$ and a target DQN with parameters $\boldsymbol{\theta}^-$. All option experiences are stored in a single replay buffer \mathcal{D}_M . The loss function performs an SMDP Q-learning update analogous to that in Equation 4.1:

$$\mathbb{E}_{\langle \mathbf{s}_t, u, \omega_t, \mathbf{s}_{t+k}, u' \rangle \sim U(\mathcal{D}_M)} \left[\left(r + \gamma^k \max_{\omega' \in \Omega_{u'}} q(\mathbf{s}_{t+k}, u', \omega'; \boldsymbol{\theta}^-) - q(\mathbf{s}_t, u, \omega_t; \boldsymbol{\theta}) \right)^2 \right].$$

Unlike the tabular case, the RM state is represented using a one-hot vector and passed as an input to the DQN. The DQN outputs are determined by the formulas appearing in the RM and the number of actions. For instance, given the RM in Figure 3.2, a value is output for each entry in $\{\blacktriangle \wedge \neg o \wedge \neg *, \blacktriangle \wedge o \wedge \neg *, o \wedge \neg *, *\} \cup \{\text{up, down, left, right}\}$. High-level options and primitive actions are respectively available at non-terminal and terminal RM states. Therefore, masking must be performed to avoid selecting options unavailable in a given RM state.

Option Policies. An *option policy* $\pi_\phi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ aiming to satisfy a formula ϕ is not learned using the rewards from the RM. Instead, we use a pseudoreward function $r_\phi : 2^{\mathcal{P}} \times \{\perp, \top\} \times \{\perp, \top\} \rightarrow$

²If the RM perfectly captures any history and s^T is \top , u' is either u^A or u^R .

\mathbb{R} defined as

$$r_\phi(\mathcal{L}, s^T, s^G) = \begin{cases} r_{success} & \text{if } \mathcal{L} \models \phi; \\ r_{deadend} & \text{if } s^T = \top \wedge s^G = \perp; \\ r_{step} & \text{otherwise,} \end{cases}$$

where $r_{success} > 0$ is given when the next label satisfies ϕ ; $r_{deadend} \leq 0$ is given if the history becomes a dead-end history; and $r_{step} \leq 0$ is given after every step otherwise. We remark on two subtleties of this function:

- The second case incurs an *assumption*: dead-end histories depend on the last state, not on the history of labels (e.g., all dead-end histories end on a decoration location in OFFICEWORLD). When the assumption does not hold, learning a policy shared by several options might be unstable since rewards become non-stationary.
- The last case includes the scenario where a formula different from ϕ labeling an edge from the current RM state is satisfied.

The evaluation performed in Chapter 5 shows significant performance gains by penalizing the agent for dead-end histories and after each step, i.e. setting $r_{deadend}$ and r_{step} to be strictly lower than 0.

These policies are learned using Q-learning with ϵ -greedy exploration. The update rule for a given formula ϕ and an experience tuple $\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle$ is:

$$q_\phi(s_t, a_t) = q_\phi(s_t, a_t) + \alpha \left(r_\phi(\mathcal{L}_{t+1}, s_{t+1}^T, s_{t+1}^G) + \gamma \max_{a' \in \mathcal{A}} q_\phi(s_{t+1}, a') - q_\phi(s_t, a_t) \right), \quad (4.2)$$

where the discounted term of the target becomes 0 when either the next label satisfies ϕ (i.e., $\mathcal{L}_{t+1} \models \phi$) or the history is terminal (i.e., s_{t+1}^T is true). Similarly to r_ϕ , the latter case assumes histories are terminal (i.e., goal or dead-end) depending on the last state only.³ Toro Icarte et al. (2022) propose to model option policies considering the RM state, hindering the policy reusability across options in the RM; nonetheless, their method does not need to make assumptions on the histories. We refer the reader to Chapter 9 for an extended comparison.

Intra-option learning is easily applicable to update an option policy $\pi_{\phi'}$ while another policy π_ϕ is being followed; specifically, given an option policy π_ϕ , an experience tuple $\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle$ generated by this policy is used to update the value of $\langle s_t, a_t \rangle$ of another formula ϕ' through Equation 4.2.

In the function approximation case, the action-value function of each formula ϕ in the RM is approximated through a DQN with parameters θ_ϕ and a target DQN with parameters θ_ϕ^- . The experiences produced by the induced policies are all pushed to a shared replay buffer \mathcal{D} , hence straightforwardly performing intra-option learning. The loss function is constructed analogously to the Q-learning update in Equation 4.2:

$$\mathbb{E}_{\langle s_t, a_t, s_{t+1}, \mathcal{L}_{t+1} \rangle \sim U(\mathcal{D})} \left[\left(r_\phi(\mathcal{L}_{t+1}, s_{t+1}^T, s_{t+1}^G) + \gamma \max_{a' \in \mathcal{A}} q_\phi(s_{t+1}, a'; \theta_\phi^-) - q_\phi(s_t, a_t; \theta_\phi) \right)^2 \right].$$

³In our evaluation (see Chapter 5), the assumption is fulfilled by dead-end histories only (i.e., goal histories depend on the full state-action sequence); however, policy learning is performed successfully across domains. The assumption was present in the original work (Furelos-Blanco et al., 2021) but was not highlighted.

Optimality

In the tabular case, since we optimize each subtask individually (i.e., regardless of the overall task), the best we can hope to achieve is *recursive optimality*. If the action-value functions are approximated, policies may only be approximately optimal.

4.1.2 Learning an Option for each Reward Machine State (QRM)

Q-learning for reward machines (QRM), described in Section 2.1.5, can be framed in terms of options. Instead of learning an option for each edge and a metapolicy for each RM state, QRM learns a single policy over $\mathcal{S} \times \mathcal{U}$ by defining a single option for each RM state. Even though the policy is distributed across the RM states, it is still coupled everywhere since the action-values are bootstrapped from an RM state to the next one; therefore, each option’s policy chooses the action that appears globally best, achieving global optimality in the limit (in the tabular case). In contrast, the previous method (see Section 4.1.1) decouples option policies by making them independent of each other (i.e., each attempts to satisfy a specific formula), which may speed up learning despite not guaranteeing global optimality.

Option Modeling

Given a reward machine $M = \langle \mathcal{U}, \mathcal{P}, \varphi, u^0, u^A, u^R \rangle$, each state $u \in \mathcal{U}$ encapsulates an option $\omega_u = \langle \mathcal{I}_u, \pi_u, \beta_u \rangle$ where:

- The initiation set \mathcal{I}_u and the termination condition β_u are defined as in Section 4.1.1 for non-terminal RM states.
- The policy $\pi_u : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ selects the action that appears globally best at a given state (i.e., the action that leads to the fastest achievement of the goal). In other words, the policy does not attempt to satisfy a specific formula, but the formula that appears to be the best to reach the task’s goal.

Policy Learning

The methods described in Section 2.1.5 are adapted to our setting by expressing the reward-transition function as a mapping from RM state pairs to rewards, and using experiences $\langle \mathbf{s}_t, a_t, \mathbf{s}_{t+1}, \mathcal{L}_{t+1} \rangle$. In particular, Equation 2.1 becomes:

$$q_u(s_t, a_t) = q_u(s_t, a_t) + \alpha \left(r(u, u') + \gamma \max_{a' \in \mathcal{A}} q_{u'}(s_{t+1}, a') - q_u(s_t, a_t) \right), \quad (4.3)$$

where $u' = \delta_M(u, \mathcal{L}_{t+1})$ is the next RM state, and the discounted term becomes 0 when the history is terminal (i.e., s_{t+1}^T is true).

Optimality

In the tabular case, QRM converges to an optimal policy in the limit (Toro Icarte et al., 2018a, Theorem 4.1). Global optimality is possible since action-values are bootstrapped from one RM state to the next. Convergence is not guaranteed when the action-value function is approximated with deep neural networks.

Reward Shaping

A reward machine does not only reveal a task’s subgoals, it also gives an intuition of how far the agent is from completing the task: the closer the agent is to the accepting state, the closer it is to successfully completing the task. Therefore, we can provide the agent with an extra positive reward signal when it gets closer to the accepting state.

The idea of giving additional rewards to guide the agent’s behavior is known as *reward shaping*. Ng et al. (1999) propose a *potential-based shaping function* that provides the agent with additional reward while guaranteeing that optimal policies remain unchanged:

$$f(s_t, a_t, s_{t+1}) = \gamma \Phi(s_{t+1}) - \Phi(s_t),$$

where γ is the MDP’s discount factor and $\Phi : \mathcal{S} \rightarrow \mathbb{R}$ is a real-valued function. The RM structure can be exploited by defining $f : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ in terms of the RM states instead (Camacho et al., 2019; Furelos-Blanco et al., 2020):

$$f(u, u') = \gamma \Phi(u') - \Phi(u),$$

where $\Phi : \mathcal{U} \rightarrow \mathbb{R}$. Consequently, Equation 4.3 is rewritten as:

$$q_u(s_t, a_t) = q_u(s_t, a_t) + \alpha \left(r(u, u') + f(u, u') + \gamma \max_{a' \in \mathcal{A}} q_{u'}(s_{t+1}, a') - q_u(s_t, a_t) \right).$$

Since we want the value of $f(u, u')$ to be positive when the agent gets closer to the accepting state u^A , we define Φ as

$$\Phi(u) = |\mathcal{U}| - d(u, u^A),$$

where $|\mathcal{U}|$ is the number of RM states, and $d(u, u^A)$ is a measure of the distance between u and u^A . If u^A is unreachable from u , then $d(u, u^A) = \infty$.⁴ Note that $|\mathcal{U}|$ acts as an upper bound of the maximum length (i.e., number of directed edges) in an acyclic path between u and u^A . We consider two distance measures between states: the length of the shortest path (d_{\min}) and the length of the longest acyclic path (d_{\max}).

Example 4.1.1. Figure 4.1 shows the shaping rewards generated by the reward shaping function using d_{\min} (a) and d_{\max} (b) with $\gamma = 0.99$ in the COFFEE task’s RM illustrated in Figure 3.2. The numbers inside the states are the values returned by Φ , whereas the numbers on the edges are those returned by f . Note that (i) the number of RM states is $|\mathcal{U}| = 4$ and (ii) the only difference in the output of Φ occurs in the initial state.

4.2 Learning Reward Machines from Traces

In this section, we formalize the task of learning a reward machine from traces (Section 4.2.1) and introduce a method for solving this task using ILASP (Section 4.2.2).

4.2.1 The Reward Machine Learning Task

We here formalize the task of learning a reward machine from traces.

⁴In practice, we use a sufficiently big number to represent ∞ (e.g., 1×10^6).

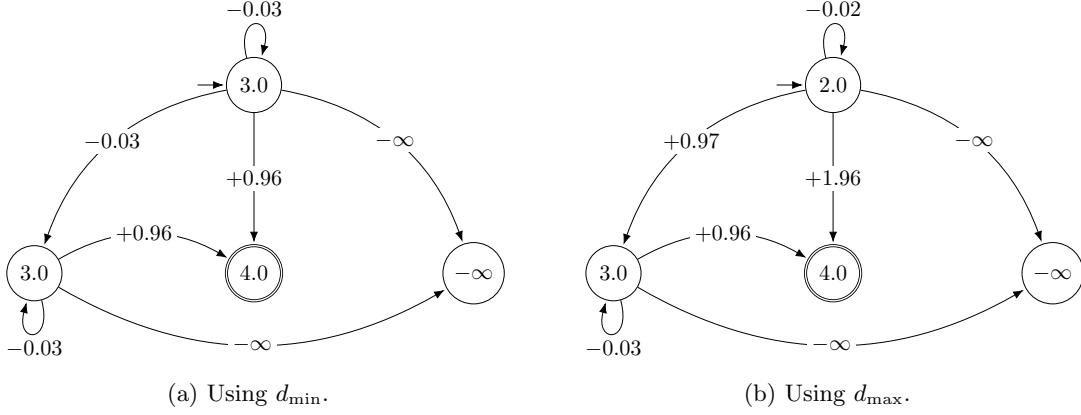


Figure 4.1: Shaping rewards produced by two distance metrics with $\gamma = 0.99$. States and edges are labeled with the output from Φ and f , respectively.

Definition 4.2.1 (RM learning task). *An RM learning task is a tuple $T_M = \langle \mathcal{U}, \mathcal{P}, u^0, u^A, u^R, \Lambda, \kappa \rangle$, where*

- $\mathcal{U} \supseteq \{u^0, u^A, u^R\}$ *is a set of RM states, where u^0 is the initial state, u^A is the accepting state and u^R is the rejecting state;*
- \mathcal{P} *is a set of propositions;*
- $\Lambda = \Lambda^G \cup \Lambda^D \cup \Lambda^I$ *is a set of traces; and*
- κ *is the maximum number of directed edges from a state $u \in \mathcal{U}$ to another state $u' \in \mathcal{U} \setminus \{u\}$.*

An RM M is a solution of T_M if and only if it is valid with respect to all the traces in Λ ; that is, if and only if it accepts all goal traces in Λ^G , rejects all dead-end traces in Λ^D , and does not accept nor reject any incomplete trace in Λ^I .

Note that (i) κ can be seen as the maximum number of disjuncts that a DNF formula $\varphi(u, u')$ between two states u and u' can have, and (ii) \mathcal{U} is the set of states of the learned RM. For simplicity, the RM learning task T_M formalization always includes u^A and u^R in the set of RM states \mathcal{U} . However, in practice, u^A is not included in \mathcal{U} when the set of goal traces Λ^G is empty; likewise, u^R is not included in \mathcal{U} when the set of dead-end traces Λ^D is empty. Removing these states when unnecessary eases RM learning since the hypothesis space shrinks.

4.2.2 Solving the Reward Machine Learning Task with ILASP

Given an RM learning task T_M , we map it into an ILASP learning task $\mathbb{A}(T_M) = \langle \mathcal{B}, \mathcal{S}_{\mathfrak{M}}, \langle \mathcal{E}^+, \emptyset \rangle \rangle$ and use the ILASP system to find a minimal inductive solution $\mathbb{A}_{\varphi}(M) \subseteq \mathcal{S}_{\mathfrak{M}}$ that covers the examples. We do not use *negative examples* ($\mathcal{E}^- = \emptyset$). We define the components of $\mathbb{A}(T_M)$ and prove its correctness in the following paragraphs.

Background Knowledge

The background knowledge $\mathcal{B} = \mathcal{B}_{\mathcal{U}} \cup \mathcal{R}$ is a set of rules that describe the general behavior of any RM. The set of rules $\mathcal{B}_{\mathcal{U}}$ consists of `state`(u) facts for each RM state $u \in \mathcal{U}$, while \mathcal{R} is the set of general

rules that defines how RMs process traces (see the definition in Section 3.3). Describing known concepts through the background knowledge is useful to avoid learning everything from scratch. In this case, we only need to learn the RM's logical transition function, not the general definitions of how RMs work (i.e., the rules in \mathcal{R}).

Hypothesis Space

The hypothesis space $\mathcal{S}_{\mathfrak{M}}$ contains all **ed** and $\bar{\varphi}$ rules that characterize a transition from a non-terminal state $u \in \mathcal{U} \setminus \{u^A, u^R\}$ to a different state $u' \in \mathcal{U} \setminus \{u\}$ using edge $i \in \{1, \dots, \kappa\}$. Formally, it is defined as

$$\mathcal{S}_{\mathfrak{M}} = \left\{ \begin{array}{l} \text{ed}(u, u', i). \\ \bar{\varphi}(u, u', i, T) : \text{-prop}(p, T), \text{step}(T). \\ \bar{\varphi}(u, u', i, T) : \text{-not prop}(p, T), \text{step}(T). \end{array} \middle| \begin{array}{l} u \in \mathcal{U} \setminus \{u^A, u^R\}, \\ u' \in \mathcal{U} \setminus \{u\}, \\ i \in \{1, \dots, \kappa\}, p \in \mathcal{P} \end{array} \right\}.$$

Loop transitions are not included since they are unsatisfiable formulas (see Section 3.2). Note that it is possible to learn unlabeled transitions, which are taken unconditionally (that is, regardless of the current label). For example, for a transition from u to u' using edge i , an inductive solution may only include $\text{ed}(u, u', i)$ and not $\bar{\varphi}(u, u', i, T)$.

As mentioned before, the learner induces the negation $\bar{\varphi}$ of a logical transition function φ .⁵ A different hypothesis space where the learned rules characterize φ directly could have been defined. However, this requires guessing the maximum number of literals that label a transition between two states.⁶ Therefore, we represent RMs using $\bar{\varphi}$ and instead of imposing a maximum size for the conjunctive formulas, we impose a limit (κ) on the number of edges from one state to another.

It is important to realize that the learned hypothesis is denoted by $\mathbb{A}_{\varphi}(M)$ and not $\mathbb{A}(M)$ (see Definition 3.3.2). The set of RM states is given in the background knowledge \mathcal{B} , and the hypothesis space $\mathcal{S}_{\mathfrak{M}}$ only contains transition rules. Hence, the hypothesis is a smallest subset of transition rules that covers all the examples. Since the set of RM states is provided through the background knowledge, a minimal RM (i.e., an RM with the minimum number of states) is only guaranteed to be learned when the set of RM states is the minimal one. The mechanism that interleaves reinforcement learning and RM learning described in Section 4.3 ensures that the learned RM is minimal for a specific value of κ .

Example Sets

Given a set of traces $\Lambda = \Lambda^G \cup \Lambda^D \cup \Lambda^I$, the set of *positive examples* is defined as

$$\mathcal{E}^+ = \{\langle e^*, \mathbb{A}(\lambda) \rangle \mid * \in \{G, D, I\}, \lambda \in \Lambda^*\},$$

where

$$\bullet e^G = \langle \{\text{accept}\}, \{\text{reject}\} \rangle,$$

⁵Remember that the set of rules \mathcal{R} introduced in Section 3.3.2 defines φ in terms of $\bar{\varphi}$.

⁶This is because ILASP has the maximum length of learnable rules as a parameter. This is a problem to enforce determinism (see Section 3.3.4) since we do not know how many literals are going to be needed to make two formulas mutually exclusive. Besides, allowing an arbitrarily large number of literals to overcome the problem massively increases the hypothesis space size.

- $e^D = \langle \{\text{reject}\}, \{\text{accept}\} \rangle$, and
- $e^I = \langle \{\}, \{\text{accept}, \text{reject}\} \rangle$

are the partial interpretations for goal, dead-end and incomplete traces. The **accept** and **reject** atoms express whether a trace is accepted or rejected by the RM; hence, goal traces must only be accepted, dead-end traces must only be rejected, and incomplete traces cannot be accepted or rejected. The context of each example is the set of ASP facts $\mathbb{A}(\lambda)$ that represents the corresponding trace (see Definition 3.3.1).

Correctness of the Learning Task

The following theorem captures the correctness of the ILASP learning task.

Theorem 4.2.1. *Given an RM learning task $T_M = \langle \mathcal{U}, \mathcal{P}, u^0, u^A, u^R, \Lambda, \kappa \rangle$, a reward machine M is a solution of T_M if and only if $\mathbb{A}_\varphi(M)$ is an inductive solution of $\mathbb{A}(T_M) = \langle \mathcal{B}, \mathcal{S}_M, \langle \mathcal{E}^+, \emptyset \rangle \rangle$.*

Proof. Assume M is a solution of T_M .

\iff M is valid with respect to all traces in Λ (i.e., M accepts all traces in Λ^G , rejects all traces in Λ^D and does not accept nor reject any trace in Λ^I).

\iff By Proposition 3.3.1, for each trace $\lambda^* \in \Lambda^*$ where $*$ $\in \{G, D, I\}$, $\mathbb{A}(M) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$ has a unique answer set A and (i) **accept** $\in A$ if and only if $*$ $= G$, and (ii) **reject** $\in A$ if and only if $*$ $= D$.

\iff For each example $e \in \mathcal{E}^+$, $\mathcal{R} \cup \mathbb{A}(M)$ accepts e .

\iff For each example $e \in \mathcal{E}^+$, $\mathcal{B} \cup \mathbb{A}_\varphi(M)$ accepts e . The two programs are identical:

$$\begin{aligned} \mathcal{R} \cup \mathbb{A}(M) &= \mathcal{R} \cup \mathbb{A}_\mathcal{U}(M) \cup \mathbb{A}_\varphi(M) \\ &= \mathcal{R} \cup \mathcal{B}_\mathcal{U} \cup \mathbb{A}_\varphi(M) \\ &= \mathcal{B} \cup \mathbb{A}_\varphi(M). \end{aligned}$$

\iff $\mathbb{A}_\varphi(M)$ is an inductive solution of $\mathbb{A}(T_M)$. □

Enforcement of Structural Properties

The formalization of the RM learning task potentially results in the induction of a non-deterministic RM; in addition, several symmetric solutions might be considered during the search. The learned ASP representation of the RM can be mapped into an equivalent factual representation to enforce determinism (Section 3.3.4) and a canonical indexing of states and edges (Section 3.4). In other words, those RMs that violate the expected structural properties are discarded as solutions to the RM learning task; unfortunately, performing the mapping for every produced candidate solution followed by the verification is prohibitively expensive. However, ILASP enables the mapping to be included in the learning task through meta-program injection (Law et al., 2018), enabling the learner to verify the properties during the search; consequently, the search space is shrunk and the learning of the RMs is performed faster.

We emphasize that the assumptions on the applicability of the proposed symmetry breaking mechanism hold during learning. As described in Section 3.4.3, Assumption 3.4.2 is enforced through the constraints outlined there. Assumption 3.4.3 holds since (i) the learned RMs are deterministic,

guaranteeing that edges from a state to two different states are always different, and (ii) ILASP induces a minimal hypothesis (i.e., a minimal set of transition rules), thus the learned RMs will never have two equal edges from one state to another.

4.3 Interleaved Learning

In this section, we describe the ISA (Induction of Subgoal Automata for Reinforcement Learning) algorithm,⁷ which interleaves reinforcement learning and the learning of the RMs. Given a labeled MDP $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma, \mathcal{P}, l, \tau \rangle$ and a maximum number of edges κ between two states, ISA aims to learn a reward machine $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$ from the experience of a reinforcement learning agent. In the limit, the learned machine M has the fewest possible states given κ and is valid with respect to all the traces observed by the agent.

4.3.1 Algorithm

Algorithm 1 contains the pseudocode describing how RL and RM learning are interleaved. The pseudocode applies to both methods described in Section 4.1 and consists of two functions related to RM learning:

- The `ISCOUNTEREXAMPLE` function (lines 26–29) checks whether the current RM state u is consistent with $\mathbf{s} = \langle s, s^T, s^G \rangle$. It returns true in the following cases:
 - the history is a goal and u is not the accepting state ($s^T = \top \wedge s^G = \top \wedge u \neq u^A$), or
 - the history is a dead-end and u is not the rejecting state ($s^T = \top \wedge s^G = \perp \wedge u \neq u^R$), or
 - the history is not terminal and u is either the accepting or rejecting state ($s^T = \perp \wedge u \in \{u^A, u^R\}$).
- The `ONCOUNTEREXAMPLEFOUND` function (lines 30–37) determines what to do when a trace λ is not correctly recognized by the current RM:
 - (a) Add λ to the corresponding set of traces (line 31):
 - to the set of goal traces Λ^G if the history is a goal ($s^T = \top \wedge s^G = \top$), or
 - to the set of dead-end traces Λ^D if the history is a dead-end ($s^T = \top \wedge s^G = \perp$), or
 - to the set of incomplete traces Λ^I if the history is non-terminal ($s^T = \perp$).
 - (b) Run the RM learner (lines 32–36). If the RM learning task is unsatisfiable, it means that the hypothesis space does not include the RM we are looking for. Therefore, we add a new state to \mathcal{U} .⁸ We adopt this iterative deepening strategy to find a *minimal* RM (i.e., with the fewest states) such that there at most κ edges from one state to another.
 - (c) When a new RM is learned, the action-value functions are reset (line 37). We later explain what resetting these functions means for the two RL algorithms we use.

We now describe the main function of the algorithm:

⁷Subgoal automata (Furelos-Blanco et al., 2020, 2021) are finite-state machines similar to the reward machines used in this thesis but without a reward-transition function; thus, for historical reasons, we have maintained the name of the algorithm.

⁸Non-special states (i.e., not u^0, u^A or u^R) are indexed from 1 upwards (u^1, u^2, \dots).

Algorithm 1 ISA Algorithm

Input: A labeled MDP environment ENV, an initial state u^0 , an accepting state u^A , a rejecting state u^R , a set of propositions \mathcal{P} , and maximum number of edges between two states (κ).

```

1:  $\mathcal{U} \leftarrow \{u^0, u^A, u^R\}$ 
2:  $M \leftarrow \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, u^A, u^R \rangle$ 
3:  $\Lambda \leftarrow \{\}$  ▷ Set of counterexamples
4: INITVALUEFUNCTIONS( $M$ )
5: for  $l = 0$  to num_episodes do
6:    $s, \mathcal{L} \leftarrow \text{ENV.INIT}()$ 
7:    $u \leftarrow \delta_M(u^0, \mathcal{L})$ 
8:    $\lambda \leftarrow \langle \mathcal{L} \rangle$  ▷ Initialize trace
9:   if ISCOUNTEREXAMPLE( $s, u$ ) then
10:     ONCOUNTEREXAMPLEFOUND( $\lambda$ )
11:      $u \leftarrow \delta_M(u^0, \mathcal{L})$ 
12:    $t \leftarrow 0$ 
13:   while  $t < \text{max\_episode\_length} \wedge s^T = \perp$  do ▷ Run episode
14:      $a \leftarrow \text{SELECTACTION}(s, u)$ 
15:      $s', \mathcal{L}' \leftarrow \text{ENV.STEP}(a)$ 
16:      $u' \leftarrow \delta_M(u, \mathcal{L}')$ 
17:      $\lambda \leftarrow \lambda \oplus \langle \mathcal{L}' \rangle$ 
18:     if ISCOUNTEREXAMPLE( $s', u'$ ) then
19:       ONCOUNTEREXAMPLEFOUND( $\lambda$ )
20:       break
21:     else
22:       UPDATEVALUEFUNCTIONS( $s, a, s', \mathcal{L}'$ )
23:      $s \leftarrow s'$ 
24:      $u \leftarrow u'$ 
25:      $t \leftarrow t + 1$ 
26: function ISCOUNTEREXAMPLE( $s, u$ )
27:   return  $(s^T = \top \wedge s^G = \top \wedge u \neq u^A) \vee$ 
28:      $(s^T = \top \wedge s^G = \perp \wedge u \neq u^R) \vee$ 
29:      $(s^T = \perp \wedge u \in \{u^A, u^R\})$ 
30: function ONCOUNTEREXAMPLEFOUND( $\lambda$ )
31:    $\Lambda \leftarrow \Lambda \cup \{\lambda\}$ 
32:   is_unsat  $\leftarrow \top$ 
33:   while is_unsat do
34:      $M, \text{is\_unsat} \leftarrow \text{LEARNREWARDMACHINE}(\mathcal{U}, u^0, u^A, u^R, \Lambda, \kappa)$ 
35:     if is_unsat then
36:        $\mathcal{U} \leftarrow \mathcal{U} \cup \{u^{|\mathcal{U}|-2}\}$ 
37:   RESETVALUEFUNCTIONS( $M$ )

```

1. Initially, the set of states is formed by the initial state u^0 , the accepting state u^A and the rejecting state u^R (line 1). The RM is initialized such that it does not accept nor reject anything; that is, there are no edges between the states in \mathcal{U} (line 2). The set of counterexample traces and the value functions are also initialized (lines 3–4).
2. When an episode starts, the current RM state u is u^0 . One transition is then applied using the initially observed label \mathcal{L} (lines 6–7). For instance, if the agent initially observes $\{\text{☕}\}$ when performing the OFFICEWORLD’s COFFEE task, then u must be the state where the agent has already observed ☕ (see Figure 3.2). The episode trace λ is initialized with the initial label \mathcal{L} (line 8). If a counterexample is detected at the beginning of the episode (line 9), a new RM is learned (line 10), the RM state is reset (line 11) and the episode continues.
3. At each episode’s step, the agent selects an action a in state s (line 14) and applies it in the environment (line 15). Based on the new label \mathcal{L}' , we get the next state u' (line 16) and update the trace λ (line 17). If a counterexample trace λ is found (line 18), a new RM is learned (line 19) and the episode ends (line 20). Otherwise, the value functions are updated (line 22), and the episode continues.

4.3.2 Properties

Theorem 4.3.1 shows that if the target RM is in the hypothesis space, there will only be a *finite number of learning steps* in the algorithm before it converges to such RM (or an equivalent one).

Theorem 4.3.1. *Given a target RM M_* , there is no infinite sequence ρ of RM-counterexample pairs $\langle M_i, e_i \rangle$ such that $\forall i$: (1) M_i covers all examples e_1, \dots, e_{i-1} , (2) M_i does not cover e_i , and (3) M_i is in the finite hypothesis space $\mathcal{S}_{\mathcal{M}}$.*

Proof. By contradiction. Assume that ρ is infinite. Given that $\mathcal{S}_{\mathcal{M}}$ is finite, the number of possible RMs is finite. Hence, some RM M must appear in ρ at least twice, say as $M_i = M_j, i < j$. By definition, M_i does not cover e_i and M_j covers e_i . This is a contradiction. \square

The iterative deepening strategy on the number of states, as mentioned in Section 4.3.1, ensures the learned RMs are *minimal* for a specific value of κ .

4.3.3 Implementation

In the following paragraphs, we describe important aspects of the algorithm’s implementation. First, we describe how the action-value functions are managed for the two algorithms we consider. Second, we introduce two optimizations to make RM learning more efficient.

Management of the Action-Value Functions

A critical aspect of the algorithm is how action-value functions are initialized, updated, and reset when a new RM is learned. In the tabular case, all state-action (and state-option in HRL) values are *initialized* to 0. The action-value functions are *updated* using the rules of the respective algorithms (see Sections 4.1.1–4.1.2). For generality, Algorithm 1 omits the metapolicies update from HRL, which occurs when the selected option terminates. Since an option terminates when the episode

ends or the RM has changed, the update is performed at the end of each step (between lines 22 and 23) if necessary.

Transferring policies (or, more precisely, value functions) from earlier RMs is beneficial when a new RM is learned; however, there are situations where it is not feasible. On the one hand, the HRL metapolicies and the QRM policies are not transferred since they depend on the global RM structure (i.e., they respectively select options and actions toward reaching the accepting state in the fewest steps possible), which changes throughout learning. The corresponding value functions are thus *reset* every time a new RM is learned; however, the reward shaping mechanism introduced for QRM and the independently trained options for HRL can alleviate this problem. On the other hand, in the case of HRL, the action-value functions for all the options used throughout learning are permanently stored and *reused* when the corresponding formulas appear. The management of these functions involves two decisions regarding:

1. **The updating regime.** There are two alternatives: (i) update all the stored functions, and (ii) update only the functions for the current RM formulas. While (i) is more costly, it ensures that all value functions are constantly updated and immediately reusable in the future. Experimentally, we use (i) for the tabular case since it does not incur a significant computational cost; in contrast, we employ (ii) in the function approximation case since updating each function is slower than in the tabular setting.
2. **The reuse for new similar formulas.** Given that value functions are linked to propositional formulas, the value function for a newly observed formula can be initialized from one linked to a similar formula. The number of matching positive literals determines the similarity between two formulas and, in case of a tie, the formula whose function has been updated the most is chosen. Experimentally, the number of matching positive literals works better than the total number of matching literals. We hypothesize that considering negative literals is detrimental since they often emerge from the determinism constraints; that is, they do not represent critical subgoal information, unlike positive literals.

Optimizations

To make the RM learning phase more efficient, we employ the following optimizations in practice:

1. The agent does not learn an RM until a goal trace is observed. Experimentally, when dead-end histories are frequently observed, the learner constantly finds counterexamples that refine paths to the rejecting state. Starting to learn RMs after observing a goal trace is (experimentally) more efficient.
2. The rejecting state u^R is not included in the set of states \mathcal{U} if the set of dead-end traces is empty, as described in Section 4.2.1; hence, the hypothesis space does not contain unnecessary rules that may slow down learning.

4.4 Summary

In this chapter, we presented ISA, an algorithm that interleaves the exploitation and learning of a minimal RM for a given task. We outline two exploitation methods formalized using the options

framework: QRM, an existing method that learns policies at a single timescale, and HRL, which learns policies at two timescales. To provide QRM with a denser reward signal, we proposed a reward shaping method based on the structure of the RMs. The learning of reward machines is accomplished using ILASP, an inductive logic programming system. ILASP learns RMs represented using ASP and leverages a symmetry breaking method that prunes equivalent solutions from the search space to improve learning performance. Both the ASP representation of the RMs and the symmetry breaking method were introduced in the previous chapter.

Chapter 5

Evaluation of Reward Machines

In this chapter, we evaluate the effectiveness of ISA in different domains. Our analysis assesses how the behavior of the RL agent and the task being learned affect RM learning and vice versa. First, we describe the general aspects of our evaluation methodology (Section 5.1). Second, we make a thorough analysis of the performance of our approach using several domains (Sections 5.2–5.4). We conclude by summarizing our findings across experiments (Section 5.5). The code is available at <https://github.com/ertsiger/induction-subgoal-automata-rl>.

5.1 Experimental Setup

In this section, we describe our experimental setup where the learned RMs must generalize to different instances (e.g., grid layouts) of a given task (Section 5.1.1). We also introduce some restrictions on the learned RMs (Section 5.1.2) and a nomenclature for the RL algorithms we evaluate (Section 5.1.3). Finally, we explain how the results are reported in the following sections (Section 5.1.4).

5.1.1 Generalization

We consider the problem of learning an RM given a set of *instances*, each a labeled MDP, for a given task (e.g., COFFEE). Instances differ in the labeling function l and the termination function τ ; nonetheless, even though τ is different, traces are commonly categorized across instances. For example, each instance of OFFICEWORLD has a different grid layout (hence, a different l and a different τ for a given task), but any trace where o is observed after ☕ is considered a goal trace for the COFFEE task regardless of the instance.

The *purpose* is to learn RMs that generalize across instances. For example, the coffee (☕) and the office (o) may be in the same location in OFFICEWORLD; therefore, as shown in Figure 3.2, the learned RM should cover both when ☕ and o are together and when they are not. In general, since a minimal RM is learned from positive examples only (see discussion in Chapter 10), using diverse sources of traces helps avoid overgeneralization. Figure 5.1 displays a grid that produces the shown RM for COFFEE if used alone. Since the agent (ⓧ) cannot see any trace that reaches o without observing ☕ , it does not learn that ☕ is needed to achieve the goal.

In the *tabular* case, a general RM is learned by extending Algorithm 1 such that a full episode is iteratively run for each instance until a maximum number of episodes per instance is reached. The

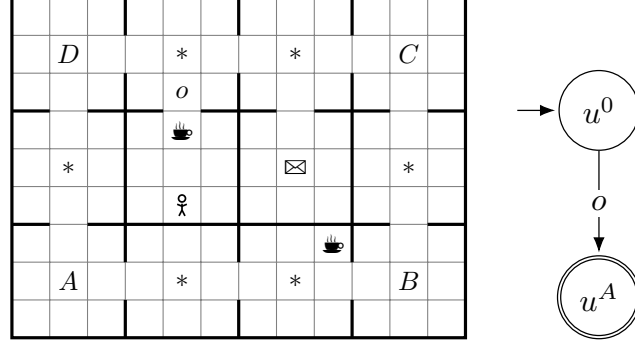


Figure 5.1: Example of an OFFICEWORLD grid whose traces lead to an overgeneral RM for COFFEE.

reason for extending the algorithm is that different policies are learned for each instance. In this case, the learned policies differ for each instance despite sharing the RM. In the *function approximation* case, the algorithm is directly applied since the instance changes every time the environment is reset, and policies generalize to the different instances. A *maximum episode length* N is enforced to guarantee all episodes terminate in a reasonable amount of time.

To reduce our bias on the RM learning process, the instance sets are randomly generated instead of handcrafted; thus, (i) there is no guarantee that certain labels are observable, and (ii) the difficulty of each instance is not fully controlled. Given these two issues, we evaluate how different sets change RL and RM learning (see Section 5.2.5). In the case of (ii), RM learning may be started from traces observed in the easier instances, and the (intermediate) learned RMs may be automatically exploited in the more complex instances.

5.1.2 Restrictions

The following restrictions can be imposed on the factual representation of the learned RM, the traces, and the proposition set to speed up the learning of the RMs. We describe the specific motivation behind them and their ASP implementation where applicable.

Avoid Learning Purely Negative Formulas

We assume that the non-occurrence of certain propositions cannot exclusively characterize subgoals; that is, the formula labeling an edge cannot be formed only by negated propositions. The minimal RMs for the tasks considered here comply with this assumption.

The following constraint encodes the assumption by enforcing a proposition to occur positively whenever a proposition appears negatively in a given edge:

$$:- \text{neg}(X, Y, E, -), \text{not pos}(X, Y, E, -).$$

Acyclicity

There are tasks whose minimal RMs do not contain cycles (i.e., a previously visited state cannot be revisited). The reward machines for the tasks we consider (including the OFFICEWORLD tasks

introduced so far) belong to this class; thus, the search space can be made smaller by ruling out solutions containing cycles.

The following set of rules enforces acyclicity. The $\text{path}(u, u')$ atoms indicate there is a directed path (i.e., a sequence of directed edges) from state u to state u' . The first rule states that there is a path from state X to state Y if there is an edge from X to Y . The second rule indicates that there is a path from X to Y if there is an edge from X to an intermediate state Z from which there is a path to Y . Finally, the third rule rules out the answer sets with a path from X to Y and vice versa.

$$\left\{ \begin{array}{l} \text{path}(X, Y) :- \text{ed}(X, Y, -). \\ \text{path}(X, Y) :- \text{ed}(X, Z, -), \text{path}(Z, Y). \\ :- \text{path}(X, Y), \text{path}(Y, X). \end{array} \right\}$$

Trace Compression

The counterexample traces employed by ISA depend on the agent behavior. While the agent has not managed to reach the goal, its behavior is random; consequently, the counterexamples provided to the RM learner can be long and include many irrelevant labels to the task at hand. As shown later, these factors negatively impact the time required to learn an RM.

We define a subtype of label trace called *compressed label trace* (or compressed trace), which is built from a label trace based on the following assumptions:

1. Empty labels are irrelevant.
2. Observing the same label twice or more in a row is equivalent to observing it once.

Definition 5.1.1 (Compressed label trace). *A compressed label trace $\hat{\lambda} = \langle \hat{\mathcal{L}}_0, \dots, \hat{\mathcal{L}}_m \rangle$ is the result of removing the empty labels and, thereafter, removing contiguous equal labels from a label trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$.*

Example 5.1.1. *The goal trace $\lambda = \langle \{\}, \{\text{☛}\}, \{\text{☛}\}, \{\}, \{\text{☛}\}, \{\}, \{\}, \{o\} \rangle$ for COFFEE is compressed into $\hat{\lambda} = \langle \{\text{☛}\}, \{o\} \rangle$.*

As shown in Section 5.2.5, compressed traces speed up RM learning; however, their applicability is limited to tasks where the assumptions above hold. For instance, these traces cannot be used to learn RMs for tasks where every label is essential, such as “observe ☛ twice in a row”; in contrast, they are applicable in the tasks considered in the experiments.

The learning of the RMs is agnostic to the type of trace; thus, no change is required in encoding the learning tasks. Nevertheless, since compressed traces lose the information about the number of performed steps and contain no empty labels, unlabeled transitions are meaningless; therefore, we rule out RMs with unlabeled edges using the following constraint when trace compression is enabled:

$$:- \text{ed}(X, Y, E), \text{not pos}(X, Y, E, -), \text{not neg}(X, Y, E, -).$$

This constraint rules out any inductive solution where an edge from X to Y with index E is not labeled by a positive or a negative literal.

In the case of the RL component, the use of compressed traces changes the RMs’ traversal. Since empty labels and contiguous duplicated labels are ignored, the state-transition function δ_M of a

reward machine M is only queried when the current label is (i) not empty and (ii) different from the previous label. Otherwise, the agent remains in the same RM state.

Restricted Proposition Set

To further simplify the traces, these can be defined only in terms of the propositions relevant to the task at hand. For example, if the task is COFFEE, the proposition set is $\hat{\mathcal{P}} = \{\text{☕}, o, *\}$ instead of $\mathcal{P} = \{\text{☕}, \boxtimes, o, A, B, C, D, *\}$. This simplifies RM learning since the hypothesis space becomes smaller, and ILASP does not have to discern which propositions are relevant.

5.1.3 Reinforcement Learning Algorithms

We use the following nomenclature for the different RL algorithms applied in the experiments:

- HRL: HRL where $r_{success} = 1.0$, $r_{deadend} = 0.0$ and $r_{step} = 0.0$.
- HRL_G: HRL where $r_{success} = 1.0$, $r_{deadend} = -N$ and $r_{step} = -0.01$. Remember that N denotes the maximum episode length.
- QRM: QRM without reward shaping.
- QRM_{min}: QRM with reward shaping based on the length of the shortest path to the accepting state (d_{min}).
- QRM_{max}: QRM with reward shaping based on the length of the longest acyclic path to the accepting state (d_{max}).

5.1.4 Reporting Results

We report results using tables and figures that result from averaging 20 independent runs, each using a different random seed. All experiments were run on 3.40GHz Intel[®] Core[™] i7-6700 processors. The reward machines were learned using ILASP2, with each individual learning task having a 2-hour timeout.

In the *tables*, we report the following RM learning metrics where the learner has not timed out and at least one RM has been learned:

- Total time (in seconds) used to run the RM learner.
- Number of examples used to learn the final RM.
- Length of the examples used to learn the final RM.

Metrics are reported as $\mu \pm \sigma$, where μ is the average, and σ is either the standard error (for the total time and the number of examples) or the standard deviation (for the example length since it is computed across all runs). We mark with an asterisk (*) cases where either no RM has been learned¹ or the learner has timed out between 1 and 10 runs. A dash (–) denotes that the number of these cases is higher than 10.

¹Remember that RMs are started to be learned once a goal trace is observed (see Section 4.3.3).

The *figures* show the average undiscounted return across instances and runs. Each point of the learning curve represents the undiscounted return obtained by the greedy policy in an episode. The greedy policy is evaluated for one episode after every training episode by default. The dotted vertical lines correspond to episodes where an RM was learned. When the learner times out, the reward is set to 0 for the entire interaction.

5.2 Experiments in OFFICEWORLD

The OFFICEWORLD domain (Toro Icarte et al., 2018a), introduced in Example 2.1.2, consists of a 9×12 grid illustrated in Figure 2.3. The proposition set is $\mathcal{P} = \{\text{☿}, \boxtimes, o, A, B, C, D, *\}$.

5.2.1 Instance Generation

The grid contains a proposition of each type except for ☿ and *, which appear 2 and 6 times, respectively. The agent and the propositions are randomly placed in the grid such that:

- The agent cannot be initially placed with decorations * or propositions $A - D$.
- The decorations * do not share a location with any other proposition.
- The decorations * and propositions $A - D$ cannot be placed next to each other (including diagonals) nor in locations that connect two rooms such as $\langle 1, 2 \rangle$ and $\langle 1, 3 \rangle$.
- Propositions $A - D$ and the office o cannot be in the same location.

Note that (i) ☿, \boxtimes , and o can be in the same location, and (ii) ☿ and \boxtimes can share a location with propositions $A - D$.

5.2.2 Tasks

The tasks are those introduced in Example 2.1.2. Their respective RMs are different and incrementally challenging to learn:

- The COFFEE task has 2 subgoals and is represented by a 4-state minimal RM.
- The COFFEEEMAIL task has 3 subgoals and is represented by a 6-state minimal RM.
- The VISITABCD task has 4 subgoals and is represented by a 6-state minimal RM.

In the tasks considered in this chapter, the number of subgoals is the number of directed edges in the longest acyclic path from the initial state to the accepting state in the minimal RM. Generally, the longer the subgoal sequence is, the harder it becomes to achieve the goal. We refer the reader to Appendix A.1 for illustrations of the RMs considered in these experiments.

5.2.3 Hyperparameters

Table 5.1 shows the hyperparameters used throughout these experiments, where α , ϵ and γ are used for both the metapolicies and option policies across all HRL variants.

Table 5.1: Hyperparameters used in the OFFICEWORLD experiments.

Learning rate α	0.1
Exploration rate ϵ	0.1
Discount factor γ	0.99
Number of episodes per instance	10,000
Avoid learning purely negative formulas	✓
Number of instances	50
Maximum episode length N	250
Trace compression	✓
Enforce acyclicity	✓
Number of disjuncts κ	1
Use restricted proposition set	✗

5.2.4 Results

Figure 5.2 shows how the learning curves for interleaved RM learning approaches (ISA-HRL, ISA-QRM) compare to those obtained by approaches exploiting handcrafted RMs (HRL, QRM). We observe the following:

- The algorithms using auxiliary guidance (HRL_G , QRM_{\min} , QRM_{\max}) converge faster than their respective standard versions (HRL and QRM). Using auxiliary reward signals helps explore the state space more effectively, which results in observing counterexamples (and, hence, learning RMs) earlier.
- QRM_{\max} converges faster than QRM_{\min} except in VISITABCD, where the curves match because there is a single path to the accepting state (i.e., the auxiliary rewards are the same). As shown in Example 4.1.1, QRM_{\max} provides a positive reward for any path that enables the agent to approach the accepting state; in contrast, QRM_{\min} only provides a positive signal for the shortest path(s). If the shortest path is not available in a certain grid (e.g., when label $\{\text{☹}, o\}$ is not observable), QRM_{\min} gives a negative reward for choosing the only available path to the accepting state and, as a result, convergence is slower than for QRM_{\max} .
- HRL converges faster than QRM. In the absence of reward shaping, QRM needs to satisfy a formula on an edge to the accepting state to start inducing changes in the action-value functions of the different RM states; in contrast, HRL independently updates the action-value functions of each formula without having to reach the accepting state.
- Approaches involving RM learning perform closely to those exploiting handcrafted RMs. Naturally, their convergence is delayed since an RM is not exploited from the first episode (e.g., a stable RM is found after several episodes in VISITABCD).

Figure 5.3 shows the impact of RM learning on the HRL and QRM curves of the COFFEE task in a single run. While HRL quickly recovers its performance after learning an RM, QRM requires a few episodes. Indeed, remember that an HRL agent forgets the metapolicies but keeps the action-value functions of the option policies unchanged, and reuses the action-value functions of existing formulas for new similar ones; in contrast, QRM forgets everything it learned.

Table 5.2 shows the RM learning metrics for the presented OFFICEWORLD tasks using HRL_G . We observe that:

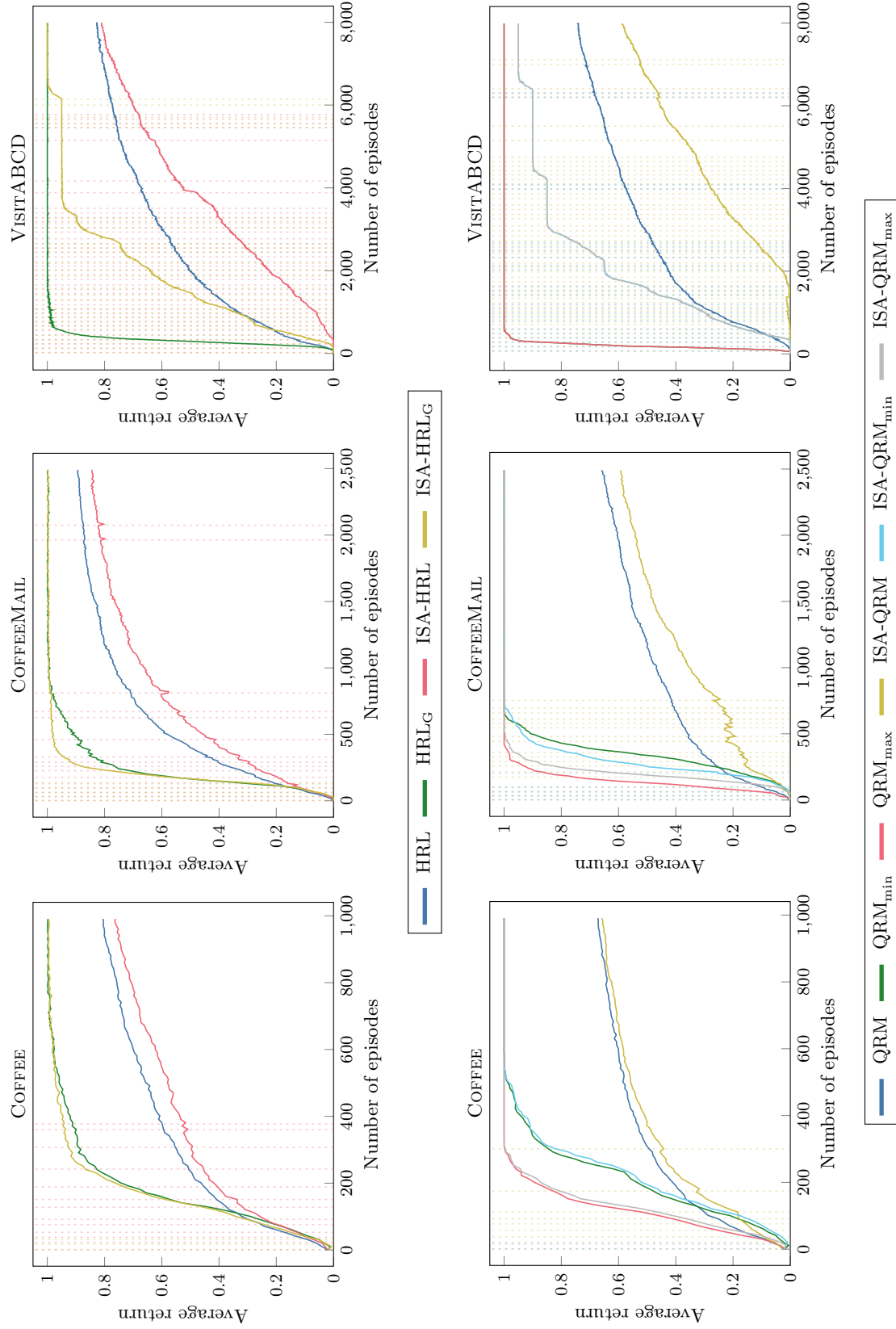


Figure 5.2: Learning curves for different RL algorithms in the OFFICEWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM).

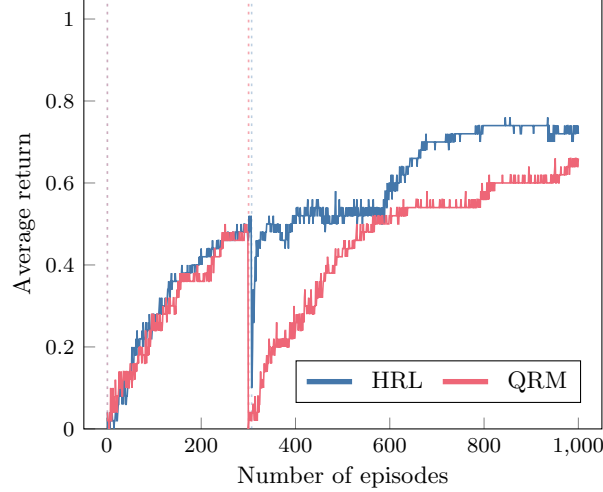


Figure 5.3: Example of the impact that interleaved RM learning has on the learning curves of the COFFEE task. An RM is learned around episode 300. While HRL quickly recovers since it only forgets the metapolicies, QRM needs more episodes because it forgets everything.

- The running time increases with the number of subgoals. Learning an RM for VISITABCD takes (on average) more time than one for COFFEEEMAIL, even though both are characterized by minimal RMs with the same number of states.
- The number of examples increases with the number of subgoals. The number of goal trace examples is approximately the number of paths to the accepting state; for instance, in VISITABCD there is only one such path, so the number of goal trace examples is approximately 1.² In contrast, the propositions that label the RM for COFFEEEMAIL can appear jointly or not; consequently, there are more paths to the accepting state, and the number of goal examples increases. Furthermore, while there is a relationship between the number of goal examples and the number of paths to the accepting state, we do not observe such a relationship between the number of dead-end examples and the number of paths to the rejecting state. The number of dead-end and incomplete examples is higher than that of goal examples; thus, we hypothesize that these examples are mainly used to refine the RM given the set of goal examples.
- The example length increases with the number of subgoals. Intuitively, the more subgoals, the longer the agent needs to interact with the environment to achieve the goal; therefore, the observed counterexamples tend to be longer for tasks with more subgoals.

5.2.5 Ablations

This section analyzes how the RL and RM learning hyperparameters impact ISA’s performance. Table 5.3 shows the hyperparameters used throughout these experiments, dividing those that remain unchanged from those that vary across experiments.

²The number of goal trace examples can be higher than 1 for VISITABCD if the first goal trace example is complex (e.g., longer than needed or with many unnecessary propositions), thus making the subgoals unclear. In such cases, a simpler goal trace might later be found as a counterexample.

Table 5.2: RM learning metrics for the OFFICEWORLD tasks using HRL_G .

	Time (s.)	# Examples				Example Length
		All	G	D	I	
COFFEE	0.4 ± 0.0	8.7 ± 0.4	2.4 ± 0.1	3.0 ± 0.1	3.2 ± 0.3	2.8 ± 2.1
COFFEE _{MAIL}	18.9 ± 3.3	29.0 ± 1.5	3.9 ± 0.3	9.3 ± 0.6	15.8 ± 1.0	4.0 ± 2.6
VISIT _{ABCD}	163.2 ± 44.3	54.9 ± 3.8	1.6 ± 0.1	15.2 ± 0.9	38.1 ± 3.1	5.5 ± 3.1

Table 5.3: Hyperparameters used in the OFFICEWORLD ablation experiments. The top of the table lists the hyperparameters that remain unchanged, while the bottom lists those that change across experiments.

Learning rate α	0.1
Exploration rate ϵ	0.1
Discount factor γ	0.99
Number of episodes per instance	10,000
Avoid learning purely negative formulas	✓
RL algorithm	HRL_G
Number of instances	50
Maximum episode length N	250
Trace compression	✓
Enforce acyclicity	✓
Number of disjuncts κ	1
Use restricted proposition set	✗

Instances and Maximum Episode Lengths

We study the performance variability across instance sets, the number of instances within each set, and maximum episode lengths. We consider two 100-instance sets, $\mathbb{I}_1 = \{I_{1,1}, \dots, I_{1,100}\}$ and $\mathbb{I}_2 = \{I_{2,1}, \dots, I_{2,100}\}$. We denote by $\mathbb{I}_j^k = \{I_{j,1}, \dots, I_{j,k}\} \subseteq \mathbb{I}_j$ the subset of the first k instances from the j -th set. These experiments are performed with $k \in \{10, 50, 100\}$ and $N \in \{100, 250, 500\}$.

Given instance set \mathbb{I}_1 , Figure 5.4 shows the learning curves for the different combinations of instance subsets and steps. We make the following observations:

- The lowest maximum episode length ($N = 100$) works fine when the goal is easy to achieve (i.e., there are few subgoals, like in COFFEE). As the number of subgoals grows, N needs to increase to achieve the goal. If N is not high enough, there is a low chance that the agent observes the counterexamples required to find the target RM. Remember that the agent needs to achieve the goal at least once to learn an RM that can be exploited. In the case of VISIT_{ABCD}, we observe barely any convergence when $N = 100$ because no RM is learned. Even when the number of instances is high (100), a goal trace to start RM learning is only found in 9 out of 20 runs.
- Small instance sets are sufficient to learn an RM and policies that achieve the goal in COFFEE; however, in tasks involving more subgoals, convergence is faster for larger sets (50 or 100) than for small ones (10). For example, RMs for VISIT_{ABCD} are rarely learned from 10 instances for any maximum episode length. Increasing the number of instances increases the chance that easier grids are included in the set and, in turn, the chance of observing counterexamples to start learning RMs.

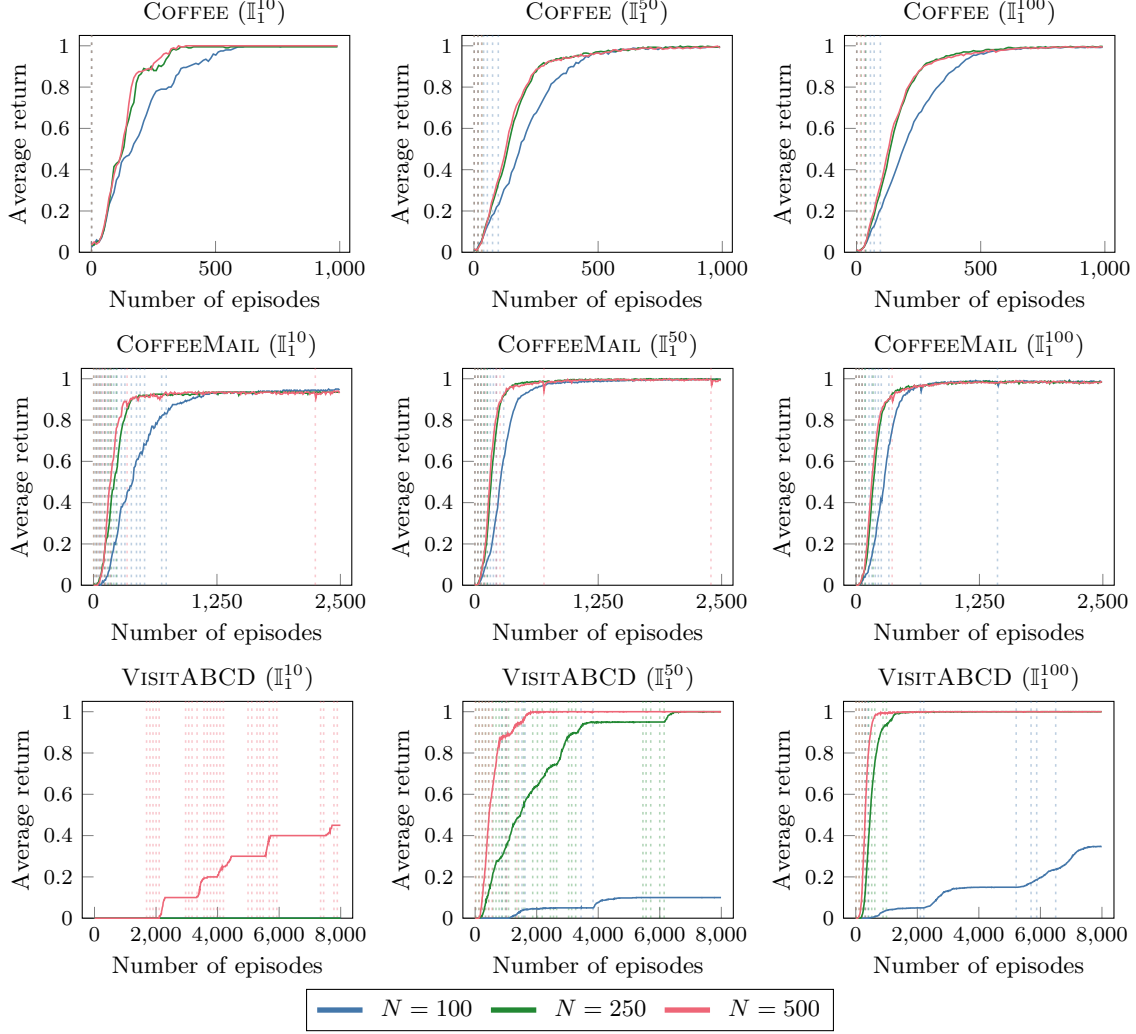


Figure 5.4: Learning curves for different combinations of instance sets (\mathbb{I}_1^{10} , \mathbb{I}_1^{50} , \mathbb{I}_1^{100}) and maximum episode lengths (100, 250, 500).

- Small values of N and fewer instances often cause RM learning to occur throughout the entire interaction, whereas higher values of N and large instance sets concentrate learning early in the interaction. When N is small, the chance of observing a counterexample is lower since the interaction is shorter, which is detrimental in combination with small sets containing instances where the goal is hard to achieve (e.g., if the propositions are sparsely distributed in the grid).

A high value of N seems to be the best choice to ensure that goal traces are observed; however, such choice produces longer traces and makes RM learning more complex since (i) the chance of observing irrelevant propositions to the task at hand increases (i.e., the system has to learn they are unimportant) and (ii) it is harder to figure out the order in which subgoals must be observed. Table 5.4 shows the total RM learning time, while Table 5.5 shows the number of examples needed to learn the last RM in each run. The following is observed from these tables:

- Running times generally increase with the maximum episode length. Table 5.6 contains the example lengths for \mathbb{I}_1^{50} , showing that the longer the episode, the longer the counterexamples.

Table 5.4: Total RM learning time in seconds for different combinations of instance sets (\mathbb{I}_1^{10} , \mathbb{I}_1^{50} , \mathbb{I}_1^{100}) and maximum episode lengths (100, 250, 500).

	\mathbb{I}_1^{10}			\mathbb{I}_1^{50}			\mathbb{I}_1^{100}		
	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$
COFFEE	0.3 ± 0.0	0.3 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	0.5 ± 0.0	0.5 ± 0.0
COFFEEMAIL	5.9 ± 1.5	4.2 ± 1.1	9.4 ± 3.3	7.8 ± 1.5	18.9 ± 3.3	49.1 ± 12.0	9.2 ± 1.2	29.1 ± 9.4	64.3 ± 15.3
VISITABCD	–	–	$2966.4 \pm 1323.9^*$	–	163.2 ± 44.3	311.9 ± 63.4	–	230.7 ± 99.9	230.8 ± 48.2

Table 5.5: Number of examples needed to learn the last RM for different combinations of instance sets (\mathbb{I}_1^{10} , \mathbb{I}_1^{50} , \mathbb{I}_1^{100}) and maximum episode lengths (100, 250, 500).

	\mathbb{I}_1^{10}			\mathbb{I}_1^{50}			\mathbb{I}_1^{100}		
	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$
COFFEE	5.8 ± 0.3	7.0 ± 0.3	7.4 ± 0.3	8.4 ± 0.3	8.7 ± 0.4	10.7 ± 0.5	8.6 ± 0.4	9.6 ± 0.5	9.4 ± 0.4
COFFEEMAIL	24.2 ± 1.5	20.6 ± 1.5	24.4 ± 1.7	24.8 ± 1.6	29.0 ± 1.5	33.0 ± 1.5	29.6 ± 1.2	35.7 ± 1.2	35.9 ± 1.6
VISITABCD	–	–	$86.0 \pm 12.0^*$	–	54.9 ± 3.8	50.6 ± 3.5	–	55.2 ± 6.0	49.9 ± 2.5

- Running times increase as the instance set becomes larger, specifically when changing from \mathbb{I}_1^{10} to \mathbb{I}_1^{50} due to some labels occurring within the latter and not in the former. For instance, label $\{\text{☞}, o\}$ is only observable in \mathbb{I}_1^{50} and \mathbb{I}_1^{100} but not in \mathbb{I}_1^{10} ; hence, time is spent on learning the $\text{☞} \wedge o$ transition for COFFEE and COFFEEMAIL.
- The number of examples is similar across instance sets. There is only a noticeable difference between \mathbb{I}_1^{10} to \mathbb{I}_1^{50} because the latter includes labels that do not happen in the former, as explained before. These differences do not occur for VISITABCD because there is a single path to the accepting state.
- The running time and the number of examples increase with the number of subgoals, as shown in Section 5.2.4. Table 5.6 shows that the example length is longer for the tasks with more subgoals, which affects the time needed to learn the RMs.

We now briefly examine the results for the instance set \mathbb{I}_2 . Figure 5.5 displays the learning curves for different combinations of instance subsets and values of N . Table 5.7 shows the RM learning time, whereas Table 5.8 contains the number of examples needed to learn an RM for different subsets of tasks and maximum episode lengths (N). Table 5.9 shows the example length of each trace type for different values of N . Table 5.10 contains the number of examples needed to learn the last RM in a specific setting. In qualitative terms, the changes we observe with respect to \mathbb{I}_1 occur for VISITABCD. While $N = 100$ was usually insufficient to learn an RM in \mathbb{I}_1^{50} , it is enough in \mathbb{I}_2^{50} . By comparing the VISITABCD curves, it is clear that \mathbb{I}_2^{50} consists of instances requiring less steps than those in \mathbb{I}_1^{50} . Furthermore, in the case of COFFEEMAIL, \mathbb{I}_1^{100} has more types of joint events than \mathbb{I}_2^{100} ; thus, more examples are needed in that case, and the final RM is slightly different. This shows

Table 5.6: Example length of the goal, dead-end and incomplete examples used to learn the last RM in the \mathbb{I}_1^{50} setting.

	$N = 100$			$N = 250$			$N = 500$		
	G	D	I	G	D	I	G	D	I
COFFEE	2.8 ± 1.0	2.1 ± 1.1	1.5 ± 0.9	3.6 ± 1.8	3.0 ± 2.5	1.9 ± 1.4	5.7 ± 4.6	3.6 ± 2.8	2.2 ± 1.4
COFFEEMAIL	4.4 ± 1.4	3.3 ± 2.0	3.1 ± 1.8	5.5 ± 2.7	4.1 ± 2.8	3.6 ± 2.2	7.2 ± 4.3	4.9 ± 3.8	3.5 ± 2.3
VISITABCD	–	–	–	9.2 ± 3.0	5.1 ± 3.0	5.4 ± 3.0	12.4 ± 4.2	6.2 ± 4.0	6.3 ± 4.2

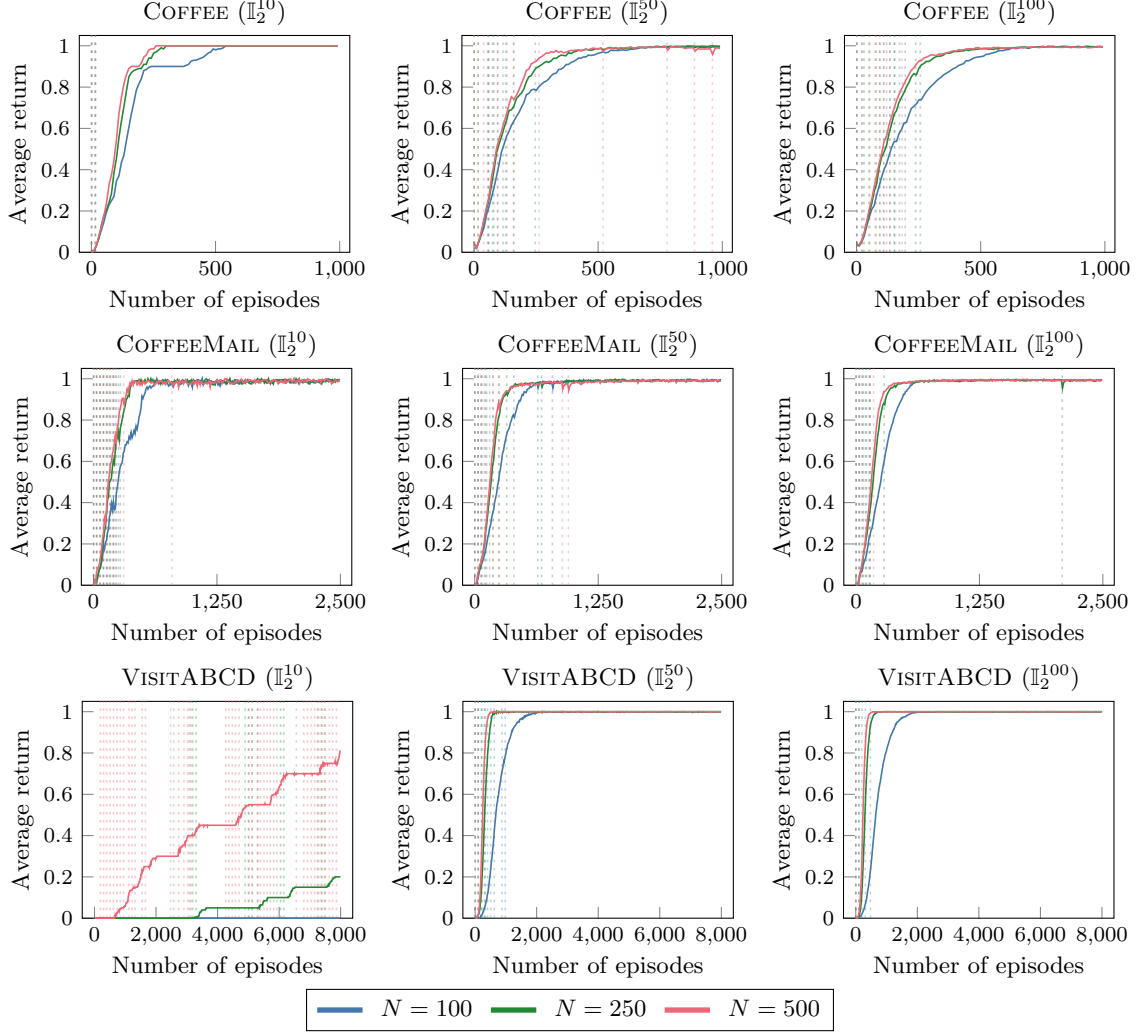


Figure 5.5: Learning curves for different combinations of instance sets (\mathbb{I}_2^{10} , \mathbb{I}_2^{50} , \mathbb{I}_2^{100}) and maximum episode lengths (100, 250, 500).

that the instance set has an impact on the learned RMs.

Trace Compression

Compressed traces are applicable to the OFFICEWORLD tasks since (i) empty labels are meaningless (i.e., the number of steps is unimportant), and (ii) observing a label for several consecutive steps is equivalent to observing it once. Table 5.11 shows the impact of compressed traces on RM learning. The learning curves are omitted since they are similar for both trace types. All runs using uncompressed traces finished on time for COFFEE and COFFEEMAIL, whereas all such runs timed out for VISITABCD; hence, compressed traces enable learning an RM for VISITABCD within the allotted time. Besides, trace compression enables learning an RM several orders of magnitude faster in COFFEEMAIL. Compressed traces are an order of magnitude shorter than uncompressed traces, even in the simplest task (COFFEE).

Table 5.7: Total RM learning time in seconds for different combinations of instance sets (\mathbb{I}_2^{10} , \mathbb{I}_2^{50} , \mathbb{I}_2^{100}) and maximum episode lengths (100, 250, 500).

	\mathbb{I}_2^{10}			\mathbb{I}_2^{50}			\mathbb{I}_2^{100}		
	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$
COFFEE	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0	0.4 ± 0.0
COFFEEEMAIL	2.8 ± 0.4	3.9 ± 1.1	8.1 ± 2.2	7.8 ± 3.1	10.3 ± 1.6	24.5 ± 7.5	6.8 ± 1.4	30.7 ± 14.6	22.2 ± 5.3
VISITABCD	–	–	802.1 ± 297.1	130.1 ± 46.4	290.2 ± 123.1	481.6 ± 115.1	53.2 ± 15.6	269.2 ± 108.0	355.8 ± 68.2

Table 5.8: Number of examples needed to learn the last RM for different combinations of instance sets (\mathbb{I}_2^{10} , \mathbb{I}_2^{50} , \mathbb{I}_2^{100}) and maximum episode lengths (100, 250, 500).

	\mathbb{I}_2^{10}			\mathbb{I}_2^{50}			\mathbb{I}_2^{100}		
	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$	$N = 100$	$N = 250$	$N = 500$
COFFEE	7.3 ± 0.3	7.6 ± 0.4	7.5 ± 0.5	8.2 ± 0.3	8.3 ± 0.3	8.6 ± 0.5	8.3 ± 0.3	8.0 ± 0.3	8.2 ± 0.3
COFFEEEMAIL	17.8 ± 1.1	17.8 ± 0.8	20.6 ± 1.4	24.4 ± 1.5	26.2 ± 1.2	27.2 ± 1.4	24.9 ± 1.2	29.1 ± 1.4	27.4 ± 1.3
VISITABCD	–	–	74.8 ± 8.2	57.7 ± 5.6	53.8 ± 6.2	58.6 ± 3.9	42.5 ± 3.3	51.9 ± 3.3	53.3 ± 3.3

Table 5.9: Example length of the goal, dead-end and incomplete examples used to learn the last RM in the \mathbb{I}_2^{50} setting.

	$N = 100$			$N = 250$			$N = 500$		
	G	D	I	G	D	I	G	D	I
COFFEE	2.8 ± 1.0	2.1 ± 1.1	1.5 ± 0.9	3.6 ± 1.8	3.0 ± 2.5	1.9 ± 1.4	5.7 ± 4.6	3.6 ± 2.8	2.2 ± 1.4
COFFEEEMAIL	3.5 ± 1.4	3.2 ± 1.7	2.7 ± 1.5	4.3 ± 2.8	4.0 ± 2.6	3.4 ± 2.6	5.1 ± 4.2	4.1 ± 3.1	3.5 ± 2.7
VISITABCD	7.1 ± 1.9	4.6 ± 2.3	4.6 ± 1.9	8.9 ± 3.7	5.5 ± 3.6	5.1 ± 2.6	11.2 ± 4.6	7.1 ± 4.8	5.8 ± 3.3

Table 5.10: Number of examples (total, goal, dead-end and incomplete) needed to learn the last RM in the \mathbb{I}_2^{50} and $N = 250$ setting.

	All	G	D	I
COFFEE	8.7 ± 0.4	2.4 ± 0.1	3.0 ± 0.1	3.2 ± 0.3
COFFEEEMAIL	26.2 ± 1.2	5.3 ± 0.3	8.2 ± 0.5	12.6 ± 0.9
VISITABCD	53.8 ± 6.2	1.6 ± 0.1	16.2 ± 1.4	36.0 ± 4.9

Table 5.11: RM learning metrics when traces are compressed (C) or uncompressed (U).

	Time (s.)		# Examples		Example Length	
	C	U	C	U	C	U
COFFEE	0.4 ± 0.0	1.5 ± 0.2	8.7 ± 0.4	11.4 ± 0.4	2.8 ± 2.1	58.1 ± 64.6
COFFEEEMAIL	18.9 ± 3.3	9314.6 ± 1859.7	29.0 ± 1.5	34.1 ± 1.4	4.0 ± 2.6	78.1 ± 65.7
VISITABCD	163.2 ± 44.3	–	54.9 ± 3.8	–	5.5 ± 3.1	–

Table 5.12: RM learning metrics when the proposition set is unrestricted (\mathcal{P}) or restricted to a particular task ($\hat{\mathcal{P}}$).

	Time (s.)		# Examples		Example Length	
	\mathcal{P}	$\hat{\mathcal{P}}$	\mathcal{P}	$\hat{\mathcal{P}}$	\mathcal{P}	$\hat{\mathcal{P}}$
COFFEE	0.4 ± 0.0	0.3 ± 0.0	8.7 ± 0.4	6.4 ± 0.2	2.8 ± 2.1	1.5 ± 0.6
COFFEEMAIL	18.9 ± 3.3	1.5 ± 0.1	29.0 ± 1.5	16.1 ± 0.6	4.0 ± 2.6	2.7 ± 1.4
VISITABCD	163.2 ± 44.3	9.2 ± 1.6	54.9 ± 3.8	30.9 ± 2.3	5.5 ± 3.1	3.7 ± 1.8

Table 5.13: Comparison of different RM learning metrics for the cases where RMs must be acyclic and where RMs can have cycles.

	Time (s.)		# Examples		Example Length	
	Acyclic	Cyclic	Acyclic	Cyclic	Acyclic	Cyclic
COFFEE	0.4 ± 0.0	0.5 ± 0.0	8.7 ± 0.4	9.6 ± 0.4	2.8 ± 2.1	2.7 ± 2.2
COFFEEMAIL	18.9 ± 3.3	774.7 ± 434.4	29.0 ± 1.5	33.8 ± 1.7	4.0 ± 2.6	4.0 ± 2.6
VISITABCD	163.2 ± 44.3	1961.7 ± 1123.8	54.9 ± 3.8	81.0 ± 6.6	5.5 ± 3.1	5.5 ± 3.3
COFFEEDROP	$13.9 \pm 8.6^*$	0.6 ± 0.0	$15.0 \pm 3.1^*$	9.9 ± 0.5	$5.4 \pm 3.5^*$	5.3 ± 3.6
COFFEEMAILDROP	–	312.2 ± 145.9	–	37.8 ± 1.7	–	7.0 ± 5.3

Restricted Proposition Set

Table 5.12 shows how using the restricted proposition set $\hat{\mathcal{P}}$ for each OFFICEWORLD task compares to using the complete set \mathcal{P} . The restricted set $\hat{\mathcal{P}}$ causes a sensible decrease in the RM learning time, especially for the more challenging tasks. Similarly, fewer examples are needed to learn a helpful RM, and the example length is also reduced. Intuitively, using only the propositions that describe the task’s subgoals eases RM learning: the hypothesis space is smaller, and no examples for discarding irrelevant labels are needed. The learning curves are not compared since they are similar.

Cyclicity

Table 5.13 shows how enforcing the RM to be acyclic changes RM learning. The tasks that we have considered so far do not require cycles. Allowing the RMs to have cycles results in a larger search space; consequently, the RM learning time considerably increases, especially for COFFEEMAIL and VISITABCD. While more examples are also needed to rule out the solutions with cycles, their length remains approximately the same.

We introduce two tasks to show that our approach can learn such RMs, COFFEEDROP and COFFEEMAILDROP, where the agent drops the coffee when it steps on a decoration (*); thus, in such cases, the agent must go back to the coffee location. Dead-end histories cannot be observed in these tasks, so the rejecting state is unreachable in the learned RMs. When cycles are allowed, the results for COFFEEDROP and COFFEEMAILDROP resemble those for COFFEE and COFFEEMAIL, respectively. The running time for COFFEEMAILDROP is lower than for COFFEEMAIL since the former has fewer states than the latter (it does not have a rejecting state); besides, the example length is higher since there are no dead-end states to avoid. In the acyclic setting, an RM is only found in 10/20 and 2/20 runs for COFFEEDROP and COFFEEMAILDROP, respectively. In this case, the number of RM states depends on how many times a coffee has been picked and dropped in the example traces. These tasks are clearly not suited to be expressed by acyclic RMs.

Table 5.14: RM learning metrics for different maximum number of edges from one state to another (κ).

	Time (s.)		# Examples		Example Length	
	$\kappa = 1$	$\kappa = 2$	$\kappa = 1$	$\kappa = 2$	$\kappa = 1$	$\kappa = 2$
COFFEE	0.4 ± 0.0	1.0 ± 0.1	8.7 ± 0.4	11.7 ± 0.6	2.8 ± 2.1	3.1 ± 2.4
COFFEEMAIL	18.9 ± 3.3	$2252.7 \pm 623.2^*$	29.0 ± 1.5	$32.5 \pm 1.6^*$	4.0 ± 2.6	$4.0 \pm 2.5^*$
VISITABCD	163.2 ± 44.3	–	54.9 ± 3.8	–	5.5 ± 3.1	–
COFFEEORMAIL	0.9 ± 0.1	1.0 ± 0.1	11.7 ± 0.6	11.2 ± 0.4	2.7 ± 1.8	2.4 ± 2.1

Table 5.15: Total RM learning time when symmetry breaking is disabled (No SB) and enabled (SB).

	Acyclic		Cyclic	
	No SB	SB	No SB	SB
COFFEE	0.5 ± 0.0	0.4 ± 0.0	0.5 ± 0.0	0.5 ± 0.0
COFFEEMAIL	277.4 ± 70.2	18.9 ± 3.3	$4204.3 \pm 1334.4^*$	774.7 ± 434.4
VISITABCD	1070.0 ± 725.6	163.2 ± 44.3	$3293.5 \pm 1199.2^*$	1961.7 ± 1123.8

Maximum Number of Edges between States

Table 5.14 shows that increasing κ from 1 to 2 negatively impacts RM learning due to a notable increase in the hypothesis space size. In the worst case, only one run does not time out for VISITABCD; besides, for COFFEEMAIL, the number of completed runs decreases from 20 to 15, and the running time becomes two orders of magnitude higher. In contrast, the number of examples and the example lengths remain similar for $\kappa = 1$.

Since the original OFFICEWORLD tasks have at most one edge from one state to another, we consider an additional task whose RM can only be learned with $\kappa > 1$. The COFFEEORMAIL task consists of going to the coffee or mail location (it does not matter which) and then going to the office while avoiding the decorations. A minimal RM consists of 4 and 5 states for $\kappa = 2$ and $\kappa = 1$, respectively. The COFFEE task is also characterized by a similar minimal RM consisting of 4 states regardless of the value of κ . The results are similar independently from the value of κ although $\kappa = 1$ results in a larger RM; however, it would not be enough to learn RMs requiring a single disjunctive transition from the initial state to the accepting state (e.g., “observe ☕ or ☒”).

Symmetry Breaking

Table 5.15 shows the symmetry breaking constraints’ effect on the time required to learn RMs. The number of examples and example lengths barely change when symmetry breaking is used, so we do not report the results. Symmetry breaking constraints speed up RM learning by an order of magnitude in COFFEEMAIL (acyclic, cyclic) and VISITABCD (acyclic); furthermore, no run times out when symmetry breaking is on, whereas two runs time out when these constraints are used and the RM is allowed to have cycles (one for COFFEEMAIL and one for VISITABCD).

5.3 Experiments in CRAFTWORLD

The CRAFTWORLD domain (Andreas et al., 2017; Toro Icarte et al., 2018a) consists of a 39×39 grid without walls. The grid contains raw materials (wood, grass, iron) and tools/workstations

(toolshed, workbench, factory, bridge, axe), which constitute the proposition set \mathcal{P} . Like in the OFFICEWORLD, (i) the agent moves in the four cardinal directions and remains in the same location upon attempting to go beyond the grid’s limits, and (ii) at each timestep, the agent knows its location coordinates (state) and observes the propositions in that location (label).

5.3.1 Instance Generation

The grids are randomly generated such that (i) all items must be in different locations (i.e., the labels consist of at most one proposition), and (ii) there are 5 and 2 labeled locations for each material and tool/workstation, respectively.

5.3.2 Tasks

The tasks consist in observing a specific sequence of materials and tools/workstations. We use the set of tasks proposed by Toro Icarte et al. (2018a):

1. MAKEPLANK: wood, toolshed.
2. MAKESTICK: wood, workbench.
3. MAKECLOTH: grass, factory.
4. MAKEROPE: grass, toolshed.
5. MAKESHEARS: iron, wood, workbench (the iron and the wood can be observed in any order).
6. MAKEBRIDGE: iron, wood, factory (the iron and the wood can be observed in any order).
7. GETGOLD: iron, wood, factory, bridge (the iron and the wood can be observed in any order).
8. MAKEBED: wood, toolshed, grass, workbench (the grass can be observed anytime before the workbench.)
9. MAKEAXE: wood, workbench, iron, toolshed (the iron can be observed anytime before the toolshed).
10. GETGEM: wood, workbench, iron, toolshed, axe (the iron can be observed anytime before the toolshed).

Tasks 1–4 have 2 subgoals and are represented by 3-state minimal RMs. Tasks 5–6 have 3 subgoals and are represented by 5-state minimal RMs. Task 7 has 4 subgoals and is represented by a 6-state minimal RM. Tasks 8–9 have 4 subgoals and are represented by 7-state minimal RMs. Task 10 has 5 subgoals and is represented by an 8-state minimal RM. In line with Assumption 3.1.1, the agent gets a reward of 1 upon achieving the goal and 0 otherwise. Unlike OFFICEWORLD, there are no dead-end histories in this domain; hence, the set of dead-end examples is always empty, and the rejecting state is unreachable. Appendix A.1 contains an illustration of a representative RM for each group of tasks.

Table 5.16: Hyperparameters used in the CRAFTWORLD experiments.

Learning rate α	0.1
Exploration rate ϵ	0.1
Discount factor γ	0.99
Number of episodes per instance	10,000
Avoid learning purely negative formulas	✓
Number of instances	100
Maximum episode length N	250
Trace compression	✓
Enforce acyclicity	✓
Number of disjuncts κ	1
Use restricted proposition set	✗

5.3.3 Hyperparameters

Table 5.16 lists the hyperparameters used in these experiments, where α , ϵ and γ are used for both the metapolicies and option policies across all HRL variants. The only difference with respect to the default parameters used in OFFICEWORLD is the number of instances: we experimentally observed that using 100 instances instead of 50 was a better choice for tasks 8–10 (as we explain later, RM learning occasionally time out for these).

5.3.4 Results

Table 5.17 shows the RM learning metrics for the presented CRAFTWORLD tasks using HRL_G. The tasks are divided into several groups according to their number of subgoals and the number of states of their minimal RMs. Figure 5.6 shows the learning curves for one representative of each group of tasks³ with and without interleaved RM learning. We observe the following:

- Like in the OFFICEWORLD tasks, the more subgoals and RM states, the higher the values for the collected metrics (running time, number of examples, and example length); besides, the number of goal examples still corresponds to the number of paths from the initial state to the accepting state. The figure shows that, as before, learning is more frequent for the harder tasks.
- The RM learning metrics are similar within the groups having simple RMs. The differences within each group become bigger (e.g., MAKEBED and MAKEAXE) as the tasks involve more subgoals since example lengths increase; consequently, observing two equivalent example sets for two different tasks is unlikely.
- The running time increases dramatically from GETGOLD to MAKEBED and MAKEAXE; indeed, the learner times out for the latter tasks (5/20 for MAKEBED and 4/20 for MAKEAXE). In the case of the hardest task, GETGEM, the learner times out on 9 occasions. The number of timeouts varies between algorithms, probably due to exploration, e.g. standard HRL times out on 8 and 1 runs for MAKEAXE and MAKEBED, respectively.
- The convergence rate difference between RM learning approaches (ISA-HRL, ISA-QRM) and approaches exploiting handcrafted RMs (HRL, QRM) is small for most tasks, showing that

³The learning curves are similar within each group; hence, we plot the curves for a representative of each group.

Table 5.17: RM learning metrics for the CRAFTWORLD tasks using HRL_G.

	Time (s.)	# Examples			Example Length
		All	G	I	
MAKEPLANK	0.2 ± 0.0	4.4 ± 0.3	1.4 ± 0.1	3.0 ± 0.3	2.2 ± 1.2
MAKESTICK	0.2 ± 0.0	3.6 ± 0.2	1.2 ± 0.1	2.4 ± 0.2	2.2 ± 1.4
MAKECLOTH	0.3 ± 0.0	4.9 ± 0.4	1.2 ± 0.1	3.6 ± 0.4	2.4 ± 1.5
MAKEROPE	0.2 ± 0.0	4.2 ± 0.3	1.2 ± 0.1	2.9 ± 0.3	2.4 ± 1.4
MAKESHEARS	2.0 ± 0.3	16.2 ± 0.8	3.3 ± 0.2	12.8 ± 0.8	3.4 ± 1.7
MAKEBRIDGE	1.7 ± 0.3	15.5 ± 1.3	3.0 ± 0.2	12.5 ± 1.2	3.0 ± 1.5
GETGOLD	60.7 ± 25.7	30.6 ± 3.2	2.2 ± 0.2	28.5 ± 3.2	4.0 ± 1.9
MAKEBED	$2140.4 \pm 1071.7^*$	$37.1 \pm 3.1^*$	$3.8 \pm 0.3^*$	$33.3 \pm 3.0^*$	$4.0 \pm 1.7^*$
MAKEAXE	$2990.3 \pm 717.7^*$	$46.6 \pm 3.5^*$	$3.3 \pm 0.2^*$	$43.3 \pm 3.5^*$	$4.3 \pm 1.9^*$
GETGEM	$6179.4 \pm 2784.8^*$	$116.8 \pm 14.7^*$	$1.2 \pm 0.1^*$	$115.6 \pm 14.7^*$	$5.2 \pm 2.0^*$

the learned RMs are useful to learn a policy that achieves the goal. The gaps for the hardest tasks (MAKEAXE, MAKEBED, and GETGEM) are usually larger due to the timeouts in the RM learning approaches.

- Given a handcrafted RM, QRM_{\min} is equivalent to QRM_{\max} since the minimum and maximum distances to the accepting state are the same in these RMs. While the curves for ISA- QRM_{\min} and ISA- QRM_{\max} are similar, they are not identical since the intermediate RMs can cause variations.
- The approaches not using guidance (HRL, QRM) start converging faster than those that use it (HRL_G, QRM_{\min} , QRM_{\max}); however, the latter eventually learn to reach the accepting state earlier. The initially slower convergence of the guidance-based approaches might be because guidance results in more initial exploration.

5.4 Experiments in WATERWORLD

The WATERWORLD domain (Karpathy, 2015; Sidor, 2016; Toro Icarte et al., 2018a), which is illustrated in Figure 5.7, consists of a 2D box containing 12 balls of 6 different colors (2 balls per color). Each ball moves at a constant speed in a given direction. The balls bounce only when they collide with a wall. The agent is a white ball that can change its velocity in any of the four cardinal directions. The proposition set $\mathcal{P} = \{r, g, b, c, m, y\}$ is formed by the balls' colors. At each step, the agent observes (i) a state vector containing the absolute position and velocity of the agent and the relative positions and velocities of the other balls, and (ii) a label containing the color of the balls it overlaps. For example, the agent observes label $\{g\}$ (green) in Figure 5.7. Note that several balls can overlap at the same time; thus, the agent can observe several colors/propositions simultaneously.

5.4.1 Instance Generation

The balls are placed randomly and given a random direction at the beginning of each episode.

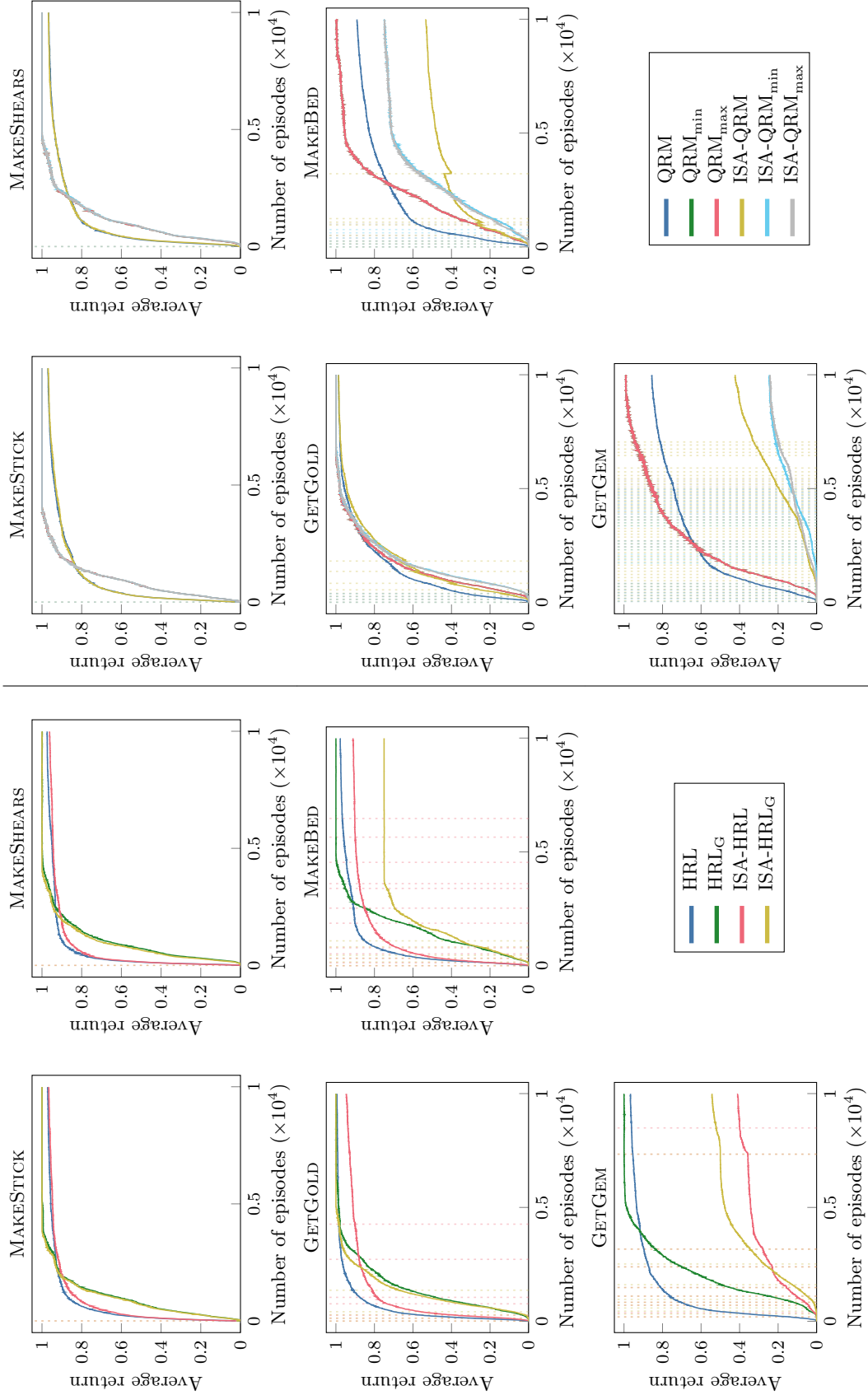


Figure 5.6: Learning curves for different RL algorithms in the CRAFTWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM). The ISA curves for MAKEBED and GETGEM are below the handcrafted RM curves because of the timeout runs. When a run finishes successfully, both types of curves are similar.

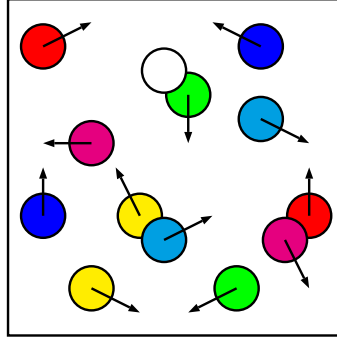


Figure 5.7: The WATERWORLD domain (Karpathy, 2015; Sidor, 2016; Toro Icarte et al., 2018a).

5.4.2 Tasks

The tasks we consider consist of observing a sequence of colors in a specific order:

- RGB: red (r) then green (g) then blue (b). It consists of 3 subgoals and is represented by a 4-state minimal RM.
- RG-B: red (r) then green (g) and (independently) blue (b). Note that there are two sequences that can be interleaved. For instance, $\langle \{r\}, \{g\}, \{b\} \rangle$, $\langle \{b\}, \{r\}, \{g\} \rangle$ and $\langle \{r\}, \{b\}, \{g\} \rangle$ are three possible goal traces. It consists of 3 subgoals and is represented by a 6-state minimal RM. The RGB task is a subcase of this one.
- RGBC: red (r) then green (g) then blue (b) then cyan (c). It consists of 4 subgoals and is represented by a 5-state minimal RM.

The agent gets a reward of 1 upon achieving the goal and 0 otherwise, complying with Assumption 3.1.1. Dead-end histories are not observable, like in CRAFTWORLD. If the agent observes more than one proposition at a time, it may achieve several subgoals simultaneously; for instance, if the agent has not overlapped with any ball yet and observes $\{r, g\}$ while performing RGB, only b remains to be observed to complete the task. We refer the reader to Appendix A.1 for an illustration of the RMs for each task.

5.4.3 Hyperparameters

Unlike OFFICEWORLD and CRAFTWORLD, the state space is continuous, so tabular learning approaches are not applicable; instead, like Toro Icarte et al. (2018a), we approximate the value functions through DDQNs consisting of a multilayer perceptron (MLP) with 4 hidden layers of 64 rectifier units each. The networks are trained using the Adam optimizer (Kingma and Ba, 2015). Table 5.18 lists the remaining hyperparameters, where α , ϵ and γ are used for both the metapolicies and option policies across all HRL variants. In this case, the agent does not only learn an RM that generalizes to several instances, but also a single general policy. The number of episodes per instance is 1 since the environment is randomly reset at the beginning of each episode.

Table 5.18: Hyperparameters used in the WATERWORLD experiments.

Learning rate α	1×10^{-5}
Exploration rate ϵ	0.1
Discount factor γ	0.9
Number of episodes per instance	1
Maximum episode length N	150
Replay memory size	50,000
Replay start size	1,000
Batch size	32
Number of instances	50,000
Target network update frequency	100
Trace compression	✓
Enforce acyclicity	✓
Number of disjuncts κ	1
Avoid learning purely negative formulas	✓
Use restricted proposition set	✗

Table 5.19: RM learning metrics for the WATERWORLD tasks using HRL_G.

	Time (s.)	# Examples			Example Length
		All	G	I	
RGB	3.7 ± 0.3	26.9 ± 1.1	7.0 ± 0.4	20.0 ± 0.9	4.3 ± 2.3
RG-B	129.5 ± 23.9	48.4 ± 1.2	15.4 ± 0.4	33.0 ± 1.0	4.4 ± 2.2
RGBC	111.8 ± 23.1	61.4 ± 2.7	11.7 ± 0.5	49.6 ± 2.4	5.5 ± 2.7

5.4.4 Results

Figure 5.8 shows the learning curves for the WATERWORLD tasks, which are obtained by evaluating the greedy policy 10 times (each on a randomly generated instance) every 500 episodes and averaging the resulting undiscounted return. Table 5.19 shows the RM learning metrics using HRL_G. We observe the following:

- The tasks with more subgoals require learning RMs more often; for instance, RM learning occurs throughout the entire interaction for RGBC, whereas it is concentrated at the beginning for RGB.
- The convergence rate for RGB is lower than for RG-B. We hypothesize this is because the former has fewer ways of achieving the goal (i.e., the reward is sparser) than the latter. However, the time and number of examples needed to learn a minimal RM for RG-B are higher than for RGB since the RM state set is larger. Finally, we observe that RG-B requires more goal traces, showing there are more paths towards the goal and, hence, making RM learning more challenging since all these paths must be captured.
- The learning time for RGBC is lower than for RG-B since the minimal RM for the latter has more states (hence, making the hypothesis space larger). In contrast, RGBC requires more examples since it has more subgoals, as observed in the previous domains; in particular, RGBC involves more incomplete examples, possibly because there are more subgoal sequences to be discarded.

- HRL-based approaches perform better than the QRM-based ones, especially in RGBC. We hypothesize there are two possible causes for this behavior:
 - (i) The resetting of the value functions. Upon learning a new RM, HRL approaches only reset the metapolicies, while QRM approaches reset all value functions. Since RMs are learned throughout the entire interaction, QRM agents rarely can converge to a stable policy.
 - (ii) While the HRL agent commits to satisfying a given formula (determined by the metapolicy) at a given step, the QRM agent selects the globally best action at each step. Unlike the previous grid-world domains, the propositions here constantly change their position, so learning a value function that generalizes to many instances is challenging.⁴

To determine which of these two causes is more plausible, we examine the performance of QRM with a handcrafted RM. In general, the performances of approaches using RM learning are very similar to those using a handcrafted RM, which shows that the forgetting effect is not as present in WATERWORLD as in the grid-world domains. The replay buffer likely alleviates the forgetting effect since value functions can be updated from past experiences without ‘relying’ them. Therefore, we conclude that our experiments better support the second cause.

- There is barely any difference between using guidance or not regarding convergence rate. Camacho et al. (2019) shows a similar behavior for QRM using handcrafted RMs with a different reward shaping mechanism. We hypothesize that since the value functions must generalize to different instances, an agent not using guidance might explore similarly to an agent that does use it; therefore, using guidance does not help much in these tasks.

5.5 Summary

Throughout this chapter, we have observed several commonalities between the results across domains. We provide a summary of the main common findings below:

- The higher the number of subgoals and required states are, the higher the collected metrics values (running time, number of examples, and example length).
- The number of goal examples used to learn an RM is approximately the same as the number of paths from the initial state to the accepting state. Incomplete and dead-end examples refine those paths and increase as the number of subgoals and RM states increases.
- Using auxiliary reward signals speeds up convergence in grid-world tasks. As mainly shown in OFFICeworld, it helps to learn an RM early in the interaction and thus reduces relearning later, which is extremely helpful in QRM since it resets all the value functions when a new RM is learned. In contrast, guidance does not provide faster convergence in WATERWORLD.

⁴We highlight that our evaluation of QRM in the WATERWORLD domain differs from that by Toro Icarte et al. (2018a). While we randomly initialize the environment at the start of every episode, Toro Icarte et al. use a fixed map. In the latter case, QRM quickly converges in RGBC—indeed, we have reproduced the results.

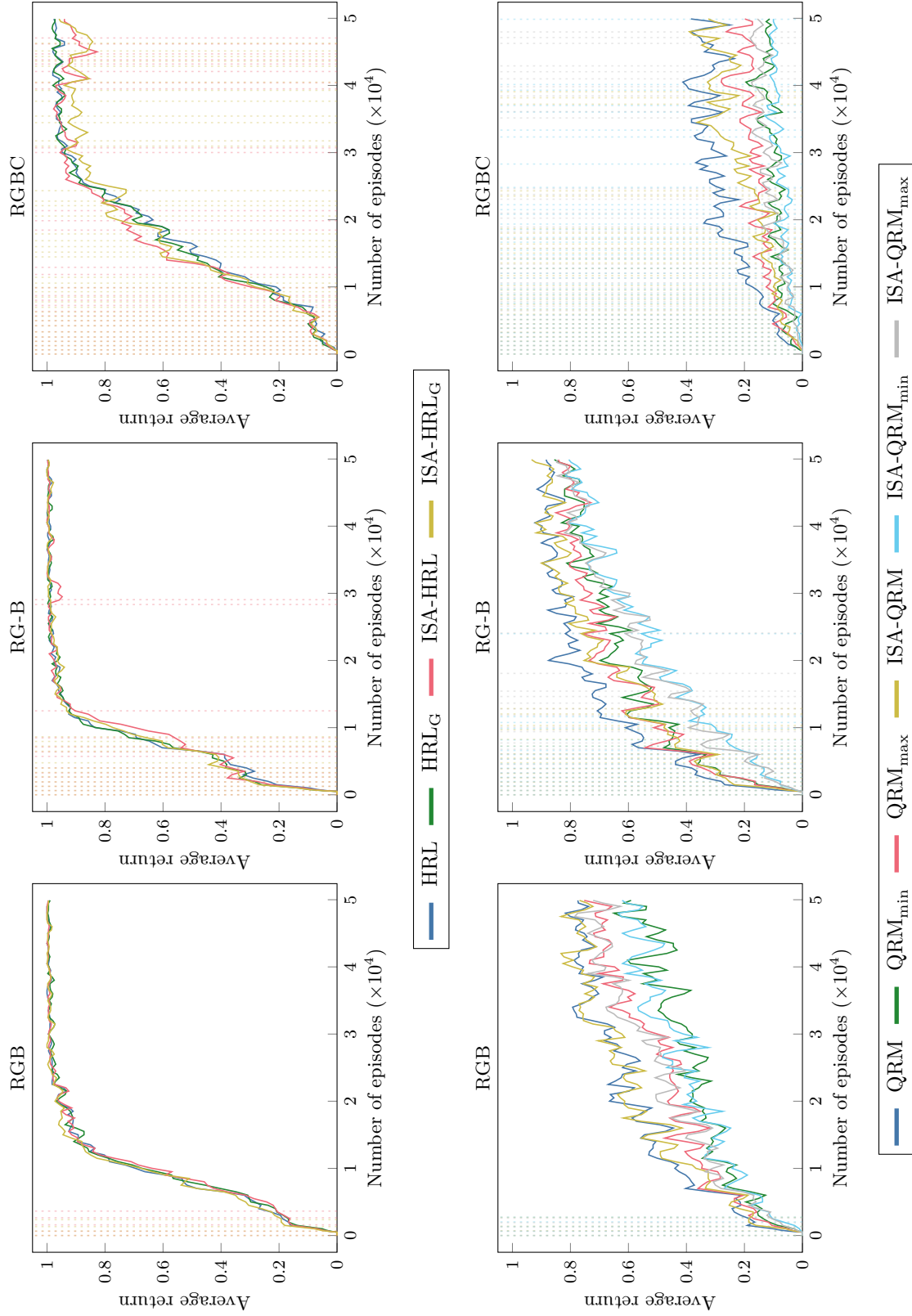


Figure 5.8: Learning curves for different RL algorithms in the WATERWORLD tasks when interleaved RM learning is off (HRL, QRM) and on (ISA-HRL, ISA-QRM).

- HRL approaches converge faster than QRM approaches, especially without guidance. In these contexts, QRM needs a formula labeling a transition to the accepting state to be satisfied to start inducing changes in the value functions. In contrast, HRL updates the value functions of all the formulas independently and, importantly, does not need to reset them when a new RM is learned.
- The performance of approaches that learn RMs resembles that of analogous approaches that exploit handcrafted RMs, except when the RMs cannot be learned under the specified timeout; thus, the learned RMs accurately represent the structure of the tasks.

Learning reward machines is costly and incurs long running times as the number of constituent states grows. We alleviate this scalability problem in Part II by learning several small RMs that can be composed hierarchically instead of learning a single large RM. Further, we examine how learning independent policies for each of these small RMs helps tackle sparse reward tasks more efficiently.

Part II

Hierarchies of Reward Machines

Chapter 6

Formalism of Hierarchies of Reward Machines

In this part of the thesis, we introduce *hierarchies of reward machines*, a formalism for hierarchically composing reward machines by endowing them with the ability to call each other. To motivate the existence of these hierarchies, let us consider the CRAFTWORLD domain illustrated in Figure 6.1. In this domain, the agent (\blacktriangle) can move forward or rotate 90° , staying put if it moves towards a wall. Locations are labeled with propositions from $\mathcal{P} = \{\text{A}, \text{B}, \text{C}, \text{D}, \text{E}, \text{F}, \text{G}, \text{H}, \text{I}, \text{J}, \text{K}, \text{L}, \text{M}, \text{N}, \text{O}, \text{P}, \text{Q}, \text{R}, \text{S}, \text{T}, \text{U}, \text{V}, \text{W}, \text{X}, \text{Y}, \text{Z}, \text{AA}, \text{AB}, \text{AC}, \text{AD}, \text{AE}, \text{AF}, \text{AG}, \text{AH}, \text{AI}, \text{AJ}, \text{AK}, \text{AL}, \text{AM}, \text{AN}, \text{AO}, \text{AP}, \text{AQ}, \text{AR}, \text{AS}, \text{AT}, \text{AU}, \text{AV}, \text{AW}, \text{AX}, \text{AY}, \text{AZ}, \text{BA}, \text{BB}, \text{BC}, \text{BD}, \text{BE}, \text{BF}, \text{BG}, \text{BH}, \text{BI}, \text{BJ}, \text{BK}, \text{BL}, \text{BM}, \text{BN}, \text{BO}, \text{BP}, \text{BQ}, \text{BR}, \text{BS}, \text{BT}, \text{BU}, \text{BV}, \text{BW}, \text{BX}, \text{BY}, \text{BZ}, \text{CA}, \text{CB}, \text{CC}, \text{CD}, \text{CE}, \text{CF}, \text{CG}, \text{CH}, \text{CI}, \text{CJ}, \text{CK}, \text{CL}, \text{CM}, \text{CN}, \text{CO}, \text{CP}, \text{CQ}, \text{CR}, \text{CS}, \text{CT}, \text{CU}, \text{CV}, \text{CW}, \text{CX}, \text{CY}, \text{CZ}, \text{DA}, \text{DB}, \text{DC}, \text{DD}, \text{DE}, \text{DF}, \text{DG}, \text{DH}, \text{DI}, \text{DJ}, \text{DK}, \text{DL}, \text{DM}, \text{DN}, \text{DO}, \text{DP}, \text{DQ}, \text{DR}, \text{DS}, \text{DT}, \text{DU}, \text{DV}, \text{DW}, \text{DX}, \text{DY}, \text{DZ}, \text{EA}, \text{EB}, \text{EC}, \text{ED}, \text{EE}, \text{EF}, \text{EG}, \text{EH}, \text{EI}, \text{EJ}, \text{EK}, \text{EL}, \text{EM}, \text{EN}, \text{EO}, \text{EP}, \text{EQ}, \text{ER}, \text{ES}, \text{ET}, \text{EU}, \text{EV}, \text{EW}, \text{EX}, \text{EY}, \text{EZ}, \text{FA}, \text{FB}, \text{FC}, \text{FD}, \text{FE}, \text{FF}, \text{FG}, \text{FH}, \text{FI}, \text{FJ}, \text{FK}, \text{FL}, \text{FM}, \text{FN}, \text{FO}, \text{FP}, \text{FQ}, \text{FR}, \text{FS}, \text{FT}, \text{FU}, \text{FV}, \text{FW}, \text{FX}, \text{FY}, \text{FZ}, \text{GA}, \text{GB}, \text{GC}, \text{GD}, \text{GE}, \text{GF}, \text{GG}, \text{GH}, \text{GI}, \text{GJ}, \text{GK}, \text{GL}, \text{GM}, \text{GN}, \text{GO}, \text{GP}, \text{GQ}, \text{GR}, \text{GS}, \text{GT}, \text{GU}, \text{GV}, \text{GW}, \text{GX}, \text{GY}, \text{GZ}, \text{HA}, \text{HB}, \text{HC}, \text{HD}, \text{HE}, \text{HF}, \text{HG}, \text{HH}, \text{HI}, \text{HJ}, \text{HK}, \text{HL}, \text{HM}, \text{HN}, \text{HO}, \text{HP}, \text{HQ}, \text{HR}, \text{HS}, \text{HT}, \text{HU}, \text{HV}, \text{HW}, \text{HX}, \text{HY}, \text{HZ}, \text{IA}, \text{IB}, \text{IC}, \text{ID}, \text{IE}, \text{IF}, \text{IG}, \text{IH}, \text{II}, \text{IJ}, \text{IK}, \text{IL}, \text{IM}, \text{IN}, \text{IO}, \text{IP}, \text{IQ}, \text{IR}, \text{IS}, \text{IT}, \text{IU}, \text{IV}, \text{IW}, \text{IX}, \text{IY}, \text{IZ}, \text{JA}, \text{JB}, \text{JC}, \text{JD}, \text{JE}, \text{JF}, \text{JG}, \text{JH}, \text{JI}, \text{JJ}, \text{JK}, \text{JL}, \text{JM}, \text{JN}, \text{JO}, \text{JP}, \text{JQ}, \text{JR}, \text{JS}, \text{JT}, \text{JU}, \text{JV}, \text{JW}, \text{JX}, \text{JY}, \text{JZ}, \text{KA}, \text{KB}, \text{KC}, \text{KD}, \text{KE}, \text{KF}, \text{KG}, \text{KH}, \text{KI}, \text{KJ}, \text{KL}, \text{KM}, \text{KN}, \text{KO}, \text{KP}, \text{KQ}, \text{KR}, \text{KS}, \text{KT}, \text{KU}, \text{KV}, \text{KW}, \text{KX}, \text{KY}, \text{KZ}, \text{LA}, \text{LB}, \text{LC}, \text{LD}, \text{LE}, \text{LF}, \text{LG}, \text{LH}, \text{LI}, \text{LJ}, \text{LK}, \text{LL}, \text{LM}, \text{LN}, \text{LO}, \text{LP}, \text{LQ}, \text{LR}, \text{LS}, \text{LT}, \text{LU}, \text{LV}, \text{LW}, \text{LX}, \text{LY}, \text{LZ}, \text{MA}, \text{MB}, \text{MC}, \text{MD}, \text{ME}, \text{MF}, \text{MG}, \text{MH}, \text{MI}, \text{MJ}, \text{MK}, \text{ML}, \text{MN}, \text{MO}, \text{MP}, \text{MQ}, \text{MR}, \text{MS}, \text{MT}, \text{MU}, \text{MV}, \text{MW}, \text{MX}, \text{MY}, \text{MZ}, \text{NA}, \text{NB}, \text{NC}, \text{ND}, \text{NE}, \text{NF}, \text{NG}, \text{NH}, \text{NI}, \text{NJ}, \text{NK}, \text{NL}, \text{NM}, \text{NO}, \text{NP}, \text{NQ}, \text{NR}, \text{NS}, \text{NT}, \text{NU}, \text{NV}, \text{NW}, \text{NX}, \text{NY}, \text{NZ}, \text{OA}, \text{OB}, \text{OC}, \text{OD}, \text{OE}, \text{OF}, \text{OG}, \text{OH}, \text{OI}, \text{OJ}, \text{OK}, \text{OL}, \text{OM}, \text{ON}, \text{OO}, \text{OP}, \text{OQ}, \text{OR}, \text{OS}, \text{OT}, \text{OU}, \text{OV}, \text{OW}, \text{OX}, \text{OY}, \text{OZ}, \text{PA}, \text{PB}, \text{PC}, \text{PD}, \text{PE}, \text{PF}, \text{PG}, \text{PH}, \text{PI}, \text{PJ}, \text{PK}, \text{PL}, \text{PM}, \text{PN}, \text{PO}, \text{PP}, \text{PQ}, \text{PR}, \text{PS}, \text{PT}, \text{PU}, \text{PV}, \text{PW}, \text{PX}, \text{PY}, \text{PZ}, \text{QA}, \text{QB}, \text{QC}, \text{QD}, \text{QE}, \text{QF}, \text{QG}, \text{QH}, \text{QI}, \text{QJ}, \text{QK}, \text{QL}, \text{QM}, \text{QN}, \text{QO}, \text{QP}, \text{QQ}, \text{QR}, \text{QS}, \text{QT}, \text{QU}, \text{QV}, \text{QW}, \text{QX}, \text{QY}, \text{QZ}, \text{RA}, \text{RB}, \text{RC}, \text{RD}, \text{RE}, \text{RF}, \text{RG}, \text{RH}, \text{RI}, \text{RJ}, \text{RK}, \text{RL}, \text{RM}, \text{RN}, \text{RO}, \text{RP}, \text{RQ}, \text{RR}, \text{RS}, \text{RT}, \text{RU}, \text{RV}, \text{RW}, \text{RX}, \text{RY}, \text{RZ}, \text{SA}, \text{SB}, \text{SC}, \text{SD}, \text{SE}, \text{SF}, \text{SG}, \text{SH}, \text{SI}, \text{SJ}, \text{SK}, \text{SL}, \text{SM}, \text{SN}, \text{SO}, \text{SP}, \text{SQ}, \text{SR}, \text{SS}, \text{ST}, \text{SU}, \text{SV}, \text{SW}, \text{SX}, \text{SY}, \text{SZ}, \text{TA}, \text{TB}, \text{TC}, \text{TD}, \text{TE}, \text{TF}, \text{TG}, \text{TH}, \text{TI}, \text{TJ}, \text{TK}, \text{TL}, \text{TM}, \text{TN}, \text{TO}, \text{TP}, \text{TQ}, \text{TR}, \text{TS}, \text{TT}, \text{TU}, \text{TV}, \text{TW}, \text{TX}, \text{TY}, \text{TZ}, \text{UA}, \text{UB}, \text{UC}, \text{UD}, \text{UE}, \text{UF}, \text{UG}, \text{UH}, \text{UI}, \text{UJ}, \text{UK}, \text{UL}, \text{UM}, \text{UN}, \text{UO}, \text{UP}, \text{UQ}, \text{UR}, \text{US}, \text{UT}, \text{UU}, \text{UV}, \text{UW}, \text{UX}, \text{UY}, \text{UZ}, \text{VA}, \text{VB}, \text{VC}, \text{VD}, \text{VE}, \text{VF}, \text{VG}, \text{VH}, \text{VI}, \text{VJ}, \text{VK}, \text{VL}, \text{VM}, \text{VN}, \text{VO}, \text{VP}, \text{VQ}, \text{VR}, \text{VS}, \text{VT}, \text{VU}, \text{VV}, \text{VW}, \text{VX}, \text{VY}, \text{VZ}, \text{WA}, \text{WB}, \text{WC}, \text{WD}, \text{WE}, \text{WF}, \text{WG}, \text{WH}, \text{WI}, \text{WJ}, \text{WK}, \text{WL}, \text{WM}, \text{WN}, \text{WO}, \text{WP}, \text{WQ}, \text{WR}, \text{WS}, \text{WT}, \text{WU}, \text{WV}, \text{WW}, \text{WX}, \text{WY}, \text{WZ}, \text{XA}, \text{XB}, \text{XC}, \text{XD}, \text{XE}, \text{XF}, \text{XG}, \text{XH}, \text{XI}, \text{XJ}, \text{XK}, \text{XL}, \text{XM}, \text{XN}, \text{XO}, \text{XP}, \text{XQ}, \text{XR}, \text{XS}, \text{XT}, \text{XU}, \text{XV}, \text{XW}, \text{XX}, \text{XY}, \text{XZ}, \text{YA}, \text{YB}, \text{YC}, \text{YD}, \text{YE}, \text{YF}, \text{YG}, \text{YH}, \text{YI}, \text{YJ}, \text{YK}, \text{YL}, \text{YM}, \text{YN}, \text{YO}, \text{YP}, \text{YQ}, \text{YR}, \text{YS}, \text{YT}, \text{YU}, \text{YV}, \text{YW}, \text{YX}, \text{YY}, \text{YZ}, \text{ZA}, \text{ZB}, \text{ZC}, \text{ZD}, \text{ZE}, \text{ZF}, \text{ZG}, \text{ZH}, \text{ZI}, \text{ZJ}, \text{ZK}, \text{ZL}, \text{ZM}, \text{ZN}, \text{ZO}, \text{ZP}, \text{ZQ}, \text{ZR}, \text{ZS}, \text{ZT}, \text{ZU}, \text{ZV}, \text{ZW}, \text{ZX}, \text{ZY}, \text{ZZ}\}$. The agent observes the propositions it steps on, e.g. the labeling function returns $\{\text{A}\}$ in the top-left corner. Table 6.1 lists the tasks we consider, which consist of observing a sequence of propositions while avoiding the lava (\blacklozenge) if present. These tasks are based on those by Andreas et al. (2017) and Toro Icarte et al. (2018a), but are definable in terms of each other; likewise, it is intuitive to compactly express an RM as a composition of other RMs.

Example 6.0.1. *Figure 6.2a shows a reward machine for BOOK, which consists of performing PAPER and LEATHER in any order followed by going to location A. The left and right paths from u^0 perform PAPER and LEATHER in a different order; hence, the representation of each subtask appears twice in the RM, once for each path. Intuitively, the presented RM could be compactly represented by encapsulating each subtask into separate callable RMs.*

Hierarchically composing RMs allows for reusing RMs across hierarchies for different tasks. Throughout this part of the thesis, we formalize these hierarchies and examine how the enabled reusability is leveraged to exploit and learn them. On the exploitation side, we present an HRL-based method that treats each RM in the hierarchy as an independently solvable subtask, effectively learning policies over the hierarchy at arbitrarily many timescales. Importantly, the RM-associated policies are reusable within different hierarchies since they are trained independently of the global task. On the learning side, we introduce a curriculum-based method for learning the hierarchies given a set of tasks such that hierarchies for complex tasks build on those learned for simpler tasks.

In this chapter, we start by formally defining hierarchies of reward machines (Section 6.1). Next, we describe two *equivalence properties* between the proposed hierarchies and standard reward machines (Section 6.2). Finally, we explain how hierarchies of reward machines are represented using ASP (Section 6.3). The tasks we here consider are formalized in Chapter 3.

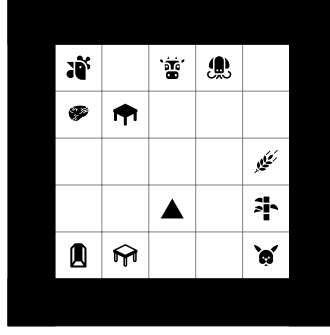


Figure 6.1: An instance of the CRAFTWORLD domain.

Table 6.1: List of CRAFTWORLD tasks. Descriptions “ $x ; y$ ” express sequential order (observe/do x then y), descriptions “ $x \& y$ ” express that x and y can be observed/done in any order, and h is the root RM’s height.

Task	h	Description	Task	h	Description
BATTER	1	(EGG & BEE) ; HOUSE	BOOK	2	(PAPER & LEATHER) ; HOUSE
BUCKET	1	BOOK ; HOUSE	MAP	2	(PAPER & COMPASS) ; HOUSE
COMPASS	1	(BOOK & EGG) ; HOUSE	MILKBUCKET	2	BUCKET ; BEE
LEATHER	1	EGG ; HOUSE	BOOKQUILL	3	BOOK & QUILL
PAPER	1	CROSS ; HOUSE	SWEETMILK	3	MILKBUCKET & SUGAR
QUILL	1	(BEE & BEE) ; HOUSE	CAKE	4	BATTER ; SWEETMILK ; HOUSE
SUGAR	1	CROSS ; HOUSE			

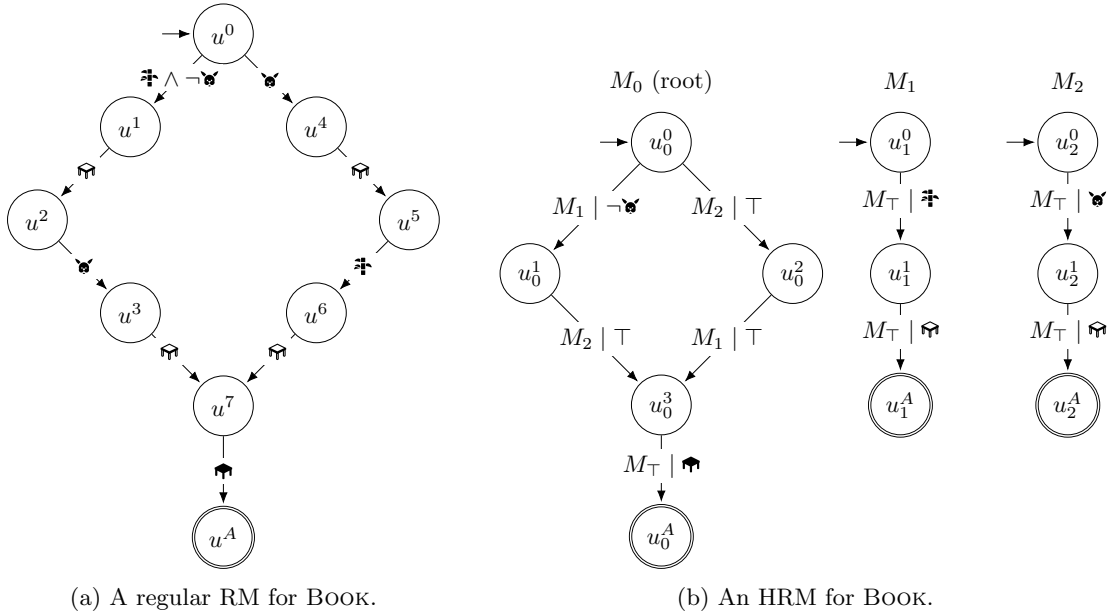


Figure 6.2: A standard RM and an HRM for BOOK. In (a), an edge from u to u' is labeled $\varphi(u, u')$. In (b), an edge from u to u' in RM M_i is labeled $M_j \mid \varphi_i(u, u', M_j)$. In both cases, accepting states are double circled, and loops are omitted.

6.1 Hierarchies of Reward Machines

In this section, we introduce the formalism for hierarchically composing reward machines. We split the definition into two parts. First, we define the concepts related to the structure of the hierarchies and state the underlying assumptions (Section 6.1.1). Then, we formally describe how an HRM processes a trace (Section 6.1.2).

6.1.1 Structure

To constitute a hierarchy of RMs, we need to endow RMs with the ability to call each other. We redefine the reward machines from Chapter 3 to enable calls through the logical transition function; besides, to simplify the formalization, reward machines include sets of accepting and rejecting states instead of a single accepting state and a single rejecting state.

Definition 6.1.1 (Reward machine). *Given a set of reward machines \mathcal{M} , a reward machine is a tuple $M = \langle \mathcal{U}, \mathcal{P}, \varphi, r, u^0, \mathcal{U}^A, \mathcal{U}^R \rangle$, where:*

- \mathcal{U} is a finite set of states;
- \mathcal{P} is a finite set of propositions that constitutes the alphabet of the reward machine;
- $\varphi : \mathcal{U} \times \mathcal{U} \times \mathcal{M} \rightarrow \text{DNF}_{\mathcal{P}}$ is the logical transition function such that $\varphi(u, u', M)$ denotes the DNF formula over \mathcal{P} that must be satisfied to transition from $u \in \mathcal{U}$ to $u' \in \mathcal{U}$ by calling RM $M \in \mathcal{M}$;
- $r : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ is the reward-transition function, which outputs the reward associated with a state transition;
- $u^0 \in \mathcal{U}$ is the unique initial state;
- $\mathcal{U}^A \subseteq \mathcal{U}$ is a set of accepting states denoting the task's goal achievement; and
- $\mathcal{U}^R \subseteq \mathcal{U}$ is a set of rejecting states denoting the unfeasibility of achieving the task's goal.

We refer to the formulas $\varphi(u, u', M)$ as *contexts* since they represent conditions under which calls are made. As we shall see later, contexts help preserve determinism and must be satisfied to start a call (a necessary but not sufficient condition). The hierarchies we consider contain an RM M_{\top} called the *leaf* RM, which solely consists of an accepting state (i.e., $\mathcal{U}_{\top} = \mathcal{U}_{\top}^A = \{u_{\top}^0\}$), and immediately returns control to the RM that calls it; indeed, as described later, reaching the accepting state of a called RM results in returning control to the calling RM. Next, we formally define the hierarchies and make some assumptions about their structure.

Definition 6.1.2 (Hierarchy of reward machines). *A hierarchy of reward machines (HRM) is a tuple $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, where $\mathcal{M} = \{M_0, \dots, M_{m-1}\} \cup \{M_{\top}\}$ is a set of m RMs and the leaf RM M_{\top} , $M_r \in \mathcal{M} \setminus \{M_{\top}\}$ is the root RM, and \mathcal{P} is a finite set of propositions used by all constituent RMs.*

Assumption 6.1.1. *HRMs do not have circular dependencies; that is, an RM cannot be called back from itself, including recursion.*

Assumption 6.1.2. *Rejecting states are global; that is, they cause the root task to fail.*

Assumption 6.1.3. *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, the logical transition function φ_i of each RM $M_i \in \mathcal{M}$ in the hierarchy is such that $\varphi_i(u, u, M) = \perp$ for $u \in \mathcal{U}$ and $M \in \mathcal{M}$.*

Assumption 6.1.4. *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, the logical transition function φ_i of each RM $M_i \in \mathcal{M}$ in the hierarchy is such that $\varphi_i(u, u', M) = \perp$ for $u \in \mathcal{U}^A \cup \mathcal{U}^R$, $u' \in \mathcal{U}$ and $M \in \mathcal{M}$.*

Assumption 6.1.5. *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, the reward-transition function of any RM M_i is $r_i(u, u') = \mathbb{1}[u \notin \mathcal{U}_i^A \wedge u' \in \mathcal{U}_i^A]$.*

Given Assumption 6.1.1, each RM M_i has a *height* h_i , which corresponds to the maximum number of nested calls needed to reach the leaf. Formally, if $i = \top$, then $h_i = 0$; otherwise, $h_i = 1 + \max_j h_j$, where j ranges over all RMs called by M_i (i.e., there exists $\langle u, v \rangle \in \mathcal{U}_i \times \mathcal{U}_i$ such that $\varphi_i(u, v, M_j) \neq \perp$). Table 6.1 shows the root's height of the HRMs of each CRAFTWORLD task. Assumption 6.1.2 is required to preserve the equivalence with standard RMs (see Section 6.2). Assumptions 6.1.3–6.1.5 are analogous to Assumptions 3.2.1–3.2.3 made in the context of standard RMs. Assumption 6.1.3 ensures that the condition labeling self-loops is unsatisfiable; indeed, as detailed later, self-loops are taken if the state cannot be left. Assumption 6.1.4 indicates that there are no outgoing transitions from accepting and rejecting states. Assumption 6.1.5 is made as per Assumption 3.1.1. The agent observes the rewards coming from the reward-transition function of the root.

Example 6.1.1. *Figure 6.2b shows the HRM for BOOK, whose root has height $h = 2$. The PAPER and LEATHER RMs invoked by the root have $h = 1$. The context $\neg \text{✂}$ in the call to M_1 preserves determinism, as detailed later.*

6.1.2 Traversal

We here describe how an HRM processes a label trace. To indicate where the agent is in an HRM, we define the notion of *hierarchy states*.

Definition 6.1.3 (Hierarchy state). *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, a hierarchy state is a tuple $\langle M_i, u, \Phi, \Gamma \rangle$, where $M_i \in \mathcal{M}$ is an RM, $u \in \mathcal{U}_i$ is a state, $\Phi \in \text{DNF}_{\mathcal{P}}$ is an accumulated context, and Γ is a call stack.*

As we will see later, *accumulated contexts* result from the conjunction of call contexts at different heights in the hierarchy; in other words, it is the *full* condition under which a call from a given state is made. By Assumption 6.1.1, the accumulated context is always \top at the root since no RM can call the root. Next, we define the call stack that constitutes hierarchy states.

Definition 6.1.4 (Call stack). *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, a call stack Γ contains tuples $\langle u, v, M_i, M_j, \phi, \Phi \rangle$, each denoting a call where $u \in \mathcal{U}_i$ is the state from which the call is made; $v \in \mathcal{U}_i$ is the next state in the calling RM $M_i \in \mathcal{M}$ after reaching an accepting state of the called RM $M_j \in \mathcal{M}$; $\phi \in \text{DNF}_{\mathcal{P}}$ are the disjuncts of $\varphi_i(u, v, M_j)$ satisfied by a label; and $\Phi \in \text{DNF}_{\mathcal{P}}$ is the accumulated context.*

Call stacks determine where to resume the execution. Each RM appears in the stack at most once since, by Assumption 6.1.1, HRMs have no circular dependencies. We use $\Gamma \oplus \langle u, v, M_i, M_j, \phi, \Phi \rangle$

to denote a stack recursively defined by a stack Γ and a top element $\langle u, v, M_i, M_j, \phi, \Phi \rangle$. The initial hierarchy state of an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ is $\langle M_r, u_r^0, \top, [] \rangle$: we are in the initial state of the root, there is no accumulated context, and the stack is empty. We remark two design decisions on the tuples $\langle u, v, M_i, M_j, \phi, \Phi \rangle$ motivated by the exploitation algorithm described in Chapter 7:

1. The fundamental elements for the formalism are the calling machine M_i , the called machine M_j , and the state v after returning control to the former machine. In contrast, the state u from which the call is made, the context ϕ , and the accumulated context Φ are just memorized so that the agent can update its choices based on the actual transitions taken in the HRM.
2. The context ϕ is not necessarily $\varphi_i(u, v, M_j)$, but the DNF formula formed by the disjuncts of $\varphi_i(u, v, M_j)$ satisfied by a label. Like the above, this is important for updating the agent's choices based on the actually satisfied conditions in the HRM. In the following paragraphs, $\varphi(\mathcal{L})$ denotes the DNF formula formed by the disjuncts of a DNF formula $\varphi \in \text{DNF}_{\mathcal{P}}$ satisfied by a label $\mathcal{L} \subseteq \mathcal{P}$; for instance, if $\varphi = (a \wedge \neg d) \vee b \vee \neg c$ and $\mathcal{L} = \{a\}$, then $\varphi(\mathcal{L}) = (a \wedge \neg d) \vee \neg c$.

At the beginning of this section, we mentioned that satisfying the context of a call is a necessary but not sufficient condition to start the call. We now introduce a sufficient condition, called *exit condition*.

Definition 6.1.5 (Exit condition). *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ and a hierarchy state $\langle M_i, u, \Phi, \Gamma \rangle$, the exit condition $\xi_{i,u,\Phi} \in \text{DNF}_{\mathcal{P}}$ is the formula that must be satisfied to leave that hierarchy state. Formally,*

$$\xi_{i,u,\Phi} = \begin{cases} \Phi & \text{if } i = \top; \\ \bigvee_{\substack{\phi = \varphi_i(u,v,M_j), \\ \phi \neq \perp, v \in \mathcal{U}_i, M_j \in \mathcal{M}}} \xi_{j,u_j^0, \text{DNF}(\Phi \wedge \phi)} & \text{otherwise,} \end{cases}$$

where $\text{DNF}(\Phi \wedge \phi)$ is $\Phi \wedge \phi$ in DNF. The formula is Φ if $M_i = M_{\top}$ since it always returns control once called; otherwise, the formula is recursively defined as the disjunction of the exit conditions from the initial state of the called RM.

Example 6.1.2. *The exit condition for the initial hierarchy state in Figure 6.2b is $(\neg \blacktriangleright \wedge \blacktriangleright) \vee \blacktriangleright$.*

We now have everything required to define the *hierarchical transition function* δ_H of a hierarchy of reward machines H .

Definition 6.1.6 (Hierarchical transition function). *Given an HRM H , the hierarchical transition function δ_H maps a hierarchy state $\langle M_i, u, \Phi, \Gamma \rangle$ into another given a label \mathcal{L} . Formally,*

$$\delta_H(\langle M_i, u, \Phi, \Gamma \rangle, \mathcal{L}) = \begin{cases} \delta_H(\langle M_j, u', \top, \Gamma' \rangle, \perp) & \text{if } u \in \mathcal{U}_i^A, |\Gamma| > 0, \\ & \Gamma = \Gamma' \oplus \langle \cdot, u', M_j, M_i, \cdot, \cdot \rangle; \\ \delta_H(\langle M_j, u_j^0, \Phi', \Gamma \oplus \langle u, u', M_i, M_j, \phi, \Phi \rangle \rangle, \mathcal{L}) & \text{if } \mathcal{L} \models \xi_{j,u_j^0,\Phi'} \text{ where} \\ & \phi = \varphi_i(u, u', M_j)(\mathcal{L}), \\ & \Phi' = \text{DNF}(\Phi \wedge \phi); \\ \langle M_i, u, \Phi, \Gamma \rangle & \text{otherwise,} \end{cases}$$

where \perp denotes a label that cannot satisfy any formula, and \cdot denotes something unimportant for the case.

The hierarchical transition function δ_H considers three different cases, which we describe below in top-down order:

1. If u is an accepting state of M_i and the stack Γ is non-empty, pop the top element of Γ and return control to the previous RM, recursively applying δ_H in case several accepting states are reached simultaneously. The accumulated context of the resulting hierarchy state is \top since a transition in M_j has been taken; that is, the condition for starting M_j has been satisfied and does not apply anymore. In accordance with the previous remarks on the call stack, some of the stored elements are unimportant for determining the transitions.
2. If \mathcal{L} satisfies the context of a call and the exit condition from the initial state of the called RM, push the call onto the stack and recursively apply δ_H until M_\top is reached. Following the previous remarks on the call stack, ϕ is a DNF formula constituted by the satisfied disjuncts of $\varphi_i(u, u', M_j)$.
3. If none of the above holds, the hierarchy state remains unchanged.

The logical transition functions φ of the RMs must be such that δ_H is *deterministic*, i.e. a label cannot simultaneously satisfy the contexts and exit conditions associated with two triplets $\langle u, v, M_i \rangle$ and $\langle u, v', M_j \rangle$ such that either (i) $v = v'$ and $i \neq j$, or (ii) $v \neq v'$. Contexts help enforce determinism by making formulas mutually exclusive. Enforcing determinism through conditions on the calls (i.e., contexts) enables reusing existing RMs effectively; indeed, the alternative consists of making a copy of the called RMs and modifying the formulas on the edges from the initial state to ensure mutual exclusivity, which is inflexible and unscalable.

Example 6.1.3. *Given the HRM from Figure 6.2b, if the call to M_1 from the initial state of M_0 had context \top instead of $\neg\text{✗}$, then M_1 and M_2 could be both started if $\{\text{✚}, \text{✗}\}$ was observed, thus making the HRM non-deterministic.*

Example 6.1.4. *Given the HRM from Figure 6.2b, the context $\neg\text{✗}$ in the call to M_1 from the initial state of the root could be removed by (i) making a copy of M_1 called M'_1 , (ii) modifying the edge from u_1^0 to u_1^1 such that the formula is $\text{✚} \wedge \neg\text{✗}$, and (iii) changing the aforementioned call such that M'_1 is invoked instead. Despite preserving determinism, this approach does not reuse existing machines which, in the worst case, leads to an exponential increase in the number of machines in the hierarchy. From the learning point of view (Chapter 7), such an increase is undesirable, and hence, contexts constitute a more flexible option for enforcing determinism.*

Finally, akin to reward machines, we introduce *hierarchy traversals* to determine how HRMs process label traces. Based on the evaluation of a trace by an HRM, we later define what it means for an HRM to be valid with respect to a trace. The latter is essential for proving the correctness of our ASP encoding (Section 6.3) and learning the HRMs (Chapter 7).

Definition 6.1.7 (Hierarchy traversal). *Given a label trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$, a hierarchy traversal $H(\lambda) = \langle v_0, v_1, \dots, v_{n+1} \rangle$ is a unique sequence of hierarchy states such that (i) $v_0 = \langle M_r, u_r^0, \top, [] \rangle$, and (ii) $\delta_H(v_i, \mathcal{L}_i) = v_{i+1}$ for $i = 0, \dots, n$. An HRM H accepts λ if $v_{n+1} = \langle M_r, u, \top, [] \rangle$ and $u \in \mathcal{U}_r^A$.*

(i.e., an accepting state of the root is reached). Analogously, H rejects λ if $v_{n+1} = \langle M_k, u, \cdot, \cdot \rangle$ and $u \in \mathcal{U}_k^R$ for any $k \in \{0, \dots, m-1\}$ (i.e., a rejecting state in the HRM is reached, in accordance with Assumption 6.1.2).

Example 6.1.5. The HRM in Figure 6.2b accepts trace $\lambda = \langle \{\clubsuit\}, \{\heartsuit\}, \{\}, \{\spadesuit\}, \{\heartsuit\}, \{\clubsuit\} \rangle$, whose traversal is $H(\lambda) = \langle v_0, v_1, v_2, v_3, v_4, v_5, v_6 \rangle$, where:

$$\begin{aligned}
v_0 &= \langle M_0, u_0^0, \top, [] \rangle, \\
v_1 &= \delta_H(v_0, \{\clubsuit\}) \\
&= \delta_H(\langle M_0, u_0^0, \top, [] \rangle, \{\clubsuit\}) \\
&= \delta_H(\langle M_1, u_1^0, \neg\spadesuit, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle] \rangle, \{\clubsuit\}) \\
&= \delta_H(\langle M_\top, u_\top^0, \neg\spadesuit \wedge \clubsuit, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle, \langle u_1^0, u_1^1, M_1, M_\top, \clubsuit, \neg\spadesuit \rangle] \rangle, \{\clubsuit\}) \\
&= \delta_H(\langle M_1, u_1^1, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle] \rangle, \perp) \\
&= \langle M_1, u_1^1, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle] \rangle, \\
v_2 &= \delta_H(v_1, \{\heartsuit\}) \\
&= \delta_H(\langle M_1, u_1^1, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle] \rangle, \{\heartsuit\}) \\
&= \delta_H(\langle M_\top, u_\top^0, \heartsuit, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle, \langle u_1^1, u_1^A, M_1, M_\top, \heartsuit, \top \rangle] \rangle, \{\heartsuit\}) \\
&= \delta_H(\langle M_1, u_1^A, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\spadesuit, \top \rangle] \rangle, \perp) \\
&= \delta_H(\langle M_0, u_0^1, \top, [] \rangle, \perp) \\
&= \langle M_0, u_0^1, \top, [] \rangle, \\
v_3 &= \delta_H(v_2, \{\}) \\
&= \delta_H(\langle M_0, u_0^1, \top, [] \rangle, \{\}) \\
&= \langle M_0, u_0^1, \top, [] \rangle, \\
v_4 &= \delta_H(v_3, \{\spadesuit\}) \\
&= \delta_H(\langle M_0, u_0^1, \top, [] \rangle, \{\spadesuit\}) \\
&= \delta_H(\langle M_2, u_2^0, \top, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle] \rangle, \{\spadesuit\}) \\
&= \delta_H(\langle M_\top, u_\top^0, \spadesuit, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle, \langle u_2^0, u_2^1, M_2, M_\top, \spadesuit, \top \rangle] \rangle, \{\spadesuit\}) \\
&= \delta_H(\langle M_2, u_2^1, \top, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle] \rangle, \perp) \\
&= \langle M_2, u_2^1, \top, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle] \rangle, \\
v_5 &= \delta_H(v_4, \{\heartsuit\}) \\
&= \delta_H(\langle M_2, u_2^1, \top, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle] \rangle, \{\heartsuit\}) \\
&= \delta_H(\langle M_\top, u_\top^0, \heartsuit, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle, \langle u_2^1, u_2^A, M_2, M_\top, \heartsuit, \top \rangle] \rangle, \{\heartsuit\}) \\
&= \delta_H(\langle M_2, u_2^A, \top, [\langle u_0^1, u_0^3, M_0, M_2, \top, \top \rangle] \rangle, \perp) \\
&= \delta_H(\langle M_0, u_0^3, \top, [] \rangle, \perp) \\
&= \langle M_0, u_0^3, \top, [] \rangle, \\
v_6 &= \delta_H(v_5, \{\clubsuit\}) \\
&= \delta_H(\langle M_0, u_0^3, \top, [] \rangle, \{\clubsuit\}) \\
&= \delta_H(\langle M_\top, u_\top^0, \clubsuit, [\langle u_0^3, u_0^A, M_0, M_\top, \clubsuit, \top \rangle] \rangle, \{\clubsuit\})
\end{aligned}$$

$$\begin{aligned}
&= \delta_H(\langle M_0, u_0^A, \top, [] \rangle, \perp) \\
&= \langle M_0, u_0^A, \top, [] \rangle.
\end{aligned}$$

Definition 6.1.8 (Validity). *Given a trace λ^* , where $*$ $\in \{G, D, I\}$, an HRM H is valid with respect to λ^* if one of the following holds:*

- *H accepts λ^* and $*$ = G (i.e., λ^* is a goal trace).*
- *H rejects λ^* and $*$ = D (i.e., λ^* is a dead-end trace).*
- *H does not accept nor reject λ^* and $*$ = I (i.e., λ^* is an incomplete trace).*

6.2 Properties

In this section, we state and prove two properties on the *equivalence* between HRMs and the RMs introduced in Chapter 3. Indeed, the hierarchy formalism described in Section 6.1 is conceived to preserve the equivalence with RMs through the behavior of the hierarchical transition function. Before posing the properties, we formally define what *flat* RMs and HRMs are.

Definition 6.2.1 (Flat RM). *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, a constituent RM $M_i \in \mathcal{M}$ is flat if its height h_i is 1; that is, M_i only calls the leaf M_\top .*

Definition 6.2.2 (Flat HRM). *An HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ is flat if the root RM M_r is flat.*

In conformity with these definitions, a flat HRM is effectively one of the RMs from Chapter 3 since the root RM of a flat HRM only performs calls to the leaf; for instance, Figure 6.2a is a flat HRM for BOOK. Hence, in what follows, we focus on discussing how the behavior of an arbitrary HRM can be reproduced by an equivalent flat HRM. We now define what it means for two HRMs to be equivalent based on a similar definition from the automaton theory literature (Sipser, 1997).

Definition 6.2.3 (HRM equivalence). *Given a set of propositions \mathcal{P} and a labeling function l , two HRMs $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ and $H' = \langle \mathcal{M}', M'_r, \mathcal{P} \rangle$ are equivalent if for any label trace λ one of the following conditions holds: (i) both HRMs accept λ , (ii) both HRMs reject λ , or (iii) neither of the HRMs accepts or rejects λ .*

The equivalence properties between an arbitrary HRM and a flat HRM are stated below. First, we formally show that any HRM can be transformed into an equivalent flat one, which we prove by construction in Section 6.2.1.

Theorem 6.2.1. *Given an HRM H , there exists an equivalent flat HRM \bar{H} .*

Given the construction used in Theorem 6.2.1, we show that the number of states and edges of the resulting flat HRM can be *exponential* in the height of the root (see Theorem 6.2.2). We prove this in Section 6.2.2 through an instance of a general HRM parametrization where the constituent RMs are *highly reused*, hence illustrating the convenience of HRMs to compose existing knowledge succinctly. In line with the theory, learning a non-flat HRM can take a few seconds, whereas learning an equivalent flat HRM is often unfeasible (see Chapter 8).

Theorem 6.2.2. *Let $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ be an HRM and h_r be the height of its root M_r . The number of states and edges in an equivalent flat HRM \bar{H} can be exponential in h_r .*

6.2.1 Proof of Theorem 6.2.1

To prove the theorem, we introduce an algorithm for flattening any HRM. Without loss of generality, we work on the case of an HRM with two hierarchy levels; that is, an HRM consisting of a root RM that calls flat RMs. An HRM with an arbitrary number of levels can be flattened by considering the RMs in two levels at a time. We start flattening RMs in the second level (i.e., with height 2), which use RMs in the first level (by definition, these are already flat), and once the second level RMs are flat, we repeat the process with the levels above until the root is reached. This process is applicable since, by Assumption 6.1.1, the hierarchies do not have cyclic dependencies (including recursion). In line with Assumption 6.1.5, we assume that the reward-transition function of each resulting flat RM M_i is $r_i(u, u') = \mathbb{1}[u \notin \mathcal{U}_i^A \wedge u' \in \mathcal{U}_i^A]$; however, the proof could be adapted to arbitrary definitions of r_i .

Preliminary Transformation Algorithm

Before proving Theorem 6.2.1, we introduce an intermediate step that transforms a flat HRM into an equivalent one that takes contexts with which it may be called into account. Remember that a call to an RM is associated with a context. In the case of two-level HRMs, such as the ones we are considering in this flattening process, the context and the exit condition from the initial state of the called flat RM must be satisfied. Crucially, the context must only be satisfied at the time of the call; that is, it only lasts for a single transition. Therefore, if we revisit the initial state of the called RM by taking an edge to it, the context should not be checked anymore.

To make the need for this transformation clearer, we exemplify it through the HRM in Figure 6.3a. The flattening algorithm described later embeds the called RM into the calling one; crucially, the context of the call is taken into account by putting it in conjunction with the outgoing edges from the initial state of the called RM.¹ Figure 6.3b is a flat HRM obtained using the flattening algorithm; however, it does not behave like the HRM in Figure 6.3a. Following the definition of the hierarchical transition function δ_H , the context of a call only lasts for a single transition in the called RM in Figure 6.3a (i.e., $a \wedge \neg c$ is only checked when M_1 is started), but the context is kept permanently in Figure 6.3b, which is problematic if the initial state is revisited. We later come back to this example after presenting the transformation algorithm.

To deal with the situation above, we need to transform an RM to ensure that contexts are only checked once from the initial state. We describe this transformation as follows. Given a flat HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ with root $M_r = \langle \mathcal{U}_r, \mathcal{P}, \varphi_r, r_r, u_r^0, \mathcal{U}_r^A, \mathcal{U}_r^R \rangle$, we construct a new HRM $H' = \langle \mathcal{M}', M'_r, \mathcal{P} \rangle$ with root $M'_r = \langle \mathcal{U}'_r, \mathcal{P}, \varphi'_r, r'_r, u_r^0, \mathcal{U}_r^A, \mathcal{U}_r^R \rangle$ such that:

- $\mathcal{U}'_r = \mathcal{U}_r \cup \{\hat{u}_r^0\}$, where \hat{u}_r^0 plays the role of the initial state after the first transition is taken.
- The state transition function φ'_r is built by copying φ_r and applying the following changes:

1. Remove the edges to the actual initial state from any state $v \in \mathcal{U}'_r$: $\varphi'_r(v, u_r^0, M_\top) = \perp$.

Note that since the RM is flat, the only callable RM is the leaf M_\top .

¹We refer the reader to the forthcoming *Flattening Algorithm* subsection for specific details.

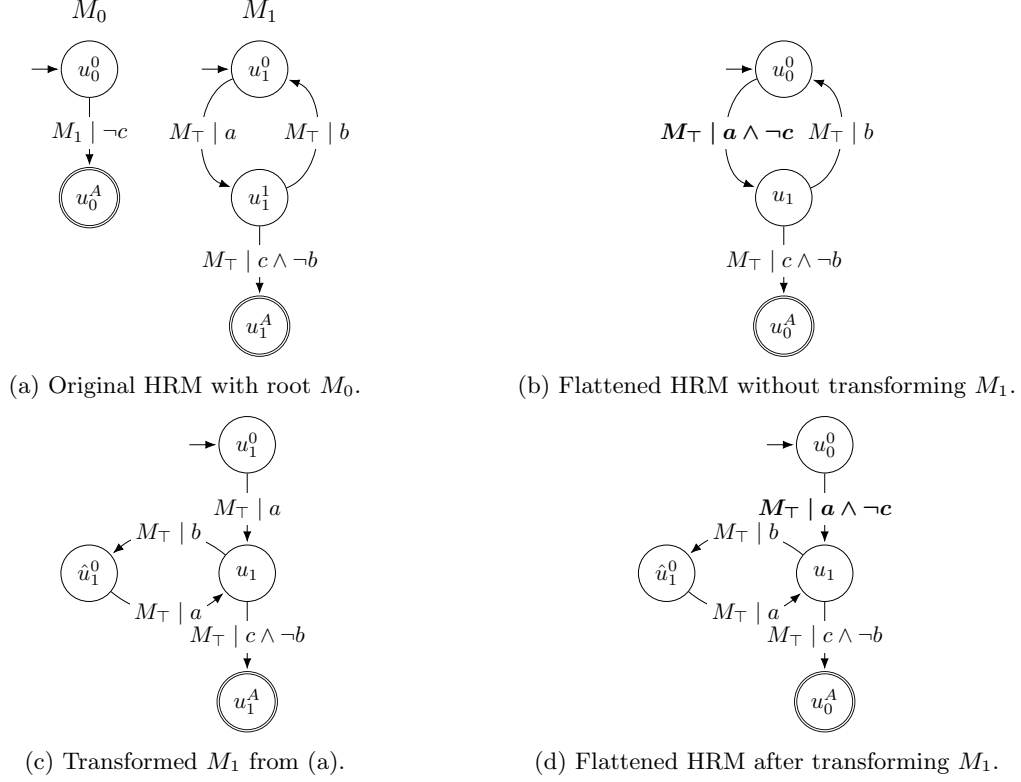


Figure 6.3: Example to justify the need for the preliminary transformation algorithm.

2. Add edges to the dummy initial state \hat{u}_r^0 from all states $v \in \mathcal{U}'_r$ that had an edge to the actual initial state: $\varphi'_r(v, \hat{u}_r^0, M_\top) = \varphi_r(v, u_r^0, M_\top)$.
 3. Add edges from the dummy initial state \hat{u}_r^0 to all those states $v \in \mathcal{U}'_r$ that the actual initial state u_r^0 points to: $\varphi'_r(\hat{u}_r^0, v, M_\top) = \varphi_r(u_r^0, v, M_\top)$.
- The reward transition function $r'_r(u, u') = \mathbb{1}[u \notin \mathcal{U}'_r \wedge u' \in \mathcal{U}'_r]$ is defined as stated at the beginning of the section.

The HRM H' is such that $\mathcal{M}' = \{M'_r, M_\top\}$. Note that this transformation is only required in HRMs where the RMs have initial states with incoming edges.

We now prove that this transformation is correct; that is, the HRMs are equivalent. There are two cases depending on whether the initial state has incoming edges or not. First, if the initial state u_r^0 does not have incoming edges, step 1 does not remove any edges going to u_r^0 , and step 2 does not add any edges going to \hat{u}_r^0 , making it unreachable. Even though edges from \hat{u}_r^0 to other states may be added, it is irrelevant since it is unreachable. Therefore, we can safely say that in this case, the transformed HRM is equivalent to the original one. Second, if the initial state has incoming edges, we prove equivalence by examining the traversals $H(\lambda)$ and $H'(\lambda)$ for the original HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ and the transformed one $H' = \langle \mathcal{M}', M'_r, \mathcal{P} \rangle$ given a generic label trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$. By construction, both $H(\lambda)$ and $H'(\lambda)$ will be identical until reaching a state w with an outgoing transition to u_r^0 in the case of H and the dummy initial state \hat{u}_r^0 in the case of H' . More specifically, upon reaching w and satisfying an outgoing formula to the aforementioned states,

the traversals are:

$$\begin{aligned} H(\lambda) &= \langle \langle M_r, u_r^0, \top, [] \rangle, \dots, \langle M_r, w, \top, [] \rangle \rangle, \\ H'(\lambda) &= \langle \langle M'_r, u_r^0, \top, [] \rangle, \dots, \langle M'_r, w, \top, [] \rangle \rangle. \end{aligned}$$

By construction, state w is in both HRMs, and both of the aforementioned transitions from this state are associated with the same formula, i.e. $\varphi_r(w, u_r^0, M_\top) = \varphi'_r(w, \hat{u}_r^0, M_\top)$. Therefore, if one of them is satisfied, the other will be too, and the traversals will become:

$$\begin{aligned} H(\lambda) &= \langle \langle M_r, u_r^0, \top, [] \rangle, \dots, \langle M_r, w, \top, [] \rangle, \langle M_r, u_r^0, \top, [] \rangle \rangle, \\ H'(\lambda) &= \langle \langle M'_r, u_r^0, \top, [] \rangle, \dots, \langle M'_r, w, \top, [] \rangle, \langle M'_r, \hat{u}_r^0, \top, [] \rangle \rangle. \end{aligned}$$

We stay in u_r^0 and \hat{u}_r^0 until a transition to a state w' is satisfied. By construction, w' is in both HRMs and the same formula is satisfied, i.e., $\varphi_r(u_r^0, w', M_\top) = \varphi'_r(\hat{u}_r^0, w', M_\top)$. The hierarchy traversals then become:

$$\begin{aligned} H(\lambda) &= \langle \langle M_r, u_r^0, \top, [] \rangle, \dots, \langle M_r, w, \top, [] \rangle, \langle M_r, u_r^0, \top, [] \rangle, \dots, \langle M_r, u_r^0, \top, [] \rangle, \langle M_r, w', \top, [] \rangle \rangle, \\ H'(\lambda) &= \langle \langle M'_r, u_r^0, \top, [] \rangle, \dots, \langle M'_r, w, \top, [] \rangle, \langle M'_r, \hat{u}_r^0, \top, [] \rangle, \dots, \langle M'_r, \hat{u}_r^0, \top, [] \rangle, \langle M'_r, w', \top, [] \rangle \rangle. \end{aligned}$$

From here, both traversals will be the same until transitions to u_r^0 and \hat{u}_r^0 are respectively satisfied again (if any) in H and H' . Clearly, the only change in $H(\lambda)$ with respect to $H'(\lambda)$ (except for the different roots) is that the hierarchy states of the form $\langle M'_r, \hat{u}_r^0, \top, [] \rangle$ in the latter appear as $\langle M_r, u_r^0, \top, [] \rangle$ in the former. We now check if the equivalence conditions from Definition 6.2.3 hold:

- If $H(\lambda)$ ends with state u_r^0 , $H'(\lambda)$ ends with state \hat{u}_r^0 following the reasoning above. By construction, neither of these states is accepting or rejecting; therefore, neither of these HRMs accepts or rejects λ .
- If $H(\lambda)$ ends with state w , $H'(\lambda)$ will also end with this state following the reasoning above. Therefore, if w is an accepting state, both HRMs accept λ ; if w is a rejecting state, both HRMs reject λ ; and if w is not an accepting or rejecting state, neither of the HRMs accepts or rejects λ .

Since all equivalence conditions are satisfied for any trace λ , H and H' are equivalent.

Figure 6.3c exemplifies the output of the transformation algorithm given M_1 in Figure 6.3a as input, whereas Figure 6.3d is the output of the flattening algorithm discussed next, which correctly handles the context unlike the HRM in Figure 6.3b.

Flattening Algorithm

We describe the algorithm for flattening an HRM. As previously stated, we assume without loss of generality that the HRM to be flattened consists of two hierarchy levels (i.e., the root calls flat RMs). We also assume that the flat RMs have the form produced by the previously presented transformation algorithm.

Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ with root $M_r = \langle \mathcal{U}_r, \mathcal{P}, \varphi_r, r_r, u_r^0, \mathcal{U}_r^A, \mathcal{U}_r^R \rangle$, we build a flat RM $\bar{M}_r = \langle \bar{\mathcal{U}}_r, \bar{\mathcal{P}}, \bar{\varphi}_r, \bar{r}_r, \bar{u}_r^0, \bar{\mathcal{U}}_r^A, \bar{\mathcal{U}}_r^R \rangle$ using the following steps:

1. Copy the sets of states and initial state from M_r (i.e., $\bar{\mathcal{U}}_r = \mathcal{U}_r, \bar{u}_r^0 = u_r^0, \bar{\mathcal{U}}_r^A = \mathcal{U}_r^A, \bar{\mathcal{U}}_r^R = \mathcal{U}_r^R$).
2. Loop through the non-false entries of the transition function φ_r and decide what to copy. That is, for each triplet $\langle u, u', M_j \rangle$ where $u, u' \in \mathcal{U}_r$ and $M_j \in \mathcal{M}$ such that $\varphi_r(u, u', M_j) \neq \perp$:
 - (a) If $M_j = M_\top$ (i.e., the called RM is the leaf), we copy the transition: $\bar{\varphi}_r(u, u', M_\top) = \varphi_r(u, u', M_\top)$.
 - (b) If $M_j \neq M_\top$, we embed the transition function of $M_j = \langle \mathcal{U}_j, \mathcal{P}, \varphi_j, r_j, u_j^0, \mathcal{U}_j^A, \mathcal{U}_j^R \rangle$ into \bar{M}_r . Remember that M_j is flat. To do so, we run the following steps:
 - i. Update the set of states by adding all non-initial and non-accepting states from M_j . Similarly, the set of rejecting states is also updated by adding all rejecting states of the called RM. The initial and accepting states from M_j are unimportant: their roles are played by u and u' respectively. In contrast, the rejecting states are important since, by Assumption 6.1.2, they are global. The added states v are renamed to $v_{u,u',j}$ in order to take into account the edge being embedded: if the same state v was reused for another edge, then we would not be able to distinguish them.

$$\begin{aligned}\bar{\mathcal{U}}_r &= \bar{\mathcal{U}}_r \cup \{v_{u,u',j} \mid v \in (\mathcal{U}_j \setminus (\{u_j^0\} \cup \mathcal{U}_j^A))\}, \\ \bar{\mathcal{U}}_r^R &= \bar{\mathcal{U}}_r^R \cup \{v_{u,u',j} \mid v \in \mathcal{U}_j^R\}.\end{aligned}$$

- ii. Embed the transition function φ_j of M_j into $\bar{\varphi}_r$. Since M_j is flat, we can make copies of the transitions straightaway: the only important thing is to check whether these transitions involve initial or accepting states which, as stated before, are going to be replaced by u and u' accordingly. Given a triplet $\langle v, w, M_\top \rangle$ such that $v, w \in \mathcal{U}_j$ and for which $\varphi_j(v, w, M_\top) = \phi$ and $\phi \neq \perp$ we update $\bar{\varphi}_r$ as follows:²
 - A. If $v = u_j^0$ and $w \notin \mathcal{U}_j^A$, then $\bar{\varphi}_r(u, w_{u,u',j}, M_\top) = \text{DNF}(\phi \wedge \varphi_r(u, u', M_j))$. The initial state of M_j has been substituted by u , we use the clone of w associated with the call $(w_{u,u',j})$, and append the context of the call to M_j to the formula ϕ .
 - B. If $v = u_j^0$ and $w \in \mathcal{U}_j^A$, then $\bar{\varphi}_r(u, u', M_\top) = \text{DNF}(\phi \wedge \varphi_r(u, u', M_j))$. Like the previous case but performing two substitutions: u replaces v and u' replaces w . The context is appended since it is a transition from the initial state of M_j .
 - C. If $v \neq u_j^0$ and $w \in \mathcal{U}_j^A$, then $\bar{\varphi}_r(v_{u,u',j}, u', M_\top) = \phi$. We substitute the accepting state w by u' , and use the clone of v associated with the call $(v_{u,u',j})$. This time, the call's context is not added since v is not the initial state of M_j .
 - D. If none of the previous cases holds, there are no substitutions to be made nor contexts to be taken into account. Hence, $\bar{\varphi}_r(v_{u,u',j}, w_{u,u',j}, M_\top) = \phi$. We just use the clones of v and w corresponding to the call $(v_{u,u',j}$ and $w_{u,u',j})$.

3. Form a new flat HRM $\bar{H} = \langle \{\bar{M}_r, M_\top\}, \bar{M}_r, \mathcal{P} \rangle$ with the flattened root \bar{M}_r .

²We do not cover the case where v is an accepting state since, by Assumption 6.1.4, there are no outgoing transitions from it. In the case of rejecting states, we keep all of them as explained in the previous case and, therefore, there are no substitutions to be made. We also do not cover the case where $w = u_j^0$ since the input flat machines never have edges to their initial states, but to the dummy initial state.

The reward transition function $r'_r(u, u') = \mathbb{1}[u \notin \bar{\mathcal{U}}_r^A \wedge u' \in \bar{\mathcal{U}}_r^A]$ is defined as per Assumption 6.1.5. If the resulting flattened root is called by an RM M with a higher height, the previous transformation algorithm is applied to it before flattening M .

Given the flattening algorithm introduced above, we restate the theorem and prove it in the following paragraphs.

Theorem 6.2.1. *Given an HRM H , there exists an equivalent flat HRM \bar{H} .*

Proof. Let us assume that an HRM $\bar{H} = \langle \bar{\mathcal{M}}, \bar{M}_r, \mathcal{P} \rangle$, where $\bar{M}_r = \langle \bar{\mathcal{U}}_r, \mathcal{P}, \bar{\varphi}_r, \bar{r}_r, \bar{u}_r^0, \bar{\mathcal{U}}_r^A, \bar{\mathcal{U}}_r^R \rangle$, is a flat HRM that results from applying the flattening algorithm on an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, where $M_r = \langle \mathcal{U}_r, \mathcal{P}, \varphi_r, r_r, u_r^0, \mathcal{U}_r^A, \mathcal{U}_r^R \rangle$. For these HRMs to be equivalent, any label trace $\lambda = \langle \mathcal{L}_0, \dots, \mathcal{L}_n \rangle$ must satisfy one of the conditions in Definition 6.2.3. To prove the equivalence, we examine the hierarchy traversals $H(\lambda)$ and $\bar{H}(\lambda)$ given a generic label trace λ .

Let $u \in \mathcal{U}_r$ be a state in the root M_r of H and let $\varphi_r(u, u', M_\top)$ be a satisfied transition from that state. By construction, u is also in the root \bar{M}_r of the flat hierarchy \bar{H} , and \bar{M}_r has an identical transition $\bar{\varphi}_r(u, u', M_\top)$, which must also be satisfied. If the hierarchy states are $\langle M_r, u, \top, [] \rangle$ and $\langle \bar{M}_r, u, \top, [] \rangle$ for H and \bar{H} respectively, then the next hierarchy states upon application of δ_H will be $\langle M_r, u', \top, [] \rangle$ and $\langle \bar{M}_r, u', \top, [] \rangle$. Therefore, both HRMs behave equivalently when calls to the leaf RM are made.

We now examine what occurs when a non-leaf RM is called in H . Let $\varphi_r(u, u', M_j)$ be a satisfied transition in M_r , and let $\varphi_j(u_j^0, w, M_\top)$ be a satisfied transition from M_j 's initial state. By construction, \bar{M}_r contains a transition whose associated formula is the conjunction of the previous two, i.e. $\varphi_r(u, u', M_j) \wedge \varphi_j(u_j^0, w, M_\top)$. Now, the hierarchy traversals will be different depending on w :

- If $w \notin \mathcal{U}_j^A$ (i.e., w is not an accepting state of M_j), by construction, \bar{M}_r contains the transition $\bar{\varphi}_r(u, w_{u,u',j}, M_\top) = \varphi_r(u, u', M_j) \wedge \varphi_j(u_j^0, w, M_\top)$. If the current hierarchy states are (the equivalent) $\langle M_r, u, \top, [] \rangle$ and $\langle \bar{M}_r, u, \top, [] \rangle$ for H and \bar{H} , then the next hierarchy states upon application of δ_H are $\langle M_j, w, \top, [\langle u, u', M_r, M_j, \varphi_r(u, u', M_j), \top \rangle] \rangle$ and $\langle \bar{M}_r, w_{u,u',j}, \top, [] \rangle$. These hierarchy states are equivalent since $w_{u,u',j}$ is a clone of w that saves all the call information (i.e., a call to machine M_j for transitioning from u to u').
- If $w \in \mathcal{U}_j^A$ (i.e., w is an accepting state of M_j), by construction, \bar{M}_r contains the transition $\bar{\varphi}_r(u, u', M_\top) = \varphi_r(u, u', M_j) \wedge \varphi_j(u_j^0, w, M_\top)$. If the current hierarchy states are (the equivalent) $\langle M_r, u, \top, [] \rangle$ and $\langle \bar{M}_r, u, \top, [] \rangle$ for H and \bar{H} , then the next hierarchy states upon application of δ_H are $\langle M_r, u', \top, [] \rangle$ and $\langle \bar{M}_r, u', \top, [] \rangle$. These hierarchy states are equivalent since the machine states are identical.

We now check the case in which we are inside a called RM. Let $\varphi_r(u, u', M_j)$ be the transition that caused H to start running M_j , and let $\varphi_j(v, w, M_\top)$ be a satisfied transition within M_j such that $v \neq u_j^0$. By construction, \bar{M}_r contains a transition associated with the same formula $\varphi_j(v, w, M_\top)$. The hierarchy traversals vary depending on w :

- If $w \notin \mathcal{U}_j^A$ (i.e., w is not an accepting state of M_j), by construction, \bar{M}_r contains the transition $\bar{\varphi}_r(v_{u,u',j}, w_{u,u',j}, M_\top) = \varphi_j(v, w, M_\top)$. For the transition to be taken in H , the hierarchy state must be $\langle M_j, v, \top, [\langle u, u', M_r, M_j, \varphi_r(u, u', M_j), \top \rangle] \rangle$, whereas in \bar{H} it will be $\langle \bar{M}_r, v_{u,u',j}, \top, [] \rangle$. These hierarchy states are clearly equivalent: $v_{u,u',j}$ is a clone of v that

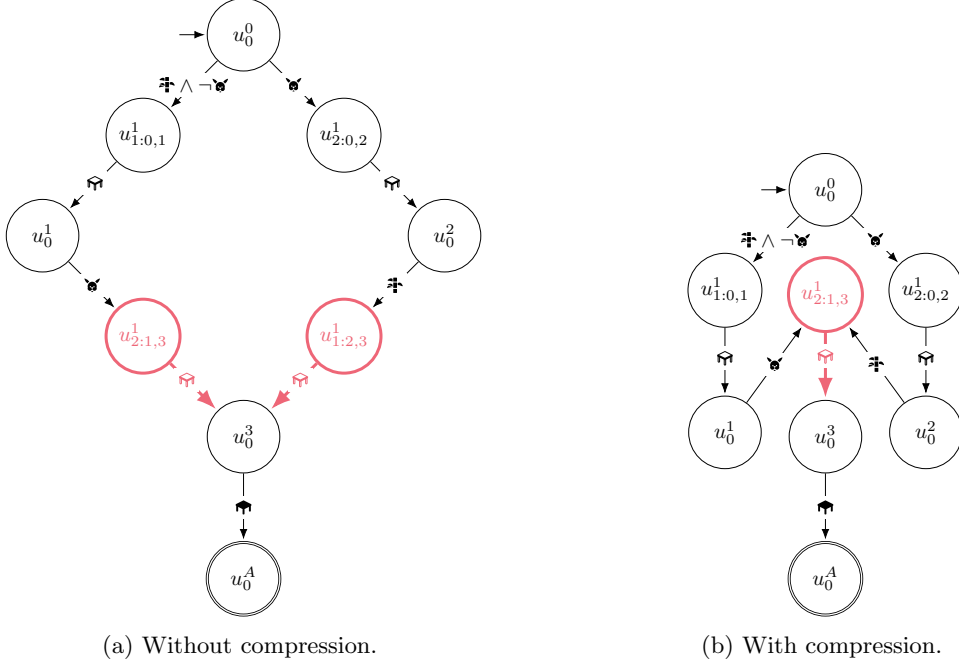


Figure 6.4: Results of flattening the HRM in Figure 6.2b, where $u_{j:v,w}^i$ denotes the i -th state of RM j in the call between states v and w in the parent RM. Note that v and w appear only if the state is from a called RM. The highlighted states and edges in (a) could be compressed as shown in (b).

saves all information related to the call being made (the called machine, and the starting and resulting states in the transition). Upon application of δ_H , the hierarchy states will remain equivalent: $\langle M_j, w, \top, [\langle u, u', M_r, M_j, \varphi_r(u, u', M_j), \top \rangle] \rangle$ and $\langle \bar{M}_r, w_{u,u',j}, \top, [] \rangle$ (again $w_{u,u',j}$ saves all the call information, just like the stack).

- If $w \in \mathcal{U}_j^A$ (i.e., w is an accepting state of M_j), by construction, \bar{M}_r contains the transition $\bar{\varphi}_r(v_{u,u',j}, u', M_\top) = \varphi_j(v, w, M_\top)$. This case corresponds to that where control is returned to the calling RM. Like in the previous case, for the transition to be taken in H , the hierarchy state must be $\langle M_j, v, \top, [\langle u, u', M_r, M_j, \varphi_r(u, u', M_j), \top \rangle] \rangle$, whereas in \bar{H} it will be $\langle \bar{M}_r, v_{u,u',j}, \top, [] \rangle$. The resulting hierarchy states then become $\langle M_r, u', \top, [] \rangle$ and $\langle \bar{M}_r, u', \top, [] \rangle$ respectively, which are clearly equivalent (the state is exactly the same and both come from equivalent hierarchy states).

We have shown that both HRMs have equivalent traversals for any given trace, implying that both will accept, reject, or not accept nor reject a trace. Therefore, the HRMs are equivalent. \square

Figure 6.4a shows the result of applying the flattening algorithm on the BOOK HRM shown in Figure 6.2b; indeed, the resulting flat HRM is like the RM in Figure 6.2a but naming the states according to the algorithm. We emphasize that the presented algorithm is not guaranteed to produce a smallest possible flat equivalent. For instance, the previous flat HRM has two states with an outgoing edge labeled by \curvearrowright to u_0^3 ; therefore, the flat HRM can be compressed by merging the aforementioned states, producing the HRM shown in Figure 6.4b. Estimating how much a flat HRM (or any HRM) can be compressed and designing an algorithm to perform such compression are left as future work.

6.2.2 Proof of Theorem 6.2.2

We prove the theorem by first characterizing an HRM H using a set of abstract parameters. Then, we describe how the number of states and edges in an HRM and its corresponding flat equivalent are computed, and use these quantities to give an example for which the theorem holds. The parameters are the following:

- The height of the root h_r .
- The number of RMs with height i , $N^{(i)}$.
- The number of states in an RM with height i , $U^{(i)}$.
- The number of edges from each state in an RM with height i , $E^{(i)}$.

Note that $N^{(h_r)} = 1$ since, by definition, there is a single root. In addition, we make the following assumptions on the structure of the hierarchy.

Assumption 6.2.1. *The RMs with height i only call RMs with height $i - 1$.*

Assumption 6.2.2. *All RMs have a single accepting state and no rejecting states.*

Assumption 6.2.3. *All RMs except for the root are called.*

Assumption 6.2.4. *The HRM is well-formed (i.e., it behaves deterministically and there are no cyclic dependencies).*

Assumption 6.2.1 can be made since for the root to have height h_r we need it to call at least one RM with height $h_r - 1$. Considering that all called RMs have the same height simplifies the analysis since we can characterize the RMs at each height independently. Assumption 6.2.2 is safe to make since a single accepting state is enough, and helps simplify the counting since only some RMs could have rejecting states. Assumption 6.2.3 ensures that the flat HRM will comprise all RMs in the original HRM. This is also a fair assumption: if a given RM is not called by any RM in the hierarchy, we could remove it beforehand.

The number of states $|H|$ in the HRM H is obtained by summing the number of states of each RM:

$$|H| = \sum_{i=1}^{h_r} N^{(i)} U^{(i)}.$$

The number of states $|\bar{H}|$ in the flat HRM \bar{H} is given by the number of states in the flattened root RM

$$|\bar{H}| = \bar{U}^{(h_r)},$$

where $\bar{U}^{(i)}$ is the number of states in the flattened representation of an RM with height i , which is recursively defined as:

$$\bar{U}^{(i)} = \begin{cases} U^{(i)} & \text{if } i = 1; \\ U^{(i)} + (\bar{U}^{(i-1)} - 2) (U^{(i)} - 1) E^{(i)} & \text{if } i > 1. \end{cases}$$

That is, the number of states in a flattened RM with height i has all states that the non-flat HRM had. In addition, for each of the $U^{(i)} - 1$ non-accepting states in the non-flat RM, there are $E^{(i)}$

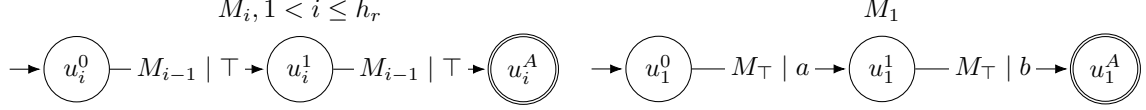


Figure 6.5: Example of an HRM, whose root has height h_r , used in the proof of Theorem 6.2.2.

edges, each of which calls an RM with height $i - 1$ whose number of states is $\bar{U}^{(i-1)}$. These edges are replaced by the called RM except for the initial and accepting states, whose role is now played by the states involved in the substituted edge (hence the -2). This construction process corresponds to the one used to prove Theorem 6.2.1.

The *total of number of edges* in an HRM is given by:

$$\sum_{i=1}^{h_r} N^{(i)} (U^{(i)} - 1) E^{(i)},$$

where $(U^{(i)} - 1)E^{(i)}$ is the total number of edges in an RM with height i (the -1 is because the accepting state is discarded), so $N^{(i)}(U^{(i)} - 1)E^{(i)}$ determines how many edges there are across RMs with height i .

The *total number of edges in the flat HRM* is given by the total number of edges in the flattened root RM, $\bar{E}^{(h_r)}$, where $\bar{E}^{(i)}$ is the total number of edges in the flattened representation of an RM with height i , which is recursively defined as follows:

$$\bar{E}^{(i)} = \begin{cases} (U^{(i)} - 1)E^{(i)} & \text{if } i = 1; \\ (U^{(i)} - 1)E^{(i)}\bar{E}^{(i-1)} & \text{if } i > 1. \end{cases}$$

That is, each of the $(U^{(i)} - 1)E^{(i)}$ edges in an RM with height i is replaced by $\bar{E}^{(i-1)}$ edges given by an RM with height $i - 1$ (if any).

The *key intuition* is that an HRM with root height $h_r > 1$ is beneficial representation-wise if the number of calls across RMs with height i is higher than the number of RMs with height $i - 1$; in other words, RMs with lower heights are being reused. Numerically, the total number of edges/calls in an RM with height i is $(U^{(i)} - 1)E^{(i)}$ and, therefore, the total number of calls across RMs with height i is $(U^{(i)} - 1)E^{(i)}N^{(i)}$. If this quantity is higher than $N^{(i-1)}$, then RMs with lower heights are reused, and therefore having RMs with different heights is beneficial.

Having characterized hierarchies of reward machines through abstract parameters, we now restate the theorem and prove it with an example.

Theorem 6.2.2. *Let $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ be an HRM and h_r be the height of its root M_r . The number of states and edges in an equivalent flat HRM \bar{H} can be exponential in h_r .*

Proof. By example. Let $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ be an HRM whose root M_r has height h_r and is parameterized by $N^{(i)} = 1$, $U^{(i)} = 3$, $E^{(i)} = 1$ for $i = 1, \dots, h_r$. Figure 6.5 shows an instance of this hierarchy. Let us write the number of states in the flat RMs of each level:

$$\begin{aligned} \bar{U}^{(1)} &= U^{(1)} = 3, \\ \bar{U}^{(2)} &= U^{(2)} + (\bar{U}^{(1)} - 2)(U^{(2)} - 1)E^{(2)} = 3 + (3 - 2)(3 - 1)1 = 5, \end{aligned}$$

$$\begin{aligned}
\bar{U}^{(3)} &= U^{(3)} + (\bar{U}^{(2)} - 2) (U^{(3)} - 1) E^{(3)} = 3 + (5 - 2) (3 - 1) 1 = 9, \\
&\vdots \\
\bar{U}^{(i)} &= 2\bar{U}^{(i-1)} - 1 = 2^i + 1.
\end{aligned}$$

Hence, the number of states in the flat HRM is $|\bar{H}| = \bar{U}^{(h_r)} = 2^{h_r} + 1$, which grows exponentially with the height of the root. In contrast, the number of states in the HRM grows linearly with the height of the root, $|H| = \sum_{i=1}^{h_r} N^{(i)} U^{(i)} = \sum_{i=1}^{h_r} 1 \cdot 3 = 3h_r$.

In the case of the total number of edges, we again write some iterations to derive a general expression:

$$\begin{aligned}
\bar{E}^{(1)} &= (U^{(1)} - 1) E^{(1)} = (3 - 1) 1 = 2, \\
\bar{E}^{(2)} &= (U^{(2)} - 1) E^{(2)} \bar{E}^{(1)} = (3 - 1) \cdot 1 \cdot 2 = 4, \\
\bar{E}^{(3)} &= (U^{(3)} - 1) E^{(3)} \bar{E}^{(2)} = (3 - 1) \cdot 1 \cdot 4 = 8, \\
&\vdots \\
\bar{E}^{(i)} &= 2\bar{E}^{(i-1)} = 2^i.
\end{aligned}$$

Therefore, the total number of edges in the flat HRM is $\bar{E}^{(h_r)} = 2^{h_r}$. In contrast, the total number of edges in the HRM grows linearly: $\sum_{i=1}^{h_r} N^{(i)} (U^{(i)} - 1) E^{(i)} = \sum_{i=1}^{h_r} 1(3 - 1) 1 = 2h_r$.

Finally, we emphasize that the resulting flat HRM cannot be compressed, unlike the HRM in Figure 6.4: each state has at most one incoming edge, so there are no parallel paths that can be merged. We have thus shown that there are HRMs for which the states and edges of an equivalent flat HRM grow exponentially with the height of the root. \square

Using the aforementioned intuition, we observe that the hierarchical structure is actually expected to be useful: the number of calls across RMs with height i is $(U^{(i)} - 1) E^{(i)} = (3 - 1) 1 = 2$, which is greater than the number of RMs with height $i - 1$ (only 1).

In some cases, having a multi-level hierarchy (i.e., with $h_r > 1$) is not beneficial. For instance, given an HRM whose root has height h_r and parameterized by $N^{(i)} = 1$, $U^{(i)} = 2$ and $E^{(i)} = 1$, the number of states in the equivalent flat HRM is constant (namely 2), whereas in the HRM itself it grows linearly with h_r . The same occurs with the number of edges. By checking the previously introduced intuition, we observe that $(U^{(i)} - 1) E^{(i)} N^{(i)} = (2 - 1) \cdot 1 \cdot 1 = 1 \not\geq N^{(i-1)} = 1$, which verifies that having non-reused RMs with multiple heights is not useful.

6.3 Representation in Answer Set Programming

In this section, we describe how HRMs are represented using answer set programming (ASP). First, we describe how traces (Section 6.3.1) and HRMs themselves are represented (Section 6.3.2). Next, we prove the correctness of the proposed representation (Section 6.3.3). Finally, we devise rules for enforcing determinism (Section 6.3.4) and whether the HRM complies with a canonical indexing of states and edges for breaking symmetries (Section 6.3.5).

6.3.1 Traces

The ASP representation of a trace does not change with respect to that introduced in Section 3.3.1. Indeed, the `prop`, `step`, and `last` predicates are also used in the HRM representation presented next.

6.3.2 Hierarchies of Reward Machines

We here describe the ASP representation for HRMs. To simplify the representation (and thus the learning of the HRMs in Chapter 7), we make the following assumption.

Assumption 6.3.1. *Given an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, the set of accepting states \mathcal{U}_i^A and the set of rejecting states \mathcal{U}_i^R of each constituent RM $M_i \in \mathcal{M}$ are singletons (i.e., consist of a single state).*

This assumption is made without loss of generality since accepting and rejecting states do not have transitions to other states by Assumption 6.1.4; therefore, any constituent RM M_i with multiple accepting and rejecting states can be mapped into an equivalent RM with a single accepting state u_i^A and a single rejecting state u_i^R . Formalizing RMs using sets of accepting and rejecting states helps simplify the state-transition function since it enables treating the leaf RM seamlessly (remember its initial state is an accepting state); however, as we will later see, the leaf RM is not explicitly represented here and, thus, we do not need to consider the aforementioned sets of states.

The rest of this section is organized as follows. First, we explain the particular rules that characterize a given HRM. Second, we introduce the rules representing how the traversal in any HRM is performed for any trace.

Structure

Akin to regular RMs (see Section 3.3.2), we introduce two representations of the structure of a specific HRM, each with a different purpose:

- A *non-factual* representation, where the logical transition function of each constituent RM is expressed in terms of rules (Definition 6.3.1), which is used by the HRM traversal rules introduced later. The logical transition function of a learned root RM (Chapter 7) is expressed using this representation.
- A *factual* representation, where the logical transition function of each constituent RM is expressed in terms of facts (Definition 6.3.2), which is used to verify whether the HRM (or individual constituent RMs) complies with certain structural properties such as determinism (Section 6.3.4).

In both cases, the predicates are similar to those employed in representing RMs in Chapter 3. We start defining and exemplifying the former representation, and continue with the latter.

Definition 6.3.1 (Non-factual ASP representation of a hierarchy of reward machines). *Given a hierarchy of reward machines $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$, $\mathbb{A}(H) = \bigcup_{M_i \in \mathcal{M} \setminus \{M_r\}} \mathbb{A}(M_i)$ denotes the set of ASP rules representing it, where each non-leaf RM M_i in the hierarchy is associated with its own set of rules $\mathbb{A}(M_i) = \mathbb{A}_{\mathcal{U}}(M_i) \cup \mathbb{A}_{\varphi}(M_i)$ such that:*

$$\mathbb{A}_{\mathcal{U}}(M_i) = \{\text{state}(u, M_i). \mid u \in \mathcal{U}_i\},$$

and

$$\mathbb{A}_\varphi(M_i) = \left\{ \begin{array}{l} \text{call}(u, u', x + y, M_i, M_j). \\ \bar{\varphi}(u, u', x + y, M_i, T) : \text{-not prop}(p_1, T), \text{step}(T). \\ \vdots \\ \bar{\varphi}(u, u', x + y, M_i, T) : \text{-not prop}(p_n, T), \text{step}(T). \\ \bar{\varphi}(u, u', x + y, M_i, T) : \text{-prop}(p_{n+1}, T), \text{step}(T). \\ \vdots \\ \bar{\varphi}(u, u', x + y, M_i, T) : \text{-prop}(p_m, T), \text{step}(T). \end{array} \middle| \begin{array}{l} M_j \in \mathcal{M}, u, u' \in \mathcal{U}_i, \\ \varphi_i(u, u', M_j) \neq \perp, \\ x = \sum_{k=0}^{j-1} |\varphi_i(u, u', M_k)|, \\ 1 \leq y \leq |\varphi_i(u, u', M_j)|, \\ \phi_y \in \varphi_i(u, u', M_j), \\ \phi_y = p_1 \wedge \dots \wedge p_n \\ \wedge \neg p_{n+1} \wedge \dots \wedge \neg p_m \end{array} \right\}.$$

The rule set $\mathbb{A}_\mathcal{U}(M_i)$ represents the set of states of reward machine M_i , whereas $\mathbb{A}_\varphi(M_i)$ represents the logical transition function of M_i . The constituent rules of these sets are described as follows:

- Facts $\text{state}(u, M_i)$ indicate that u is a state of RM M_i .
- Facts $\text{call}(u, u', e, M_i, M_j)$ indicate that edge e between states u and u' in RM M_i is labeled with a call to RM M_j .
- Normal rules whose *head* is of the form $\bar{\varphi}(u, u', e, M_i, T)$ indicate that the transition from state u to u' with edge e in RM M_i does not hold at step T . The *body* of these rules consists of a single $\text{prop}(p, T)$ literal and a $\text{step}(T)$ atom indicating that T is a step.

There are some important aspects to take into account regarding the encoding:

- The leaf RM M_\top is not represented; that is, even though it can be called, its state set and logic transition function are not encoded. We later introduce the ASP rules that represent what occurs when it is called.
- The edge identifiers e between a given pair of states $\langle u, u' \rangle$ range from 1 to the total number of disjuncts between them; that is, the identifiers are accumulated across calls to different RMs.
- Just like in the non-factual ASP representation of regular RMs (see Definition 3.3.2), $\bar{\varphi}$ represents the negation of the logical transition function φ . The underlying motivation remains: learning $\bar{\varphi}$ instead of φ makes the search space smaller, which speeds up HRM learning (Chapter 7).

Example 6.3.1. *The non-factual ASP representation of the HRM in Figure 6.2b is:*

$$\left\{ \begin{array}{l} \text{state}(u_0^0, M_0). \text{state}(u_1^0, M_0). \text{state}(u_2^0, M_0). \\ \text{call}(u_0^0, u_0^1, 1, M_0, M_1). \text{call}(u_0^0, u_0^2, 1, M_0, M_2). \\ \text{call}(u_0^3, u_0^A, 1, M_0, M_\top). \\ \bar{\varphi}(u_0^3, u_0^A, 1, M_0, T) : \text{-not prop}(\heartsuit, T), \text{step}(T). \end{array} \quad \begin{array}{l} \text{state}(u_0^3, M_0). \text{state}(u_0^A, M_0). \\ \text{call}(u_1^0, u_0^3, 1, M_0, M_2). \text{call}(u_0^2, u_0^3, 1, M_0, M_1). \\ \bar{\varphi}(u_0^0, u_0^1, 1, M_0, T) : \text{-prop}(\spadesuit, T), \text{step}(T). \end{array} \right\} \cup$$

$$\left\{ \begin{array}{l} \text{state}(u_1^0, M_1). \text{state}(u_1^1, M_1). \text{state}(u_1^A, M_1). \\ \bar{\varphi}(u_1^0, u_1^1, 1, M_1, T) : \text{-not prop}(\clubsuit, T), \text{step}(T). \\ \text{state}(u_2^0, M_2). \text{state}(u_2^1, M_2). \text{state}(u_2^A, M_2). \\ \bar{\varphi}(u_2^0, u_2^1, 1, M_2, T) : \text{-not prop}(\spadesuit, T), \text{step}(T). \end{array} \quad \begin{array}{l} \text{call}(u_1^0, u_1^1, 1, M_1, M_\top). \text{call}(u_1^1, u_1^A, 1, M_1, M_\top). \\ \bar{\varphi}(u_1^1, u_1^A, 1, M_1, T) : \text{-not prop}(\heartsuit, T), \text{step}(T). \\ \text{call}(u_2^0, u_2^1, 1, M_2, M_\top). \text{call}(u_2^1, u_2^A, 1, M_2, M_\top). \\ \bar{\varphi}(u_2^1, u_2^A, 1, M_2, T) : \text{-not prop}(\heartsuit, T), \text{step}(T). \end{array} \right\}.$$

As described in Section 3.3.2, non-factual representations represent the logical transition functions using rules, making the verification of structural properties (e.g., determinism) hard. In contrast, verifying properties through a factual representation is straightforward since there are no variable-dependent rules. We define the factual representation of an HRM and exemplify it below.

Definition 6.3.2 (Factual ASP representation of a hierarchy of reward machines). *Given the ASP representation $\mathbb{A}(H)$ of a hierarchy of reward machines H , the factual ASP representation $\mathbb{A}^F(H)$ of H is the result of mapping the $\bar{\varphi}$ rules into $\text{pos}(u, u', e, M, p)$ and $\text{neg}(u, u', e, M, p)$ facts expressing that proposition p appears positively (resp. negatively) in the edge e from state u to state u' of RM M . Formally,*

$$\mathbb{A}^F(H) = \left\{ \begin{array}{l|l} \text{state}(u, M). & \text{state}(u, M). \\ \text{call}(u, u', e, M, M'). & \text{call}(u, u', e, M, M'). \\ \text{pos}(u, u', e, M, p_1). & \bar{\varphi}(u, u', e, M, T) : - \text{not prop}(p_1, T), \text{step}(T). \\ \vdots & \vdots \\ \text{pos}(u, u', e, M, p_n). & \bar{\varphi}(u, u', e, M, T) : - \text{not prop}(p_n, T), \text{step}(T). \\ \text{neg}(u, u', e, M, p_{n+1}). & \bar{\varphi}(u, u', e, M, T) : - \text{prop}(p_{n+1}, T), \text{step}(T). \\ \vdots & \vdots \\ \text{neg}(u, u', e, M, p_m). & \bar{\varphi}(u, u', e, M, T) : - \text{prop}(p_m, T), \text{step}(T). \end{array} \right\},$$

where the rules on the right hand side are those within $\mathbb{A}(H)$.

Example 6.3.2. The following rules constitute the factual ASP representation built from the rule set in Example 6.3.1:

$$\left\{ \begin{array}{l} \text{state}(u_0^0, M_0). \text{state}(u_1^1, M_0). \text{state}(u_2^2, M_0). \quad \text{state}(u_0^3, M_0). \text{state}(u_0^A, M_0). \\ \text{call}(u_0^0, u_0^1, 1, M_0, M_1). \text{call}(u_0^0, u_0^2, 1, M_0, M_2). \quad \text{call}(u_0^1, u_0^3, 1, M_0, M_2). \text{call}(u_0^2, u_0^3, 1, M_0, M_1). \\ \text{call}(u_0^3, u_0^A, 1, M_0, M_\top). \quad \text{neg}(u_0^0, u_0^1, 1, M_0, \clubsuit). \\ \text{pos}(u_0^3, u_0^A, 1, M_0, \spadesuit). \end{array} \right\} \cup$$

$$\left\{ \begin{array}{l} \text{state}(u_1^0, M_1). \text{state}(u_1^1, M_1). \text{state}(u_1^A, M_1). \quad \text{call}(u_1^0, u_1^1, 1, M_1, M_\top). \text{call}(u_1^1, u_1^A, 1, M_1, M_\top). \\ \text{pos}(u_1^0, u_1^1, 1, M_1, \spadesuit). \quad \text{pos}(u_1^1, u_1^A, 1, M_1, \clubsuit). \\ \text{state}(u_2^0, M_2). \text{state}(u_2^1, M_2). \text{state}(u_2^A, M_2). \quad \text{call}(u_2^0, u_2^1, 1, M_2, M_\top). \text{call}(u_2^1, u_2^A, 1, M_2, M_\top). \\ \text{pos}(u_2^0, u_2^1, 1, M_2, \clubsuit). \quad \text{pos}(u_2^1, u_2^A, 1, M_2, \spadesuit). \end{array} \right\}.$$

Our approach for learning HRMs introduced in Chapter 7, akin to that in Chapter 4, learns a non-factual representation and maps it into a factual one on which structural properties are verified; thus, the set of rules for modeling hierarchy traversals are defined over the non-factual representation.

General Rules

The following sets of rules, whose union is denoted by $\mathcal{R} = \bigcup_{i=0}^5 \mathcal{R}_i$, encode HRM traversals and the acceptance/rejection criteria. For simplicity, the initial, accepting, and rejecting states of any RM M_i are respectively denoted by u^0 , u^A and u^R instead of u_i^0 , u_i^A and u_i^R since the RM they belong to is explicitly mentioned in each of the rules.

The rule set \mathcal{R}_0 contains a rule encoding the inversion of the negation of the logical transition function $\bar{\varphi}$. An analogous version of this rule is used in Section 3.3.2 for regular RMs. The φ atoms

include the called RM M_2 as an argument, which makes the following rules easier to express.

$$\mathcal{R}_0 = \left\{ \varphi(X, Y, E, M, M_2, T) : \neg \text{not } \bar{\varphi}(X, Y, E, M, T), \text{call}(X, Y, E, M, M_2), \text{step}(T). \right\}$$

The set of rules \mathcal{R}_1 introduces the $\text{pre_sat}(u, M, t)$ atoms, which encode the exit condition (see Definition 6.1.5) indicating whether a call from state u of RM M can be started at time t . The first rule corresponds to the base case: if the leaf M_\top is called, the condition is satisfied if the associated formula is satisfied. The second rule applies to calls to non-leaf RMs, where we need to satisfy the context of the call (like in the base case), and also check whether a call from the initial state of the potentially called RM can be started.

$$\mathcal{R}_1 = \left\{ \begin{array}{l} \text{pre_sat}(X, M, T) : \neg \varphi(X, \neg, \neg, M, M_\top, T). \\ \text{pre_sat}(X, M, T) : \neg \varphi(X, \neg, \neg, M, M_2, T), \text{pre_sat}(u^0, M_2, T), M_2! = M_\top. \end{array} \right\}$$

The rule set \mathcal{R}_2 introduces the $\text{reachable}(u, M, t, t')$ atoms, which indicate that state u of RM M is reached between steps t and t' . The latter step can also be seen as the step the agent is currently at. The first fact indicates that the initial state of the root RM is reached from step 0 to step 0. The second rule indicates that the initial state of a non-root RM is reached from step T to step T (i.e., it is reached anytime). The third rule represents the self-loop transition in the initial state of the root M_r : the agent stays there if no call can be started at T (i.e., the agent is not moving in the HRM). The fourth rule is analogous to the third but for the accepting state of the root instead of the initial state. Remember this is the only accepting state in the HRM that does not return control to a calling RM. The fifth rule is also similar to the previous ones: it applies to states reached after T_0 that are non-accepting, which excludes self-looping in initial states of non-root RMs at the time of starting them (i.e., self-loops are permitted in the initial state of a non-root RM if we can reach it afterward by going back to it). The last rule indicates that Y is reached at step T_2 in RM M started at T_0 if there is an outgoing transition from the current state X to Y at time T that holds between T and T_2 , and state X has been reached between T_0 and T . We will later see how δ is defined.

$$\mathcal{R}_2 = \left\{ \begin{array}{l} \text{reachable}(u^0, M_r, 0, 0). \\ \text{reachable}(u^0, M, T, T) : \neg \text{state}(u^0, M), M! = M_r, \text{step}(T). \\ \text{reachable}(X, M, T_0, T_0+1) : \neg \text{reachable}(X, M, T_0, T), \text{not pre_sat}(X, M, T), \\ \quad \text{step}(T), X = u^0, M = M_r. \\ \text{reachable}(X, M, T_0, T_0+1) : \neg \text{reachable}(X, M, T_0, T), \text{not pre_sat}(X, M, T), \\ \quad \text{step}(T), X = u^A, M = M_r. \\ \text{reachable}(X, M, T_0, T_0+1) : \neg \text{reachable}(X, M, T_0, T), \text{not pre_sat}(X, M, T), \\ \quad \text{step}(T), T_0 < T, X! = u^A. \\ \text{reachable}(Y, M, T_0, T_2) : \neg \text{reachable}(X, M, T_0, T), \delta(X, Y, M, T, T_2). \end{array} \right\}$$

The rule set \mathcal{R}_3 introduces two predicates: **satisfied** and **failed**. The $\text{satisfied}(M, t, t')$ atoms indicates that RM M is satisfied if its accepting state u^A is reached between steps t and t' . Likewise, the $\text{failed}(M, t, t')$ atoms indicate that RM M fails if its rejecting state u^R is reached between steps t and t' . These two descriptions correspond to the first and third rules. The second rule applies to the leaf RM M_\top , which always returns control immediately; thus, it is always satisfied

between any two consecutive steps.

$$\mathcal{R}_3 = \left\{ \begin{array}{l} \text{satisfied}(M, T0, TE) :- \text{reachable}(u^A, M, T0, TE). \\ \text{satisfied}(M_T, T, T+1) :- \text{step}(T). \\ \text{failed}(M, T0, TE) :- \text{reachable}(u^R, M, T0, TE). \end{array} \right\}$$

The following set, \mathcal{R}_4 , encodes multi-step transitions within an RM. The $\delta(u, u', M, t, t')$ atoms express that the transition from state u to state u' in RM M is satisfied between steps t and t' . The first rule indicates that this occurs if the context labeling a call to an RM $M2$ is satisfied and that RM is also satisfied (i.e., its accepting state is reached) between these two steps. In contrast, the second rule is used for the case in which the rejecting state of the called RM is reached between those steps. In the latter case, we transition to the local rejecting state u^R of M (i.e., the state we would have transitioned to does not matter), which follows from Assumption 6.1.2: rejecting states are global. The idea of the last rule is that rejection is propagated bottom-up in the HRM.

$$\mathcal{R}_4 = \left\{ \begin{array}{l} \delta(X, Y, M, T, T2) :- \varphi(X, Y, -, M, M2, T), \text{satisfied}(M2, T, T2). \\ \delta(X, u^R, M, T, T2) :- \varphi(X, -, -, M, M2, T), \text{failed}(M2, T, T2). \end{array} \right\}.$$

The last set, \mathcal{R}_5 , encodes acceptance and rejection. Remember that the $\text{last}(t)$ atoms indicate that t is the last step of a trace. The trace is accepted if the root RM is satisfied from the initial step 0 to step $T+1$ (i.e., the step after the last step of the trace, once the final label has been processed). In contrast, the trace is rejected if a rejecting state in the hierarchy is reached between these two same steps.

$$\mathcal{R}_5 = \left\{ \begin{array}{l} \text{accept} :- \text{last}(T), \text{satisfied}(M_r, 0, T+1). \\ \text{reject} :- \text{last}(T), \text{failed}(M_r, 0, T+1). \end{array} \right\}$$

Unlike the formalism introduced in Section 6.1, the encoding does not model call stacks, which is costly. Here, the processed trace is known and, therefore, it can be evaluated bottom-up in the hierarchy. Namely, the encoding evaluates the lowest level RMs on the different substraces; then, the resulting evaluations are subsequently used in higher level RMs.

Example 6.3.3. *Given the HRM in Figure 6.2b and the trace $\lambda = \langle \{\ddagger\}, \{\heartsuit\}, \{\}, \{\clubsuit\}, \{\heartsuit\}, \{\spadesuit\} \rangle$, the encoding proceeds by checking which substraces reach an accepting state in each RM. The substraces $\langle \{\ddagger\}, \{\heartsuit\} \rangle$ and $\langle \{\clubsuit\}, \{\heartsuit\} \rangle$ reach the accepting states of M_1 and M_2 , respectively. This information is leveraged to determine that the accepting state of M_0 is reached by calling M_1 and M_2 , as illustrated below:*

$$\underbrace{\underbrace{\langle \{\ddagger\}, \{\heartsuit\} \rangle}_{M_1}, \{\}, \underbrace{\langle \{\clubsuit\}, \{\heartsuit\} \rangle}_{M_2}, \{\spadesuit\}}_{M_0}.$$

Intuitively, the subtrace $\langle \{\}, \{\clubsuit\}, \{\heartsuit\} \rangle$ also reaches the accepting state of M_2 (the first label would perform a loop in the initial state of the RM); however, the fifth rule in \mathcal{R}_2 prevents this from happening in order to comply with the definition of HRM traversals.

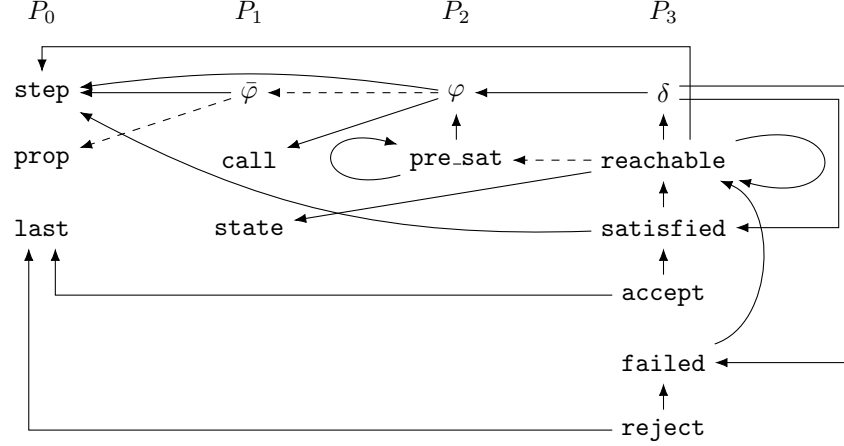


Figure 6.6: Predicate dependencies in the program of Proposition 6.3.1, where the predicates are grouped by the partitions used in the respective proof. Edges follow the specification from Figure 3.3.

6.3.3 Proof of Correctness

We prove the correctness of the ASP representation in the following lines. The non-factual representation of the HRMs is used since the traversal rules are defined over this representation. The result presented here is essential to prove the correctness of the HRM learning task in Chapter 7.

Proposition 6.3.1 (Correctness of the ASP encoding). *Given a finite label trace λ^* , where $*$ $\in \{G, D, I\}$, and an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ that is valid with respect to λ^* , the program $P = \mathbb{A}(H) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$ has a unique answer set A and (i) **accept** $\in A$ if and only if $* = G$, and (ii) **reject** $\in A$ if and only if $* = D$.*

Proof. First, we prove that the program $P = \mathbb{A}(H) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$, where $\mathcal{R} = \bigcup_{i=0}^5 \mathcal{R}_i$, has a unique answer set. If P is stratified then it has a unique answer set (see Section 2.2.1); hence, we prove that P is stratified. The program can be partitioned as $P = P_0 \cup P_1 \cup P_2 \cup P_3$, where

$$P_0 = \mathbb{A}(\lambda^*), \quad P_1 = \mathbb{A}(H), \quad P_2 = \mathcal{R}_0 \cup \mathcal{R}_1, \quad P_3 = \mathcal{R}_2 \cup \mathcal{R}_3 \cup \mathcal{R}_4 \cup \mathcal{R}_5.$$

Analogously to the proof of Proposition 3.3.1, Figure 6.6 graphically proves that this partitioning complies with the definition of a stratified program (see Section 2.2.1): dashed lines always point to predicates in lower-indexed partitions, whereas solid lines always point to predicates in the same partition or lower-indexed partitions.

The unique answer set $A = A_0 \cup A_1 \cup A_2 \cup A_3$, where A_i corresponds to partition P_i , is shown in Figure 6.7. To simplify the expression, we denote by $M_i(\lambda^*)$ the hierarchy traversal using RM M_i as the root.

We now prove that **accept** $\in A$ if and only if $* = G$ (i.e., the trace achieves the goal). If $* = G$ then, since the hierarchy is valid with respect to λ^* (see Definition 6.1.8), the hierarchy traversal $H(\lambda^*)$ finishes in the accepting state u^A of the root; that is, $H(\lambda^*)[n+1] = \langle M_r, u_r^A, \cdot, \cdot \rangle$. This holds if and only if **accept** $\in A$. The proof showing that **reject** $\in A$ if and only if $* = D$ (i.e., the trace reaches a dead-end) is similar to the previous one. If $* = D$ then, since the hierarchy is valid with respect to λ^* , the hierarchy traversal $H(\lambda^*)$ finishes in a rejecting state u^R ; that is, $H(\lambda^*)[n+1] = \langle M_k, u^R, \cdot, \cdot \rangle$, where $M_k \in \mathcal{M}$. This holds if and only if **reject** $\in A$. \square

$$\begin{aligned}
A_0 &= \{\text{prop}(p, t). \mid 0 \leq t \leq n, p \in \mathcal{L}_t\} \cup \{\text{step}(t). \mid 0 \leq t \leq n\} \cup \{\text{last}(n). \}, \\
A_1 &= \left\{ \begin{array}{l} \text{state}(u, M_i). \mid M_i \in \mathcal{M} \setminus \{M_\top\}, u \in \mathcal{U}_i \cup \\ \text{call}(u, u', x + y, M_i, M_j). \mid \begin{array}{l} M_i \in \mathcal{M} \setminus \{M_\top\}, M_j \in \mathcal{M}, u, u' \in \mathcal{U}_i, \varphi_i(u, u', M_j) \neq \perp, \\ x = \sum_{k=0}^{j-1} |\varphi_i(u, u', M_k)|, 1 \leq y \leq |\varphi_i(u, u', M_j)| \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \bar{\varphi}(u, u', x + y, M_i, t). \mid \begin{array}{l} 0 \leq t \leq n, M_i \in \mathcal{M} \setminus \{M_\top\}, M_j \in \mathcal{M}, u, u' \in \mathcal{U}_i, \\ \varphi_i(u, u', M_j) \neq \perp, x = \sum_{k=0}^{j-1} |\varphi_i(u, u', M_k)|, \\ 1 \leq y \leq |\varphi_i(u, u', M_j)|, \lambda^*[t] \models \varphi_i(u, u', M_j)[y] \end{array} \end{array} \right\}, \\
A_2 &= \left\{ \begin{array}{l} \varphi(u, u', x + y, M_i, t). \mid \begin{array}{l} 0 \leq t \leq n, M_i \in \mathcal{M} \setminus \{M_\top\}, M_j \in \mathcal{M}, u, u' \in \mathcal{U}_i, \\ \varphi_i(u, u', M_j) \neq \perp, x = \sum_{k=0}^{j-1} |\varphi_i(u, u', M_k)|, \\ 1 \leq y \leq |\varphi_i(u, u', M_j)|, \lambda^*[t] \models \varphi_i(u, u', M_j)[y] \end{array} \end{array} \right\} \cup \\
&\quad \{\text{pre_sat}(u, M_i, t). \mid 0 \leq t \leq n, M_i \in \mathcal{M} \setminus \{M_\top\}, u \in \mathcal{U}_i, \lambda^*[t] \models \xi_{i, u, \top}\}, \\
&\quad \left\{ \begin{array}{l} \text{reachable}(u^0, M_r, 0, 0). \} \cup \\ \text{reachable}(u^0, M_i, t, t). \mid 0 \leq t \leq n, M_i \in \mathcal{M} \setminus \{M_\top, M_r\}, u^0 \in \mathcal{U}_i \} \cup \\ \left\{ \begin{array}{l} \text{reachable}(u, M_r, t_1, t_2). \mid \begin{array}{l} 0 \leq t_1 < t_2 \leq n+1, u \in \mathcal{U}_r, \\ H(\lambda^*[t_1:])[t_2 - t_1] = \langle M_r, u, \cdot, \cdot \rangle \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \text{reachable}(u, M_i, t_1, t_2). \mid \begin{array}{l} 0 \leq t_1 < t_2 \leq n+1, M_i \in \mathcal{M} \setminus \{M_r, M_\top\}, u \in \mathcal{U}_i, \\ \lambda^*[t_1] \models \xi_{i, u^0, \top}, \\ M_i(\lambda^*[t_1:])[t_2 - t_1] = \langle M_i, u, \cdot, \cdot \rangle, \\ M_i(\lambda^*[t_1:])[t_2 - t_1 - 1] \neq \langle M_i, u^A, \cdot, \cdot \rangle \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \text{satisfied}(M_r, t_1, t_2) \mid 0 \leq t_1 < t_2 \leq n+1, H(\lambda^*[t_1:])[t_2 - t_1] = \langle M_r, u^A, \cdot, \cdot \rangle \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \text{satisfied}(M_i, t_1, t_2). \mid \begin{array}{l} 0 \leq t_1 < t_2 \leq n+1, M_i \in \mathcal{M} \setminus \{M_r, M_\top\}, \\ \lambda^*[t_1] \models \xi_{i, u^0, \top}, \\ M_i(\lambda^*[t_1:])[t_2 - t_1] = \langle M_i, u^A, \cdot, \cdot \rangle, \\ M_i(\lambda^*[t_1:])[t_2 - t_1 - 1] \neq \langle M_i, u^A, \cdot, \cdot \rangle \end{array} \end{array} \right\} \cup \\
&\quad \{\text{satisfied}(M_\top, t, t+1) \mid 0 \leq t \leq n\} \cup \\
A_3 &= \left\{ \begin{array}{l} \text{failed}(M_r, t_1, t_2) \mid 0 \leq t_1 < t_2 \leq n+1, H(\lambda^*[t_1:])[t_2 - t_1] = \langle \cdot, u^R, \cdot, \cdot \rangle \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \text{failed}(M_i, t_1, t_2). \mid \begin{array}{l} 0 \leq t_1 < t_2 \leq n+1, M_i \in \mathcal{M} \setminus \{M_r, M_\top\}, \\ \lambda^*[t_1] \models \xi_{i, u^0, \top}, \\ M_i(\lambda^*[t_1:])[t_2 - t_1] = \langle \cdot, u^R, \cdot, \cdot \rangle \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \delta(u, u', M_i, t, t+1). \mid \begin{array}{l} 0 \leq t \leq n, M_i \in \mathcal{M} \setminus \{M_\top\}, u, u' \in \mathcal{U}_i, \\ \lambda^*[t_1] \models \varphi_i(u, u', M_\top) \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \delta(u, u', M_i, t_1, t_2). \mid \begin{array}{l} 0 \leq t_1 < t_2 \leq n+1, M_i \in \mathcal{M} \setminus \{M_\top\}, u, u' \in \mathcal{U}_i, \\ \exists M_j \in \mathcal{M} \setminus \{M_\top\} \text{ s.t. } \phi = \varphi_i(u, u', M_j), \lambda^*[t_1] \models \xi_{j, u^0, \phi}, \\ M_j(\lambda^*[t_1:])[t_2 - t_1] = \langle M_j, u^A, \cdot, \cdot \rangle, \\ M_j(\lambda^*[t_1:])[t_2 - t_1 - 1] \neq \langle M_j, u^A, \cdot, \cdot \rangle \end{array} \end{array} \right\} \cup \\
&\quad \left\{ \begin{array}{l} \delta(u, u^R, M_i, t_1, t_2). \mid \begin{array}{l} M_i \in \mathcal{M} \setminus \{M_\top\}, u \in \mathcal{U}_i, 0 \leq t_1 < t_2 \leq n+1, \\ \exists M_j \in \mathcal{M} \setminus \{M_\top\} \text{ s.t. } \phi = \varphi_i(u, u', M_j), \lambda^*[t_1] \models \xi_{j, u^0, \phi}, \\ M_j(\lambda^*[t_1:])[t_2 - t_1] = \langle M_k, u^R, \cdot, \cdot \rangle, M_k \in \mathcal{M} \end{array} \end{array} \right\} \cup \\
&\quad \{\text{accept} \mid H(\lambda^*)[n+1] = \langle M_r, u^A, \cdot, \cdot \rangle\} \cup \\
&\quad \{\text{reject} \mid H(\lambda^*)[n+1] = \langle M_k, u^R, \cdot, \cdot \rangle, M_k \in \mathcal{M} \setminus \{M_\top\}\}.
\end{aligned}$$

Figure 6.7: Answer sets for each of the partitions in the program $P = \mathbb{A}(H) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$, where H is an HRM, \mathcal{R} is the set of general rules and λ^* is a label trace.

6.3.4 Determinism

The hierarchical transition function δ_H of an HRM H must be deterministic. To ensure an HRM is deterministic, as described in Section 6.1, the logical transition functions of the constituent RMs must be such that a label cannot simultaneously satisfy the contexts and exit conditions associated with two triplets $\langle u, v, M_i \rangle$ and $\langle u, v', M_j \rangle$ such that either (i) $v = v'$ and $i \neq j$, or (ii) $v \neq v'$. Akin to regular RMs, determinism is guaranteed if the formulas associated with such triplets are mutually exclusive.

In what follows, we describe how to verify whether the root of an HRM is deterministic using an ASP technique called *saturation* (Eiter and Gottlob, 1995). The idea is to check determinism top-down by selecting two edges from a given state in the root, each associated with a set of literals. Initially, each set of literals is formed by those in the formula labeling the edge (i.e., the context). If a selected edge calls a non-leaf RM, we select an edge from the initial state of the called RM, augment the respective set of literals with the associated formula, and repeat the process until the leaf RM is called. We then check if the literal sets are mutually exclusive. If there is a pair of non-mutually exclusive literal sets, the root of the HRM is non-deterministic.

The idea above is represented using the following set of rules built on the factual representation of an HRM. The first rule states that we keep two saturation IDs, one for each of the edges we select next and for which mutual exclusivity is checked. The second rule chooses a state X of the root, whereas the third rule selects two edges from this state and assigns a saturation ID to each of them. The fourth rule indicates that if one of the edges we have selected so far calls a non-leaf RM, we select an edge from the initial state of the called RM and create a new edge with the same saturation ID. The fifth (resp. sixth) rule takes the propositions appearing positively (resp. negatively) for each set of edges (one per saturation ID). The next three rules indicate that if the edges are mutually exclusive (i.e., a proposition appears positively in one set and negatively in the other) or they are the same, then the answer set is saturated. The saturation itself is encoded in the following three rules: an answer set is saturated by adding every possible `ed_mtx` and `root_point` atoms to the answer set. Due to the minimality of answer sets in disjunctive answer set programming, this *maximal* interpretation can only be an answer set if there is no smaller answer set. This will be the case if and only if every choice of edges satisfies the condition (i.e., every choice of `ed_mtx` and `root_point` atoms results in saturation). The constraint encoded in the final rule then discards answer sets in which saturation did not occur, meaning that the remaining solutions must satisfy the condition.

$$\left\{ \begin{array}{l} \text{sat_id}(1; 2). \\ \text{root_point}(X, M) : \text{call}(X, _, _, M, _), M = M_r. \\ \text{ed_mtx}((X, Y, E, M, M2), \text{SatID}) : \text{call}(X, Y, E, M, M2) : - \text{root_point}(X, M), \text{sat_id}(\text{SatID}). \\ \text{ed_mtx}((u^0, Y2, E2, M2, M3), \text{SatID}) : \text{call}(u^0, Y2, E2, M2, M3) : - \text{ed_mtx}((_, _, _, _, M2), \text{SatID}), M2 \neq M_\top. \\ \text{pos_prop}(P, \text{SatID}) : - \text{ed_mtx}((X, Y, E, M, _), \text{SatID}), \text{pos}(X, Y, E, M, P). \\ \text{neg_prop}(P, \text{SatID}) : - \text{ed_mtx}((X, Y, E, M, _), \text{SatID}), \text{neg}(X, Y, E, M, P). \\ \text{saturate} : - \text{pos_prop}(P, 1), \text{neg_prop}(P, 2). \\ \text{saturate} : - \text{pos_prop}(P, 2), \text{neg_prop}(P, 1). \\ \text{saturate} : - \text{ed_mtx}((X, Y, _, M, M2), 1), \text{ed_mtx}((X, Y, _, M, M2), 2), \text{root_point}(X, M). \\ \text{root_point}(X, M) : - \text{call}(X, _, _, M, _), \text{saturate}, M = M_r. \\ \text{ed_mtx}((X, Y, E, M, M2), \text{SatID}) : - \text{call}(X, Y, E, M, M2), M = M_r, \text{sat_id}(\text{SatID}), \text{saturate}. \\ \text{ed_mtx}((u^0, Y, E, M, M2), \text{SatID}) : - \text{call}(u^0, Y, E, M, M2), \text{sat_id}(\text{SatID}), \text{saturate}. \\ :- \text{not saturate}. \end{array} \right\}$$

While determinism is only checked in the root of the HRM (and, indirectly, from the initial states of the subsequently called RMs), the algorithm for learning HRMs described in Chapter 7 learns each constituent RM individually; hence, determinism across the entire HRM is guaranteed. Alternatively, the second rule can be modified such that the state is not restricted to be one in the root; however, this would involve checking mutual exclusivity for each state in the HRM, which is costly (especially in our HRM learning context, where non-root RMs behave deterministically).

6.3.5 Symmetry Breaking

We extend the symmetry breaking method proposed for RMs in Section 3.4.3 to HRMs. The symmetry breaking is applied to the root only instead of all the constituent RMs. This follows from the reason described in Section 6.3.4 regarding the rules for verifying determinism, which are also applied to the root of the HRM only: the HRM learning method described in Chapter 7 learns each RM individually. Hence, we focus on breaking symmetries in the root. In any case, the extended method is applicable to each constituent RM separately; thus, verifying whether all RMs follow the proposed canonical way of indexing states and edges is feasible.

The reward machines we here consider differ from those in Part I: the edges are not only labeled by formulas, but also by calls to other machines. Therefore, the set of labels that determines a given RM's indexing consists of both propositions (and their negations) and the potentially called RMs. Formally, given a proposition set \mathcal{P} , an RM set \mathcal{M} , a bijective function $f : \mathcal{P} \rightarrow \{1, \dots, |\mathcal{P}|\}$ mapping each proposition to a different integer between 1 and $|\mathcal{P}|$, and a bijective function $g : \mathcal{M} \rightarrow \{1, \dots, |\mathcal{M}|\}$ mapping each RM to a different integer between 1 and $|\mathcal{M}|$, the set of labels is

$$\begin{aligned} \mathcal{L}_{\text{sb}} = & \{f(p) \mid p \in \mathcal{P}\} \cup \\ & \{|\mathcal{P}| + g(M) \mid M \in \mathcal{M}\} \cup \\ & \{|\mathcal{P}| + |\mathcal{M}| + f(p) \mid p \in \mathcal{P}\} \end{aligned}$$

where labels $1, \dots, |\mathcal{P}|$ correspond to propositions, labels $|\mathcal{P}| + 1, \dots, |\mathcal{P}| + |\mathcal{M}|$ correspond to RMs, and labels $|\mathcal{P}| + |\mathcal{M}| + 1, \dots, 2|\mathcal{P}| + |\mathcal{M}|$ correspond to the negated propositions. This differs from the set in Equation 3.1, which did not include labels for RMs. The mapping from propositions and RMs to integer labels is encoded in ASP using the following atoms:

- `symbol_id(s, l)` indicates that symbol (i.e., proposition or RM) $s \in \mathcal{P} \cup \mathcal{M}$ is associated with label l .
- `num_symbols(i)` indicates that the number of symbols (propositions and RMs) is i .
- `valid_sb_label(l)` indicates that l is a label.

We ground the above atoms according to their descriptions:

$$\begin{aligned} & \{\text{symbol_id}(p, f(p)). \mid p \in \mathcal{P}\} \cup \\ & \{\text{symbol_id}(M, |\mathcal{P}| + g(M)). \mid M \in \mathcal{M}\} \cup \\ & \{\text{num_symbols}(|\mathcal{P}| + |\mathcal{M}|). \} \cup \\ & \{\text{valid_sb_label}(l). \mid 1 \leq l \leq 2|\mathcal{P}| + |\mathcal{M}|\}. \end{aligned}$$

To complete the mapping, we map the formulas on the edges into label sets leveraging the factual representation from Definition 6.3.2. This mapping builds upon the *Efficient Alternative Encoding* described in Section 3.4.3, which is modified as follows:

- The rule set below results from that in Equation 3.2. First, the `prop_id` and `num_props` predicates in the first two rules are replaced with the `symbol_id` and `num_symbols` predicates. Second, the rules are defined exclusively on the root M_r of the HRM. Third, a new rule is added to set MID as a label of edge (Y, E) from X if the corresponding RM M is called in that edge.

$$\left\{ \begin{array}{l} \text{label}(X, (Y, E), \text{PID}) :- \text{pos}(X, Y, E, M_r, P), \text{symbol_id}(P, \text{PID}). \\ \text{label}(X, (Y, E), \text{PID}+N) :- \text{neg}(X, Y, E, M_r, P), \text{symbol_id}(P, \text{PID}), \text{num_symbols}(N). \\ \text{label}(X, (Y, E), \text{MID}) :- \text{call}(X, Y, E, M_r, M), \text{symbol_id}(M, \text{MID}). \end{array} \right\}$$

- The first and third rules in Equation 3.3 are changed by defining them on the root M_r of the HRM:

$$\left\{ \begin{array}{l} 1\{\text{ed_lt}(X, (Y, E), (Y, EP)); \text{ed_lt}(X, (Y, EP), (Y, E))\} 1 :- \text{ed}(X, Y, E, M_r), \text{ed}(X, Y, EP, M_r), \\ \quad (Y, E) < (Y, EP). \\ :- \text{ed_lt}(X, \text{Edge1}, \text{Edge2}), \text{ed_lt}(X, \text{Edge2}, \text{Edge3}), \text{not ed_lt}(X, \text{Edge1}, \text{Edge3}), \\ \quad \text{Edge1} \neq \text{Edge3}. \\ :- \text{ed_lt}(X, (Y, E), (Y, EP)), \text{ed}(X, Y, E, M_r), \text{ed}(X, Y, EP, M_r), E > EP. \end{array} \right\}.$$

- The second rule in Equation 3.4 is altered by defining it on the root M_r of the HRM:

$$\left\{ \begin{array}{l} \text{edge_id}(1..K). \\ :- \text{ed}(X, Y, E, M_r), \text{not ed}(X, Y, E-1, M_r), \text{edge_id}(E), E > 1. \end{array} \right\}.$$

- Equation 3.5 is modified to define the rule on the root M_r of the HRM:

$$\text{ed_sb}(X, Y, E) :- \text{ed}(X, Y, E, M_r), \text{state_id}(Y, -).$$

6.4 Summary

In this chapter, we have laid the foundations for Part II by introducing HRMs, a formalism for hierarchically composing RMs. We proved that these hierarchies are equivalent to regular RMs; besides, under certain conditions, the equivalent standard RM can have exponentially more states and edges than the hierarchy. In the following chapter, we describe exploitation and learning methods that leverage the decomposition into smaller RMs enabled by hierarchies. The learning method we propose learns HRMs expressed using the ASP representation presented in this chapter.

Chapter 7

Learning and Exploiting Hierarchies of Reward Machines

In this chapter, we introduce a hierarchical RL algorithm for *exploiting* the structure of a hierarchy of reward machines (Section 7.1) and a method for *learning* hierarchies from traces (Section 7.2). These methods are then combined into a curriculum-based algorithm that *interleaves* them (Section 7.3).

7.1 Exploiting Hierarchies of Reward Machines

In this section, we describe a method for exploiting hierarchies of reward machines using the options framework. First, we formalize the types of options we consider (Section 7.1.1). Second, we explain how these options are selected, updated, and interrupted during an episode (Section 7.1.2). Third, and finally, we introduce two strategies for efficiently updating the policies of the options in the hierarchy (Section 7.1.3).

7.1.1 Options

In what follows, we explain how to *exploit* the temporal structure of an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ using *options*. Given an RM $M_i \in \mathcal{M}$, a state $u \in \mathcal{U}_i$ and a context $\Phi \in \text{DNF}_{\mathcal{P}}$, the set of available options in a hierarchy state $\langle M_i, u, \Phi, \Gamma \rangle$ is

$$\Omega_{i,u,\Phi} = \left\{ \omega_{i,u,\Phi}^{j,\phi} \mid \phi \in \varphi_i(u, u', M_j), u' \in \mathcal{U}_i, M_j \in \mathcal{M}, \phi \neq \perp \right\};$$

that is, an option $\omega_{i,u,\Phi}^{j,\phi}$ is derived for each non-false disjunct ϕ of each transition $\varphi_i(u, u', M_j)$, where $u' \in \mathcal{U}_i$ and $M_j \in \mathcal{M}$. The stack Γ is not required to define the options. An option can be one of the following depending on the called machine M_j :

- A *formula option* if $j = \top$ (i.e., the M_{\top} is called), which aims to reach a label that satisfies $\phi \wedge \Phi$ through primitive actions.
- A *call option* if $j \neq \top$, which aims to reach an accepting state of the called RM M_j under context $\phi \wedge \Phi$ by invoking other options.

In the following paragraphs, we describe the initiation set, the policy, and the termination condition for each type of option.

Initiation Set

The initiation set of any option $\omega_{i,u,\Phi}^{j,\phi}$ is $\mathcal{I}_{i,u,\Phi} = \mathcal{S}$; that is, any option available in hierarchy state $\langle M_i, u, \Phi, \Gamma \rangle$ can be started in any state. As previously mentioned, the call stack Γ is irrelevant to defining the options.

Policy

We here describe the policy for each option type and the resulting optimality guarantees. The policies for both option types are derived from action-value functions approximated by DQNs, being ϵ -greedy during training, and greedy during evaluation.

Formula Options. A *formula option's policy* $\pi_{\phi \wedge \Phi} : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ is derived from an action-value function $q_{\phi \wedge \Phi}(s, a; \theta_{\phi \wedge \Phi})$ approximated by a DQN with parameters $\theta_{\phi \wedge \Phi}$, which outputs the value of each action given an MDP state. The experiences $\langle s_t, a, s_{t+1}, \mathcal{L}_{t+1} \rangle$ obtained by all formula options in the HRM are stored in a single replay buffer \mathcal{D} , thus performing intra-option learning. The Q-learning update uses the following loss function:

$$\mathbb{E}_{\langle s_t, a, s_{t+1}, \mathcal{L}_{t+1} \rangle \sim \mathcal{D}} \left[\left(r_{\phi \wedge \Phi}(\mathcal{L}_{t+1}) + \gamma \max_{a' \in \mathcal{A}} q_{\phi \wedge \Phi}(s_{t+1}, a'; \theta_{\phi \wedge \Phi}^-) - q_{\phi \wedge \Phi}(s_t, a; \theta_{\phi \wedge \Phi}) \right)^2 \right], \quad (7.1)$$

where $\theta_{\phi \wedge \Phi}^-$ are the parameters of a fixed target DQN. The reward $r_{\phi \wedge \Phi}(\mathcal{L}_{t+1})$ is +1 if $\phi \wedge \Phi$ is satisfied by label \mathcal{L}_{t+1} and 0 otherwise; hence, the policy aims to observe a label that satisfied $\phi \wedge \Phi$. The discounted term of the target becomes 0 when either $\phi \wedge \Phi$ is satisfied or the history is a dead-end history (i.e., $s_{t+1}^T = \top$ and $s_{t+1}^G = \perp$); akin to Equation 4.2, the latter case assumes that dead-end histories depend on the last state only.

Each formula option $\omega_{i,u,\Phi}^{\top,\phi}$ explores using an exploration factor $\epsilon_{\phi \wedge \Phi}$, which linearly decreases with the number of steps performed using the policy induced by $q_{\phi \wedge \Phi}$. Similarly, Kulkarni et al. (2016) keep an exploration factor for each subgoal, but vary it depending on the option's success rather than the number of performed steps.

Call Options. A *call option's policy* $\pi_{i,u,\Phi} : \mathcal{S} \rightarrow \Delta(\Omega_{i,u,\Phi})$ is induced by an option-value function $q_i(s, u, \Phi, \langle M_j, \phi \rangle; \theta_i)$ associated with the called RM M_i and approximated by a DQN with parameters θ_i . The DQN outputs the value of each call in the RM given an MDP state, an RM state, and a context. Each call option explores using an exploration factor $\epsilon_{i,u,\Phi}$ that linearly decreases with the number of times an option starting in the triplet $\langle M_i, u, \Phi \rangle$ terminates. Option experiences $\langle s_t, \omega_{i,u,\Phi}^{j,\phi}, s_{t+k} \rangle$ are stored in the replay buffer \mathcal{D}_i associated with the RM M_i where they are defined. The DQN parameters are updated by performing SMDP Q-learning using the following loss:

$$\mathbb{E}_{\langle s_t, \omega_{i,u,\Phi}^{j,\phi}, s_{t+k} \rangle \sim \mathcal{D}_i} \left[\left(r + \gamma^k \max_{j', \phi'} q_i(s_{t+k}, u', \Phi', \langle M_{j'}, \phi' \rangle; \theta_i^-) - q_i(s_t, u, \Phi, \langle M_j, \phi \rangle; \theta_i) \right)^2 \right],$$

where

- θ_i^- are the parameters of a fixed target DQN;
- k is the number of steps between s_t and s_{t+k} ;
- r is the sum of discounted rewards emitted by the reward-transition function r_i during this time;
- u' and Φ' are the RM state and context after running the option;
- $M_{j'}$ and ϕ' correspond to an outgoing transition from u' , i.e. $\phi' \in \varphi_i(u', \cdot, M_{j'})$; and
- the discounted term becomes 0 if u' is an accepting or rejecting state.

We make two observations. First, the policies on a given RM are trained to reach an accepting state in the fewest possible steps (given the selected options' policies) since, by Assumption 6.1.5, the reward-transition function of an RM M_i is $r_i(u, u') = \mathbb{1}[u \notin \mathcal{U}_i^A \wedge u' \in \mathcal{U}_i^A]$; thus, the cumulative discounted reward r only has a non-zero value of γ^{k-1} when u' is an accepting state since a reward of +1 is given after $k-1$ steps in that situation. Second, by Definition 6.1.6, the accumulated context Φ' becomes \top when the hierarchy state changes (i.e., the context of a call is lost once a transition is taken from the initial state of the called RM); therefore, $\Phi' = \top$ if $u' \neq u$, and $\Phi' = \Phi$ otherwise.

Just like for the metapolicies in Section 4.1.1, the RM states are encoded using one-hot vectors. A context ϕ , which is either \top or a DNF formula with a single disjunct, is encoded using a vector where each position corresponds to a proposition $p \in \mathcal{P}$ whose value is (i) +1 if p appears positively in ϕ , (ii) -1 if p appears negatively in ϕ , or (iii) 0 if p does not appear in ϕ . Note that if $\phi = \top$, the vector solely consists of zeros. The output of each DQN is masked to avoid considering unavailable or unsatisfiable¹ calls for a given RM state-context pair.

Example 7.1.1. *The DQN for M_0 in the HRM of Figure 6.2b outputs a value for each possible call, i.e. $\langle M_1, \neg\blacktriangledown \rangle$, $\langle M_2, \top \rangle$, $\langle M_1, \top \rangle$, and $\langle M_\top, \blacktriangledown \rangle$. If the input RM state-context pair is $\langle u_0^0, \top \rangle$, only the values for $\langle M_1, \neg\blacktriangledown \rangle$ and $\langle M_2, \top \rangle$ are considered since they are the only available calls from u_0^0 .*

Learning a call option's policy and lower-level option policies at once can be unstable due to *non-stationarity* (Levy et al., 2019), e.g. a lower-level option may sometimes fail to achieve its goal. To relax the issue, experiences are added to the buffer only when options achieve their goal (i.e., call options assume lower-level options terminate successfully).

Optimality. The policies will be *recursively optimal* at best as each subtask is optimized individually; however, since the action-value functions are approximated, policies may only be approximately optimal. The result is in line with the optimality of the method described in Section 4.1.1; indeed, the main difference is that the method presented here makes decisions at arbitrarily many hierarchical levels instead of at only two levels.

Termination Condition

In general, options terminate when the history is terminal (i.e., a goal or a dead-end history). Ideally, if the HRM perfectly captures histories, this entails that (i) goal traces reach an accepting state of the root, and (ii) dead-end traces reach any of the rejecting states in the HRM. In the following paragraphs, we describe the specific termination conditions of each option type.

¹A call is unsatisfiable if the conjunction of its context ϕ and the input accumulated context Φ cannot be satisfied.

Formula Options. The termination condition $\beta_{i,u,\Phi}$ of a formula option generalizes that of the options from Section 4.1.1; namely, instead of checking whether a formula labeling an outgoing edge from the current state is satisfied, we check whether the exit condition for a hierarchy state $\langle M_i, u, \Phi, \cdot \rangle$ is satisfied. Formally,

$$\beta_{i,u,\Phi}(s) = \begin{cases} 1 & \text{if } l(s) \models \xi_{i,u,\Phi}; \\ 0 & \text{otherwise.} \end{cases}$$

In practice, the agent determines termination using the label emitted by the environment (i.e., it does not evaluate the labeling function l by itself); moreover, we reduce the condition to checking whether the hierarchy state has changed, which is equivalent since it means that the previous hierarchy state has been exited. Note that these options may terminate even if the target formula $\phi \wedge \Phi$ is not satisfied since termination is not determined by the formula, but by exiting the hierarchy state where the option is available.

Call Options. Unlike formula options, call options may remain active after a change in the hierarchy state; hence, the termination condition must account for the fact that several such changes might happen before the option ends. Hierarchy states keep track of the active calls in the HRM; therefore, we can check whether there exists a stack item $\langle u, \cdot, M_i, M_j, \phi, \Phi \rangle$ that corresponds to the call option $\omega_{i,u,\Phi}^{j,\phi}$. If such a stack item does not exist, the option terminates since either the RM M_j called under context ϕ was not started or its accepting state was reached. For clarity, the termination of these options is exemplified in Section 7.1.2.

7.1.2 Algorithm

Algorithm 2 shows how options are selected, updated, and interrupted during an episode. Lines 1–3 correspond to the algorithm’s initialization. The initial state is that of the environment, while the initial hierarchy state is formed by the root RM M_r , its initial state u_r^0 , an empty context (i.e., $\Phi = \top$), and an empty call stack. A hierarchical transition is applied to the initial hierarchy state using the initial label \mathcal{L}_0 , which returns the actual initial hierarchy state based on what the agent initially observes. The (initially empty) *option stack* Ω_H contains the currently executing options, where options at the front are the shallowest ones; for instance, the first option in the list is taken in the root RM. The steps taken during an episode are shown in lines 4–14, which are grouped as follows:

1. The agent fills the option stack Ω_H by selecting options in the HRM from the current hierarchy state using call option policies until a formula option is chosen (lines 15–25). A formula option will eventually be selected since HRMs have no circular dependencies by Assumption 6.1.1. The context is propagated and augmented through the HRM (i.e., the context of the calls is conjuncted with the propagating context and converted into DNF form). Note that the context is initially \top , and not that of the last option in the option stack. No new options may be selected if the formula option chosen in a previous step has not terminated yet.
2. The agent chooses an action according to the last option in the option stack (line 6), which will always be a formula option whose policy maps states into actions. The action is applied,

and the agent observes the next state and label (line 7). The next hierarchy state is obtained by applying the hierarchical transition function δ_H using the observed label (line 8). The action-value functions associated with formula options' policies are updated after this step (line 9).

3. The option stack Ω_H is updated by removing those options that have terminated (lines 10, 26–45). The terminated options are saved in a different list Ω_β to update the value functions of the RMs where they were initiated later on (line 11). The options are terminated as described in Section 7.1.1. All options terminate if the history is terminal (lines 27–28). Otherwise, we check options in Ω_H from deeper to shallower levels. The first checked option is always a formula option, which terminates if the hierarchy state has changed (line 40). In contrast, a call option terminates if it does not appear in the stack (lines 33, 46–51).² When an option is found to terminate, it is added to Ω_β and removed from Ω_H (lines 35–36, 41–42). If a non-terminating option is found (lines 37, 43), we stop checking for termination (no higher level options can have terminated in this case).
4. If at least one option has terminated (line 12), the option stack Ω_H is updated such that it contains all options appearing in the call stack (lines 13, 52–70). By aligning the option stack with the call stack, we can later update the value functions for options that ended up being run in *hindsight* and which would have been otherwise ignored. Options are derived for the full stack if Ω_H is empty (lines 53–54), or for the part of the stack not appearing in Ω_H (lines 56–59). The new derived options (lines 61–70) from the call stack are assumed to start in the same state as the last terminated option (i.e., the shallowest terminated option, line 63) and to have been run for the same number of steps too. Crucially, the contexts should be accumulated accordingly, starting from the context of the last terminated option (line 69).

As a result of the definition of the hierarchical transition function δ_H , the contexts in the stack may be DNF formulas with more than one disjunct. In contrast, the contexts associated with options are either \top or DNFs with a single disjunct (remember that an option is formed for each disjunct). For instance, this occurs if the context is $a \vee b$ and $\{a, b\}$ is observed: since both disjuncts are satisfied, the context shown in the call stack will be the full disjunction $a \vee b$. In the simplest case, the derived option (which, as said before, is associated with a DNF with a single disjunct or \top) can include one of these disjuncts chosen uniformly at random (line 67). Alternatively, we could memorize all the derived options and perform identical updates for both once terminated.

Figure 7.1 illustrates the core procedures that constitute the option selection algorithm: (i) filling the option stack, (ii) selecting an action using the formula option in the option stack, and (iii) applying the action and updating the value functions and the option stack accordingly.

Examples

We briefly describe some examples of how policy learning is performed in the HRM of Figure 6.2b. We first enumerate the options in the hierarchy. The formula options are $\omega_{1,0,\neg\forall}^{\top,\#}$, $\omega_{2,0,\top}^{\top,\forall}$, $\omega_{1,0,\top}^{\top,\#}$, $\omega_{1,1,\top}^{\top,\forall}$,

²We denote by $\phi \subseteq \phi'$, where $\phi, \phi' \in \text{DNF}\mathcal{P}$, the fact that all the disjuncts of ϕ appear in ϕ' . For instance, $(a \wedge \neg c) \subseteq (a \wedge \neg c) \vee d$. If $\phi = \top$, ϕ' must also be \top for the containment relationship to hold (and vice versa).

Algorithm 2 Episode execution using an HRM (continues on the next page)

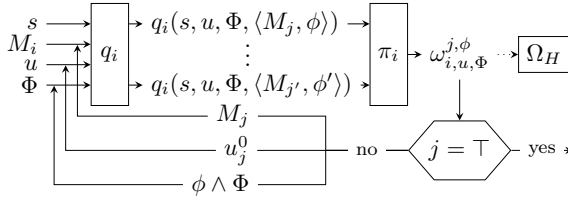
Input: an HRM $H = \langle \mathcal{M}, M_r, \mathcal{P} \rangle$ and an environment $\text{ENV} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma, \mathcal{P}, l, \tau \rangle$.

- 1: $s_0, \mathcal{L}_0 \leftarrow \text{ENV.INIT}()$ ▷ Initial MDP tuple and label
- 2: $\langle M_i, u, \Phi, \Gamma \rangle \leftarrow \delta_H(\langle M_r, u_r^0, \top, [] \rangle, \mathcal{L}_0)$ ▷ Initial hierarchy state
- 3: $\Omega_H \leftarrow []$ ▷ Initial option stack
- 4: **for** each step $t = 0, \dots$, **do**
- 5: $\Omega_H \leftarrow \text{FILLOPTIONSTACK}(s_t, \langle M_i, u, \Phi, \Gamma \rangle, \Omega_H)$ ▷ Expand the option stack
- 6: $a \leftarrow \text{SELECTACTION}(s_t, \Omega_H)$ ▷ Choose a according to the last option in Ω_H
- 7: $s_{t+1}, \mathcal{L}_{t+1} \leftarrow \text{ENV.STEP}(a)$
- 8: $\langle M_j, u', \Phi', \Gamma' \rangle \leftarrow \delta_H(\langle M_i, u, \Phi, \Gamma \rangle, \mathcal{L}_{t+1})$ ▷ Apply transition function
- 9: $\text{UPDATEFORMULAVALEFUNCTIONS}(s_t, a, s_{t+1}, \mathcal{L}_{t+1})$
- 10: $\Omega_\beta, \Omega_H \leftarrow \text{TERMINATEOPTIONS}(\Omega_H, s, \langle M_i, u, \Phi, \Gamma \rangle, \langle M_j, u', \Phi', \Gamma' \rangle)$
- 11: $\text{UPDATECALLVALUEFUNCTIONS}(\Omega_\beta, s_{t+1}, \mathcal{L}_{t+1})$
- 12: **if** $|\Omega_\beta| > 0$ **then**
- 13: $\Omega_H \leftarrow \text{ALIGNOPTIONSTACK}(\Omega_H, \Gamma', \Omega_\beta)$
- 14: $\langle M_i, u, \Phi, \Gamma \rangle \leftarrow \langle M_j, u', \Phi', \Gamma' \rangle$
- 15: **function** $\text{FILLOPTIONSTACK}(s, \langle M_i, u, \cdot, \Gamma \rangle, \Omega_H)$
- 16: $\Omega'_H \leftarrow \Omega_H$
- 17: $\Phi \leftarrow \top$ ▷ The context is initially true
- 18: $M_j \leftarrow M_i; v \leftarrow u$ ▷ The RM-state pair in which an option is selected
- 19: **while** the last option in Ω'_H is not a formula option **do**
- 20: $\omega_{j,v,\Phi}^{x,\phi} \leftarrow \text{SELETOPTION}(s, M_j, v, \Phi)$ ▷ Select an option (e.g., with ϵ -greedy)
- 21: **if** $x \neq \top$ **then** ▷ If the option is a call option
- 22: $M_j \leftarrow M_x; v \leftarrow u_x^0$ ▷ Next option is chosen on the called RM's initial state
- 23: $\Phi \leftarrow \text{DNF}(\Phi \wedge \phi)$ ▷ Update the context
- 24: $\Omega'_H \leftarrow \Omega'_H \oplus \omega_{j,v,\Phi}^{x,\phi}$ ▷ Update the option stack (concatenate new option)
- 25: **return** Ω'_H
- 26: **function** $\text{TERMINATEOPTIONS}(\Omega_H, s, \langle M_i, u, \Phi, \Gamma \rangle, \langle M_j, u', \Phi', \Gamma' \rangle)$
- 27: **if** $s^T = \top$ **then**
- 28: **return** $\Omega_H, []$ ▷ All options terminate
- 29: $\Omega_\beta \leftarrow []; \Omega'_H \leftarrow \Omega_H$ ▷ Initialize structures
- 30: **while** $|\Omega'_H| > 0$ **do** ▷ While the option stack is not empty
- 31: $\omega_{k,v,\Psi}^{x,\phi} \leftarrow \text{last option in } \Omega'_H$
- 32: **if** $x \neq \top$ **then** ▷ If the option is a call option
- 33: $\text{in_stack}, _ \leftarrow \text{OPTIONINSTACK}(\omega_{k,v,\Psi}^{x,\phi}, \Gamma')$
- 34: **if** $\neg \text{in_stack}$ **then**
- 35: $\Omega_\beta \leftarrow \Omega_\beta \oplus \omega_{k,v,\Psi}^{x,\phi}$ ▷ Update the list of terminated options
- 36: $\Omega'_H \leftarrow \Omega'_H \ominus \omega_{k,v,\Psi}^{x,\phi}$ ▷ Remove the last option from the option stack
- 37: **else**
- 38: **break** ▷ Stop terminating
- 39: **else**
- 40: **if** $\langle M_i, u, \Phi, \Gamma \rangle \neq \langle M_j, u', \Phi', \Gamma' \rangle$ **then** ▷ If the hierarchy state has changed...
- 41: $\Omega_\beta \leftarrow \Omega_\beta \oplus \omega_{k,v,\Psi}^{x,\phi}$ ▷ Update the list of terminated options
- 42: $\Omega'_H \leftarrow \Omega'_H \ominus \omega_{k,v,\Psi}^{x,\phi}$ ▷ Remove the last option from the option stack
- 43: **else**
- 44: **break** ▷ Stop terminating
- 45: **return** Ω_β, Ω'_H

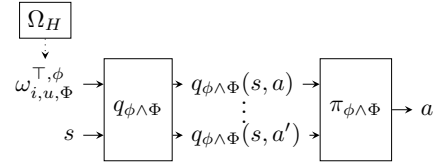
```

46: function OPTIONINSTACK( $\omega_{k,v,\Phi}^{x,\phi}, \Gamma$ )
47:   for  $l = 0 \dots |\Gamma| - 1$  do
48:      $\langle u_f, \cdot, M_i, M_j, \phi', \Phi' \rangle \leftarrow \Gamma_l$ 
49:     if  $u_f = v \wedge i = k \wedge j = x \wedge \phi \subseteq \phi' \wedge \Phi \subseteq \Phi'$  then  $\triangleright$  The call option is in the call stack
50:       return  $\top, l$   $\triangleright$  Return whether it appears in the stack and the index
51:   return  $\perp, -1$ 
52: function ALIGNOPTIONSTACK( $\Omega_H, \Gamma, \Omega_\beta$ )
53:   if  $|\Omega_H| = 0$  then
54:     return ALIGNOPTIONSTACKHELPER( $\Omega_H, \Gamma, \Omega_\beta, 0$ )
55:   else
56:      $\omega_{k,v,\Phi}^{x,\phi} \leftarrow$  last option in  $\Omega_H$ 
57:      $\text{in\_stack}, \text{stack\_index} \leftarrow$  OPTIONINSTACK( $\omega_{k,v,\Phi}^{x,\phi}, \Gamma$ )
58:     if  $\text{in\_stack}$  then
59:       return ALIGNOPTIONSTACKHELPER( $\Omega_H, \Gamma, \Omega_\beta, \text{stack\_index}$ )
60:   return  $\Omega_H$ 
61: function ALIGNOPTIONSTACKHELPER( $\Omega_H, \Gamma, \Omega_\beta, \text{stack\_index}$ )
62:    $\Omega'_H \leftarrow \Omega_H$ 
63:    $\omega_{i',\Phi}^{j',\phi} \leftarrow$  last option in  $\Omega_\beta$   $\triangleright$  Shallowest terminated option
64:    $\Phi' \leftarrow \Phi$   $\triangleright$  Context initialized from last option
65:   for  $l = \text{stack\_index} \dots |\Gamma| - 1$  do
66:      $\langle u_f, \cdot, M_i, M_j, \phi, \cdot \rangle \leftarrow \Gamma_l$ 
67:      $\phi_{sel} \leftarrow$  Select disjunct from  $\phi$  (e.g., randomly)
68:      $\Omega'_H \leftarrow \Omega'_H \oplus \omega_{i,u_f,\Phi'}^{j,\phi_{sel}}$   $\triangleright$  Append new option to the option stack
69:      $\Phi' \leftarrow \text{DNF}(\Phi' \wedge \phi_{sel})$ 
70:   return  $\Omega'_H$ 

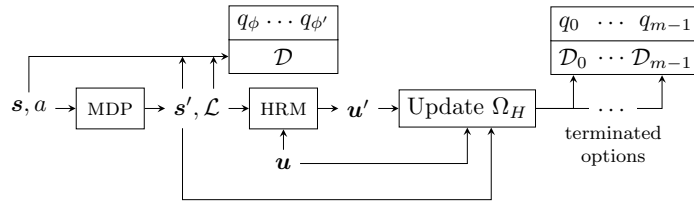
```



(a) **Fill the Option Stack Ω_H .** Call options are iteratively selected until reaching a formula option. When a call option is chosen, the next option is selected from the initial state u_j^0 of the called RM M_j using the accumulated context $\phi \wedge \Phi$.



(b) **Select an Action.** The formula option on the option stack Ω_H determines the action a to execute in state s .



(c) **Apply the Action.** The selected action a is applied in the environment, producing as a result a tuple \mathbf{s}' and a label \mathcal{L} . The tuple $\langle \mathbf{s}, a, \mathbf{s}', \mathcal{L} \rangle$ is added to the buffer \mathcal{D} and the DQNs $q_\phi, \dots, q_{\phi'}$ for the formulas in the HRM are updated. Given the new label and the current hierarchy state \mathbf{u} , the HRM determines the new hierarchy state \mathbf{u}' . The hierarchy states are used to update the option stack Ω_H , the terminated options' experiences are pushed into the corresponding buffers, and the associated DQNs q_0, \dots, q_{m-1} are updated.

Figure 7.1: The core procedures involved in the policy learning algorithm that exploits HRMs.

$\omega_{2,1,\top}^{\top,\wp}$, and $\omega_{0,3,\top}^{\top,\wp}$. The first option should lead the agent to observe the label $\{\ddagger\}$ to satisfy $\ddagger \wedge \neg\wp$. The value functions associated with this set of options are $q_{\ddagger \wedge \neg\wp}$, q_{\wp} , q_{\ddagger} , q_{\wp} and q_{\wp} . Both $\omega_{1,1,\top}^{\top,\wp}$ and $\omega_{2,1,\top}^{\top,\wp}$ are associated with q_{\wp} . Conversely, the call options are $\omega_{0,0,\top}^{1,\neg\wp}$, $\omega_{0,0,\top}^{2,\top}$, $\omega_{0,1,\top}^{2,\top}$, and $\omega_{0,2,\top}^{1,\top}$, where the first one achieves its local goal if formula options $\omega_{1,0,\neg\wp}^{\top,\ddagger}$ and $\omega_{1,1,\top}^{\top,\wp}$ sequentially achieve theirs. The associated value functions are q_0 , q_1 and q_2 . Both $\omega_{0,0,\top}^{2,\top}$ and $\omega_{0,1,\top}^{2,\top}$ are associated with q_2 . Examples 7.1.2 and 7.1.3 describe a few steps of the option selection algorithm in two different scenarios given the grid instance from Figure 6.1.

Example 7.1.2. *In this scenario, we observe what occurs when all chosen options are run to completion (i.e., until their local goals are achieved):*

1. The initial hierarchy state is $\langle M_0, u_0^0, \top, [] \rangle$ and the option stack Ω_H is empty. We select options to fill Ω_H . The first option is chosen from u_0^0 in M_0 using a policy induced by q_0 . At this state, the available options are $\omega_{0,0,\top}^{1,\neg\wp}$ and $\omega_{0,0,\top}^{2,\top}$. Let us assume that the former is chosen. Then an option from the initial state of M_1 under context $\neg\wp$ is chosen, which can only be $\omega_{1,0,\neg\wp}^{\top,\ddagger}$. Since this option is a formula option (the call is made to M_\top), no more options are selected and the option stack is $\Omega_H = \langle \omega_{0,0,\top}^{1,\neg\wp}, \omega_{1,0,\neg\wp}^{\top,\ddagger} \rangle$.
2. The agent selects options according to the formula option in Ω_H , $\omega_{1,0,\neg\wp}^{\top,\ddagger}$, whose policy is induced by $q_{\ddagger \wedge \neg\wp}$. Let us assume that the policy tells the agent to rotate right. Since the label at this location is empty, the hierarchy state remains the same; therefore, no options terminate, and the option stack does not change.
3. Let us assume that the agent moves forward twice, thus observing $\{\ddagger\}$. The hierarchy state then becomes $\langle M_1, u_1^1, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\wp, \top \rangle] \rangle$ (see Example 6.1.5 for a step-by-step application of the hierarchical transition function). We check which options in Ω_H have terminated starting from the last chosen one. The formula option $\omega_{1,0,\neg\wp}^{\top,\ddagger}$ terminates because the hierarchy state has changed. In contrast, the call option $\omega_{0,0,\top}^{1,\neg\wp}$ does not terminate since there is an item in the call stack, $\langle u_0^0, u_0^1, M_0, M_1, \neg\wp, \top \rangle$, that it can be mapped to (i.e., the option is still running).
4. An experience $\langle \mathbf{s}, \omega_{1,0,\neg\wp}^{\top,\ddagger}, \mathbf{s}' \rangle$ is formed for the terminated option, where \mathbf{s} and \mathbf{s}' are the observed tuples on initiation and termination respectively. This tuple is added to the replay buffer associated with the RM where the option appears, \mathcal{D}_1 , since it achieved its goal (i.e., a label that satisfied $\ddagger \wedge \neg\wp$ was observed).
5. We align Ω_H with the new stack. In this case, Ω_H remains unchanged since its only option can be mapped into an item of the new stack.
6. We start a new step. Since the option stack does not contain a formula option, we select new options from the current hierarchy state according to a policy induced by q_1 . In this case, there is a single eligible option: $\omega_{1,1,\top}^{\top,\wp}$.

Example 7.1.3. *In this scenario, we observe what occurs when the HRM traversal differs from the options chosen by the agent:*

1. The initial step is like the one in Example 7.1.2, but we assume $\omega_{0,0,\top}^{2,\top}$ is selected instead. Then, since this is a call option, an option from the initial state of M_2 under context \top is chosen, which can only be $\omega_{2,0,\top}^{\top,\wp}$. The option stack thus becomes $\Omega_H = \langle \omega_{0,0,\top}^{2,\top}, \omega_{2,0,\top}^{\top,\wp} \rangle$.

2. *Let us assume that by taking actions according to $\omega_{2,0,\top}^{\top,\forall}$ we end up observing $\{\clubsuit\}$. Like in Example 7.1.2, the hierarchy state becomes $\langle M_1, u_1^1, \top, [\langle u_0^0, u_0^1, M_0, M_1, \neg\forall, \top \rangle] \rangle$. We check which options in Ω_H have terminated. The formula option $\omega_{2,0,\top}^{\top,\forall}$ terminates since the hierarchy state has changed, and the call option $\omega_{0,0,\top}^{2,\top}$ also terminates since it cannot be mapped into an item of the call stack. Intuitively, these options should finish since the HRM is being traversed through a path different from that chosen by the agent.*
3. *The replay buffers are not updated for these options since they have not achieved their local goals.*
4. *The option stack Ω_H is aligned with the updated call stack. The only item of the stack $\langle u_0^0, u_0^1, M_0, M_1, \neg\forall, \top \rangle$ can be mapped into option $\omega_{0,0,\top}^{1,\neg\forall}$. We assume that this option starts on the same tuple \mathbf{s} and that it has run for the same number of steps as the last terminated option $\omega_{0,0,\top}^{2,\top}$.*

7.1.3 Implementation

We describe two strategies for efficiently updating the value functions for the formulas appearing in the hierarchies.

Action-Value Function Update Regime

For efficiency, the entire set of action-value functions associated with formula options is not updated after each step; instead, only some of these functions are updated. Both approaches lead to similar convergence rates, but the latter is less costly. To determine the set of value functions to update, we keep (i) an update counter c_ϕ for each function q_ϕ and (ii) a global counter c (i.e., the total number updates), and compute the probability p_ϕ of choosing q_ϕ for an update as:

$$p_\phi = \frac{s_\phi}{\sum_{\phi'} s_{\phi'}}, \text{ where } s_\phi = c - c_\phi - 1.$$

A fixed number of value functions are selected without replacement using the resulting probability distribution, which prioritizes value functions with fewer updates (intuitively, the estimates produced by such functions are less precise and require more training). Notably, only the value functions in the current HRM are considered for selection.

The Formula Tree

Each formula option's policy is induced by an action-value function associated with a formula, as described in Section 7.1.1. In domains where certain proposition sets cannot occur, it is unnecessary to consider formulas that cover some of these sets. For instance, in a domain where two propositions a and b cannot be simultaneously observed (i.e., it is impossible to observe $\{a, b\}$), formulas such as $a \wedge \neg b$ or $b \wedge \neg a$ could instead be represented by the more abstract formulas a or b ; therefore, $a \wedge \neg b$ and a could be both associated with an action-value function q_a , whereas $b \wedge \neg a$ and b could be both associated with an action-value function q_b . Reducing the number of value functions makes learning more efficient, especially when only a subset of them are updated, as explained at the beginning of this section.

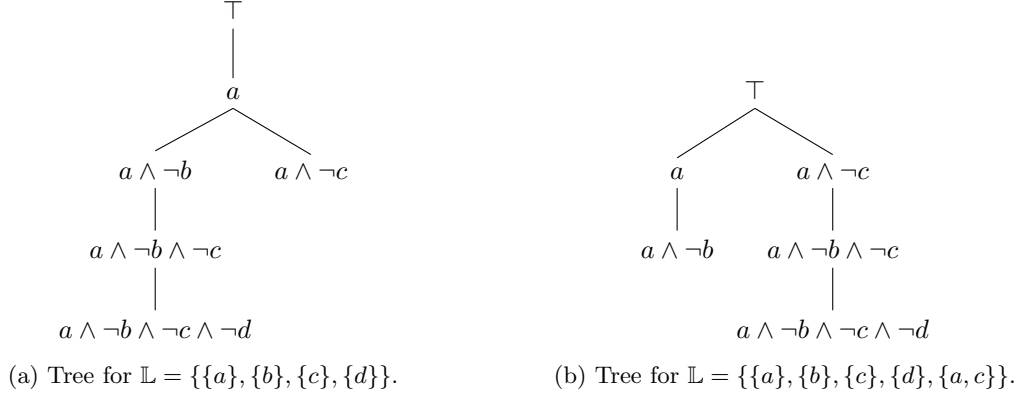


Figure 7.2: Examples of formula trees for different sets of labels. The node $a \wedge \neg b \wedge \neg c$ in (a) could have been a child of $a \wedge \neg c$ instead (the parent depends on the insertion order).

We represent relationships between formulas using a *formula tree*, which arranges a set of formulas in a tree structure. Formally, given a set of propositions \mathcal{P} , a formula tree is a tuple $\langle \mathcal{F}, F_r, \mathbb{L} \rangle$, where \mathcal{F} is a set of nodes, each associated with a formula; $F_r \in \mathcal{F}$ is the root of the tree and it is associated with the formula \top ; and $\mathbb{L} \subseteq (2^{\mathcal{P}})^*$ is a set of labels. All the nodes in the tree except for the root are associated with conjunctions. Let $\nu(X) \subseteq 2^{\mathcal{P} \cup \bar{\mathcal{P}}}$, where $\bar{\mathcal{P}} = \{\neg p \mid p \in \mathcal{P}\}$, denote the set of literals of a formula X ; for instance, if $X = a \wedge \neg b$, then $\nu(X) = \{a, \neg b\}$. A formula X *subsumes* a formula Y if (1) $X = \top$, or (2.i) $\nu(X) \subseteq \nu(Y)$ and (2.ii) for all labels $\mathcal{L} \in \mathbb{L}$, either $\mathcal{L} \models X$ and $\mathcal{L} \models Y$, or $\mathcal{L} \not\models X$ and $\mathcal{L} \not\models Y$. Case (2) indicates that Y is a special case of X (it adds literals but is satisfied by exactly the same labels). The tree is organized such that the formula at a given node subsumes all its descendants. The children of the root determine the set of action-value functions.

During the agent-environment interaction, the formula tree is updated if (i) a new formula appears in the learned HRMs, or (ii) a new label is observed. Algorithm 3 contains the pseudo-code for updating the tree in these two cases. When a new formula is added (line 1), we create a node for the formula (line 2) and add it to the tree. The insertion place is determined by exploring the tree top-down from the root F_r (lines 3–19). First, we check whether a child of the current node subsumes the new node (line 7). If such a node exists, then we go down this path (lines 8–9); otherwise, the new node is going to be a child of the current node (lines 16–17). In the latter case, in addition, all those children nodes of the current node that are subsumed by the new node need to become children of the new node (lines 11–15). The other core case in which the tree may need an update occurs when a new label is observed (lines 20–25) since we need to make sure that parenting relationships comply with the set of labels \mathbb{L} . First, we find nodes inconsistent with the new label: a parenting relationship is broken (line 39) when the formula of the parent non-root node is satisfied by the label but the formula of the child node is not (or vice versa). Once the inconsistent nodes are found, we remove their current parenting relationship (lines 45–46) and reinsert them (line 47).

Example 7.1.4. Figure 7.2 shows two simple examples of formula trees. The formula tree in (b) results from adding the label $\{a, c\}$ to the label set: the formula a stops subsuming $a \wedge \neg c$ (i.e., their truth values differ for the new label), and the latter hence becomes a child of the root. The action-value functions are q_a in (a), and q_a and $q_{a \wedge \neg c}$ in (b).

Algorithm 3 Formula tree operations

Input: a formula tree $\langle \mathcal{F}, F_r, \mathbb{L} \rangle$, where \mathcal{F} is a set of nodes, $F_r \in \mathcal{F}$ is the root node (associated with the formula \top), and \mathbb{L} is a set of labels.

```

1: function ADDFORMULA( $f$ )
2:   ADDNODE(CREATENODE( $f$ ))
3: function ADDNODE(new_node)
4:   current_node  $\leftarrow F_r$ 
5:   added_node  $\leftarrow \perp$ 
6:   while added_node =  $\perp$  do
7:     child_node  $\leftarrow$  FINDSUBSUMINGCHILD(current_node, new_node)
8:     if child_node  $\neq$  nil then ▷ Keep exploring down this path
9:       current_node  $\leftarrow$  child_node
10:    else ▷ Insert the node
11:      subsumed_children  $\leftarrow$  GETSUBSUMEDCHILDREN(current_node, new_node)
12:      new_node.children  $\leftarrow$  new_node.children  $\cup$  subsumed_children
13:      for child  $\in$  subsumed_children do
14:        current_node.children  $\leftarrow$  current_node.children  $\setminus$  {child}
15:        child.parent  $\leftarrow$  new_node
16:      current_node.children  $\leftarrow$  current_node.children  $\cup$  {new_node}
17:      new_node.parent  $\leftarrow$  current_node
18:      added_node  $\leftarrow \top$ 
19:    $\mathcal{F} \leftarrow \mathcal{F} \cup \{\text{new\_node}\}$ 
20: function ONLABEL( $\mathcal{L}$ )
21:    $\mathbb{L} \leftarrow \mathbb{L} \cup \{\mathcal{L}\}$ 
22:   inconsistent_nodes  $\leftarrow \{\}$ 
23:   for child  $\in F_r$ .children do
24:     FINDINCONSISTENTNODES(child,  $\mathcal{L}$ , inconsistent_nodes)
25:   REINSERTINCONSISTENTNODES(inconsistent_nodes)
26: function FINDSUBSUMINGCHILD(current_node, new_node)
27:   for child  $\in$  current_node.children do
28:     if child.formula subsumes new_node.formula then
29:       return child
30:   return nil
31: function GETSUBSUMEDCHILDREN(current_node, new_node)
32:   subsumed_children  $\leftarrow \{\}$ 
33:   for child  $\in$  current_node.children do
34:     if new_node.formula subsumes child.formula then
35:       subsumed_children  $\leftarrow$  subsumed_children  $\cup$  {new_node}
36:   return subsumed_children
37: function FINDINCONSISTENTNODES(current_node,  $\mathcal{L}$ , inconsistent_nodes)
38:   for child  $\in$  current_node.children do
39:     if  $\mathcal{L} \models \text{current\_node.formula} \oplus \mathcal{L} \models \text{child.formula}$  then
40:       inconsistent_nodes  $\leftarrow$  inconsistent_nodes  $\cup$  {child}
41:     else
42:       FINDINCONSISTENTNODES(child,  $\mathcal{L}$ , inconsistent_nodes)
43: function REINSERTINCONSISTENTNODES(inconsistent_nodes)
44:   for node  $\in$  inconsistent_nodes do
45:     node.parent.children  $\leftarrow$  node.parent.children  $\setminus$  {node}
46:     node.parent  $\leftarrow$  nil
47:     ADDNODE(node)

```

7.2 Learning Hierarchies of Reward Machines from Traces

In this section, we formalize the task of learning a hierarchy of reward machines from traces (Section 7.2.1) and introduce a method for solving this task using ILASP (Section 7.2.2).

7.2.1 The Hierarchy of Reward Machines Learning Task

We formalize the task of learning a hierarchy of reward machines from traces as learning the root of a hierarchy given a set of callable RMs; that is, we look for a way of hierarchically composing existing RMs such that the learned HRM is valid with respect to a set of traces.

Definition 7.2.1 (HRM learning task). *An HRM learning task is a tuple $T_H = \langle r, \mathcal{U}, \mathcal{P}, \mathcal{M}, \mathcal{M}_C, u^0, u^A, u^R, \Lambda, \kappa \rangle$, where*

- r is the index of the root RM in the HRM;
- $\mathcal{U} \supseteq \{u^0, u^A, u^R\}$ is a set of states of the root RM, which always contains an initial state u^0 , an accepting state u^A , and a rejecting state u^R ;
- \mathcal{P} is a set of propositions;
- $\mathcal{M} \supseteq \{M_\top\}$ is a set of RMs that always includes the leaf RM;
- $\mathcal{M}_C \subseteq \mathcal{M}$ is a set of callable RMs;
- $\Lambda = \Lambda^G \cup \Lambda^D \cup \Lambda^I$ is a set of traces; and
- κ is the maximum number of directed edges from a state $u \in \mathcal{U}$ to another state $u' \in \mathcal{U} \setminus \{u\}$.

An HRM $H = \langle \mathcal{M} \cup \{M_r\}, M_r, \mathcal{P} \rangle$ is a solution of T_H if and only if it is valid with respect to all the traces in Λ ; that is, if and only if it accept all goal traces in Λ^G , rejects all dead-end traces in Λ^D , and does not accept nor reject any incomplete trace in Λ^I .

The HRM learning task differs from the RM learning task (see Definition 4.2.1) in that (i) the learned RM is associated with an index r , and (ii) the learned RM calls other RMs. Following Assumption 6.3.1, without loss of generality, the learned RM has a single accepting state and a single rejecting state.³ We also make the following assumptions about the set of RMs \mathcal{M} to ensure the resulting HRM is well-formed.

Assumption 7.2.1. *All RMs reachable from RMs in \mathcal{M}_C are in \mathcal{M} .*

Assumption 7.2.2. *All RMs in \mathcal{M} are deterministic.*

Assumption 7.2.3. *All RMs in \mathcal{M} are defined over the same proposition set \mathcal{P} or a subset of it.*

³In practice, as described in Section 4.2.1, the accepting (resp. rejecting) state is not considered unless the set of goal (resp. dead-end) traces is non-empty.

7.2.2 Solving the Hierarchy of Reward Machines Learning Task with ILASP

Given an HRM learning task T_H , we map it into an ILASP learning task $\mathbb{A}(T_H) = \langle \mathcal{B}, \mathcal{S}_{\mathfrak{M}}, \langle \mathcal{E}^+, \emptyset \rangle \rangle$ and use the ILASP system to find an inductive solution $\mathbb{A}_{\varphi}(M_r) \subseteq \mathcal{S}_{\mathfrak{M}}$ that covers the examples. Like in Section 4.2.2, we do not use *negative examples* ($\mathcal{E}^- = \emptyset$). We define the components of $\mathbb{A}(T_H)$ below.

Background Knowledge

The background knowledge $\mathcal{B} = \mathcal{B}_{\mathcal{U}} \cup \mathcal{B}_{\mathcal{M}} \cup \mathcal{R}$ is a set of rules that describe the behavior of the HRM. The set $\mathcal{B}_{\mathcal{U}}$ consists of **state**(u, M_r) facts for each state $u \in \mathcal{U}$ of the root RM with index r we aim to induce, whereas $\mathcal{B}_{\mathcal{M}} = \bigcup_{M_i \in \mathcal{M} \setminus \{M_r\}} \mathbb{A}(M_i)$ contains the ASP representations of all RMs. Finally, \mathcal{R} is the set of general rules introduced in Section 6.3.2 defining how HRMs are traversed by a trace. Importantly, the index of the root r in these rules must correspond to the one used in T_H .

Hypothesis Space

The hypothesis space $\mathcal{S}_{\mathfrak{M}}$ contains all **call** and $\bar{\varphi}$ rules that characterize a transition from a non-terminal state $u \in \mathcal{U} \setminus \{u^A, u^R\}$ to a different state $u' \in \mathcal{U} \setminus \{u\}$ using edge $e \in \{1, \dots, \kappa\}$ by calling RM $M \in \mathcal{M}_{\mathcal{C}}$. Formally, it is defined as

$$\mathcal{S}_{\mathfrak{M}} = \left\{ \begin{array}{l} \text{call}(u, u', e, M_r, M). \\ \bar{\varphi}(u, u', e, M_r, T) :- \text{prop}(p, T), \text{step}(T). \\ \bar{\varphi}(u, u', e, M_r, T) :- \text{not prop}(p, T), \text{step}(T). \end{array} \middle| \begin{array}{l} u \in \mathcal{U} \setminus \{u^A, u^R\}, \\ u' \in \mathcal{U} \setminus \{u\}, e \in \{1, \dots, \kappa\}, \\ M \in \mathcal{M}_{\mathcal{C}}, p \in \mathcal{P} \end{array} \right\}.$$

Learning the negation $\bar{\varphi}$ of the logical transition function φ offers more flexibility than learning the latter directly. We refer the reader to the analogous subsection in Section 4.2.2 for specific details.

Example Sets

The set of positive examples \mathcal{E}^+ is defined as in Section 4.2.2.

Correctness of the Learning Task

The following theorem captures the correctness of the HRM learning task.

Theorem 7.2.1. *Given an HRM learning task $T_H = \langle r, \mathcal{U}, \mathcal{P}, \mathcal{M}, \mathcal{M}_{\mathcal{C}}, u^0, u^A, u^R, \Lambda, \kappa \rangle$, an HRM $H = \langle \mathcal{M} \cup \{M_r\}, M_r, \mathcal{P} \rangle$ is a solution of T_H if and only if $\mathbb{A}_{\varphi}(M_r)$ is an inductive solution of $\mathbb{A}(T_H) = \langle \mathcal{B}, \mathcal{S}_{\mathfrak{M}}, \langle \mathcal{E}^+, \emptyset \rangle \rangle$.*

Proof. Assume H is a solution of T_H .

$\iff H$ is valid with respect to all traces in Λ (i.e., H accepts all traces in Λ^G , rejects all traces in Λ^D and does not accept nor reject any trace in Λ^I).

\iff By Proposition 6.3.1, for each trace $\lambda^* \in \Lambda^*$ where $*$ $\in \{G, D, I\}$, $\mathbb{A}(H) \cup \mathcal{R} \cup \mathbb{A}(\lambda^*)$ has a unique answer set A and (i) **accept** $\in A$ if and only if $*$ $= G$, and (ii) **reject** $\in A$ if and only if $*$ $= D$.

\iff For each example $e \in \mathcal{E}^+$, $\mathcal{R} \cup \mathbb{A}(H)$ accepts e .

\iff For each example $e \in \mathcal{E}^+$, $\mathcal{B} \cup \mathbb{A}_\varphi(M_r)$ accepts e . The two programs are identical:

$$\begin{aligned}
\mathcal{R} \cup \mathbb{A}(H) &= \mathcal{R} \cup \mathbb{A}(H) \\
&= \mathcal{R} \cup \bigcup_{M_i \in \mathcal{M} \setminus \{M_\top\}} \mathbb{A}(M_i) \\
&= \mathcal{R} \cup \bigcup_{M_i \in \mathcal{M} \setminus \{M_\top, M_r\}} \mathbb{A}(M_i) \cup \mathbb{A}(M_r) \\
&= \mathcal{R} \cup \mathcal{B}_\mathcal{M} \cup \mathbb{A}_\mathcal{U}(M_r) \cup \mathbb{A}_\varphi(M_r) \\
&= \mathcal{R} \cup \mathcal{B}_\mathcal{M} \cup \mathcal{B}_\mathcal{U} \cup \mathbb{A}_\varphi(M_r) \\
&= \mathcal{B} \cup \mathbb{A}_\varphi(M_r).
\end{aligned}$$

$\iff \mathbb{A}_\varphi(M_r)$ is an inductive solution of $\mathbb{A}(T_H)$. □

Enforcement of Structural Properties

The hypothesis space of the HRM learning task presented above potentially considers symmetric solutions as well as solutions with non-deterministic roots. Discarding symmetric solutions from the hypothesis space enables a faster search, whereas ruling out non-deterministic roots results in well-formed HRMs. Analogously to the learning methodology for regular RMs (see Section 4.2.2), we enforce the determinism and symmetry breaking constraints from Sections 6.3.4–6.3.5 during the search by including them (along with the mapping to a factual representation) in the learning task through meta-program injection (Law et al., 2018).

The following constraints are also enforced in the learning task to learn sensible HRMs. The first rule prevents an edge from being labeled with calls to two different RMs. The second rule prevents edges from being labeled with the same literal positively and negatively.

$$\left\{ \begin{array}{l} \text{:- call}(X, Y, E, M, M2), \text{call}(X, Y, E, M, M3), M2 \neq M3. \\ \text{:- pos}(X, Y, E, M, P), \text{neg}(X, Y, E, M, P). \end{array} \right\}$$

7.3 Interleaved Learning

We here describe LHRM, a method that *interleaves* the exploitation of HRMs with their learning from interaction. We consider a *multi-task* setting. Given a set of tasks \mathbb{T} and a set of instances \mathbb{I} (e.g., grids) of an environment, the agent learns (i) an HRM for each task using traces from several instances, and (ii) policies that generalize to the different task-instance pairs. Namely, the agent interacts with $\mathbb{T} \times \mathbb{I}$ labeled MDPs. The learning proceeds from simpler to harder tasks such that HRMs for the latter build on the former. We make the following assumptions on the MDPs.

Assumption 7.3.1. *All MDPs share propositions \mathcal{P} and actions \mathcal{A} , and those defined on a given instance share states \mathcal{S} and labeling function l .*

Assumption 7.3.2. *Dead-end histories are determined by the last state only.*

Assumption 7.3.3. *The root’s height of a task’s HRM (or task level, for brevity) is known.*

Assumption 7.3.1 resembles the experimental setup described in Section 5.1.1; indeed, likewise, while the termination function differs for each task-instance pair since it depends on history, the traces across instances for a given task are commonly categorized. Assumption 7.3.2 emerges from Equation 7.1, which implicitly makes this assumption by setting the discounted term to 0 when the history is a dead-end history. When the assumption does not hold, learning might become unstable since the discounted term is not always canceled. By this assumption, the dead-end histories for a given instance are common across tasks, which stabilizes policy learning since the discounted term is always canceled under the same condition.

In what follows, we introduce the building blocks of LHRM. First, we describe a curriculum learning methodology for learning the HRMs from lower to higher levels (Section 7.3.1). Then, we describe how the exploitation of the HRMs is interleaved with their learning (Section 7.3.2).

7.3.1 Curriculum Learning

LHRM learns the tasks' HRMs from lower to higher levels following a curriculum learning (Bengio et al., 2009) method due to Pierrot et al. (2019). By Assumption 7.3.3, the level of each task is known; for instance, Table 6.1 shows the level of each CRAFTWORLD task. Formally, let $f : \mathbb{T} \rightarrow \mathbb{N}$ be a function that maps each task to a level, and $\iota \in \mathbb{N}$ be the currently active level. For clarity, we provide a high-level description of the method followed by an explanation of its two key steps.

Overview

Before starting an episode, LHRM selects an MDP \mathbb{M}_{ij} , where $\langle i, j \rangle \in \mathbb{T} \times \mathbb{I}$. Initially, only level 1 tasks can be chosen. The probability of selecting an MDP \mathbb{M}_{ij} is determined by an estimate of the average undiscounted return \bar{R}_{ij} obtained by the agent, with lower returns mapped to higher probabilities. When the minimum average return across MDPs up to the current level ι surpasses a given threshold, ι increases by 1, ensuring the learned HRMs and their associated policies are reusable in higher level tasks.

Task-Instance Selection

The selection of a labeled MDP \mathbb{M}_{ij} at the beginning of an episode is performed by first selecting a task $i \in \mathbb{T}$ and then selecting an instance $j \in \mathbb{I}$. The probabilities used in each choice are determined by computing a score $c_{ij} = 1 - \bar{R}_{ij}$ for each task-instance pair based on the estimated average undiscounted returns. This scoring function (Andreas et al., 2017) assumes that the returns range between 0 and 1, which is true in our setting by Assumption 6.1.5. Higher probabilities are assigned to higher scores; hence, tasks and instances where the agent performs poorly are more likely to be selected.

The probability p_i of choosing a task $i \in \mathbb{T}$ is computed as

$$p_i = \begin{cases} \frac{\max_{j \in \mathbb{I}} c_{ij}}{\sum_{k \in \mathbb{T} | f(k) \leq \iota} \max_{j \in \mathbb{I}} c_{kj}} & \text{if } f(i) \leq \iota; \\ 0 & \text{otherwise.} \end{cases}$$

That is, the probability is non-zero only if the level of i , $f(i)$, is lower or equal than the currently

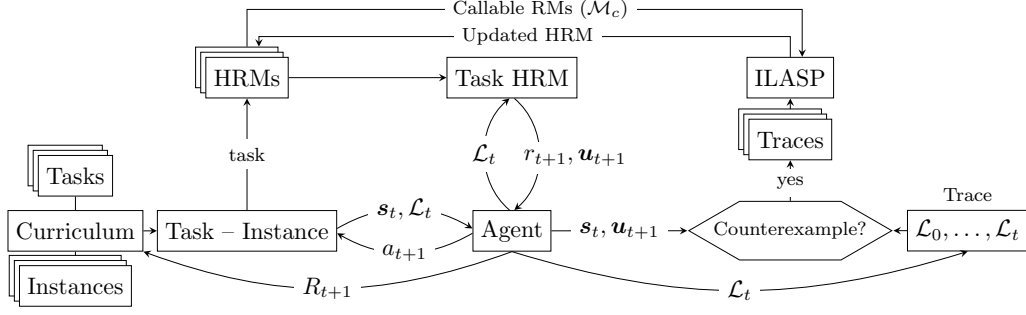


Figure 7.3: Overview of the LHRM algorithm (see main text for a description).

active level. For each task in a level lower or equal than ι , the lowest score across instances is taken as a representative of the agent’s performance for that task, and higher probabilities are assigned to tasks for which there is some instance where the agent performs worse.

Having selected a task $i \in \mathbb{T}$, the probability q_j of choosing an *instance* $j \in \mathbb{I}$ is computed similarly:

$$q_j = \frac{c_{ij}}{\sum_{k \in \mathbb{I}} c_{ik}}.$$

That is, instances where the agent performs worse are more likely to be chosen.

Return Estimate Update

The average undiscounted return \bar{R}_{ij} for each labeled MDP M_{ij} such that $f(i) \leq \iota$ (i.e., an MDP whose corresponding task is in an active level) is periodically updated as $\bar{R}_{ij} \leftarrow \beta \bar{R}_{ij} + (1 - \beta)R$, where $\beta \in [0, 1]$ is a hyperparameter and R is the undiscounted return obtained by the greedy policy in a single evaluation episode.

7.3.2 Interleaving Algorithm

LHRM *interleaves* the learning of HRMs with their exploitation akin to ISA (see Section 4.3). Learning an HRM consists of learning its root, as described in Section 7.2; therefore, given a set of tasks, LHRM essentially performs ISA for each task in the set and enables reusing previously learned RMs within other RMs through calls. Given the similarity between LHRM and ISA, we focus on providing a high-level overview of the algorithm, describing some optimizations for enhancing the algorithm’s performance and restating a property of ISA that also characterizes LHRM.

Overview

Figure 7.3 illustrates the core blocks of the algorithm. Given a set of tasks and a set of instances, the curriculum selects a task-instance MDP at the start of an episode, and the HRM for the chosen task is taken from the bank of HRMs. Initially, the HRM’s root of each task consists of 3 states (the initial, accepting, and rejecting states) and neither accepts nor rejects anything. At each step, the agent observes a tuple s_t and a label L_t from the task-instance MDP, and performs an action a_{t+1} . The label is used to (i) determine the next hierarchy state u_{t+1} and the reward r_{t+1} , and (ii) update the trace $\langle L_0, \dots, L_t \rangle$. When the current HRM is not valid with respect to the trace (i.e., the trace

is a counterexample), a new HRM is learned using ILASP. By default, the learned root RM can call any RM from a lower level task. If an HRM that covers the observed counterexamples cannot be learned, the number of states in the root increases by 1; hence, the root RM is guaranteed to be *minimal* for a specific value of κ . When an HRM for a task $i \in \mathbb{T}$ is learned, the current level is set to that of i , and the estimated returns \bar{R}_{ij} are reset to 0 for all instances $j \in \mathbb{I}$; therefore, returns depend on the current HRM only, ensuring that the level only increases once the policies associated with HRMs in active levels are effective.⁴ The curriculum, as described in Section 7.3.1, is periodically updated with undiscounted episode returns.

Properties

Theorem 4.3.1, which states that there will only be a *finite number of RM learning steps* before ISA converges to a target RM (or an equivalent one), is applicable to LHRM since the hypothesis space is still finite (i.e., the number of possible root RMs is finite even though other RMs can now be called). We emphasize that the hypothesis space is not constrained by the set of callable RMs as long as all propositions appearing in the traces can be used to label calls to the leaf RM. Furthermore, as described in algorithm overview, the learned root is guaranteed to be *minimal* for a specific value of κ since an iterative deepening strategy on the number of states is followed.

Optimizations

The following are some optimizations we introduce to enhance LHRM performance.

Learning Initial HRMs from Trace Sets. ISA learns a first RM from a single (counterexample) trace, while other contemporary RM learning methods (Toro Icarte et al., 2019; Xu et al., 2020; Hasanbeig et al., 2021) learn a first RM from a set of traces. LHRM follows the latter strategy to learn a first HRM with a slight change: it starts collecting a set of ρ goal traces and, once they are all collected, it employs the ρ_s shortest ones to learn the HRM.

Exploration with Options. LHRM leverages options from lower level HRMs to collect goal traces for learning the first HRM; specifically, if no option is being run, an option is selected uniformly at random and performed greedily until it terminates. Options enable exploring the environment more efficiently than by performing random walks with primitive actions; thus, they ease the observation of goal traces, especially in tasks where achieving the goal is challenging.

7.4 Summary

In this chapter, we introduced algorithms that leverage the composability enabled by hierarchies of reward machines. First, we presented a method for exploiting the hierarchies using the options framework. The formulas and RMs in the hierarchy constituted independently solvable (and, hence, reusable) subtasks. Second, we described a method for learning an HRM from traces using the ILASP inductive logic programming system. Third, and finally, we devised LHRM, a curriculum-based algorithm that interleaves the presented exploitation and learning methods. Given a list of

⁴Resetting the returns to 0 is essential when the level of i is lower than the current level; otherwise, even if the current level is set to that of i , it is immediately increased since the (unchanged) returns remain above the threshold.

tasks, LHRM learns an HRM for each task such that HRMs for complex tasks reuse those learned for simpler tasks.

Chapter 8

Evaluation of Hierarchies of Reward Machines

In this chapter, we evaluate the policy and HRM learning approaches described in Chapter 7. First, we describe the common experimental setup across experiments, including the employed domains and how results are reported (Section 8.1). Second, we show that interleaving the learning and exploitation of HRMs produces more effective policies than memoryless and LSTM-based approaches (Section 8.2). Third, we analyze how learning HRMs compares to learning RMs using different approaches (Section 8.3). Finally, we examine whether learning policies in handcrafted HRMs results in an improvement over learning policies in handcrafted RMs (Section 8.4). The code is available at <https://github.com/ertsiger/hrm-learning>.


8.1 Experimental Setup

This section describes our experimental setup by first introducing the domains we consider (Section 8.1.1). Next, we enumerate the hyperparameters of our approach and some of the baselines, and explain some of the restrictions on the HRM learning process (Section 8.1.2). Finally, we detail how the results are reported (Section 8.1.3).

8.1.1 Domains

We describe the domains used in our evaluation, the instance types we consider and how they are generated, the tasks for which HRMs will be learned and exploited, and the associated network architectures.

CRAFTWORLD

The specification of this domain is described in Section 6.1. The implementation is based on MiniGrid (Chevalier-Boisvert et al., 2023), thus inheriting many of its features. At each step, the agent observes a $W \times H \times 3$ tensor, where W and H are the width and height of the grid. The three channels contain the object IDs, the color IDs, and object state IDs (including the agent’s orientation), respectively. Each object we define (except for the lava , which already existed in MiniGrid) has

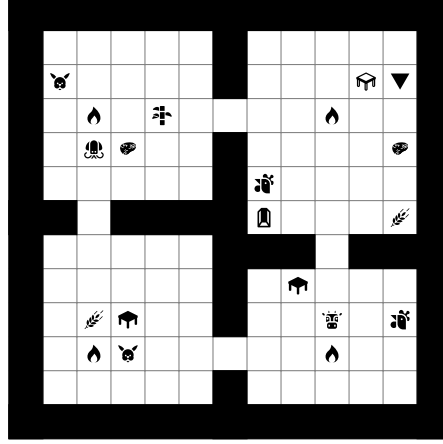


Figure 8.1: An instance of the CRAFTWORLD grid in the FRL setting.

its own object and color IDs. Before providing the agent with the state, the content of all matrices is scaled between -1 and $+1$.

Instance Types. We consider four grid types: open plan 7×7 grids (OP, Figure 6.1), open plan 7×7 grids with a lava location (OPL), 13×13 grids with four rooms (FR; Sutton et al., 1999), and 13×13 grids with four rooms and a lava location per room (FRL, Figure 8.1). The lava must be avoided.

Instance Generation. The grids are randomly generated. In all settings (OP, OPL, FR, FRL), the agent and the objects are randomly assigned an unoccupied position. In the case of FR and FRL, no object occupies a position between rooms or its adjoining positions. There is a single object per object type (i.e., proposition) in OP and OPL, whereas there can be one or two per type in FR and FRL. Finally, there is a single randomly placed lava location in OPL (like the rest of the propositions); in contrast, in FRL, there are four fixed lava locations placed in the intersections between doors, as shown in Figure 8.1.

Tasks. The tasks we consider are listed in Table 6.1. We refer the reader to Appendix B.1 for an illustration of the HRMs for each task.

Network Architectures. The networks are DDQNs consisting of a 3-layer convolutional neural network (CNN) with 16, 32, and 32 filters, respectively. All kernels are 2×2 and use a stride of 1. In FR and FRL instances, there is a max pooling layer with kernel size 2×2 after the first convolutional layer. This part of the architecture is based on that by Igl et al. (2019) and Jiang et al. (2021), who also employ MiniGrid using the full view of the grid. The remainder of the DQN depends on the policy type (see Section 7.1.1):

- In formula-associated DQNs, the CNN’s output is fed to a 3-layer multilayer perceptron (MLP) where the hidden layer has 256 rectifier units and the output layer has a single output for each action.

Table 8.1: List of WATERWORLD tasks. Descriptions follow the nomenclature in Table 6.1.

Task	h	Description	Task	h	Description
RG	1	$r ; g$	RG&MY	2	RG & MY
BC	1	$b ; c$	RGB	2	RG ; b
MY	1	$m ; y$	CMY	2	$c ; MY$
RG&BC	2	RG & BC	RGB&CMY	3	RGB & CMY
BC&MY	2	BC & MY			

- In RM-associated DQNs, the output of the CNN is extended with the encoding of the RM state and the context (see Section 7.1.1) before being fed to a 3-layer MLP where the hidden layer has 256 rectifier units, and the output layer has a single output for each call in the RM.

WATERWORLD

The specification and implementation of this domain are described in Section 5.4.

Instance Types. There are two types of instances: without dead-ends (WOD) and with dead-ends (WD). The former follows the standard WATERWORLD specification given in Section 5.4. In contrast, the latter includes two additional balls of a new color that must be avoided; therefore, the new color incurs a new proposition, and dead-end histories are observable.

Instance Generation. Both WOD and WD instances are generated as described in Section 5.4.1.

Tasks. The tasks we consider are listed in Table 8.1. We emphasize two important aspects regarding their specification:

- Subtasks in a sequence cannot be completed simultaneously, unlike those in Section 5.4.2. For instance, let us consider the RG task and the trace $\lambda = \langle \{\}, \{r, g\} \rangle$. Following the specification from Section 5.4.2, λ is a goal trace. However, λ is not a goal trace here since r and g must be satisfied at different steps; therefore, λ must be extended with a label containing g to become a goal trace, e.g. $\langle \{\}, \{r, g\}, \{r, g\} \rangle$ and $\langle \{\}, \{r, g\}, \{r\}, \{g\} \rangle$ are goal traces for RG.

This specification feature ensures that the HRMs for tasks with $h > 2$ can be represented in terms of independent lower level HRMs. For example, under the specification from Section 5.4.2, the label $\{r, g, b\}$ completes subtask RG and partially completes BC in a single step, which does not enable reusing the HRMs for these subtasks; that is, the HRM would need to be flat with a single transition from the initial state labeled $r \wedge g \wedge b \wedge \neg c$.

- Subtasks that can be performed in any order must be such that they do not take precedence over each other. Let us consider the task RG&BC and a label $\mathcal{L} \supseteq \{r, b\}$ to exemplify it. Upon observing \mathcal{L} , neither the subtask RG nor the subtask BC should be started; that is, one is not prioritized to start over the other. Call contexts break these preferences; for instance, RG requires not to observe b to be started and, likewise, BC requires not to observe r to be started.

We refer the reader to Appendix B.1 for an illustration of the HRMs for each task.

Network Architectures. The architectures for WATERWORLD are a simple modification of the ones described in Section 5.4.3. Formula-associated DQNs consist of a 5-layer MLP, where each of the 3 hidden layers has 512 rectifier units; in contrast, RM-associated networks share the same architecture but, like in CRAFTWORLD, the input consists of the state from the environment and encodings for the RM state and the context.

8.1.2 Hyperparameters and Restrictions

We here enumerate and describe the hyperparameters in our evaluation, and introduce some practical constraints on the learnable HRMs.

Hyperparameters

Table 8.2 lists the hyperparameters for our approach and other methods (DRQN, CRM) used in the evaluation.

The HRMs are learned using ILASP2. The calls to the underlying ASP solver are made using the argument `--opt-mode=ignore`, which reduces the search time since no optimization is made upon calling ILASP; hence, non-optimal solutions might be learned. Non-optimal solutions may be non-minimal, i.e. a subset of the returned solution may also be a solution; consequently, the learned root might contain unnecessary edges. Nevertheless, in practice, the solutions rarely contain such edges or eventually disappear by observing an appropriate counterexample. This notion of minimality is not related to minimal RMs (i.e., RMs with the fewest possible states) since the number of states of the root in each learning task is fixed.

Restrictions

To further constrain the structure of the learnable HRMs, we introduce some rules similar to those for learning RMs (see Section 5.1.2). For simplicity, some of these constraints use the auxiliary rule below to define the `ed(X, Y, E, M)` atoms, which are equivalent to the `call(X, Y, E, M, M2)` atoms but omitting the called RM:

$$\text{ed}(X, Y, E, M) :- \text{call}(X, Y, E, M, _).$$

The following inductive solutions are ruled out:

- Solutions containing an edge calling the leaf M_{\top} and labeled by a formula formed only by negative literals. The rule below enforces a proposition to occur positively whenever a proposition appears negatively in an edge calling M_{\top} :

$$:- \text{neg}(X, Y, E, M, _), \text{not pos}(X, Y, E, M, _), \text{call}(X, Y, E, M, M_{\top}).$$

- Solutions containing an unlabeled edge calling the leaf M_{\top} (i.e., a call to the leaf with context \top) to avoid unconditional transitions. The rule below enforces the constraint:

$$:- \text{not pos}(X, Y, E, M, _), \text{not neg}(X, Y, E, M, _), \text{call}(X, Y, E, M, M_{\top}).$$

Table 8.2: List of hyperparameters and their values. The *HRM policy learning* hyperparameters annotated with SMDP correspond to SMDP Q-learning updates (i.e., policies over options).

Parameter	CRAFTWORLD	WATERWORLD
<i>General</i>		
Episodes		
Without HRM learning	100,000 (OP, OPL); 200,000 (FR, FRL)	100,000 (WOD); 200,000 (WD)
With HRM learning	150,000 (OP, OPL); 300,000 (FR, FRL)	150,000 (WOD); 300,000 (WD)
Maximum episode length	1,000	1,000
Num. of instances $ \mathbb{I} $	10	10
<i>HRM policy learning (Section 7.1)</i>		
Learning rate α	5×10^{-4}	1×10^{-5}
Learning rate (SMDP) α	5×10^{-4}	1×10^{-3}
Optimizer	RMSprop (Hinton et al., 2012a)	RMSprop (Hinton et al., 2012a)
Discount γ	0.9	0.9
Discount (SMDP) γ	0.99	0.99
Updated formula Q-functions per step	4	4
Replay memory size	500,000	500,000
Replay start size	100,000	100,000
Target network update frequency	1,500	1,500
Replay memory size (SMDP)	10,000	10,000
Replay start size (SMDP)	1,000	1,000
Target network update frequency (SMDP)	500	500
Minibatch size	32	32
Initial exploration	1.0	1.0
Final exploration	0.1	0.1
Annealing steps	2,000,000	5,000,000
Annealing steps (SMDP)	10,000	10,000
<i>HRM learning (Section 7.3)</i>		
Curriculum weight β	0.99	0.99
Curriculum threshold	0.85	0.75
Curriculum update frequency (# episodes)	100	100
ILASP time budget	2 hours	2 hours
Num. collected goal traces ρ (height 1)	25	25
Num. collected goal traces ρ (height ≥ 2)	150	150
Num. goal traces ρ_s to learn first HRM	10	10
Max. num. edges between states κ	1	1
<i>DRQN</i>		
Learning rate α	1×10^{-4}	—
Optimizer	RMSprop (Hinton et al., 2012a)	—
Discount γ	0.99	—
Replay memory size	1,000	—
Replay start size	100	—
Network update frequency	16	—
Network update scheme	Bootstrapped random updates	—
Target network update frequency	1,500	—
Num. sampled episodes	1	—
Sampled sequence length	128	—
Initial exploration	1.0	—
Final exploration	0.1	—
Annealing episodes	300,000	—
<i>CRM</i>		
Learning rate α	5×10^{-4}	1×10^{-5}
Optimizer	RMSprop (Hinton et al., 2012a)	RMSprop (Hinton et al., 2012a)
Discount γ	0.99	0.99
Replay memory size	1,000,000	1,000,000
Replay start size	100,000	100,000
Target network update frequency	1,500	1,500
Minibatch size	32	32
Initial exploration	1.0	1.0
Final exploration	0.1	0.1
Annealing steps	100,000,000	2,000,000

- Solutions containing non-accepting and non-rejecting states without outgoing edges. These states are only required in scenarios where the accepting or rejecting states must be unreachable from them. In practice, these scenarios are not considered, so we rule them out through the following rule:

$$\left\{ \begin{array}{l} \text{has_outgoing_edges}(X, M) :- \text{ed}(X, -, -, M). \\ :- \text{state}(X, M), \text{not has_outgoing_edges}(X, M), X \neq u^A, X \neq u^R. \end{array} \right\}.$$

- Solutions containing cycles; that is, solutions where two states can be reached from each other. These solutions are discarded to speed up the learning. The following set of rules, which is analogous to that in Section 5.1.2, enforces the constraint:

$$\left\{ \begin{array}{l} \text{path}(X, Y, M) :- \text{ed}(X, Y, -, M). \\ \text{path}(X, Y, M) :- \text{ed}(X, Z, -, M), \text{path}(Z, Y, M). \\ :- \text{path}(X, Y, M), \text{path}(Y, X, M). \end{array} \right\}.$$

Furthermore, we *compress* traces similarly to Definition 5.1.1 with the difference that empty labels are not removed; hence, the compressed traces here considered only consist of removing contiguous equal labels from a given trace. For instance, the trace $\langle \{\}, \{\mathbb{A}\}, \{\mathbb{A}\}, \{\}, \{\}, \{\clubsuit\}, \{\clubsuit\} \rangle$ is compressed to $\langle \{\}, \{\mathbb{A}\}, \{\}, \{\clubsuit\}, \{\clubsuit\} \rangle$. Even though LHRM does not require traces to be compressed, performance is enhanced since traces are often shortened, as shown in Chapter 5.

8.1.3 Reporting Results

We report the results using tables and figures that result from averaging 5 independent runs, each using a different random seed (hence, a different set of random instances). Experiments involving (H)RM learning had a 2-hour budget exclusively addressed to learning the machines for the entire run (i.e., not for each individual learning task). All *timed* experiments were run on 3.40GHz Intel® Core™ i7-6700 processors, while *non-timed* experiments were run on 2.90GHz Intel® Core™ i7-10700, 4.20GHz Intel® Core™ i7-7700K, and 3.20GHz Intel® Core™ i7-8700 processors.

The metrics reported for each *table* vary across experiments; hence, we describe them in the respective sections for clarity. Results are reported as $\mu \pm \sigma$, where μ is the average and σ is the standard error for a given metric. We mark with a dash (–) cases where the (H)RM learner has timed out across all runs. The *learning curves* show the average undiscounted return obtained by the greedy policy every 100 episodes across instances and runs. The dotted vertical lines correspond to episodes where an HRM was learned.

8.2 Learning Non-Flat Hierarchies of Reward Machines

In this section, we evaluate the effectiveness of LHRM for interleaving the exploitation and learning of HRMs in CRAFTWORLD and WATERWORLD. As discussed throughout the thesis, (H)RMs compactly encode history in terms of high-level propositional events. Here, we also experiment with two alternative ways of handling history described in Section 2.1.2: (i) regular DQNs, which do not handle history (i.e., they are memoryless), and (ii) DRQNs, which extend DQNs by employing LSTMs

to capture histories. Unlike LSTMs, (H)RMs enable task decomposition and are interpretable. In what follows, we describe the setup and the obtained results for the experiments.

8.2.1 Experimental Setup

The comparison with memoryless (DQNs) and neural memory (DRQNs) approaches is performed across different CRAFTWORLD tasks; hence, the following implementation details apply to this domain only. The curriculum learning method is the same as for LHRM but performed for a single task; that is, the curriculum only chooses across instances at the start of each episode since there is a single task. Both DQNs and DRQNs are implemented as DDQNs.

The network architecture of the baseline DQNs resembles that of the formula-associated DQNs in our approach (see Section 8.1.1). The only difference is that the MLP takes the concatenation of (i) the CNN’s output and (ii) a vector encoding of the label at that step, where occurring propositions take a value of 1 and 0 otherwise. We provide (ii) to leverage the same information exploited by LHRM. The hyperparameters are the same as for the formula-associated DQNs (see Table 8.2) except for the exploration factor, which is linearly annealed from 1.0 to 0.1 across 300,000 episodes.

The architecture of the DRQNs is similar to the baseline DQNs’ but includes an LSTM instead; specifically, following the implementation of Hausknecht and Stone (2015), we replace the first fully connected layer with an LSTM layer of the same size. The rest of the hyperparameters are listed in Table 8.2, where the replay memory size and the replay start size are given in terms of episodes (i.e., not steps, unlike the other approaches). Since updates are more costly in DRQNs, they are applied every 16 steps instead of after each step. The network is updated using a 128-step sequence from a single episode (if the sampled episode is shorter, the sequence is padded). Even though different combinations of learning rates (5×10^{-5} , 1×10^{-4} , 5×10^{-4}), discount factors (0.9, 0.99, 0.999), target network update frequencies (1,500, 5,000), sequence lengths (4, 8, 16, 32) and number of sampled episodes (1, 2, 4, 8) were tested, performance was always poorer than LHRM’s.

8.2.2 Results

We here show that LHRM effectively interleaves the exploitation and learning of HRMs in CRAFTWORLD and WATERWORLD, and consider some ablations that impact both types of learning. We also compare the performance of LHRM to that obtained by memoryless and neural memory approaches. The findings for LHRM’s learning component derive from the tables in Appendix B.2. We refer the reader to the specific tables where appropriate.

Main Results

Figure 8.2 shows the LHRM learning curves for CRAFTWORLD (FRL) and WATERWORLD (WD). These settings are the most challenging due to the inclusion of dead-ends since (i) they hinder the observation of goal examples in level 1 tasks using random walks, (ii) the RMs must include rejecting states, (iii) formula options must avoid dead-ends, and (iv) call options must avoid invoking options leading to rejecting states. In line with the curriculum method, LHRM does not start learning a level h task until the average return for tasks from levels $1, \dots, h - 1$ surpasses a given threshold. The convergence for high-level tasks is often fast due to the reuse of lower level HRMs and policies.

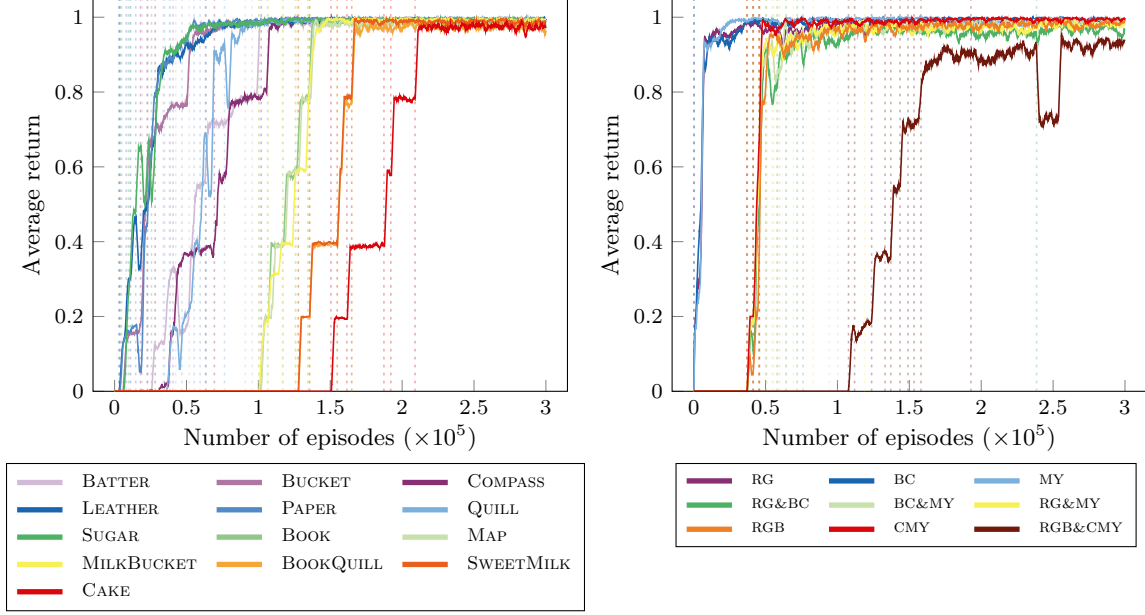


Figure 8.2: LHRM learning curves for CRAFTWORLD (FRL) and WATERWORLD (WD).

The sudden performance decay for RGB&CMY around episode 2.5×10^5 is due to a new RM for RG&BC being learned—a vertical line for the latter is shown at the time of the decay. Following our curriculum method, the average return for RG&BC is reset to 0 and the current level is set to 2; hence, the agent stops performing RGB&CMY (level 3), which causes the performance decay (the return is 0 while a task is not active). When the average return across level 2 tasks is again above the threshold, the agent continues learning RGB&CMY.

The average time (in seconds) spent on learning *all* HRMs is 1009.8 ± 122.3 for OP, 1622.6 ± 328.7 for OPL, 1031.6 ± 150.3 for FR, 1476.8 ± 175.3 for FRL, 35.4 ± 2.0 for WOD, and 67.0 ± 6.2 for WD—see Tables B.1–B.2 for details. Dead-ends (OPL, FRL, WD) incur longer times since (i) there is one more proposition, (ii) there are edges to the rejecting state(s), and (iii) there are dead-end traces to cover. We observe that the complexity of learning an HRM does not necessarily correspond with the instance-type complexity (e.g., the times for OP and FRL are close). Learning in WATERWORLD is faster than in CRAFTWORLD since the RMs have fewer states and there are fewer callable RMs.

Ablations

By *restricting the callable RMs* to those required by the HRM (e.g., *just* using PAPER and LEATHER RMs to learn BOOK’s), there are fewer ways to label the edges of the induced RM. Learning is $5\text{--}7\times$ faster using 20% fewer calls to the learner (i.e., fewer examples) in CRAFTWORLD, and $1.5\times$ faster in WATERWORLD; thus, HRM learning becomes less scalable as the number of tasks and levels grows. This is an instance of the *utility* problem (Minton, 1988). How to refine the callable RM set, prior to HRM learning, is an avenue for future work. We refer the reader to Tables B.3–B.4 for specifics.

We evaluate the performance of *exploration with options* using the number of episodes needed to collect the ρ goal traces for a given task since the activation of its level. Intuitively, the agent rarely moves far from a region of the state space using primitive actions only, thus taking longer to collect the traces; in contrast, options enable the agent to explore the state space efficiently. In

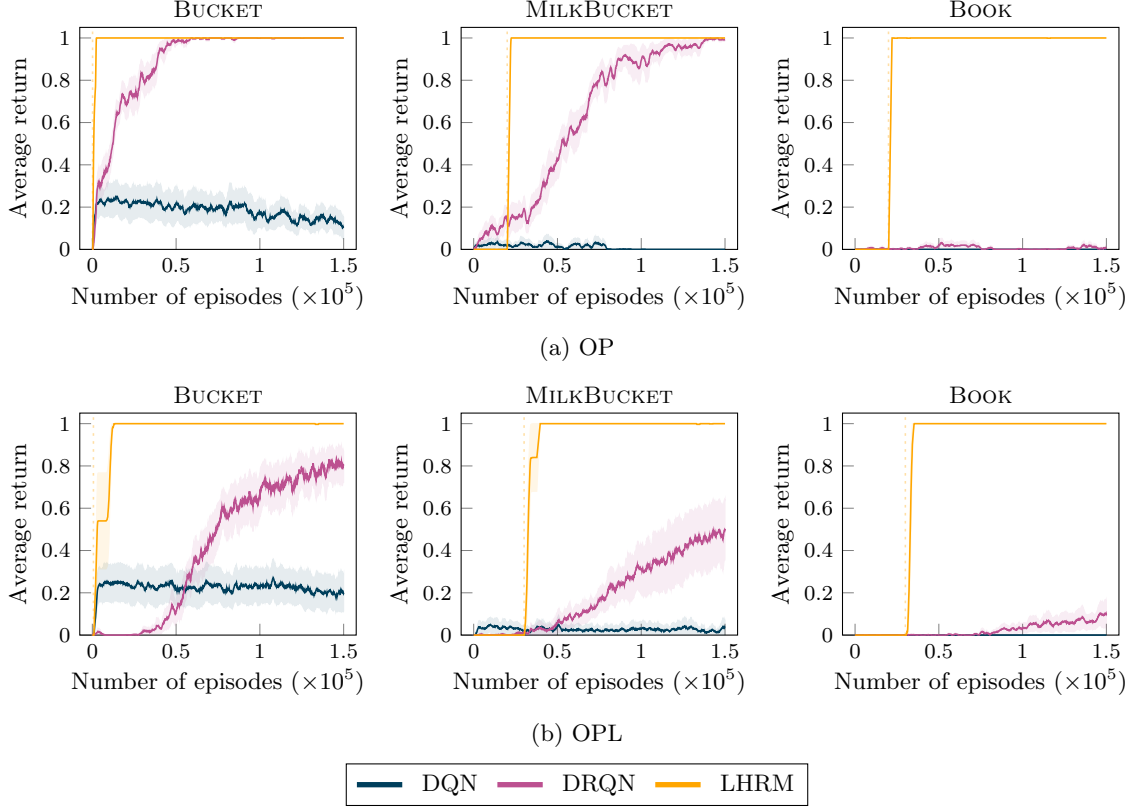


Figure 8.3: Learning curves for DQN, DRQN and LHRM in CRAFTWORLD.

CRAFTWORLD’s FRL setting, using primitive actions requires $128.1\times$ more episodes than options in MILKBUCKET, the only level 2 task for which ρ traces are collected. Likewise, primitive actions take $53.1\times$ and $10.1\times$ more episodes in OPL and WD, respectively. In OP and WOD, options are not as beneficial since episodes are relatively long (1,000 steps), there are no dead-ends, and it is easy to observe the different propositions. For specific details, see Tables B.5–B.6.

Learning the first HRMs using a *single goal trace* ($\rho = \rho_s = 1$) incurs frequent timeouts in all CRAFTWORLD settings, thus showing the value of using many short traces instead.

Comparison with Memoryless and Neural Memory Approaches

Figure 8.3 shows that LHRM outperforms DQN and DRQN across different CRAFTWORLD tasks in OP and OPL instances. DRQN increasingly struggles with the complexity of the tasks, while the poor performance of DQN shows the need for capturing histories. The apparent convergence delay in LHRM is due to the curriculum method; e.g., the policies and HRMs for MILKBUCKET and BOOK become learnable once level 2 in the curriculum activates. Previous works on learning RMs (Toro Icarte et al., 2019) show similar results using other memoryless and LSTM-based approaches.

8.3 Learning Flat Hierarchies of Reward Machines

Learning a flat HRM is often less scalable than learning a non-flat equivalent since (i) already learned HRMs cannot be reused, and (ii) a flat HRM usually has more states and edges (as shown

in Theorem 6.2.2, growth can be exponential). We compare the performance of learning (from interaction) a non-flat HRM using LHRM with that of an equivalent flat HRM using LHRM (i.e., our own approach but without calling lower level RMs), DeepSynth (Hasanbeig et al., 2021), JIRP (Xu et al., 2020) and LRM (Toro Icarte et al., 2019).¹ In the following paragraphs, we describe the evaluated RM learning approaches, and discuss the results. The descriptions of the RM learning approaches include a brief qualitative comparison to LHRM, with a focus on the learning aspects here evaluated. For comparisons including the exploitation aspects, we refer the reader to Section 9.1.1.

8.3.1 Experimental Setup

The core difference with respect to learning non-flat HRMs is that there is a single task for which the HRM is learned. In this setting, our method (LHRM) is therefore not allowed to reuse previously learned HRMs for other tasks; however, it still uses the same hyperparameters (see Table 8.2). In the case of DeepSynth, JIRP and LRM, we exclusively evaluate their RM learning components using traces collected through random walks. For a fair comparison against LHRM (both in the non-flat and flat learning cases), we (i) compress the traces using the methodology described in Section 8.1.2, and (ii) use the OP and WOD settings of CRAFTWORLD and WATERWORLD respectively, where observing goal traces by randomly exploring the environment is relatively easy (especially for simple tasks such as MILKBUCKET).² In these approaches, a different instance is selected at each episode following a cyclic order (i.e., 1, 2, ..., $|\mathbb{I}| - 1$, $|\mathbb{I}|$, 1, 2, ...); besides, if needed, they include a proposition for when no original proposition is observed.

In what follows, we briefly describe the RM learning component and hyperparameters of DeepSynth, JIRP, and LRM.³ In all cases (including ours), learning is performed from positive examples (i.e., traces observable from interaction) only; besides, they all relearn RMs from counterexamples. DeepSynth, JIRP, and LRM learn RMs whose edges are labeled by proposition sets. Since proposition sets are mutually exclusive by default, these methods do not need to enforce mutual exclusivity between the edges to two different RM states; in contrast, LHRM employs propositional logic formulas and needs to enforce mutual exclusivity. To the best of our knowledge, unlike LHRM, these methods do not explicitly break symmetries during the search for an RM.

DeepSynth

DeepSynth (Hasanbeig et al., 2021) learns RMs using a program synthesis method that tackles long traces efficiently by segmenting them (Jeppu et al., 2020). Both DeepSynth and LHRM aim to learn minimal RMs by performing iterative deepening on the number of states; however, their ability to learn minimal RMs depends on different hyperparameters. LHRM depends on the maximum number of edges κ from one state to another, whereas DeepSynth depends on two hyperparameters. First, DeepSynth specifies the size w of the sliding window for segmenting the traces; crucially, this helps control the algorithm complexity by only considering unique segments. Second, as discussed later, algorithms that learn minimal finite-state machines only from positive examples tend to overgen-

¹The codebases for DeepSynth (<https://github.com/grockious/deepsynth>) and LRM (<https://bitbucket.org/RToroIcarte/lrm>) are linked in the papers, whereas the one for JIRP (<https://github.com/corazza/stochastic-reward-machines>) was referred to us by one of the authors through personal communication.

²Following Table 8.2, each run for these instances consists of 150,000 episodes.

³DeepSynth actually learns FSMs different from RMs; however, our experimental analysis focuses on capturing the task structure, so we here consider DeepSynth as an RM learning method.

eralize; hence, to alleviate this problem, a hyperparameter l is introduced to control the degree of generalization by eliminating sequences of length l present in the RM but not in the traces. In other words, w controls the length of the positive (i.e., observable, feasible) examples, whereas l controls the length of the negative (i.e., unfeasible) examples.

Both DeepSynth and LHRM learn an initial RM from a set of traces collected through random exploration. DeepSynth updates the RM upon observing a trace containing a new label or, more generally, when no transition from the current RM state is labeled with the currently observed label. The RMs learned by DeepSynth, like ours, contain a set of accepting states. The accepting traces change throughout the process by observing new labels, which results in an incremental build of the RM from previous RMs. This strategy removes the need for observing goal traces from random exploration and incrementally builds the RM associated with the task from partial traces; nevertheless, this strategy assumes that each newly observed label is important to complete the underlying task, which might not be true. We hypothesize that due to this strategy and the previous hyperparameters, DeepSynth often learns an RM that accurately predicts the next event/label instead of a minimal one.

By default, DeepSynth calls the RM learner periodically, even if the example set remains unchanged; experimentally, this strategy led to avoidable timeouts (i.e., the time budget was spent unnecessarily), so we modified the code to call the learner only upon observing a counterexample. The rest of the hyperparameters are left unchanged. The agent collects traces for 50,000 steps before learning an initial RM, whereas the values of w and l are 3 and 2, respectively.

JIRP

JIRP (Xu et al., 2020) casts the RM learning task as a SAT problem. Like in our case, the tasks considered by Xu et al. are episodic and fulfill Assumption 3.1.1 (i.e., the reward is 1 for goal histories, and 0 otherwise). Both JIRP and LHRM learn minimal RMs by performing iterative deepening on the number of states. JIRP determines that a trace is a counterexample if the sequence of rewards it yields in the RM differs from that in the environment; hence, under Assumption 3.1.1, JIRP learns RMs that distinguish between goal and incomplete traces. JIRP periodically learns (if needed) a new RM from a set of counterexamples; in contrast, LHRM only uses a set of counterexamples to learn the first RM, and then learns a new RM every time a counterexample is observed.

The experiments were performed using the PySAT solver (Ignatiev et al., 2018). The approach consists of two hyperparameters. The maximum number of states is set to the size of a minimal RM for the task at hand. The frequency with which the RM is updated from counterexamples is set to 20 episodes, which is the default value used in the released code.

LRM

LRM (Toro Icarte et al., 2019) formulates the RM learning problem as a discrete optimization problem, which is solved using a local search algorithm called Tabu search (Glover and Laguna, 1997). Given a candidate RM, Tabu search derives a list of neighbors (in this case, RMs that differ by one transition), and the neighbor minimizing a specific optimization metric is selected as the next candidate RM; crucially, Tabu search maintains a (tabu) list of the previously selected candidate RMs and prevents them from being chosen again. LRM aims to find an RM that is good at predicting the next different label given a maximum number of states; that is, unlike LHRM, it

Table 8.3: Results of learning non-flat and flat HRMs using different methods. The fields are (left-to-right): the method name, the number of completed runs without timing out, the amount of time needed to learn the HRMs or RMs, and the number of states and edges of the RM.

Method	Task	C	Time (s.)	States	Edges
LHRM (Non-Flat)	MILKBUCKET	5	1.5 ± 0.2	3.0 ± 0.0	2.0 ± 0.0
	BOOK	5	191.2 ± 36.4	5.0 ± 0.0	5.8 ± 0.2
	BOOKQUILL	5	17.9 ± 1.4	4.0 ± 0.0	4.0 ± 0.0
	CAKE	5	74.5 ± 25.7	4.0 ± 0.0	3.2 ± 0.2
	RG	5	0.9 ± 0.0	3.0 ± 0.0	2.0 ± 0.0
	RG&BC	5	4.5 ± 0.3	4.0 ± 0.0	4.0 ± 0.0
	RGB&CMY	5	15.1 ± 1.7	4.0 ± 0.0	4.0 ± 0.0
LHRM (Flat)	MILKBUCKET	5	3.2 ± 0.6	4.0 ± 0.0	3.6 ± 0.2
	BOOK	0	—	—	—
	BOOKQUILL	0	—	—	—
	CAKE	0	—	—	—
	RG	5	0.9 ± 0.0	3.0 ± 0.0	2.0 ± 0.0
	RG&BC	0	—	—	—
	RGB&CMY	0	—	—	—
DeepSynth	MILKBUCKET	5	325.6 ± 29.7	13.4 ± 0.4	93.2 ± 1.7
	BOOK	5	288.9 ± 31.7	16.6 ± 3.1	119.0 ± 19.4
	BOOKQUILL	5	308.6 ± 52.6	12.8 ± 0.5	92.8 ± 2.3
	CAKE	4	290.6 ± 36.4	17.2 ± 2.5	110.2 ± 11.6
	RG	0	—	—	—
	RG&BC	0	—	—	—
	RGB&CMY	0	—	—	—
JIRP	MILKBUCKET	5	17.1 ± 5.5	4.0 ± 0.0	3.0 ± 0.0
	BOOK	0	—	—	—
	BOOKQUILL	0	—	—	—
	CAKE	0	—	—	—
	RG	5	32.3 ± 7.9	3.8 ± 0.2	82.4 ± 9.1
	RG&BC	0	—	—	—
	RGB&CMY	0	—	—	—
LRM	MILKBUCKET	5	347.5 ± 64.5	4.0 ± 0.0	14.0 ± 1.0
	BOOK	5	2261.0 ± 552.2	8.0 ± 0.0	31.2 ± 2.0
	BOOKQUILL	0	—	—	—
	CAKE	0	—	—	—
	RG	0	—	—	—
	RG&BC	0	—	—	—
	RGB&CMY	0	—	—	—

does not look for a minimal RM. LRM does not exclusively address goal-oriented tasks; therefore, the learned RMs do not have explicit accepting and rejecting states, and different trace types (goal, dead-end, incomplete) are not distinguished. Both LRM and LHRM learn an initial RM from a set of traces, although in the latter case the set only consists of goal traces. LRM determines that a trace is a counterexample if it contains a label l' that is observed for the first time from an RM state u after having observed a different label l ; intuitively, given the aforementioned optimization objective, when the new RM might need to refine its ability to predict the next event from the previous one.

The experimental hyperparameters are the following. The maximum number of RM states is set to that of a minimal RM for the task at hand. The rest of the values are the ones set by default. Random traces are collected for 1,000 episodes before learning an initial RM. The tabu list consists of 10,000 entries, and each search consists of 100 steps.

8.3.2 Results

Table 8.3 shows the results for the different learning approaches. A non-flat HRM for MILKBUCKET (level 2) is learned in 1.5 ± 0.2 seconds, whereas flat HRMs take longer: 3.2 ± 0.6 w/LHRM, 325.6 ± 29.7 w/DeepSynth, 17.1 ± 5.5 w/JIRP and 347.5 ± 64.5 w/LRM. LHRM and JIRP learn minimal RMs, hence producing the same RM consisting of 4 states and 3 edges. DeepSynth and LRM do not learn a minimal RM but one that is good at predicting the next possible label given the current one (and the current RM state in the case of LRM). In domains like ours where propositions can be observed anytime (i.e., without temporal dependencies between them), these methods tend to overfit the input traces and output large RMs that barely reflect the task’s structure, e.g. DeepSynth learns RMs with 13.4 ± 0.4 states and 93.2 ± 1.7 edges. In contrast, methods learning minimal RMs from observable traces only may suffer from *overgeneralization* (Angluin, 1980) in other domains (e.g., with temporally-dependent propositions), as acknowledged in our previous work (Furelos-Blanco et al., 2021) and, recently, by Toro Icarte et al. (2023). The earlier observations for MILKBUCKET also apply to more complex tasks (i.e., involving more high-level temporal steps and multiple paths to the goal), such as BOOK (level 2), BOOKQUILL (level 3) and CAKE (level 4); namely, LHRM learns non-flat HRMs for these tasks in (at most) a few minutes, while learning an informative flat HRM is unfeasible.

DeepSynth, JIRP and LRM perform poorly in WATERWORLD. Unlike LHRM, these methods learn RMs whose edges are not labeled by formulas but proposition sets; hence, the RMs may have exponentially more edges (e.g., 64 instead of 2 for RG), and become unfeasible to learn. Indeed, flat HRM learners time out in RG&BC and RGB&CMY, while LHRM only needs a few seconds.

8.4 Exploiting Handcrafted Hierarchies of Reward Machines

We compare the performance of policy learning in handcrafted non-flat HRMs against that in flat equivalents, which are guaranteed to exist by Theorem 6.2.1. For fairness, the flat HRMs are minimal. To exploit the flat HRMs, we apply our HRL algorithm (Section 7.1) and CRM (Section 2.1.5), which learns an action-value function over $\mathcal{S} \times \mathcal{U}$ using synthetic counterfactual experiences for each RM state. Next, we briefly describe how we employ CRM and discuss the evaluation results.

8.4.1 Experimental Setup

The CRM networks are DDQNs that resemble the formula-associated networks in our approach (see Section 8.1.1). The difference is that the CNN output is concatenated with a one-hot representation of the RM state and then fed to the MLP. Table 8.2 lists the rest of the hyperparameters.

8.4.2 Results

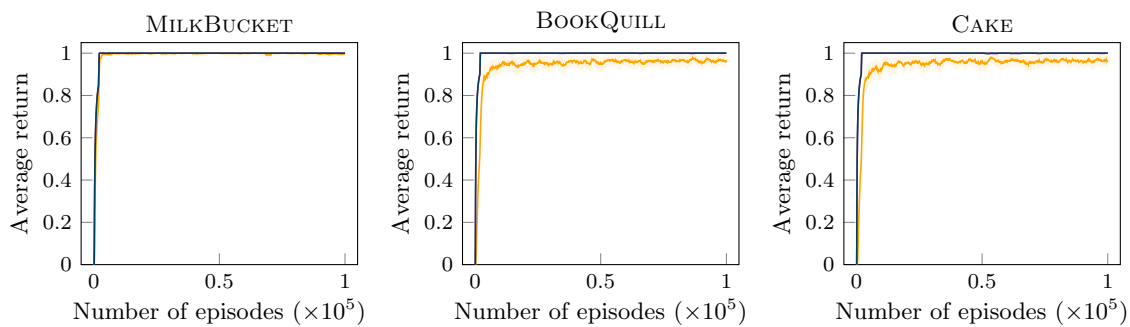
Figure 8.4d shows the learning curves for some CRAFTWORLD tasks using FRL instances. The convergence rate is similar in the simplest task (MILKBUCKET), but higher for non-flat HRMs in the hardest ones. Unlike the HRL approaches, CRM does not decompose the subtask into independently solvable subtasks and, hence, deals with sparser rewards that result in a slower convergence; indeed, the only non-zero reward comes from the transitions to an accepting state. In the case of the HRL approaches, since both use the same set of formula option policies, differences arise from flat HRMs’ lack of modularity. Call options, which are not present in flat HRMs, form independent modules that reduce reward sparsity. MILKBUCKET involves fewer high-level steps than BOOKQUILL and CAKE; thus, the reward is less sparse and non-flat HRMs are not as beneficial.

The efficacy of non-flat HRMs, as shown in Figures 8.4a–8.4c and 8.5, is also limited when (i) the task’s goal is reachable regardless of the chosen options (e.g., if there are no rejecting states, like in OP and FR), and (ii) the reward is less sparse, like in OPL (the grid is small) or WATERWORLD (the balls easily get near the agent, so even a poor agent can achieve the goal if the number of steps per episode is large). In general, we also observe that CRM’s performance improves in the latter scenarios except for RGB&CMY, where the average return eventually decreases. Similar phenomena in simpler tasks were addressed via hyperparameter tuning (e.g., learning rate, discount, buffer size, network architecture, annealing steps for the exploration factor); nonetheless, CRM’s long-term performance was consistently unstable on RGB&CMY.

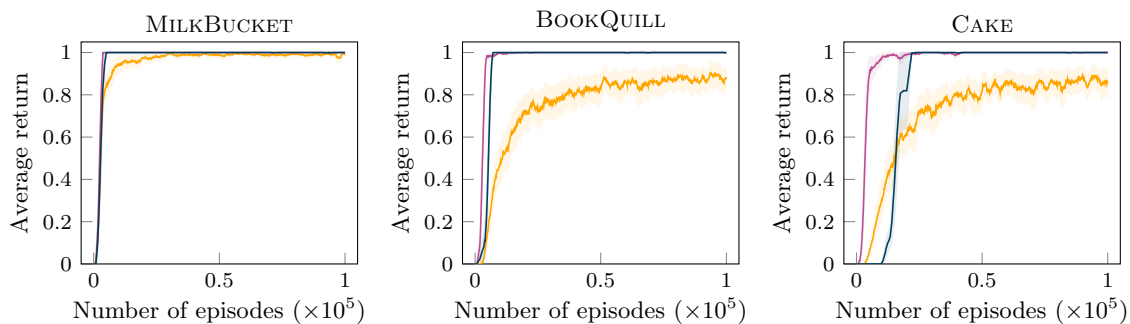
Figure 8.4b shows a case where the convergence in the non-flat case is delayed with respect to the flat one. Remember that in DQNs, learning does not start until the buffers contain a certain number of experiences. In our approach, as described in Section 7.1.1, there is a DQN and a replay buffer for each RM; thus, in the flat case, there is a single DQN and buffer, while there are several in the non-flat case. Filling the buffers in the non-flat case is slower since there are higher-level options (i.e., call options) that do not occur as often as others (i.e., formula options), which explains the slight convergence delay.

8.5 Summary

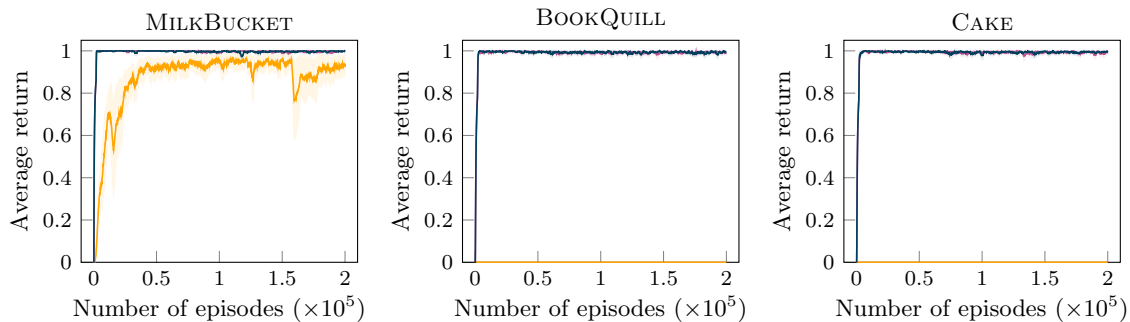
In this chapter, we evaluated the exploitation and HRM learning methods introduced in the previous chapter. First, we have shown that LHRM, our method for interleaving exploitation and HRM learning, manages to learn the HRMs and performant policies for a set of tasks in two different domains; besides, learning HRMs and exploiting them leads to faster convergence than memoryless and RNN-based approaches. Second, in line with the theory from Chapter 6, learning an HRM is feasible in cases where a flat equivalent HRM is not. Third, our HRL exploitation method converges faster by exploiting a non-flat HRM instead of a flat equivalent one. These results empirically demonstrate the advantages of hierarchically composing RMs on the exploitation and learning fronts.



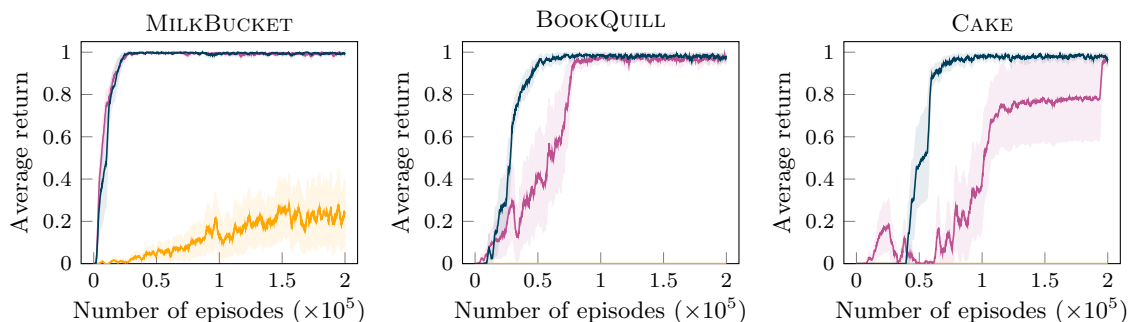
(a) CRAFTWORLD – OP



(b) CRAFTWORLD – OPL



(c) CRAFTWORLD – FR



(d) CRAFTWORLD – FRL

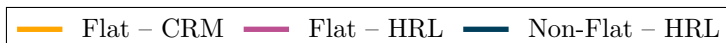


Figure 8.4: Learning curves for three CRAFTWORLD tasks using handcrafted HRMs.

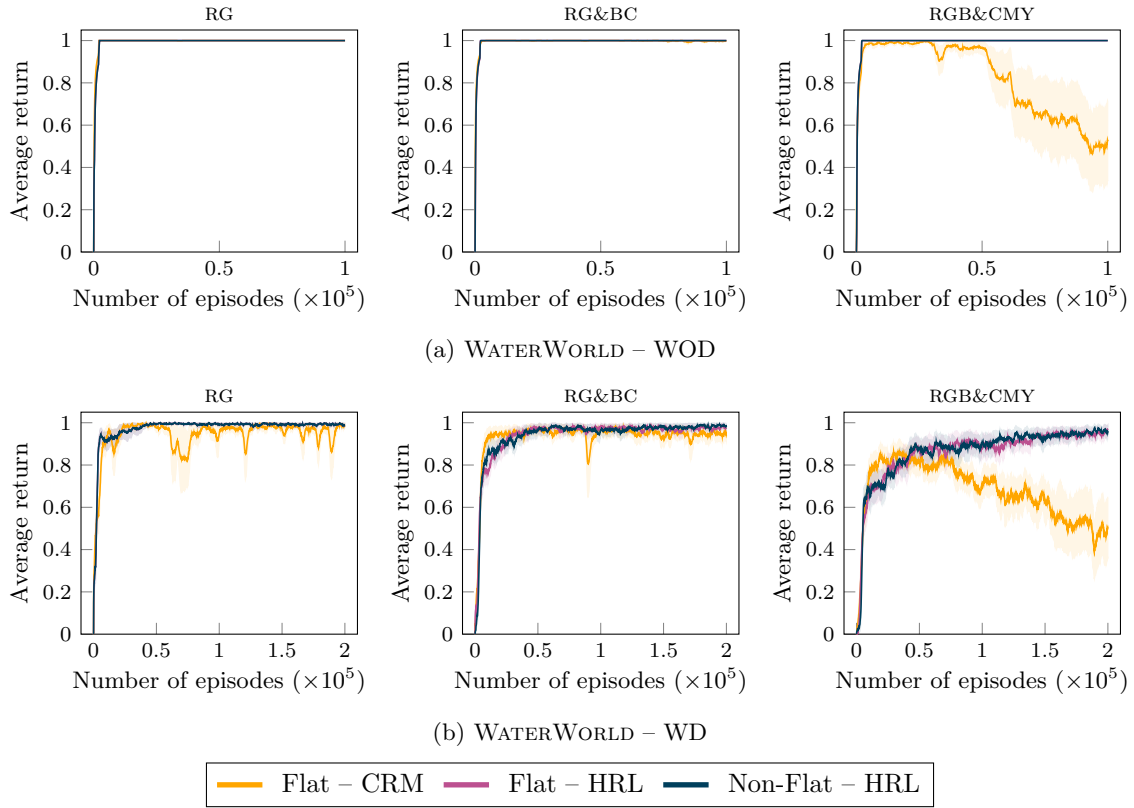


Figure 8.5: Learning curves for three WATERWORLD tasks using handcrafted HRMs.

Chapter 9

Related Work

In this chapter, we survey research relevant to our work. First, we look at approaches that employ finite-state machines in reinforcement learning (Section 9.1). Then, we discuss relevant work in hierarchical reinforcement learning (Section 9.2) and curriculum learning (Section 9.3). Finally, we examine methods focused on breaking symmetries in graphs (Section 9.4).

9.1 Finite-State Machines in Reinforcement Learning

This section discusses different approaches that use finite-state machines in reinforcement learning. We start focusing on recent works around reward machines (Section 9.1.1); then, we briefly describe other formalisms based on finite-state machines and relate them to our work (Section 9.1.2).

9.1.1 Reward Machines

Reward machines build on the idea of revealing a task’s reward function to the agent for enabling *task decomposition* previously suggested by Karlsson (1994). The high-level propositional events employed in RMs and similar approaches resemble the *salient events* from the intrinsic motivation literature (Singh et al., 2004). In what follows, we review the main research areas that have been explored since the introduction of RMs by Toro Icarte et al. (2018a).

Formalism

The reward machines considered in this thesis differ from those introduced by Toro Icarte et al. (2018a, 2022) in three main regards: (i) accepting and rejecting states are included, (ii) transitions are labeled with propositional logic formulas in disjunctive normal form instead of arbitrary formulas or proposition sets, and (iii) the reward-transition function is defined over state pairs rather than state-label pairs. We refer the reader to Sections 3.2 and 6.1 for further details.

The labeling function considered in this thesis is deterministic (i.e., there is no uncertainty on the labels observed by the agent); likewise, the state-transition and reward-transition functions are also deterministic. Other works have recently considered scenarios where the labeling function is noisy (Li et al., 2022; Verginis et al., 2022), and RMs whose state-transition and/or reward-transition functions are stochastic (Corazza et al., 2022; Dohmen et al., 2022). Zhou and Li (2022a) augment RMs by

allowing transitions over predicates instead of only employing proposition sets or propositional logic formulas.

Diverse ways of *composing* RMs have been considered in the literature. In this thesis, we have proposed to compose them hierarchically. De Giacomo et al. (2020) compose RMs by merging their state and reward transition functions. In multi-agent scenarios, the RMs of the different agents are composed in parallel (Neary et al., 2021; Dann et al., 2022; Ardon et al., 2023). Task composability in RL has also been modeled through several other methods, including context-free grammars (Chevalier-Boisvert et al., 2019), logic-based task algebras (Nangue Tasse et al., 2020), programs (Verma et al., 2018; Sun et al., 2020; Yang et al., 2021; Zhou and Li, 2022b), subtask sequences (Andreas et al., 2017) and graphs (Sohn et al., 2018), symbolic planning (Illanes et al., 2020), temporal logics (Toro Icarte et al., 2018b; León et al., 2020; Wang et al., 2020; Vaezipoor et al., 2021; León et al., 2022), and other specification languages (Jothimurugan et al., 2019).

Exploitation

Toro Icarte et al. (2022) propose a hierarchical RL algorithm for exploiting RMs similar to the one we describe in Section 4.1.1. Similarities and differences between these methods are as follows:

- Both define an option for each transition. However, unlike our method, theirs includes options for self-transitions and does not include options for transitions to undesirable terminal RM states (e.g., rejecting states in our framework).
- Both perform intra-option learning.
- In our method, the option policies are determined by the formulas labeling the RM edges, whereas Toro Icarte et al. determine them based on each edge’s source and target states. Learning formula-associated policies enables reusability since there might be several edges with the same formula.
- The reward functions that Toro Icarte et al. use to learn the option policies depend on the reward-transition function of the RM, and add a penalty for transitioning to an unintended RM state. In contrast, our reward functions for learning options do not depend on the RM (i.e., the subtasks are completely independent of the RM); however, we assume dead-end histories to be common across different stages of the task (i.e., the dead-end indicator at each step can be used to train all policies).

The task structure encoded by RMs has been leveraged to give bonus reward signals. Camacho et al. (2019) propose a reward shaping method based on running value iteration over the RM states, whereas we employ the maximum and minimum distances to the accepting state. Similarly, Camacho et al. (2017) use automata representations of non-Markovian rewards and exploit their structure to guide the search of an MDP planner using reward shaping (e.g., using the minimum distance to an accepting state).

On the more theoretical side, Bourel et al. (2023) derive high-probability regret bounds for RMs. Other exploitation methods have been designed to support the scenarios outlined in the previous section.

Derivation and Learning

The literature on reward machines mainly consists of works that either *derive* them (or similar FSMs) from formal language specifications (Camacho et al., 2019; Araki et al., 2021) and expert demonstrations (Camacho et al., 2021), or *learn* them from experience using discrete optimization (Toro Icarte et al., 2019; Christoffersen et al., 2020; Toro Icarte et al., 2023), SAT solving (Xu et al., 2020; Corazza et al., 2022), active learning (Gaon and Brafman, 2020; Memarian et al., 2020; Xu et al., 2021; Dohmen et al., 2022), state-merging (Xu et al., 2019; Gaon and Brafman, 2020) and program synthesis (Hasanbeig et al., 2021). This thesis proposes learning (H)RMs using an inductive logic programming system. Our method, LHRM, constitutes the first approach to learning hierarchies of reward machines.

Most approaches for learning reward machines, including ours, have addressed the problem of making the reward signal Markovian in NMRDPs (Xu et al., 2019; Christoffersen et al., 2020; Gaon and Brafman, 2020; Memarian et al., 2020; Xu et al., 2020, 2021; Corazza et al., 2022; Dohmen et al., 2022). Other approaches (Toro Icarte et al., 2019; Hasanbeig et al., 2021; Toro Icarte et al., 2023) aim to learn reward machines that accurately predict the next label from the previous one. The latter methods are better suited to POMDPs with non-Markovian observation transitions; indeed, as observed in Section 8.3, they learn RMs that overfit label traces in NMRDPs since different labels can be observed anytime (i.e., there is no particular temporal-dependence between the labels). However, NMRDP-based methods tend to suffer from overgeneralization in POMDP scenarios and, more generally, when there is an apparent temporal dependence between the labels, as exemplified in Section 5.1.1 and later in Chapter 10. Overgeneralization is caused by the aim to learn minimal RMs from feasible traces only; that is, not employing unfeasible traces prevents the agent from detecting temporal dependencies between the environment labels.

In what follows, we qualitatively compare our methods, ISA and LHRM, with a method from each of the families above: DeepSynth (Hasanbeig et al., 2021), JIRP (Xu et al., 2020), LRM (Toro Icarte et al., 2019), and those proposed by Gaon and Brafman (2020). DeepSynth, JIRP and LRM are outlined and evaluated against LHRM in Section 8.3. We start by enumerating some unmentioned algorithmic commonalities and differences with respect to DeepSynth, JIRP and LRM, and continue with a description of the approaches presented by Gaon and Brafman (2020):

- DeepSynth, JIRP, and LRM learn an initial RM from a trace set obtained through random exploration. Similarly, LHRM uses a set of goal traces collected by randomly exploring the environment, but possibly using policies from lower level tasks. In contrast, ISA uses a single goal trace.
- DeepSynth, ISA, LHRM, and LRM emphasize the need for learning from short traces. DeepSynth segments traces using a sliding window of a fixed size. LHRM uses the shortest traces from a set of randomly collected traces to learn the first candidate HRM. ISA, LHRM, and LRM compress traces by removing contiguous equal labels.
- DeepSynth, JIRP, and LRM learn RMs whose edges are labeled with proposition sets, whereas ISA and LHRM employ propositional logic formulas. Further, LHRM also labels edges with calls to other RMs.
- ISA, JIRP, and LHRM aim to learn minimal RMs that model non-Markovian reward, whereas

DeepSynth and LRM aim to predict the next label from the previous label (and RM state in the case of LRM) accurately.

- ISA, JIRP, and LHRM distinguish between different trace types based on whether a goal history is observed or not. DeepSynth and LRM do not make such a distinction, which makes them more amenable to continuing tasks.
- ISA, JIRP, and LRM use the QRM algorithm to exploit the structure of the learned RMs. ISA and LRM relearn policies from scratch when the RM is updated. JIRP, in contrast, attempts to reuse the action-value functions from the previous RM based on a notion of equivalence between RM states: two RM states are equivalent if they yield the same sequence of rewards for all traces in the counterexample set. Xu et al. (2020) claim that the optimal action-value functions are the same for equivalent pairs of states. DeepSynth exploits RMs using an algorithm reminiscent of QRM, with the only apparent difference being the lack of counterfactual updates. Since DeepSynth builds the RMs incrementally, it keeps the action-value functions for each RM state during learning. In the case of the HRL algorithms used by ISA and LHRM, only the metapolicies associated with the learned RM are reset.

Gaon and Brafman (2020) learn RMs whose edges are labeled with actions; essentially, this means the example traces are produced through a labeling function $l : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{A}$ mapping state-action-state triplets to actions. Like in our case, tasks are episodic, and the RMs contain accepting states representing the completion of a task (i.e., obtaining a non-Markovian reward). The authors interleave Q-learning and R-MAX (Brafman and Tenenbholz, 2002) in tabular tasks with two algorithms from the grammatical inference literature (de la Higuera, 2010) that learn minimal machines:

- L^* (Angluin, 1987), an active learning algorithm that learns deterministic finite automata (DFA) from (typically) two query types: (i) membership queries, which request whether a given trace belongs to the target language, and (ii) equivalence queries, which request whether a proposed DFA captures the target language (if it does not, a counterexample is returned). Gaon and Brafman use the agent itself to answer these queries. A membership query is answered by attempting to reproduce the sequence of actions in the given trace. An equivalency query is answered by checking whether any past observed trace is a counterexample to the proposed DFA.
- Evidence Driven State Merging (EDSM; Lang et al., 1998), a state-merging approach consisting of two phases: (i) building a prefix tree acceptor (PTA), a DFA built from the prefixes of a finite set of traces such that positive traces are accepted and negative traces are not, and (ii) iteratively merging pairs of equivalent states, stopping when no merging is possible. The learned DFA depends on the quality of the example set and, under specific conditions, the algorithm converges to the minimal DFA. The complexity is polynomial in the number of examples. Gaon and Brafman apply EDSM by recording which traces reach the goal and which do not, akin to our approach. State-merging methods learn minimal machines by shrinking the state set, whereas ours do the opposite; nevertheless, both approaches may overgeneralize if only positive examples (i.e., observable traces from interaction) are used.

In some experiments, the authors gradually limit the length of the traces towards that of the minimal positive one to control the size of the resulting DFAs. This strategy is in line with that of some of the previously outlined methods, which also favor learning from short traces.

In our approaches, the minimality of the learned RM is conditioned to the maximum number of edges κ allowed from one state to another; in contrast, other methods for learning minimal RMs do not have such a dependence (Gaon and Brafman, 2020; Xu et al., 2020).

9.1.2 Other Formalisms

In what follows, we describe some works employing other types of finite-state machines in RL. We refer the reader to Section 9.2 for a comparison between (H)RMs and hierarchical abstract machines (HAMs; Parr and Russell, 1997; Parr, 1998), an alternative framework to options for hierarchical RL based on finite-state machines.

Meuleau et al. (1999) propose to represent policies with finite memory using a class of finite-state machines called *policy graphs*. Actions are selected stochastically given a state of the graph; likewise, transitions within the graph are also taken stochastically given a source state and an observation. In contrast, the transitions of the RMs we considered are determined by propositional logic formulas over high-level events and taken deterministically. The paper presents a method for learning a policy graph with a specific number of nodes from traces observed by the agent; specifically, the action-selection and node-transition functions are parameterized and learned through stochastic gradient descent. The proposed method is restricted to goal-achievement tasks, akin to our case.

Torrey et al. (2007) learn *relational macros*, finite-state machines whose states and transitions are characterized by first-order logic formulas. These formulas are expressed using first-order logic predicates that describe the environment states. The formulas associated with the states indicate what action to take, while those associated with the transitions indicate when the transition is taken. The learning of the relational macros is done in two phases. First, the *structure learning* phase finds a sequence of actions distinguishing traces that reach the goal from those that do not and composes them into an FSM. Second, the *ruleset learning* phase learns the conditions for choosing actions and taking transitions. The inductive logic programming system Aleph (Srinivasan, 2001) learns the rules in both phases. The learned FSM is used for transfer learning: the target task follows the strategy encoded by the FSM for some steps to estimate the action-values of the actions in the strategy, and then stops using the FSM and acts according to the action-values. In summary, their approach differs from ours in that:

- Traces are formed by actions instead of sets of high-level propositional events; however, in both approaches, traces are divided into groups depending on whether the goal is reached.
- Transitions are labeled by first-order logic formulas instead of propositional logic formulas.
- Logic rules describing what action to take in each FSM state are learned; instead, our policies do not depend on logic rules.
- Relational macros require tasks to share the same action space to be reusable; in contrast, an RM can be reused in another task if the proposition set is shared.

Koul et al. (2019) transform the policy encoded by an RNN into a Moore machine, where the latter is defined in terms of quantized state and observation representations of the former. The

resulting Moore machine is exclusively employed to interpret the policy; that is, its structure is not exploited since the policies have been previously learned.

The tasks considered in this thesis are formalized as labeled Markov decision processes, where non-Markovian rewards can be expressed in terms of label traces. Brafman and De Giacomo (2019) introduce *regular decision processes*, a similar formalism for modeling non-Markovian dynamics and rewards using regular expressions. Reward machines and regular decision processes are closely related since regular expressions can be transformed into finite-state machines; indeed, Abadi and Brafman (2020) represent regular decision processes using Mealy machines (a generalization of RMs) and learn them with existing automaton learning methods.

den Hengst et al. (2022) propose *option machines*, which output a set of options upon observing a high-level event instead of outputting a reward (or a reward function). Akin to HRL methods for RMs, a metapolicy chooses an option within the output set, and the policy for the selected option is subsequently executed. Nangue Tasse et al. (2022) introduce *skill machines*, which combine the temporal abstraction of reward machines with the logical composition of task primitives (Nangue Tasse et al., 2020) to solve unseen tasks without further learning.

9.2 Hierarchical Reinforcement Learning

We here relate our work to relevant literature in hierarchical reinforcement learning (HRL), including the classical frameworks and methods for discovering the elements they respectively build upon.

Formalisms

The algorithms described in this thesis for exploiting (H)RMs build upon the options framework (Sutton et al., 1999; Precup, 2001), introduced in Section 2.1.3. In the function approximation case, our exploitation methods resemble the hierarchies of DQNs by Kulkarni et al. (2016). Call option policies in HRMs are trained from experiences where the invoked options achieve their goals; in a similar vein, Levy et al. (2019) learn policies from multiple hierarchical levels in parallel by training each level as if the lower levels were optimal.

In what follows, we briefly describe other HRL formalisms and relate them to our work. Hierarchical abstract machines (HAMs; Parr and Russell, 1997; Parr, 1998) approach HRL through hierarchies of finite-state machines; hence, they resemble RMs and, especially, HRMs since both compose finite-state machines hierarchically (i.e., by enabling machines to call each other). Despite employing finite-state machines, their structures differ. First, unlike HAMs, RMs include reward-transition functions. Second, HAMs consist of four types of states, each defining a different behavior: choice states (non-deterministically choose the next machine state), call states (call a specific machine), action states (perform a specific primitive action), and stop states (stop the machine and return control to the machine that called it). Agents exploit HAMs by learning what to do at each choice state. In contrast, RMs do not have explicit choice, call, or action states; instead, choices depend on the exploitation algorithm and not only the structure. For instance, in regular RMs, QRM learns a policy for each RM state mapping environment states into actions; in contrast, HRL chooses a formula labeling an outgoing edge from an RM state and then chooses actions to satisfy that formula. Like in HAMs, the machine’s structure constrains the agent’s choices in the latter case. Third, in line with the previous point, (H)RMs decouple traversals from the policies, i.e. the (H)RM

must be followed independently of the agent’s choices; therefore, agents that exploit (H)RMs must be able to interrupt their choices (see Section 7.1). Even though HAMs do not support interruption, Programmable HAMs (Andre and Russell, 2000) extend them with parameterized subroutines, temporary interrupts, aborts, and memory variables; interestingly, call contexts in HRMs can be seen as parameters that last for one transition.

MAXQ (Dietterich, 2000) decomposes a task into a hierarchy of independently solvable subtasks. These hierarchies are represented through graphs showing the task dependencies. (H)RMs constrain subtasks to be performed in a particular order, whereas MAXQ hierarchies do not explicitly enforce an order (i.e., the task policy determines the subtask to perform). The author proposes an algorithm that, akin to our exploitation methods, learns the policy for each task only depending on its subtasks’ policies; hence, policies are easily reusable, and recursive optimality is achieved.

Discovery

One of the core problems in the options framework is *option discovery*; namely, identifying options from experience instead of handcrafting them. Akin to option discovery methods, ISA and LHRM induce a set of options from experience; however, while ISA’s and LHRM’s options are a byproduct of finding an (H)RM that compactly captures traces, usual option discovery methods explicitly look for them. The most explored idea is to constitute options via finding *bottlenecks* (e.g., McGovern and Barto, 2001; Menache et al., 2002; Stolle and Precup, 2002; Şimşek and Barto, 2004); that is, by finding bridges between regions of the state space. Likewise, the conditions labeling the edges of an RM also constitute bottlenecks between stages towards the completion of a task; indeed, the high-level methodology of some of these methods is reminiscent of that followed by ISA and LHRM. For instance, McGovern and Barto (2001) use diverse density to find landmark states in state trajectories, and it is similar to our approaches because (i) it learns from trajectories, (ii) it classifies trajectories into different categories depending on whether the goal is achieved or not, and (iii) it interleaves option discovery and policy learning for the discovered options. Discovering options for *exploration* is an active research topic (Bellemare et al., 2016; Machado et al., 2017; Jinnai et al., 2019; Machado et al., 2020; Dabney et al., 2021; Klissarov and Machado, 2023; Lobel et al., 2023). Although our options are not discovered for exploration, LHRM leverages options from previously learned HRMs to observe goal traces in new tasks.

The number of option policies in our approaches is bound by the number of propositions (and tasks in the case of LHRM); similarly, some option discovery methods impose an explicit bound on the number of discoverable options (McGovern and Barto, 2001; Bacon et al., 2017; Machado et al., 2017). Furthermore, ISA and LHRM require tasks to be solved at least once before learning an (H)RM (and, hence, options), just like some option discovery methods (McGovern and Barto, 2001; Stolle and Precup, 2002); in contrast, other methods (e.g., Menache et al., 2002; Şimşek and Barto, 2004; Şimşek et al., 2005; Machado et al., 2017) discover options without solving the task and, thus, are also suited to continuing tasks.

Alternative formalisms to automata for expressing formal languages, like grammars, have been used to discover options. Lange and Faisal (2019) induce a straight-line grammar, a non-branching and loop-free context-free grammar, which can only generate a single string. The authors use greedy algorithms to find a straight-line grammar from the shortest sequence of actions that leads to the goal. The production rules are then flattened, leading to one macro-action (a sequence of actions)

per production rule. These macro-actions constitute the set of options.

There has been work on learning the structures from other HRL frameworks. Leonetti et al. (2012) synthesize a HAM from the set of shortest solutions to a non-deterministic planning problem, and use it to refine the choices at non-deterministic points through RL. Mehta et al. (2008) propose a method for discovering MAXQ hierarchies from a trace that reaches the task’s goal.

9.3 Curriculum Learning

The curriculum method employed by LHRM is based on that by Pierrot et al. (2019), who learn hierarchies of neural programs. The authors assume the level of each program is known; likewise, we assume the level of each task (i.e., the height of the corresponding HRM’s root) is known. Andreas et al. (2017) employ a similar method that prioritizes tasks consisting of fewer high-level steps. Matiisen et al. (2020) propose several curriculum methods; in particular, akin to LHRM, the so-called *online* method keeps an estimate of each task’s average return, but it is not applied in an HRL scenario. Wang et al. (2020) introduce a method that initially learns linear temporal logic formulas for simple tasks, and progressively switches to harder ones leveraging previously learned formulas using a set of predefined templates.

9.4 Symmetry Breaking

The symmetry breaking mechanism proposed in this thesis has been shown to help decrease the time needed to find an (H)RM that covers a set of examples. In this section, we briefly survey recent works on breaking symmetries in graphs (specifically, finite-state machines) or that use ASP to encode problems.

SAT-based approaches to learning DFA have used symmetry breaking constraints to shrink the search space. Heule and Verwer (2010) reduce the DFA learning problem to a graph coloring problem, which is translated into SAT. The graph to be colored is derived from the examples’ Prefix Tree Acceptor (PTA, see p. 160) by connecting two states if they cannot be merged. The vertices in a k -clique must be colored differently, and there are $k!$ different ways of coloring them. The authors propose to break these symmetries by imposing a way of assigning colors after finding a large clique using an approximation algorithm (given that the problem is NP-complete). In contrast, graph indexings similar to ours based on search algorithms like breadth-first search (BFS; Ulyantsev et al., 2015; Zakirzyanov et al., 2019) and depth-first search (DFS; Ulyantsev et al., 2016) have also been proposed. State-merging approaches to learning DFA have also used BFS to break symmetries (Lambeau et al., 2008). These BFS-based methods are different from ours in that they do not need to define a comparison criteria for sets of symbols since they are applied to DFA; indeed, the edges in a DFA are labeled by a single symbol, whereas the edges in our RMs might be labeled by formulas with more than one symbol. Unlike previous works, we prove that the indexing given by our mechanism is unique.

Given the successes of symmetry breaking in SAT solving, the ASP community has also produced some work on symmetry breaking. Drescher et al. (2011) propose SBASS, a system that detects symmetries in a ground ASP program through a reduction to a graph automorphism problem; then, constraints are introduced to the initial program to break the detected symmetries.

Codish et al. (2019) propose a method that breaks symmetries in undirected graphs by imposing a lexicographical order in the rows of the adjacency matrix.

In our previous work (Furelos-Blanco et al., 2020), we introduced a method for breaking symmetries in *acyclic* reward machines, which consists of (i) assigning an integer index to each RM state such that u^0 has the lowest index and u^A and u^R have the highest indices, and (ii) enforcing traversals to go through the RM states in increasing index order. However, this method has a fundamental problem: it cannot break symmetries when no trace traverses all states in the RM (e.g., if there are two different paths to the accepting state), as shown in the following example.

Example 9.4.1. *Figure 3.4 shows two RMs whose states u^1, u^2 and u^3 can be used interchangeably if no symmetry breaking is used. If indices $0, \dots, 3$ are respectively assigned to states u^0, \dots, u^3 and the symmetry breaking mechanism described above is applied, the learned RM for VISITABCD will always be the one shown in Figure 3.4a. In contrast, the RM states u^1 and u^2 in Figure 3.4b can still be switched since there is no traversal that goes through both of them.*

The method presented in this thesis does not depend on the sequence of states visited by the traces; thus, it can break both symmetries in the example.

Chapter 10

Conclusion

In this final chapter, we summarize the key contributions presented in this thesis and discuss potential avenues for future work.

10.1 Summary of Contributions

Reward machines (RMs) are finite-state machines that represent a task’s reward function. By revealing them, an agent can effectively learn policies over histories and perform task decomposition. As a result, agents can tackle tasks with history-dependent rewards and become more sample-efficient. However, their applicability and advantages are limited by the complexity of handcrafting them, the use of algorithms for exploiting them at a single timescale, and their lack of composability.

The *objective* of this thesis was to expand the applicability and enhance the benefits of reward machines by learning them from traces, exploiting them at multiple timescales, and hierarchically composing them. Our contributions towards addressing this objective were broken down into two parts. In what follows, we summarize the core contributions of each of these parts.

In Part I, we devised techniques for learning and exploiting reward machines. We formalized two methods for *exploiting* RMs using the options framework: a novel method for exploiting RMs at multiple timescales, and an existing one for exploiting RMs at a single timescale. In the latter case, we introduced reward shaping mechanisms based on the distance to the accepting state. We proposed a method for *learning* RMs from traces of high-level events using a state-of-the-art inductive logic programming system. We enhanced the performance of the learning system through a *symmetry breaking* mechanism that enforces a unique way of indexing the states and edges of RMs, which shrinks the search space by ruling out equivalent solutions. We designed an algorithm that *interleaves* the exploitation and learning of the RMs. The algorithm learns RMs that are *minimal* given a maximum number of edges from one state to another. We proved that an RM covering all counterexamples is eventually learned. We *experimentally* showed that exploiting learned RMs and handcrafted RMs leads to similar performance.

In Part II, we introduced hierarchies of reward machines (HRMs), a *formalism* for hierarchically composing reward machines by enabling them to call each other. We proved that (i) any HRM can be mapped to an *equivalent* flat HRM (i.e., an HRM that behaves exactly like standard RMs), and (ii) under certain conditions, an equivalent flat HRM can have *exponentially* more states and

edges. We devised an algorithm for *exploiting* HRMs at arbitrarily many timescales, enabling the reusability of subtasks across different tasks. We presented a curriculum-based method for *learning* HRMs from traces given a set of composable tasks. Like in Part I, the learning of the HRMs is interleaved with their exploitation. We *evaluated* the different components of our approach across diverse domains, showing that (i) exploiting a handcrafted HRM enables faster convergence than exploiting an equivalent flat HRM, (ii) exploiting HRMs is more efficient than using other forms of memory (e.g., LSTMs), and (iii) in accordance with the theory, learning an HRM is feasible in cases where a flat equivalent HRM is not.

10.2 Future Work

In this section, we present several directions for future work in the field of reward machines.

Expressiveness

Reward machines, as stated by Toro Icarte et al. (2018a), capture *regular languages*. The expressiveness of these languages (and, hence, RMs) is limited; for instance, they cannot express tasks involving counting, such as “collect n coffees, then collect n spoons”. Future research can be directed towards designing RMs resembling the automata that capture more complex languages (e.g., context-free and context-sensitive languages) and devising methods for exploiting and learning them. The ILASP system has been previously used to learn context-sensitive grammars (Law et al., 2019).

An alternative path to making RMs more expressive and abstract is using *first-order logic*. First-order logic enables expressing conditions more compactly. For instance, a task such as “visit one of A , B , C and D ” in OFFICEWORLD requires a 2-state RM with four edges from the initial to the accepting state; in contrast, by defining an atom `room(X)` where $X \in \{A, B, C, D\}$, a single edge labeled with `room(X)` is needed. The learning of the RMs can be performed much faster by leveraging the compactness enabled by first-order logic; nevertheless, it requires injecting further human knowledge. Our methods are extensible to the first-order case since ILASP can learn first-order rules, and the exploitation algorithms are independent of how edge conditions are expressed (e.g., the policy associated with the formula `room(X)` is satisfied upon observing A , B , C or D). Zhou and Li (2022a) recently made a step in this direction by allowing RM transitions over predicates.

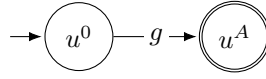
The role of *rejecting states* in our hierarchies can also be reconsidered. In this thesis, we assumed that the rejecting states of the constituent RMs in an HRM are global; that is, reaching any rejecting state makes succeeding at completing the task unfeasible. Rejecting states could also be local, only making a particular RM fail; hence, this would involve defining what caller RMs should do in these situations, and reworking the exploitation and learning algorithms accordingly. Investigating whether this assumption—or any other assumptions—limits the expressiveness of HRMs with respect to standard RMs (e.g., capturing a subset of languages) is an interesting research direction. Further compelling avenues include exploring the relaxation of other formalism aspects, such as determinism.

Learning Minimal Reward Machines from Positive and Negative Examples

The methods proposed in this thesis to learn RMs, ISA and LHRM, focus on learning minimal RMs only from observable traces (i.e., positive examples). As discussed in Sections 5.1.1 and 8.3.2, it

is known that learning minimal machines exclusively from positive examples can lead to overgeneralization. In general, our method suffers from this issue when two propositional events are (or seem, as exemplified in Figure 5.1) temporally dependent; that is, when an event is observable only after having observed another one. To address this issue, RMs must also be learned from *negative examples* (i.e., unfeasible traces); therefore, our methods must be extended to use these examples. The following example illustrates the problem.

Example 10.2.1. *Let us consider that the proposition set \mathcal{P} for OFFICEWORLD includes a proposition g observed upon completing a goal history. In the case of COFFEE, g would be observed after visiting the ☕ and o locations, thus a possible goal trace is $\langle\{\}, \{\text{☕}\}, \{\}, \{o, g\}\rangle$. The RM below, which is minimal, could then be learned. The learner will not refine this RM since there are no positive examples that contradict this RM; namely, there is no trace where g can be observed without reaching the goal. On the other hand, an incomplete (but unobservable) trace such as $\langle\{g\}\rangle$ would trigger the learning of a new RM.*



Existing approaches learn minimal RMs from negative examples. Hasanbeig et al. (2021) derive these examples from sequences appearing in the RM but not in the traces. Gaon and Brafman (2020) use an active learning algorithm that asks membership queries (i.e., whether a trace is a positive or negative example), which are answered based on the agent’s experience (i.e., whether the trace has been observed or not). Future work could consider answering these queries by leveraging RNNs akin to Weiss et al. (2018); indeed, Michalenko et al. (2019) have shown that there is a close relationship between the internal representations used by RNNs and finite-state machines.

Scaling Up Learning

Learning minimal finite-state machines from examples is a hard problem (Gold, 1978). The main factor limiting our methods’ scalability is the number of states. In this thesis, we have devised two core techniques for improving scalability: (i) a symmetry breaking method, which enforces a unique indexing of the states and edges, and (ii) the use of hierarchies of RMs, which enables learning RMs with fewer states by enabling calls to previously learned RMs. In what follows, we outline other techniques to scale up our methods.

The structures of the RMs (and, hence, the tasks) considered throughout the thesis are often similar, e.g. states may be arranged in sequences (i.e., perform tasks in a fixed order) or diamond-shaped structures (i.e., perform tasks in any order). The learning of a reward machine could potentially leverage *templates* encoding these common arrangements. For instance, a template $\text{seq}(\mathbf{X}, \mathbf{Y})$ could define a 3-state RM such that $u^0 \xrightarrow{\mathbf{X}} u^1 \xrightarrow{\mathbf{Y}} u^A$; then, the RM M_1 in Figure 6.2b could instead be represented by a 2-state RM whose transition is labeled with an instance of the template, i.e. $u^0 \xrightarrow{\text{seq}(\mathbf{r}, \mathbf{g})} u^A$. Methods for learning RMs can potentially benefit from templates since the set of states becomes smaller. Wang et al. (2020) propose a similar approach for temporal logic formulas.

Even though the number of states is the main limiting factor towards scalability, several other aspects have a substantial impact on performance, as experimentally shown in Sections 5.2.5 and 8.2. The length of the traces is one of them: the longer the example traces are, the harder it is to

learn the RM. In this thesis, we address this problem by compressing the traces and learning an initial HRM from the shortest traces of a randomly collected set, whereas Hasanbeig et al. (2021) segment traces using a fixed-size sliding window. Designing methods for compressing traces is an interesting path for future investigation; however, it is imperative to state under which assumptions such compression schemes operate. Furthermore, strategies for refining a set of traces could be considered to improve performance; for instance, discarding a complex counterexample (i.e., long and containing irrelevant propositions to the task at hand) if it seems subsumed by a simpler one.

Learning in Continual Settings

In this thesis, reward machines have been learned in scenarios where (i) the agent-environment interaction lasts for a finite number of steps, (ii) given a state-action history, the associated reward and termination information is always the same (i.e., the reward and termination functions are stationary), and (iii) the propositional high-level events are known. Furthermore, in the case of LHRM, a list of composable tasks is provided to the agent. In the real world, agents may need to adapt continuously to changing environments for an indefinite period of time (Dulac-Arnold et al., 2021). In our framework, the propositions handcrafted by a human expert and the learned RMs can eventually become obsolete in such settings; therefore, despite the potential loss of interpretability, it is crucial to progressively remove these assumptions.

Endowing agents with the ability to dynamically define high-level events on their own is a potential first step. In general, we devise methods based on detecting objects and derive events from their interaction (e.g., if they get near each other). Kulkarni et al. (2019) recently proposed an approach for learning temporally consistent object keypoints in RL, which could constitute the starting point for this research direction. Learning the proposition set from the agent-environment interaction implies learning the labeling function as well; that is, the agent must also learn a mapping from states to sets of high-level events. The resulting labeling function is inherently noisy since the event detection algorithm is likely imperfect. The standard RM formalism assumes both the labeling and transition functions to be deterministic; hence, both the formalism and the algorithms need to be adapted to support noisy labeling functions. Li et al. (2022) and Verginis et al. (2022) have recently proposed approaches for exploiting (and learning, in the latter case) RMs in these contexts; however, to the best of our knowledge, learning the proposition set and the labeling function for RMs remains unexplored. Our RM learning method is extensible to the noisy setting since ILASP supports noisy examples. In parallel, developing agents that autonomously propose their own composable tasks represented as HRMs is also a stepping stone to more autonomous agents.

We refer the reader to the work by Khetarpal et al. (2022) for a review of different approaches to RL in continual settings.

Bibliography

- E. Abadi and R. I. Brafman. Learning and Solving Regular Decision Processes. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1948–1954, 2020.
- D. Abel. *A Theory of Abstraction in Reinforcement Learning*. PhD thesis, Brown University, 2020.
- D. Abel, W. Dabney, A. Harutyunyan, M. K. Ho, M. L. Littman, D. Precup, and S. Singh. On the Expressivity of Markov Reward. In *Proceedings of the 35th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 7799–7812, 2021.
- D. Andre and S. J. Russell. Programmable Reinforcement Learning Agents. In *Proceedings of the 14th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1019–1025, 2000.
- J. Andreas, D. Klein, and S. Levine. Modular Multitask Reinforcement Learning with Policy Sketches. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 166–175, 2017.
- D. Angluin. Inductive Inference of Formal Languages from Positive Data. *Information and Control*, 45(2):117–135, 1980.
- D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
- B. Araki, X. Li, K. Vodrahalli, J. A. DeCastro, M. J. Fry, and D. Rus. The Logical Options Framework. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 307–317, 2021.
- L. Ardon, D. Furelos-Blanco, and A. Russo. Learning Reward Machines in Cooperative Multi-Agent Tasks. In *Proceedings of the Neuro-Symbolic AI for Agent and Multi-Agent Systems (NeSyMAS) Workshop at the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2023.
- K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.
- K. J. Åström. Optimal Control of Markov Processes with Incomplete State Information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965.
- F. Bacchus, C. Boutilier, and A. J. Grove. Rewarding Behaviors. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1160–1167, 1996.

- P. Bacon, J. Harb, and D. Precup. The Option-Critic Architecture. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 1726–1734, 2017.
- A. G. Barto and S. Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.
- H. Bastani, K. Drakopoulos, V. Gupta, I. Vlachogiannis, C. Hadjichristodoulou, P. Lagiou, G. Magiorkinis, D. Paraskevis, and S. Tsiodras. Efficient and targeted COVID-19 border testing via reinforcement learning. *Nature*, 599(7883):108–113, 2021.
- M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying Count-Based Exploration and Intrinsic Motivation. In *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1471–1479, 2016.
- M. G. Bellemare, S. Candido, P. S. Castro, J. Gong, M. C. Machado, S. Moitra, S. S. Ponda, and Z. Wang. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588(7836):77–82, 2020.
- R. E. Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.
- Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum Learning. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 41–48, 2009.
- B. Bonet, H. Palacios, and H. Geffner. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 34–41, 2009.
- H. Bourel, A. Jonsson, O.-A. Maillard, and M. Sadegh Talebi. Exploration in Reward Machines with Low Regret. In *Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 4114–4146, 2023.
- S. J. Bradtke and M. O. Duff. Reinforcement Learning Methods for Continuous-Time Markov Decision Problems. In *Proceedings of the 8th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 393–400, 1994.
- R. I. Brafman and G. De Giacomo. Regular Decision Processes: A Model for Non-Markovian Domains. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5516–5522, 2019.
- R. I. Brafman and M. Tennenholtz. R-MAX – A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning Research*, 3:213–231, 2002.
- R. A. Brooks. A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network. *Neural Computation*, 1(2):253–262, 1989.
- M. Buckland. *Programming Game AI by Example*. Jones & Bartlett Learning, 2004.
- F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub. ASP-Core-2 Input Language Format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020.

- A. Camacho, O. Chen, S. Sanner, and S. A. McIlraith. Non-Markovian Rewards Expressed in LTL: Guiding Search Via Reward Shaping. In *Proceedings of the 10th International Symposium on Combinatorial Search (SOCS)*, pages 159–160, 2017.
- A. Camacho, R. Toro Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 6065–6073, 2019.
- A. Camacho, J. Varley, A. Zeng, D. Jain, A. Iscen, and D. Kalashnikov. Reward Machines for Vision-Based Robotic Manipulation. In *Proceedings of the 38th IEEE International Conference on Robotics and Automation (ICRA)*, pages 14284–14290, 2021.
- M. Chevalier-Boisvert, D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen, and Y. Bengio. BabyAI: A Platform to Study the Sample Efficiency of Grounded Language Learning. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.
- M. Chevalier-Boisvert, B. Dai, M. Towers, R. de Lazcano, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. In *Proceedings of the 37th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- P. J. K. Christoffersen, A. C. Li, R. Toro Icarte, and S. A. McIlraith. Learning Symbolic Representations for Reinforcement Learning of Non-Markovian Behavior. In *Proceedings of the 4th Knowledge Representation and Reasoning Meets Machine Learning Workshop (KR2ML) at the 34th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- K. L. Clark. Negation as Failure. In *Logic and Data Bases*, pages 293–322, 1977.
- M. Codish, A. Miller, P. Prosser, and P. J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints*, 24(1):1–24, 2019.
- J. Corazza, I. Gavran, and D. Neider. Reinforcement Learning with Stochastic Reward Machines. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 6429–6436, 2022.
- W. Dabney, G. Ostrovski, and A. Barreto. Temporally-Extended ϵ -Greedy Exploration. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*, 2021.
- M. Dann, Y. Yao, N. Alechina, B. Logan, and J. Thangarajah. Multi-Agent Intention Progression with Reward Machines. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 215–222, 2022.
- J. S. de Cani. A Dynamic Programming Algorithm for Embedded Markov Chains when the Planning Horizon is at Infinity. *Management Science*, 10(4):716–733, 1964.
- G. De Giacomo, M. Favorito, L. Iocchi, F. Patrizi, and A. Ronca. Temporal Logic Monitoring Rewards via Transducers. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 860–870, 2020.

- C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
- J. Degraeve, F. Felici, J. Buchli, M. Neunert, B. D. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. de Las Casas, C. Donner, L. Fritz, C. Galperti, A. Huber, J. Keeling, M. Tsim-poukelli, J. Kay, A. Merle, J. Moret, S. Noury, F. Pesamosca, D. Pfau, O. Sauter, C. Sommariva, S. Coda, B. Duval, A. Fasoli, P. Kohli, K. Kavukcuoglu, D. Hassabis, and M. A. Riedmiller. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897): 414–419, 2022.
- F. den Hengst, V. François-Lavet, M. Hoogendoorn, and F. van Harmelen. Reinforcement Learning with Option Machines. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2909–2915, 2022.
- T. G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.
- T. G. Dietterich, P. M. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli. Structured machine learning: the next ten years. *Machine Learning*, 73(1):3–23, 2008.
- T. Dohmen, N. Topper, G. K. Atia, A. Beckus, A. Trivedi, and A. Velasquez. Inferring Probabilistic Reward Machines from Non-Markovian Reward Signals for Reinforcement Learning. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 574–582, 2022.
- C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking Answer Set Solving. *AI Communications*, 24(2):177–194, 2011.
- G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.
- T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.
- J. Fu and U. Topcu. Probably Approximately Correct MDP Learning and Control With Temporal Logic Constraints. In *Proceedings of the 10th Robotics: Science and Systems Conference (RSS)*, 2014.
- D. Furelos-Blanco, M. Law, A. Russo, K. Broda, and A. Jonsson. Induction of Subgoal Automata for Reinforcement Learning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 3890–3897, 2020.
- D. Furelos-Blanco, M. Law, A. Jonsson, K. Broda, and A. Russo. Induction and Exploitation of Subgoal Automata for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 70: 1031–1116, 2021.
- D. Furelos-Blanco, M. Law, A. Jonsson, K. Broda, and A. Russo. Hierarchies of Reward Machines. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 10494–10541, 2023.

- M. Gaon and R. I. Brafman. Reinforcement Learning with Non-Markovian Rewards. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 3980–3987, 2020.
- M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, and P. Wanko. Potassco User Guide – Version 2.2.0, 2019. URL <https://github.com/potassco/guide/releases>.
- M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP)*, pages 1070–1080, 1988.
- M. Ghavamzadeh. *Hierarchical Reinforcement Learning in Continuous State and Multi-agent Environments*. PhD thesis, University of Massachusetts Amherst, 2005.
- F. W. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.
- E. M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.
- I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. MIT Press, 2016.
- M. Hasanbeig, N. Y. Jeppu, A. Abate, T. Melham, and D. Kroening. DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI)*, pages 7647–7656, 2021.
- S. W. Hasinoff. Reinforcement Learning for Problems with Hidden State. Technical report, University of Toronto, Department of Computer Science, September 2003. URL <https://people.csail.mit.edu/hasinoff/pubs/hasinoff-rlhidden-2002.pdf>.
- M. J. Hausknecht and P. Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. In *Proceedings of the AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*, pages 29–37, 2015.
- M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 3215–3222, 2018.
- M. Heule and S. Verwer. Exact DFA Identification Using SAT Solvers. In *Proceedings of the 10th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI)*, pages 66–79, 2010.
- G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning – Lecture 6e – RMSprop: Divide the Gradient by a Running Average of its Recent Magnitude. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2012a.
- G. E. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012b.

- M. K. Ho, D. Abel, T. L. Griffiths, and M. L. Littman. The value of abstraction. *Current Opinion in Behavioral Sciences*, 29:111–116, 2019.
- S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
- Y. Hu and G. De Giacomo. A Generic Technique for Synthesizing Bounded Finite-State Controllers. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 109–116, 2013.
- M. Igl, K. Ciosek, Y. Li, S. Tschiatschek, C. Zhang, S. Devlin, and K. Hofmann. Generalization in Reinforcement Learning with Selective Noise Injection and Information Bottleneck. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, pages 13956–13968, 2019.
- A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 428–437, 2018.
- L. Illanes, X. Yan, R. Toro Icarte, and S. A. McIlraith. Symbolic Plans as High-Level Instructions for Reinforcement Learning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 540–550, 2020.
- N. Y. Jeppu, T. F. Melham, D. Kroening, and J. O’Leary. Learning Concise Models from Long Execution Traces. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.
- W. S. Jewell. Markov-Renewal Programming. I: Formulation, Finite Return Models. *Operations Research*, 11(6):938–948, 1963.
- M. Jiang, E. Grefenstette, and T. Rocktäschel. Prioritized Level Replay. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 4940–4950, 2021.
- Y. Jinnai, J. W. Park, D. Abel, and G. D. Konidaris. Discovering Options for Exploration by Minimizing Cover Time. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 3130–3139, 2019.
- K. Jothimurugan, R. Alur, and O. Bastani. A Composable Specification Language for Reinforcement Learning Tasks. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, pages 13021–13030, 2019.
- L. P. Kaelbling. Learning to Achieve Goals. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1094–1099, 1993.
- L. P. Kaelbling. The foundation of efficient robot learning. *Science*, 369(6506):915–916, 2020.
- L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

- S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent Experience Replay in Distributed Reinforcement Learning. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.
- J. Karlsson. Task Decomposition in Reinforcement Learning. In *Proceedings of the AAAI Spring Symposium on Goal-Driven Learning*, pages 46–53, 1994.
- A. Karpathy. REINFORCEjs: WaterWorld demo, 2015. URL <http://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html>. Accessed: 2023-09-07.
- K. Kheterpal, M. Riemer, I. Rish, and D. Precup. Towards Continual Reinforcement Learning: A Review and Perspectives. *Journal of Artificial Intelligence Research*, 75:1401–1476, 2022.
- D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
- M. Klissarov and M. C. Machado. Deep Laplacian-based Options for Temporally-Extended Exploration. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 17198–17217, 2023.
- G. Konidaris. On the necessity of abstraction. *Current Opinion in Behavioral Sciences*, 29:1–7, 2019.
- A. Koul, A. Fern, and S. Greydanus. Learning Finite State Representations of Recurrent Policy Networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.
- A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 26th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1106–1114, 2012.
- T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. In *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 3675–3683, 2016.
- T. D. Kulkarni, A. Gupta, C. Ionescu, S. Borgeaud, M. Reynolds, A. Zisserman, and V. Mnih. Unsupervised Learning of Object Keypoints for Perception and Control. In *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 10723–10733, 2019.
- B. Lambeau, C. Damas, and P. Dupont. State-Merging DFA Induction Algorithms with Mandatory Merge Constraints. In *Proceedings of the 9th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI)*, pages 139–153, 2008.
- G. Lample and D. S. Chaplot. Playing FPS Games with Deep Reinforcement Learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 2140–2146, 2017.

- K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI)*, pages 1–12, 1998.
- R. T. Lange and A. Faisal. Semantic RL with Action Grammars: Data-Efficient Learning of Hierarchical Task Abstractions. In *Proceedings of the Deep Reinforcement Learning Workshop at the 33rd Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
- M. Law. Logic-based Learning, 2015. URL <https://www.doc.ic.ac.uk/~ml1909/teaching>. Accessed: 2023-09-07.
- M. Law, A. Russo, and K. Broda. The ILASP System for Learning Answer Set Programs, 2015a. URL <https://www.ilasp.com>.
- M. Law, A. Russo, and K. Broda. Simplified Reduct for Choice Rules in ASP. Technical report, DTR2015-2, Imperial College London, Department of Computing, April 2015b. URL <https://www.doc.ic.ac.uk/research/technicalreports/2015/DTR15-2.pdf>.
- M. Law, A. Russo, and K. Broda. Iterative Learning of Answer Set Programs from Context Dependent Examples. *Theory and Practice of Logic Programming*, 16(5-6):834–848, 2016.
- M. Law, A. Russo, and K. Broda. The Meta-program Injection Feature in ILASP. Technical report, Imperial College London, Department of Computing, June 2018. URL <https://www.doc.ic.ac.uk/~ml1909/ILASP/inject.pdf>.
- M. Law, A. Russo, E. Bertino, K. Broda, and J. Lobo. Representing and Learning Grammars in Answer Set Programming. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 2919–2928, 2019.
- B. G. León, M. Shanahan, and F. Belardinelli. Systematic Generalisation through Task Temporal Logic and Deep Reinforcement Learning. *arXiv preprint*, arXiv:2006.08767, 2020.
- B. G. León, M. Shanahan, and F. Belardinelli. In a Nutshell, the Human Asked for This: Latent Goals for Following Temporal Specifications. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, 2022.
- M. Leonetti, L. Iocchi, and F. Patrizi. Automatic Generation and Learning of Finite-State Controllers. In *Proceedings of the 15th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, pages 135–144, 2012.
- A. Levy, G. D. Konidaris, R. Platt, and K. Saenko. Learning Multi-Level Hierarchies with Hindsight. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.
- A. C. Li, Z. Chen, P. Vaezipoor, T. Q. Klassen, R. Toro Icarte, and S. A. McIlraith. Noisy Symbolic Abstractions for Deep RL: A case study with Reward Machines. In *Proceedings of the Deep Reinforcement Learning Workshop at the 36th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

- T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous Control with Deep Reinforcement Learning. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.
- L. J. Lin. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8:293–321, 1992.
- M. L. Littman. The Reward Hypothesis. <https://www.coursera.org/lecture/fundamentals-of-reinforcement-learning/michael-littman-the-reward-hypothesis-q6x0e>, 2018. Accessed: 2022-11-20.
- S. Lobel, A. Bagaria, and G. Konidaris. Flipping Coins to Estimate Pseudocounts for Exploration in Reinforcement Learning. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 22594–22613, 2023.
- M. C. Machado, M. G. Bellemare, and M. H. Bowling. A Laplacian Framework for Option Discovery in Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 2295–2304, 2017.
- M. C. Machado, M. G. Bellemare, and M. Bowling. Count-Based Exploration with the Successor Representation. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 5125–5133, 2020.
- T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher-Student Curriculum Learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3732–3740, 2020.
- A. McGovern and A. G. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *Proceedings of the 18th International Conference on Machine Learning (ICML)*, pages 361–368, 2001.
- N. Mehta, S. Ray, P. Tadepalli, and T. G. Dietterich. Automatic Discovery and Transfer of MAXQ Hierarchies. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 648–655, 2008.
- F. Memarian, Z. Xu, B. Wu, M. Wen, and U. Topcu. Active Task-Inference-Guided Deep Inverse Reinforcement Learning. In *Proceedings of the 59th IEEE Conference on Decision and Control (CDC)*, pages 1932–1938, 2020.
- I. Menache, S. Mannor, and N. Shimkin. Q-Cut - Dynamic Discovery of Sub-goals in Reinforcement Learning. In *Proceedings of the 13th European Conference on Machine Learning (ECML)*, pages 295–306, 2002.
- N. Meuleau, L. Peshkin, K. Kim, and L. P. Kaelbling. Learning Finite-State Controllers for Partially Observable Environments. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 427–436, 1999.
- J. J. Michalenko, A. Shah, A. Verma, R. G. Baraniuk, S. Chaudhuri, and A. B. Patel. Representing Formal Languages: A Comparison Between Finite Automata and Recurrent Neural Networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

- S. Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. In *Proceedings of the 7th AAAI Conference on Artificial Intelligence (AAAI)*, pages 564–569, 1988.
- V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- S. H. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.
- G. Nangue Tasse, S. D. James, and B. Rosman. A Boolean Task Algebra for Reinforcement Learning. In *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*, pages 9497–9507, 2020.
- G. Nangue Tasse, D. Jarvis, S. James, and B. Rosman. Skill Machines: Temporal Logic Composition in Reinforcement Learning. *arXiv preprint*, arXiv:2205.12532, 2022.
- C. Neary, Z. Xu, B. Wu, and U. Topcu. Reward Machines for Cooperative Multi-Agent Reinforcement Learning. In *Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 934–942, 2021.
- A. Y. Ng, D. Harada, and S. Russell. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *Proceedings of the 16th International Conference on Machine Learning (ICML)*, pages 278–287, 1999.
- R. Parr. *Hierarchical Control and Learning in Markov Decision Processes*. PhD thesis, University of California, Berkeley, 1998.
- R. Parr and S. Russell. Reinforcement Learning with Hierarchies of Machines. In *Proceedings of the 11th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1043–1049, 1997.
- T. Pierrot, G. Ligner, S. E. Reed, O. Sigaud, N. Perrin, A. Laterre, D. Kas, K. Beguir, and N. de Freitas. Learning Compositional Neural Programs with Recursive Tree Search and Planning. In *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 14646–14656, 2019.
- D. Precup. *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 2001.
- M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
- J. Segovia-Aguas, S. Jiménez, and A. Jonsson. Computing Hierarchical Finite State Controllers With Classical Planning. *Journal of Artificial Intelligence Research*, 62:755–797, 2018.
- M. Sergot. Knowledge Representation, 2017. URL <https://www.doc.ic.ac.uk/~mjs/teaching/491.html>. Accessed: 2023-09-07.
- M. Shanahan and M. Mitchell. Abstraction for Deep Reinforcement Learning. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5588–5596, 2022.

- S. Sidor. Reinforcement Learning with Natural Language Signals. Master’s thesis, Massachusetts Institute of Technology, 2016.
- D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- Ö. Şimşek and A. G. Barto. Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, 2004.
- Ö. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying Useful Subgoals in Reinforcement Learning by Local Graph Partitioning. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pages 816–823, 2005.
- S. Singh, A. G. Barto, and N. Chentanez. Intrinsically Motivated Reinforcement Learning. In *Proceedings of the 18th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1281–1288, 2004.
- M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
- S. Sohn, J. Oh, and H. Lee. Hierarchical Reinforcement Learning for Zero-shot Generalization with Subtask Dependencies. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*, pages 7156–7166, 2018.
- M. T. J. Spaan. Partially Observable Markov Decision Processes. In *Reinforcement Learning, Adaptation, Learning, and Optimization*, pages 387–414. Springer, 2012.
- A. Srinivasan. The Aleph Manual, 2001. URL <https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html>. Accessed: 2020-07-07.
- M. Stolle and D. Precup. Learning Options in Reinforcement Learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 212–223, 2002.
- S. Sun, T. Wu, and J. J. Lim. Program Guided Agent. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.
- I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 28th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 3104–3112, 2014.
- R. S. Sutton. The Reward Hypothesis. <http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/rewardhypothesis.html>, 2004. Accessed: 2022-11-20.
- R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.
- R. S. Sutton, D. Precup, and S. P. Singh. Intra-Option Learning about Temporally Abstract Actions. In *Proceedings of the 15th International Conference on Machine Learning (ICML)*, pages 556–564, 1998.

- R. S. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.
- R. Toro Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 2112–2121, 2018a.
- R. Toro Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith. Teaching Multiple Tasks to an RL Agent using LTL. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 452–461, 2018b.
- R. Toro Icarte, E. Waldie, T. Q. Klassen, R. A. Valenzano, M. P. Castro, and S. A. McIlraith. Learning Reward Machines for Partially Observable Reinforcement Learning. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, pages 15497–15508, 2019.
- R. Toro Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.
- R. Toro Icarte, T. Q. Klassen, R. Valenzano, M. P. Castro, E. Waldie, and S. A. McIlraith. Learning reward machines: A study in partially observable reinforcement learning. *Artificial Intelligence*, 323:103989, 2023.
- L. Torrey, J. W. Shavlik, T. Walker, and R. Maclin. Relational Macros for Transfer in Reinforcement Learning. In *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP)*, pages 254–268, 2007.
- V. Ulyantsev, I. Zakirzyanov, and A. Shalyto. BFS-Based Symmetry Breaking Predicates for DFA Identification. In *Proceedings of the 9th International Conference on Language and Automata Theory and Applications (LATA)*, pages 611–622, 2015.
- V. Ulyantsev, I. Zakirzyanov, and A. Shalyto. Symmetry Breaking Predicates for SAT-based DFA Identification. *arXiv preprint*, arXiv:1602.05028, 2016.
- P. Vaezipoor, A. C. Li, R. Toro Icarte, and S. A. McIlraith. LTL2Action: Generalizing LTL Instructions for Multi-Task RL. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 10497–10508, 2021.
- H. van Hasselt. Double Q-learning. In *Proceedings of the 24th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 2613–2621, 2010.
- H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, pages 2094–2100, 2016.
- C. K. Verginis, C. Köprülü, S. Chinchali, and U. Topcu. Joint Learning of Reward Machines and Policies in Environments with Partially Known Semantics. *arXiv preprint*, arXiv:2204.11833, 2022.

- A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5052–5061, 2018.
- O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, Ç. Gülçehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. P. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.
- J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1947.
- G. Wang, C. Trimbach, J. K. Lee, M. K. Ho, and M. L. Littman. Teaching a Robot Tasks of Arbitrary Complexity via Human Feedback. In *Proceedings of the 15th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 649–657, 2020.
- C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.
- G. Weiss, Y. Goldberg, and E. Yahav. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5244–5253, 2018.
- S. D. Whitehead and L. J. Lin. Reinforcement Learning of Non-Markov Decision Processes. *Artificial Intelligence*, 73(1-2):271–306, 1995.
- D. Won, K. Müller, and S. Lee. An adaptive deep reinforcement learning framework enables curling robots with human-like performance in real-world conditions. *Science Robotics*, 5(46):9764, 2020.
- P. R. Wurman, S. Barrett, K. Kawamoto, J. MacGlashan, K. Subramanian, T. J. Walsh, R. Capobianco, A. Devlic, F. Eckert, F. Fuchs, L. Gilpin, P. Khandelwal, V. Kompella, H. Lin, P. MacAlpine, D. Oller, T. Seno, C. Sherstan, M. D. Thomure, H. Aghabozorgi, L. Barrett, R. Douglas, D. Whitehead, P. Dürr, P. Stone, M. Spranger, and H. Kitano. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228, 2022.
- Z. Xu, I. Gavran, Y. Ahmad, R. Majumdar, D. Neider, U. Topcu, and B. Wu. Joint Inference of Reward Machines and Policies for Reinforcement Learning. *arXiv preprint*, arXiv:1909.05912, 2019.
- Z. Xu, I. Gavran, Y. Ahmad, R. Majumdar, D. Neider, U. Topcu, and B. Wu. Joint Inference of Reward Machines and Policies for Reinforcement Learning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 590–598, 2020.
- Z. Xu, B. Wu, A. Ojha, D. Neider, and U. Topcu. Active Finite Reward Automaton Inference and Reinforcement Learning Using Queries and Counterexamples. In *Proceedings of the International Cross-Domain Conference for Machine Learning and Knowledge Extraction (CD-MAKE)*, pages 115–135, 2021.

- Y. Yang, J. P. Inala, O. Bastani, Y. Pu, A. Solar-Lezama, and M. C. Rinard. Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In *Proceedings of the 35th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 29669–29683, 2021.
- I. Zakirzyanov, A. Morgado, A. Ignatiev, V. Ulyantsev, and J. Marques-Silva. Efficient Symmetry Breaking for SAT-Based Minimum DFA Inference. In *Proceedings of the 13th International Conference on Language and Automata Theory and Applications (LATA)*, pages 159–173, 2019.
- W. Zhou and W. Li. A Hierarchical Bayesian Approach to Inverse Reinforcement Learning with Symbolic Reward Machines. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, pages 27159–27178, 2022a.
- W. Zhou and W. Li. Programmatic Reward Design by Example. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI)*, pages 9233–9241, 2022b.

Appendix A

Reward Machines

In this appendix, we present reward machines for the tasks introduced in Part I of the thesis.

A.1 Examples

Figures 3.2, 3.4a and 3.4b show minimal RMs for the OFFICEWORLD tasks COFFEE, VISITABCD and COFFEEMAIL, respectively. Figure A.1 illustrates minimal RMs for other tasks considered in the ablation experiments from Section 5.2.5. Figures A.2 and A.3 contain minimal RMs for the CRAFTWORLD and WATERWORLD tasks described in Sections 5.3.2 and 5.4.2, respectively.

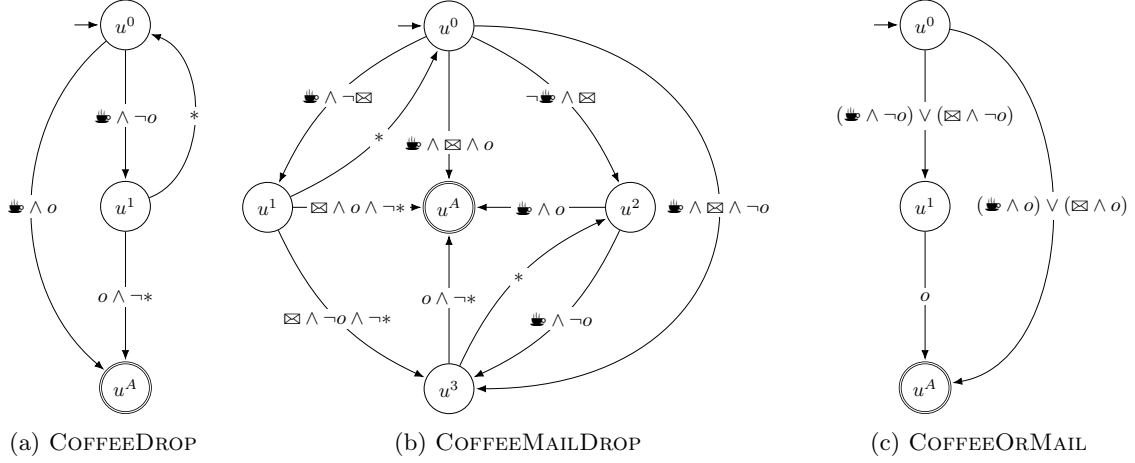


Figure A.1: Reward machines for the OFFICEWORLD tasks in the ablation experiments from Section 5.2.5. For simplicity, the rejecting state is omitted from (c).

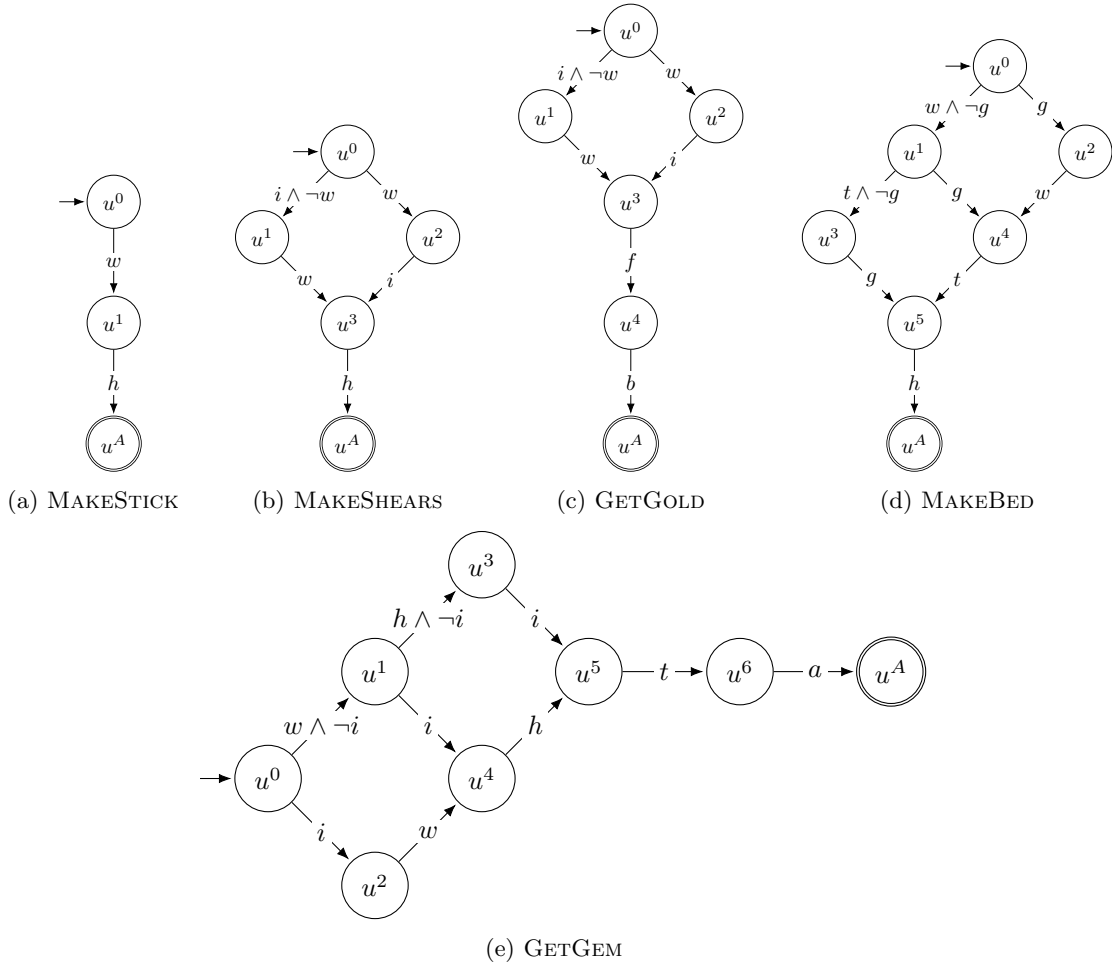
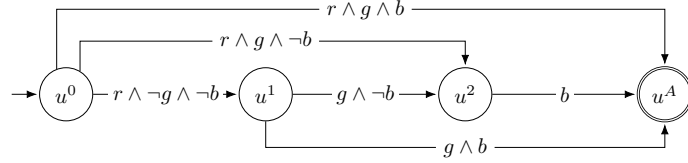
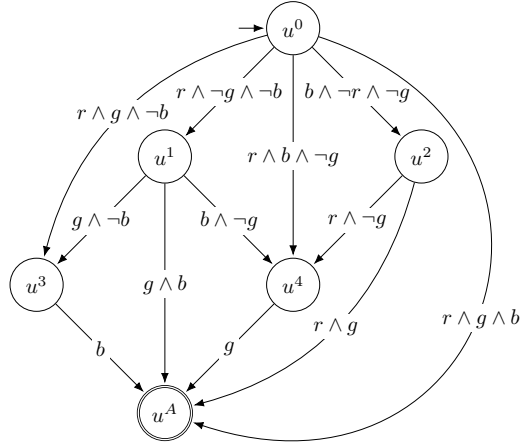


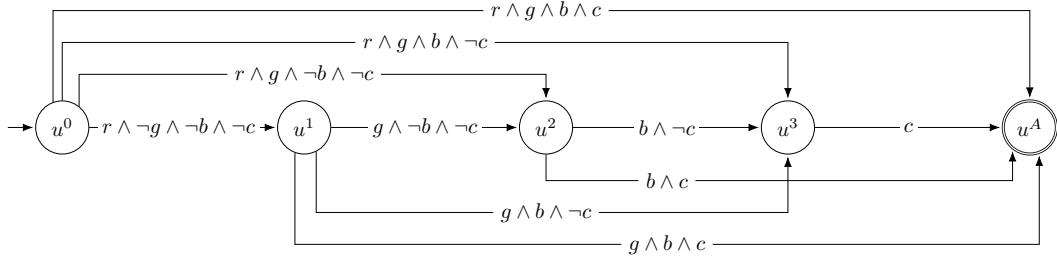
Figure A.2: Reward machines for some of the CRAFTWORLD tasks in Chapter 5. The propositions are a (axe), b (bridge), f (factory), g (grass), h (workbench), i (iron), t (toolshed) and w (wood).



(a) RGB



(b) RG-B



(c) RGBC

Figure A.3: Reward machines for the WATERWORLD tasks in Chapter 5.

Appendix B

Hierarchies of Reward Machines

In this appendix, we present hierarchies of reward machines and extended results for the tasks and experiments in Part II of the thesis.

B.1 Examples

Figures B.1 and B.2 show minimal root RMs for the CRAFTWORLD and WATERWORLD tasks considered in Chapter 8, respectively. In both cases, the RMs correspond to the settings without dead-ends; thus, they do not include rejecting states. In the case of CRAFTWORLD, the mutual exclusivity can be enforced differently since the observed labels will never consist of two or more propositions.

B.2 Learning Non-Flat Hierarchies of Reward Machines

We here provide the tables for the experiments described in Section 8.2. The tables contain the following HRM learning information left-to-right for each task: (i) task name; (ii) number of runs in which at least one goal trace was observed; (iii) number of runs in which at least one HRM was learned; (iv) time spent on learning HRMs; (v) number of calls made to ILASP, i.e. the number of HRMs learned for a given task; (vi) number of states in the root of the final HRM; (vii) number of edges in the root of the final HRM; (viii) number of episodes between the learning of the first HRM and the activation of the task’s level; (ix) number of example traces for each trace type; and (x) length of the example traces for each trace type. Letters *G*, *D* and *I* denote goal, dead-end and incomplete traces, respectively. The bottom of each table shows the number of completed runs (i.e., the number of runs that have not timed out), the total time spent on learning HRMs, and the total number of calls made to ILASP.

In the case of CRAFTWORLD, Table B.1 shows the results in the default experimental setting, whereas Table B.3 outlines the results when the set of callable RMs contains only those actually needed to learn the HRM for a given task, and Table B.5 presents the results of using primitive actions for exploration instead of options. Tables B.2, B.4 and B.6 show analogous results for WATERWORLD.

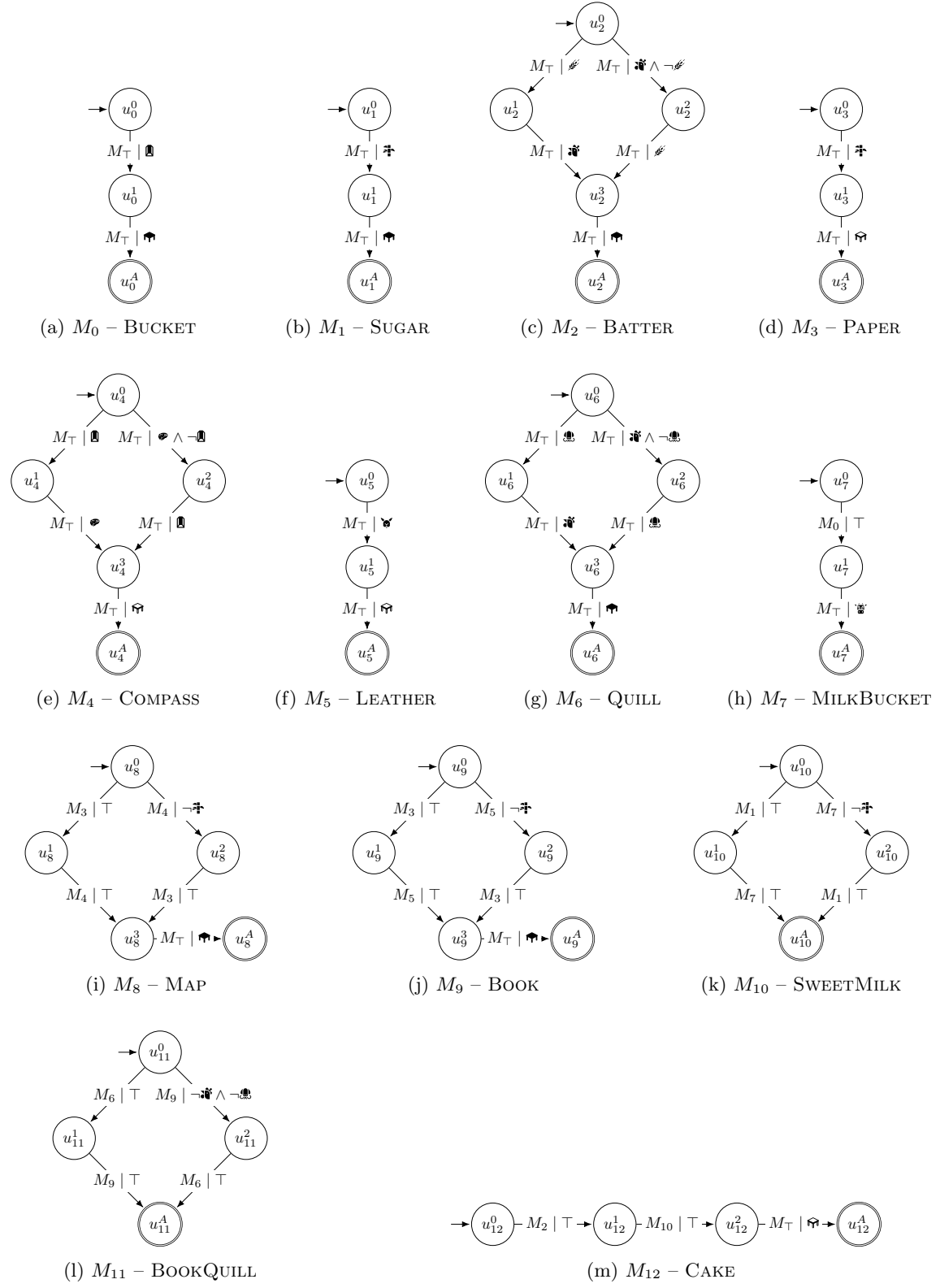


Figure B.1: Root reward machines for each of the CRAFTWORLD tasks in Chapter 8.

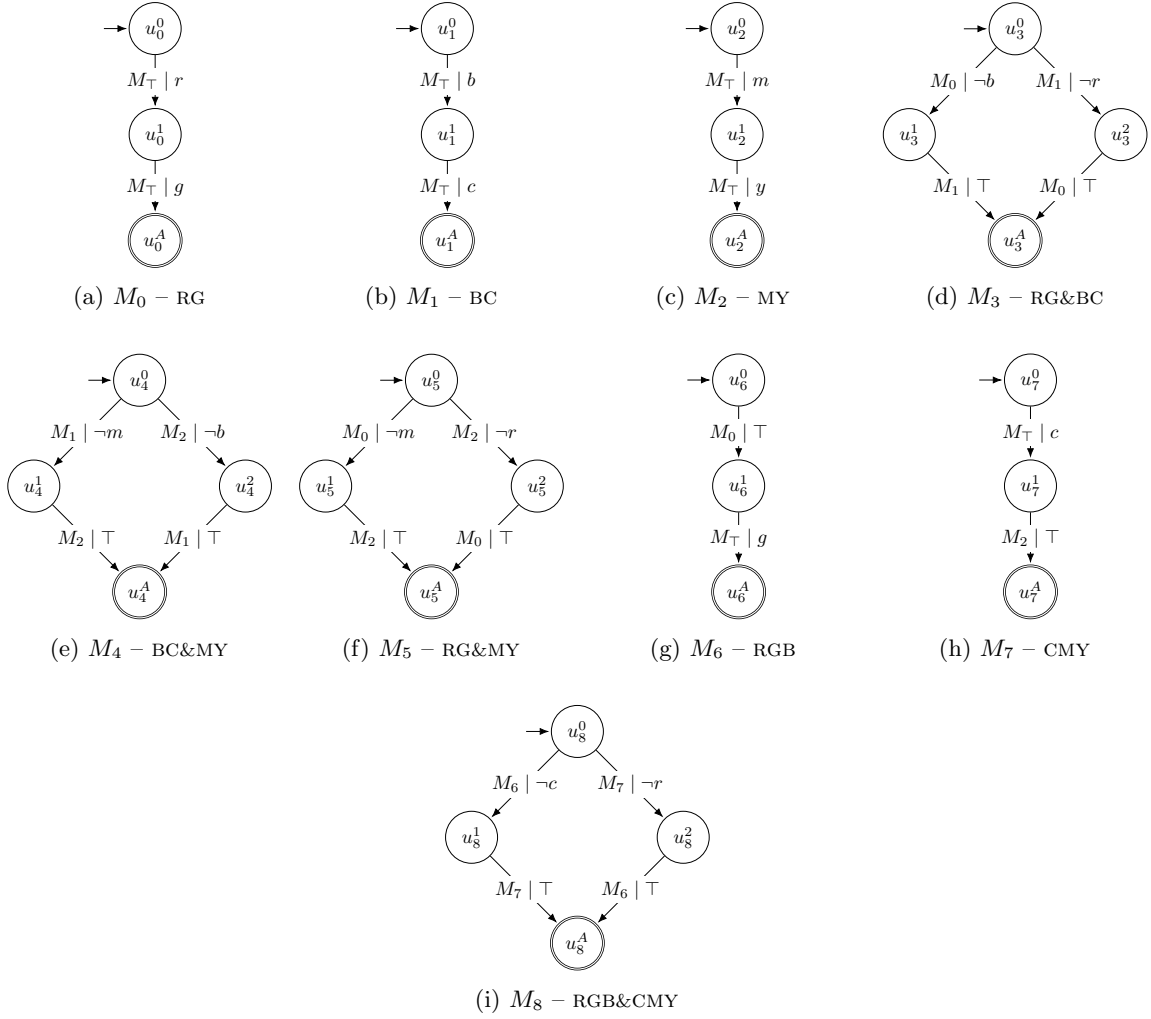


Figure B.2: Root reward machines for each of the WATERWORLD tasks in Chapter 8.

Table B.1: Results of LHRM in CRAFTWORLD for the default case.

	Task	G	L	Time (s.)	Calls	States	Edges	Ep. First ($\times 10^2$)	# Examples			Example Length		
									G	D	I	G	D	I
OP	BATTER	5	5	11.1 \pm 1.7	17.8 \pm 1.9	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.1	12.2 \pm 0.7	0.0 \pm 0.0	11.6 \pm 1.4	26.5 \pm 2.1	0.0 \pm 0.0	24.2 \pm 3.2
	BUCKET	5	5	0.9 \pm 0.0	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	19.4 \pm 1.1	0.0 \pm 0.0	19.3 \pm 5.7
	COMPASS	5	5	135.4 \pm 73.3	18.6 \pm 1.6	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.2	11.8 \pm 0.6	0.0 \pm 0.0	12.8 \pm 1.4	28.7 \pm 1.9	0.0 \pm 0.0	20.3 \pm 2.8
	LEATHER	5	5	0.9 \pm 0.0	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.7 \pm 1.7	0.0 \pm 0.0	17.9 \pm 4.4
	PAPER	5	5	0.8 \pm 0.1	3.4 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.6 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.4 \pm 0.2	19.8 \pm 2.0	0.0 \pm 0.0	40.6 \pm 27.0
	QUILL	5	5	18.0 \pm 3.5	19.8 \pm 1.2	5.0 \pm 0.0	5.2 \pm 0.2	2.1 \pm 0.1	13.2 \pm 0.4	0.0 \pm 0.0	12.6 \pm 1.1	29.6 \pm 2.5	0.0 \pm 0.0	24.4 \pm 3.2
	SUGAR	5	5	0.8 \pm 0.1	3.2 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	1.2 \pm 0.2	17.7 \pm 1.6	0.0 \pm 0.0	17.5 \pm 3.2
	BOOK	5	5	191.2 \pm 36.4	22.8 \pm 2.6	5.0 \pm 0.0	5.8 \pm 0.2	6.0 \pm 0.2	11.4 \pm 0.7	0.0 \pm 0.0	17.4 \pm 2.2	20.5 \pm 1.8	0.0 \pm 0.0	24.8 \pm 1.5
	MAP	5	5	549.4 \pm 149.5	33.4 \pm 3.2	5.0 \pm 0.0	5.6 \pm 0.2	6.0 \pm 0.2	12.2 \pm 0.6	0.0 \pm 0.0	27.2 \pm 2.9	29.5 \pm 3.2	0.0 \pm 0.0	28.7 \pm 1.7
	MILKBUCKET	5	5	1.5 \pm 0.2	4.6 \pm 0.4	3.0 \pm 0.0	2.0 \pm 0.0	6.8 \pm 0.5	10.0 \pm 0.0	0.0 \pm 0.0	2.6 \pm 0.4	11.6 \pm 0.7	0.0 \pm 0.0	15.3 \pm 4.3
	BOOKQUILL	5	5	17.9 \pm 1.4	19.6 \pm 1.1	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	16.6 \pm 1.1	27.2 \pm 1.3	0.0 \pm 0.0	20.8 \pm 1.4
	SWEETMILK	5	5	7.3 \pm 1.2	12.4 \pm 1.2	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.1	10.2 \pm 0.2	0.0 \pm 0.0	9.2 \pm 1.2	16.9 \pm 0.8	0.0 \pm 0.0	14.3 \pm 1.7
	CAKE	5	5	74.5 \pm 25.7	26.4 \pm 3.7	4.0 \pm 0.0	3.2 \pm 0.2	2.1 \pm 0.1	10.2 \pm 0.2	0.0 \pm 0.0	23.2 \pm 3.6	38.4 \pm 0.9	0.0 \pm 0.0	22.7 \pm 1.6
	Completed Runs = 5				Total Time (s.) = 1009.8 \pm 122.3				Total Calls = 189.4 \pm 4.1					
ODL	BATTER	5	5	13.7 \pm 2.9	23.0 \pm 3.0	6.0 \pm 0.0	9.2 \pm 0.2	12.0 \pm 1.0	11.4 \pm 0.4	7.0 \pm 1.2	10.6 \pm 1.6	20.4 \pm 1.1	18.7 \pm 1.6	12.1 \pm 1.7
	BUCKET	5	5	1.8 \pm 0.2	7.2 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	8.0 \pm 0.5	10.2 \pm 0.2	2.2 \pm 0.2	2.8 \pm 0.4	10.2 \pm 0.5	13.4 \pm 1.9	6.8 \pm 1.7
	COMPASS	5	5	13.1 \pm 1.7	22.0 \pm 1.7	6.0 \pm 0.0	9.2 \pm 0.2	10.4 \pm 1.4	11.0 \pm 0.6	6.8 \pm 1.0	10.2 \pm 1.0	17.2 \pm 1.6	20.9 \pm 1.9	14.3 \pm 0.8
	LEATHER	5	5	1.9 \pm 0.2	7.0 \pm 0.5	4.0 \pm 0.0	4.0 \pm 0.0	6.9 \pm 0.5	10.0 \pm 0.0	2.4 \pm 0.2	2.6 \pm 0.4	11.1 \pm 0.9	16.9 \pm 5.6	8.9 \pm 3.3
	PAPER	5	5	2.0 \pm 0.2	7.6 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	7.7 \pm 1.1	10.0 \pm 0.0	3.0 \pm 0.3	2.6 \pm 0.4	10.1 \pm 0.9	18.9 \pm 3.3	5.6 \pm 0.8
	QUILL	5	5	11.3 \pm 1.2	22.0 \pm 1.2	6.0 \pm 0.0	9.2 \pm 0.2	12.8 \pm 1.5	10.6 \pm 0.2	6.4 \pm 0.7	11.0 \pm 0.9	15.3 \pm 1.3	13.5 \pm 1.0	12.1 \pm 1.4
	SUGAR	5	5	1.7 \pm 0.1	6.4 \pm 0.4	4.0 \pm 0.0	4.0 \pm 0.0	6.5 \pm 0.7	10.0 \pm 0.0	2.4 \pm 0.2	2.0 \pm 0.3	9.6 \pm 0.6	15.3 \pm 3.6	16.6 \pm 9.2
	BOOK	5	5	427.8 \pm 201.6	32.6 \pm 4.2	6.0 \pm 0.0	6.6 \pm 0.2	5.6 \pm 0.2	12.0 \pm 0.3	3.6 \pm 0.7	23.0 \pm 3.4	21.6 \pm 1.5	25.9 \pm 3.4	23.7 \pm 1.3
	MAP	5	5	647.9 \pm 110.7	38.6 \pm 3.6	6.0 \pm 0.0	6.4 \pm 0.2	5.6 \pm 0.2	11.2 \pm 0.4	3.8 \pm 0.9	29.6 \pm 3.5	23.1 \pm 1.0	27.8 \pm 4.6	26.1 \pm 0.4
	MILKBUCKET	5	5	2.1 \pm 0.2	5.4 \pm 0.4	4.0 \pm 0.0	3.0 \pm 0.0	7.6 \pm 0.5	10.0 \pm 0.0	1.4 \pm 0.4	2.0 \pm 0.0	11.1 \pm 0.5	26.3 \pm 6.5	15.2 \pm 5.8
	BOOKQUILL	5	5	18.7 \pm 2.3	16.6 \pm 1.3	4.0 \pm 0.0	4.0 \pm 0.0	3.7 \pm 0.2	10.0 \pm 0.0	0.4 \pm 0.2	13.2 \pm 1.4	29.0 \pm 1.1	6.2 \pm 5.5	27.8 \pm 1.4
	SWEETMILK	5	5	7.7 \pm 0.7	12.2 \pm 0.9	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.2	10.0 \pm 0.0	0.2 \pm 0.2	9.0 \pm 0.9	16.0 \pm 0.9	1.6 \pm 1.6	16.3 \pm 1.3
	CAKE	5	5	472.9 \pm 216.6	36.0 \pm 6.0	5.0 \pm 0.0	4.6 \pm 0.2	2.1 \pm 0.0	10.0 \pm 0.0	1.6 \pm 0.4	31.4 \pm 5.7	39.5 \pm 1.2	41.5 \pm 8.6	26.9 \pm 0.8
	Completed Runs = 5				Total Time (s.) = 1622.6 \pm 328.7				Total Calls = 236.6 \pm 9.3					
FRL	BATTER	5	5	12.3 \pm 1.7	17.6 \pm 1.3	5.0 \pm 0.0	5.4 \pm 0.2	9.2 \pm 1.2	11.6 \pm 0.4	0.0 \pm 0.0	12.0 \pm 1.2	30.3 \pm 2.3	0.0 \pm 0.0	27.8 \pm 2.0
	BUCKET	5	5	1.2 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	6.7 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.6 \pm 2.5	0.0 \pm 0.0	28.5 \pm 4.1
	COMPASS	5	5	14.1 \pm 1.6	20.2 \pm 1.7	5.0 \pm 0.0	5.2 \pm 0.2	9.8 \pm 0.7	11.6 \pm 0.6	0.0 \pm 0.0	14.6 \pm 1.2	26.5 \pm 0.8	0.0 \pm 0.0	26.5 \pm 2.1
	LEATHER	5	5	1.1 \pm 0.1	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	4.5 \pm 0.7	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	13.4 \pm 1.3	0.0 \pm 0.0	16.7 \pm 3.6
	PAPER	5	5	1.2 \pm 0.0	4.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.0	4.9 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	2.0 \pm 0.0	12.4 \pm 1.1	0.0 \pm 0.0	10.9 \pm 2.5
	QUILL	5	5	8.9 \pm 0.9	16.0 \pm 0.8	5.0 \pm 0.0	5.2 \pm 0.2	9.4 \pm 1.7	10.6 \pm 0.2	0.0 \pm 0.0	11.4 \pm 0.6	25.4 \pm 0.3	0.0 \pm 0.0	25.5 \pm 2.7
	SUGAR	5	5	1.1 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	5.2 \pm 0.3	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	15.3 \pm 1.7	0.0 \pm 0.0	21.0 \pm 10.1
	BOOK	5	5	220.2 \pm 83.3	25.2 \pm 3.4	5.0 \pm 0.0	5.6 \pm 0.2	6.1 \pm 0.2	10.2 \pm 0.2	0.0 \pm 0.0	21.0 \pm 3.4	21.9 \pm 1.0	0.0 \pm 0.0	18.4 \pm 0.7
	MAP	5	5	628.3 \pm 85.4	37.8 \pm 3.7	5.0 \pm 0.0	5.6 \pm 0.2	5.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	33.8 \pm 3.7	26.4 \pm 1.0	0.0 \pm 0.0	21.4 \pm 0.7
	MILKBUCKET	5	5	1.9 \pm 0.2	5.0 \pm 0.3	3.0 \pm 0.0	2.0 \pm 0.0	9.8 \pm 0.7	10.0 \pm 0.0	0.0 \pm 0.0	3.0 \pm 0.3	13.2 \pm 0.7	0.0 \pm 0.0	12.8 \pm 3.2
	BOOKQUILL	5	5	12.9 \pm 2.2	15.6 \pm 1.7	4.0 \pm 0.0	4.0 \pm 0.0	3.9 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	12.6 \pm 1.7	29.0 \pm 1.5	0.0 \pm 0.0	13.3 \pm 0.8
	SWEETMILK	5	5	7.2 \pm 0.6	12.0 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	3.9 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	9.0 \pm 0.7	18.9 \pm 0.9	0.0 \pm 0.0	10.1 \pm 1.0
	CAKE	5	5	121.1 \pm 41.1	34.0 \pm 4.8	4.0 \pm 0.0	3.0 \pm 0.0	2.2 \pm 0.0	10.0 \pm 0.0	0.0 \pm 0.0	31.0 \pm 4.8	42.2 \pm 1.7	0.0 \pm 0.0	16.2 \pm 1.1
	Completed Runs = 5				Total Time (s.) = 1031.6 \pm 150.3				Total Calls = 198.6 \pm 11.3					
FRL	BATTER	5	5	11.3 \pm 1.4	23.4 \pm 2.5	6.0 \pm 0.0	9.2 \pm 0.2	468.4 \pm 121.9	10.4 \pm 0.2	7.6 \pm 0.9	11.4 \pm 1.9	11.9 \pm 0.6	10.1 \pm 1.3	9.9 \pm 0.4
	BUCKET	5	5	2.3 \pm 0.2	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	129.5 \pm 69.4	10.2 \pm 0.2	2.8 \pm 0.2	2.0 \pm 0.3	7.8 \pm 0.5	9.9 \pm 1.7	6.4 \pm 2.1
	COMPASS	5	5	13.0 \pm 1.9	24.6 \pm 2.2	6.0 \pm 0.0	9.4 \pm 0.2	550.8 \pm 156.4	10.4 \pm 0.2	7.8 \pm 1.0	12.4 \pm 1.2	12.5 \pm 1.6	9.4 \pm 1.0	8.4 \pm 0.5
	LEATHER	5	5	2.5 \pm 0.3	7.8 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	89.0 \pm 18.0	10.0 \pm 0.0	3.2 \pm 0.4	2.6 \pm 0.4	7.3 \pm 0.4	9.3 \pm 1.7	3.7 \pm 0.4
	PAPER	5	5	2.2 \pm 0.1	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	82.7 \pm 18.8	10.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.3	6.9 \pm 0.7	10.2 \pm 1.8	4.7 \pm 2.7
	QUILL	5	5	11.6 \pm 1.1	23.8 \pm 1.5	6.0 \pm 0.0	9.6 \pm 0.2	458.9 \pm 61.0	10.6 \pm 0.2	8.0 \pm 0.9	11.2 \pm 1.2	11.9 \pm 0.6	13.1 \pm 2.7	9.2 \pm 0.8
	SUGAR	5	5	2.7 \pm 0.2	8.4 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	103.5 \pm 39.5	10.0 \pm 0.0	3.6 \pm 0.4	2.8 \pm 0.5	8.2 \pm 0.7	10.1 \pm 1.9	5.0 \pm 1.1
	BOOK	5	5	301.7 \pm 98.1	36.4 \pm 1.9	6.0 \pm 0.0	6.8 \pm 0.2	5.3 \pm 0.1	10.2 \pm 0.2	5.0 \pm 0.7	27.2 \pm 1.9	21.7 \pm 1.1	18.8 \pm 2.2	16.1 \pm 0.6
	MAP	5	5	754.1 \pm 158.2	44.6 \pm 2.6	6.0 \pm 0.0	7.0 \pm 0.0	5.5 \pm 0.2	10.2 \pm 0.2	5.2 \pm 0.4	35.2 \pm 2.3	25.6 \pm 0.5	20.4 \pm 2.9	18.7 \pm 0.6
	MILKBUCKET</													

Table B.3: Results of LHRM in CRAFTWORLD with a restricted set of callable RMs.

	Task	G	L	Time (s.)	Calls	States	Edges	Ep. First ($\times 10^2$)	# Examples			Example Length		
									G	D	I	G	D	I
OP	BATTER	5	5	11.2 \pm 1.6	17.8 \pm 1.9	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.1	12.2 \pm 0.7	0.0 \pm 0.0	11.6 \pm 1.4	26.5 \pm 2.1	0.0 \pm 0.0	24.2 \pm 3.2
	BUCKET	5	5	0.9 \pm 0.0	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	19.4 \pm 1.1	0.0 \pm 0.0	19.3 \pm 5.7
	COMPASS	5	5	15.5 \pm 4.2	18.6 \pm 1.6	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.2	11.8 \pm 0.6	0.0 \pm 0.0	12.8 \pm 1.4	28.7 \pm 1.9	0.0 \pm 0.0	20.3 \pm 2.8
	LEATHER	5	5	0.9 \pm 0.0	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.7 \pm 1.7	0.0 \pm 0.0	17.9 \pm 4.4
	PAPER	5	5	0.9 \pm 0.0	3.4 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.6 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.4 \pm 0.2	19.8 \pm 2.0	0.0 \pm 0.0	40.6 \pm 27.0
	QUILL	5	5	18.2 \pm 3.5	19.8 \pm 1.2	5.0 \pm 0.0	5.2 \pm 0.2	2.1 \pm 0.1	13.2 \pm 0.4	0.0 \pm 0.0	12.6 \pm 1.1	29.6 \pm 2.5	0.0 \pm 0.0	24.4 \pm 3.2
	SUGAR	5	5	0.8 \pm 0.0	3.2 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	1.2 \pm 0.2	17.7 \pm 1.6	0.0 \pm 0.0	17.5 \pm 3.2
	BOOK	5	5	45.8 \pm 4.5	19.6 \pm 0.9	5.0 \pm 0.0	5.6 \pm 0.2	6.0 \pm 0.2	11.2 \pm 1.0	0.0 \pm 0.0	14.4 \pm 0.9	21.6 \pm 1.8	0.0 \pm 0.0	21.0 \pm 1.7
	MAP	5	5	64.1 \pm 10.6	22.0 \pm 2.6	5.0 \pm 0.0	5.2 \pm 0.2	6.1 \pm 0.2	10.8 \pm 0.4	0.0 \pm 0.0	17.2 \pm 2.7	22.5 \pm 1.6	0.0 \pm 0.0	23.0 \pm 1.2
	MILKBUCKET	5	5	1.2 \pm 0.1	4.4 \pm 0.4	3.0 \pm 0.0	2.0 \pm 0.0	6.8 \pm 0.3	10.0 \pm 0.0	0.0 \pm 0.0	2.4 \pm 0.4	12.1 \pm 0.7	0.0 \pm 0.0	15.3 \pm 1.6
	BOOKQUILL	5	5	4.5 \pm 0.8	10.2 \pm 1.4	4.0 \pm 0.0	4.0 \pm 0.0	3.9 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	7.2 \pm 1.4	26.1 \pm 0.8	0.0 \pm 0.0	22.4 \pm 0.9
	SWEETMILK	5	5	3.5 \pm 0.5	9.6 \pm 1.3	4.0 \pm 0.0	4.0 \pm 0.0	3.9 \pm 0.1	10.2 \pm 0.2	0.0 \pm 0.0	6.4 \pm 1.2	17.4 \pm 0.5	0.0 \pm 0.0	12.5 \pm 0.8
	CAKE	5	5	9.1 \pm 0.9	17.0 \pm 0.9	4.0 \pm 0.0	3.2 \pm 0.2	2.1 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	14.0 \pm 0.9	37.5 \pm 1.9	0.0 \pm 0.0	18.0 \pm 1.9
	Completed Runs = 5				Total Time (s.) = 176.6 \pm 13.1					Total Calls = 153.0 \pm 3.6				
OPL	BATTER	5	5	13.9 \pm 3.0	23.0 \pm 3.0	6.0 \pm 0.0	9.2 \pm 0.2	12.0 \pm 1.0	11.4 \pm 0.4	7.0 \pm 1.2	10.6 \pm 1.6	20.4 \pm 1.1	18.7 \pm 1.6	12.1 \pm 1.7
	BUCKET	5	5	1.8 \pm 0.1	7.2 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	8.0 \pm 0.5	10.2 \pm 0.2	2.2 \pm 0.2	2.8 \pm 0.4	10.2 \pm 0.5	13.4 \pm 1.9	6.8 \pm 1.7
	COMPASS	5	5	13.2 \pm 1.7	22.0 \pm 1.7	6.0 \pm 0.0	9.2 \pm 0.2	10.4 \pm 1.4	11.0 \pm 0.6	6.8 \pm 1.0	10.2 \pm 1.0	17.2 \pm 1.6	20.9 \pm 1.9	14.3 \pm 0.8
	LEATHER	5	5	1.9 \pm 0.1	7.0 \pm 0.5	4.0 \pm 0.0	4.0 \pm 0.0	6.9 \pm 0.5	10.0 \pm 0.0	2.4 \pm 0.2	2.6 \pm 0.4	11.1 \pm 0.9	16.9 \pm 5.6	8.9 \pm 3.3
	PAPER	5	5	2.0 \pm 0.2	7.6 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	7.7 \pm 1.1	10.0 \pm 0.0	3.0 \pm 0.3	2.6 \pm 0.4	10.1 \pm 0.9	18.9 \pm 3.3	5.6 \pm 0.8
	QUILL	5	5	11.5 \pm 1.3	22.0 \pm 1.2	6.0 \pm 0.0	9.2 \pm 0.2	12.8 \pm 1.5	10.6 \pm 0.2	6.4 \pm 0.7	11.0 \pm 0.9	15.3 \pm 1.3	13.5 \pm 1.0	12.1 \pm 1.4
	SUGAR	5	5	1.6 \pm 0.1	6.4 \pm 0.4	4.0 \pm 0.0	4.0 \pm 0.0	6.5 \pm 0.7	10.0 \pm 0.0	2.4 \pm 0.2	2.0 \pm 0.3	9.6 \pm 0.6	15.3 \pm 3.6	16.6 \pm 9.2
	BOOK	5	5	69.0 \pm 20.5	21.8 \pm 2.2	6.0 \pm 0.0	6.2 \pm 0.2	5.5 \pm 0.1	10.4 \pm 0.2	5.2 \pm 0.9	12.2 \pm 1.9	20.4 \pm 1.3	21.2 \pm 2.0	20.8 \pm 1.7
	MAP	5	5	76.5 \pm 6.0	24.2 \pm 1.3	6.0 \pm 0.0	6.4 \pm 0.2	5.7 \pm 0.3	11.6 \pm 0.8	4.0 \pm 0.3	14.6 \pm 1.0	24.8 \pm 3.0	21.4 \pm 2.1	25.7 \pm 0.8
	MILKBUCKET	5	5	1.7 \pm 0.2	6.0 \pm 0.6	4.0 \pm 0.0	3.0 \pm 0.0	7.5 \pm 0.7	10.2 \pm 0.2	1.4 \pm 0.2	2.4 \pm 0.2	11.7 \pm 0.7	25.4 \pm 6.4	14.2 \pm 3.4
	BOOKQUILL	5	5	5.3 \pm 0.9	10.8 \pm 1.4	4.0 \pm 0.0	4.0 \pm 0.0	3.7 \pm 0.1	10.0 \pm 0.0	1.0 \pm 0.5	6.8 \pm 0.9	27.7 \pm 1.0	11.2 \pm 5.4	21.1 \pm 1.7
	SWEETMILK	5	5	4.0 \pm 0.9	9.8 \pm 1.7	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.1	10.0 \pm 0.0	1.6 \pm 0.7	5.2 \pm 1.2	18.4 \pm 0.7	8.3 \pm 2.9	15.6 \pm 1.7
	CAKE	5	5	16.2 \pm 0.4	20.8 \pm 0.2	5.0 \pm 0.0	4.0 \pm 0.0	2.1 \pm 0.1	10.0 \pm 0.0	3.2 \pm 0.2	14.6 \pm 0.2	38.1 \pm 0.9	22.5 \pm 3.3	25.8 \pm 1.7
	Completed Runs = 5				Total Time (s.) = 218.6 \pm 21.1					Total Calls = 188.6 \pm 5.4				
FR	BATTER	5	5	12.6 \pm 1.8	17.6 \pm 1.3	5.0 \pm 0.0	5.4 \pm 0.2	9.2 \pm 1.2	11.6 \pm 0.4	0.0 \pm 0.0	12.0 \pm 1.2	30.3 \pm 2.3	0.0 \pm 0.0	27.8 \pm 2.0
	BUCKET	5	5	1.2 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	6.7 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.6 \pm 2.5	0.0 \pm 0.0	28.5 \pm 4.1
	COMPASS	5	5	14.1 \pm 1.5	20.2 \pm 1.7	5.0 \pm 0.0	5.2 \pm 0.2	9.8 \pm 0.7	11.6 \pm 0.6	0.0 \pm 0.0	14.6 \pm 1.2	26.5 \pm 0.8	0.0 \pm 0.0	26.5 \pm 2.1
	LEATHER	5	5	1.1 \pm 0.1	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	4.5 \pm 0.7	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	13.4 \pm 1.3	0.0 \pm 0.0	16.7 \pm 3.6
	PAPER	5	5	1.2 \pm 0.1	4.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.0	4.9 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	2.0 \pm 0.0	12.4 \pm 1.1	0.0 \pm 0.0	10.9 \pm 2.5
	QUILL	5	5	9.3 \pm 0.8	16.0 \pm 0.8	5.0 \pm 0.0	5.2 \pm 0.2	9.4 \pm 1.7	10.6 \pm 0.2	0.0 \pm 0.0	11.4 \pm 0.6	25.4 \pm 0.3	0.0 \pm 0.0	25.5 \pm 2.7
	SUGAR	5	5	1.4 \pm 0.2	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	5.2 \pm 0.3	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	15.3 \pm 1.7	0.0 \pm 0.0	21.0 \pm 10.1
	BOOK	5	5	43.8 \pm 13.0	20.0 \pm 1.9	5.0 \pm 0.0	5.4 \pm 0.2	6.0 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	16.0 \pm 1.9	21.9 \pm 1.0	0.0 \pm 0.0	14.7 \pm 1.4
	MAP	5	5	85.2 \pm 13.4	22.2 \pm 2.5	5.0 \pm 0.0	5.2 \pm 0.2	5.9 \pm 0.1	10.2 \pm 0.2	0.0 \pm 0.0	18.0 \pm 2.6	26.5 \pm 0.9	0.0 \pm 0.0	18.2 \pm 1.2
	MILKBUCKET	5	5	1.4 \pm 0.1	4.4 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	10.2 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	2.4 \pm 0.2	13.0 \pm 0.8	0.0 \pm 0.0	12.2 \pm 2.8
	BOOKQUILL	5	5	6.3 \pm 0.9	13.2 \pm 1.7	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	10.2 \pm 1.7	30.6 \pm 2.0	0.0 \pm 0.0	11.9 \pm 1.2
	SWEETMILK	5	5	4.8 \pm 0.6	11.8 \pm 1.3	4.0 \pm 0.0	4.0 \pm 0.0	3.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	8.8 \pm 1.3	19.8 \pm 0.7	0.0 \pm 0.0	8.6 \pm 1.0
	CAKE	5	5	12.5 \pm 1.8	20.8 \pm 2.5	4.0 \pm 0.0	3.0 \pm 0.0	2.3 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	17.8 \pm 2.5	44.2 \pm 2.6	0.0 \pm 0.0	13.1 \pm 1.2
	Completed Runs = 5				Total Time (s.) = 194.9 \pm 17.6					Total Calls = 161.4 \pm 7.0				
FRL	BATTER	5	5	11.2 \pm 1.4	23.4 \pm 2.5	6.0 \pm 0.0	9.2 \pm 0.2	468.4 \pm 121.9	10.4 \pm 0.2	7.6 \pm 0.9	11.4 \pm 1.9	11.9 \pm 0.6	10.1 \pm 1.3	9.9 \pm 0.4
	BUCKET	5	5	2.4 \pm 0.1	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	129.5 \pm 69.4	10.2 \pm 0.2	2.8 \pm 0.2	2.0 \pm 0.3	7.8 \pm 0.5	9.9 \pm 1.7	6.4 \pm 2.1
	COMPASS	5	5	13.1 \pm 1.9	24.6 \pm 2.2	6.0 \pm 0.0	9.4 \pm 0.2	550.8 \pm 156.4	10.4 \pm 0.2	7.8 \pm 1.0	12.4 \pm 1.2	12.5 \pm 1.6	9.4 \pm 1.0	8.4 \pm 0.5
	LEATHER	5	5	2.5 \pm 0.4	7.8 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	89.0 \pm 18.0	10.0 \pm 0.0	3.2 \pm 0.4	2.6 \pm 0.4	7.3 \pm 0.4	9.3 \pm 1.7	3.7 \pm 0.4
	PAPER	5	5	2.1 \pm 0.1	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	82.7 \pm 18.8	10.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.3	6.9 \pm 0.7	10.2 \pm 1.8	4.7 \pm 2.7
	QUILL	5	5	11.6 \pm 1.2	23.8 \pm 1.5	6.0 \pm 0.0	9.6 \pm 0.2	458.9 \pm 61.0	10.6 \pm 0.2	8.0 \pm 0.9	11.2 \pm 1.2	11.9 \pm 0.6	13.1 \pm 2.7	9.2 \pm 0.8
	SUGAR	5	5	2.6 \pm 0.2	8.4 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	103.5 \pm 39.5	10.0 \pm 0.0	3.6 \pm 0.4	2.8 \pm 0.5	8.2 \pm 0.7	10.1 \pm 1.9	5.0 \pm 1.1
	BOOK	5	5	62.2 \pm 13.2	27.4 \pm 2.2	6.0 \pm 0.0	6.6 \pm 0.2	5.3 \pm 0.1	10.2 \pm 0.2	5.6 \pm 0.6	17.6 \pm 1.7	23.0 \pm 1.0	16.5 \pm 2.0	13.4 \pm 1.0
	MAP	5	5	131.3 \pm 28.0	34.0 \pm 3.0	6.0 \pm 0.0	6.6 \pm 0.2	5.5 \pm 0.2	10.2 \pm 0.2	6.8 \pm 0.7	23.0 \pm 2.4	26.2 \pm 0.7	16.9 \pm 1.6	14.5 \pm 0.5
	MILKBUCKET	5	5	2.7 \pm 0.7	6.6 \pm 0.6	4.0 \pm 0.0	3.0 \pm 0.0	6.8 \pm 0.3	10.0 \pm 0.0	2.2 \pm 0.2	2.4 \pm 0.4	12.0 $\$		

Table B.5: Results of LHRM in CRAFTWORLD without exploration using options.

	Task	G	L	Time (s.)	Calls	States	Edges	Ep. First ($\times 10^2$)	# Examples			Example Length		
									G	D	I	G	D	I
OP	BATTER	5	5	11.2 \pm 1.7	17.8 \pm 1.9	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.1	12.2 \pm 0.7	0.0 \pm 0.0	11.6 \pm 1.4	26.5 \pm 2.1	0.0 \pm 0.0	24.2 \pm 3.2
	BUCKET	5	5	0.9 \pm 0.0	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	19.4 \pm 1.1	0.0 \pm 0.0	19.3 \pm 5.7
	COMPASS	5	5	15.6 \pm 4.1	18.6 \pm 1.6	5.0 \pm 0.0	5.2 \pm 0.2	1.8 \pm 0.2	11.8 \pm 0.6	0.0 \pm 0.0	12.8 \pm 1.4	28.7 \pm 1.9	0.0 \pm 0.0	20.3 \pm 2.8
	LEATHER	5	5	0.9 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.8 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.7 \pm 1.7	0.0 \pm 0.0	17.9 \pm 4.4
	PAPER	5	5	0.9 \pm 0.1	3.4 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.6 \pm 0.1	10.0 \pm 0.0	0.0 \pm 0.0	1.4 \pm 0.2	19.8 \pm 2.0	0.0 \pm 0.0	40.6 \pm 27.0
	QUILL	5	5	18.3 \pm 3.6	19.8 \pm 1.2	5.0 \pm 0.0	5.2 \pm 0.2	2.1 \pm 0.1	13.2 \pm 0.4	0.0 \pm 0.0	12.6 \pm 1.1	29.6 \pm 2.5	0.0 \pm 0.0	24.4 \pm 3.2
	SUGAR	5	5	0.9 \pm 0.0	3.2 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	1.7 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	1.2 \pm 0.2	17.7 \pm 1.6	0.0 \pm 0.0	17.5 \pm 3.2
	BOOK	5	5	529.0 \pm 164.2	21.2 \pm 1.4	5.0 \pm 0.0	5.8 \pm 0.2	6.8 \pm 0.2	10.2 \pm 0.2	0.0 \pm 0.0	17.0 \pm 1.5	33.0 \pm 2.6	0.0 \pm 0.0	23.7 \pm 1.3
	MAP	5	5	1924.2 \pm 443.5	28.0 \pm 3.8	5.0 \pm 0.0	5.4 \pm 0.2	7.8 \pm 0.4	10.4 \pm 0.2	0.0 \pm 0.0	23.6 \pm 3.7	40.1 \pm 1.0	0.0 \pm 0.0	29.4 \pm 1.3
	MILKBUCKET	5	5	1.6 \pm 0.2	4.4 \pm 0.4	3.0 \pm 0.0	2.0 \pm 0.0	6.1 \pm 0.3	10.0 \pm 0.0	0.0 \pm 0.0	2.4 \pm 0.4	16.0 \pm 1.0	0.0 \pm 0.0	14.2 \pm 1.3
	BOOKQUILL	5	5	42.7 \pm 10.1	24.6 \pm 3.9	4.0 \pm 0.0	4.0 \pm 0.0	6.8 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	21.6 \pm 3.9	55.8 \pm 2.7	0.0 \pm 0.0	21.2 \pm 1.1
	SWEETMILK	5	5	8.1 \pm 0.8	11.8 \pm 1.0	4.0 \pm 0.0	4.0 \pm 0.0	4.9 \pm 0.1	10.2 \pm 0.2	0.0 \pm 0.0	8.6 \pm 1.2	31.1 \pm 0.7	0.0 \pm 0.0	13.1 \pm 0.8
	CAKE	5	5	198.3 \pm 47.5	43.0 \pm 5.3	4.0 \pm 0.0	3.8 \pm 0.2	5.5 \pm 0.2	10.0 \pm 0.0	0.0 \pm 0.0	40.0 \pm 5.3	65.0 \pm 0.9	0.0 \pm 0.0	22.0 \pm 0.9
	Completed Runs = 5				Total Time (s.) = 2752.8 \pm 503.2					Total Calls = 203.2 \pm 11.8				
OPL	BATTER	5	5	14.1 \pm 3.2	23.0 \pm 3.0	6.0 \pm 0.0	9.2 \pm 0.2	12.0 \pm 1.0	11.4 \pm 0.4	7.0 \pm 1.2	10.6 \pm 1.6	20.4 \pm 1.1	18.7 \pm 1.6	12.1 \pm 1.7
	BUCKET	5	5	1.8 \pm 0.1	7.2 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	8.0 \pm 0.5	10.2 \pm 0.2	2.2 \pm 0.2	2.8 \pm 0.4	10.2 \pm 0.5	13.4 \pm 1.9	6.8 \pm 1.7
	COMPASS	5	5	13.5 \pm 1.8	22.0 \pm 1.7	6.0 \pm 0.0	9.2 \pm 0.2	10.4 \pm 1.4	11.0 \pm 0.6	6.8 \pm 1.0	10.2 \pm 1.0	17.2 \pm 1.6	20.9 \pm 1.9	14.3 \pm 0.8
	LEATHER	5	5	1.8 \pm 0.1	7.0 \pm 0.5	4.0 \pm 0.0	4.0 \pm 0.0	6.9 \pm 0.5	10.0 \pm 0.0	2.4 \pm 0.2	2.6 \pm 0.4	11.1 \pm 0.9	16.9 \pm 5.6	8.9 \pm 3.3
	PAPER	5	5	2.0 \pm 0.2	7.6 \pm 0.6	4.0 \pm 0.0	4.0 \pm 0.0	7.7 \pm 1.1	10.0 \pm 0.0	3.0 \pm 0.3	2.6 \pm 0.4	10.1 \pm 0.9	18.9 \pm 3.3	5.6 \pm 0.8
	QUILL	5	5	11.8 \pm 1.3	22.0 \pm 1.2	6.0 \pm 0.0	9.2 \pm 0.2	12.8 \pm 1.5	10.6 \pm 0.2	6.4 \pm 0.7	11.0 \pm 0.9	15.3 \pm 1.3	13.5 \pm 1.0	12.1 \pm 1.4
	SUGAR	5	5	1.6 \pm 0.1	6.4 \pm 0.4	4.0 \pm 0.0	4.0 \pm 0.0	6.5 \pm 0.7	10.0 \pm 0.0	2.4 \pm 0.2	2.0 \pm 0.3	9.6 \pm 0.6	15.3 \pm 3.6	16.6 \pm 9.2
	BOOK	5	5	224.8 \pm 71.6	27.0 \pm 1.9	6.0 \pm 0.0	6.4 \pm 0.2	139.7 \pm 21.8	11.6 \pm 0.4	3.2 \pm 0.4	18.2 \pm 1.4	22.0 \pm 1.6	24.7 \pm 6.5	23.5 \pm 1.2
	MAP	5	5	339.9 \pm 33.6	33.0 \pm 2.8	6.0 \pm 0.0	6.4 \pm 0.2	204.8 \pm 27.1	10.6 \pm 0.2	2.8 \pm 0.5	25.6 \pm 2.5	25.4 \pm 0.8	21.8 \pm 3.1	25.2 \pm 1.1
	MILKBUCKET	5	5	3.5 \pm 0.3	8.2 \pm 0.6	4.0 \pm 0.0	3.0 \pm 0.0	47.6 \pm 3.7	10.2 \pm 0.2	2.6 \pm 0.4	3.4 \pm 0.4	10.3 \pm 0.7	16.2 \pm 1.7	14.2 \pm 1.8
	BOOKQUILL	5	5	19.0 \pm 2.2	15.4 \pm 1.5	4.0 \pm 0.0	4.0 \pm 0.0	383.4 \pm 83.7	10.0 \pm 0.0	1.0 \pm 0.3	11.4 \pm 1.3	38.2 \pm 1.6	14.1 \pm 4.9	23.8 \pm 1.0
	SWEETMILK	5	5	11.4 \pm 2.1	14.4 \pm 1.7	4.0 \pm 0.0	4.0 \pm 0.0	87.4 \pm 8.9	10.4 \pm 0.2	1.0 \pm 0.4	10.0 \pm 1.3	19.7 \pm 1.2	8.7 \pm 4.9	17.6 \pm 1.2
	CAKE	4	1	277.4 \pm 0.0	33.0 \pm 0.0	5.0 \pm 0.0	4.0 \pm 0.0	264.1 \pm 0.0	10.0 \pm 0.0	2.0 \pm 0.0	28.0 \pm 0.0	46.7 \pm 0.0	36.0 \pm 0.0	22.9 \pm 0.0
	Completed Runs = 5				Total Time (s.) = 701.0 \pm 111.2					Total Calls = 199.8 \pm 6.9				
FR	BATTER	5	5	12.3 \pm 1.6	17.6 \pm 1.3	5.0 \pm 0.0	5.4 \pm 0.2	9.2 \pm 1.2	11.6 \pm 0.4	0.0 \pm 0.0	12.0 \pm 1.2	30.3 \pm 2.3	0.0 \pm 0.0	27.8 \pm 2.0
	BUCKET	5	5	1.2 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	6.7 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	16.6 \pm 2.5	0.0 \pm 0.0	28.5 \pm 4.1
	COMPASS	5	5	14.2 \pm 1.7	20.2 \pm 1.7	5.0 \pm 0.0	5.2 \pm 0.2	9.8 \pm 0.7	11.6 \pm 0.6	0.0 \pm 0.0	14.6 \pm 1.2	26.5 \pm 0.8	0.0 \pm 0.0	26.5 \pm 2.1
	LEATHER	5	5	1.1 \pm 0.1	3.6 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	4.5 \pm 0.7	10.0 \pm 0.0	0.0 \pm 0.0	1.6 \pm 0.2	13.4 \pm 1.3	0.0 \pm 0.0	16.7 \pm 3.6
	PAPER	5	5	1.2 \pm 0.0	4.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.0	4.9 \pm 0.9	10.0 \pm 0.0	0.0 \pm 0.0	2.0 \pm 0.0	12.4 \pm 1.1	0.0 \pm 0.0	10.9 \pm 2.5
	QUILL	5	5	9.0 \pm 0.8	16.0 \pm 0.8	5.0 \pm 0.0	5.2 \pm 0.2	9.4 \pm 1.7	10.6 \pm 0.2	0.0 \pm 0.0	11.4 \pm 0.6	25.4 \pm 0.3	0.0 \pm 0.0	25.5 \pm 2.7
	SUGAR	5	5	1.1 \pm 0.1	3.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	5.2 \pm 0.3	10.0 \pm 0.0	0.0 \pm 0.0	1.8 \pm 0.2	15.3 \pm 1.7	0.0 \pm 0.0	21.0 \pm 10.1
	BOOK	5	5	157.9 \pm 36.0	31.4 \pm 2.8	5.0 \pm 0.0	5.6 \pm 0.2	148.9 \pm 25.5	10.6 \pm 0.2	0.0 \pm 0.0	26.8 \pm 2.6	23.4 \pm 1.5	0.0 \pm 0.0	17.2 \pm 1.2
	MAP	5	5	612.0 \pm 84.4	48.6 \pm 2.6	5.0 \pm 0.0	5.6 \pm 0.2	507.3 \pm 256.1	10.8 \pm 0.4	0.0 \pm 0.0	43.8 \pm 2.2	33.6 \pm 1.4	0.0 \pm 0.0	17.7 \pm 1.8
	MILKBUCKET	5	5	1.9 \pm 0.1	4.8 \pm 0.2	3.0 \pm 0.0	2.0 \pm 0.0	47.9 \pm 8.1	10.0 \pm 0.0	0.0 \pm 0.0	2.8 \pm 0.2	16.2 \pm 1.3	0.0 \pm 0.0	10.0 \pm 2.0
	BOOKQUILL	5	2	30.9 \pm 11.8	24.0 \pm 3.0	4.0 \pm 0.0	4.0 \pm 0.0	1391.8 \pm 815.3	10.0 \pm 0.0	0.0 \pm 0.0	21.0 \pm 3.0	38.5 \pm 2.2	0.0 \pm 0.0	15.2 \pm 3.5
	SWEETMILK	5	5	13.0 \pm 1.9	17.0 \pm 2.0	4.0 \pm 0.0	4.0 \pm 0.0	135.3 \pm 21.0	10.2 \pm 0.2	0.0 \pm 0.0	13.8 \pm 1.9	26.6 \pm 1.1	0.0 \pm 0.0	11.9 \pm 1.7
	CAKE	1	0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0
	Completed Runs = 5				Total Time (s.) = 837.1 \pm 124.2					Total Calls = 180.4 \pm 7.0				
FRL	BATTER	5	5	11.1 \pm 1.4	23.4 \pm 2.5	6.0 \pm 0.0	9.2 \pm 0.2	468.4 \pm 121.9	10.4 \pm 0.2	7.6 \pm 0.9	11.4 \pm 1.9	11.9 \pm 0.6	10.1 \pm 1.3	9.9 \pm 0.4
	BUCKET	5	5	2.2 \pm 0.1	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	129.5 \pm 69.4	10.2 \pm 0.2	2.8 \pm 0.2	2.0 \pm 0.3	7.8 \pm 0.5	9.9 \pm 1.7	6.4 \pm 2.1
	COMPASS	5	5	12.9 \pm 1.9	24.6 \pm 2.2	6.0 \pm 0.0	9.4 \pm 0.2	550.8 \pm 156.4	10.4 \pm 0.2	7.8 \pm 1.0	12.4 \pm 1.2	12.5 \pm 1.6	9.4 \pm 1.0	8.4 \pm 0.5
	LEATHER	5	5	2.8 \pm 0.4	7.8 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	89.0 \pm 18.0	10.0 \pm 0.0	3.2 \pm 0.4	2.6 \pm 0.4	7.3 \pm 0.4	9.3 \pm 1.7	3.7 \pm 0.4
	PAPER	5	5	2.1 \pm 0.1	7.0 \pm 0.3	4.0 \pm 0.0	4.0 \pm 0.0	82.7 \pm 18.8	10.0 \pm 0.0	3.0 \pm 0.0	2.0 \pm 0.3	6.9 \pm 0.7	10.2 \pm 1.8	4.7 \pm 2.7
	QUILL	5	5	11.6 \pm 1.1	23.8 \pm 1.5	6.0 \pm 0.0	9.6 \pm 0.2	458.9 \pm 61.0	10.6 \pm 0.2	8.0 \pm 0.9	11.2 \pm 1.2	11.9 \pm 0.6	13.1 \pm 2.7	9.2 \pm 0.8
	SUGAR	5	5	2.6 \pm 0.3	8.4 \pm 0.7	4.0 \pm 0.0	4.0 \pm 0.0	103.5 \pm 39.5	10.0 \pm 0.0	3.6 \pm 0.4	2.8 \pm 0.5	8.2 \pm 0.7	10.1 \pm 1.9	5.0 \pm 1.1
	BOOK	5	0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0
	MAP	3	0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0	0.0 \pm 0.0
	MILKBUCKET	5	2	4.7 \pm 0.5	11.0 \pm 1									