Policy Search Based Relational Reinforcement Learning using the Cross-Entropy Method

A thesis
submitted in fulfillment
of the requirements for the degree
of
Doctor of Philosophy
in
Computer Science

at
The University of Waikato

by
Samuel Sarjant

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Department of Computer Science
Hamilton, New Zealand
2013

Abstract

Relational Reinforcement Learning (RRL) is a subfield of machine learning in which a learning agent seeks to maximise a numerical reward within an environment, represented as collections of objects and relations, by performing actions that interact with the environment. The relational representation allows more dynamic environment states than an attribute-based representation of reinforcement learning, but this flexibility also creates new problems such as a potentially infinite number of states.

This thesis describes an RRL algorithm named CERRLA that creates policies directly from a set of learned relational "condition-action" rules using the Cross-Entropy Method (CEM) to control policy creation. The CEM assigns each rule a sampling probability and gradually modifies these probabilities such that the randomly sampled policies consist of 'better' rules, resulting in larger rewards received. Rule creation is guided by an inferred partial model of the environment that defines: the minimal conditions needed to take an action, the possible specialisation conditions per rule, and a set of simplification rules to remove redundant and illegal rule conditions, resulting in compact, efficient, and comprehensible policies.

CERRLA is evaluated on four separate environments, where each environment has several different goals. Results show that compared to existing RRL algorithms, CERRLA is able to learn equal or better behaviour in less time on the standard RRL environment. On other larger, more complex environments, it can learn behaviour that is competitive to specialised approaches. The simplified rules and CEM's bias towards compact policies result in comprehensive and effective relational policies created in a relatively short amount of time.

Acknowledgements

First and foremost, my deepest gratitude goes to my chief supervisor Bernhard Pfahringer. Bernhard had already shown himself to be an excellent supervisor as my Honours supervisor, but he was even better for my PhD. He kept me motivated and focused, answered my many questions, suggested various improvements or alternatives to the algorithm, and was always happy to meet with me outside of our regular meetings. Bernhard's wealth of knowledge in many aspects of AI and his excellent eye for detail have helped shape this research into something far beyond what I could have ever done alone.

My other supervisors, Kurt Driessens and Tony Smith, have also been a great help throughout my research. Kurt, who was there at the beginning of RRL, helped me get started and directed towards a goal. He was also an excellent source of RRL information, both through direct communication and from his significant contributions to the RRL field (which may not even be where it is today if it were not for Kurt). What Tony lacked in RRL expertise, he more than made up for in his enthusiasm for my research and his impeccable spelling and grammar skills. Tony's background allowed him to provide interesting alternatives for the research, and his passion for Pac-Man also helped.

A big thank you to my examiners Dr. Peter Andreae at Victoria University in Wellington, New Zealand and Dr. Martijn van Otterlo at Radboud University, Nijmegen in the Netherlands. I met each of examiner early on in my PhD and each one helped me focus my research into what resulted in this thesis. They then graciously helped out once more by examining

the thesis, providing excellent feedback and suggestions that polished the work into the state it is today.

Thank you to my parents and siblings for shaping me into the person I am today. None of them may understand a word of what I am saying when I explain my research, but they at least courteously nod and smile. Thank you for supporting me throughout both my PhD and my life in general.

In Belgium, I thank Lieve and Maurice Bruynooghe for hosting me during my time there, and my coworkers in the oh-so-slightly crowded lab at the Catholic University of Leuven for helping me out and showing me around the city.

To all of my friends and coworkers at The University of Waikato: thank you. Going through a PhD has been much easier knowing that you are all suffering with me as well. My friends both inside and outside Uni provide the social interaction that I would go mad without and have made this journey enjoyable. I'd also like to thank the Tertiary Education Commission, BuildIT, and The University of Waikato Department of Computer Science for funding my research.

Other things that kept me sane are metal music, video-games, board-games (European style, of course!), D&D, and my many other geeky pursuits. Also, though I may not actively train anymore, I must thank Hanshi David Nips and all of my fellow martial artists at Taekidokai Martial Arts for strengthening my discipline, confidence, and resolve in many areas of my life.

Finally, I thank my partner of nearly six years, Darnielle for being a constant source of love, support, and amusement. She has been a motivating force, always quick to tell me if I was slacking... and also always ready to provide me with reasons to stay home for the day.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
CEM	Cross-Entropy Method
CERRLA	Cross-Entropy Relational Reinforcement Learning Agent
DP	Dynamic Programming
EA	Evolutionary Algorithm
GA	Genetic Algorithm
GGP	General Game Playing
ILP	Inductive Logic Programming
JESS	Java Expert System Shell
KL	Kullback-Leibler
LCS	Learning Classifier System
LHS	Left-Hand Side
LOMDP	Logical Markov Decision Process
MDP	Markov Decision Process
ML	Machine Learning
POMDP	Partially Observable Markov Decision Process
RHS	Right-Hand Side
RL	Reinforcement Learning

RLGG Relative Least General Generalisation

RMDP Relational Markov Decision Process

RRL Relational Reinforcement Learning

SARSA State-Action-Reward-State-Action

TD Temporal Difference

TL Transfer Learning

Publications

The following papers have been published throughout the course of this research:

Sarjant, S. (2013). A Direct Policy-Search Algorithm for Relational Reinforcement Learning. In *New Zealand Computer Science Research Student Conference (NZCSRSC) 2013*.

Sarjant, S. (2012). Using the online cross-entropy method to learn relational policies for playing different games. In *New Zealand Computer Science Research Student Conference (NZCSRSC) 2012*.

Sarjant, S., Pfahringer, B., Driessens, K., Smith, T. (2011) Using the online cross-entropy method to learn relational policies for playing different games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pp. 182–189. IEEE.

Sarjant, S. (2011). CERRLA: Cross-entropy relational reinforcement learning agent. In *New Zealand Computer Science Research Student Conference (NZCSRSC) 2011*.

Sarjant, S. (2010). Cross-entropy relational reinforcement learning. In *New Zealand Computer Science Research Student Conference (NZCSRSC) 2010*.

1

Introduction

Look around. What do you see? Perhaps a computer monitor, sitting on a desk before you. Perhaps a collection of pages bound together with ink printed upon them. You may even see other people, doing whatever it is that they're doing. You are probably reading this thesis because it has some meaning to you and your goal is to understand the information contained within. Reading a thesis (or any written document), involves relatively few actions. Turning the pages (or scrolling, if digital) and reading the information in front of you is basically all you need to do, perhaps occasionally looking up a cited paper that interests you. You will continue to read and turn pages until you have achieved your goal, whether that is to read the entire thesis, or just find the 'juicy parts.'

The above scenario could be represented as a Relational Reinforcement Learning (RRL) problem: there is a collection of objects (tangible and intangible) and relations between those objects (e.g. *contains(thesis, page1)* is a *relation* that states that the object *thesis* contains the object *page1*). An 'agent' (i.e. the reader) can act upon these objects with the intent of achieving a goal such as reading the entire thesis (e.g. *turnPage(thesis, page1, rightHand)* causes the agent to turn *page1* in *thesis* with its *rightHand*), preferably achieving that goal in a minimal amount of time. For example, some actions could be *turnPage(thesis, page1, rightHand)*, *readPage(thesis, page1, reader)*, *makeCoffee(reader, mug)*, etc. In this scenario, every second spent reading the thesis is a second not used for other enjoyable activities,[1] which

[1] But what could possibly be more enjoyable than reading this thesis?

could be represented numerically as a 'reward' of -1 per second with perhaps some large positive reward upon completing reading. Hence, the quicker an agent completes reading the thesis, the better the accumulated reward.

But how does an agent formally represent thesis-reading behaviour? Some approaches include:

1. A naive approach is to define the appropriate actions to perform *for every* possible state of thesis reading (e.g. per page, per thesis, per reading-format, etc.), but this approach is not generalisable and representation grows exponentially larger with the number of possible objects and relations involved in the thesis-reading problem.
2. A better approach is to define some form of abstraction, such that given a rough description of a state, the agent knows which action leads to maximal reward. This still requires the agent to learn which actions are best in what state, but the abstraction allows it to represent this information much more compactly than the first approach.
3. An even more general approach is to define some simple rules for reading the thesis: read the page until it is completed, then move on to the next page. This behaviour is the implicit result in the prior two approaches, but it skips the first steps of explicitly representing which actions have the greatest value.

The algorithm developed in this research attempts to learn behaviour for solving a problem using the third approach. The problem with this approach is that the algorithm needs to be able to create useful behaviour without explicitly learning per state which actions lead towards the greatest reward.

Before explaining the formal goal of this research, the following section provides a broad overview of the fields that it is based within.

1.1 Research Fields

1.1.1 Artificial Intelligence

The field of RRL is based within the broad field of Artificial Intelligence (AI). AI is a field within computer science that is concerned with the development of intelligent machines. This is a broad definition, as intelligence covers a wide range of behaviour and is difficult to formally define. There have been many definitions of AI, but they generally define AI as "an *agent* or *system* that 'thinks' and acts in a rational or human manner" (Russell and Norvig, 2003). Initially, early AI researchers were optimistic regarding how soon human-level intelligence AI was going to be developed, but this proved to be much more difficult than anticipated. AI research gravitated towards specialised applications (e.g. an AI that only plays Chess, or filters spam, etc.), but lately research has begun to return towards creating AI that can perform multiple tasks effectively.

1.1.2 Machine Learning

Machine Learning (ML) is a branch of AI concerned with learning solutions to problems when they are encountered, rather than simply acting out a rigid behaviour. A famous definition of ML by Tom Mitchell is:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E (Mitchell, 1997).

That is, if a program's performance increases after being provided with experience, it is said to be capable of learning. ML techniques can be broadly divided into three separate subfields: *supervised learning* (learning a model from labelled training data), *unsupervised learning* (learning the structure of unlabelled data), and *reinforcement learning* (learning which actions to take to maximise numerical reward).

1.1.3 Reinforcement Learning

Reinforcement Learning (RL) is a form of ML in which an *agent* seeks to maximise a *numerical reward* by performing *actions* within an *environment*.

Actions are selected by using the current observed *state* as an input to the agent's *policy*, which outputs the actions the agent takes. RL differs from supervised learning in that the 'correct' action is never explicitly stated; an agent only ever receives a numerical reward, and this reward may not even be received directly after an action is taken. For example, when playing a game such as CHESS or CHECKERS, a player only receives a single reward at the end of a game of either *win* (+1), *loss* (-1), or *draw* (0). The actions performed throughout the game contributed to this reward, and so an agent must learn a policy that outputs an effective combination of actions and achieves the greatest reward.

1.1.4 Relational Reinforcement Learning

Relational Reinforcement Learning (RRL) is the name given to RL performed within environments represented as *first-order* objects and relations between the objects. The relational representation allows a flexibility in the state and action descriptions that otherwise could not be achieved with standard reinforcement learning. A relational state can be composed of any number of objects and relations and the number of actions available to the agent also varies based on the objects and relations present. However, this flexibility also results in an enormous (even infinite) number of possible states, complicating the learning process. Since its conception in 1998 (Džeroski et al., 1998), numerous algorithms have been developed for solving RRL problems, though many have only been tested upon the benchmark BLOCKS WORLD environment (Section 3.3). The majority of RRL algorithms use value-based approaches to represent expected reward for relational states. An agent's behaviour is then extracted from these values by greedily performing actions with the largest expected reward.

1.2 Motivation and Goal

The goals of this research are to:

- Develop a new RRL algorithm that learns effective behaviour using direct policy search methods.
- Investigate the utility of the algorithm over a range of environments of differing sizes and formats.

The decision to learn behaviour via direct policy search methods was made because firstly, there already exists a large number of different value-based approaches with varying levels of performance, and secondly, direct policy search methods do not need to learn the expected value of actions, and so are unaffected by changes in the reward function (by changing the size of the environment, or as a result of modified behaviour).

The proposed approach for the algorithm is to utilise the Cross-Entropy Method (CEM), a distribution-based optimisation method, to generate rule-based policies by storing relational rules within distributions and generating policies by randomly sampling rules, where the probability of sampling a rule is increased with the rule's usefulness. This approach allows the algorithm to automatically explore different policies as random samples, but gradually modifies the sampling distribution such that the generated policies result in a greater reward. The CEM has been shown to be effective in a range of different problems, so this research will investigate an application of the CEM towards learning behaviour in RRL problems.

The second goal is concerned with applying RRL algorithms to larger problems. Most RRL algorithms are primarily evaluated on the benchmark 'Blocks World' environment, which is ideal for demonstrating the core challenges of RRL, but remains an artificial 'toy' problem. This research will be tested both upon Blocks World problems, and larger problems with more complex interactions. Games provide excellent environments for this purpose because they have a set of well-defined gameplay rules, an obvious reward function (the score), object-orientated elements, complex and often random gameplay elements, and are relatable to humans. To demonstrate the algorithm's ability to learn behaviour over a range of environments, three different games will be used as testbeds for the algorithm.

Regarding additional requirements, the algorithm should be able to:

- Learn behaviour quickly. If it takes a long time to learn effective behaviour, then the algorithm's usefulness is reduced.
- Learn effective behaviour without guidance from an external 'expert.' It needs to be able to infer its own useful rules when only provided with observations on the environment and the language in which the

environment is represented.

- Represent the behaviour in a comprehensible manner, such that it is obvious to a human viewer how the policy selects its actions.

1.3 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 describes the fundamental concepts behind this work and presents an introduction to the existing related work. These include an introduction to Reinforcement Learning (RL) and a brief description of the various approaches for RL algorithms, a formal description of Relational Reinforcement Learning (RRL) and a summary of the algorithms developed for it, and an overview of various AI applications to playing games.
- Chapter 3 formally defines the syntax that is used by the algorithm and the specification language that each environment is represented in. Each of the four environments used within this research are also formally defined here.
- Chapter 4 presents a full explanation of the algorithm developed in this research, the Cross-Entropy Relational Reinforcement Learning Agent (CERRLA). This chapter primarily describes how the algorithm utilises the CEM to explore and exploit the relational rules that are created by the algorithm (detailed in Chapter 5).
- Chapter 5 describes how the algorithm extracts information about the environment to create and explore relational rules for acting within the environment. This includes initial rule creation, specialisation conditions, and inferring simplification rules for removing redundant rule conditions and reducing the effective number of rules the algorithm needs to search.
- Chapter 6 presents evaluation results for CERRLA on the four environments defined in Chapter 3. These results include the performances on different environmental goals, the effects of alternative environmental representations, and comparisons to other learning algorithms.

- Finally, Chapter 7 discusses the algorithm presented in this dissertation and summarises the work presented in previous chapters, presenting conclusions on the outcome of the work and identifying possible future work.

A list of the figures, tables, algorithms, acronyms and publications can be found directly after the table of contents.

2

Background

The previous chapter introduced the concepts that are necessary for understanding the aim of this research. This chapter should give the reader a solid understanding of various solutions for Relational Reinforcement Learning (RRL) and how this research fits into the RRL context. This chapter describes the current state of RRL research, but before that, it describes the key concepts and existing approaches for Reinforcement Learning (RL) problems as many RRL algorithms are inspired by propositional RL algorithms and 'lifted up' to the relational setting. Three of the testing environments used in this research are games, so we also look at various AI applications towards playing games.

We begin by firstly reviewing the RL framework and existing approaches towards solving RL problems (Section 2.1). Section 2.2 then formally defines RRL, describing how RL aspects can be 'lifted' to the relational setting. We also examine existing RRL algorithms, investigating their relation to existing RL algorithms and their strengths and weaknesses. Section 2.4 outlines the various reinforcement learning and other related learning algorithm approaches that have been applied to games. Finally, Section 2.5 summarises the content presented in this chapter and discusses how it applies to the algorithm presented in the following chapters.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a method of machine learning in which a learning *agent* seeks to maximise a numerical *reward* by interacting with its *environment* (Sutton and Barto, 1998; Kaelbling et al., 1996). An *agent* interacts with an *environment* in discrete time steps t and at every time step the environment provides a description of the current *state* of the environment s_t to the agent. The agent then selects an *action* a_t to perform, which is returned to the environment. This causes the environment state to transition to another state s_{t+1} and produce numerical feedback about the quality of the state transition. The goal of the agent is to maximise the overall feedback received. Figure 2.1 presents an illustration of the reinforcement learning loop.

Example 2.1.1. For example, an agent's interaction with a generic environment described by a set of numerical features is as follows:

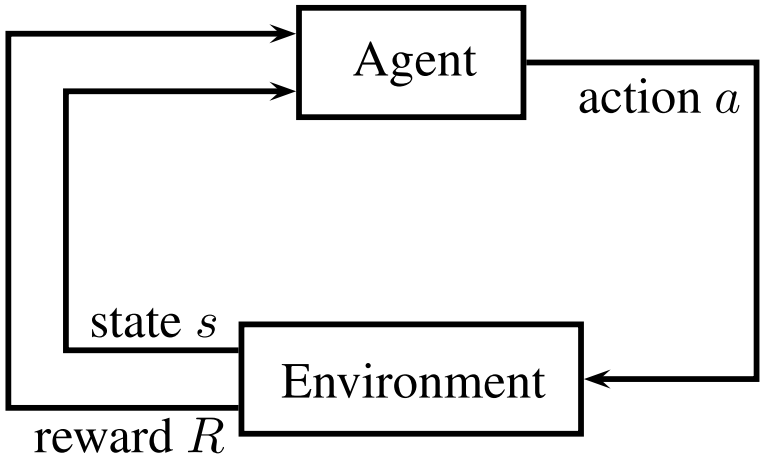


Figure 2.1: An illustration of the reinforcement learning framework.

Environment:	At time step 0, you are in state 23. Feature 2, 6 and 9 are true. You have 4 possible actions.
Agent:	I'll take action 3.
Environment:	You receive a reward of 2. At time step 1, you are now in state 16. Feature 2, 3 and 5 are true. You have 6 possible actions.
Agent:	I'll take action 1.
Environment:	You receive a reward of -4. At time step 2, you are now in state 3. Feature 9 is true. You have 2 possible actions.
⋮	⋮

Unlike most forms of machine learning, the 'correct' action is not known to the agent; there is only a numerical reward. This is one of the main challenges in RL: the problem of *exploration vs. exploitation*. The agent needs to *exploit* actions that it knows produce high reward, but also needs to *explore* other actions to check if they produce even higher reward. A greedy agent would simply exploit the first strategy that provides reward, which is probably not optimal, therefore a learning agent needs some sort of exploration strategy. This is complicated by the fact that the learning is performed *online* within the environment, meaning the environment is a 'black box'; states can only be accessed by taking the necessary actions to get to them. *Offline* learning allows an agent to select any state and perform an action, but this thesis will not cover this form of RL.

A comprehensive explanation of RL techniques can be found in Kaelbling et al. (1996), Sutton and Barto (1998), Szepesvári (2010), Buşoniu et al. (2010) and Wiering and van Otterlo (2012).

2.1.1 Markov Decision Process

Markov Decision Processes (MDP) (Bellman, 1956; Puterman, 1994) are an intuitive framework for representing reinforcement learning (Bertsekas and Tsitsiklis, 1996; Kaelbling et al., 1996; Sutton and Barto, 1998), decision-theoretic planning (Boutilier and Dearden, 1994) and other stochastic state-driven domains. A Markov Decision Process (MDP) represents a problem as a set of connected states that are navigated by selecting actions. Each transition between states has a probability and a reward associated with it and the goal of the agent acting within the MDP is to maximise the amount

of reward received by selecting an appropriate action at each time step.

Definition 2.1.1 (Markov Decision Process (MDP))**.** Formally, an MDP is a tuple $M = \langle S, A, T, R \rangle$, defined as:

- A finite set of states S,
- A finite set of actions A,
- A transition function $T : S \times A \times S \to [0, 1]$,
- A reward function $R : S \times A \times S \to \mathbb{R}$.

For every state $s \in S$, the agent is provided with the set of actions $A(s)$ that can be performed for the current state. When action a is applied in state s, the transition function defines the probability of transitioning to state $s' \in S$ as $T(s, a, s')$. Every $T(s, a, s') \geq 0$ and $T(s, a, s') \leq 1$, and for every s and a, $\sum_{s' \in S} T(s, a, s') = 1$. A numerical reward is also produced using $R(s, a, s')$, where the value may be any real numerical value.

The agent's job is to learn a *policy* $\pi : S \to A$ (or a probabilistic policy $\pi : S \times A \to [0, 1]$, but this work focuses on the deterministic form), which maps states to actions ($\pi(s) = a$). The policy is the agent's method of interaction with the environment and the agent's goal is to create a policy that receives maximal reward when interacting with the environment.

An MDP may also specify a distribution of starting states and/or terminal states. The starting states define the first state an agent may begin in when learning begins, and terminal states define states in which the *episode* is complete (either because the agent reached the goal, or cannot act anymore).

A core aspect of MDPs is the *Markov assumption* which states that: "the current state provides enough information to make an optimal decision." This clause restricts the number of environments that fit into the MDP framework, as environments that are not fully-observable (e.g. hidden-information domains such as Poker) do not fit this assumption. Nonetheless, the MDP framework provides an approximate description for such environments.

Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) is a generalisation of an MDP where the agent does not have access to all state observations (Kaelbling et al., 1998). A POMDP assumes there is an MDP modelling the environment, but the agent only has access to a partial observation of it. Many real-world environments are only partially observable, due to an element of randomness, imperfect sensors, the presence of other 'black box' agents, etc. The three game environments presented in the next chapter could all be classified as POMDPs because each environment contains competing agents with unknown behaviour (as well as other unknown elements of the environment).

Definition 2.1.2 (Partially Observable Markov Decision Process (POMDP))**.** Formally, a POMDP is a tuple $\langle S, A, O, T, R, \Omega \rangle$, such that S, A, T, R are defined as usual, O is a set of observations upon the actual state S, and Ω is the observation function $\Omega : S \times A \times O \rightarrow [0, 1]$ defining a probability distribution over observations received given an action and resulting state.

A POMDP can be treated as an MDP, but the learning algorithm may need to make use of a *belief state* to probabilistically infer what fully-observed state the agent is in. Without knowing what state the agent is actually in, calculations that make use of previous rewards cannot be effectively utilised.

2.1.2 Solving Markov Decision Processes

The most obvious approach to solving reinforcement learning problems is to maintain a *value function* $V^\pi = S \rightarrow \mathbb{R}$ that returns an *expected reward* for state s if following policy π. The goal is then to create a policy π^* such that the value function V^{π^*} achieves the maximal possible reward in every state. Value functions are defined as:

$$V^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})] \tag{2.1}$$

where $0 < \gamma < 1$ (usually $\gamma = 0.9$) to prevent the sum of rewards going to infinity in environments without a terminal state. This definition states that the value of a state while following policy π is equal to the expected reward of all following states.

A value function can also be recorded for every action a in state s, known as the Q-function $Q^{\pi}(s,a)$ (Quality-function). Instead of estimating the expected reward for every state, the Q-function estimates the expected reward for every state-action pair:

$$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a, s_{t+1})] \tag{2.2}$$

The values of each function can be estimated by recording an average of the rewards following each state, or for the Q-function, the rewards following each individual action taken from the state. As the number of times the state value is updated approaches infinity, the estimated value becomes closer to the true value of the state (or state-action).

Existing Value-Based Algorithms

Reinforcement learning problems can be solved with two main approaches: learning (or being provided with) a model of the environment and using *dynamic programming* (DP) to iteratively determine the optimal policy, or learn the values for states with *temporal-difference learning*.

Dynamic Programming (DP) computes the value of states by iteratively propagating rewards back through the MDP using the known transition and reward functions. While not strictly part of RL, DP provides an alternative method of solving MDPs. DP approaches are guaranteed to find the optimal policy because the optimal value function V^* can be represented as the following equation (using the *infinite horizon metric* as the optimality metric, Equation 2.1):

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^*(s') \Big) \tag{2.3}$$

which can be used to calculate the optimal policy π^* (by always selecting the action that leads to the greatest reward). This equation is known as the *Bellman optimality equation* (Bellman, 1956) which states that the value of a state s is equal to the immediate reward received $R(s, \pi(S), s')$ following the current policy π plus the average expected reward for the following states $V(s')$ with respect to their transition probability $T(s,a,s')$.

Two core DP algorithms are *value iteration* (Bellman, 1956) and *policy it-*

eration (Howard, 1960). Value iteration computes the optimal value function using the Bellman optimality equation to propagate rewards between states. Policy iteration switches between recomputing the value function and improving the current policy based on the recomputed value function, eventually converging to an optimal solution. Both methods are guaranteed to find the optimal solution as each iteration always improves the quality of the agent's behaviour.

The main problem with dynamic programming approaches is that the transition and reward functions are usually not known. In this case, an agent must learn the transition and reward functions if it is to use DP techniques (known as *indirect* RL). The DYNA architecture (Sutton, 1991) combines Q-learning and DP by learning a model while concurrently acting in the environment. The learned model is then used to generate extra learning experience by simulating extra interaction with the environment. Prioritised Sweeping (Moore and Atkeson, 1993) improves upon this idea by prioritising updates of the learned model to areas where the change in observed values is greatest.

The other option for value-based learning is Temporal Difference (TD) learning (*direct* RL). TD learning incrementally updates the expected value of states using the immediate observed reward and the estimated rewards of future states (known as *bootstrapping*). TD(0) (Sutton and Barto, 1998) is the simplest form of TD learning. At every time step, the algorithm can update the value of state s using the observed reward r and value estimate for the following state $V(s')$.

$$V_{k+1}(s) = V_k(s) + \alpha\Big(r + \gamma V_k(s') - V_k(s)\Big) \tag{2.4}$$

where $\alpha \in [0, 1]$ is the *step-size* (or *learning rate*) parameter that controls how much values get updated. Like Equation 2.3, the value of a state depends on following states, but instead of a weighted average of all following states, TD learning only uses the *observed* transition to update the value.

To avoid converging to a single, possibly sub-optimal strategy and to search for potentially better strategies, the agent needs to occasionally explore actions that are not simply selecting the action that leads to the largest expected reward. The simplest approach is *ϵ-greedy exploration* which selects

a random action with probability ϵ, otherwise it selects a greedy action. A problem with this strategy is that it performs unnecessary exploration in the later stages of learning and that random actions can lead to highly undesirable states. Another approach is to use *Boltzmann exploration*, which uses a 'temperature' variable T and the current Q-value estimates to control action selection:

$$P(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{a' \in A(s)} e^{\frac{Q(s,a')}{T}}} \tag{2.5}$$

where $P(a)$ defines the probability of selecting action a. The temperature T is gradually decreased to reduce exploration and increase exploitation.

Q-learning is a popular variation of TD learning, which learns Q-values for states in a similar manner to TD(0) (Watkins and Dayan, 1992). The Q-learning update equation is:

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\Big(r + \gamma \max_{a' \in A(s')} Q_k(s',a') - Q_k(s,a)\Big) \tag{2.6}$$

This equation is nearly identical to Equation 2.4, except it utilises the max operator to select the best estimated value for the next state. Because of this operator, Q-learning is an *off-policy* algorithm, because it only updates Q-values with the best estimated values. An *on-policy* variation of Q-learning is SARSA:

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\Big(r + \gamma Q_k(s',\pi(s')) - Q_k(s,a)\Big) \tag{2.7}$$

Note that the SARSA equation uses the policy's output action for Q-value updates instead of the max operator. Both techniques are guaranteed to converge to the optimal solution given infinite samples, but SARSA requires that the algorithm eventually ceases to explore.

An alternative value-based class of algorithms are *actor-critic* methods (Witten, 1977; Barto et al., 1990; Konda and Tsitsiklis, 2003). Actor-critic methods maintain an explicitly separate policy to the value function. The policy is known as the *actor* because it selects the actions and the value function is known as the *critic* because it criticises the actions performed by the policy. The criticism is in the form of TD error:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.8}$$

If positive, the critic strengthens the probability of selecting the action and vice-versa. The following update equation defines the *preference* of selecting action a_t in a given state s_t (where actions with a higher preference are more likely to be selected):

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t \tag{2.9}$$

where β is a step-size parameter determining how much the value is updated. Note that together, Equation 2.8 and 2.9 are very similar to the value and Q-learning update equations.

2.1.3 Generalisations and Abstractions

In small environments, maintaining a table of values for every state (or every state-action in the case of Q-learning) is enough to learn an optimal policy in reasonable time. But for larger or more complex environments, the size of the table grows exponentially larger and becomes difficult to manage, both in terms of memory usage and value-propagation, resulting in a slower rate of learning. The following subsections define core generalisations or abstractions that can be applied to RL techniques to reduce or approximate the state space of an environment.

Value Function Approximation

Instead of recording the expected value of each state directly, a parameterised function can be used to represent the value function of states (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998; Buşoniu et al., 2010). By representing the expected reward of states as a function, the agent only needs to learn the function, which allows it to estimate the expected value of unseen states as well. This is known as learning a regression model for predicting a state's estimated value. A learned model takes a state (and action) as input and outputs the expected value of the state, in accordance with the model.

Learning a regression model is a well-understood problem in *supervised learning*, but the key difference with learning a regression model in RL is that learning is performed online, with non-stationary expected state rewards. Therefore, the regression model needs to be able to incorporate new examples and changes to the existing data as the expected values for states

change. This can be achieved either by using an *incremental regression model* or learning function approximators with *batches* of examples (known as *batch RL*). Theoretically, any supervised learning algorithm can be used as a function approximator, either in an incremental or batch fashion, e.g. *linear function approximation* (Samuel, 1967; Utgoff and Precup, 1998), *decision trees* (Chapman and Kaelbling, 1991; Wang and Dietterich, 1999), *neural networks* (Tesauro, 1994; Bertsekas and Tsitsiklis, 1996), *evolutionary methods* (Whiteson and Stone, 2006), *kernel-based methods* (Ormoneit and Sen, 2002) and *support-vector machines* (Dietterich and Wang, 2001).

Direct Policy Search

Most approaches in RL learn an optimal policy by maximising the expected value of actions, where the expected value is either computed from a table of values or a value-function approximation. Direct policy search completely skips the need for expected values by instead computing a policy directly. Value functions are typically larger than the policies generated from them, and usually encode more information than the policy requires. For example, given a simple 'corridor' environment of length ten, where the actions available are to go `left` or `right`, and the goal is to be in the far right edge of the corridor, a value function needs to represent the expected value for each state and each action, whereas a policy simply needs to say *go right*.

An obvious method of learning policies is to treat RL as a *supervised learning* problem by gathering a number of good examples (goal-achieving) and use them as input to a classification model. This method transforms RL into a series of supervised learning tasks (Barto and Dietterich, 2004; Langford and Zadrozny, 2005). Lagoudakis and Parr (2003) use an *approximate policy iteration* (API) framework that uses *policy rollouts* (Boyan and Moore, 1995) to create a number of policy samples as input for a classifier algorithm.

P-learning (Džeroski et al., 2001) is partially a value-function approximation method as it maintains the same structure as Q-learning, but instead of encoding the expected state values, it simply encodes whether an action within a state is optimal or not (1 for optimal, 0 for non-optimal). This usually results in a smaller representation of the state compared to Q-learning, and does not need to maintain the expected value of state-actions.

A *policy gradient* approach uses gradient-descent techniques to locate the optimal policy. By representing the policy such that a gradient can be defined for its parameters, the optimal policy can be found by using techniques such as hill-climbing. The REINFORCE algorithm (Williams, 1992) learns the policy gradient by repeatedly testing the policy against the environment, then updating the weights of the policy through hill-climbing. A problem with this algorithm is that it is on-policy and can be relatively slow. Sutton et al. (1999a) extend the REINFORCE algorithm by combining it with function approximation to aid the policy gradient estimation and speed up the rate of convergence.

Baird and Moore (1999) present an alternative policy gradient method named VAPS (Value and Policy Search). The VAPS algorithm combines both value function approximation and policy search, allowing the agent to select actions using either technique. Wierstra and Schmidhuber (2007) adapt the actor-critic method to learn policy gradient critics in POMDP environments.

Policies can be generated through the use of an Evolutionary Algorithm (EA) (Holland, 1992; Goldberg, 1989), either by evolving the entire policy, or the individual rules that compose the policy (Moriarty et al., 1999). EA maintains a *population* of *chromosomes*, where the best chromosomes are *mutated* to produce different chromosomes. EAs require two key factor for creating policies: 1) an evolvable policy representation, 2) an appropriate fitness function for the policies. The reward function typically serves as the fitness function (i.e. reward received for the episode). EAs have been combined with RL to evolve rule-based policies (Smith, 1983; Grefenstette et al., 1990) and neural networks (*neuro-evolution*) (Belew et al., 1992; Whitley et al., 1993; Moriarty et al., 1999).

An alternative application of EA is to learn parts of the policy and bring them together in combination. In these systems, learning is performed both on the overall structure of the policy and on the individual components that make up the policy. Learning Classifier Systems (LCSs) (Holland, 1995; Lanzi et al., 2000) use a EA and RL techniques to maintain a population of 'if-then' *classifiers* that map input to an action. The classifiers represent the agent's policy, so when input is received, *all* classifiers with matching conditions activate. Every classifier has a *strength* associated with

it that records the expected reward (like value-based RL techniques) and that strength is also used as a fitness function for selecting classifiers for genetic mutation operations. Dorigo and Colombetti (1998) combine several LCSs hierarchically to learn behaviour for multiple subtasks.

The XCS classifier (Wilson, 1995) alters the LCS algorithm by using the *accuracy* of a classifier as the fitness function for genetic mutations instead of the strength. The accuracy of a classifier is the error between the classifier's expected reward and the actual reward received. The fitness of a classifier is a function of the inverse error such that classifiers that accurately predict the reward received are more favourable than those that simply have a high, but erroneous, expected reward. The 'HAYEK machine' (Baum, 1999) is similar to an LCS in that it maintains a collection of agents that bid on which actions to take, where agents that bid on high-quality actions receive a relative reward. This strategy allows each agent to focus its bids and rule learning on sub-problems within the environment. The rules each agent uses are created through evolutionary methods of mutation and random initial conditions.

Symbiotic, Adaptive Neuro-Evolution (SANE) (Moriarty and Mikkulainen, 1996) uses a neuro-evolution approach to learning behaviour by using *symbiotic evolution* to learn weights for individual neurons within a larger fixed-size neural network. Each neuron only learns a portion of the policy but they rely on other neurons to create effective behaviour. Neuro-Evolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) is an extension to SANE that allows the topology of the network to change rather than using static-structure networks. Potter and De Jong (2000) define a rule-based form of symbiotic evolution, where each chromosome in the population represents a set of rules that only address a subset of the task.

The Cross-Entropy Method (CEM) is a relatively recent optimisation algorithm similar to Learning Classifier Systems and Evolutionary Algorithms (Rubinstein, 1997; De Boer et al., 2004). The CEM can be summarised in two steps: 1) Generate a number of random samples from the current distribution of data, 2) Update the data distribution such that the best subset of the random samples (*elite samples*) are more likely to be sampled in the next iteration (i.e. minimise the *cross-entropy* distance between the current distribution and the observed elite samples). Applied to RL problems, it

can be used to generate a number of policies, the best of which are used to influence the sampling distribution such that they are more likely to be randomly sampled again. Although CEM has been applied to a multitude of different problems, this subsection only describes the applications of CEM to RL. Other applications include clustering (Kroese et al., 2007), control and navigation (Helvik and Wittner, 2001), DNA sequencing (Keith and Kroese, 2002), network reliability (Hui et al., 2005), and continuous optimisation (Kroese et al., 2006).

Mannor et al. (2003) demonstrate a simple application of CEM to a maze-traversing RL by representing the agent's policy as an action distribution (e.g. move up, down, left, or right) for every location in the maze. The algorithm learns by generating N policy samples, testing them, and the best policy samples (*elite samples*) are used to update the sampling probabilities for every action distribution, such that favourable actions are more likely to be sampled. Chaslot et al. (2008) apply CEM towards playing the board game Go by using it to tune the parameters of a *Monte-Carlo Tree Search* algorithm, improving the results of the algorithm over the non-tuned learner. Szita and Lörincz (2006) apply CEM to the Tetris video-game by representing the policy as a vector of weights for features in the game. They also inject noise into the sampling process to reduce the likelihood of early convergence. Thiery and Scherrer (2009) improve upon this work by adding additional features and Kistemaker (2008) also applies CEM to learning to play Tetris.

Szita and Lörincz (2007) create rule-based decision-list policies for playing the Ms. Pac-Man game by using the CEM to identify which rules are useful and what order they should be used in. Each decision-list policy is created from multiple rule distributions by sampling one rule from each distribution where each distribution also has a probability of being included in the sampled policy. Each distribution contains the same 'condition-action' rules that use high-level actions for the agent's behaviour (e.g. `toDot`, `fromGhost`, etc. rather than directional movement). The order of the rules in the policy is dependent on the order of the distributions. The sampled policies are also hierarchically structured such that multiple rules can be activated at once (e.g. Ms. Pac-Man can eat dots while avoiding ghosts). The algorithm evaluates a number of sampled policies and uses the best policies (the elite policies) to alter the rule distributions such that effective rules

are more likely to be included in later policies. The authors use predefined rules, but they also run experiments using (bounded) randomly generated rules. The research presented in this thesis was initially based upon this work and uses a similar (relational) method of acting in the Ms. Pac-Man environment.

2.1.4 Reinforcement Learning Summary

The above summary of algorithms shows that there already exist a large number of solutions for RL problems but all solutions have a common weakness: they can only learn behaviour in environments where the representation is a static set of features. In many cases this is sufficient, but often an environment will utilise a changing number of objects and relations between objects. This can be dealt with by adjusting the state representation to represent all possible aspects of the environment but every additional object or relation can exponentially increase the number of features required to model every possible state of the environment. The following section introduces Relational Reinforcement Learning (RRL), a subfield of RL in which an environment is represented as a collection of objects and relations, providing more freedom in expressing the environment state.

2.2 Relational Reinforcement Learning

Relational Reinforcement Learning (RRL) is a representational generalisation of RL that expresses the environment as logical relations between objects and actions taken upon those objects (Džeroski et al., 2001). Traditional RL algorithms are based in *propositional* environments, where the structure of the states is fixed. But in more complex environments this form of state representation is not sufficient. States can be dynamically changing, introducing new objects or removing old objects. RRL represents these object-based environments using first-order logic, both for observations on states, and for actions to take within states.

RRL is strongly based on the field of Inductive Logic Programming (ILP), a subfield of ML in which hypotheses are inductively learned from a set of logically-defined examples (Lloyd, 1993; Genesereth and Nilsson, 1987; Muggleton, 1991; Dzeroski, 2001). Examples are represented as sets of facts

consisting of objects and relations, and additional information about the examples can be inferred using background knowledge to infer new facts. ILP's expressive representation of facts is ideal for representing problems with a non-fixed number of features.

The syntax used in RRL is strongly based on ILP syntax, defined below (for a full definition, see Lloyd (1993), Genesereth and Nilsson (1987) or Dzeroski (2001)):

Definition 2.2.1 (Logic Programming Syntax)**.** Each environment defines an *alphabet* Γ of *predicates* that make up the *relational state observations*, a separate set of predicates that make up the available *relational actions* an agent can take, and a set of named objects (*constants*) that are present within the environment. Each predicate is instantiated with *terms*: either a *constant*, *variable*, or *function*. *Constants* represent unique objects within the environment, *variables* are placeholders for constants, and *functions* return a value when provided with argument terms. An *atom* is a predicate that contains terms. A *literal* is a negated or non-negated atom. If an atom or literal does not contain any variable terms, it is *grounded*. A *substitution* is a set of assignments of terms to variables, where each variable is only assigned a single term.

The *Herbrand base* of Γ (HB^{Γ}) is the set of all ground atoms that can be constructed with the state predicates P_S (and action predicates P_A) and the constants C. A *Herbrand interpretation* is a subset of HB^{Γ}.

The main advantage of this alternative representation is the flexibility in expressing facts about the environment. Where traditional RL defines a fixed set of attributes with which to represent the environment, RRL is able to describe any number of facts about any number of objects.

2.2.1 Relational Markov Decision Process

Relational environments are structured using the Relational Markov Decision Process (RMDP) framework; an extension of the MDP framework seen in Section 2.1.1. There are multiple definitions of RMDPs (e.g. Wang et al. (2008), Croonenborghs et al. (2007), Kersting and Raedt (2004), Fern et al. (2006)), and we use the same one given in Croonenborghs et al. (2007):

Definition 2.2.2 (Relational Markov Decision Process (RMDP))**.** An RMDP

is defined as the five-tuple $M = \langle P_S, P_A, C, T, R \rangle$, where P_S is a set of state predicates, P_A is a set of action predicates, and C is a set of constants. A ground state (action) atom is of the form $p(c_1, \ldots, c_n)$ with $p/n \in P_S$ ($p/n \in P_A$) and $\forall i : c_i \in C$. A state in the state space S is a set of ground state atoms; an action in the action state A is a ground action atom. The transition function T and reward function R are defined as usual by $T : S \times A \times S \to [0, 1]$ and $R : S \times A \times S \to \mathbb{R}$.

The Herbrand base for an RMDP defines all the possible atoms used to describe a state, though not every atom is necessarily legal. Some combinations of constants could be infeasible for the current environment, and some combinations of atoms could represent illegal states.

Note that compared to an MDP definition, the RMDP definition is one which implicitly defines the state and action space, as it simply defines the components that compose the state and action space. Because states may have any number of facts, an explicit definition of the state space is impossible because the number of states may be infinite. However, this flexibility is also the primary benefit of RRL as there are no restrictions on which objects or facts are present.

An example environment commonly used in RRL and planning algorithms is the Blocks World environment (Slaney and Thiébaux, 2001). It consists of a number of blocks stacked on top of one-another, and a floor upon which to stack the blocks. A full definition of Blocks World can be found in Section 3.3. A possible RMDP for a small Blocks World is as follows:

Example 2.2.1 (Blocks World RMDP)**.** Defining the Blocks World alphabet as $P_S = \{on/2, clear/1\}$ (such that *on* takes two arguments and *clear* takes one argument), $P_A = \{move/2\}$, and $C = \{a, b, c, d, e, f, fl\}$, a possible state could be $s_1 = \{clear(a), on(a, b), on(b, d), on(d, e), on(e, c), on(c, fl)\}$. s_1 defines a single stack of blocks, with a on top and a single action $A(s_1) = \{move(a, fl)\}$, which moves block a to the fl, resulting in state s_2 (with some probability given by $T(s_1, a_1, s_2)$ and reward given by $R(s_1, a_1, s_2)$). Note that the block f is not present in the state, because relational states do not necessarily need to include every object.

2.2.2 Benefits and Challenges of RRL

The relational format has several benefits over the propositional representation in RL:

- States may contain any (legal) combination of objects and relations. Each state is a snapshot of the current state of the environment, with information about each object and the relations between the objects composing the state description.
- Actions can directly relate to the objects in the state. In propositional representations, actions may only implicitly relate to objects (e.g. $openDoor1$, $openDoor2$), whereas relational representations explicitly define the objects required for the action (e.g. $open(door1)$, $open(door2)$).
- The first-order representation allows agents to leverage variables to generalise across objects. This is one of the most important abstractions of RRL as it allows an agent to define generalised behaviour by acting upon objects that satisfy the relational properties, rather than defining behaviour for each individual object.
- *Background knowledge* can be provided by the environment that defines rules to automatically infer new facts and define illegal states.

However, it also introduces a number of new challenges as well:

- The flexibility of state descriptions results in an enormous number of possible states, even when using *background knowledge* to remove illegal states. This makes brute-force state-action tables impractical, so abstractions must be used to create effective learners.
- Measuring distances between first-order states is more difficult than propositional representations due to the variable number of facts and objects present between states.
- First-order reasoning is generally slower than propositional methods.

van Otterlo and Kersting (2004) provide more detail about the challenges faced by RRL.

2.3 Existing RRL Algorithms

This section briefly summarises the distinct approaches that have been used to learn behaviour within RRL problems, many of which are based on techniques presented in Section 2.1. Refer to the following for comprehensive surveys on RRL techniques: van Otterlo and Kersting (2004), Tadepalli et al. (2004), van Otterlo (2005), van Otterlo (2009), or the most recent survey Wiering and van Otterlo (2012).

There are three primary approaches towards solving RRL problems: *static generalisation methods*, which provide the environment generalisations for value-based methods prior to learning (model-based methods also fall into this approach); *dynamic generalisation methods*, which create generalisations for the environment and use value-based methods for learning; and *policy search methods*, which create policies directly, thereby removing the need to generalise the states of the environment. This section will describe each approach and discuss existing algorithms that have been created for each approach.

Static Generalisation Methods

Static generalisation methods provide an abstraction of the state-actions table such that each entry represents a partial, possibly variable, abstract state. Standard value-based learning techniques are then used to locate the optimal policy for the abstract state-action table. This includes algorithms such as CARCASS (van Otterlo, 2004), Logical Markov Decision Process (LOMDP) framework (Kersting and Raedt, 2004), and Relational Q-learning (RQ) (Morales, 2003).

Dynamic Generalisation Methods

One of the earliest algorithms for solving RRL problems was the RRL-system (Džeroski et al., 2001; Driessens, 2004), which combined RL and ILP to define a general system for learning Q-functions in RRL problems. The basic idea of the algorithm is to gather a collection of state-action examples over the course of an episode, updating the Q-value of each state-action pair using Q-learning, then use the examples as input into a relational regression classifier to produce a compact classifier that predicts the Q-value for a state-action.

The first implementation of the RRL-system used the TILDE-RT relational decision-tree learner (Blockeel and De Raedt, 1998) to represent the Q-function (denoted as the Q-RRL algorithm). Each node contains a test consisting of a single query that may share variables with other nodes and the leaves of the tree contain Q-values. The main problems with Q-RRL is that it needs to store each state-action example in order to learn an accurate function approximator and that it builds a new tree after every episode (which takes increasingly longer as the set of examples grows). Driessens (2004) created several incremental algorithms that remove the need to store each example: RRL-TG, an incremental relational decision-tree learner that learns trees of the same structure as the TILDE-RT (Driessens et al., 2001); RRL-RIB, an instance-based learner that uses a set of well-chosen, experienced state-action instances to calculate *distances* for state-action pairs (Driessens and Ramon, 2003). This distance metric needs to be defined beforehand by the user. RRL-KBR uses graph kernels and Gaussian processes as a regression technique for approximating the value of state-action pairs (Driessens et al., 2006).

The RRL-TG algorithm was also extended into several different directions: TGR incorporates tree-restructuring operations to mitigate the effects of ineffective splits or tests by pruning sub-trees or revising the split (Ramon et al., 2007). TGR is also able to capably deal with *concept drift* (goal of the environment changes). Driessens and Džeroski (2005) combined RRL-TG and RRL-RIB to create TRENDI, a tree-based model with instance-based representation for the leaves of the tree.

The NPPG algorithm (Kersting and Driessens, 2008) uses policy-gradient techniques to optimise a weighted sum of regression models created in stage-wise optimisation during learning. The regression models are created using *boosting*: each regression model is created to cover the examples previous models do not adequately cover. In the relational setting, this can mean creating a model for previously unseen features of the state. Each model is also *weighted* by a value that is multiplied with the model's output predictions. This value can be changed using policy gradient techniques to create better weights. The overall combined model resulting from the regression models accurately approximates the value function for relational, propositional and continuous domains (by using the appropriate regression models for the problem) and does so relatively quickly. Natara-

jan et al. (2011) adapted this algorithm for imitation learning in relational domains.

An alternative to learning expected values is to learn the *structure* of an environment as a probabilistic model of transitions and reward, then use the model to create the best policy. SVRRL (Sanner, 2005) learns new features by probabilistically observing frequencies of joint features as input to a relational naive Bayes network of success within win-lose environments (environments in which the agent receives a single terminal reward: either 1 or 0). It is able to learn a concise set of features and an effective strategy in a relatively low number of training episodes. The QLARC algorithm (Croonenborghs et al., 2004) learns probabilistic rules for the effects actions have upon the state that the agent can utilise to perform look-ahead optimisation for achieving the goal. MARLIE (Croonenborghs et al., 2007) extends this by learning a probability tree that models if a given predicate will be true after taking an action, given the current state and action.

Policy Search Approaches

RRL algorithms employ many of the same direct policy search algorithms as seen in RL algorithms. As with RL techniques, direct policy search algorithms are able to ignore the value-function representation of the state space, which is a considerable advantage in relational state spaces, which can be enormous (or even infinitely large). Furthermore, policies are able to generalise over unseen states and scale to larger problems.

As with RL, the most straightforward approach to learning a policy is to treat the RRL problem as a classification problem by using Inductive Logic Programming (ILP) algorithms to learn the policy. The Q-RRL algorithm (Džeroski et al., 2001) uses the TILDE algorithm to learn a P-function in place of a Q-function (that is, assign actions as *optimal* or *non-optimal*). This variation is known as P-RRL. Other approaches (Khardon, 1999; Yoon et al., 2002; Martín and Geffner, 2004) also follow a similar method of gathering state-action pairs of optimal policies for learning a model of the policy using the pairs as positive examples. One problem with supervised learning approaches is that learning requires sufficient positive examples to learn an effective model of the policy. Driessens and Džeroski (2004) demonstrate the positive effects of introducing guidance to learning, however an agent may not always have an oracle from which to get guidance.

The LRW-API approach (Fern et al., 2006) learns a policy by iteratively performing batches of *policy rollouts* (Boyan and Moore, 1995) as an *approximate policy iteration* algorithm. The algorithm assumes an environment model, such that any state can be sampled at any time, and estimates the value of an action by creating w policy trajectories of length h from state s. These trajectories approximate the expected *advantage* of each action, which is the expected gain over existing Q-value estimates, such that learning is focused on examples with a greater advantage. The algorithm is able to learn policies for complex problems by first *learning on random worlds* (LRW) which are initially very small. Each world is created by performing n random actions to set a goal state which is not trivial to reach with the current policy. A policy is learned to solve that goal, then n is increased again, and so on. The main disadvantage of this method is the 'controlled experiment' assumption that the world model can be accessed at any state, whereas RRL world models are typically 'black boxes' that only allow a single action per state.

There are a number of evolutionary approaches to creating relational policies. GREY (Muller and van Otterlo, 2005) is an application of the standard Genetic Algorithm (GA) (Goldberg, 1989) to learning relational decision list policies. Each *chromosome* in the *population* is a relational policy containing a set of probabilistically evaluated relational rules. Mutation involves adding or altering rules (by adding conditions or grounding variables to constants) in a policy and combination of chromosomes using one-point crossover to combine rules from different policies. GAPI (van Otterlo and De Vuyst, 2009) is a similar implementation using the GA, but it includes the use of *goal variables* which allow the learned policies to be parameterisable. The previously mentioned HAYEK machine (Baum, 1999), which can also be altered to operate within relational environments, was able to solve a 10-block BLOCKS WORLD problem by dividing the work amongst many agents. Each agent contained evolutionarily created and mutated rules which it applied to the problem by 'bidding' when it chose to apply them.

FOXCS (Mellor, 2008a,b) extends the XCS system ((Wilson, 1995, see Section 2.1.3) by learning first-order rules for acting in relational environments. Instead of the bit-string representation, FOXCS uses a first-order clause to match states. New rules are created by setting the rule conditions as a random generalised subset of the current state when no other rules match the

state. Better rules are found by performing GA mutations on existing rules by altering their conditions (adding/removing literals, generalising/specifying variable terms). A FOXCS policy is the total population of all rules, where the action output for a state is probabilistically selected from the set of all matching rules, using rule accuracy (inverse of prediction error) to bias selection.

2.4 Application to Game Environments

Games make ideal testbeds for reinforcement learning algorithms due to their obvious reward signal (win, lose, or a numerical score), simple rules, and complex gameplay. Games can be in many different formats, from board games such as CHESS, GO and BACKGAMMON, to classic video-games like TETRIS and PAC-MAN, to real-world (robotic) sports games like the ROBOCUP tasks.

Board Games

One of the earliest achievements of RL (and AI game-playing) was the CHECKERS playing algorithm by Samuel (1967), which uses a linear function-approximator to represent the expected return of states. Another famous approach is TD-GAMMON, a BACKGAMMON playing algorithm developed by Tesauro (1994). The algorithm used a three-layer neural network as a value-function approximator in combination with temporal difference learning to produce an advanced agent that played at the level of human experts, even learning a never-before-seen strategy. Sanner (2005) also uses the BACKGAMMON game as a testbed. Other 'classic' board games that have been the focus of RL algorithms are CHESS (Thrun, 1995; Baxter et al., 1998) and Go (Silver et al., 2007; Mayer, 2007). In most approaches, agent training was achieved by playing against itself.

Modern board games, otherwise known as *German-style board games* or *eurogames*, are also beginning to be used as testbeds for AI. Modern board games, compared to 'American-style' board games such as MONOPOLY[1] or THE GAME OF LIFE[2], focus more on strategic elements rather than luck, and typically keep all players in the game until it ends. Several learn-

[1] MONOPOLY was originally published by Parker Brothers in 1904.
[2] THE GAME OF LIFE was originally created by Milton Bradley in 1860.

ing algorithms have been developed for playing the resource-trading game SETTLERS OF CATAN[3] (Pfeiffer, 2004; Szita et al., 2009). The game CARCASSONNE[4] is a well-known game of tile placement with relatively simple rules. The variability of the tile-placements results in too many possible states for a brute-force propositional approach. So far, there only appears to be a single approach to developing an AI for CARCASSONNE (Heyden, 2009), though there are AI players available in commercial[5] and open-source implementations.[6]

Video-games

Classic video-games are video-games that typically have a 2D layout, simplistic graphics, basic but not necessarily easy gameplay, and were typically created in the 1980's or earlier. Example games and the learning algorithms applied to them are defined below:

- TETRIS[7]: policy iteration (Bertsekas and Tsitsiklis, 1996), the RRL methods developed by Driessens (2004) (RRL-TG, RRL-RIB, and RRL-KBR), genetic algorithm (Böhm et al., 2005), and policy-gradient cross-entropy method (Szita and Lörincz, 2006; Thiery and Scherrer, 2009).
- PAC-MAN and Ms. PAC-MAN[8]: rule-based evolutionary approach (Gallagher and Ryan, 2003), neural network (Lucas, 2005), rule-based cross-entropy method (Szita and Lörincz, 2007), and Monte-Carlo tree search (Ikehata and Ito, 2011).
- MARIO, based on the *Super Mario Bros.* game[9]: neuro-evolution (Togelius et al., 2009), cognitive architecture (Mohan and Laird, 2009), and grammatically evolved behaviour trees (Perez et al., 2011).

In all the previously mentioned games, the gameplay is simple enough for a child to grasp, but often proves to be difficult for an AI to learn effective behaviour for.

[3] SETTLERS OF CATAN was designed by Klaus Teuber and published by Franckh-Kosmos Verlag in 1995.
[4] CARCASSONNE was designed by Klaus-Jürgen Wrede and published by Rio Grande Games in 2000.
[5] Carcassonne for iOS: http://carcassonneapp.com/
[6] jCloisterZone: http://jcloisterzone.com/
[7] TETRIS was created by Alexey Pajitnov in 1984.
[8] PAC-MAN and Ms. PAC-MAN are trademark Namco Bandai Games.
[9] Super Mario Bros. was developed by Nintendo for the Nintendo Entertainment System.

In the past few years, learning algorithms have been applied to modern video-games as well. Some example testbed games include: first-person shooter UNREAL TOURNAMENT[10] (Jacobs et al., 2005; van Hoorn et al., 2009), 3D car racing games (Whiteson et al., 2005; Togelius and Lucas, 2006), role-playing game BALDUR'S GATE[11] (Szita et al., 2008), simulation game CIVILISATION II[12] (Branavan et al., 2011), and real-time strategy games (Guestrin et al., 2003; Ponsen et al., 2006; Sharma et al., 2007).

It is often the case that an algorithm interacts directly with the video-game software itself; states and actions are extracted through some 'wrapper' interface. In these cases, the (R)MDP is assumed to be implicit for the environment, though games are often POMDPs.

Other games

TICTACTOE is a commonly-used toy problem for minimax problems and has also been used to demonstrate behaviour in several RL problems, such as multi-agent learning and transfer learning (Boyan, 1992; Olson, 1993; Sutton and Barto, 1998; Ramon et al., 2007; Croonenborghs et al., 2008).

A grand goal for AI is to be advanced enough such that, when combined with robotics, it could fully control a functional soccer team and win against the best human team. This is the goal of the ROBOCUP competition. Although RL techniques (and the field of robotics) are not yet advanced enough to control a fully functional robotic soccer team, the 'keep-away' subtask of the ROBOCUP (Stone et al., 2005a) is an ideal multi-agent environment focusing on cooperation (Walker et al., 2004; Taylor and Stone, 2005; Stone et al., 2005b).

General Game Playing (GGP) (Genesereth et al., 2005) is a subfield of AI in which an agent is able to play any game, given the full specifications of the game in *Game Description Language*. This language is represented as a set of logical facts and rules, making it ideal for RRL algorithms. Because the environment model is known, planning or dynamic programming approaches would also work as well.

[10] UNREAL TOURNAMENT was developed by Epic Games and Digital Extremes in 1999.
[11] BALDUR'S GATE was developed by BioWare in 1998.
[12] CIVILISATION II was developed by Brian Reynolds, Douglas Caspian-Kaufman and Jeff Briggs in 1996.

2.5 Summary and Discussion

The above summary of various approaches towards solving RRL (and RL) problems is only an overview of the various existing algorithms but demonstrates that there are many different methods of solving RRL problems. This research aims to solve problems in which the environment model is not known, nor is it known if the environment model conforms to the MDP framework, so dynamic programming approaches are not likely to be useful. Static generalisations for environments are also unlikely to be utilised, as they typically require human intervention to define an effective generalisation and one of this goals of this research is to minimise human input to the learning process as much as possible.

The majority of algorithms for solving RRL or RL problems are value-based, but value-based methods have a common weakness: they are required to predict an expected value for *every* state (or in the case of Q-learning, every state-action combination). A naive method of achieving this is to repeatedly attempt to visit every state and perform every action until a close approximation to the true values is learned, but this is only practical in very small problems. Techniques for approximating the value-function (or Q-function) have been shown to be effective for a range of different approximation techniques, but the number of states still affects the learning rate and/or performance achieved. Using guidance or domain-specific techniques can simplify the learning problem, but this requires intervention from some external agent.

Policy-search learning methods attempt to learn the agent's policy directly, avoiding the need to record rewards received for every state. Value functions typically represent more than they need to and implicitly encode the 'distance' to the problem goal, making them vulnerable to non-(R)MDP or partially observable environments, whereas policy-search methods are less affected by such environments as they do not need to record values for individual states. Policy search as *classification* requires enough positive examples to be able to build an effective model, something which can be difficult to achieve in the beginning of learning without guidance. The LRW-API (Fern et al., 2006) algorithm was able to overcome this problem by learning policies for reduced versions of the problem, but it also required access to a world model that allows it to sample any state at any

time in order to learn approximate Q-values for actions. FOXCS's method of learning values for rules does encodes a value-function of sorts, and so is affected by the number of states (performance decreases with larger environments, Mellor (2008a)).

GREY and GAPI perform direct policy-search using Evolutionary Algorithm (EA) techniques where an agent's policy is represented as a decision-list of condition-action rules. This form of policy-search does not rely on human intervention and produces relatively comprehensible behaviour, satisfying two of the three goals of this research. A downside with these EA approaches is that they often require large populations of samples in order to learn effective solutions. One of the EA-style algorithms reviewed was the Cross-Entropy Method (CEM), which maintains a probabilistic distribution of solutions, rather than the harsher approaches of genetic algorithms in which solutions may be lost through random mutation. The CEM has been shown to be effective in RL environments, so this research will investigate the applicability of the CEM to learning decision-list style policies in RRL settings. Furthermore, (Szita and Lörincz, 2008) proposed two incremental alternatives to the standard population-based model of learning, providing a possible solution to the third research goal of learning behaviour quickly.

RRL algorithms have reached the point where learning an optimal policy for the standard environment BLOCKS WORLD with the OnG_0G_1 goal (place block X onto block Y) is relatively straightforward (Fern et al., 2006; Kersting and Driessens, 2008; Mellor, 2008a; van Otterlo and De Vuyst, 2009). Section 2.4 listed a number of games that learning algorithms have been applied to. Most of the games are relatively easy for humans to play (and usually somewhat harder to master, e.g. CHESS, GO, TETRIS), but AI techniques still struggle. Many of the algorithms only perform well on the games they were created for (with the obvious exception of GGP algorithms), but they do not necessarily generalise well to different games. This research aims to create an agent capable of playing a wide range of games, and so will be explicitly tested upon a range of game environments. The algorithm created will also be tested on BLOCKS WORLD to ensure it is comparable to existing techniques (and to easily demonstrate key concepts using a well-known environment).

3

Relationally Defined Environments

The goal of this research is to develop a learning agent capable of effectively learning human-comprehensible behaviour in a range of environments, large and small. In order to achieve such a goal, there needs to be a common representation for the environments, such that the same learning agent is able to interact with each one without drastically altering its method of learning. Relational representations are flexible enough to represent many different environments, as environmental state observations can be composed of any number of objects and relations using the predicates defined by the environment. The actions available to the agent can also change based on the objects and relations present in the state.

This chapter focuses on fully defining the structures used throughout the thesis. Before any high level definitions are explained, the terminology used throughout the rest of the thesis is defined, beginning with the specific syntax of the rule-engine used to define the relational states, to the various argument types and internal conditions seen within state facts and rules created by the agent.

The second part of this chapter (Section 3.2) defines the common relational structure in which the environments are specified. Each environment is required to have a set of observations, a set of actions, and a goal. Environments may also optionally specify rules that automatically generate observations and define the preconditions to generate actions, as well as other optional additions.

Finally, the four relational environments used to evaluate the algorithm's effectiveness are defined (Sections 3.3–3.6), explaining how each one interacts with the agent and what each environment contributes as an agent testing platform. Each of the environments must produce relational observations for the agent, accept relational actions that affect the state of the environment, and produce a reward based on the agent's behaviour. The four testing environments each present different challenges to the learning agent, which the agent must be able to handle if it is to be deemed a 'general learner.'

3.1 Terminology

Before explaining how the developed algorithm learns behaviour for acting in relational environments, the language used to describe the algorithm and environments throughout the remainder of the thesis must first be defined.

3.1.1 Syntax and Semantics

This research uses first-order relational rules to interact with a relationally defined environment, in which the state is described by a number of objects and relations. To understand the language used to convey this representation, some concepts must first be defined. The following definitions use standard ILP syntax (Lavrac and Dzeroski, 1993) with the addition of typed argument requirements:

- A *term* t may be either a *constant* or *variable*.
- A *constant* c is a string of characters, beginning with a lowercase letter, representing a uniquely named object (e.g. a, $blinky$, $cerrla$).
- A *number* is simply a numerical value (integer or floating point), interpreted as a number.
- A *variable* V is a string of characters, beginning with an uppercase letter, that serves as an abstract reference to a constant (e.g. X, N_3). When evaluated in a query, a variable binds to a single term using a *substitution map* $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ such that all occurrences of variable X_i are considered to be term t_i (written as $X_i\theta$). In this work,

variables of different names are implicitly defined to bind to different terms (except for the anonymous variable, see below).

- The *anonymous variable* '?' is a special variable that represents any object. It may bind to *any* term (including objects that other variables already bind to) but does not specify a substitution in θ.
- A *predicate* p is a string of characters, beginning with a lowercase letter, representing a named relation between one or more objects. Each predicate must define the number and *type* of arguments it accepts as a bracketed expression immediately following the predicate name with n capitalised *type* names. If a predicate requires a numerical argument, the type is written as $\#X$ where X may be any capitalised letter. E.g. $on(Block, Thing)$ defines the predicate on, which accepts a *block* as the first argument, and a *thing* as the second argument.
- A *type predicate* p_t is a special predicate for defining the type of an object. It has an arity of 1 (with no type required) and must be defined for every object (e.g. *thing*, *ghost*, *player* are all types that can be assigned to constants). Objects may be of multiple types (e.g. objects of type *block* are also of type *thing*, but not necessarily vice-versa). Numbers do not need to define their type.
- An *atom* $p(t_1, \ldots, t_n)$ is a predicate with terms for arguments. Each term must be of the appropriate type defined by the predicate. E.g. $on(X, c)$ is an atom with variable X representing a *block* and constant c (which must of type *thing*).
- A *ground atom* $p(c_1, \ldots, c_n)$ only uses constants for arguments. Each constant must be of the appropriate type defined by the predicate.
- A *fact* is a ground atom that is considered true for the current state of the system. State observations define the conjunction of facts which are true for the current environment state. E.g. *block(a)*, *block(b)*, *on(a, b)* are three facts describing the truth of the state.
- A *literal* L is an atom or its negation. A negated atom is defined by prefixing the word *not* to the atom, e.g. *not on(a, b)* states that the relation *on(a, b)* is not true.
- A *clause* C is a disjunction of literals $\forall X_1, \ldots, \forall X_n(L_1, \ldots, L_m)$ where

each L_i is a literal and each X_i is a variable occurring in one or more of the literals. A clause can be written in the form $L_1, \ldots, L_n \rightarrow L_{n+1}, \ldots, L_m$ where the commas on the Left-Hand Side (LHS) represents a conjunction and the commas on the Right-Hand Side (RHS) represent a disjunction. Typically there is only a single non-negated literal on the RHS.

- A clause C_i *θ-subsumes* another clause C_j if there exists one or more substitutions for the variables in C_i such that $C_i\theta \subseteq C_j$.
- A *rule* $r = L_1, \ldots, L_n \rightarrow T$ is a combination of literals such that the LHS of the rule is a conjunction of literals representing a pattern to match, and the RHS of the rule is a conjunction of one or more non-negated atoms (T). If T contains variables, they must also be used in the LHS of the rule. A substitution map is applied to a rule ($r\theta$) by applying the substitution θ to every literal in r.

  Evaluating a rule against a set of atoms s involves checking if the conditions of the rule θ-subsume the atoms in s. If so, the RHS atom with the substitution(s) applied is output from the rule as the result of the evaluation. The application of the output atom(s) depends on the context in which the rule is evaluated.

As in ILP, all constants and predicates are interpreted with a *Herbrand interpretation*; that is, every constant is interpreted as itself, and every predicate is interpreted as the predicate that applies it.

Environment Representation

An environment is defined by three sets of predicates:

- *State predicates* $P_s = \{p_{s,1}, \ldots, p_{s,n}\}$ define what relations are used to describe the state observations.
- *Type predicates* $P_t = \{p_{t,1}, \ldots, p_{t,n}\}$ are a subset of state predicates and define the types of every object described in the current state.
- *Action predicates* $P_a = \{p_{a,1}, \ldots, p_{a,n}\}$ define the action predicates an agent can use to interact with the environment.

An environment state observation consists of a set of facts composed of state predicates and the current objects of the state describing the current

state of the environment $s = \{p_{s,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{s,m}(c_{m,1}, \ldots, c_{m,n})\}$. The facts of the current state observation are *asserted* to memory, such that they are considered true for the current state. The set of valid actions for the state consist of a set of facts composed of action predicates and objects of the state $A(s) = \{p_{a,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{a,m}(c_{m,1}, \ldots, c_{m,n})\}$. If the environment's current goal specifies any constants, these are defined by the goal variable substitution map $\theta_G = \{G_i/c_i\}$, where G_i is an index-dependent reference to the i^{th} constant in the goal. When it is evaluated, it is substituted by constant c_i. The use of goal variables instead of direct constants allows the learned behaviour to be parameterisable to any combination of constants in the goal.

Further information can be added to the state of the environment by using *background knowledge* to infer new knowledge from existing knowledge. Background knowledge is defined as a rule such that when the LHS of the rule is true with respect to the current state description s, the RHS is also true and the substituted atom is added to s. E.g. *highest(X)* $\rightarrow$ *clear(X)* states that whenever an object is *highest*, it is also *clear*.

Details of how each environment transitions from state to state and how the set of valid actions is created is described in Section 3.2.

Policy Representation

The policies learned by CERRLA are defined as a decision list of condition-action rules $r = p_{s,1}(t_{1,1}, \ldots, t_{1,n}), \ldots, p_{s,m}(t_{m,1}, \ldots, t_{m,n} \rightarrow p_{a,i}(t_{i,1}, \ldots, t_{i,n})$, where the policy is evaluated from top-to-bottom. Each rule is evaluated against the current state observations s and the outputs of each rule are returned to the environment in the same order as the rule's position in the policy. Each rule in a policy is evaluated independently of other policy rules (i.e. variable substitutions are not shared between rules). How the actions are resolved by the environment is dependent on the environment (see Section 3.2).

E.g. the rule *clear(X), highest(Y)* $\rightarrow$ *move(X, Y)* defines a pattern in which one object must be *clear* and another must be *highest*. If such a pattern is found in the current state, the substitution(s) are applied to the *move* action to produce one or more ground atom *move* actions.

The arguments of every atom in the rule consist of either constants, num-

bers, variables, anonymous variables, or *range variables*. Range variables are specially named variables written as N_i, where i is an identifier for the particular range. Alone, a range variable binds to any number, but if constrained by dynamic range variables $(Low_i \leq N_i \leq High_i)$, it must be between the Low_i and $High_i$ bounds (inclusive). The bounds are defined as variable values because they may be subject to change. For more information about dynamic ranges, refer to Section 5.5.2.

3.1.2 JESS Rule Engine

This research uses the Java Expert System Shell (Jess) first-order rules engine to facilitate the relational aspect of the environment and algorithm. Jess was developed by Ernest-Friedman Hill at the Sandia National Laboratories as a Java-based rule engine (Hill, 2003). Jess' syntax is a superset of the CLIPS[1] programming language, where statements (atoms, rules, etc.) consist of nested brackets containing function names and various symbols (e.g. `(block a)`, `(defrule (clear ?X) (highest ?Y) => (assert (move ?X ?Y)))`). This thesis will use Prolog syntax instead of Jess's native syntax for readability, but the semantic intent is the same.

Rete Algorithm

Jess makes use of the Rete algorithm to store facts and efficiently evaluate rules and queries. The Rete algorithm is an efficient pattern matching algorithm originally designed by Forgy (1982). The algorithm emphasises speed at the cost of memory by pre-computing partial matches for each of the facts contained within the structure, such that evaluating pattern-matches is just a matter of joining partial results.

The algorithm creates a network of interconnected *nodes*, where each node takes input, processes the input and, if the node's test is successful, produces output to pass down the network. If a set of facts filters all the way to the bottom of the network, those facts are considered a match for the pattern (e.g. rule or query) being evaluated. The Rete algorithm is efficient because each node also has a memory of the outputs, such that query results are pre-computed when the network is constructed and new information or structure can be quickly processed. The network consists of

[1] **C** Language **I**ntegrated **P**roduction **S**ystem

several node types: pattern-matching nodes, join nodes, and result nodes.

Pattern-matching nodes consist of a single test condition (such as matching the structure of a fact) that tests each input fact received. If the test is successful, the fact is passed on as output, otherwise it does nothing. The testing conditions consist of the conditions present in the rules being evaluated by the system. E.g. *clear(X)*, *above(Y, ?)* $\rightarrow$ *move(X)*Y creates two pattern-matching nodes for the two LHS conditions in the rule.

Join nodes combine the outputs of other nodes into a single set of results using joining tests (e.g. equal variable substitutions). Join nodes define the structure of the network by defining the connections between pattern-matching nodes. Join nodes also define special relationships between nodes such as negated atoms or disjunctions. Because there may be multiple joins between the inputs, join nodes remember all facts that they receive from each input (*left* and *right memory*). By using this memory, the algorithm is able to quickly evaluate rules, as each fact need only be tested once per node.

Result nodes simply output the set of valid substitutions for a rule's conditions such that the rule conditions θ-subsume the set of currently true facts. These substitutions are computed from previous joins and pattern-matching nodes higher up in the network.

The Rete network is also compact, as it reuses results for identical conditions or patterns in separate rules. For example, if we had two rules: A, B, C and A, B, the network can be compressed by adding an extra output to the A, B join node (one to a join node with the C pattern-matching node and one to a result node for the second rule).

3.2 Environment Specification Language

The environments used in this thesis each use a common specification to define how the environment is represented, and how it is interacted with. How a problem is presented to a learning agent (known as the *language bias*) is very important, as the agent's behaviour, no matter how advanced, is restricted by the environment representation. Furthermore, the representation of the problem can also effect how efficient the environment is at representing the state and processing the agent's actions. As many of

these environments are new to RRL, the choice of environment representation was selected in such a way that a learning algorithm should be able to create effective behaviour using it.

As defined in Section Section 3.1.1, each environment defines *state*, *type*, and *action* predicates to represent the state of the environment and the actions an agent can take within it. *Background knowledge rules* can be used to automatically create new knowledge to represent the state, and *action rules* can be defined which automatically identify the set of valid actions available to an agent. Each environment includes a *transition function* which either defines formal rules that alter the logical state representation of the environment, or describes how each action acts upon the environment outside of the logical state representation. An environment also defines a *reward function* which describes how reward is allocated to the agent. Other environmental details include: *constant facts*, *maximum number of steps per episode*, and *goal states*.

State Predicates State predicates define the language in which an environmental state is described. Each state predicate is defined by predicate name, number of arguments, and type of arguments. When a set of facts representing the environment state is provided to the agent as observations on the current state, the facts all consist of either state or type predicates (with constant or number arguments).

Some state predicates are for environmental use only and are not visible to the agent. I.e. these predicates are only to be used with background knowledge to infer new information. These predicates are marked with the symbol †.

Example state predicates include: *on(Block, Thing)*, *flying(Enemy)*, *height(Thing, #H)*.

Example state facts are: *on(a, fl)*, *flying(goomba_42)*, *height(c, 2)*.

Type Predicates Each object in the environment is defined by one or more *type facts* (if an object has multiple types, each type must not conflict with the other types). Type predicates are single-argument predicates used to bind objects to typed-groups to assist the learning agent by restricting the types of objects that can be present in relation and action facts (known as *declarative bias*). A type predicate is explicitly defined

as a type predicate in the environment specification.

Types can also be hierarchically arranged by defining an immediate parent type, such that a single object may be of multiple broader types. Hierarchical relations between types are written as $a \leftarrow b; c; d; \ldots$ (where each letter represents a different type predicate) such that any object of type b, c, or d is also of type a. Formally this is defined as background knowledge $b(X) \rightarrow a(X)$.

Example type predicates are: *thing*, *block*, *ghost*

Example type facts are: *thing(mario)*, *block(a)*, *ghost(blinky)*

Action Predicates Action predicates define the actions the agent can use to interact with the environment. Each state predicate is defined by predicate name, number of arguments, and type of arguments. For every state in an environment, the set of *valid actions* (action facts) is calculated using the *action rules* (see below).

Example action predicates are: *move(Block, Thing)*, *moveTo(Thing, #D)*, *placeTile(Player, Tile, Location, Orientation)*

Example action facts are: *move(a, b)*, *moveTo(dot_12, 16)*, *placeTile(cerrla, tile_8, loc_-1_0, r90)*

Background Rules Background rules use existing facts in the state to derive new facts, such that facts do not need to be manually asserted to the state. When the facts of a state match the LHS of a background knowledge rule, the fact on the RHS is asserted to the state. Note that if the state changes such that the LHS of the rule is no longer true, the RHS fact is retracted from the state. Background rules are optional and may not be required by every environment.

Example background rules are: *on(X, Y)* $\rightarrow$ *above(X, B)* (whenever X is *on* Y, X is also *above* Y), *on(X, Y)*, *above(Y, Z)* $\rightarrow$ *above(X, Z)* (whenever X is *on* Y and Y is *above* Z, X is *above* Z).

Action Rules Each action predicate has one or more associated action rules which define the preconditions necessary for grounded actions to be available within the current state as valid actions. When the facts of a state are asserted, the action rules are evaluated against the current state facts. If the rule produces one or more outputs, each produced

action is included in the set of valid actions for the current state.

An example action rule is: *clear(X)*, *block(X)*, *clear(Y)*, *not on(X, Y)* $\rightarrow$ *move(X, Y)* which states that if *block X* is *clear*, and *Y* is also *clear* and *not* under *X*, then the action to *move X* to *Y* is a valid action.

Transition Rules In some environments, transition rules can be set up which automatically define the effects of actions selected by the agent. Transition rules allow the state to automatically be modified when an action is taken, rather than reasserting all the facts of the environment at every step. Transition rules make use of two special operators *assert* and *retract* which each take one fact as an argument to assert/retract the fact to/from the state observations respectively. In other environments, defining formal transition rules for modifying the state observations is not practical (e.g. when the state observations are observations of an external model of the environment). In that case, the environment specification informally describes how each action affects the state.

An example transition rule is: *move(X, Y)*, *on(X, Z)* $\rightarrow$ *assert(on(X, Y))*, *retract(move(X, Y))*, *retract(on(X, Z))* which moves *block X* onto *Z*, removing the *move* action and the *on(X, Y)* facts while asserting *on(X, Z)*.

Reward Function Each environment must provide a reward R to the agent through a reward function. This function does not need to be defined as a JESS rule, but should be consistent: better performance within the environment should consistently receive a greater reward than poor performance.

Constant Facts An environment may define a collection of constant facts that are true in all states. These are formally defined by background knowledge that is always true, i.e. has no facts on the LHS.

Max Episode Steps Each episode of an environment may be bounded by a maximum number of steps.

Goal States Goal states define the state in which the goal is achieved. For every state of an episode, if the goal state θ-subsumes the current facts of the state, the goal has been achieved. The environment may also define an informal goal state which is not logically defined, but still exists in the internal model of the environment.

An example goal state is: $\forall X$ *block(X) clear(X)*

3.2.1 State Description

The learning agent does not have access to the full environmental model. At every state, it receives the following information only:

State Observations: represent the current state of the environment. Each state is composed of both type and relation facts, defined by the environment specification.

Valid Actions: define the set of all valid actions that can be taken from this state (where the action predicates are provided by the state specification).

Reward: apart from the first state of the episode, the agent always receives a reward value, based on the environment's reward function. This reward could be as a result of the previous action, or a delayed reward as a result of many actions.

Goal Substitution Map: defines the constants used in the episode's current goal in the form of a goal substitution map $\theta_G = \{G_i/const\}$. If there are no constants in the episode goal, no goal substitution map is provided.

Terminal Flag: indicates if the current state is a terminal state and the episode has ended. This is to alert the agent that the episode is complete and provide the agent with the final reward.

Current Agent: in multi-agent environments (e.g. CARCASSONNE), there may be multiple agent-behaviours controlled by a single learning agent. The *current agent* parameter allows the learning agent to determine which behaviour to use for the current agent.

3.3 Blocks World

BLOCKS WORLD is perhaps the most famous environment in relational planning and learning (Nilsson, 1980; Slaney and Thiébaux, 2001; Russell and Norvig, 2003). Although it has little practical application (except, perhaps for shifting containers at a port, for example), it provides a simple environment for demonstrating many of the core problems faced by relational

reinforcement learners. BLOCKS WORLD consists of a fixed number of blocks and a floor large enough to hold them all. Each block is assigned a unique identifier ($a, b, c, \ldots$). The blocks may be stacked on top of one another or placed on the floor, but only one block may be moved at a time. BLOCKS WORLD goals are defined as various specified configurations of the blocks.

BLOCKS WORLD is probably the least complex environment that the agent is tested on (measuring complexity by number of predicates and possible constants), but possibly the most common environment used within the RRL field. It introduces the problems of randomised starting states, specific object-based goals, and success-based rewards (non-minimal rewards are only received when the goal is achieved). Despite its simplicity, finding an optimal BLOCKS WORLD solution has been shown to be NP-hard (Gupta and Nau, 1992).

Slaney and Thiébaux (2001) presents a formula for calculating the number of states for worlds with n blocks as:

$$g(n,k) \leftarrow \sum_{i=0}^{n} \binom{n}{i} \frac{(n+k-1)!}{(i+k-1)!} \tag{3.1}$$

where k is the number of grounded[2] towers of blocks ($k = 0$ when calculating all possible state configurations). The number of actions is between 1 and $n^2 - n$ (Mellor, 2008a). In a 10-block BLOCKS WORLD environment, there are just under 59 million unique states and between 1–90 actions to take per state, so representing the environment propositionally would be computationally infeasible.

3.3.1 Episodic Description

Each episode of BLOCKS WORLD starts with a randomly generated state of n blocks, with the constraint that the active goal is currently not true. The initial states are generated using the algorithm defined in Slaney and Thiébaux (2001):

1. Start with an empty floor and n ungrounded towers each consisting of a single block.
2. Repeat until all towers are grounded:

[2] Have been placed on the floor.

a) Arbitrarily select one of the ϕ yet ungrounded towers.

b) Select the floor with probability $g(\phi - 1, \tau + 1)/g(\phi, \tau)$ (τ is the number of grounded towers) or one of the other towers (grounded or not) each with probability $g(\phi - 1, \tau)/g(\phi, \tau)$, and place the selected ungrounded tower on to it.

The agent is required to move blocks around until either the goal is met, or the maximum number of steps are reached.

3.3.2 Specification

State Predicates See Table 3.1 for the state predicates. When a BLOCKS WORLD state is observed, only the *on* and *block* facts are asserted as true; all other facts can be inferred from these facts using the *background rules*.

Type Predicates Table 3.1 defines the type predicates and their hierarchy. Each hierarchical rule is added to the *background knowledge*.

Action Predicates See Table 3.1 for the action predicates.

Background Rules The following background rules are used to assert the remainder of the object relations to the state:

block(Y), not on(?, Y) $\rightarrow$ *clear(Y)*

on(X, Y) $\rightarrow$ *above(X, Y)*

on(X, Y), above(Y, Z) $\rightarrow$ *above(X, Z)*

Table 3.1: Predicate definitions for Blocks World.

State Predicates	
on(Block, Thing)	▷ *Block* is directly on *Thing*
above(Block, Thing)	▷ *Block* is somewhere above *Thing*
clear(Thing)	▷ *Thing* can have *Blocks* placed upon it
highest(Block)	▷ *Block* is (one of) the highest in the state
†*height(Thing, #H)*	▷ The height #H of *Thing*. Not visible to agent.
Action Predicates	
move(Block, Thing)	▷ Move a *Block* on to *Thing*
Type Hierarchy	
thing $\leftarrow$ *block; floor*	▷ *block* and $floor$ are *things*

on(X, Y), height(Y, N) $\rightarrow$ *height(X, (N + 1))*

height(X, N_N*),* $\forall Y$ *(thing(Y), height(Y,* $(N_M \leq N_N)$*))* $\rightarrow$ *highest(X)*

Action Rules The *move action rule* defines the valid actions that can be taken in each state:

clear(X), block(X), clear(Y), not on(X, Y) $\rightarrow$ *move(X, Y)*

Transition Rules Actions are resolved using the *move transition rule*: *move(X, Y), on(X, Z)* $\rightarrow$ *assert(on(X, Y)), retract(move(X, Y)), retract(on(X, Z))*

If no action is selected in a state, the episode ends with reward 0.

Reward Function When an episode is complete, either when the goal is achieved or the maximum number of steps are taken, the agent receives a reward R of:

$$R \leftarrow 1 - \frac{t - o}{M - o}$$

where t is the number of steps taken, o is the optimal (minimum) number of steps needed to achieve the goal, and M is the maximum number of steps allowed. Because Blocks World is a simple environment, the optimal policy can be manually defined to precompute the minimum number of steps required to complete the goal (the agent does not have access to this policy or its execution). This reward function is used in place of a simple -1 per step function because the number of steps to a goal varies depending on the initial state of the episode (which would result in large negative rewards for unlucky, but otherwise optimal policies).

Constant Facts The constant facts that are always true in every Blocks World goal or size are all constant facts relating to the floor: *floor(fl), clear(fl), height(fl, 0)*.

Max Episode Steps The maximum number of steps allowed per episode is: $M \leftarrow 2n$, where n is the number of blocks in the environment. This is sufficient for all goals evaluated.

3.3.3 Goals

The Blocks World environment has four primary goals:

Stack Place all the blocks into a single tower, such that only one block is

on the floor and the rest are above that block. As the blocks may be in any order, there are multiple states that satisfy this goal. Formally, this goal is defined as:

$$on(X, fl), not\ on(Y, fl)$$

Unstack Place all blocks on the floor (so every block is clear). There is only one state that satisfies this goal. Formally, this goal is defined as:

$$\forall X\ block(X), on(X, fl)$$

ClearG_0 Clear a single block G_0, where the replacement for G_0 is a block constant that is not already clear. This goal is also known as *ClearA* in other work (G_0 is used to emphasise the use of parameterisable goal variable). G_0 changes every episode to a non-clear block. Formally, this goal is defined as:

$$clear(G_0), block(G_0)$$

OnG_0G_1 Place block G_0 on block G_1, where the replacements for G_0 and G_1 are two different blocks and G_0 is not already on G_1. This goal is also known as *OnAB* in other work (G_0G_1 was used for clarity). Formally, this goal is defined as:

$$on(G_0, G_1), block(G_0), block(G_1)$$

While there are many other possible goals, these four goals are standard goals in RRL experiments, and will be used for experiments. The size of the Blocks World environment only changes the number of steps required to achieve each goal; the definition of the goals remain unchanged. At the start of an episode, if the goal is already achieved, the environment is re-initialised with a new state until a state is created where the goal is not achieved.

3.4 Ms. Pac-Man

Ms. Pac-Man is the (unauthorised) sequel to the famous Pac-Man arcade video game[3] (see Figure 3.1 for an example screenshot). The goal of the game is for Ms. Pac-Man (the agent) to achieve a high score by eating

[3]Pac-Man and Ms. Pac-Man are trademark Namco Bandai Games.

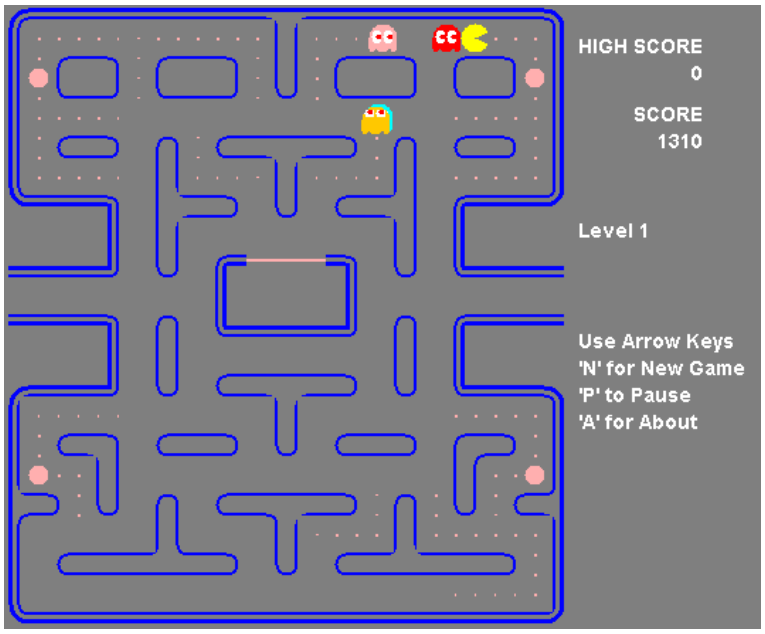


Figure 3.1: A screenshot of a portion of the MS. PAC-MAN environment.

dots within the level, avoiding hostile *ghosts*, and eating edible *ghosts*. The Ms. PAC-MAN environment was originally built from an open-source implementation of Pac-Man.[4]

Ms. Pac-Man has four simple directional actions, though these are abstracted into higher level actions for the purpose of learning strategic behaviour rather than learning how to effectively move about the maze. While a low-level representation is possible with a relational representation, the main problem in the environment would be learning how to navigate the maze, rather than learning effective strategies for achieving the highest score.

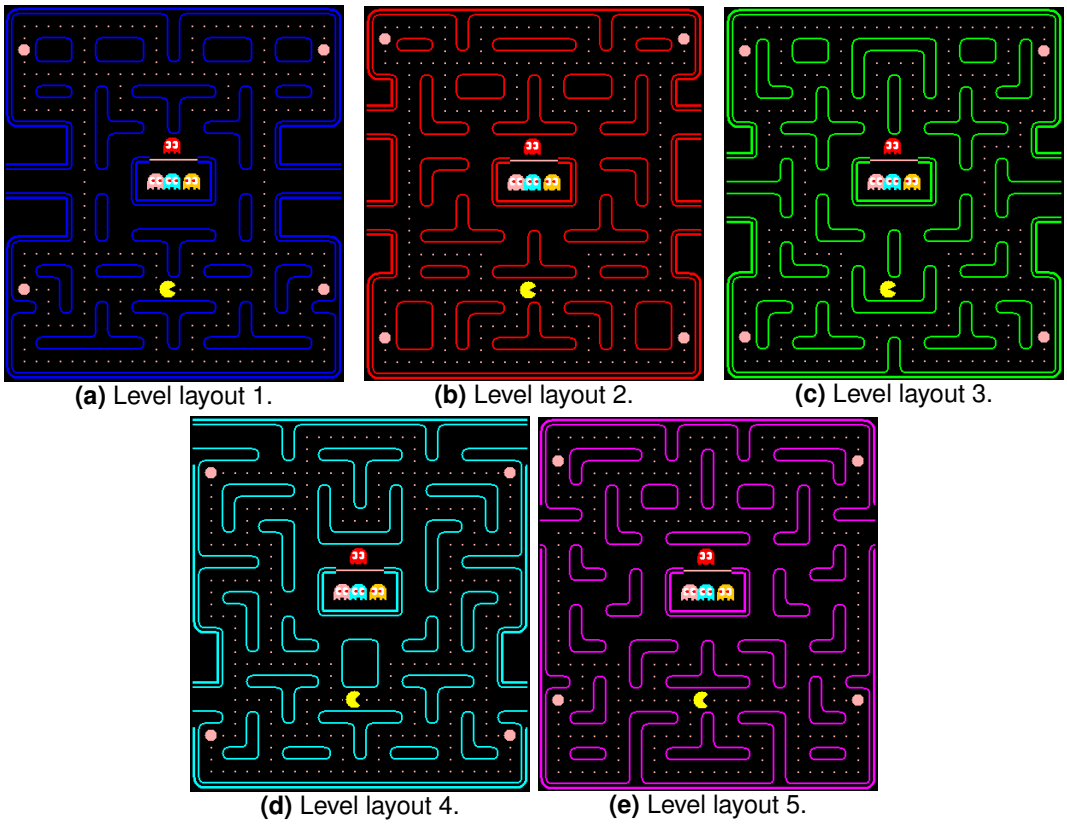
There are four hostile ghosts, each with individual behaviour,[5] periodically released from their cage, which move at the same speed as Ms. Pac-Man. If a hostile ghost touches Ms. Pac-Man, the agent loses a life and both the ghosts and Ms. Pac-Man return to their starting positions. Unlike the original Pac-Man game, the ghosts in Ms. PAC-MAN have a 25% chance of choosing a non-default behaviour direction at a junction (but cannot turn directly back) so a level cannot be completed by taking a predefined sequence of actions.

[4]Originally found at http://www.bennychow.com
[5]The Pac-Man Dossier (Pittman, 2011) defines each ghost's behaviour.

When Ms. Pac-Man eats a *powerdot*, the ghosts become edible, move at 60% speed, and move in the opposite direction to their normal behaviour (away from Ms. Pac-Man) for a limited time (the time the ghosts remain edible decreases per level, see The Pac-Man Dossier (Pittman, 2011, for exact time). While edible, the ghosts can be eaten by Ms. Pac-Man for an increasing score bonus for every ghost eaten (see Section 3.4.2 for point value). When a ghost is eaten, it returns to the cage and is released again (as a hostile ghost again) after a short time.

Ms. Pac-Man works well as a RRL problem because:

- The state is fully observable with clear objects and relations.
- The reward signal (the score) is obvious and relates directly to the game.
- Ghosts introduce hostile agents, which actively attempt to limit the learning agent's performance. Furthermore the ghost's behaviour is non-deterministic, making rigid planned behaviour ineffective.
- It is the first environment to actively require numerical specialisations for the actions, testing how well the agent deals with numerical values.
- The agent's low-level directional movement is calculated from multiple high-level actions, where the agent may follow one or more actions simultaneously (e.g. eat dots while avoiding hostile ghosts). Multiple actions are used to break ties when a rule produces multiple facts that result in conflicting low-level movement. Szita and Lörincz (2007) present results demonstrating that following multiple actions simultaneously achieves better performance than restricting the agent to a single action.
- Viglietta (2012) proves Pac-Man is NP-hard (to complete a single level without losing a life). As Ms. Pac-Man is strongly based on Pac-Man (probably more complex than Pac-Man), the Ms. Pac-Man problem is also NP-hard.



(a) Level layout 1. **(b)** Level layout 2. **(c)** Level layout 3.

(d) Level layout 4. **(e)** Level layout 5.

Figure 3.2: Initial level layouts for the MS. PAC-MAN environment. Each layout is used for two levels.

3.4.1 Episodic Description

The agent begins each episode on level one, with three lives available to lose. When the agent successfully eats all *dots* in the maze, the next level is loaded, the ghosts are reset to their starting positions, and the agent is reset to its starting position. Ms. Pac-Man receives a bonus life when it reaches 10,000 points. Every two levels the layout of the maze changes and the ghosts in the level become faster and the time in which they are edible is decreased. If the agent loses all of its lives, the episode ends. The five level layouts are shown in Figure 3.2

3.4.2 Specification

State Predicates See Table 3.2 for the state predicates. Because there are multiple non-stationary objects in Ms. PAC-MAN, all facts are re-asserted for every state instead of retracting and asserting new information.

The *distance(Thing, #D)* predicate is defined as the length of the short-

Table 3.2: Predicate definitions for MS. PAC-MAN.

State Predicates	
distance(Thing, #D)	▷ *Thing* is #*D* units from Ms. PAC-MAN
junctionSafety(Junction, #J)	▷ *Junction* has safety value #*J*
edible(Ghost)	▷ *Ghost* is edible
blinking(Ghost)	▷ *Ghost* is blinking
Action Predicates — where #*D* (*distance*) and #*J* (*junctionSafety*) are meta-information for resolving actions.	
moveTo(Thing, #D)	▷ Move towards *Thing*
moveFrom(Thing, #D)	▷ Move away from *Thing*
toJunction(Junction, #J)	▷ Move towards *Junction*
Type Hierarchy	
thing ← *ghost; dot; powerdot; ghostCentre*	▷ All objects are *things*
junction	▷ An intersection of paths

est path between Ms. Pac-Man and *thing*. The *junctionSafety(Junction, #J)* predicate is defined as the shortest distance between the *junction* and nearest *ghost* minus the distance between the *junction* and Ms. Pac-Man. E.g. a *junctionSafety(junc_10_12, 4)* implies Ms. Pac-Man can reach *junc_10_12* four steps prior to the nearest ghost.

edible and *blinking* are both qualities the ghosts can have after Ms. Pac-Man eats a powerdot.

Type Predicates Table 3.2 defines the type predicates and their hierarchy. Each hierarchical rule is added to the *background knowledge*.

Action Predicates See Table 3.2 for the action predicates. The numerical argument within each action is used as meta-information for resolving Ms. Pac-Man's movement.

Background Rules Ms. PAC-MAN does not define any background rules.

Action Rules The rules for producing the valid actions the agent can take are as follows:

$$distance(X, N_D) \rightarrow moveTo(X, N_D)$$

$$distance(X, N_D) \rightarrow moveFrom(X, N_D)$$

$$junctionSafety(X, N_J) \rightarrow toJunction(X, N_J)$$

where each numerical N_i is used as meta-data for resolving the action

into low-level movement.

Transition Rules Ms. PAC-MAN does not specify any formal JESS-syntax transition rules, but Ms. Pac-Man's movement is determined by iteratively resolving the actions produced by the policy until a single low-level direction is determined. Ms. Pac-Man can always move in two or more directions and the action resolution process selects a single direction to move in.

For each set of action facts produced by the next rule in the policy (evaluated first-last), the objects that are closest (#*D*) or have the highest junction safety (#J) determine the direction(s) (towards or from) selected. If moving *towards* an object, the directions of the shortest path to the objects are used. If Ms. Pac-Man is moving *from* an object, the movement is resolved by removing the shortest path direction as a possible direction to move in. If there are multiple directions remaining, the next closest/highest junction actions are used to determine direction. If there are no actions remaining in the current set of actions, the next rule's set of actions are used to resolve movement. If no single direction is selected, either take the same direction as last step, or if not possible, select a perpendicular direction.

Reward Function The reward function is simply the agent score throughout the episode. The score is calculated as: 10 points per dot, 50 points per powerdot, 200, 400, 800, 1600 points respectively for each consecutive ghost eaten while ghosts are edible. The maximum score that can be achieved in level one of Ms. PAC-MAN is 15,370 points, though this is very difficult to achieve.

Constant Facts There are no constant facts in Ms. PAC-MAN. While the ghosts are present for the majority of the game, they are not observed when they are 'locked-up' in their cage at the start of a level.

Max Episode Steps There is no bound to the number of steps the agent may take per episode.

3.4.3 Goals

While there is only one general goal in Ms. PAC-MAN (maximise score by eating dots and ghosts), there are three fundamentally different experimen-

tal setups that alter how this goal is achieved:

Single Level This goal focuses on reward maximisation within the first level. This goal biases the agent's behaviour towards more reckless behaviour, as it has three lives for a single level. Formally, the goal state is: *level(2)*.

Ten Levels This is essentially the normal setup of the environment. The agent attempts to maximise score over ten levels of Ms. Pac-Man. Typically, the episode ends when the agent loses all lives, rather than completing level ten (which is very difficult, due to increasing ghost difficulty). Formally, the goal state is simply: *level(11)*.

Single Life This goal focuses on survivability and learning cautious behaviour. The agent only has a single life, but may earn an extra life at 10,000 points. Though unlikely to be achieved, the goal is the same as *Ten Levels*, and so has the same formal goal state (*level(11)*).

3.5 Mario

The Mario environment is a clone of the Super Mario Bros.[6] video-game (see Figure 3.3 for an example screenshot). It uses a modification of the Infinite Mario game,[7] an open-source clone of the original game as the envi-

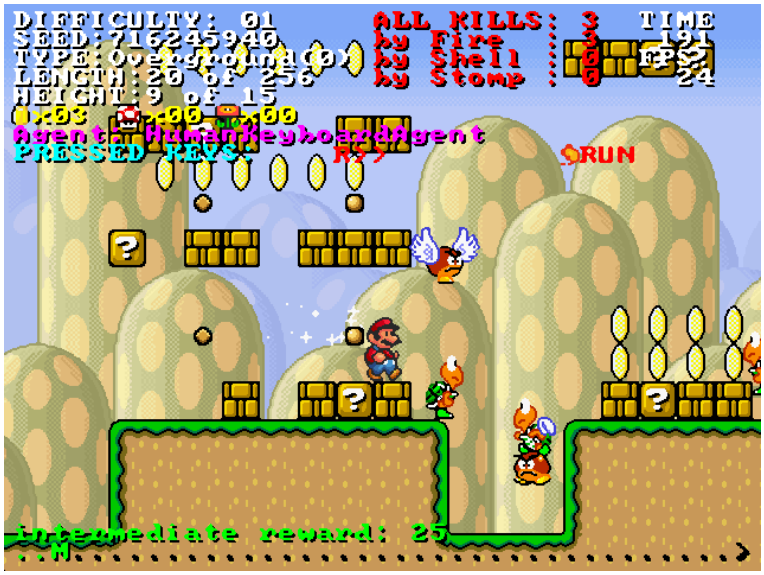


Figure 3.3: A screenshot of the Mario environment.

[6]Super Mario Bros. was developed by Nintendo for the Nintendo Entertainment System.
[7]Can be played at http://www.mojang.com/notch/mario/

ronment implementation. Infinite Mario was modified further to facilitate AI agent interfacing with the environment for the Mario AI competition.[8]

The agent is in control of Mario, who must traverse a fixed-length two-dimensional level of hazards in an attempt to reach the goal within a finite time period. Mario can move $left$, $right$, $jump$, and run. Mario can also shoot fireballs (while in *fire* form), and pickup and shoot *koopa* shells. Each level is randomly-generated (but fixed length), constrained by difficulty parameters, and completable, so the agent must be able to learn how to handle many different situations. Each level consists of terrain of varying heights (including deadly pits), a number of different types of enemies, interactive bricks, and collectable coins and powerups.

Mario has three forms (in decreasing order): *fire*, *large* and *small*. When in *fire* and *large* form, Mario's height is two units (32 pixels), while in *small* form, he is only one unit tall (16 pixels). Whenever Mario is hit by an enemy, his form decreases. Mario can increase his form by 'searching' a *box* (by hitting it from underneath with his head), and collecting either a *mushroom* (increase to *large*) or a *fireFlower* (increase to *fire*) that may come out the top of the box.

Mario can dispatch enemies by jumping on to them (except for *spiky* and *piranhaPlant*) or, if in fire mode, shooting a fireball (except for *spiky*). If an enemy has wings (is *flying*), jumping on them only removes the wings. When a *koopa* is jumped on, it leaves behind a *shell* that can be picked up and used as a one-hit shield or released as a bouncing projectile to destroy *enemies*, *bricks* and damage Mario himself. The only difference between *redKoopas* and *greenKoopas* (other than colour), is that *redKoopas* will turn around if they reach a cliff edge.

MARIO works well as an RRL problem because:

- The environment is made up of objects that interact in different ways.
- Each level is randomly generated to a set of constraints, such that the agent must learn flexible behaviour for facing a multitude of different scenarios.
- The agent is required to advance towards the goal while navigating

[8]Version 0.1.9. Download the source from http://www.marioai.org/

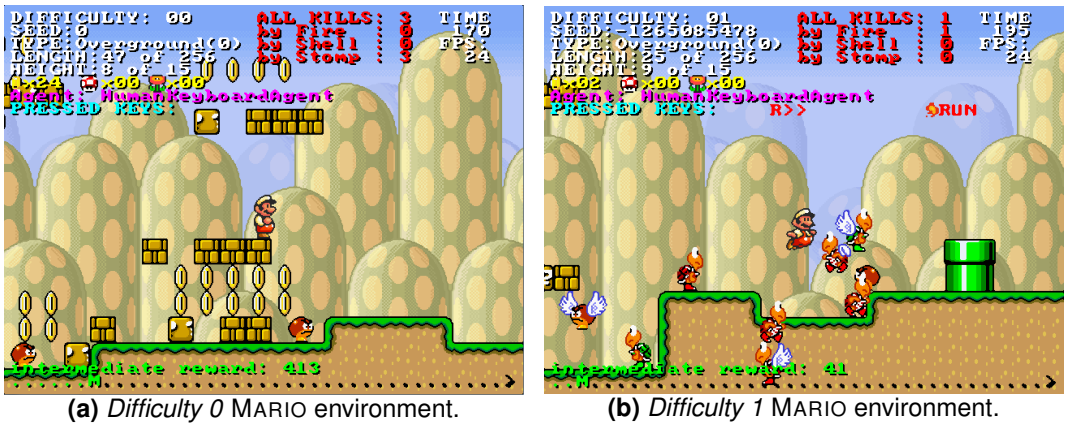
the immediate challenges of level geometry and enemies.

- Aloupis et al. (2012) proves the original Super Mario Bros. is NP-hard. Although Infinite Mario does not have the same game mechanics as the original Super Mario Bros., it is still of a high complexity (and may still be NP-hard).

3.5.1 Episodic Description

Regardless of the complexity of the randomly-generated level, each episode of MARIO begins with Mario starting the level on the far-left side of the level in *fire* form, falling onto a flat stretch of ground (usually with a group of enemies advancing towards him). In order to complete the level, Mario must reach the far-right side of the level. Each level is of length 256 units (where Mario represents 1 unit, and each unit is 16×16 pixels; the granularity of the distances measured in the state observations). If Mario is hit by an enemy while in *small* form, the episode ends. The episode is also ended if the agent fails to select an action for 30 steps (the agent's behaviour is considered inoperative). Examples of the different level difficulties are shown in Figure 3.4.

The details of the level-generation code are outside the scope of this thesis, but the source code can be found at http://www.marioai.org/gameplay-track/getting-started.



(a) *Difficulty 0* MARIO environment. **(b)** *Difficulty 1* MARIO environment.

Figure 3.4: Example screenshots of the two MARIO difficulties.

3.5.2 Specification

The MARIO environment is more complex than the Ms. PAC-MAN environment, as evidenced by the greater number of state and action predicates required to represent the state (see Table 3.3). Note that not every predicate will need to be used in all difficulty levels, as some enemy types are only present at higher difficulty levels.

State Predicates See Table 3.3 for the state predicates. Like Ms. PAC-MAN, there are many non-stationary objects in MARIO, and so all facts are re-asserted for every state instead of retracting and asserting new information. Only objects that are within the current view (and the goal) are asserted.

Table 3.3: Predicate definitions for Mario.

Relation Predicates	
distance(Thing, #D)	▷ *Thing* is horizontally #*D* right of Mario.
heightDiff(Thing, #H)	▷ *Thing* is vertically #*H* above Mario.
canJumpOnto(Thing)	▷ Mario can feasibly jump on to *Thing*
canJumpOver(Thing)	▷ Mario can feasibly jump over *Thing*
flying(Enemy)	▷ *Enemy* has wings
squashable(Enemy)	▷ *Enemy* can be jumped on
blastable(Enemy)	▷ *Enemy* can be shot with fireball
width(Thing, #W)	▷ The horizontal size #*W* of a *Thing*
carrying(Shell)	▷ If Mario is carrying *Shell*
passive(Shell)	▷ If *Shell* is not moving
Action Predicates — where #*D* (*distance*) and #*W* (*width*) are meta-information for resolving actions.	
moveTo(Thing, #D)	▷ Move towards *Thing*
search(Brick, #D)	▷ Search *Brick* (hit from beneath)
jumpOnto(Thing, #D)	▷ Jump on to *Thing*
jumpOver(Thing, #D, #W)	▷ Jump over *Thing* of width #*W*
pickup(Shell, #D)	▷ Picks up a *Shell*
shootFireball(Enemy, #D, MarioPower)	▷ Shoot *Enemy* with a fireball (when $MarioPower = fire$)
shootShell(Enemy, #D, Shell)	▷ Shoot *Enemy* with a held *Shell*
Type Hierarchy	
thing ← *brick; enemy; item; goal; pit; shell*	▷ All objects are *things*
enemy ← *goomba; koopa; piranhaPlant; spiky; bulletBill*	▷ Various *enemy* types
koopa ← *greenKoopa; redKoopa*	▷ Two types of *koopa*
item ← *mushroom; coin; fireFlower*	▷ Items/powerups
marioPower	▷ Mario's modes: $fire$, $large$ or $small$

The *distance(Thing, #D)* is calculated as the horizontal difference (in pixels) between *thing* and Mario, such that if Thing is to the left of Mario, *#D* is negative. *height(Thing, #H)* is calculated as the vertical difference (in pixels) between *thing* and Mario, such that if Thing is below Mario, *#H* is negative.

canJumpOnto(Thing) defines objects that could be landed upon (or entered into if *thing* is an *item*) in a single jump from Mario's last grounded position. This measure is only roughly defined as a rectangle encompassing all objects that are within Mario's maximum jump height and jump distance, and are not blocked by solid objects directly overhead (or two units overhead, if Mario is currently in *large* or *fire* form). *canJumpOver(Thing)* is defined similarly, except that the height of things that can be jumped over is decreased by 1.5 units (24 pixels). Technically, Mario could not jump over objects at the very limit of his horizontal jump range, but Mario only begins jumping when it is possible to jump on/over the object (in *transition rules*).

flying, *squashable*, and *blastable* are all qualities of enemies, but only *flying* is asserted directly (the other two are covered by background knowledge).

width(Thing, #W) defines the width of *thing*, which is 16 pixels for all objects except *pits* which have a variable width (at multiples of 16).

carrying and *passive* relate to shells left behind by jumped on *koopas*. If Mario picks up a *shell*, *carrying(Shell)* is true. If the *shell* is not moving and on the ground, *passive(Shell)* is true.

Type Predicates Table 3.3 defines the type predicates and their hierarchy. Each hierarchical rule is added to the *background knowledge*.

Action Predicates See Table 3.3 for the action predicates. Like Ms. Pac-Man, the numerical arguments within the actions are meta-information for resolving Mario's movement.

Background Rules There are only two background knowledge rules; they define how enemies can be killed:

enemy(X), not spiky(X), not piranhaPlant(X) $\rightarrow$ *assert(squashable(X))*

enemy(X), not spiky(X) $\rightarrow$ *assert(blastable(X))*

A *spiky* can only be killed by shooting a *shell* at it.

Action Rules The rules for producing the valid actions the agent can take are as follows:

canJumpOn(X), thing(X), distance(X, ((N_D *< -16) ∨ (*N_D *> 16))) → moveTo(X,* N_D*)*

brick(X), distance(X, (-32 ≤ N_D *≤ 32)), heightDiff(X, (16 ≤* N_H *≤ 80)) → search(X,* N_D*)*

canJumpOn(X), thing(X), distance(X, (-160 ≤ N_D *≤ 160)) → jumpOnto(X,* N_D*)*

canJumpOver(X), thing(X), distance(X, (-160 ≤ N_D *≤ 160)), width(X,* N_W*) → jumpOver(X,* N_D*,* N_W*)*

canJumpOn(X), passive(X), shell(X), distance(X, N_D*) → pickup(X,* N_D*)*

marioPower(fire), distance(X, N_D*), heightDiff(X, (-16 ≤* N_H *≤ 16)), enemy(X) → shootFireball(X,* N_D*, fire)*

carrying(Z), shell(Z), distance(X, N_D*), heightDiff(X, (-16 ≤* N_H *≤ 16)), enemy(X) → shootShell(X,* N_D*, Z)*

The rule for the *moveTo* action uses a negated test such that Mario can only move to objects not currently horizontally intersecting Mario's current position.

Transition Rules As for Ms. Pac-Man, there are no formally-defined Jess-syntax transition rules. Mario's low-level movement can be determined by multiple actions, as Mario is capable of doing several things at once: move $left$ or *right*, *jump*, and *run* (also used for shooting). Each relational action contributes a partial low-level action so Mario's low-level actions can be determined from multiple rules. When a relational action is resolved, it sets one or more of Mario's unset low-level actions to either 'on' or 'off'. Each rule in the policy is evaluated until either all rules have been evaluated (unset actions default to 'off'), or all of Mario's low-level actions have been determined. If a rule produces multiple action instantiations, Mario only acts upon the closest one (#D closest to 0).

When resolving actions in Mario, multiple actions may be utilised in

order to achieve the original action. In order to successfully calculate jumps, Mario's maximum jumping distance (while running) is predefined as a constant by the environment.

Selecting an action in MARIO does not guarantee that the entire action will occur. Resolving an action in MARIO usually requires multiple time steps (e.g. jumping onto something) but because the agent makes a decision at every time-step, the original action may not be completed. Furthermore, due to multiple factors such as Mario's current position, momentum, the target's position, and the layout of the level, resolving an action is not straightforward.

The *moveTo(Thing, #D)* action involves moving closer to *thing*, jumping over (via *jumpOnto*) obstacles if necessary. This is achieved by checking if there are any obstacles directly between Mario and either *thing* or half of Mario's maximum jumping distance, whichever is closer. If there is an obstacle, Mario jumps on to the top of the obstacle (whatever the highest point of the obstacle is). This action sets either $left$ or *right* to 'on' (and the opposite direction to 'off') and possibly sets $jump$ to 'on' if necessary.

The *search(Brick, #D)* action involves moving close enough to be under *brick*, then jumping into it. If Mario is more than two units horizontally away from *brick*, Mario first moves closer to *brick* (via *moveTo*). When Mario is close enough, $jump$ is set to 'on' and either $left$ or *right* is set to 'on' (and the opposite direction to 'off') to move towards the *brick* until it is struck.

The *jumpOnto(Thing, #D)* and *jumpOver(Thing, #D)* actions are resolved by using *moveTo* to get close enough to *thing* such that Mario could feasibly jump on to/over the object (assuming no obstacles obstruct the jump). When jumping on to an object, the point being jumped to is defined as directly above the object (or the object itself, if it is an *item*). When jumping over *things*, the point being jumped to is defined as 1.5 units (24 pixels) from the object (on the opposite side of the object to Mario), incorporating the width of the object into the calculation. If the Manhattan distance between Mario and the point being jumped to is greater than Mario's maximum jumping distance, Mario first moves to the object (possibly jumping closer, as per the *moveTo* action). When

close enough, *jump* is set to 'on' until Mario is one-third of the horizontal distance between the jumping point and the target point. *jumpOnto* and *jumpOver* use the same low-level actions as *moveTo* as well as possibly setting *run* to 'on' if Mario needs to move faster to jump onto an object.

The *pickup(Shell, #D)* action involves moving to (via *moveTo*) the shell, then picking it up by setting the *run* action (the action also used to hold the shell) to 'on' until the *shootShell* action is activated or Mario no longer holds the shell.

The *shootFireball(Enemy, #D)* action involves facing the correct direction ($left$ or $right$) and setting *run* to the opposite of the previous *run* action.

The *shootShell(Enemy, #D)* action involves facing the correct direction ($left$ or $right$) and setting *run* to 'off'.

The resulting low-level behaviour may perform multiple actions simultaneously (e.g. *moveTo* and *shootFireball*).

Reward Function The agent's reward is calculated using the default reward defined by the Mario AI environment which is based on a combination of factors (enemies killed, items collected, time remaining, distance travelled, Mario's mode, etc.):

$$\begin{aligned} R \leftarrow\ & 8 \times timeLeft + 1024 \times isGoal + distancePixels + 32 \times marioPower \\ & + 64 \times fireFlowers + 58 \times mushrooms + 16 \times coins + 42 \times kills \\ & + 12 \times jumpKills + 4 \times fireballKills + 17 \times shellKills \end{aligned}$$

The agent only receives reward when the episode ends, not during the episode.

The maximum reward for Mario could not easily be computed, due to the effect time has on reward. Hence, there are two primary reward-maximising strategies: attempt to complete the level as quickly as possible (maximising time remaining), or explore the level thoroughly, collecting coins, items and dispatching enemies where possible (maximising all other factors).

Constant Facts There is one constant in all Mario levels: the goal, with the

constant facts *goal(flag)*, *canJumpOn(flag)*, and *height(flag, 0)*.

Max Episode Steps Each level has a finite time limit of 200 seconds, which corresponds to 15 actions.

3.5.3 Goals

As in Ms. PAC-MAN, there is only one goal to achieve: reach the end of the level. However, MARIO can generate levels of varying difficulty:

Difficulty 0 The level has no *pits*, and only has non-*flying goombas* and *piranhaPlants* as enemies.

Difficulty 1 The level has a few short *pits*, and both *flying* and non-*flying goombas*, *koopas* (*red* and *green*) and *piranhaPlants*.

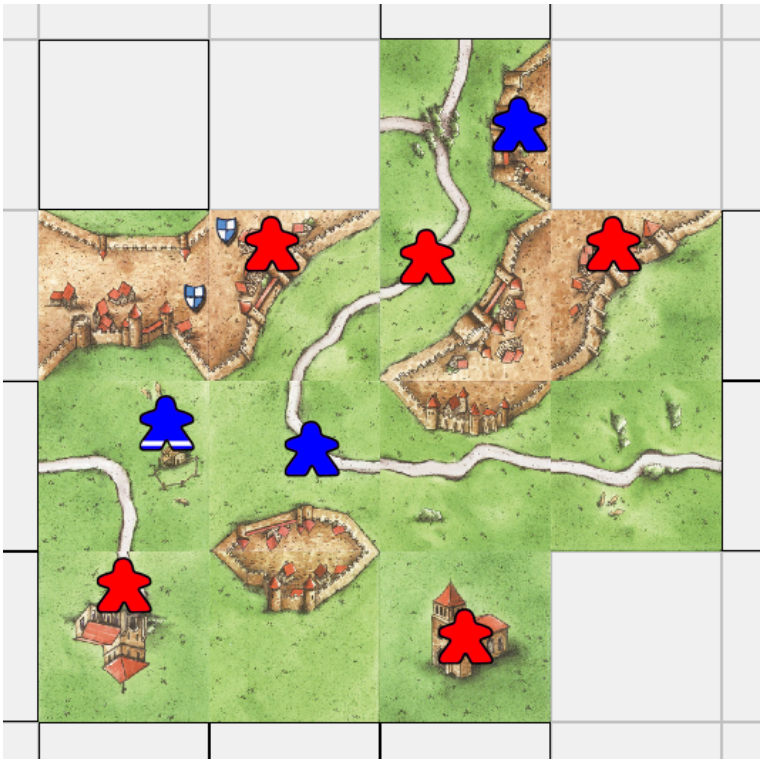


Figure 3.5: A screenshot of the CARCASSONNE environment with two players. Note that each terrain is controlled by only one meeple except a road, which was linked during a tile placement. The meeple with the white band across the legs is a farmer.

3.6 Carcassonne

Carcassonne is a medieval-themed board game designed by Klaus-Jürgen Wrede and published by Rio Grande Games (see Figure 3.5 for an example screenshot). The agent uses an open-source implementation of the game called JCloisterZone[9] for representing the game. In terms of the number of predicates for describing the state, the Carcassonne environment is the most complex environment of the testing environments, as it involves a large number of predicates describing the state of the game, though the agent can only perform two actions.

The game involves one or more players, where each player's turn consists of two consecutive actions. A player's first action is to place a randomly drawn tile adjacent to an existing tile on the board, such that the placed tile's edges match all adjacent existing tile's edges and extends the existing terrain. The player may then optionally place a 'meeple' (coloured figure representing the player's resources) on one of the terrain types present on the placed tile. Meeples may only be placed on terrain that does not already contain a meeple. Points are scored throughout the game by using these meeples to control various types of terrain and completing the terrain's end condition. Points are also scored for any meeples remaining on the board at the end of the game.

In Carcassonne there are four types of terrain: *city*, *road*, *cloister*, and *farm*[10]. Each tile contains one or more terrain types, orientated in various ways. Apart from the *cloister* (which cannot be linked), terrain linked over multiple tiles is regarded as a single terrain feature. When a terrain is completed, the person with the most meeples receives points for the terrain and all meeple(s) within the terrain are returned to the player to be used in future meeple placements.

A city is completed by closing all open edges (so it cannot be expanded upon, such as the small city made from 2 tiles in Figure 3.5). A road is completed by closing each end of the road (with an intersection or cloister, for example). A cloister is completed when it is surrounded by eight other tiles. A farm is never completed (or scored) during gameplay, it is only

[9] Download the source from http://jcloisterzone.com/en/

[10] More terrain types are added with game expansions. This work only uses the base game.

scored at the end of the game.

Carcassonne works well as an RRL problem because:

- There is only a small number of objects and actions, but there is a high level of strategy required to play effectively.
- Because the drawn tiles are randomised, the probability of playing the same game twice is very low.
- Because the game board can take nearly any shape during play, a propositional representation is effectively impossible.
- Carcassonne is the first environment to allow multiple (non-static) agents. These agents can be the same learning agent, a predefined AI, or even a human player.

3.6.1 Episodic Description

Each episode of Carcassonne starts with a single tile on the board (always the same starting tile), and a number of players (specified by the experiment setup, Section 3.6.3). The order of the players and tiles are randomised, and play begins. The current player draws a tile and places it in a valid position with any rotation. If the tile cannot possibly be placed anywhere, it is discarded and a new tile is drawn. If the placement results in a completed terrain, the terrain is scored and any meeples on the completed terrain are returned to their respective owners.

After placing their tile, a player may also place a meeple on any of the terrain on the placed tile. They may place it on any terrain that is not already claimed by a player (by extension across multiple tiles). If the meeple is placed on terrain that is already completed, the player receives points for the terrain and the meeple is immediately returned to their stock. The next player then takes their turn and play repeats until there are no tiles remaining to be placed. Any meeples still on the board at the end of the episode are scored with reduced points and the final scores are calculated.

Figure 3.6 lists all the tiles in a game of Carcassonne and their counts. Table 3.4 defines the scoring method for different terrain types.

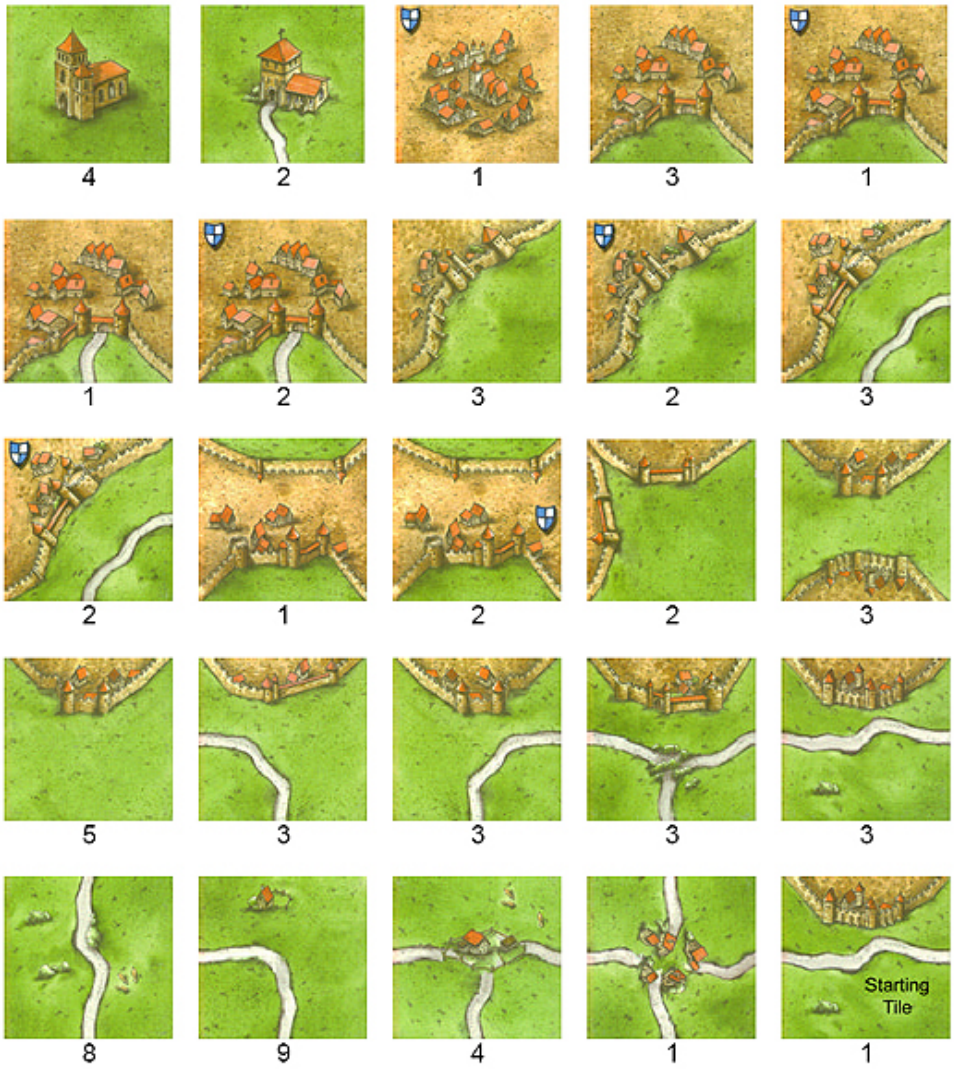


Figure 3.6: The set of tiles used in the game of CARCASSONNE. The number beneath each tile represents the number of copies of the tile. Copyright Rio Grande Games and Roy Levien. Reprinted with permission.

Table 3.4: Terrain scoring in CARCASSONNE. A pennant is a special icon found on city terrain. See *reward function* for further details.

Terrain	**Completed**	**Game end**
City	2 per tile + 2 per pennant	1 per tile + 1 per pennant
Road	1 per tile	
Cloister	1 + 1 per surrounding tile	
Farm	0	3 per completed bordering city

Table 3.5: Predicate definitions for CARCASSONNE.

Relation Predicates	
currentTile(Tile)	▷ The current *Tile* drawn/placed
currentPlayer(Player)	▷ The current turn *Player*
tileEdge(Tile, Edge, Terrain)	▷ The *Terrain* on the *Edge* of *Tile*
tileContains(Tile, Terrain)	▷ The *Terrain* contained within *Tile*
tileLocation(Tile, Location)	▷ The current *Location* of *Tile*
nextTo(Location, Edge, Terrain)	▷ The *Edge* of *Location* borders *Terrain*
numSurroundingTiles(Location, #N)	▷ #*N* tiles surround *Location* (1–8)
cloisterZone(Location, Cloister)	▷ A *Location* neighbouring *Cloister*
validLoc(Tile, Location, Orientation)	▷ A valid *Tile* placement at *Location* with *Orientation*
meepleLoc(Tile, Terrain)	▷ A valid meeple placement on *Terrain* in *Tile*
controls(Player, Terrain)	▷ *Player* controls *Terrain*
placedMeeples(Player, #P, Terrain)	▷ *Player* has #P meeples on *Terrain*
open(Terrain, #O)	▷ *Terrain* has #O edges to close
completed(Terrain)	▷ *Terrain* is completed
worth(Terrain, #W)	▷ *Terrain* is worth #W points (if complete)
meeplesLeft(Player, #M)	▷ *Player* has #M meeples unplaced
score(Player, #S)	▷ *Player's* current score is #S
tilesLeft(#T)	▷ There are #T tiles left to place
†*locationXY(Location, #Xl, #Yl)*	▷ The #*Xl*, #*Yl* coordinates for *Location*. Not visible to agent
†*edgeDirection(Edge, #Xe, #Ye)*	▷ The cardinal direction for each *Edge*. Not visible to agent
cEdge(Edge1, Edge2)	▷ *Edge*2 is clockwise from *Edge*1
ccEdge(Edge1, Edge2)	▷ *Edge*2 is counter-clockwise from *Edge*1
oppEdge(Edge1, Edge2)	▷ *Edge*2 is opposite *Edge*1
Action Predicates	
placeTile(Player, Tile, Location, Orientation)	▷ *Player* places *Tile* at the given *Location* and *Orientation*
placeMeeple(Player, Tile, Terrain)	▷ *Player* places a meeple on *Terrain* in *Tile*
Type Hierarchy	
terrain ← *city; road; cloister; farm*	▷ Four *terrain* types
tile	▷ One of the 72 tiles
player	▷ An individual player
location	▷ A location on the board
edge	▷ Four edges: *north*, *east*, *south*, *west*
orientation	▷ Four orientations: *r*0, *r*90, *r*180, *r*270

3.6.2 Specification

State Predicates See Table 3.5 for the state predicates. The set of state facts is extracted from the game state after every action.

The *currentTile* is drawn by the *currentPlayer* at the start of their turn. Each tile has four *edges*, where each *tileEdge* has some *terrain* bordering it. *tileContains* also defines all the *terrain* contained by the *tile*.

All placed *tiles* have a *location*, and each *location* borders one or more *terrains* on 1–4 *edges*. Each *location* also has a *numSurroundingTiles* between 1–8. If a *location* is one of the eight locations around a *cloister*, it is also a *cloisterZone*.

The set of *validLocs* is defined for the *currentTile*, with the correct *orientation*. A valid location is a vacant location adjacent to one or more existing tiles where the current tile can be placed (with some rotation) such that each edge of the current tile matches the edges of the existing tile(s). When the tile has been placed, *meepleLoc* defines which *terrain* types on the *tile* the agent can place a meeple on. A valid meeple location is a terrain location on the current tile that has no other meeple on it.

Each *terrain* has four other facts: which *player controls* it, how many meeples have been placed on it (*placedMeeples*), if it is still *open* or already *completed*, and what it would be *worth* if it was *completed*.

meeplesLeft defines how many meeples each *player* has remaining to place, *score* states what each *player*'s current score is, and *tilesLeft* states how many tiles are remaining.

The final facts encode low-level information about the *locations* and *edges* for background knowledge facts.

Type Predicates Table 3.5 defines the type predicates and their hierarchy. Each hierarchical rule is added to the *background knowledge*.

Action Predicates See Table 3.5 for the action predicates.

Background Rules CARCASSONNE has three background knowledge rules:

$$edge(north) \rightarrow cEdge(north, east), ccEdge(north, west), oppEdge(north, south)$$

cEdge(N, E), *ccEdge(N, W)*, *oppEdge(N, S)* $\rightarrow$ *cEdge(E, S)*, *ccEdge(E, N)*, *oppEdge(E, W)*

locationXY(L1, N_0, N_1), *not tileLocation(T1, L1)*, *edgeDirection(E, N_2, N_3)*, *locationXY(L2, (N_0 + N_2), (N_1 + N_3))*, *tileLocation(T2, L2)*, *oppEdge(E, Eopp)*, *tileEdge(T2, Eopp, Ter)* $\rightarrow$ *nextTo(L1, E, Ter)*

Action Rules The rules for producing the valid actions the agent can take are as follows:

currentPlayer(X), *validLoc(Y, Z, W)* $\rightarrow$ *placeTile(X, Y, Z, W)*

currentPlayer(X), *meepleLoc(Y, Z)*, *meeplesLeft(X, ($N_M > 0$))* $\rightarrow$ *placeMeeple(X, Y, Z)*

Transition Rules Like for Ms. Pac-Man and Mario, there are no formally-defined Jess-syntax transition rules. Only one action is required per step, but if a policy rule produces more than one action, a random action is selected from the produced actions.

The *placeTile(Player, Tile, Location, Orientation)* action is resolved by adding the current tile to the board at the given location and orientation. If any terrain is completed, any meeples on the completed terrain are returned to their respective owners. If no tile placing action is selected, and there is only one learning agent, the game is over. If there are multiple learning agents playing, a random placement is selected instead. If no agents are selecting tile placements, the episode ends prematurely.

The *placeMeeple(Player, Tile, Terrain)* action is resolved by adding a *player*'s meeple to the designated *terrain*. If the meeple is placed on completed terrain, the player receives both points for the terrain, and the placed meeple back. If no meeple placing action is selected, no meeple is placed, and the next player's turn begins.

If a static-behaviour AI is present (see Section 3.6.3), its entire turn is performed automatically by the environment.

Reward Function Like Ms. Pac-Man, the reward function is simply the score received from the game. Table 3.4 defines how scoring is calculated. However, there are two exceptions: a two-tile completed city is only worth 2 points (+ 2 per pennant), and when scoring a farm, a city may only be scored once regardless of the number of farms bordering

it.

When multiple learning agents are playing, if an agent does not select a tile placement on its turn, it receives an arbitrary -1000 reward penalty, as it is considered to have 'given up.' This separates policies that 'gave up' from policies that selected a tile placement every turn.

Because of the random nature of the game, calculating the maximum number of points is difficult. However, in Section 6.5, the average score received by the built-in AI is presented as a rough idea of effective behaviour.

Constant Facts There are twelve constants, eight *edge* facts and four tile *orientations*:

edge(north), *edge(east)*, *edge(south)*, *edge(west)*, *edgeDirection(north, 0, -1)*, *edgeDirection(east, 1, 0)*, *edgeDirection(south, 0, 1)*, *edgeDirection(west, -1, 0)*, *orientation(R0)*, *orientation(R90)*, *orientation(R180)*, *orientation(R270)*

The *edgeDirections* are primarily used to resolve the *nextTo* background rule and are probably of little value to a learning agent.

Actions Per Step The agent is only required to provide 1 *action per step*, but if it provides more than one (e.g. if a single policy rule produces multiple actions), a random action is selected from the actions. Note that not providing an action during the *placeMeeple* phase is also allowed.

Max Episode Steps Because there are only 72 tiles per game (71 of which the players can place), each episode is of a fixed length bounded by the number of tiles remaining.

3.6.3 Goals

Carcassonne only has one goal: maximise your score within the 70 tile and meeple placements. But there are multiple environment setups that can fundamentally modify the challenge of the environment.

Single Player Typically, Carcassonne is a multiplayer game, but it can be played with a single player. This goal biases the agent's learning towards creating high-valued terrain features without being interrupted by opposing players.

Agent vs. Random This goal matches a learning agent against a random-behaviour player. For selecting actions, the random-behaviour player selects a random valid location and orientation to place a tile, then places a meeple on a random valid terrain with 50% probability (otherwise, it does not place a meeple). This goal demonstrates how the learning agent performs against random behaviour.

Agent vs. Static AI The JCloisterZone game includes a static-behaviour AI agent that uses a one-step look-ahead maximisation strategy that utilises tile probabilities to place tiles in an effective manner, both for personally gaining points, and for blocking other players from completing terrain types. This goal matches the learning agent against a number of AI opponents, such that the learning is biased towards behaviour for dealing with a skilled opponent(s).

Agent vs. Agent This goal matches the *same* agent against itself. That is, the same learning agent controls all players (but the policies generated for each agent may not be the same). This goal biases learning towards defeating opponents of the same skill level, learning a strategy that has no particular bias against defeating other strategies (which could also be a drawback). An added benefit of this experimental setup is the agent receives K samples per episode, where K is the number of players.

3.7 Summary

This chapter has introduced the terminology that will be used throughout the remainder of the thesis. The Rete algorithm used by Jess is a key factor in the efficiency of the Cerrla algorithm described in later chapters. This chapter also introduced the standard formatting of the policies Cerrla produces, such that the rules of a policy may only contain specific argument types and the conditions of the rules are sorted to heuristically improve rule evaluation efficiency.

This chapter also introduced the framework for representing a relational environment, as well as the four environments that are used for testing the agent in Chapter 6. Table 3.1, 3.2, 3.3 and 3.5 describe the environment predicates used in each environment and provide a reference for comprehending the rules of Cerrla's produced policies.

4

CERRLA

The aim of this research is to develop a learning algorithm capable of solving problems within large, relational, reward-driven environments. In Chapter 2, various existing approaches were described that solve similar problems. Some approaches (Džeroski et al., 2001; Driessens et al., 2001, 2006; Dabney and McGovern, 2007) used value-based table formalisms that work fine in computationally small environments, but, without some form of pre-defined state abstraction, will fail to perform adequately in large, complex environments. Other approaches search the policy space directly, avoiding the need to store each state and value, and taking advantage of relational variables to generalise over collections of objects without need for pre-defined abstractions.

The Cross-Entropy Method (CEM) is one such policy searching method. It is similar to evolutionary algorithms, in that it uses a population of samples and evaluates samples through some fitness function, but the CEM guides its sampling process in a statistically optimal manner using Kullback-Leibler (KL) divergence. The CEM has previously been successfully applied to games (e.g. Tetris (Kistemaker, 2008), (Szita and Lörincz, 2006) and Ms. Pac-Man (Szita and Lörincz, 2007)).

The algorithm developed here has been named Cross-Entropy Relational Reinforcement Learning Agent (CERRLA). CERRLA was originally based upon work by Szita and Lörincz (2007), in which the CEM was used to optimise a static set of hand-coded rules to generate a policy for playing

Ms. Pac-Man. CERRLA has since expanded the scope of that algorithm to function in a range of relational environments, generate and specialise relational rules dynamically, and perform updates in an online, rather than population-based, manner. The resulting behaviour learned by CERRLA outputs low complexity, easy-to-read relational policies that obtain large rewards when evaluated in their respective environments.

This chapter begins with a high-level overview of how CERRLA creates and optimises relational policies for a given problem. The next section then formally describes the CEM, the algorithm forming the basis of CERRLA's learning. The remaining sections are structured sequentially with respect to the algorithm's execution and present a policy-level description of the agent's learning process. The details of rule creation and specialisation are largely found in Chapter 5.

4.1 CERRLA Overview

Algorithm 4.1 Pseudocode summary of the CERRLA algorithm. The algorithm creates and optimises a list of relational rules for acting effectively within a given environment.

```
1: Initialise the environment                                   ▷ Chapter 3
2: Initialise distribution set 𝔻                                ▷ Section 4.2.1
3: Observe environment and determine RLGGs                      ▷ Section 4.3
4: repeat
5:   Generate policy π_i from 𝔻                                 ▷ Section 4.4
6:   for j ← 1 to 3 do                  ▷ Evaluate each policy three times
7:     Evaluate policy π_i against the environment              ▷ Section 4.5
8:   end for
9:   Note policy sample and update distribution 𝔻               ▷ Section 4.6
10:  Specialise rules (if 𝔻 is ready)                           ▷ Section 4.7
11: until IsConverged(𝔻)                                        ▷ Section 4.6
```

CERRLA learns behaviour by randomly sampling rules from a set of *candidate rule distributions* and combines these rules into decision lists which act as a policy for the agent (an example policy is described in Section 4.1.1 below). The rule distributions are initially uniform, but as empirically useful rules are identified, their respective sampling probabilities are increased. The process of randomly sampling data from distributions and increasing the sampling probability of empirically useful data is known as

the Cross-Entropy Method (CEM), which forms the backbone of CERRLA's probability optimisation aspect (described in the rest of Chapter 4). Algorithm 4.1 roughly describes the CERRLA learning process in pseudocode.

CERRLA's rules are created in a top-down fashion: beginning with an Relative Least General Generalisation (RLGG) rule for each action (a rule which defines the minimal preconditions for performing an action), search for better rules by gradually specialising empirically useful rules, guiding the search with the probabilities learned by the CEM. The RLGG rule is created by observing which patterns of conditions are always true whenever an action is available in the valid actions for every state (using variable arguments where appropriate). These rules form a foundation from which to specialise new rules using three separate specialisation operators: adding a condition to the rule, replacing a variable with a goal variable, and splitting a numerical range into a subrange. Except for RLGG rules, every rule is a specialisation of another rule. To avoid creating redundant or illegal rules, CERRLA also infers and uses a set of *simplification rules* to remove redundant or illegal condition combinations from the created rules. The details of rule discovery are described in Chapter 5.

When beginning learning in a new environment, CERRLA identifies the RLGG rules and uses each one to *seed* a new candidate rule distribution (one for each action). Each specialisation of the RLGG rules also *seeds* new candidate rule distributions. When a distribution is *seeded*, it is filled with all immediate specialisations of the seed rule. The result is one distribution for each RLGG rule and each specialisation of the RLGG rules.[1]

A policy is generated by sampling one rule from each candidate rule distribution, with respect to the current sampling probabilities for each rule in the distribution. The distributions themselves also have two properties that affect policy generation: the probability of any rule being sampled from the distribution at all, and the relative position of the sampled rule to other rules in the policy.

Once a policy is generated, it is tested in three separate episodes and assigned a value equal to the average total reward received per episode. If the value of the policy is higher than the N_E^{th} best policy thus far (where

[1] Not every specialisation of the RLGG immediately creates a new candidate rule distribution, see Section 4.7.

N_E represents the number of 'elite' policy samples to use for updating), it is stored as an elite sample. The elite samples are a subset of all samples created thus far, and they are defined as the best subset of samples according to the values of the samples. A sample remains an elite sample until there are N_E higher-valued elite samples.

After a policy is tested, the probabilities of the distributions are updated. This involves adjusting the sampling probabilities for the rules in each distribution and adjusting the distribution's properties such that the policies present in the elite samples are more likely to be generated again. This process repeats, generating and testing policies, then adjusting the sampling probabilities of the distributions to generate better policies. When a rule becomes highly probable, it *branches* from the distribution to create a new distribution with the probable rule as the *seed*. This new distribution is filled with specialisations of the seed rule and the seed rule is removed from the old distribution. A rule will not branch from a distribution if it was the original seed rule for the distribution.

CERRLA continues to generate and test policies, update the distributions, and branch rules from distributions until the probabilities for each distribution become stable (i.e. are considered converged). Once converged, the best elite sample is output as a solution to the problem.

4.1.1 Example Policy

Below is an example policy produced by CERRLA:

$clear(G_0)$, $clear(G_1)$, $block(G_0) \rightarrow move(G_0, G_1)$
$above(X, G_1)$, $clear(X)$, $floor(Y) \rightarrow move(X, Y)$
$above(X, G_0)$, $clear(X)$, $floor(Y) \rightarrow move(X, Y)$

This policy is in fact an optimal policy for solving the onG_0G_1 goal in the BLOCKS WORLD environment. The behaviour of the policy is to:

1. Place goal block G_0 onto goal object G_1 if both are clear.
2. If a block X is on top of goal block G_1 and it is clear, place it on the floor.
3. Same as 2, but with goal block G_0 instead of G_1.

Each of the rules of the policy were sampled from a separate distribution. When this policy was produced (when CERRLA had converged), all other distributions had a near-zero usage probability, while the distributions for each of these rules were near-one. The three rules shown all also had a high sampling probability, as they were more effective than all other rules in their respective distributions. Each of these rules required at least two specialisation operations to get from the RLGG rule to their current state, and each rule used simplification rules to remove redundant conditions (e.g. when *floor(Y)* is true, *clear(Y)*, a condition present in the RLGG rules, is redundant).

4.2 Cross-Entropy Method

The CEM is an optimisation algorithm that maintains a distribution of possible solutions to a problem and revises the probabilities of the distribution with every iteration. Originally developed by Rubinstein (1997), the CEM was created as an adaptive algorithm for estimating rare event probabilities in complex stochastic processes. It has since been used for a number of different applications, including game-playing agents (Szita and Lörincz, 2006, 2007; Kistemaker, 2008; Tak, 2010), clustering (Kroese et al., 2007), control and navigation (Helvik and Wittner, 2001), and continuous optimisation (Kroese et al., 2006) to name a few. For a comprehensive exploration of the CEM, see De Boer et al. (2004).

CERRLA uses an *online* variation of the CEM to learn the optimal sampling probabilities for *multiple* distributions of rules in parallel. For simplicity, the following description of the CEM is population-based and only uses a single distribution, but the changes to CERRLA's core algorithm are described Section 4.2.1.

The algorithm is essentially composed of two steps:

1. Generate N samples from a probability distribution of data and evaluate them, assigning a value to them.
2. Sort the samples into descending value order, then use the top subset of samples E to decrease the KL divergence between the data distribution and E, thereby increasing the chance of sampling the data present in E again.

Intuitively, the algorithm works as follows: in the early stages, the algorithm does not perform any worse than random guessing, but as it gathers samples, it shapes the distribution such that guessing becomes more and more biased towards high-value samples.

Algorithm 4.2 Pseudocode for the cross-entropy method. Locates the highest performing sample in a collection of data.

Require: $X = \{x_1, \ldots, x_n\}$ ▷ The data distribution (with probabilities $p_1, \ldots, p_n$)
Require: N ▷ The population size
Require: ρ ▷ The proportion of elite samples
Require: α ▷ The step-size update parameter
  $N_E = \rho \cdot N$ ▷ Define the minimum number of elites
  for $t \leftarrow 0$; IsConverged; $t \leftarrow t + 1$ **do** ▷ Loop until converged
    for $i \leftarrow 1$ **to** N **do** ▷ Generate N samples
      sample $\mathbf{x}_i = x_j \in X_t$ with probability $p_{t,j}$ ▷ Sample N samples
      $f_i \leftarrow f(\mathbf{x}_i)$ ▷ Evaluate each sample
    end for
    sort $f_1 \ldots f_N$ into descending order
    $\gamma_{t+1} \leftarrow f_{N_E}$ ▷ Determine the elites threshold
    $E_{t+1} \leftarrow \{\mathbf{x}_i \mid f_i \geq \gamma_{t+1}\}$ ▷ Extract the elites
    for $j \leftarrow 1$ **to** n **do** ▷ For every element in distribution
      $p'_j \leftarrow (\sum_{\mathbf{x}_i \in E_{t+1}} 1_{\mathbf{x}_i = x_j}) / |E_{t+1}|$ ▷ Calculate observed distribution
      $p_{t+1,j} \leftarrow \alpha \cdot p'_j + (1 - \alpha) \cdot p_{t,j}$ ▷ Step-size update probabilities
    end for
  end for

Formally the CEM algorithm (shown in Algorithm 4.2) is as follows: the algorithm begins with a distribution of data ($X = \{x_1, \ldots, x_n\}$), where each data item x_i has a corresponding sampling probability $p_i \in [0, 1]$: $\sum_{j=1}^{n} p_j = 1$ (a distribution is typically uniform at the outset). N samples are generated ($\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$), selecting data based on its (initially equal) probability where $\mathbf{x}_i = x_j$ with probability p_j. The samples are then evaluated with some function $f(\mathbf{x})$ and sorted into descending order. The samples with $f(\mathbf{x}) \geq \gamma_{t+1}$ are extracted as 'elite samples' E_{t+1}, where γ_{t+1} is equal to the value of the N_E^{th} sorted sample. The minimum number of elite samples is defined as $N_E = \rho \cdot N$ (typically $\rho = 0.05$). Note that there may be more than N_E elite samples, as multiple samples could have a value equal to the threshold.

The observed distribution $\mathbf{p}'_{t+1}(\mathbf{X}) = \{p'_1, \ldots, p'_n\}$ is then calculated using

the frequency of data seen within the elite samples, defined as:

$$p'_{j,k} \leftarrow \Big(\sum_{\mathbf{x}_i \in E_{t+1}} \left\{ \begin{array}{ll} 1 & \text{if } \mathbf{x}_i = x_j \\ 0 & \text{otherwise} \end{array} \right\} \Big) \Big/ |E_{t+1}| \tag{4.1}$$

meaning p'_j is equal to the proportion of samples in the elites that are x_j.

Instead of directly setting the new probabilities equal to the observed probabilities, the update process can be 'softened' by using a step-size update parameter α (typically α is between 0.4 and 0.9, De Boer et al. (2004)) to smoothly modify the distribution probabilities:

$$p_{t+1,j} \leftarrow \alpha \cdot p'_j + (1 - \alpha) \cdot p_{t,j} \tag{4.2}$$

This sample-update loop is repeated until some convergence measure is reached; usually either a pre-defined maximum number of iterations have passed, or all probabilities have converged to either 0 or 1, or the KL divergence between the observed distribution and the current distribution is less than β for some number of iterations (where β is some positive value < 0.1).

Costa et al. (2007) prove the convergence properties of the CEM, such that given a constant α parameter, the algorithm will eventually converge to a point where all probabilities are either 0 or 1. Furthermore, the paper proves that the probability for an optimal sample to be drawn is inversely proportional to α. Hence, a balance between fast convergence and optimal convergence must be decided upon by the choice of α parameter.

4.2.1 Application to RRL

CERRLA uses a modified form of the CEM to sample and update multiple *candidate rule distributions* D in a set of candidate rule distributions $\mathbb{D} = \{D_0, D_1, \ldots\}$ which are used to create decision-lists of rules that represent the agent's policy.

CERRLA uses an online variation of the CEM such that it becomes an incremental method instead of a batch-based method. Szita and Lörincz (2008) define an 'online CEM' which, instead of sampling batches of N samples, uses a sliding window of N samples, such that the elites E consist of the

best samples from the last N samples (instead of the best samples in a batch). The minimal size of the elites is still N_E.

Initially, $\mathbb{D}$ is empty, but as CERRLA learns new behaviour, the number of rule distributions increases. Each candidate rule distribution contains a number of rules $D = \{r_1, \ldots, r_n\}$ consisting of a single *seed* rule and all immediate specialisations of that rule (see Section Section 4.7), where each rule r_j has a corresponding probability p_j ($\sum_{j=1}^{|S|} p_j = 1$).

Two metrics are used for measuring a distribution: $|D|$ represents the number of rules within D, and $KL(D)$ represents the inverse Kullback-Leibler (KL) divergence, or inverse distance from the uniform distribution, of D. The KL divergence is a non-symmetric measure of the difference between distribution P and distribution Q (Kullback and Leibler, 1951). Given the formula for calculating the KL divergence from P to Q as:

$$d_{KL}(P\|Q) = \sum_i \ln\Big(\frac{P(i)}{Q(i)}\Big)P(i) \tag{4.3}$$

the inverse distance from the uniform distribution $KL(D)$ is defined as:

$$\begin{aligned} KL(D) &= |D| \cdot \big(1 - d_{KL}(D\|D_{uniform})\big) \\ &= |D| \cdot \big(1 - \sum_{r\in D} \log_{|D|}(\frac{p_r}{|D|^{-1}})p_r\big) \\ &= |D| \cdot \big(1 - \sum_{r\in D} \log_{|D|}(p_r \cdot |D|)p_r\big) \end{aligned} \tag{4.4}$$

where $\log_{|D|}$ is used instead of ln to normalise the KL divergence to between $[0, 1]$. A uniform (not yet updated) distribution has $KL(D) = |D|$, but a distribution with a single high probability rule (e.g. $p_j \geq 0.95$) has $KL(D) \approx 0$. The closer $KL(D)$ is to 0, the more 'converged' it is considered. This is used for population calculations (Section 4.6.1) and specialisation triggering (Section 4.7.2).

Each D also has two properties: the probability that a rule from D is present within a policy, $p(D) \in [0, 1]$ (initially $p(D) \leftarrow 0.5$); and the average relative position of sampled rules within generated policies, $q(D) \in [0, 1]$, where 0 represents the first position and 1 represents the last (initially $q(D) \leftarrow 0.5$).

Policy samples are generated from $\mathbb{D}$ (Section 4.4) and evaluated against the environment (Section 4.5). Online CEM allows the algorithm to imme-

diately add the sample to the elite samples (depending on sample value) and update the distribution (Section 4.6). Another modification to the CEM is that the data changes when the algorithm selects a rule for specialisation, creating a new candidate rule distribution with new rules (Section 4.7).

4.3 Algorithm Initialisation

The agent begins the learning process with an initially empty set of distributions $\mathbb{D} = \{\}$, therefore it must create rules that allow it to act within the environment. To create the rules, it begins the first episode of the experiment and observes the state of the environment, and from that observation, the agent is able to create a Relative Least General Generalisation (RLGG) rule for each action present in the state (see Chapter 5 for details of how this rule is created). Each RLGG rule defines the minimally general conditions for taking the action and the basis of the algorithm's rule exploration process.

For every action a, an RLGG rule r^a_{RLGG} is created by setting the LHS of the rule as the conditions that are always true whenever action a is present (this is fully explained in Section 5.2). Whenever the observations model modify the always-true action-conditions, RLGG rule conditions are updated with the changed conditions.

Each RLGG rule r^a_{RLGG} then *seeds* a newly created candidate rule distribution D^a_{RLGG}. If there are existing agent observations, the distribution is filled with all single-step specialisations, as well as creating the initial 'branched' candidate rule distributions (explained later in Section 4.7). Otherwise, the distribution only contains the RLGG rule (hence $p_{r^a_{\text{RLGG}}} = 1$). Because the agent's initial policy is empty when it first creates these rules, it immediately adds all RLGG rules to the policy in random order.

Example 4.3.1. The RLGG rule calculated for the Blocks World *move* action is (after simplification, see Section 5.4):

$$r^{move}_{\text{RLGG}} = (clear(X), clear(Y), block(X) \ \rightarrow \ move(X, Y)) \tag{4.5}$$

This rule covers every possible *move* action in Blocks World while remaining specific enough to describe the minimal preconditions required for taking the action.

4.4 Generating Policy Samples

A policy π_i is generated by firstly determining which candidate rule distributions $D \in \mathbb{D}$ will be used, then sampling a rule from those that are included. For each $D \in \mathbb{D}$, the distribution will only be in the policy with probability $p(D)$. The position of the distribution's rule in the policy is based on $q(D)$ and the relative positions of other distributions, but some randomness is added to explore different positions. When a distribution is to be used, a relative position $relQ(D)$ is calculated as a Gaussian distributed value with the parameters $q(D)$ for the mean of the distribution, and $q_\sigma(D)$ as the standard deviation which is based on how close $p(D)$ is to 0.5: if $p(D) < 0.5 : q_\sigma(D) = p(D) \cdot 0.5$, otherwise $q_\sigma(D) = (1 - p(D)) \cdot 0.5$. Therefore, when a distribution is initialised with $p(D) = 0.5$, its relative position in policies varies wildly, but as the sampling probability $p(D)$ converges to 0 or 1, the relative position becomes more fixed.

When all utilised distributions have calculated a $relQ(D)$ value, they are ordered in increasing order and a rule is sampled from each one by selecting a rule r_i with rule probability p_i.

Example 4.4.1. An example Blocks World policy sample is:

$$\begin{aligned} \mathit{above}(X, G_1), \mathit{clear}(X), \mathit{floor}(Y) &\rightarrow \mathit{move}(X, Y) \\ \mathit{clear}(G_0), \mathit{clear}(G_1), \mathit{block}(G_0) &\rightarrow \mathit{move}(G_0, G_1) \\ \mathit{above}(X, G_0), \mathit{clear}(X), \mathit{floor}(Y) &\rightarrow \mathit{move}(X, Y) \end{aligned}$$

This example policy is in fact an optimal policy for solving the OnG_0G_1 task in Blocks World. Each rule in this policy was sampled from a different candidate rule distribution.

4.5 Evaluating a Policy

The agent's policy π_i is evaluated when the agent receives the current observations for the state of the environment. Starting with the first rule in the policy, each rule's is evaluated against the state. During this evaluation, the current goal substitution map is applied to the rule (see Section 3.2.1 for details) and any numerical bounds in the form N_1^{min} or N_1^{max}, defined as the observed numerical bounds for the variable N_1 (see Section 5.5), are

replaced by their respective numerical values.

Recall that the JESS rules engine is used in this research to represent the environment (Section 3.1.2). CERRLA's policy evaluation can be streamlined by taking advantage of the Rete network JESS uses. The rules of the policy define the structure of the network, where each node is either a rule condition (identical rule conditions between rules can be shared) or a join node between two conditions (defining the conjunction of rule conditions). When the facts defining the current state are asserted to the network as the current state observations, they are immediately processed through the nodes of the network, such that matches to the rules of the policy are immediately known. Because an agent's policy does not change per episode, the Rete network does not need to be recreated every state, resulting in efficient policy evaluation.

If the rule query is successful, the resulting variable bindings are applied to the rule action, creating one or more ground action atoms. Any actions not present in the set of valid actions provided by the state are removed. Each rule's set of actions are stored within a list of sets of resulting actions, and once enough actions have been created, the list of collections of actions is returned to the environment to be applied. In most environments, only a single action will be required, selected randomly from the first set of actions. However, environments like Ms. PAC-MAN which can utilise multiple actions per step may require the entire list of output actions. Depending on the environment, the episode may end early if the policy does not produce any actions (usually accompanied by a large negative reward).

Throughout the episode, the agent receives a reward value, and by the end of the episode, the policy achieves total reward R_j. To reduce variance between episodic reward received, each policy sample π_i is evaluated over n episodes and the average reward $R_i \leftarrow \sum_{j=1}^{n} R_j/n$ is used as the 'value' of the policy. In experiments, $n = 3$, which is small enough to quickly evaluate different policies, but also large enough to reduce major variance.

4.6 Updating the Distributions

Because CERRLA uses an online CEM, the elite samples are a 'floating window' representing the highest valued samples from the past N iterations.

When sample π_i has been evaluated and has an averaged value R_i, it is added to the sorted elite samples E and the elite samples are then used to update the distributions.

4.6.1 Determining Elite Samples

The typical CEM is used to optimise a pre-defined set of data that does not change in size so the population size N and minimum number of elites N_E can remain static. However, CERRLA operates in a wide range of environments and the number of rules and level of convergence in CERRLA is variable at any given episode, so the number of samples required for obtaining a representative sample of the current rule distributions also needs to be flexible. As the number of rules present in $\mathbb{D}$ increases, the number of samples required to get a representative sample should also increase. But, N_E also needs to be small enough to only represent the best samples.

Determining Population Size

The aim of the dynamic population size is to maintain an elite set of samples that can approximately represent the current state of the set of distributions. E.g. the observed elites distribution can feasibly represent the approximate probabilities of any given distribution. $KL(D)$ provides a good indication of the number of rules needed to observe similar probabilities. However, each distribution also has a sampling probability property $p(D)$, which represents how important *any* of the rules in the distribution are. Therefore, the minimum number of elites N_E is set as the largest $KL(D)$ distribution (weighted by the distribution's sampling probability $p(D)$). To avoid N_E becoming too small (e.g. $N_E \leq 1$), we also set the minimum to the sum of distribution sample probabilities. This means that policies involving multiple distributions have a larger minimum elite sample set than simpler policies. The equation for calculating N_E is defined below:

$$N_E = \max \left\lceil \underbrace{\arg\max_{S \in \mathbb{D}} \left(KL(D) \cdot p(D) \right)}_{\text{largest distribution}}, \quad \underbrace{\sum_{S \in \mathbb{D}} p(D)}_{\substack{\text{sum distribution} \\ \text{sampling probabilities}}} \right\rceil \tag{4.6}$$

where $N = N_E/\rho$, as with the regular CEM. This results in a relatively large N_E at the beginning of learning which gradually decreases as rule

and distribution sampling probabilities change.

Adding a Sample

Once N_E is known, the elite samples can be calculated. To avoid converging to the same set of samples, any elite samples that have existed for N iterations are removed. The current sample π_i is then added to the elite samples with value R_i and the elite samples are sorted into descending order. The threshold value is then computed as $\gamma \leftarrow E_{N_E}$, which is the value of the N_E^{th} element of the elite samples (or the last value, if $|E| < N_E$). Any samples valued less than the threshold are dropped from the elite set.

4.6.2 Iterative Updates

CERRLA performs an update at every iteration, but in order to match the regular CEM update process, the update parameter needs to be reduced to a single-step value: $\alpha_1 = \alpha / N$. The resulting α_1 parameter is small, but after N iterations, matches the standard α update (assuming a sample remains in the elite samples for all N iterations).

A restriction applies to updates: a distribution is only updated if it has been sampled a 'fair' amount of times to avoid update bias towards early samples. Using a coefficient C to determine a fair sample, a distribution is only updated when $n(D) \geq C \cdot |D|$ ($n(D)$ represents the number of times D has been sampled). E.g. a distribution D, where $|D| = 15$, will not be updated until it has been sampled $15 \cdot C$ times. When a distribution is finally able to be updated, the elites should represent the best rules it was able to produce.

An appropriate value for C can be determined by solving Equation 4.7, which defines the proportion of rules $p(x)$ that are sampled at least once from a uniform distribution X after $C \cdot |X|$ samples (Aslam et al., 2007):

$$\begin{aligned} p(x) &= 1 - \left(\frac{|X| - 1}{|X|}\right)^{C \cdot |X|} \\ &= 1 - e^{-C} \end{aligned} \tag{4.7}$$

Solving for $p(x) = 0.95$, Equation 4.7 produces $C = 2.996$ ($C = 3$ to simplify values). As C is increased, the probability of sampling each rule from a candidate rule distribution increases, but at the cost of requiring

more samples before a distribution is updated.

In some environments (e.g. BLOCKS WORLD), finding a sample with a non-minimal reward can be rare, and if finding such a sample is less probable than ρ (the elites proportion), the threshold value for the elites will equal the minimal reward, resulting in the elites representing every sample (the randomly sampled distribution $E \approx X$). As each sample in the elites corresponds to a proportion of the update, an update can still be performed even if $|E| = N$ by adjusting γ to the next highest threshold, resulting in a smaller set of elites E'. To match this smaller sample size, α_1 is decreased by a factor of $|E'|/N_E$. In the case where there is no sample better than any other, no update is performed.

4.6.3 Updating the Distributions

A sample in CERRLA is a policy consisting of multiple relational rules, where each rule is sampled from a separate candidate rule distribution. Only the rules that were used throughout the sample's testing episodes matter, therefore unused rules are not included in the update and, implicitly, negatively updated.

Updating Distribution Properties

The distribution sampling probability $p(D)$ and position $q(D)$ values are updated using the following equations (derived from Equation 4.2 in Section 4.2) with the single-step α_1 value.

$$p_{t+1}(D) \leftarrow \alpha_1 \cdot p'(D) + (1 - \alpha_1) \cdot p_t(D) \tag{4.8}$$

$$q_{t+1}(D) \leftarrow \alpha_1 \cdot q'(D) + (1 - \alpha_1) \cdot q_t(D) \tag{4.9}$$

To calculate $q'(D)$, the following equation determines the average position of D within the elites (where a value of 0 represents the first distribution in the policy and a value of 1 represents last in the policy). Note that the position depends only on the samples that contain D, written as $E(D)$:

$$q'(D) \leftarrow \frac{1}{|E(D)|} \sum_{\pi \in E(D)} index(D, \pi) \tag{4.10}$$

where $E(D)$ are the policies in E that utilise distribution D — rather,

utilise a rule from D that produced an action during policy testing — and $index(D, \pi) \in [0, 1]$ returns the normalised index of D with respect to the rules that produced actions in the policy, where 0 is first and 1 is last (if π only used one rule, $index(D, \pi)$ returns 0.5). If $E(D)$ is empty, $q'(D)$ is not updated, because if D is not present in E, we cannot determine its observed position (though $p(D)$ will decrease).

$p'(D)$ is simply calculated as the proportion of policies in E where a rule from D produced an action ($p'(D) = 0$ if no rules from D were used).

Updating Rule Distributions

The rule probabilities within the candidate rule distributions are updated in the normal CEM manner as defined by Equation 4.2 using α_1 as the step-size parameter, but because the samples in CERRLA are entire policies consisting of multiple rules, each distribution only accounts for the rules that originated from it. After the rules are updated, all rule probabilities are normalised to ensure all probabilities sum to 1.0.

4.6.4 Convergence

CERRLA's learning is considered converged when all candidate rule distributions have converged with respect to a convergence threshold β ($\beta = 0.01$ in experiments). A distribution is considered converged when either $p(D) < \beta$ or the sum KL divergence of the distribution's rules, normalised with respect to α_1, is $< \beta$:

$$\beta \geq \frac{\sum_{n=1}^{|D|} |p_{t+1,n} - p_{t,n}|}{2 \cdot \alpha_1} \tag{4.11}$$

Note that the maximum possible divergence a distribution can achieve is equal to $2 \cdot \alpha_1$. If the normalised divergence is less than β (a convergence threshold), the distribution is considered converged.

Experiments can also specify a fixed number of training episodes, such that the algorithm will continue to run and update until the fixed number is reached.

4.7 Rule Specialisation and Exploration

Initially, CERRLA starts without any rules, but it quickly learns RLGG rules so it can act within the environment. In order to learn better behaviour, the agent needs to explore more specialised rules. CERRLA's rule exploration proceeds using a 'top-down' approach, where the 'top' rules are the most general RLGG rules (in terms of the actions they cover).

4.7.1 Rule Specialisation

In CERRLA, rule specialisation only occurs when a new candidate rule distribution D is created. The rule r that seeded D and *all* possible single-step specialisations $r'_1, \ldots, r'_k$ of r (created using the specialisation operations described in Section 5.5) are added to the new distribution with a uniform probability of $1/(k+1)$. As stated in Section 5.5, each of the specialised rule's conditions are simplified and checked for illegal conditions (using the simplification rules in Section 5.4).

There is a special case for the beginning of learning: after evaluating the first policy π_0 (which only consists of RLGG rules), CERRLA creates the initial RLGG distributions of $\mathbb{D}$. As well as creating a distribution for every RLGG rule, the algorithm also creates a distribution for every 'distinct' specialisation of the RLGG rules, where distinct means the specialisation either introduces a new, non-negated predicate to the rule conditions, or the specialisation replaces an action variable for a goal variable. By maintaining one candidate rule distribution per specialisation, CERRLA is able to test every distinct subset of the actions simultaneously.

Each distinct specialisation creates a new distribution using the *branching* procedure (see following subsection) and all non-distinct specialisations are added to their respective RLGG distributions, with uniform probabilities. If the RLGG rules or specialisation conditions change, these initial distributions are recalculated, and any rules (or encompassing distributions) that are no longer valid are removed.

Example 4.7.1. In BLOCKS WORLD, for the OnG_0G_1 goal, there is only one action: *move(X, Y)*. After the agent has learned all specialisation conditions and simplification rules (Section 5.5 and 5.3), there are 17 initial candidate rule distributions, each containing an average of 15.4 rules. These include:

- the RLGG distribution (4 rules),
- one *floor(Y)* distribution (11 rules),
- one *block(Y)* distribution (20 rules),
- two *highest(*$[X, Y]$*)* distributions ($\sim$ 18.5 rules),
- four *above(*$[X, Y]$, $[G_0, G_1]$*)* distributions (17 rules),
- four *on(*$[X, Y]$, $[G_0, G_1]$*)* distributions ($\sim$ 15.5 rules),
- four action-variable replaced ($[X, Y]/[G_0, G_1]$) distributions ($\sim$ 12.5 rules).

4.7.2 Rule Exploration

CERRLA explores the set of possible rules by creating new candidate rule distributions seeded with high-valued rules in search of even higher-valued ones. The assumption is that high-valued rules will either specialise to higher valued rules, or are already the highest-valued rules.

With every CEM update, the values of a candidate rule distribution will change, generally decreasing the $KL(D)$. When a distribution's $KL(D) \leq \delta \cdot |D|$, it is ready to 'branch' into a new candidate rule distribution, where $\delta = \min\left[(depth(D) + 1)^{-1}, p(D)\right]$, representing the branching point with respect to the 'depth' of distribution D or number of branches away from the initial RLGG distribution.

A branch involves *removing* the highest probability rule r' from the distribution D and using it to 'seed' a new candidate rule distribution $D_{r'}$ with $depth(D') = depth(D) + 1$ (increasing the 'depth'), populating the new distribution with r' and all immediate specialisations of r' (using the specialisation operations described in Section 5.5). $KL(D)$ is then recomputed for D and if $KL(D) \leq \delta \cdot |D|$ it branches again (using the new highest probability rule). The one exception and stopping criterion for branching is if r' is the rule that originally seeded the distribution. In this case, the rule is not removed and no branch is made.

The resulting exploration strategy explores current candidate rule distributions and, upon determining highly effective rules within those distributions, specialises those rules in an effort to find even better rules. This

results in rule exploration focusing on rules that are frequently positively updated (present in the set of elite samples), potentially creating even better rules.

4.7.3 Rule Representation

In order to minimise evaluation time of the rules, a rule's conditions are heuristically ordered such that the number of partial matches for each condition are probabilistically minimised (the ordering may not be optimal). The ordering places the conditions with fewest likely matches at the beginning of the rule to minimise the number of matches for each following condition:

1. If condition A is not negated, and condition B is, A is before B.
2. Compare by argument types within conditions A and B. Referring to the previous subsection for the hierarchy of arguments, this heuristic orders conditions based on the proportion of argument types each condition contains. Starting with the most specific argument type (*constants*) and iteratively checking each argument type, if A has a greater proportion of the argument type than B, A is before B.
3. Compare by number of arguments, where A is before B if A has more arguments than B.
4. If condition B is a type predicate, and A is not, A is before B.
5. Otherwise, compare A and B alphabetically.

4.8 Seeding Rules

CERRLA was designed to create its own rules when it begins learning, but it can also be initialised with rules to aid the learning process. Section 4.7.2 describes how new candidate rule distributions are created: by seeding them with a rule and filling the rest of the distribution with immediate specialisations of the seed rule. We can use this technique to introduce user-provided rules into CERRLA at the beginning of learning as a performance-boosting technique. This technique can also be used to *transfer* knowledge learned from one goal within the environment to the current goal by using the rules from the output policy as seeds within the current

distribution.

Beginning a problem with no previous information is advantageous in that the algorithm has no previous biases, and demonstrates a stronger ability to learn. However, it can be difficult for an agent to learn a goal-achieving strategy when it has no initial behaviour to guide it towards the goal. One option is to take advantage of Transfer Learning (TL). This is defined as using knowledge learned in one problem and applying it to another related problem. By providing the agent with an initial 'good' strategy, it can build upon that strategy to create an ideal strategy for the current problem.

A set of rules is seeded into a CERRLA distribution by providing a file containing the JESS-compatible rules when the algorithm is initialised. Each rule in the file is used as a seed for a new candidate rule distribution D_{seed} and added to the distribution $\mathbb{D}$ (assuming the distribution does not already exist in $\mathbb{D}$). D_{seed} is initialised with $p(D_{seed}) \leftarrow 1.0$ and $q(D_{seed}) \leftarrow 0.5$ (or the existing distribution's $p(D_{seed})$ and $q(D_{seed})$ are changed). As per usual, each newly created distribution is filled with immediate specialisations of the rule and the rule itself (Section 4.7) if the agent has determined the specialisation conditions (Section 5.5). Like the RLGG rules, if the agent observations change, the specialisations for the seeded rules are recalculated and added to their respective distributions.

Although the seeded distributions have $p(D_{seed}) = 1.0$ initially, resulting in the distribution being present in every sampled policy, it is possible for $p(D_{seed})$ to decrease if rules from the distribution are not evaluated (i.e. the rule is not used for determining the agent's behaviour), so seeded rules are not guaranteed to be present in CERRLA's final output policy.

4.9 Discussion and Future Work

The Cross-Entropy Method (CEM) provides a solid base for RRL because it balances exploration and exploitation through the use of guided random sampling. The algorithm begins with no particular bias towards any given rules, but gradually exploits higher-achieving rules, eventually selecting them for specialisation to explore a particular subset of the rule space in an effort to find higher-achieving rules. A problem with the CEM is that it can converge to a solution too quickly, but this is mitigated by the branch-

ing mechanism, which removes high-probability rules and places them in their own distribution with a uniform probability, allowing the algorithm to investigate the high-probability sample separate from the other rules.

Policies are probabilistically sampled using a number of different probabilities to control which distributions are sampled, where they are placed, and what rules are sampled from them. A problem with the distribution sampling probability $p(D)$ is that it reflects how *often* a distribution is *used*, not how *important* that distribution is for achieving high reward. If the policy contains a rule that is highly effective in a given situation which randomly occurs, but is otherwise unused, then the distribution for that rule will only be updated if the policy encounters the situation during testing. This causes $p(D)$ to decrease, even though the rules contained within the distribution are highly effective. This problem is reduced by testing each policy three times, providing more opportunities for the rule to be used, but for extremely rare events, this is not enough. Furthermore, testing each policy three times reduces the variance of reward received within environments.

By restricting a distribution from updating until it has probabilistically sampled every rule at least once, the elite samples are able to represent a fair representation of the best policies. The C coefficient is the key factor that determines how quickly an agent converges to a particular strategy and how effective that strategy is. A low C results in quick convergence, but not necessarily maximally effective behaviour, whereas a high C results in slower learning, but is more likely to produce better behaviour. $C = 3$ was found to be a good value for balancing performance with speed (with a theoretical 95% testing coverage of all rules before updating).

The 'top-down' specialisation strategy forms a principled approach to creating rules as it begins with the most general valid rule for an action and iteratively produces *every* possible single-step specialisation of the rule, steering specialisation toward high-valued rules. In contrast, a Genetic Algorithm (GA) approach creates random mutations and crossovers of high-valued samples. However, a GA is able to quickly (but randomly) create complex samples, whereas the incremental specialisation is required to branch distributions with high probability samples until the complex sample is created. This is a potential problem if the stopping criteria for branching is met before the ideal child sample is able to be created; but,

because a child can be created from multiple parents, the probability of it being selected is high. Another advantage GA has is that it can remove specialisations, which if implemented in CERRLA could create loops and increase the number of rules produced, slowing down evaluation.

CERRLA is able to easily incorporate user-provided rules into the learning process by seeding new distributions with the rules at the beginning of learning. These seeded rules can be created by the user, or are the rules from a learned policy in a simpler problem within the same environment. A problem with seeding the rules into distributions is that the rules can only get more specialised; they cannot generalise. However, the original seeded rule is still part of the distribution, so if the specialisations of the rule are not useful, the agent would either learn to use the seed rule, or simply decrease the entire distribution sampling probability and ignore that seeded rule.

The learning process could potentially be made quicker by probabilistically estimating if a sample is unlikely to be added to the elites. Tak (2010) defines a method of 'cutting out' samples from the CEM during evaluation if it is unlikely that they will be added to the elite samples for the SameGame environment[2] (originally inspired from Chaslot et al. (2008)). This method was shown to reduce the training time of the algorithm by approximately 40%. The same technique could be used for CERRLA by observing the mean recorded reward at every time-step and prematurely ending any samples that have a drastically lower reward (by say, three standard deviations).

Szita and Lörincz (2006) outline a problem with the CEM in that it converges too quickly to sub-optimal policies. They provide a solution to this problem by injecting noise into the (numerical) sampling distribution, which significantly increased performance on the Tetris[3] environment. Noise could also be applied to CERRLA's distributions in an attempt to achieve better results, but the noise injection works best with a steadily decreasing noise function, which would need to relate to CERRLA's current state of distributions. An additional problem is that CERRLA's data values (rules) are discrete, not numeric, so noise would need to be applied to

[2]SameGame is an NP-hard (Kendall et al., 2008) tile-matching puzzle video-game originally developed by Kuniaki Moribe (under the name "Chain Shot!")

[3]Tetris is an NP-hard (Demaine et al., 2003) tile-matching puzzle video-game originally developed by Alexey Pajitnov.

alternative areas of the algorithm (e.g. sampling probabilities).

5

Agent Observations Model

The previous chapter defines how the Cross-Entropy Method (CEM) is used in a Relational Reinforcement Learning (RRL) context, specifically focusing on the higher-level details of how policies are generated, evaluated and updated. However, without rules to optimise, the agent has no behaviour. This chapter defines the 'agent observations' model, which is crucial in creating, maintaining and specialising rules.

The agent observations model is concerned with learning details about the environment as the agent encounters new states. These details include: observed conditions for action-related objects, implication relationships between state facts, and minimum and maximum observed values for numerical terms. The model is constantly updated as CERRLA encounters new states, though if the model observations do not change, the updates occur less frequently.

CERRLA saves each agent observations model throughout learning, such that it can be loaded and re-used when appropriate. When it is saved, observations relating to the current goal are stored separately from general environment observations such that information about the general environment is retained between experiments, even when the goal is changed.

There are three primary uses for the agent observations model: learning the RLGG rule for each action, which is the starting point for CERRLA's rule exploration (Section 5.2); creating and applying simplification rules for removing redundant and illegal conditions from rules (Section 5.3 and

5.4); and identifying the set of specialisation conditions for each action (Section 5.5).

Throughout this chapter, examples of the agent observation techniques used will often refer to the example BLOCKS WORLD state shown in Figure 5.1 as a running example. Ordinarily, the agent would not have access to the *height* observations (denoted by †), but they will be treated as non-internal predicates for this chapter to demonstrate how CERRLA handles numerical facts.

5.1 State Scanning Triggers

CERRLA begins learning the agent observations model as soon as the episode starts by scanning each state the agent encounters and extracting any relevant information that aids the learning process. Technically, this state scanning process could occur for every state, but because the process is time-consuming, it is only triggered when any one of the following conditions are met:

1. The agent's Relative Least General Generalisation (RLGG) rules do not cover all possible actions for the state. RLGG rules should produce all valid actions for the state and need to be generalised if they do not (see Section 5.2 for more details).

2. The state contains a fact composed of a predicate that the observations model has never processed. This is captured by defining basic

Relational State Observations:

block(a)	*clear(a)*	*above(a, fl)*
block(b)	*clear(c)*	*above(b, fl)*
block(c)	*clear(fl)*	*above(c, fl)*
floor(fl)	*highest(a)*	†*height(a, 2)*
thing(a)	*on(a, b)*	†*height(b, 1)*
thing(b)	*on(b, fl)*	†*height(c, 1)*
thing(c)	*on(c, fl)*	†*height(fl, 0)*
thing(fl)	*above(a, b)*	

Valid Actions:

move(a, c)	*move(a, fl)*	*move(c, a)*

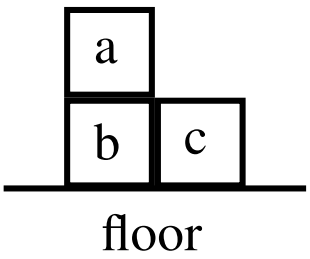


Figure 5.1: A 3-block BLOCKS WORLD state observation example. *a* is on *b* which is on the *floor*, and *c* is also on the *floor*.

rules for each unseen predicate (e.g. *edible(?)*) and triggering the state scanning process when the rule produces an output.

3. A periodic scan is triggered every 2^I steps, where $I \leftarrow 0$ initially. I is incremented by one if a scan of the state *does not* change the agent observations and reset to $I \leftarrow 0$ if a scan *does* change the agent observations.

5.2 RLGG Rule Creation

The RLGG rules are a set of rules, one for each action in the environment, that define the least-general generalisation conditions required for taking an action — that is, whenever an action is available for the agent to take, the conditions of the corresponding RLGG rule θ-subsume the facts of the state and produce the same action, while simultaneously being as specific as possible. The RLGG technique described here is a simplified version of the original RLGG algorithm Plotkin (1970), as it only uses facts directly related to the action (containing one or more of the same terms in the action atom) as input to the process, rather than incorporating every fact in the state.

Determining the RLGG was originally devised by Plotkin (1970) as an ILP method to determine a minimally general clause that represents two clauses; that is, to create a clause that is able to represent both clauses with minimal generalisation (not just creating an empty clause, which simply states all facts are true). A problem with the original RLGG algorithm proposed by Plotkin (1970) is that it creates a large number of redundant facts describing every potential merging of clauses and can even create an infinite number of RLGGs. Sammut (1998) defines a constrained RLGG algorithm that reduces the number of possible RLGGs by utilising the background knowledge to limit the merging possibilities. The RLGG algorithm has also been utilised as the basis of the GOLEM and PROGOL algorithms (Muggleton and Feng, 1992; Muggleton, 1995), which use the RLGG algorithm to efficiently create clauses for defining positive examples within a data set. RLGG rules are useful in the RRL context because they present the minimally-general conditions needed to take an action within the environment, resulting in a rule that clearly defines an action's preconditions.

The RLGG algorithm presented in the following subsection is much simpler than the one presented in Plotkin (1970) and Sammut (1998), because it only records the terms used in the action (represented by X, Y, ...) and numerical values (represented by range variables N_i). All other terms are replaced by anonymous variables. This results in a more general RLGG than the standard RLGG, losing some information, but it also reduces the set of possible specialisation conditions to those that are directly related to the action, reducing the search space of rules. The key assumption behind this decision is that actions specify important objects as their terms; all other objects are unimportant to making informed decisions. This is discussed further in Section 7.3.

The RLGG for an action is calculated as the lgg (least general generalisation) of the set of facts in the state. Ordinarily, the RLGG process incorporates background knowledge into the process, but because all facts of a state are present (via forward chaining), there is no need to utilise background knowledge to infer new knowledge. The lgg operation is defined by the following rules (Lavrac and Dzeroski, 1993):

$$lgg(t,t) = t \tag{5.1}$$

$$lgg(s,t) = V, \text{ where } s \neq t \tag{5.2}$$

$$lgg(s,?) = ? \tag{5.3}$$

$$lgg(p(t_1,\ldots,t_n), p(s_1,\ldots,s_n)) = p(lgg(t_1,s_1),\ldots,lgg(t_n,s_n)) \tag{5.4}$$

$$lgg(p(t_1,\ldots,t_n), q(s_1,\ldots,s_m)) \text{ is undefined if } p \neq q \tag{5.5}$$

$$lgg(\{L_1,\ldots,L_n\},\{K_1,\ldots,K_n\}) = \{L_{ij} = lgg(L_i,K_j) \text{ if defined}\} \tag{5.6}$$

Note that the lgg of an anonymous variable '?' remains anonymous (Equation 5.3).

The RLGG uses the facts related to the current action as input to the procedure. Given a state s and a set of valid actions $A(s) = \{a_1,\ldots,a_n\} : a_i = p_{a,i}(c_{i,1}, c_{i,2}, \ldots)$, the set of state facts directly related to an action a_i is defined as $rel(s,a_i)$:

$$rel(s,a_i) = \{p_{s,j}(c_1,\ldots,c_n) \in s \mid \exists c_k \big(p_{s,j}(\ldots,c_k,\ldots) \wedge p_{a,i}(\ldots,c_k,\ldots)\big)\} \tag{5.7}$$

which states that the related facts $rel(s,a_i)$ are all facts in state s that contain

at least one term that action a_i contains.

These related facts are used as the conditions for the rule $r_{a_i} = rel(s, a_i) \rightarrow a_i$, which we use to update the RLGG rule for action predicate p_a:

$$r^{p_a}_{RLGG,t} = lgg\left(r^{a}_{RLGG,t-1}, r_{a_i}\theta^{-1}_{a_i}\right) \tag{5.8}$$

where $r^{p_a}_{RLGG,t-1}$ is the existing RLGG rule for action predicate p_a and $r^{a}_{RLGG,t}$ is the updated rule (if there is no existing rule, $r^{p_a}_{RLGG,t} \leftarrow r_{a_i}\theta^{-1}_{a_i}$). The RLGG of the two rules uses a lossy inverse substitution defined by the current terms of the atomic action a_i, such that $\theta^{-1}_{a_i} = \{c_{i,1}/X, c_{i,2}/Y, \ldots\}$. Any non-numerical constants not included in $\theta^{-1}_{a_i}$ are replaced by the anonymous variable '?'; numerical constants are replaced by unique range variables N_j (these can later be constrained to be within a given numerical range, see Section 5.5). The resulting rule encodes a rough approximation (due to lossy inverse substitution) of the least general set of conditions required for taking action p_a.

Example 5.2.1. Referring to Figure 5.1 (replicated in Figure 5.2), the RLGG calculation process for the three valid actions *move(a, c)*, *move(a, fl)*, *move(c, a)* is described in the following example, processing one rule at a time (beginning with $t = 1$):

$r_{move(a,\ c)}$ = *block(a), block(c), thing(a), thing(c), clear(a), clear(c), on(a, b), on(c, fl), above(a, b), above(a, fl), above(c, fl), height(a, 2), height(c, 1)* $\rightarrow$ *move(a, c)*

$\theta^{-1}_{move(a,\ c)} = \{a/X, c/Y\}$

$r^{move}_{RLGG,1}$ = *block(X), block(Y), thing(X), thing(Y), clear(X), clear(Y), on(X, ?), on(Y, ?), above(X, ?), above(Y, ?), height(X, N_0), height(Y, N_1)* $\rightarrow$ *move(X,*

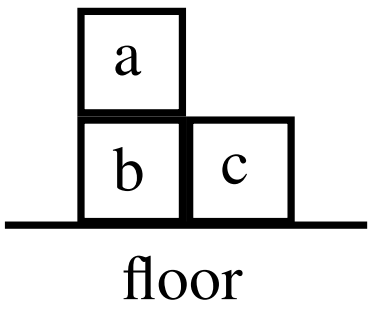


Figure 5.2: An example 3-block BLOCKS WORLD state also given in Figure 5.1.

Y)

This is already very close to the actual RLGG; only the conditions *block(Y)*, *on(Y, ?)*, and *above(Y, ?)* are not always true, as evidenced in the following example:

$r_{move(a,\,fl)}$ = *block(a)*, *floor(fl)*, *thing(a)*, *thing(fl)*, *clear(a)*, *clear(fl)*, *on(a, b)*, *on(b, fl)*, *on(c, fl)*, *above(a, b)*, *above(a, fl)*, *above(b, fl)*, *above(c, fl)*, *height(a, 2)*, *height(fl, 0)* $\rightarrow$ *move(a, fl)*

$\theta^{-1}_{move(a,\,fl)} = \{a/X, fl/Y\}$

$r^{move}_{RLGG,2}$ = *block(X)*, *thing(X)*, *thing(Y)*, *clear(X)*, *clear(Y)*, *on(X, ?)*, *above(X, ?)*, *height(X, N_0)*, *height(Y, N_1)* $\rightarrow$ *move(X, Y)*

This rule is in fact the RLGG for the Blocks World *move* action, so there is no need to describe the process for the final action of the state (as the rule canot generalise any further). Many of the conditions in this rule are redundant with respect to other facts though (e.g. *on(X, ?)* is always true if *above(X, ?)* is true) and can be removed using the simplification rules described in the following section. The simplified rule is:

$$r^{move}_{\text{RLGG},\,2} = \textit{clear(X)}, \textit{clear(Y)}, \textit{block(X)} \rightarrow \textit{move(X, Y)}$$

All other conditions in the rule are implied from these three conditions.

5.3 Inferring Simplification Rules

Simplification rules are basic rules defining causal or correlated relationships between patterns of literals, inferred from the environment state observations. The simplification rules can be applied to Cerrla's condition-action rules to remove redundant and illegal conditions. Simplification rules define either implication (causation) or equivalence (correlation) relationships between both negated and non-negated literals. They are created in a manner similar to learning the RLGG rules (Section 5.2), but observe the relationships between state facts, rather than the action's relationship to state facts. Whenever Cerrla creates a new rule, the simplification rules are immediately applied to the rule to remove redundant conditions or mark the rule as illegal. This reduces the number of possible rules created during rule specialisation and minimises the number of conditions

required for each rule.

Creating implication and equivalence simplification rules is achieved by identifying causal relationships between state observation facts, and for equivalence rules, checking if these relationships are symmetric. These relationships are discovered by CERRLA using the RLGG method defined in the previous section to identify patterns in the state observations.

5.3.1 Identifying Causal Relationships

Given a state $s = \{x_1, \ldots, x_n\}$ (such that $x_i = p_{s,i}(c_{i,1}, c_{i,2}, \ldots)$), CERRLA infers a set of implication rules by identifying which related literals are always true when another given literal is true. This process is nearly identical to the RLGG rule learning process, except instead of identifying the RLGG for each action predicate p_a (e.g. *move*), the RLGG is learned for every state predicate $p_{s,i}$ (e.g. *block*, *clear*, *on*, ...).

For every $x_i \in s$, the set of *always true* literals are calculated by first identifying the related facts $rel(s, x_i)$:

$$rel(s, x_i) = \{p_{s,j}(c_1, \ldots, c_n) \in s \mid \exists c_k \big(p_{s,j}(\ldots, c_k, \ldots) \wedge p_{s,i}(\ldots, c_k, \ldots)\big)\} \tag{5.9}$$

which states that the related facts $rel(s, a_i)$ are all facts in state s that contain at least one term also contained in fact x_i (note that x_i is also an element of $rel(s, x_i)$.

These related facts are used to update the set of *always true* literals for state predicate $p_{s,i}$ at time t, written as $\mathbf{T}_{p_{s,i},t}$ (using the same lgg definitions seen in Section 5.2):

$$\mathbf{T}_{p_{s,i},t} = lgg\big(\mathbf{T}_{p_{s,i},t-1}, rel(s, x_i)\theta_{x_i}^{-1}\big) \tag{5.10}$$

This RLGG process is slightly different from the RLGG rule creation process in that it does not create a rule from the related conditions. Here, $\theta_{x_i}^{-1}$ is used as the inverse substitution map, such that when it is applied to $rel(s, x_i)$, all terms in x_i are replaced by variables and all terms not in x_i are replaced by anonymous variables. If $t = 1$, $\mathbf{T}_{p_{s,i},1}$ simply becomes the inversely-substituted related literals $rel(s, x_i)\theta_{x_i}^{-1}$.

When simplification rules are applied to rules, anonymous variables are a special case: they may *only* bind to other anonymous variables. This is

a necessary modification to match with the rule's use of the anonymous variable, i.e. terms that are not relevant to the rule's action. E.g. when applying a simplification rule, the literal *above(X, ?)* will not θ-subsume the rule condition *above(Y, X)* or *above(?, X)*, but will θ-subsume *above(Y, ?)* (where $\theta = \{X/Y\}$). Further details about applying the simplification rules can be found in Section 5.4.

The resulting set of RLGG conditions for each state predicate $p_{s,i}$ encode the abstract, variable-term literals that are true whenever a literal with predicate $p_{s,i}$ is true. E.g. in the BLOCKS WORLD example, whenever a literal matching *on(X, Y)* is true, the literal *above(X, Y)* is also true (where X and Y are replaced by their respective constants).

Example 5.3.1. The set of *always true* literals for state predicate *block* are:

$$\mathbf{T}_{block} = \textit{above(X, ?)}, \textit{block(X)}, \textit{on(X, ?)}, \textit{thing(X)}$$

That is, whenever *block(X)* is true, these literals are also true, where X is substituted by some term.

Negated Relationships: Existence Implies Non-Existence

But what about relationships between true and false facts? Sometimes, whenever a fact is true, another fact is always false (e.g. if *floor(X)* is true, *block(X)* is never true, where X is the same object). Identifying these relationships is a little more tricky. For a state fact x_i, the set of related state facts can be identified by $rel(s, x_i)$ (Equation 5.9). After applying the inverse substitution operator $\theta_{x_i}^{-1}$ to $rel(s, x_i)$, the related facts are transformed into a set of abstract literals using terms from the finite alphabet $\Gamma(x_i) = \{V_1, \ldots, V_n, ?\}$, where each V_i is a different variable present in $\theta_{x_i}^{-1}$ and n is number of unique terms in x_i.

This alphabet of terms $\Gamma(x_i)$ and the state predicates of the environment $P_s = \{p_{s,1}, \ldots, p_{s,n}\}$ can be combined together to produce the set of all possible atoms that are related to x_i (the Herbrand base of $\Gamma(x_i)$ and P_s: $HB(\Gamma(x_i), P_s)$). Note that literals only containing anonymous variables are not included in $HB(\Gamma(x_i), P_s)$. The type constraints for fact x_i can be used to restrict the number of atoms formed by ensuring that each variable in $\Gamma(x_i)$ is only used in predicates where the original type of the variable could be true. For example, let $x_i =$ *floor(fl)*, resulting in $\Gamma(x_i) = \{X, ?\}$.

The Herbrand base $HB(\{X, ?\}, P_s)$ would not include *block(X)*, as X could not possibly be a valid term for *block* with the type restricted to $floor$ or any types implied by $floor$ (i.e. *thing*).

Using $HB(\Gamma(x_i), P_s)$ and $rel(s, x_i)\theta_{x_i}^{-1}$, the set of possibly-related literals (restricted to literals containing at least one of x_i's terms) that are *always false* for the current state can be calculated as:

$$\neg rel(s, x_i) = HB(\Gamma(x_i), P_s) \setminus (rel(s, x_i)\theta_{x_i}^{-1}) \tag{5.11}$$

Like the related facts, these untrue related literals $\neg rel(s, x_i) = \{\overline{L_0}, \ldots, \overline{L_n}\}$ (the line indicates the literals are not true) can be used as input to the RLGG method to produce a set of literals that are *always false* $\mathbf{F}_{p_{s,i},t}$ for the state predicate $p_{s,i}$.

Example 5.3.2. The set of *always false* literals for state predicate *block* are:

$$\mathbf{F}_{block} = \textit{above(X, X)}, \textit{floor(X)}, \textit{on(X, X)}$$

That is, whenever *block(X)* is true, these literals are always false, where X is substituted by some term.

Negated Relationships: Non-Existence Implies Existence

Just as the existence of a fact implies the non-existence of another fact, CERRLA can also calculate which facts are implied to be true when another fact is explicitly false. That is, when a rule explicitly has the negation of an atom as a condition, which literals are always true when the negated atom is false?

The previous subsection described how the set of untrue related literals $\neg rel(s, x_i) = \{\overline{L_0}, \ldots, \overline{L_n}\}$ can be calculated using the known related facts $rel(s, x_i)$, and the Herbrand base of variable term state atoms $HB(\Gamma(x_i), P_s)$. From this information, CERRLA can infer a set of *always true* literals for every untrue related literal $\overline{L_j} \in \neg rel(s, x_i)$ using the same RLGG process seen in Equation 5.10 with one caveat: the set of related facts can only contain literals containing terms present in L_j. Note that separate sets of *always true* literals are maintained for predicates with different terms because they may include anonymous variables, altering how the simplification rule can

be applied to a policy-rule's conditions:

$$\mathbf{T}_{\overline{L_j},t} = lgg\left(\mathbf{T}_{\overline{L_j},t-1}, rel(s, x_i)\theta^{-1}_{\overline{L_j}}\right) \tag{5.12}$$

where $\theta^{-1}_{\overline{L_j}}$ is an inverse substitution map containing only the substitutions in $\theta^{-1}_{x_i}$ which replace a constant with a variable present in $\overline{L_j}$. The resulting set of *always true* facts represent the literals that are always true when $\overline{L_j}$ is untrue (e.g. no fact with a substitution for the variables in L_j is present in the state).

These 'non-existence implies existence' simplification rules can only be applied to clauses that explicitly state if a literal is false (negated). It cannot be used to infer the existence of other literals with the *closed world assumption*: "any facts not asserted as true are assumed false," e.g. the representation used to represent environment states.

Example 5.3.3. In the Blocks World environment, the *always true* literals for *not on(X, ?)*:

$$\mathbf{T}_{not\ on(X,\ ?)} = above(?,\ X),\ floor(X),\ on(?,\ X),\ clear(X),\ thing(X)$$

This encodes the set of facts that are true if *not on(X, ?)* is true (there is no substitution for X such that it is *on* some anonymous, non-action-related object). *not on(X, ?)* is actually equivalent to the literal *floor(X)* and this relationship can be utilised to replace occurrences of *not on(X, ?)* with *floor(X)* (Section 5.3.3).

Pairwise Relationships

Sometimes relationships between literals are more complex than a one-to-one causal relationship. With a small addition to the previous methods, Cerrla can also observe causal relationships for pairs of literals. While this could be extended to observe triplicate causal relationships and beyond, the theoretical benefit would not be worth the associated cost of learning the relationships.

Given a fact $x_i \in s : x_i = p_{s,i}(c_{i,1}, c_{i,2}, \ldots)$ and its inversely substituted related literals $rel(s, x_i)\theta^{-1}_{x_i} = \{p_{s,j}(V_0, \ldots, V_n), \ldots\}$, a set of *always true* literals and *always false* literals can be calculated for every *pair* of literals $(x_i \wedge p_{s,j}(V_0, \ldots, V_n))$. The procedure for doing so uses the same RLGG processes

(Equation 5.10 and 5.11), but the always true/false sets $\mathbf{T}_{p_{s,i} \wedge p_{s,j}(V_0,\ldots,V_n),t}$ / $\mathbf{F}_{p_{s,i} \wedge p_{s,j}(V_0,\ldots,V_n),t}$ are unique to the pair (where different terms for $p_{s,j}$ use different always true/false sets).

Example 5.3.4. The pairwise set of always true literals for *clear(X)* and *on(?, X)* is:

$$\mathbf{T}_{clear \wedge on(?,\ X)} = above(?,\ X), floor(X), on(?,\ X), clear(X), thing(X)$$

This encodes the relationship that states if X is *clear* and an anonymous, non-action-related object is *on* X, then X is the $floor$ (only the floor can be clear when something is *on* it). This relationship could not be encoded using only singular relationships between literals.

5.3.2 Creating Implication Rules

Given the always true/false sets of literals, creating implication rules is straightforward. For every state predicate p_s, a simplification rule is created for each *always true* literal in $\mathbf{T}_{p_s}$ in the form $p_s(X, Y, \ldots) \Rightarrow T$, where T is one of the literals in $\mathbf{T}_{p_s}$ (an implication rule is not created if $p_s(X, Y, \ldots) = T$). Similarly, for every *always false* literal in $\mathbf{F}_{p_s}$, a simplification rule is created as $p_s(X, Y, \ldots) \Rightarrow not\ F$, where F is one of the literals in $\mathbf{F}_{p_s}$. Pairwise simplification rules are created in a similar manner, except the pair of conditions are on the LHS of the rule.

The rules are interpreted by simplifying to the LHS of the rule, such that if a set of literals $\{A, B, C, \ldots\}$ is simplified with the simplification rule $A \Rightarrow B$, the resulting simplified set of literals is $\{A, C, \ldots\}$, because B is redundant when A is present (logical resolution, Robinson (1965)). Furthermore, illegal sets of conditions can be identified by checking for a negated post-condition. E.g. given the set $\{A, \neg B, \ldots\}$, the rule $A \Rightarrow B$ identifies this set as an illegal set.

Example 5.3.5. The relationships described in Example 5.3.1, 5.3.2, and 5.3.4 produce the following simplification rules:

block(X) $\Rightarrow$ *above(X, ?)*,

block(X) $\Rightarrow$ *thing(X)*,

block(X) $\Rightarrow$ *not above(X, X)*,

block(X) $\Rightarrow$ *not floor(X)*,

clear(X), on(?, X) ⇒ above(?, X),

clear(X), on(?, X) ⇒ floor(X)

5.3.3 Creating Equivalence Rules

Implication rules are useful for removing redundant conditions, but even stronger equivalence rules $A \Leftrightarrow B$ can be created if fully correlated combinations of conditions exist. Equivalence rules are used for simplification by *replacing* any occurrence of the right-side fact B for the left-side fact A.

During the simplification rule creation process, an equivalence rule $A \Leftrightarrow B$ is created instead of an implication rule $A \Rightarrow B$ only if $A \Rightarrow B \wedge B\theta_B \Rightarrow A\theta_B$. A and B represent one or more literals, and θ_B is a substitution map used to ensure the RHS of the simplification rule does not contain any variables not found in the LHS of the rule. θ_B is defined to replace every non-anonymous term in B with a variable $(X, Y, \ldots)$. All other terms not substituted by θ_B are replaced by the anonymous variable.

If $B\theta_B \Rightarrow A\theta_B$ exists, an equivalence rule $A \Leftrightarrow B$ or $B\theta_B \Leftrightarrow A\theta_B$ can be created, where the preferred equivalence rule is selected using the descending preference list for the LHS literal(s): {type facts, relation facts, negated facts} (where facts with fewer terms, and alphabetical comparison are used for breaking ties).

Example 5.3.6. The implication rule *on(X, ?) ⇒ block(X)* can be replaced with the equivalence rule *block(X) ⇔ on(X, ?)* because *on(X, Y) ⇒ block(X)* and *block(X)* is the preferred LHS because it is a type literal.

Pairwise equivalence rules can also be created in a similar manner to singular fact equivalence rules, except the implication rules are checking for two conditions, i.e. reversing the implication $A \wedge B \Rightarrow C$ results in a check for $C\theta_C \Rightarrow A\theta_C \wedge C\theta_C \Rightarrow B\theta_C$. If both A and B are implied by C, the equivalence rule $C \Leftrightarrow A \wedge B$ is created (the LHS is always the singular condition).

Example 5.3.7. The pairwise implication rule *clear(X), on(?, X) ⇒ floor(X)* (from Example 5.3.5) can be replaced with the equivalence rule *floor(Y) ⇔ clear(Y) ∧ on(X, Y)* because *floor(X) ⇒ on(?, X)* and *floor(X) ⇒ clear(X)*.

5.3.4 Recording Simplification Rules

It is not necessary for the agent to record every implication or equivalence rule created. Because equivalence rules replace facts on the right-side of the rule whenever they are encountered, any other rules containing those facts will never trigger (assuming the equivalence rules are evaluated first, which is the case, see Section 5.4). Furthermore, if two equivalence rules have the same condition on the right-side, which one triggers? To resolve these issues, when a simplification rule is created, the following steps are checked:

1. If the rule is an implication rule $A \Rightarrow B$, it is only added if there are no existing equivalence rules $D \Leftrightarrow B$ (the same B) or $E \Leftrightarrow A$ (the same A).
2. If the rule is an equivalence rule $B \Leftrightarrow C$, it is not added if there is an existing equivalence rule $A \Leftrightarrow B$ (as B is simplified to A).
3. If there is an existing equivalence rule $C \Leftrightarrow B$ (same right-side condition), the equivalence rule with the simplest conditions is kept (refer to the descending preference list in the prior subsection).
4. When an equivalence rule $A \Leftrightarrow B$ is added, any existing implication rules mentioning B are removed.

The inferred simplification rules for each environment can be found online at http://www.samsarjant.com/cerrla/.

5.4 Evaluating Simplification Rules

When a rule is created, CERRLA's inferred simplification rules are applied to the rule's conditions to remove redundant predicates, producing a semantically equivalent simplified set of conditions (the original conditions are also recorded for use in further rule specialisations, see Section 5.5).

Simplification rules are applied by checking whether a simplification rule θ-subsumes the rule conditions and if so, removes the redundant facts defined by the rule. CERRLA already uses an efficient method for checking if a rule's conditions match a set of facts: the Rete algorithm employed by JESS. By using a Rete network to represent the simplification rules, CERRLA can

efficiently determine matches to the simplification rules and automatically remove redundant conditions by treating a rule's conditions as asserted atoms. However, because the set of rule conditions contain variable terms and negated literals, simplifying them requires pre- and post-processing to assert them as facts to the state. Before describing how the Rete simplification network is created, this section first describes the special transformation process that the rule conditions are subject to in order for them to be simplified.

5.4.1 Transforming the Rule Conditions

Because the rule conditions contain variables, they cannot be asserted directly to the Rete simplification network. Each condition is preprocessed to replace terms with constant terms, so the simplification rules can be applied to them. For every condition in the rule, the following preprocessing steps are performed:

1. If the condition is negated, the name of the condition is prefixed by '*neg_*' and the condition is treated as not negated. This is because negated facts cannot be asserted as facts to the Rete simplification network. E.g. *not clear(...)* becomes *neg_clear(...)*.

2. If a condition term is not present in the rule's action literal (e.g. anonymous variables), then it is replaced with the constant *free*. When simplification rules are evaluated, free variables in the conditions can *only* match with anonymous variables in the simplification rule. E.g. *clear(?)* would be represented as *clear(free)*.

3. Otherwise, the term is replaced with a unique identifier constant *id#*, where # represents some number. This replacement is recorded in a replacement map θ^*, such that any other occurrences of the term are replaced by the same identifier constant. This allows the simplification rules to treat variable terms in the rule conditions as constants to be matched against variables in the simplification rules. E.g. *above(X, fl)* would be represented as *above(id0, id1)*, where $\theta^* = \{X/id0, fl/id1\}$.

Example 5.4.1. Transforming the conditions *clear(X)*, *clear(Y)*, *height(Y, N_0)*, *not on(X, ?)* produces the following:

$$clear(id0), clear(id1), height(id1, id2), neg\_on(id0, free)$$

where $\theta^* = \{X/id0, Y/id1, N_0/id2\}$.

5.4.2 Asserting the Simplification Rules

Creating the Rete network of simplification rules (separate to the network representing the environment state) involves defining each simplification rule in a JESS-compatible format, such that redundant conditions are automatically removed and illegal condition combinations are identified. Each simplification rule (implication or equivalence) asserts two separate rules to the simplification Rete network: the redundant condition removal rule and the illegal state rule. An illegal state is identified by negating the non-preferred (RHS) of the rule and evaluating it. If the negated rule matches the set of conditions, then a new fact *illegal()* is asserted to the Rete network, representing an illegal condition combination.

When a simplification rule is evaluated, anonymous variables in the rule can *only* match with anonymous terms. To enforce this restriction, when the rule is asserted to the simplification Rete network all anonymous variables are replaced with the constant *free*. Because the rule conditions also perform the same replacement, anonymous variables in simplification rules will only match with corresponding *free* constants in rules.

When the conditions of a simplification rule are asserted, negated conditions are also prefixed with '*neg_*' (as per step 1 in the previous subsection). The JESS-compatible rules for each type of simplification rules are defined below.

Each simplification rule uses the special *assert* and *retract* operators (previously defined in Section 3.2, under *Transition Rules*) which add/remove facts to/from the set of asserted facts.

Implication Rules

An implication rule $X \Rightarrow Y$ is asserted as:

$$X, Y \rightarrow retract(Y)$$

This rule checks for a match to both X and Y of the simplification rule and if it exists, removes the condition(s) Y represents.

Example 5.4.2. The implication rule *above(X, Y)* $\Rightarrow$ *not highest(Y)* is asserted as:

$$above(X, Y), neg\_highest(Y) \rightarrow retract(neg\_highest(Y))$$

Equivalence Rules

An equivalence rule $X \Leftrightarrow Y$ is asserted as:

$$Y \rightarrow retract(Y), assert(X)$$

This rule checks for a match to Y of the simplification rule and if it exists, removes the condition(s) Y represents and asserts the condition(s) in X.

Example 5.4.3. The equivalence rule *floor(Y)* $\Leftrightarrow$ *clear(Y)* $\wedge$ *above(X, Y)* is asserted as:

$$clear(Y), above(X, Y) \rightarrow retract(clear(Y)), retract(above(X, Y)), assert(floor(Y))$$

Illegal State Rules

Illegal state rules are created for both implication and equivalence rules to check if the illegal state of both X and $\neg Y$ exist. Note that if Y contains multiple conditions, they are represented in disjunctive format (via De Morgan's laws). Also, any double negative conditions are simplified to a non-negated condition. As usual, negated conditions are represented with the *neg_* prefix.

$$X, not\ Y \rightarrow assert(illegal())$$

This rule checks for a match to X and negated Y of the simplification rule and if it exists, asserts the *(illegal)* fact to the Rete network, indicating the condition combination represents an illegal state.

Example 5.4.4. The two rules from Example 5.4.2 and 5.4.3 also assert the following illegal state rules respectively:

$$above(X, Y), highest(Y) \rightarrow assert(illegal())$$

$$floor(Y), neg\_clear(Y) \vee neg\_above(X, Y) \rightarrow assert(illegal())$$

5.4.3 Recreating the Rule Conditions

After the simplification rules have been run, unless the conditions are *illegal()*, the rule conditions are extracted from the network and recreated.

This is accomplished by simply applying the formatting steps in reverse, using the inverse of replacement map θ^* to replace all unique identifier constants with their original terms. *free* terms are replaced with anonymous variables. Any facts prefixed by *neg_* are returned to their original fact form and negated.

If a rule's conditions are *illegal*, the conditions are not recreated and the rule is marked as illegal. Illegal rules are not used by CERRLA and removed from the specialisation process.

5.5 Rule Specialisation

The previous sections defined how the agent organises the information observed in the environment. This section defines how that information is used to specialise rules. When a rule r is specialised, *all* possible single-step specialisations $\{r'_1, \ldots, r'_i\}$ are created using the following specialisation operations.

Example 5.5.1. The examples in this section use the simplified BLOCKS WORLD *move* RLGG rule (first seen in Example 5.2.1)

$$r^{move}_{RLGG} = clear(X), clear(Y), block(X) \rightarrow move(X, Y)$$

After every specialised rule is created, the conditions of the rule are simplified (Section 5.4) by removing redundant conditions and checking if the conditions are illegal. In the latter case, the rule will not be added to the set of specialisations. If the specialised and simplified rule differs from the rule that created it, it is added to the set of specialisations.

5.5.1 Additive Specialisation

Additive specialisation involves specialising a rule by adding more conditions to it. The set of specialisations for each action is identified while creating the RLGG rule; they are the inversely substituted conditions that *are not* RLGG rule conditions (Section 5.2). Additive specialisation only uses conditions it has observed to be true to ensure that specialisation only adds feasibly possible conditions to the rules (though combinations of conditions may result in illegal rules anyway, but this is detected using the simplification rules, Section 5.4).

Recall that for every valid action, a temporary rule consisting of related conditions $r_{a_i} = L_1, \ldots, L_n \rightarrow a_i$ (where $L_1, \ldots, L_n$ are literals representing a_i's related conditions as per $rel(s, a_i)$) and a lossy inverse substitution $\theta_{a_i}^{-1}$ were used to update the conditions of the RLGG rule $r^a_{RLGG,t} = M_1, \ldots, M_n \rightarrow p_a(X, Y, \ldots)$ (where $M_1, \ldots, M_n$ are literals representing the RLGG conditions at time t). After updating the RLGG rule with action a_i, the set of specialisation literals for rules with action predicate a is iteratively updated as:

$$specs_{a,t} \leftarrow specs_{a,t-1} \cup \left((\{L_1, \ldots, L_n\}\theta_{a_i}^{-1}) \setminus \{M_1, \ldots, M_n\} \right) \tag{5.13}$$

Each literal in $specs_{a,t}$ represents two possible specialisations: the negated and non-negated versions of each.

The specialisation conditions can also include specialisation conditions that directly relate to the environment goal. By expanding the inverse substitution map $\theta_{a_i}^{-1}$ to include inverse substitutions for every substitution in the goal substitution map $\theta_G = \{G_i/c_i\}$, relationships between the action's terms and the goal terms can be encoded as potential specialisation conditions. This expanded inverse substitution map $\theta_{a_i,G}^{-1} = \{c_1/X, c_2/Y, \ldots, c_i/G_0, c_{i+1}/G_1, \ldots\}$ is applied in the same way as $\theta_{a_i}^{-1}$ in Equation 5.13 to add specialisation conditions concerning goal and action terms to the set of specialisation conditions. However, there is one restriction: added literals must contain at least one variable present in the rule's action (i.e. $X, Y, \ldots$).

Additive specialisations are applied to a rule r by creating a new rule r' for every negated and non-negated literal in $specs_{a,t}$ (where a is the same action predicate used in r) with the literal added to the conditions. If, after simplifying the rule, the conditions of r' differ from r, and r' is not illegal according to the simplification rules, it is added to the set of specialisations for r.

Note that the rule specialisation conditions *only* include conditions that contain at least one mention of the terms used in the action. This focuses rule specialisation towards defining the conditions for the directly relevant objects to the rule's action. The assumption is that the terms of a rule's action are the most relevant objects for performing the action. If specialisation conditions that do not explicitly reference the action's terms are also included as specialisation operators, the number of possible special-

isations increases dramatically and results in many specialised rules with conditions of little utility. Further discussion is given in Section 5.6.

Example 5.5.2. Some example rules created by adding conditions to r^{move}_{RLGG} (Example 5.5.1). Note that each rule has been simplified using the simplification rules to remove redundant literals (evident for r^{move}_1 and r^{move}_3):

$r^{move}_1 = \mathit{clear(X)}, \mathit{block(X)}, \mathit{floor(Y)} \rightarrow \mathit{move(X, Y)}$

$r^{move}_2 = \mathit{clear(X)}, \mathit{clear(Y)}, \mathit{block(X)}, \mathit{not\ highest(X)} \rightarrow \mathit{move(X, Y)}$

$r^{move}_3 = \mathit{above(X, G_0)}, \mathit{clear(X)}, \mathit{clear(Y)} \rightarrow \mathit{move(X, Y)}$

5.5.2 Transforming Specialisation

Specialisation can also occur by modifying the existing conditions. There are two methods of doing this: range splitting and goal term replacement.

Range Splitting

Range splitting creates specialised rules by splitting an existing range (or a variable representing a number) into up to five overlapping sub-ranges: the *lower half*, the *upper half*, a *central half*, and if applicable, a negative sub-range (*lower bound* to 0), and a positive sub-range (0 to *upper bound*). As part of the agent observations model, the observed values for each numerical term are recorded with respect to every action predicate, using the inverse substitution map $\theta^{-1}_{a_i}$ (defined in Section 5.2) to convert the literal containing the numerical term into a variable format key for accessing the range. That is, each range is stored within a map of the form $(L_j\theta^{-1}_{a_i}) \mapsto [N^{min}_j, N^{max}_j]$, where N^{min}_j and N^{max}_j represent the real-valued minimum and maximum observed values for the range respectively.

The observed values for each range are recorded because it allows ranges to be specified as variable fractions of the observed range instead of fixed numerical values, which means that if the observed values change, the variable fractions still represent an identical subset of the range. The bounds of a range can be expressed either as a real number (i.e. when 0 is a bound), an observed range bound N^{min}_j or N^{max}_j, or as a linearly interpolated value between N^{min}_j and N^{max}_j, written as $lerp(N^{min}_j, N^{max}_j, \mathbb{R} \in [0, 1])$. This func-

tion is interpreted as:

$$lerp(N^{min}, N^{max}, \alpha) = (1 - \alpha) \cdot N^{min} + \alpha \cdot N^{min} \tag{5.14}$$

Example 5.5.3. The ranges produced by splitting N_1 in r^{move}_{RLGG} (Example 5.5.1):

$r_1^{move} = clear(X), clear(Y), height(Y, (N_1^{min} \leq N_1 \leq lerp(N_1^{min}, N_1^{max}, 0.5))), block(X) \rightarrow move(X, Y)$

$r_2^{move} = clear(X), clear(Y), height(Y, (lerp(N_1^{min}, N_1^{max}, 0.25) \leq N_1 \leq lerp(N_1^{min}, N_1^{max}, 0.75))), block(X) \rightarrow move(X, Y)$

$r_3^{move} = clear(X), clear(Y), height(Y, (lerp(N_1^{min}, N_1^{max}, 0.5) \leq N_1 \leq N_1^{max})), block(X) \rightarrow move(X, Y)$

In Chapter 6, all ranges are provided with numerical values so the reader does not need to refer to observed range values for an environment.

Goal Argument Replacement

Goal term replacement involves substituting all occurrences of one of the variables in the rule's action with a goal variable. For every variable in the rule's action atom $(X, Y, \ldots)$ and every current goal term $(G_0, G_1, \ldots \in \theta_G$, the current goal substitution map), a substitution θ_H is applied to the rule r, such that one of the variables in the rule is replaced by one of the goal variables to create a new rule r'. To ensure the rule is legal (i.e. the replacement has not created impossible conditions), every condition in the rule that contains a goal term is checked to see if it exists in the agent's goal term observations, which observe every different literal and term position each goal variable has been present in.

For every state s, the set of observed goal term positions is updated by adding $\theta_G s$ to the set, where all non-goal related terms are replaced by anonymous variables. If every condition in $\theta_G r'$ is in the set of observed goal term positions, the rule is considered legal.

Example 5.5.4. The specialisations of r^{move}_{RLGG} (Example 5.2.1) produced by goal term replacement for the OnG_0G_1 goal are:

$r_1^{move} = clear(G_0), clear(Y), block(G_0), height(G_0, N_0), height(Y, N_1) \rightarrow move(G_0, Y)$

$r_2^{move} = clear(X), clear(G_0), block(X), height(X, N_0), height(G_0, N_1) \rightarrow move(X, G_0)$

$r_3^{move} = clear(G_1), clear(Y), block(G_1), height(G_1, N_0), height(Y, N_1) \rightarrow move(G_1, Y)$

$r_4^{move} = clear(X), clear(G_1), block(X), height(X, N_0), height(G_1, N_1) \rightarrow move(X, G_1)$

5.5.3 Refining the Rule Conditions

Whenever a rule is created, the conditions are simplified using the simplification process detailed in Section 5.4. The rule conditions are then heuristically sorted as defined in Section 4.7.3, such that the conditions with the fewest likely matches are at the beginning of the rule to quickly refine the number of possible variable replacements for a rule.

5.6 Discussion and Future Work

The use of the observation model allows the algorithm to keep the number of specialisations low, both by restricting the number of possible specialisations and by simplifying specialised rules to remove redundancies. The specialisation conditions are limited to action-related conditions to limit the number of possible rules created, thereby increasing the efficiency of the learning process. Chapter 6 examines the effects of using simplification rules on CERRLA's performance.

The observations model is composed of a separate goal-based model and environment model to allow learned relationships to be applied to new problems in the same environment. By explicitly separating the environment observations from the current goal observations, the agent can reuse learned environment models on alternative goals within the same environment.

CERRLA's current language bias results in relatively few specialisation conditions because it does not concern itself with non-action related terms for the rules that are created. This results in simpler rules, smaller rule distributions and therefore, a shorter training time to create effective behaviour. However, this restriction may also negatively affect CERRLA's performance.

A future direction for CERRLA is to allow non-action-related conditions to be included as specialisation conditions. This should be implemented carefully, to avoid creating needlessly complex rules defining situations that have no affect on the rule's behaviour.

Another area of future work is defining a better method to handle numerical terms. The current method of defining ranges and splitting them into 3–5 arbitrary sub-ranges is effective, but crude. A possible alternative is to organise observed numerical values into a Gaussian distribution such that ranges are defined by standard deviations from the mean.

6

Algorithm Evaluation

This chapter describes the experiments performed to evaluate CERRLA, and the results obtained. Where possible, experiments are evaluated against related approaches to provide a comparison to the algorithm's performance. Unless stated otherwise, parameters are set as described in Chapter 4: $\alpha = 0.6$ (Section 4.2), $\beta = 0.01$ (Section 4.6.4), $C = 3$ (Section 4.6.2), $\rho = 0.05$ (Section 4.2), and distribution properties are initialised as $p(D) = 0.5$ and $q(D) = 0.5$ (Section 4.2.1).

6.1 Experiment Methodology

All results shown are the averaged result of ten experiments, where each experiment consists of CERRLA learning behaviour in an environment for a specific goal, beginning with no prior knowledge of the environment (unless otherwise stated). The standard deviation shown for results is the deviation between experiments. The random generators for each experiment are seeded with the experiment iteration number and unless specified otherwise, each experiment begins without any information from prior experiments. Each policy sample is evaluated over three episodes to produce the averaged value (see Section 4.5 for details). Experiments were performed on Intel® Core™ i7-2600 CPU @ 3.40GHz machines, where each experiment only uses a single core and was limited to 4GB RAM. The output files created for all experiments can be found at http://www.samsarjant.com/cerrla/.

Performance is measured with two metrics (each metric represents the mean value over ten experiments): the online performance of the algorithm's *sampled policies*, and the greedy performance (the performance of the *best elite policies*).

The online performance is measured as the mean score of a sliding window of samples (where the window contains 100 samples). This is not an exact measure of the sampled policy's performances, but it provides a close approximation to the actual performance. When learning is completed (either after a fixed number of episodes or β convergence is reached), all probabilities are fixed (no more updates) and 100 policies are generated and tested to produce the *true* mean online performance. Some figures may also include a Standard Deviation (SD) of the sample performance between experiments.

The greedy performance is calculated as the mean performance of the best elite sample, measured every 300 episodes. The mean is calculated as the average reward received by testing the current best elite sample in 100 episodes (these episodes are not included in the episode count or training time).

Each environment also lists the final (averaged) results for each goal in the experiments in tables (e.g. Table 6.1). Each row in the table presents the goal, the average number of episodes required to converge (if using β convergence, see below), the final mean reward for sampled policies ($\pm$ standard deviation), the final mean reward for the greedy policy ($\pm$ standard deviation), the mean number of candidate rule distributions at the beginning and end of learning (which increases due to branching, Section 4.7.2), the mean number of rules at the beginning and end of learning (where rules are created whenever a distribution is created, Section 4.7.2), and the mean training time in seconds for an experiment to be completed (this does not include time spent testing greedy policies or determining the true sampled policy performance). All measurements of time should be regarded as approximate values and are only presented as rough guides.

Each environment also presents example policies created by CERRLA for each goal. The rules of the policies have not been changed except for any defined sub-ranges, which have instantiated any dynamic range bounds with values and altered the appearance of the range to be more compre-

hensible. The environmental specification tables in Chapter 3 list the definitions of each of the predicates used in the environments.

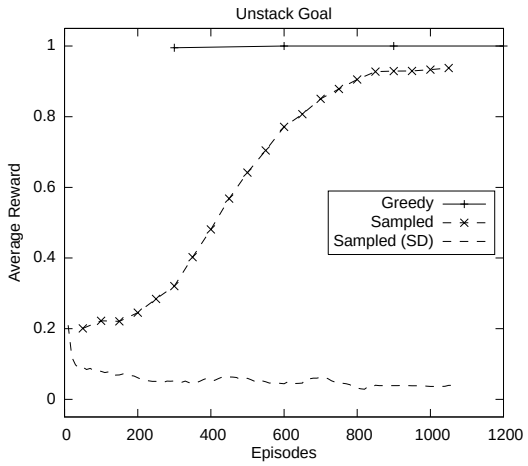
Convergence is either determined by CERRLA (using β, see Section 4.6.4), or the algorithm is run for a fixed number of episodes, regardless of β convergence. If training for a fixed number of episodes, CERRLA is restricted from branching further distributions after 90% of the training episodes have passed. This is to stop CERRLA creating new, uniform-distributions immediately before performing the final test of the experiment.

6.2 Blocks World Evaluation

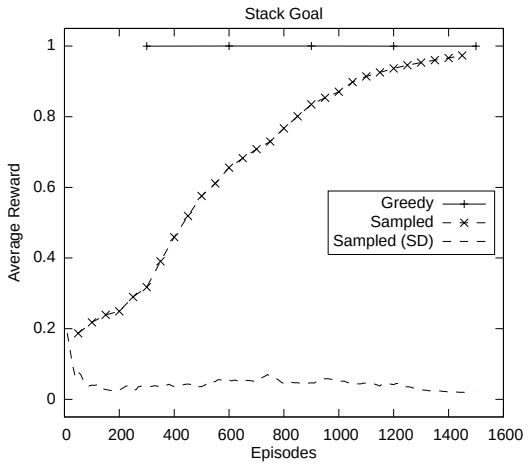
The first evaluation of the algorithm tries to determine whether it is able to learn behaviour for the standard RRL benchmark environment BLOCKS WORLD (defined in Section 3.3). As stated in Section 3.3.3, the algorithm was tested on the *Stack*, *Unstack*, $ClearG_0$, and OnG_0G_1 goals, with different numbers of blocks, and the results are compared to the published results for other algorithms (Section 6.2.1). The BLOCKS WORLD environment is also used to examine the effects of the simplification rules created by the *agent observations model* (Section 6.2.4). The language in which an environment is presented can affect an agent's performance, so Section 6.2.5 examines the effects of representing BLOCKS WORLD using a different relational specification. Section 6.2.6 explores the effects on CERRLA's performance when actions in BLOCKS WORLD have non-deterministic effects. Finally, Section 6.2.7 presents a summary and discussion of CERRLA's performance in the BLOCKS WORLD environment.

6.2.1 Standard CERRLA Performance

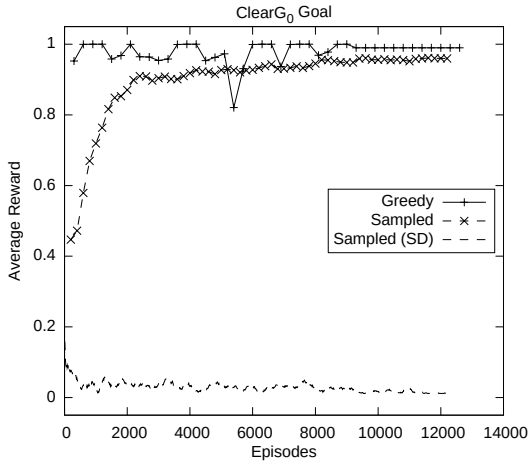
The first set of experiments for BLOCKS WORLD simply tests the default learning behaviour of CERRLA on the four BLOCKS WORLD goals. Each BLOCKS WORLD environment is initialised with ten blocks, and in the $ClearG_0$ and OnG_0G_1 case, the goal blocks are selected as any blocks that do not immediately satisfy the goal. For each goal, Figure 6.1 presents the performance, Figure 6.2 lists example policies that are generated by CERRLA at the end of learning, and Table 6.1 lists other details about the learning process.



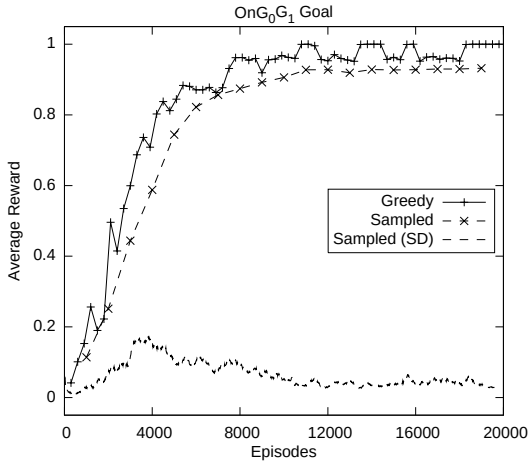
(a) *Unstack* goal, 10 blocks, CERRLA uses β convergence.



(b) *Stack* goal, 10 blocks, CERRLA uses β convergence.



(c) *Clear*G_0 goal, 10 blocks, CERRLA uses β convergence.



(d) *On*G_0G_1 goal, 10 blocks, CERRLA uses β convergence.

Figure 6.1: CERRLA's performance for the four BLOCKS WORLD goals. Included is the average performance for the greedy policy, the average sampled policy performance, and the standard deviation of the average sampled policy performance between experiments.

Table 6.1: Averaged results (over ten experiments) regarding CERRLA's performance in the four goals of the BLOCKS WORLD environment. Convergence is determined with β convergence.

Goal	Episodes	Sampled	Greedy	# Distributions	# Rules	Time (s)
Unstack	785 ± 117	0.94 ± 0.04	1.0 ± 0.0	5–6	24–24	4
Stack	1265 ± 170	0.98 ± 0.02	1.0 ± 0.0	5–5	24–23	4
ClearG_0	8507 ± 2280	0.96 ± 0.01	0.99 ± 0.03	9–17	80–123	22
OnG_0G_1	10677 ± 4744	0.93 ± 0.05	1.0 ± 0.0	17–25	243–302	44

CERRLA consistently learns an optimal policy for every problem in BLOCKS WORLD, especially quickly for the *Unstack* and *Stack* goals (Figure 6.1a and 6.1b), as the rule required for optimal behaviour is immediately available within the initial distributions (as indicated by the greedy policy performance). The number of rules actually goes down due to branching creating an existing distribution in the *Stack* goal. The *Unstack* goal has the opposite case: a branch occurs and creates a new distribution of size 1 due to no more possible specialisations. CERRLA also consistently finds optimal policies (except for a single case for $ClearG_0$) for both the $ClearG_0$ and OnG_0G_1 goals within $\sim$10,000 episodes (shown by the greedy performance), but the *sampled performance* does not meet the greedy performance because the distribution and/or rule probabilities do not converge to 0 or 1. This problem is largely due to how the distribution's usage probabilities ($p(D)$) are updated.

Each distribution's $p(D)$ reflects the *observed* probability of a distribution being present within the elite samples. In the $ClearG_0$ case there are usually multiple rules, each existing in different distributions, that are capable of achieving the goal. The resulting elite samples then consist of equally-valued samples from multiple distributions, which result in $p'(D) < 1$, and no single distribution being updated to $p(D) = 1$. This problem is gradually resolved with further updates, as random selection will eventually favour one distribution over another.

In the OnG_0G_1 case, the distribution usage problem is a side effect of the randomised environment. In the OnG_0G_1 problem, there is a chance that the initial states of all three evaluation episodes of a policy will not require all three rules of the optimal OnG_0G_1 policy (Figure 6.2d). This results in the elites containing subsets of the optimal policy, thereby observing a usage of $p'(D) < 1$ for the unused rules. The following updates cause $p(D)$ for the unused rules to decrease, resulting in sampled policies that do not contain the distribution. Although the rule is clearly optimal when it is needed, because the environment does not always require it, $p(D)$ is updated to reflect the observed probability that the rule *will be* required to solve the goal. This problem is mitigated by requiring that each sampled policy be tested three times, resulting in a larger probability that all optimal rules will be utilised.

clear(X), block(X), floor(Y) $\rightarrow$ *move(X, Y)*

(a) An example optimal policy generated by CERRLA for the *Unstack* goal after 5000 episodes of learning.

clear(X), highest(Y), block(X) $\rightarrow$ *move(X, Y)*

(b) An example optimal policy generated by CERRLA for the *Stack* goal after 5000 episodes of learning.

above(X, G_0), clear(X), clear(Y) $\rightarrow$ *move(X, Y)*

(c) An example optimal policy generated by CERRLA for the *Clear*G_0 goal after 10,000 episodes of learning.

clear(G_0), clear(G_1), block(G_0) $\rightarrow$ *move(G_0, G_1)*
above(X, G_1), clear(X), floor(Y) $\rightarrow$ *move(X, Y)*
above(X, G_0), clear(X), floor(Y) $\rightarrow$ *move(X, Y)*

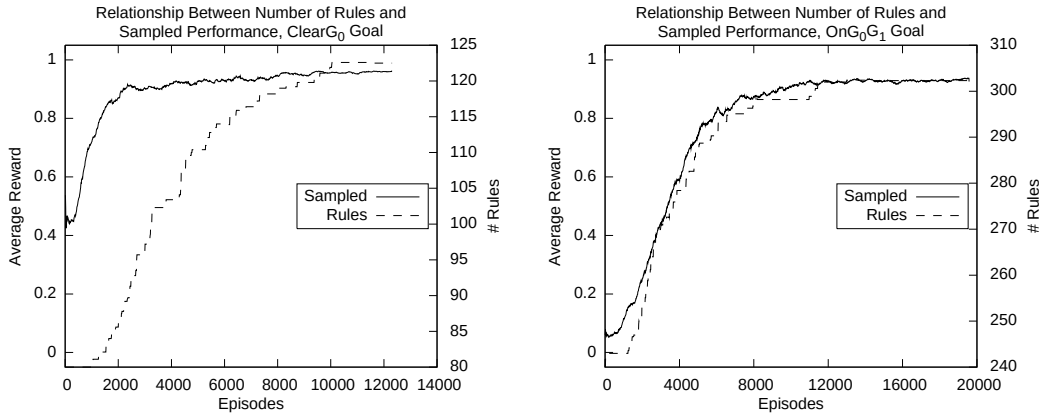
(d) An example optimal policy generated by CERRLA for the *On*G_0G_1 goal after 20,000 episodes of learning.

Figure 6.2: Example policies created by CERRLA for the four BLOCKS WORLD goals.

The policies in Figure 6.2 represent the best elite policies generated at the end of training for each of the respective goals from an arbitrary experiment. CERRLA does not always converge to the exact same policies; the rules may differ slightly (but remain optimal, e.g. for *Clear*G_0, shifting *blocks* to the *floor or* to *blocks*), or for the *On*G_0G_1 goal, the order of the rules may change. These policies also only contain the minimum rules necessary to (optimally) achieve their respective goals; no redundant rules remain. Furthermore, each rule only contains the minimal conditions required to define the intent of the rule. Note that each rule also contains inequality tests between variables (Section 5.5.3) — these have been hidden for clarity.

Rule and Slot Growth

Figure 6.3 shows the relationship between the mean number of rules and the mean sampled performance over the course of learning. Because no distribution is updated until it has evaluated a 'fair' number of rules (defined in Section 4.6.2), no new distributions are created until approximately 1000 episodes have passed. For both the *Clear*G_0 and *On*G_0G_1 goals, CERRLA ceases to explore further rules at approximately the 10,000 episode mark, indicating that further specialisations would only decrease its performance. Any further episodes are concerned primarily with determining the optimal probabilities for the current rules present in the distribution. The exploration for the *Clear*G_0 goal takes longer to converge due to multi-



(a) The relationship between the number of CERRLA's rules and the performance in BLOCKS WORLD for the *Clear*G_0 goal.

(b) The relationship between the number of CERRLA's rules and the performance in BLOCKS WORLD for the *On*G_0G_1 goal.

Figure 6.3: The relationship between the number of CERRLA's rules and the performance in BLOCKS WORLD for the *Clear*G_0 and *On*G_0G_1 goals.

ple possibilities for optimal rules, whereas the OnG_0G_1 exploration closely matches the performance because further exploration only creates sub-optimal rules.

6.2.2 Scale-free Policies

Figure 6.4 and Table 6.2 illustrate the differences in learning in BLOCKS WORLD environments of different sizes. CERRLA also requires roughly the

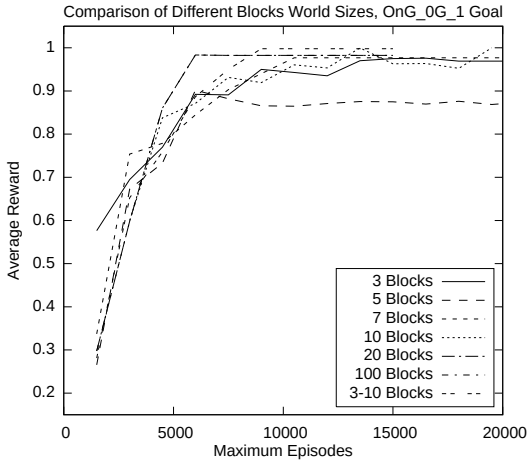


Figure 6.4: A comparison of CERRLA's rate of learning on different sized BLOCKS WORLDS for the *On*G_0G_1 goal. Shown are the averaged greedy performances over ten runs, using β convergence.

Table 6.2: Averaged results (over ten experiments) for different sized BLOCKS WORLD environments using the OnG_0G_1 goal. Training and testing is performed with the same number of blocks.

# Blocks	Episodes	Greedy	Greedy (10)	# Distributions	# Rules	Time (s)
3 Blocks	11115 ± 7386	0.97 ± 0.07	0.73 ± 0.16	17–27	237–313	22
5 Blocks	11120 ± 5558	0.87 ± 0.14	0.76 ± 0.17	17–26	243–306	28
7 Blocks	10539 ± 5738	0.98 ± 0.06	0.97 ± 0.08	17–24	243–301	35
10 Blocks	10677 ± 4744	1.0 ± 0.0	1.0 ± 0.0	17–25	243–302	44
20 Blocks	8486 ± 2821	0.98 ± 0.04	0.98 ± 0.04	17–26	242–330	86
100 Blocks	10016 ± 3205	0.97 ± 0.02	1.0 ± 0.0	17–30	244–332	1,639
3–10 Blocks	10263 ± 2756	1.0 ± 0.01	0.99 ± 0.03	17–25	242–302	34

same number of training episodes. These experiments clearly demonstrate CERRLA's indifference to state size, as the 100 block BLOCKS WORLD has over 10^{163} possible states, but only takes 37 times longer than for 10 block BLOCKS WORLD (with approximately 5.9×10^7 possible states). The size of the state does not affect the learning rate because CERRLA searches for a policy in *rule space*, which remains constant, rather than in *state space*, which increases exponentially with the number of blocks.

The *structure* of the environment determines the structure of the policies that CERRLA learns. For example, Figure 6.5 shows a policy that is optimal in 3-block BLOCKS WORLD, but sub-optimal in 10-block BLOCKS WORLD. The same policy is only optimal in 5 and 7-block environments in specific situations, but these occur frequently enough that it may be produced as CERRLA's final policy. 10-block and larger environments decrease the chances of such policies becoming elite policies because such a simple strategy is usually sub-optimal.

CERRLA performs better in larger environments because there are fewer policies that can achieve the goal in minimal steps, whereas in small environments there are more policies that can achieve the goal in minimal steps. This would result in potentially sub-optimal policies within the elite

$clear(G_0), clear(G_1), block(G_0) \rightarrow move(G_0, G_1)$
$clear(X), block(X), floor(Y) \rightarrow move(X, Y)$

Figure 6.5: A optimal OnG_0G_1 policy for 3-block BLOCKS WORLD environments produced by CERRLA.

samples, affecting the rule and distribution updates.

In the 3-block environment, it is relatively easy for the agent to achieve the goal, though the resulting policy is only near-optimal for 3-block and not 10-block Blocks World. In the 5-block environment, because sub-optimal policies are often 'optimal enough,' Cerrla struggles to converge to a single solution, resulting in an overall decreased level of performance when testing on both the 5 and 10-block environments. 7-block Blocks World is large enough such that Cerrla generally produces optimal policies for 7 and 10 block environments. The 10, 20, and 100-block environments typically produce optimal policies, though there are some exceptions. The rarity of randomly achieving the goal state in these larger environments reduces the set of possibly useful rules, focusing the set of elites to only include optimal or very near optimal samples.

When training on a varied number of blocks, Cerrla performs well, outputting the optimal policy in the majority of cases. This is probably because sub-optimal samples are less likely to be in the elites due to each sample being tested three times. If just one of the tests is in a larger environment, the sample may not be recorded as an elite sample, resulting in the elite samples containing mostly optimal samples.

6.2.3 Comparison to Existing Algorithms

As Blocks World is a common testing environment in RRL algorithms, a direct comparison of performance can be made between Cerrla and other RRL algorithms, though there is no guarantee that the specification of the environments are identical. Most Blocks World experiment setups for other RRL algorithms vary the number of blocks between 3–5 throughout training (Driessens and Džeroski, 2005; Mellor, 2008a), typically to help value-based algorithms generalise over Blocks World environments of different sizes, so Cerrla is trained on a varying number of blocks (but tested in a 10 block environment). The performance of the greedy (best elite) policies is used for Cerrla's performances.

Results for the policy-based Grey (Muller and van Otterlo, 2005) and Gapi (van Otterlo and De Vuyst, 2009) algorithms are not included due to an obscure evaluation metric. LRW-API (Fern et al., 2006) is also not included as it uses a version of Blocks World in which any state is accessible at any

time.

Table 6.3 lists performance results for the *Stack* and OnG_0G_1 goals in BLOCKS WORLD. It also presents the (approximate) number of training episodes required to learn the resulting behaviour. CERRLA ranks among the best learners in terms of performance for both the *Stack* and OnG_0G_1 goals when compared to other algorithms, though it requires more episodes than most other algorithms. When trained within an environment of varying numbers of blocks, CERRLA does not consistently find an optimal policy for OnG_0G_1, but still maintains a relatively high performance and requires fewer training episodes for the OnG_0G_1 goal. Although it is difficult to compare training time of learning without recreating all experiments, CERRLA's training time is similar to the faster algorithms such as RRL-TG and TRENDI.

The GREY and GAPI algorithms (Muller and van Otterlo, 2005; van Otterlo and De Vuyst, 2009) are algorithmically most similar to CERRLA, but unfortunately their results were not published in a quantifiable format. Both use a form of the Genetic Algorithm (GA) to learn decision-list policies.

Table 6.3: A comparison of performances for various RRL algorithms in the BLOCKS WORLD environment for the *Stack* and OnG_0G_1 goals. Also included are the number of episodes required for training (where information is available). Sources for performances are from Džeroski et al. (2001), Driessens and Džeroski (2005), Kersting and Driessens (2008), Croonenborghs et al. (2007), Mellor (2008b). Note that some figures are approximate readings from a graph.

Algorithm	Average Reward		# of Training Episodes ($\times$1000)	
	Stack	OnG_0G_1	*Stack*	OnG_0G_1
CERRLA*	1.0	1.0	1.3	10.7
CERRLA†	1.0	$\sim$ 0.99	1.6	10.3
P-RRL	1.0	$\sim$ 0.9	0.045	0.045
RRL-TG	$\sim$ 0.88	$\sim$ 0.92	0.5	12.5
RRL-TG‡	1.0	$\sim$ 0.92	30	30
RRL-RIB	$\sim$ 0.98	$\sim$ 0.9	0.5	2.5
RRL-KBR	1.0	$\sim$ 0.98	0.5	2.5
TRENDI	1.0	$\sim$ 0.99	0.5	2.5
TREENPPG	—	$\sim$ 0.99	—	2
MARLIE	1.0	$\sim$ 0.98	2	2
FOXCS	1.0	$\sim$ 0.98	20	50

*10 blocks. †3–10 blocks. ‡P-learning.

They are able to learn optimal policies for the OnG_0G_1 goals, though it is unclear exactly how many training episodes each one requires. Because GREY and GAPI use the GA to create their policies, the output policies can include 'useless' rules and conditions that have no direct negative effect on the semantic intent of the policy, but add clutter to the behaviour. CERRLA's probabilistic updating of rule probabilities and use of simplification rules result in less 'cluttered' policies, while maintaining the semantic intent of the policy.

The main disadvantage of the CERRLA algorithm is that it requires a relatively large number of episodes to construct an optimal policy, in comparison to various value-based algorithms. As shown in the previous subsection, the number of episodes remains roughly constant regardless of BLOCKS WORLD size. However, because it only uses the results of a policy per episode (rather than per state), computation is quite fast. An additional advantage is that CERRLA does not require a distance-metric (RRL-RIB, TRENDI) or kernel (RRL-KBR) to be defined.

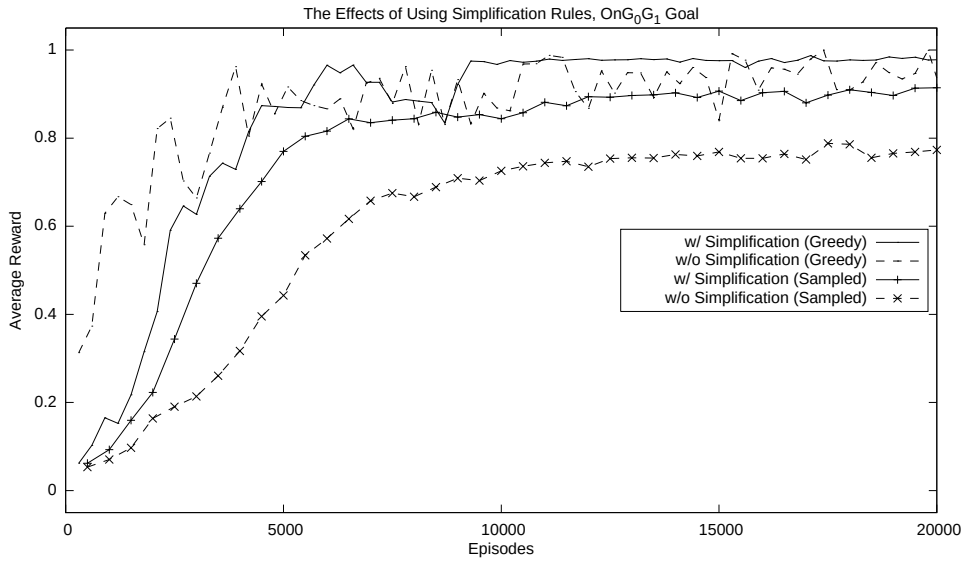


Figure 6.6: A comparison of performance for the OnG_0G_1 goal in the BLOCKS WORLD environment between using agent observations to simplify rules, and not using them. Greedy performances have been omitted for clarity. Each experiment ran for a fixed 20,000 episodes.

Table 6.4: Averaged results (over ten experiments) between using agent observations to simplify rules, and not using them for the OnG_0G_1 goal in the BLOCKS WORLD environment. Each experiment ran for a fixed 20,000 episodes.

OnG_0G_1	**Sampled**	**Greedy**	**# Distributions**	**# Rules**	**Time (s)**
w/ Simplification	0.91 ± 0.04	0.98 ± 0.04	17–27	243–321	87
w/o Simplification	0.77 ± 0.05	0.91 ± 0.17	26–40	764–1144	232

6.2.4 Agent Observation Simplification

To test whether rule simplification aids performance, CERRLA is tested on the OnG_0G_1 goal within BLOCKS WORLD without the use of simplification rules created through agent observations (Section 5.3). Figure 6.6 and Table 6.4 illustrate the results, compared against using simplification rules. Learning is fixed to 20,000 episodes per experiment because the algorithm does not converge in a reasonable time when not using simplification rules.

There is a clear difference in performance, where the lack of simplification rules results in a dramatically larger number of rules and distributions, slowing down the learning process and affecting the sampled performance with its constant exploration. Furthermore, the greedy performance of CERRLA when not using agent observations is worse than the sampled performance (though CERRLA still manages to create optimal policies in most experiments).

No comparison is performed on episodic convergence speed because when simplification rules are disabled the algorithm continues to explore and does not reach β-convergence in a reasonable amount of time. For this reason training is fixed to 20,000 episodes. The lack of simplification rules results in training taking almost three times as long than when using simplified rules, which is likely due to the increased number of distributions and (redundant) conditions within each rule, thus increasing the overall rule evaluation time.

Another interesting result is the drop in performance of the normal CERRLA algorithm when training episodes are fixed to 20,000. The performance drops from the normal optimal performance due to the algorithm running longer than it needs to. This is probably due to a small number of elite samples, resulting in a non-insignificant probability of only containing samples

in which all three rules of the optimal policy are not used during testing. This causes the distribution's usage to shift to some value $p(D) < 0.5$, affecting performance.

Figure 6.7 shows one of the best elite policies produced by CERRLA without the use of simplification rules. Each rule in the policy clearly contains redundant conditions and some conditions could be simplified to a simpler form (e.g. *clear(Y)* and *on(?, Y)* are equivalent to *floor(Y)*), but semantically the policy is optimal.

6.2.5 Language Bias

The representation of an environment can have a significant effect on an algorithm's performance. The BLOCKS WORLD environment can be represented in several different forms, though each form has the same basic actions (block manipulation). One possible alternative representation is an abstraction of the BLOCKS WORLD specification given in Section 3.3, which we will call $BW^{noFloor}$. This version removes explicit references to the *floor* object, replaces all references to *thing* with *block*, represents relations to the *floor* with the predicate *onFloor(Block)*, and interacts with the *floor* by adding the action predicate *moveFloor(Block)*. The added predicates encompass all interaction with the *floor*; all other predicates explicitly deal with *blocks*.

CERRLA's performance for the OnG_0G_1 goal in $BW^{noFloor}$ is shown in Figure 6.8 and Table 6.5, contrasted against the performance of CERRLA in the

above(X, G_0), clear(X), clear(Y), above(X, ?), on(X, ?), on(?, Y) → move(X, Y)
above(X, G_1), above(X, Y), clear(X), clear(Y), above(X, ?), on(X, ?), not above(Y, ?) → move(X, Y)
clear(G_0), clear(G_1), block(G_0), above(G_0, ?), on(G_0, ?), not above(G_0, G_1) → move(G_0, G_1)

Figure 6.7: An optimal OnG_0G_1 BLOCKS WORLD policy produced by CERRLA after 20,000 episodes without using simplification rules.

Table 6.5: CERRLA's performance using an alternative representation of BLOCKS WORLD for the OnG_0G_1 goal. Convergence is determined with β convergence.

OnG_0G_1	**Episodes**	**Sampled**	**Greedy**	**# Distributions**	**# Rules**	**Time (s)**
Normal BW	10677 ± 4744	0.93 ± 0.05	1.0 ± 0.0	17–25	243–302	44
$BW^{noFloor}$	6346 ± 1048	0.83 ± 0.08	0.97 ± 0.06	27–31	364–398	29

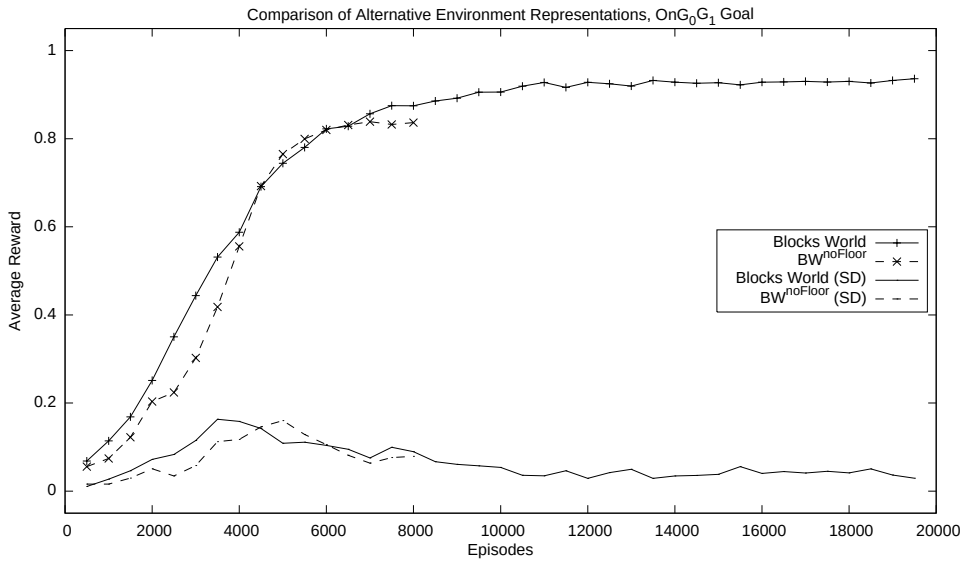


Figure 6.8: CERRLA's performance using an alternative representation of BLOCKS WORLD for the OnG_0G_1 goal. Greedy performances have been omitted for clarity. Convergence is determined with β convergence.

standard BLOCKS WORLD. The change in environment representation does not affect CERRLA's greedy performance much, though sampled performance is lower. The alternative representation also results in much quicker average convergence speed, probably because all the rules that can form an optimal policy exist in the initial distributions (whereas BLOCKS WORLD rules require at least one branch). To check that CERRLA is not converging too early in $BW^{noFloor}$, another experiment fixes the number of training episodes to 20,000, but the results do not significantly improve. The alternative representation also results in a much larger initial distribution of rules and distributions than the regular representation, but this does not appear to negatively affect CERRLA's convergence to a final solution.

6.2.6 Stochastic Blocks World

Another alternative BLOCKS WORLD setup is to allow the actions to have non-deterministic effects to observe how CERRLA's behaviour changes. With probability $p = 0.8$, an action behaves as normal, $p_{null} = 0.1$ the action does nothing, and $p_{rand} = 0.1$ the action is instead a random valid action. To compensate for the reduced probability of success, the maximum number of episodes M is set as $M \leftarrow 2n/p$, where n is the number of blocks in the environment. However, the reward function remains the same, such that

an agent may receive a non-optimal reward even with an optimal policy (and vice-versa). During testing, the environment is deterministic ($p = 1.0$) so accurate performance can be measured.

As shown in Figure 6.9 and Table 6.6, the sampled performance is lower than the regular performance, which was expected, as the reward function was not changed to accommodate random actions during training. However, the greedy performance is tested on a deterministic BLOCKS WORLD, so 0.92 ± 0.14 is an accurate measure of performance. Because the random behaviour reduces each policy's observed reward, policies that run for multiple steps are less likely to be included within the elite samples (even if the policy is optimal). This results in a more stable set of elite samples, but these elites may not be optimal. Nonetheless, the greedy performance is still reasonably good, which shows CERRLA is not significantly affected by non-deterministic action effects.

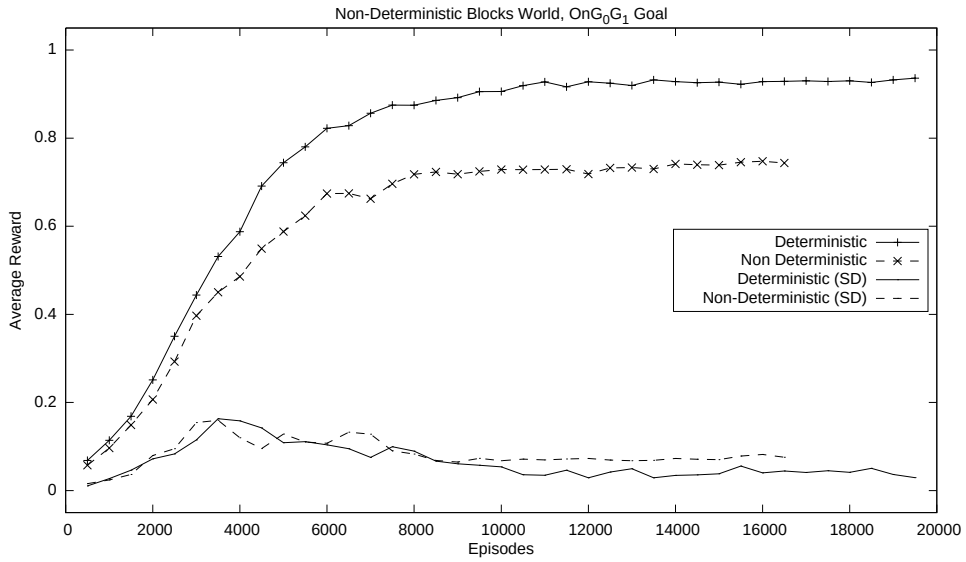


Figure 6.9: CERRLA's performance in a stochastic BLOCKS WORLD for the OnG_0G_1 goal. Convergence is determined with β convergence.

Table 6.6: CERRLA's performance in a stochastic BLOCKS WORLD for the OnG_0G_1 goal. Convergence is determined with β convergence.

OnG_0G_1	**Episodes**	**Sampled**	**Greedy**	**# Distributions**	**# Rules**	**Time (s)**
Deterministic	10677 ± 4744	0.93 ± 0.05	1.0 ± 0.0	17–25	243–302	44
Stochastic	9168 ± 3368	0.74 ± 0.07	0.92 ± 0.14	17–28	243–339	42

What is not shown by Table 6.6 is that from the ten non-deterministic experiments, six produce optimal policies, three produce sub-optimal but consistently goal-achieving policies, and one produces a policy that only achieves the goal $\sim 66\%$ of the time.

6.2.7 Blocks World Discussion

This section examined several aspects of both Blocks World and Cerrla's learning algorithm. Section 6.2.1 showed that Cerrla is able to consistently learn an optimal or near-optimal policy for all four Blocks World goals, but only when the number of blocks is > 10. In smaller Blocks Worlds, the reduced number of blocks makes it easier to achieve goal states, causing the elite samples to become noisy and affect optimal policy convergence. When the number of blocks varies from three to ten, Cerrla learns near-optimal behaviour (> 0.99 average reward) in a relatively short amount of time. In comparison to other RRL algorithms, Cerrla is better than or equal in performance and is of a similar speed to the fastest algorithms.

The use of simplification rules (created via agent observations) is clearly beneficial (Section 6.2.4), both in terms of performance and run time. An alternative representation of Blocks World can decrease training time by producing more initial distributions for the agent to examine. Changing Blocks World's action resolution to be non-deterministic also influences Cerrla's performance, but average performance remains near-optimal ($>$ 0.92).

6.3 Ms. Pac-Man Evaluation

The Ms. Pac-Man environment contains three different goals for the agent to be tested upon: *Single Level*, *Single Life*, and *Ten Levels*. As in the previous section, we examine the effects of an alternative environment representation in the *Single Level* goal. Section 6.3.3 investigates the effects of transferring knowledge learned in the *Single Level* and *Single Life* goals to an agent in the *Ten Levels* goal, as well as testing the effect of adding hand-coded rules to the learning process.

The Ms. Pac-Man experiments were limited to a fixed number of episodes because it was found that Cerrla occasionally did not achieve β-convergence,

resulting in experiments that took far to long to complete. The values for the fixed number of episodes were selected to arbitrarily limit the length of each experiment.

6.3.1 Standard CERRLA Performance

The default parameters for CERRLA produce the results seen in Figure 6.10a, 6.10b, 6.10c and Table 6.7. CERRLA shows roughly the same performance curve for all three goals, though the scale differs per goal. An unfortunate side-effect of the Ms. PAC-MAN environment (as well as the MARIO environment) is that as the agent improves its behaviour, each episode takes longer to complete, increasing the overall training time.

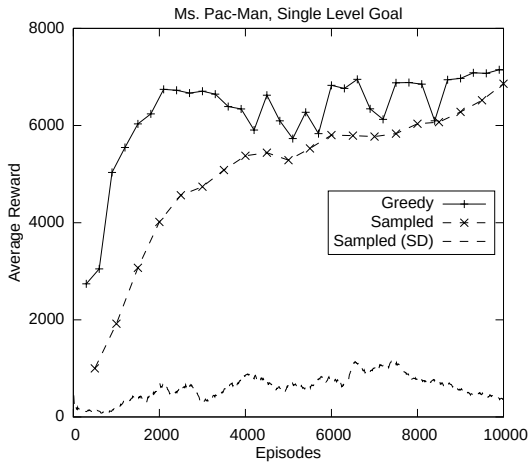
Within a *Single Level*, CERRLA achieves an average greedy performance of 7196 points per episode. Comparing this to the scores presented in Szita and Lörincz (2007),[1] CERRLA learns a policy that performs better than the conceptually equivalent CE-RANDOMRB agent (achieves mean score of 6382 in 50,000 episodes), but worse than CE-FIXEDRB (achieves mean score of 8186 in 50,000 episodes). Figure 6.11a shows an example elite policy produced by CERRLA at the end of training. The policy behaviour focuses primarily on eating *edible ghosts*, but when no *edible ghosts* are available, it eats *powerdots* (breaking ties by moving to things that are not ghosts or the ghost centre point) and finally eating any remaining *dots*.

With only a *Single Life*, CERRLA achieves an average greedy performance of 4616 points per episode. It achieves this by learning a policy similar to the one learned for the *Single Level* goal — that is, eating *edible ghosts* (Figure 6.11b). This strategy is relatively safe because when a *ghost* is *edible*, it is not

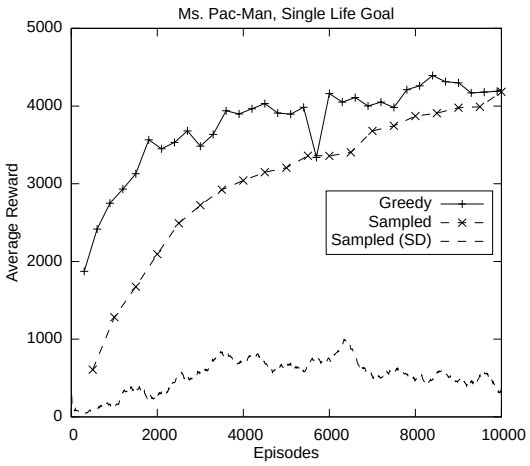
Table 6.7: Averaged results (over ten experiments) regarding CERRLA's performance for the three goals of the MS. PAC-MAN environment at the end of a fixed number of episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Single Level	6861 ± 357	7137 ± 277	15–28	86–184	21,063
Single Life	4183 ± 422	4274 ± 279	15–26	86–154	13,473
Ten Levels	11500 ± 1963	12473 ± 1984	15–41	86–199	89,154

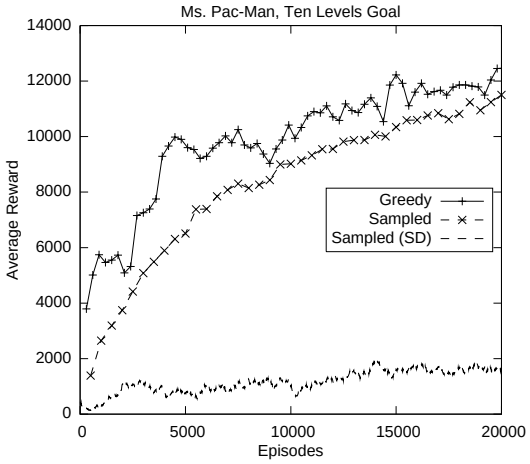
[1] It is likely that the Ms. PAC-MAN environments used are not identical, but the reward function should be identical.



(a) *Single Level* goal, limited to 10,000 episodes.



(b) *Single Life* goal, limited to 10,000 episodes.



(c) *Ten Levels* goal, limited to 20,000 episodes.

Figure 6.10: CERRLA's performance for the three goals in MS. PAC-MAN. Included is the average greedy performance, the average sampled policy performance, and the standard deviation of the average sampled policy performance between experiments. Each experiment runs for a fixed number of episodes.

edible(X), distance(X, N_2) → moveTo(X, N_2)
powerdot(X), distance(X, N_2) → moveTo(X, N_2)
thing(X), distance(X, N_2), not ghost(X), not ghostCentre(X) → moveTo(X, N_2)
dot(X), distance(X, (26.0 ≤ N_0 ≤ 52.0)) → moveFrom(X, N_0)

(a) Example *Single Level* MS. PAC-MAN policy generated by CERRLA. Achieves an average reward of 7534.

edible(X), distance(X, N_5) → moveTo(X, N_5)
powerdot(X), distance(X, N_5) → moveTo(X, N_5)

(b) Example *Single Life* MS. PAC-MAN policy generated by CERRLA. Achieves an average reward of 4542.

dot(X), distance(X, N_2) → moveTo(X, N_2)
thing(X), distance(X, (13.0 ≤ N_0 ≤ 39.0)), not ghost(X) → moveFrom(X, N_0)
ghost(X), distance(X, N_0), not edible(X) → moveFrom(X, N_0)
powerdot(X), distance(X, (26.0 ≤ N_0 ≤ 52.0)) → moveFrom(X, N_0)
distance(X, (13.0 ≤ N_0 ≤ 39.0)), not dot(X), not edible(X) → moveFrom(X, N_0)
thing(X), distance(X, (26.0 ≤ N_0 ≤ 39.0)), not dot(X), not ghostCentre(X) → moveFrom(X, N_0)

(c) Example *Ten Levels* MS. PAC-MAN policy generated by CERRLA. Achieves an average reward of 12,457.

Figure 6.11: Example policies created by CERRLA for the three MS. PAC-MAN goals.

hostile. However, CERRLA does not learn to use any defensive rules, such as moving from hostile *ghosts*, possibly because the strategy of keeping ghosts edible has little need of such a rule.

In the largest Ms. PAC-MAN goal, *Ten Levels*, CERRLA achieves an average greedy performance of 12,473 points per episode. Because it typically achieves $> 10,000$ points, it also receives an extra life, which increases the performance of the agent by increasing survivability. Figure 6.11c shows an example elite policy produced after 20,000 episodes of training. The policy behaviour basically just moves towards *dots*, but when faced with *dots* at equal distances in multiple directions, there is a bias towards moving from hostile *ghosts* and other things at a mid-distance. In all *Ten Levels* goal experiments, the agent produced policies of a similar structure, indicating that the basic behaviour of eating dots is a useful strategy.

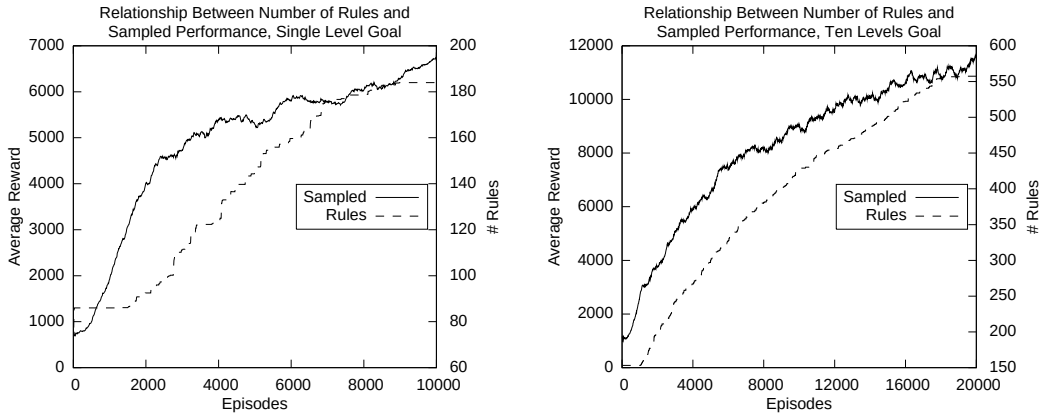
The *Ten Levels* goal searches significantly more distributions and rules than the prior two goals, but this is partially due to the number of training

episodes being twice as long. At episode 9000 (which is when branching would be disabled if training was fixed at 10,000 episodes like the other goals), CERRLA had an average of 31 distributions and 190 rules for the *Ten Levels* goal. However, performance continues to increase after 10,000 episodes, so the extra distributions and rules may be required for reaching the final level of performance. The side-effect of increased training time proportional to performance is evident for the *Ten Levels* goal, as episodes take significantly longer to evaluate than for the prior two goals.

Other AI approaches to the Ms. PAC-MAN environment are able to score $> 10,000$ points in the *Single Life* goal (Galván-López et al., 2010) and $> 20,000$ points for a goal similar to the *Ten Levels* goal (Ikehata and Ito, 2011) (with many more algorithms competing in the Pacman-vs-Ghosts competition[2]), so CERRLA is far from the best Ms. PAC-MAN player. But, as an algorithm capable of learning behaviour in multiple environments, CERRLA is competitive with some of the specialised algorithms for playing Ms. PAC-MAN.

Rule and Slot Growth

Figure 6.12 shows the relationship between the mean number of rules and the mean sampled performance in the *Single Level* and *Ten Levels* goals. Like



(a) The relationship between the number of CERRLA's rules and the performance in MS. PAC-MAN for the *Single Level* goal.

(b) The relationship between the number of CERRLA's rules and the performance in MS. PAC-MAN for the *Ten Levels* goal.

Figure 6.12: The relationship between the number of CERRLA's rules and the performance in MS. PAC-MAN for the *Single Level* and *Ten Levels* goals.

[2]http://www.pacman-vs-ghosts.net/

Blocks World, rule specialisation does not begin until approximately 1000 episodes. Note that rule specialisation is automatically disabled in the final 10% of training episodes. In each goal, performance generally increases with increases in the number of rules. Rule exploration appears to be close to convergence in Figure 6.12a, though performance does continue to increase after specialisation is disabled. In Figure 6.12b, both rule exploration and sampled performance increase in a smooth upward curve, and could potentially continue upwards if the number of training episodes was not fixed at 20,000.

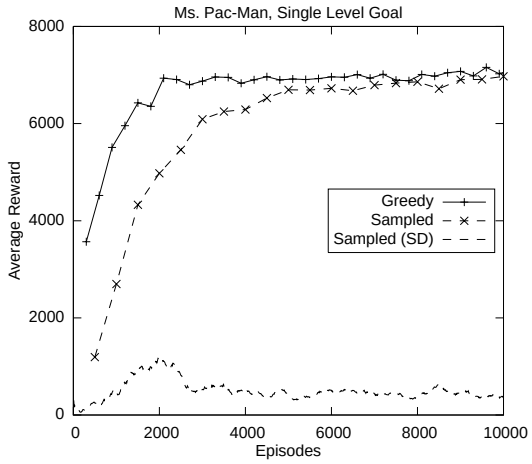
6.3.2 Language Bias

Like Blocks World, this section investigates the effects on Cerrla's performance when the environment representation is altered. The current Ms. Pac-Man environment only has three actions: *moveTo*, *moveFrom*, and *toJunction*. The alternative version of Ms. Pac-Man expands these actions by using the more descriptive actions: *toDot*, *toPowerDot*, *toGhost*, *toGhostCentre*, *toJunction*, *fromPowerDot*, *fromGhost*, and *fromGhostCentre*. These actions introduce a bias towards the actions the agent can perform, as they restrict the agent's possible actions to a subset of Ms. Pac-Man behaviour (e.g. there is no *fromDot* action, which is a practically useless action).

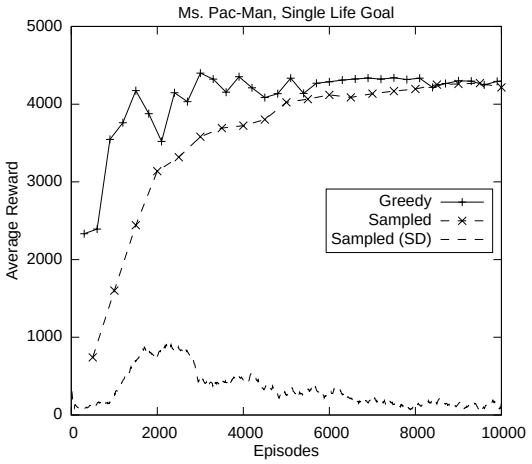
Figure 6.13a, 6.13b, 6.13c and Table 6.8 show the results achieved by Cerrla within the same finite number of episodes for the alternative representation of the Ms. Pac-Man environment. Cerrla's performance (and generated policies) using the alternative representation are not significantly different from that of the normal representation. This is likely because all specialised actions can be semantically replicated by the general Ms. Pac-Man actions. Unlike Blocks World, the alternative representation actually be-

Table 6.8: Averaged results (over ten experiments) regarding Cerrla's performance for the three goals using an alternative representation of the Ms. Pac-Man environment at the end of a fixed number of episodes.

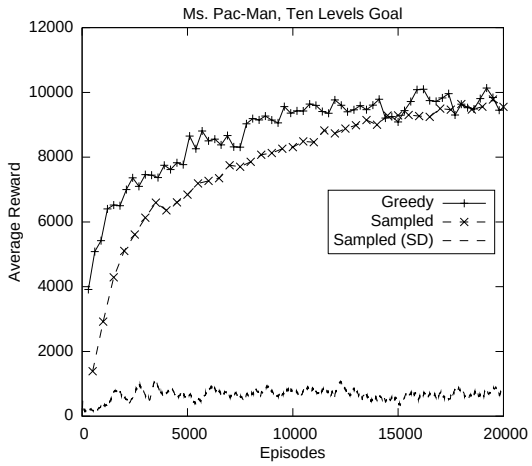
Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Single Level	6973 $\pm$ 398	7118 $\pm$ 258	12–20	58–90	18,612
Single Life	4216 $\pm$ 265	4312 $\pm$ 182	12–17	58–79	10,609
Ten Levels	9555 $\pm$ 748	9524 $\pm$ 1519	12–33	58–142	77,033



(a) *Single Level* goal (alternative representation), limited to 10,000 episodes.



(b) *Single Life* goal (alternative representation), limited to 10,000 episodes.



(c) *Ten Levels* goal (alternative representation), limited to 20,000 episodes.

Figure 6.13: CERRLA's performance for the three goals in an alternative representation of MS. PAC-MAN. Included is the average greedy performance, the average sampled policy performance, and the standard deviation of the average sampled policy performance between experiments. Each experiment runs for a fixed number of episodes.

edible(X), distance(X, N_5) → toGhost(X, N_5)
powerdot(X), distance(X, N_1) → toPowerDot(X, N_1)
dot(X), distance(X, N_3) → toDot(X, N_3)

(a) Example *Single Level* MS. PAC-MAN policy using an alternative representation generated by CERRLA. Achieves an average reward of 7376.

edible(X), distance(X, N_29) → toGhost(X, N_29)
powerdot(X), distance(X, N_25) → toPowerDot(X, N_25)
ghost(X), distance(X, N_30), not edible(X) → fromGhost(X, N_30)

(b) Example *Single Life* MS. PAC-MAN policy using an alternative representation generated by CERRLA. Achieves an average reward of 4527.

dot(X), distance(X, N_3) → toDot(X, N_3)
powerdot(X), distance(X, $(22.4375 \leq N_2 \leq 28.5625)$) → fromPowerDot(X, N_2)
powerdot(X), distance(X, $(25.5 \leq N_2 \leq 50.0)$) → fromPowerDot(X, N_2)
powerdot(X), distance(X, $(13.25 \leq N_2 \leq 37.75)$) → fromPowerDot(X, N_2)
powerdot(X), distance(X, N_1) → toPowerDot(X, N_1)

(c) Example *Ten Levels* MS. PAC-MAN policy using an alternative representation generated by CERRLA. Achieves an average reward of 13,386.

Figure 6.14: Example policies created by CERRLA for the three goals of an alternative representation of MS. PAC-MAN.

gins with fewer distributions and rules than the original representation, which is probably because the alternative representation does not define 'useless' actions (e.g. *fromDot*).

The most obvious difference between the original and alternative performances is the standard deviation of the sampled performances between experiments. The alternative representation has a high initial standard deviation that gradually decreases to a low value, indicating that at the beginning of learning, CERRLA has multiple strategies available to it for achieving high reward (e.g. focus on eating edible ghosts or just eating dots) but it eventually converges to a similarly performing strategy for all ten experiments. In the original representation, the large standard deviation occurs later in the experiment because the algorithm needs to firstly remove useless rules (such as *moveFrom dot*), and the strategies that CERRLA converges to are less similar to each other.

Example policies produced by CERRLA in the alternative Ms. PAC-MAN environment are shown in Figure 6.14a, 6.14b, and 6.14c. The policies have

the same general strategy for each of the goals (eat *ghosts*, eat *ghosts*, and eat *dots* respectively) as the original representation policies.

The comparison between the original and alternative representations suggests that a change of representation has little effect on CERRLA's learning behaviour in the Ms. PAC-MAN environment. If the alternative representation is a fundamental shift in how the environment is represented (e.g. using low level data about exact positions of objects and low-level direction actions), CERRLA may learn significantly different behaviour.

6.3.3 Transfer Learning

Transfer Learning

Because the *Single Level* and *Single Life* problems are essentially subsets of the *Ten Levels* problem, behaviour learned for solving each goal can be used to seed a new CERRLA distribution for the *Ten Levels* goal. A successful policy for completing the first level (or maximising reward within the first level) should improve initial performance within the larger *Ten Levels* goal. Only one experiment is performed here, using the rules from the policy given in Figure 6.11a as the seeded rules, because the policies produced for the *Single Life* goal are very similar to *Single Level* policies.

Figure 6.15 and Table 6.9 show the performance of CERRLA on the *Ten Levels* goal when seeded with the *Single Level* example policy from Figure 6.11a. Although performance is lower than unseeded performance initially, at approximately 10,000 episodes seeded performance outperforms unseeded, converging to an average greedy performance of 16,443, nearly 4000 points higher than unseeded. The initial bias the seeded rules provide towards eating edible ghosts allows CERRLA to focus its learning around those rules, rather than attempting to find an arbitrary strategy, resulting in an even

Table 6.9: Averaged results (over ten experiments) comparing CERRLA's seeded and unseeded learning for the *Ten Levels* goal in the MS. PAC-MAN environment at the end of 20,000 training episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Unseeded	11500 ± 1963	12473 ± 1984	15–58	86–557	500,987
Seeded	12840 ± 2529	16443 ± 1083	17–85	94–797	705,624

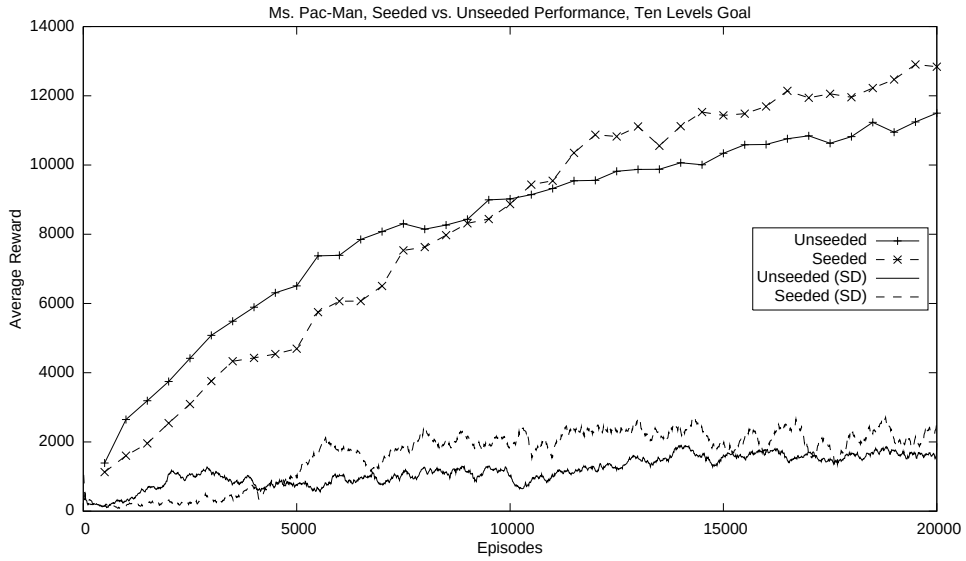


Figure 6.15: CERRLA's performance on the *Ten Levels* goal when seeded with a *Single Level* policy in the MS. PAC-MAN environment. Included is the sampled policy performance (and standard deviation) for seeded and unseeded CERRLA. Each experiment trains for 20,000 episodes.

better learned strategy.

Hand-Coded Rules

As a closer comparison to the hand-coded results presented in Szita and Lörincz (2007), and to evaluate whether the agent learns a better strategy, this experiment examines the effects of seeding CERRLA with the rules in Figure 6.16.

Incorporating the hand-coded rules does not have any significant effect on CERRLA's performance (Figure 6.17 and Table 6.10). CERRLA did not incorporate the hand-coded rules (or specialisations thereof) in all of the final policies produced in experiments and when it did, it was only the first (*moveTo*) hand-coded rule. This rule was typically followed by a more

powerdot(X), distance(X, ($0 \leq N_0 \leq 12.5$*)), ghost(Z), distance(Z, (*$0 \leq N_1 \leq 12.5$*)), not edible(Z)* $\rightarrow$ *moveTo(X,* N_0*)*
powerdot(X), distance(X, ($0 \leq N_0 \leq 12.5$*)), ghost(Z), distance(Z, (*$0 \leq N_1 \leq 12.5$*)), edible(Z)* $\rightarrow$ *moveFrom(X,* N_0*)*

Figure 6.16: The hand-coded rules used to seed CERRLA. Note that the distance sub-ranges are explicitly represented with the *range* function.

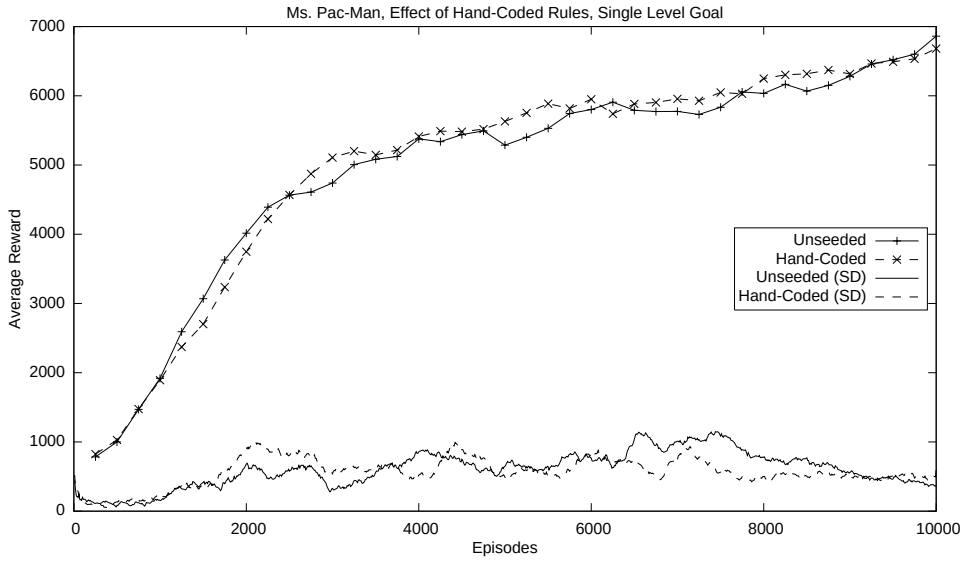


Figure 6.17: CERRLA's performance on the *Single Level* goal using the seeded rules from Figure 6.16 in the MS. PAC-MAN environment. Included is the sampled policy performance (and standard deviation) for normal and hand-coded seeded CERRLA. Each experiment trains for 10,000 episodes.

Table 6.10: CERRLA's performance on the *Single Level* goal using the seeded rules from Figure 6.16 in the MS. PAC-MAN environment at the end of 10,000 training episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Unseeded	6861 $\pm$ 357	7137 $\pm$ 277	15–28	86–184	21,063
Hand-coded	6682 $\pm$ 600	6879 $\pm$ 412	17–30	100–183	16,613

general *moveTo(powerdot, N_0)* rule, resulting in the hand-coded rule being redundant.

Szita and Lörincz (2007)'s CEM algorithm using hand-coded rules achieved an average performance of 8186, compared to CERRLA seeded with hand-coded rules achieving an average greedy performance of 6879. The rules given in Figure 6.16 are not the only rules used in the hand-coded CEM algorithm (there are 42 in total), but they are rules that CERRLA is unable to create due to restrictions in the specialisation process (the restriction to only include action-related rule conditions, see Section 5.5.1). If CERRLA is seeded with the entire set of hand-coded rules, it may achieve a better performance, but because every other hand-coded rule is able to be created by CERRLA, the difference in performance may not be significant.

6.3.4 Ms. Pac-Man Discussion

CERRLA is able to learn behaviour for acting effectively within the Ms. PAC-MAN environment. For the *Single Level* goal, CERRLA learns a slightly better strategy to the comparable CEM algorithm (Szita and Lörincz, 2007) that uses random rules, but it does not learn a better strategy than the hand-coded rules version. In the *Ten Levels* goal (e.g. full Ms. PAC-MAN game), CERRLA's average score is not enough to challenge the current state-of-the-art algorithms,[3] which are able to achieve mean scores $> 31,000$ points per episode (Ikehata and Ito, 2011). It appears that CERRLA's *Ten Levels* performance could be improved with further training, as the gradient of performance in Figure 6.10c is still positive at 20,000 episodes.

The current (and alternative, see Section 6.3.2) representation of Ms. PAC-MAN results in relatively few rules to explore, which in turn results in simplistic, but effective policies. The representation also restricts the rules CERRLA can create. For example, CERRLA cannot create a rule that moves to a *powerdot* when a *ghost* is near because conditions regarding *ghosts* cannot be added to rules concerning *powerdots* in the action. Section 6.3.3 demonstrates the beneficial effects of using previously learned behaviour for a smaller problem as input to a larger problem.

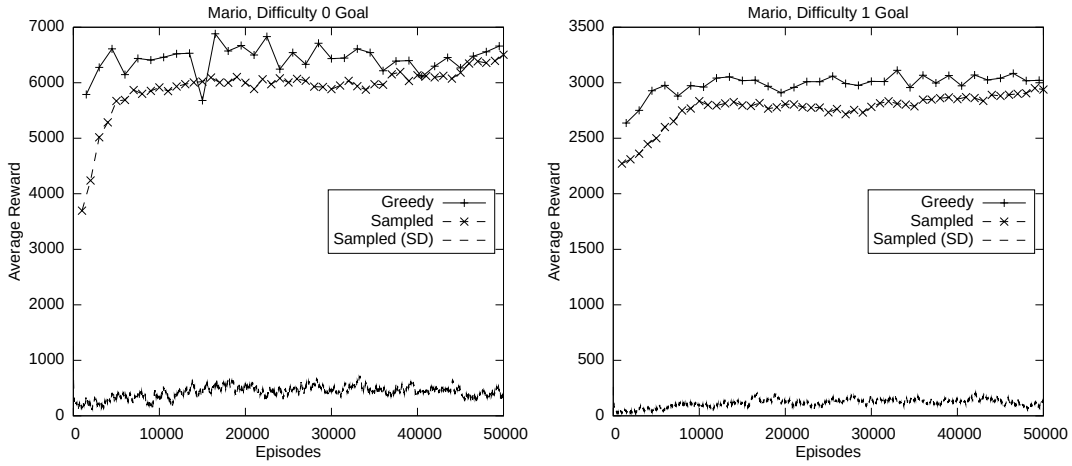
6.4 Mario Evaluation

The MARIO environment contains two difficulty-based goals to be tested upon: *Difficulty 0* (low-threat enemies and level layout) and *Difficulty 1* (more difficult enemies and level layout). In the former difficulty, the enemies are easy to avoid/dispatch, so the primary source of reward is how quickly Mario completes the level. In the latter difficulty, enemies are too numerous and dangerous to avoid, so reward is primarily received by defeating enemies carefully and advancing as far as possible. CERRLA trains for a fixed 50,000 episodes for each goal , which was selected arbitrarily as an ample training period. Section 6.4.1 presents the results of CERRLA's learning for each goal, and Section 6.4.2 investigates the effects of seeding behaviour learned for the *Difficulty 0* goal into the *Difficulty 1* goal.

[3] 'The Ms. Pac-Man Competition,' http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html and similar environment 'Ms. Pac-Man vs. Ghosts Competition,' http://www.pacman-vs-ghosts.net/

6.4.1 Standard CERRLA Performance

Figure 6.18a, 6.18b and Table 6.11 show CERRLA's results for the MARIO environment. The MARIO environment has a large number of rules and distributions to evaluate, resulting in a slow initial learning rate for each goal. There is a large difference between the training time for each goal, where *Difficulty 0* takes over five times longer than *Difficulty 1*. This is probably because Mario is less likely to die in *Difficulty 0*, therefore causing each episode to last longer. CERRLA's performance in MARIO is unable to be compared to other algorithms that use the MARIO environment due to different reward functions and gameplay mechanics but, as a rough comparison, the approximate average performance that I personally achieve is provided using the same reward function defined in Section 3.5.2.



(a) *Difficulty 0* goal, limited to 50,000 episodes. **(b)** *Difficulty 1* goal, limited to 50,000 episodes.

Figure 6.18: CERRLA's performance for the two difficulty goals in MARIO. Included is the average greedy performance, the average sampled policy performance, and the standard deviation of the average sampled policy performance between experiments. Each experiment trains for 50,000 episodes.

Table 6.11: Averaged results (over ten experiments) regarding CERRLA's performance for the two goals of the MARIO environment at the end of 50,000 training episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Diff 0	6504 ± 451	6683 ± 515	46–164	702–2851	122,637
Diff 1	2939 ± 156	3142 ± 83	83–183	1527–3783	25,246

For the *Difficulty 0* goal, CERRLA achieves an average greedy performance of 6683 per episode (and an average *sampled* performance of 6504). As a comparison, I personally achieve approximately 7300 ± 2113. Throughout training, the best elite policy consistently achieves ~ 6500, but the sampled performance does not reach that point until the algorithm is forced to stop branching at episode 45,000 onward. The lower sampled performance is likely due to the constant exploration of new distributions and rules (evidenced in the Rule and Slot Growth figures at the end of this subsection). The sampled performance has a relatively high standard deviation, possibly because the agent is able to complete a level in most cases, but it occasionally gets stuck or is hit by too many enemies, resulting in a lower episode reward.

As a performance comparison, the static 'FORWARDJUMPINGAGENT' (a simple AI that runs and jumps forward continuously) achieves an average reward of 6951 per episode. Clearly, CERRLA has trouble learning high reward behaviour for the *Difficulty 0* goal. CERRLA typically learns policies that gather nearby powerups and stomp on nearby *goombas*, which is a more dangerous and slower strategy than FORWARDJUMPINGAGENT.

The example *Difficulty 0* policy presented in Figure 6.19 is similar to FORWARDJUMPINGAGENT's behaviour, but with additional actions while advancing. The first rule causes Mario to continuously shoot fireballs at *goombas* (when in *fire* mode) and the second rule is a forward-jumping rule. While moving forward and shooting *goombas*, the remaining rules are for: grabbing *coins*, *searching* bricks, jumping onto non-powerup objects (such

marioPower(fire), canJumpOn(X), goomba(X), heightDiff(X, N_0), width(X, N_1), distance(X, N_2) → shootFireball(X, N_2, fire)
canJumpOn(X), heightDiff(X, N_0), distance(X, (37.0 ≤ N_1 ≤ 304.0)), not powerup(X), not enemy(X) → jumpOnto(X, N_1)
canJumpOn(X), canJumpOver(X), coin(X), heightDiff(X, (-171.75 ≤ N_0 ≤ 0.75)), distance(X, (-163.25 ≤ N_1 ≤ -29.75)) → moveTo(X, N_1)
brick(X), heightDiff(X, N_2), distance(X, (-75.75 ≤ N_0 ≤ 74.75)), width(X, (16.0 ≤ N_1 ≤ 16.0)), not canJumpOver(X) → search(X, N_2)
canJumpOn(X), distance(X, N_1), heightDiff(X, (-85.5 ≤ N_0 ≤ 87.0)), not fireFlower(X), not item(X), not powerup(X) → jumpOnto(X, N_1)
canJumpOn(X), coin(X), heightDiff(X, N_0), distance(X, N_1) → jumpOnto(X, N_1)

Figure 6.19: Example *Difficulty 0* MARIO policy generated by CERRLA. Achieves an average reward of 7277.

as bricks, enemies, and coins), and the final rule is another coin collecting rule if nothing else fires. Because this policy causes CERRLA to actively dispatch *goombas* and collect *coins* while advancing forward, it achieves a greater average reward than FORWARDJUMPINGAGENT of 7277.

For the *Difficulty 1* goal, CERRLA achieves an average greedy performance of 3142 per episode (and an average *sampled* performance of 2939). As a comparison, I personally achieve approximately 3130 ± 1010. As with *Difficulty 0*, the best elite policy stays relatively constant at $\sim$ 3000 and the sampled performance only begins to match that performance in the final 10% of training when rule exploration is disabled. The standard deviation of this goal is proportionally much lower than the *Difficulty 0* goal, possibly because the environment is much more difficult to complete, therefore the majority of policies will not receive a bonus reward for completing the level. The FORWARDJUMPINGAGENT achieves an average reward of 3049 per episode. In this case, CERRLA performs slightly better than the FORWARDJUMPINGAGENT, possibly because for this difficulty it is harder to reach the goal by simply jumping forward; completing a *Difficulty 1* level requires more advanced behaviour.

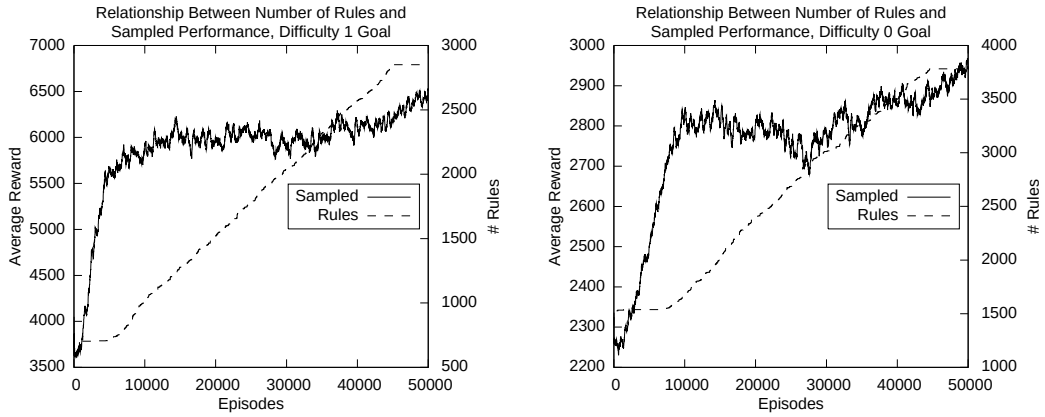
marioPower(fire), squashable(X), heightDiff(X, N_0), width(X, N_1), distance(X, N_2) $\rightarrow$ shootFireball(X, N_2, fire)
marioPower(fire), canJumpOn(X), enemy(X), heightDiff(X, N_0), distance(X, N_2), width(X, $(16.0 \leq N_1 \leq 16.0)$), not flying(X) $\rightarrow$ shootFireball(X, N_2, fire)
marioPower(fire), flying(X), heightDiff(X, N_0), distance(X, N_2), width(X, $(16.0 \leq N_1 \leq 16.0)$), not goomba(X) $\rightarrow$ shootFireball(X, N_2, fire)
marioPower(fire), flying(X), heightDiff(X, N_0), width(X, N_1), distance(X, $(60.5 \leq N_2 \leq 345.0)$) $\rightarrow$ shootFireball(X, N_2, fire)
canJumpOn(X), heightDiff(X, N_0), distance(X, $(46.5 \leq N_1 \leq 304.0)$), not item(X), not pit(X), not redKoopa(X) $\rightarrow$ jumpOnto(X, N_1)
flag(X), distance(X, N_1), heightDiff(X, $(-69.5 \leq N_0 \leq 87.0)$) $\rightarrow$ jumpOnto(X, N_1)
canJumpOn(X), heightDiff(X, N_0), distance(X, N_1), not goomba(X), not item(X), not piranhaPlant(X), not redKoopa(X) $\rightarrow$ jumpOnto(X, N_1)
canJumpOn(X), heightDiff(X, $(-69.5 \leq N_0 \leq 87.0)$), distance(X, $(46.5 \leq N_1 \leq 304.0)$), not redKoopa(X), not passive(X) $\rightarrow$ jumpOnto(X, N_1)
canJumpOn(X), distance(X, N_1), heightDiff(X, $(-69.5 \leq N_0 \leq 8.75)$), not brick(X), not mushroom(X) $\rightarrow$ jumpOnto(X, N_1)
canJumpOn(X), heightDiff(X, N_0), distance(X, $(46.5 \leq N_1 \leq 304.0)$), not pit(X), not redKoopa(X) $\rightarrow$ jumpOnto(X, N_1)

Figure 6.20: Example *Difficulty 1* MARIO policy generated by CERRLA. Achieves an average reward of 3221.

The example *Difficulty 1* policy presented in Figure 6.20 involves more rules than the *Difficulty 0* example policy. It achieves an average reward of 3221. The first four rules are all concerned with shooting enemies (while in *fire* mode) and the fifth and sixth rules result in Mario advancing through the level while shooting (jumping onto things while doing so). The remaining rules are all concerned with jumping onto various things, using negated conditions to exclude particular objects. Interestingly, Mario avoids jumping on *redKoopas* for many of these rules. This could be a coincidence, or it could be an informed choice, as *redKoopas* do not walk off the edges of terrain, so they are only a threat if they are at the same level as Mario. This policy lacks behaviour for picking up and shooting shells, but that may be because Mario is not able to effectively use them without hitting himself. It also does not define explicit rules for jumping over pits, though many rules do have the condition *not* to jump into pits.

Rule and Slot Growth

Due to the large number of rules present in the MARIO environment at the beginning of learning resulting in a larger elites set E, CERRLA does not perform any rule specialisation until approximately episode 6000 and 10,000 in the *Difficulty 0* and *Difficulty 1* goals respectively (Figure 6.21). However, this does not mean the agent was not performing updates to the



(a) The relationship between the number of CERRLA's rules and the performance in MARIO for the *Difficulty 0* goal.

(b) The relationship between the number of CERRLA's rules and the performance in MARIO for the *Difficulty 1* goal.

Figure 6.21: The relationship between the number of CERRLA's rules and the performance for the two MARIO goals.

rule probabilities, as evidenced by the increase in performance. After rule specialisation begins, the performance does not increase by much, indicating that the initial rules are all that are needed for the agent to reach the level of performance it is at when 50,000 episodes have passed.

Rule exploration is roughly linear and does not appear to decrease over time, which may be a result of the large number of predicates and numerical ranges present in the MARIO environment. With the large number of predicates, there are many different specialisation conditions for rules, especially when using negated conditions. With the current representation of MARIO, negated specialisation conditions do not restrict the scope of a rule as much as non-negated conditions do, especially if the negated condition is rare. The large numerical ranges present in MARIO (*distance*, *height*, and *width*) can be split into sub-ranges multiple times before there are any major effects on CERRLA's behaviour, hence the algorithm may continue investigating these sub-ranges for several splits.

6.4.2 Transfer Learning

An obvious area for transfer learning in the MARIO environment is to first train on the *Difficulty 0* goal and then seed the resulting policy into a new CERRLA distribution for the *Difficulty 1* goal. This allows the agent to get a head-start in how to deal with basic level traversal and handling *enemies*.

Figure 6.22 and Table 6.12 show the performance of CERRLA on the *Difficulty 1* goal when seeded with the policy in Figure 6.19. The sampled performance of the seeded and unseeded algorithms is nearly identical, indicating that the seeded rules from the *Difficulty 0* goal are not helpful (nor a hindrance) to the agent's performance. This could be a result of the *Difficulty 0* rules being ineffective in the *Difficulty 1* environment (because they only concern *goombas*) or due to the large number of distributions and

Table 6.12: Averaged results (over ten experiments) comparing CERRLA's seeded and unseeded learning for the *Difficulty 1* goal in the MARIO environment at the end of 50,000 training episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Unseeded	2939 ± 156	3142 ± 83	83–183	1527–3783	25,246
Seeded	2878 ± 83	3070 ± 103	91–177	1654–3736	19,974

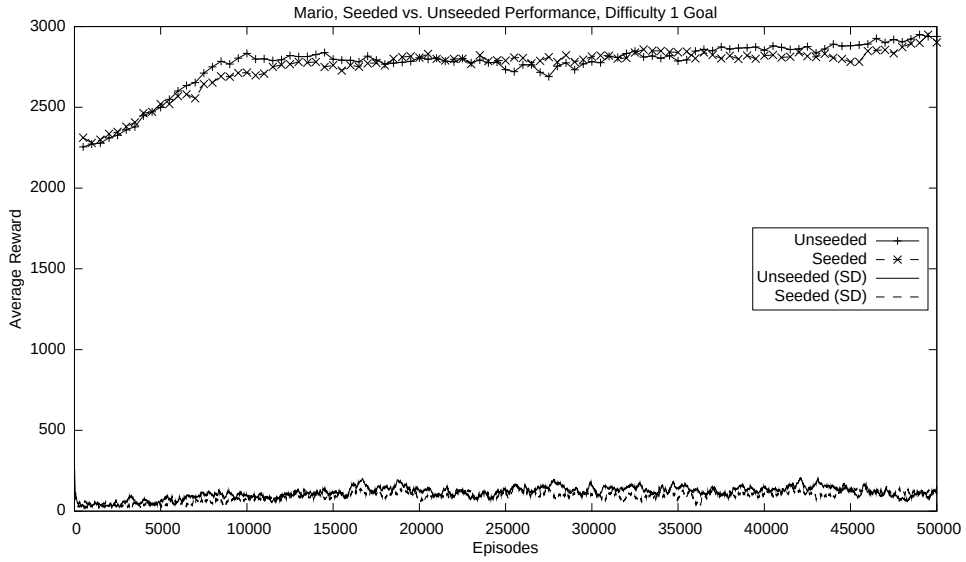


Figure 6.22: CERRLA's performance on the *Difficulty 1* goal when seeded with a *Difficulty 0* policy in the MARIO environment. Included is the sampled policy performance (and standard deviation) for seeded and unseeded CERRLA. Each experiment trains for 50,000 episodes.

rules created for the MARIO environment, reducing the likelihood of using the seeded rules.

The seeded rules do result in a larger average greedy performance than the unseeded rules, though not by much. Because the seeded rules are initialised with $p(D) = 1$, they are likely to be present in all greedy policies throughout learning, so the greedy policies are likely to have a higher minimum performance.

6.4.3 Mario Discussion

The MARIO environment is quite challenging for CERRLA, as it is unable to learn behaviour better than a simple FORWARDJUMPINGAGENT. In CERRLA's defense, FORWARDJUMPINGAGENT is probably the fastest possible agent to reach the goal, if it is lucky enough to survive the level. Nonetheless, CERRLA clearly learns somewhat effective behaviour, as evidenced by the upward slope of the performances.

CERRLA's low performance could be due to multiple factors, such as:

Large search space: MARIO contains many different objects, which result in many rules and distributions. Every rule in the distribution increases

the complexity of the problem and reduces the speed at which CERRLA learns effective behaviour.

Not enough training episodes: There is a possibility that CERRLA could learn better behaviour if given more training episodes, but the MARIO environment suffers from the curse of proportional training time to reward (bounded by level length and time limit), as evident for the *Difficulty 0* goal. Possible solutions include: restricting the number of actions performed per episodes, reducing the length of the levels, or reducing the maximum time available per episode but each measure changes the nature of the problem and could result in the agent learning fundamentally different behaviour.

Stochastic action resolution: When the environment grounds the relational actions CERRLA returns, it does not guarantee that Mario will achieve the action CERRLA selected. This is because there are multiple factors to take into account when resolving actions (e.g. terrain obstacles, enemy movement, Mario's momentum) and the algorithm that resolves Mario's high level actions only resolves the action with an imperfect series of low-level actions. Furthermore, resolution of an action requires multiple time-steps, during which objects in the environment may move or the agent may select different high-level actions to resolve, complicating the process further.

Ineffective representation: The relational representation for MARIO (see Section 3.5.2) may not be expressive enough for CERRLA to create better behaviour that is capable of consistently completing a level. One possible deficiency is that the agent does not have access to a retreating or avoiding action, in the case of enemies that cannot be killed easily. A possible solution is to refine the actions into specialised actions that only deal with a specific type of object (e.g. jump onto enemies, move to coin), but this has been shown in prior environments to have little effect on performance.

Complicated reward function: While not technically a problem, the complex reward function, combined with randomised level layout, results in an impossible-to-create optimal policy (MARIO has been shown to be an NP-hard problem, Aloupis et al. (2012)). Future experiments could separate the problem into different reward functions, such as: fastest level

completion, most enemy kills, and most items collected.

The greedy performance metric in MARIO is not always better than the sampled performance value. In the *Difficulty 0* goal, the greedy policy is significantly affected by probability changes in the distributions. In the *Difficulty 1* goal, it is much closer to the sampled performance, but never rises significantly higher. In each case, it converges to no worse than the sampled performance.

Seeding CERRLA with rules produced from the *Difficulty 0* goal and learning in the *Difficulty 1* goal does not have much of an effect on CERRLA's performance, possibly because the seeded rules are already present in the initial distributions.

In the MARIO environment, CERRLA creates a large number of rules because there are so many different objects in the MARIO environment. The size of the policies may be too large for the CERRLA algorithm to learn in an efficient manner. As seen in Section 6.4.1, no new rules are created until well into learning, which reflects the algorithm's inability to quickly find a reasonable solution upon which to build other rules. Another problem linked to this is that CERRLA is unlikely to achieve β-convergence because the algorithm continues to explore new distributions of rules until it gets to a point where there are either no more specialisations, or further specialisations are clearly less useful than the rule that created them.

A possible solution for this is to hierarchically restrict type-predicate specialisations, such that rules only use type specialisations from the next level down in the type hierarchy (see Section 3.2). E.g. a rule containing *thing(X)* can only specialise that type to the next level down in the type hierarchy (e.g. *brick*, *enemy*, *item*, *goal*, *pit*, or *shell* are the only valid specialisations that are one step lower in the type hierarchy).

6.5 Carcassonne Evaluation

There are multiple different goals in the CARCASSONNE environment: *Single Player*, CERRLA *vs. Random AI*, CERRLA *vs. Static AI* (two, four, and six players), and CERRLA *vs.* CERRLA (two, four, and six players). Section 6.5.1 presents the results CERRLA achieves for each of the goals, as well as evaluating learned behaviour on different goals. Like for previous environ-

ments, behaviour from one goal can be seeded into another, and the results are presented in Section 6.5.2.

6.5.1 Standard CERRLA Performance

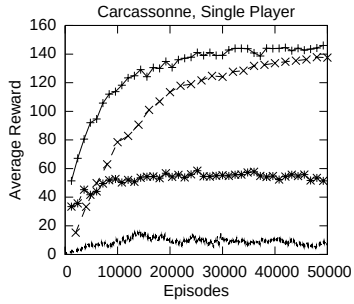
As a comparison, the static AI provided with the *JCloisterZone* program is used on the same goals, citing the average performance achieved over 100 episodes. The static AI selects actions using a one-step look-ahead maximisation strategy to test every possible tile/meeple placement, where the best action is selected as the one which ranks the highest according to an internal score function. This function is tuned towards both increasing game score, and blocking opponent's future moves (by tracking which tiles are left to place). The resulting behaviour is a skilled AI player that presents a challenge for human (and AI) opponents.

We also cite the average performance of a random AI opponent (random tile placement, and 50% chance to place a meeple on random terrain) as a lower bound for performance.

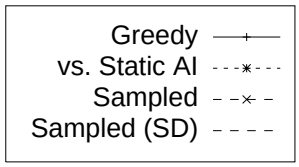
Table 6.13 and Figure 6.23 show the performances of CERRLA for the different CARCASSONNE goals. In every goal CERRLA begins with a relatively small number of distributions, though each distribution contains an aver-

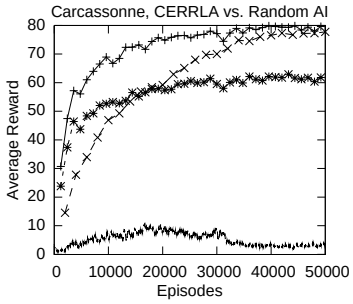
Table 6.13: Averaged results (over ten experiments) regarding CERRLA's performance for the various game-types in the CARCASSONNE environment at the end of 50,000 training episodes. Some behaviour is also tested against the static AI agent (denoted by $\hookrightarrow$).

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Single Player	138 ± 6	146 ± 5	14–90	325–2011	71,066
$\hookrightarrow$ vs. Static AI	—	53 ± 6	—	—	—
CERRLA vs. Random AI	78 ± 4	78 ± 4	14–90	337–2003	34,786
$\hookrightarrow$ vs. Static AI	—	63 ± 5	—	—	—
CERRLA vs. Static AI	60 ± 3	63 ± 3	14–98	335–2160	43,596
CERRLA vs. 3 Static AI	37 ± 3	38 ± 2	14–79	330–1752	38,180
CERRLA vs. 5 Static AI	25 ± 2	27 ± 2	14–99	322–2235	41,336
CERRLA vs. CERRLA	49 ± 3	49 ± 4	14–81	343–1850	26,234
$\hookrightarrow$ vs. Static AI	—	40 ± 5	—	—	—
CERRLA vs. 3 CERRLA	29 ± 3	29 ± 3	14–89	343–2005	17,840
$\hookrightarrow$ vs. 3 Static AI	—	27 ± 2	—	—	—
CERRLA vs. 5 CERRLA	20 ± 2	20 ± 2	15–87	348–1871	13,199
$\hookrightarrow$ vs. 5 Static AI	—	22 ± 1	—	—	—

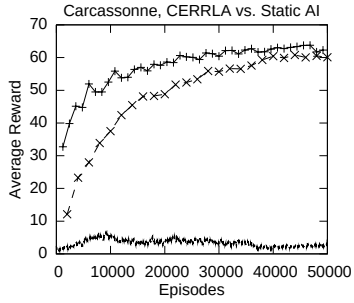


(a) *Single Player* goal, limited to 50,000 episodes. Also evaluated against Static AI opponent.

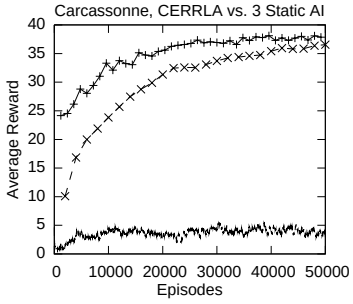




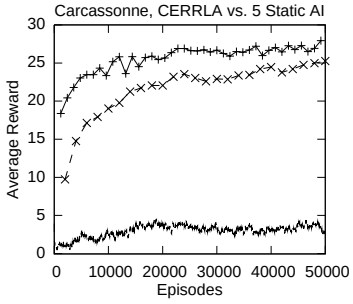
(b) *CERRLA vs. Random AI* goal, limited to 50,000 episodes. Also evaluated against Static AI opponent.



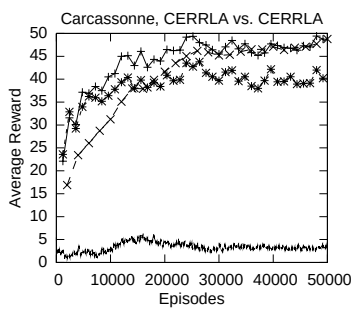
(c) *CERRLA vs. Static AI* goal, limited to 50,000 episodes.



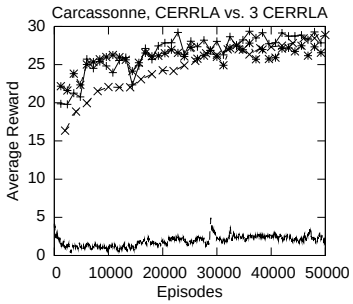
(d) *CERRLA vs. 3 Static AI* goal, limited to 50,000 episodes.



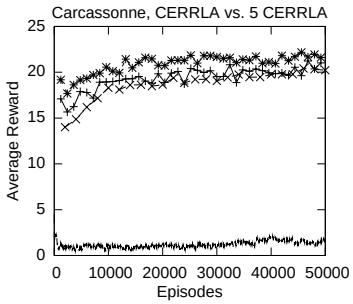
(e) *CERRLA vs. 5 Static AI* goal, limited to 50,000 episodes.



(f) *CERRLA vs. CERRLA* goal, limited to 50,000 episodes. Also evaluated against Static AI opponent.



(g) *CERRLA vs. 3 CERRLA* goal, limited to 50,000 episodes. Also evaluated against 3 Static AI opponents.



(h) *CERRLA vs. 5 CERRLA* goal, limited to 50,000 episodes. Also evaluated against 5 Static AI opponents.

Figure 6.23: CERRLA's performance for the various goals of CARCASSONNE. Included is the average greedy performance, the average sampled policy performance, the standard deviation of the average sampled policy performance between experiments, and in appropriate graphs, the performance against a Static AI. Learning is limited to 50,000 episodes for every goal.

age of $\sim$ 24 rules. The large number of training episodes results in a long training time, but each episode is of a fixed length so the algorithm does not suffer from an increased run time with increased performance. In all goals where there is only a single CERRLA agent per episode and some number of non-CERRLA opponents, training time is approximately 40,000 seconds ($\sim$11 hours per experiment). The boost in training speed is evident in the CERRLA *vs. X* CERRLA goals, where training time is roughly inversely proportional to the number of CERRLA agents, due to an increased number of policy samples received per episode. CERRLA was able to β-converge for some goals before the 50,000 training episodes were completed, indicating that the fixed limit of 50,000 training episodes is sufficient for learning effective behaviour.

A common observation among all CERRLA policies is that they each contain at least one rule that guarantees that CERRLA always selects a tile placement (typically the last rule of the policy), therefore it never ends an episode prematurely and misses out on scoring uncompleted terrain.

On the *Single Player* goal, CERRLA achieves an average greedy performance of 138 points per episode ($\sim$ 1.94 points per tile placed). As a compari-

player(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (4.5 $\leq N_2 \leq$ 8.0)) $\rightarrow$ placeTile(cerrla, Y, Z, W)
player(cerrla), meepleLoc(Y, Z), cloisterZone(?, Z) $\rightarrow$ placeMeeple(cerrla, Y, Z)
player(cerrla), meepleLoc(Y, Z), completed(Z) $\rightarrow$ placeMeeple(cerrla, Y, Z)
player(cerrla), meeplesLeft(cerrla, (0.0 $\leq N_1 \leq$ 3.5)), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 $\leq N_2 \leq$ 6.25)), cloisterZone(Z, ?) $\rightarrow$ placeTile(cerrla, Y, Z, W)
player(cerrla), controls(cerrla, ?), placedMeeples(cerrla, (1.0 $\leq N_0 \leq$ 2.0), ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (3.625 $\leq N_2 \leq$ 5.375)) $\rightarrow$ placeTile(cerrla, Y, Z, W)
meepleLoc(Y, Z), city(Z), tileEdge(Y, ?, Z), open(Z, N_1), worth(Z, (0.0 $\leq N_2 \leq$ 12.0)), not controls(cerrla, ?), not nextTo(?, ?, Z) $\rightarrow$ placeMeeple(cerrla, Y, Z)
player(cerrla), meeplesLeft(cerrla, (2.5 $\leq N_1 \leq$ 4.0)), meepleLoc(Y, Z), worth(Z, (6.0 $\leq N_2 \leq$ 18.0)), not farm(Z) $\rightarrow$ placeMeeple(cerrla, Y, Z)
player(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 $\leq N_2 \leq$ 6.25)) $\rightarrow$ placeTile(cerrla, Y, Z, W)
player(cerrla), meeplesLeft(cerrla, (4.0 $\leq N_1 \leq$ 7.0)), meepleLoc(Y, Z) $\rightarrow$ placeMeeple(cerrla, Y, Z)
player(cerrla), validLoc(Y, Z, W) $\rightarrow$ placeTile(cerrla, Y, Z, W)

Figure 6.24: An example policy generated by CERRLA for the *Single Player* CARCASSONNE goal. Achieves an average reward of 147 and 61 on Single Player and vs. Static AI goals respectively.

son, the static AI achieves an average score of 200 points per episode and the random AI achieves an average of 24 points per episode. Figure 6.24 gives an example policy produced by CERRLA at the end of 50,000 training episodes that achieves an average reward of 147. Due to CARCASSONNE's increased level of complexity, the policy is much larger than policies created for other environments. The policy's behaviour involves placing tiles in tight groups, near *cloisters* (to increase the value of the cloister) or just a default random placement. Meeples are placed on the starting *city*, *cloisters*, *completed* terrain, high *worth* terrain (between 6.0 and 18.0 *worth*), or if the agent still has many meeples left, any valid terrain. The resulting strategy builds compact maps while claiming completed terrain and any other highly valued terrain.

The *Single Player* learned behaviour is also tested against a Static AI opponent to test how effective it is when an opponent is competing for points. Compared to other experiments against a single opponent the resulting average performance of 48 is quite strong and actually performs slightly better than the agent trained in CERRLA *vs.* CERRLA, though training directly against the Static AI produces better results. Section 6.5.2 investigates using this *Single Player* policy as a seed for training against a Static AI.

currentPlayer(cerrla), validLoc(Y, Z, W), numSurroundingTiles(Z, (4.5 ≤ N_2 ≤ 8.0)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (3.625 ≤ N_2 ≤ 5.375)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meepleLoc(Y, Z), not tileEdge(Y, ?, Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), meepleLoc(Y, Z), worth(Z, (4.0 ≤ N_2 ≤ 12.0)), tileEdge(?, ?, Z), not cloisterZone(?, Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), meepleLoc(Y, Z), completed(Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)), cloisterZone(Z, ?) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meeplesLeft(cerrla, (0.0 ≤ N_1 ≤ 3.5)), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 4.5)), cloisterZone(Z, ?) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meeplesLeft(cerrla, (4.0 ≤ N_1 ≤ 7.0)), meepleLoc(Y, Z), tileEdge(Y, ?, Z), not farm(Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), validLoc(Y, Z, W) → placeTile(cerrla, Y, Z, W)

Figure 6.25: Example CERRLA *vs. Random* CARCASSONNE policy created by CERRLA. Achieves an average reward vs. Static AI of 70.

currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (4.5 ≤ N_2 ≤ 8.0)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meepleLoc(Y, Z), worth(Z, (3.0 ≤ N_2 ≤ 6.0)), not nextTo(?, ?, Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), controls(cerrla, ?), meepleLoc(Y, Z), worth(Z, (3.0 ≤ N_2 ≤ 6.0)) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), meeplesLeft(cerrla, (4.0 ≤ N_1 ≤ 7.0)), meepleLoc(Y, Z), worth(Z, (1.5 ≤ N_2 ≤ 4.5)), not completed(Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (3.625 ≤ N_2 ≤ 5.375)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meeplesLeft(cerrla, (4.0 ≤ N_1 ≤ 7.0)), meepleLoc(Y, Z), tileEdge(Y, ?, Z), open(Z, $N_1$1) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)), cloisterZone(Z, ?) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), validLoc(Y, Z, W) → placeTile(cerrla, Y, Z, W)

Figure 6.26: Example CERRLA *vs. Static AI* CARCASSONNE policy created by CERRLA. Achieves an average reward of 65.

validLoc(Y, Z, W), currentPlayer(X), numSurroundingTiles(Z, (4.0 ≤ N_2 ≤ 7.0)), controls(X, ?) → placeTile(X, Y, Z, W)
validLoc(Y, Z, W), currentPlayer(X), meeplesLeft(X, (0.0 ≤ N_1 ≤ 3.5)), numSurroundingTiles(Z, (2.5 ≤ N_2 ≤ 4.0)), controls(X, ?), not cloisterZone(Z, ?) → placeTile(X, Y, Z, W)
meepleLoc(Y, Z), currentPlayer(X), city(Z), controls(X, ?) → placeMeeple(X, Y, Z)
validLoc(Y, Z, W), currentPlayer(X), score(X, (0.0 ≤ N_0 ≤ 35.0)), meeplesLeft(X, (1.75 ≤ N_1 ≤ 5.25)), numSurroundingTiles(Z, (2.5 ≤ N_2 ≤ 4.0)), controls(X, ?), not cloisterZone(Z, ?) → placeTile(X, Y, Z, W)
meepleLoc(Y, Z), currentPlayer(X), cloisterZone(?, Z) → placeMeeple(X, Y, Z)
validLoc(Y, Z, W), currentPlayer(X), meeplesLeft(X, (2.625 ≤ N_1 ≤ 4.375)), controls(X, ?), placedMeeples(X, (1.0 ≤ N_0 ≤ 2.5), ?) → placeTile(X, Y, Z, W)
validLoc(Y, Z, W), currentPlayer(X), numSurroundingTiles(Z, (2.5 ≤ N_2 ≤ 4.0)), controls(X, ?), placedMeeples(X, (1.0 ≤ N_0 ≤ 2.5), ?) → placeTile(X, Y, Z, W)
meepleLoc(Y, Z), currentPlayer(X), open(Z, N_4) → placeMeeple(X, Y, Z)
meepleLoc(Y, Z), currentPlayer(X), farm(Z), worth(Z, (0.0 ≤ N_2 ≤ 5.0)), placedMeeples(X, (1.0 ≤ N_0 ≤ 2.0), ?), nextTo(?, ?, Z) → placeMeeple(X, Y, Z)
validLoc(Y, Z, W), currentPlayer(X) → placeTile(X, Y, Z, W)

Figure 6.27: Example CERRLA *vs.* CERRLA CARCASSONNE policy created by CERRLA. Achieves an average reward vs. Static AI of 49.

In two-player Carcassonne, Cerrla can be trained against three different opponent types: Random AI, Static AI, or Cerrla (itself). Each player in two-player Carcassonne places 35–36 tiles. To present a fair comparison between performances, the learned behaviour for the Cerrla *vs. Random AI* and Cerrla *vs.* Cerrla goals are also evaluated against a Static AI opponent, resulting in an average greedy performance of 63 and 40 respectively.

Of the three goals, Cerrla performs approximately equally well when trained against the Static or Random AI (achieving an average greedy performance of 60 and 63: 1.77 and 1.69 points per tile respectively). It is interesting to see that Cerrla learns equally effective behaviour when trained against a Random AI or a Static AI. This may be a result of Cerrla simply learning an effective strategy for playing against a single generic opponent, rather than learning an effective strategy for a particular type of opponent. In the Cerrla *vs.* Cerrla case, it is much harder to do this because the opponent is also applying the same strategy, resulting in a split reward amongst the particular terrain being claimed. Because the policies are produced from the same distribution, they are generally of equal utility, so the elite samples only contain average policies. In comparison, against a very strong or very weak opponent, the agent can continue to improve whilst the opponent continues with the same strategy.

Against a Random and Static AI, a Static AI achieves an average performance of 99 and 92 respectively and a Random AI achieves an average performance of 22 and 20 respectively. In all three two-player goals, Cerrla outperforms the Random AI, but does not match the Static AI's performance.

Like the performances achieved for the two goals, the Cerrla *vs. Static AI* (Figure 6.26) and Cerrla *vs. Random AI* policies (Figure 6.25) are fairly similar in intent. Both place tiles in tight groups when possible, as well as placing tiles near *cloisters*. Meeples are typically placed on high *worth* (> 3.0) or *cloisters* first (though *cloisters* are expressed as *not tileEdge(Y, ?, Z)* and *not nextTo(?, ?, Z)* — simplification is not able to create equivalencies for these). Other meeple placements go to *completed* terrain and any non-*farm* terrain when Cerrla still has four or more meeples left to place. The Cerrla *vs.* Cerrla also places tiles in clusters, but not explicitly by *cloisters*. It places meeples in *cities*, *cloisters*, *open* terrain, and low worth *farms* (which

seems to be a bad rule). All of the policies also contain the default tile placement rule as the final rule to ensure that a tile placement is always selected.

In four-player CARCASSONNE, CERRLA is trained against three Static AIs or itself (three times). Because there are four separate agents competing for points, completing large scoring terrain is significantly harder. Each player places between 17–18 tiles. Training against the Static AI opponents results in a stronger learned strategy (average greedy performance of 37: 2.08 points per tile), whereas training against itself in a four-player game achieves an average performance of 29 when tested against Static AI opponents. Comparatively, against Static AI opponents, a Static AI achieves an average performance of 50 and a Random AI achieves an average perfor-

currentPlayer(cerrla), controls(cerrla, ?), meepleLoc(Y, Z), worth(Z, (3.0 ≤ N_2 ≤ 6.0)) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (4.5 ≤ N_2 ≤ 6.25)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (3.625 ≤ N_2 ≤ 5.375)) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)), cloisterZone(Z, ?) → placeTile(cerrla, Y, Z, W)
currentPlayer(cerrla), meepleLoc(Y, Z), worth(Z, (1.5 ≤ N_2 ≤ 4.5)) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), meepleLoc(Y, Z), not farm(Z) → placeMeeple(cerrla, Y, Z)
currentPlayer(cerrla), validLoc(Y, Z, W) → placeTile(cerrla, Y, Z, W)

Figure 6.28: Example CERRLA *vs. 3 Static AI* CARCASSONNE policy generated by CERRLA. Achieves an average reward of 39.

meepleLoc(Y, Z), currentPlayer(X), cloisterZone(?, Z) → placeMeeple(X, Y, Z)
meepleLoc(Y, Z), currentPlayer(X), farm(Z), score(X, (0.0 ≤ N_0 ≤ 14.0)), not placedMeeples(X, $N_8$3, ?) → placeMeeple(X, Y, Z)
validLoc(Y, Z, W), currentPlayer(X), controls(X, ?), cloisterZone(Z, ?) → placeTile(X, Y, Z, W)
validLoc(Y, Z, W), currentPlayer(X), numSurroundingTiles(Z, (2.75 ≤ N_2 ≤ 6.25)), controls(X, ?) → placeTile(X, Y, Z, W)
meepleLoc(Y, Z), currentPlayer(X), tileEdge(Y, ?, Z), score(X, (0.0 ≤ N_0 ≤ 7.0)), open(Z, $N_5$69), not farm(Z) → placeMeeple(X, Y, Z)
meepleLoc(Y, Z), currentPlayer(X), not farm(Z) → placeMeeple(X, Y, Z)
validLoc(Y, Z, W), currentPlayer(X), numSurroundingTiles(Z, (1.0 ≤ N_2 ≤ 4.5)) → placeTile(X, Y, Z, W)

Figure 6.29: Example CERRLA *vs. 3* CERRLA CARCASSONNE policy generated by CERRLA. Achieves an average reward vs. Static AI of 29.

mance of 18.

The CERRLA *vs. 3 Static AI* policy (Figure 6.28) has simple meeple placement rules: claim any terrain with *worth* > 1.5 and any non-*farm* terrain. This allows the agent to claim terrain quickly but may result in it running out of meeples. Tile placement is in tight groups, as with previous policies. The CERRLA *vs. 3* CERRLA policy (Figure 6.29) contains some meeple-placing rules for specific situations: placing them on *cloisters*, on a *farm* for that player's first meeple placement, on *open* terrain when it has not scored beyond seven points, and then on any non-*farm* terrain. Tile placement is in clusters as usual. These particular rules allow the agent to claim a *farm* and other terrain early, which might be worth something later on in the game. However, it does not claim high *worth* terrain, which may be why it performs worse than the other four-player policies.

currentPlayer(cerrla), meepleLoc(Y, Z), worth(Z, (2.25 $\leq N_2 \leq$ *6.75))* $\rightarrow$ *placeMeeple(cerrla, Y, Z)*
currentPlayer(cerrla), controls(cerrla, ?), placedMeeples(cerrla, (1.0 $\leq N_0 \leq$ *2.0), ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (4.5* $\leq N_2 \leq$ *8.0))* $\rightarrow$ *placeTile(cerrla, Y, Z, W)*
currentPlayer(cerrla), score(cerrla, (0.0 $\leq N_0 \leq$ *6.5)), meepleLoc(Y, Z), open(Z, (1.0* $\leq N_1 \leq$ *2.0))* $\rightarrow$ *placeMeeple(cerrla, Y, Z)*
currentPlayer(cerrla), meeplesLeft(cerrla, (4.0 $\leq N_1 \leq$ *7.0)), meepleLoc(Y, Z), tileEdge(Y, ?, Z)* $\rightarrow$ *placeMeeple(cerrla, Y, Z)*
currentPlayer(cerrla), controls(cerrla, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, (2.75 $\leq$ *N*$_2$ $\leq$ *6.25))* $\rightarrow$ *placeTile(cerrla, Y, Z, W)*
currentPlayer(cerrla), validLoc(Y, Z, W) $\rightarrow$ *placeTile(cerrla, Y, Z, W)*

Figure 6.30: Example CERRLA *vs. 5 Static AI* CARCASSONNE policy generated by CERRLA. Achieves an average reward of 29.

validLoc(Y, Z, W), currentPlayer(X), cloisterZone(Z, ?) $\rightarrow$ *placeTile(X, Y, Z, W)*
meepleLoc(Y, Z), currentPlayer(X), city(Z) $\rightarrow$ *placeMeeple(X, Y, Z)*
meepleLoc(Y, Z), currentPlayer(X), score(X, (0.0 $\leq N_0 \leq$ *8.0)), worth(Z, (0.0* $\leq N_2 \leq$ *7.0)), not city(Z), not farm(Z), not completed(Z), not nextTo(?, ?, Z)* $\rightarrow$ *placeMeeple(X, Y, Z)*
meepleLoc(Y, Z), currentPlayer(X), controls(X, ?), not city(Z), not farm(Z), not completed(Z), not nextTo(?, ?, Z) $\rightarrow$ *placeMeeple(X, Y, Z)*
validLoc(Y, Z, W), currentPlayer(X), numSurroundingTiles(Z, (2.75 $\leq N_2 \leq$ *6.25)), controls(X, ?), not cloisterZone(Z, ?)* $\rightarrow$ *placeTile(X, Y, Z, W)*
meepleLoc(Y, Z), currentPlayer(X), not completed(Z) $\rightarrow$ *placeMeeple(X, Y, Z)*
validLoc(Y, Z, W), currentPlayer(X), nextTo(Z, ?, ?) $\rightarrow$ *placeTile(X, Y, Z, W)*

Figure 6.31: Example CERRLA *vs. 5* CERRLA CARCASSONNE policy generated by CERRLA. Achieves an average reward vs. Static AI of 22.

In six-player CARCASSONNE, CERRLA can be trained either against five Static AIs, or against itself with six separate policies. Players need to be quick to control and expand terrain, as each player only has 11–12 tile placements per episode. As with the previous goals, training against the Static AI opponents results in a stronger learned strategy (average greedy performance of 25: 2.11 points per tile). Training against itself in a six-player game achieves an average performance of 20, both against itself and Static AI opponents. Comparatively, against Static AI opponents, a Static AI achieves an average performance of 35 and a Random AI achieves an average performance of 12.

The CERRLA *vs. 5 Static AI* policy (Figure 6.30) claims terrain early in the game (when $score \leq 6.5$) or any terrain with $worth \geq 2.25$. These should be sufficient in six-player CARCASSONNE as there are only 11–12 meeple placements per game. The CERRLA *vs. 5* CERRLA policy (Figure 6.31) is more complex, placing meeples on *cities*, *cloisters* (the third rule; the algorithm could not simplify the conditions), or uncompleted terrain.

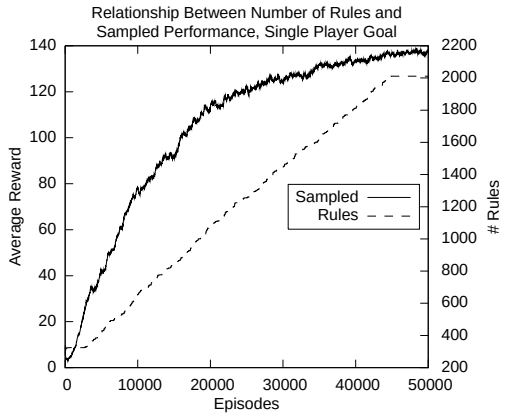
Rule and Slot Growth

CERRLA's rule specialisation in each of the four goals shown in Figure 6.32 is roughly linear, though in all goals, average performance does continue to increase with exploration, indicating a higher performance may be reached with further training episodes, though the increase may not be significant. As each goal uses almost the same rules, rule specialisation begins at approximately 3000 episodes for each goal, and continues upwards with linear growth until specialisation is disabled at 45,000 episodes. Like MARIO, CARCASSONNE has many relation predicates, though most ranges are restricted to relatively small finite integer values between 0 and 10.

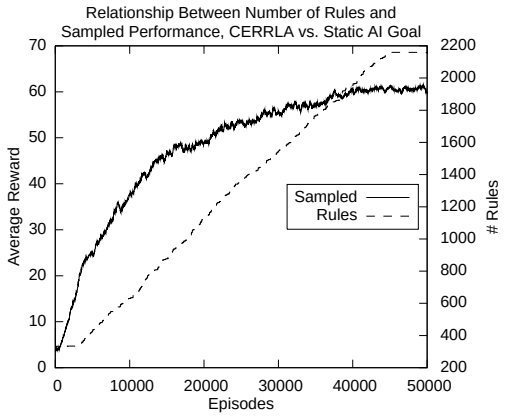
6.5.2 Transfer Learning

Like Ms. PAC-MAN and MARIO, CERRLA has the opportunity to use previously learned behaviour for one goal as a seed for another in the CARCASSONNE environment. In this case, the *Single Player* policy seen in Figure 6.24 is used to seed the algorithm in the CERRLA *vs. Static AI* goal.

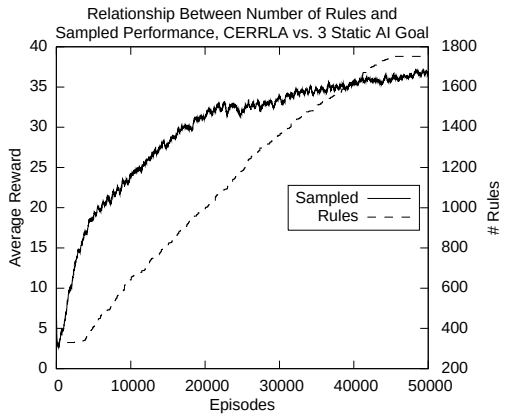
Figure 6.33 and Table 6.14 present the results of seeded vs. unseeded behaviour. The inclusion of proven useful rules at the beginning of learning is



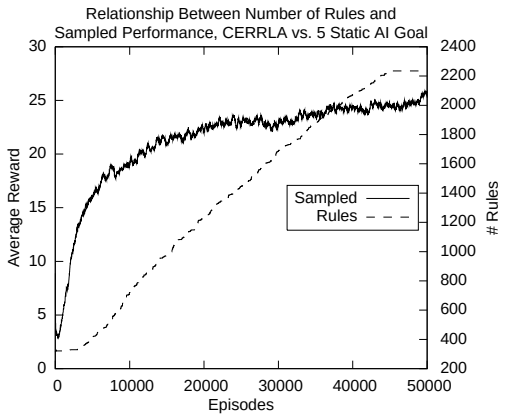
(a) The relationship between the number of CERRLA's rules and the performance in CARCASSONNE for the *Single Player* goal.



(b) The relationship between the number of CERRLA's rules and the performance in CARCASSONNE for the CERRLA *vs. Static AI* goal.



(c) The relationship between the number of CERRLA's rules and the performance in CARCASSONNE for the CERRLA *vs. 3 Static AI* goal.



(d) The relationship between the number of CERRLA's rules and the performance in CARCASSONNE for the CERRLA *vs. 5 Static AI* goal.

Figure 6.32: The relationship between the number of CERRLA's rules and the performance for the *Single Player* and CERRLA *vs. X Static AI* CARCASSONNE goals.

Table 6.14: Averaged results (over ten experiments) comparing CERRLA's seeded and unseeded learning for the CERRLA *vs. Static AI* goal in the CARCASSONNE environment at the end of 50,000 training episodes.

Goal	Sampled	Greedy	# Distributions	# Rules	Time (s)
Unseeded	60 ± 3	63 ± 3	14–98	335–2160	43,596
Seeded	61 ± 2	64 ± 3	23–102	385–2099	48,094

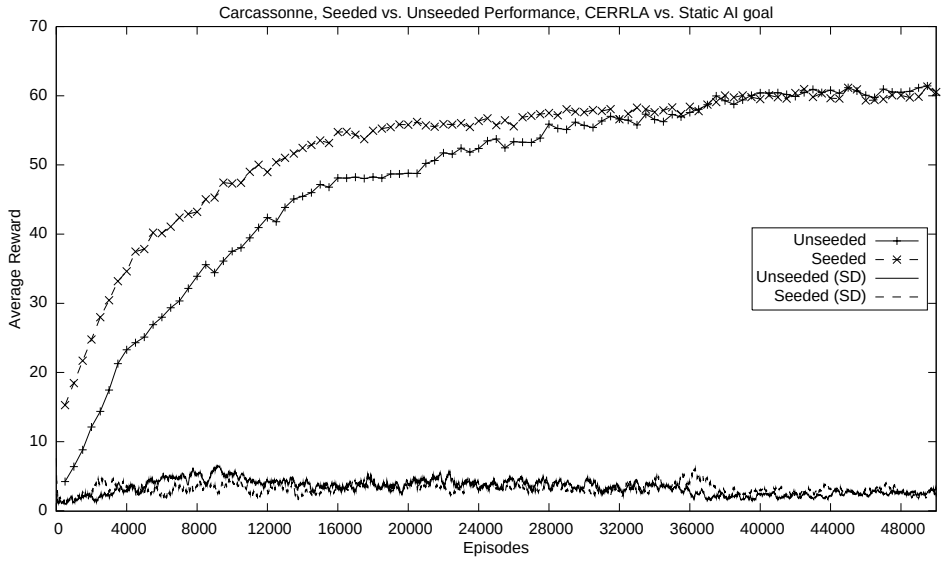


Figure 6.33: CERRLA's performance on the CERRLA *vs. Static AI* goal when seeded with a *Single Player* policy in the CARCASSONNE environment. Included is the sampled policy performance (and standard deviation) for seeded and unseeded CERRLA. Each experiment trains for 50,000 episodes.

clearly useful to CERRLA, as initial seeded performance is higher, but both converge to roughly the same sampled and greedy performances. This performance may be a natural threshold for the current language CERRLA uses to represent the rules.

6.5.3 Carcassonne Discussion

In every scenario of CARCASSONNE (single, two, four, six players), CERRLA learned behaviour for achieving roughly 2.0 points per tile placed (when trained against Static AI opponents). In comparison, the Static AI achieves approximately 2.8 points per tile placed. However, the behaviour learned is not useless, as it easily outperforms a Random AI. For some CARCASSONNE goals, CERRLA was able to β-converge, indicating that the fixed 50,000 training episodes for all experiments are sufficient for the algorithm to learn a final converged behaviour for the goal. Seeding CERRLA with an effective strategy is helpful in initial performance for the CERRLA *vs. Static AI*, but does not result in a greater final performance than unseeded learning.

All the example policies shown include a default tile placing rule (usually near the end of the policy), and most include one or more tile placing rules for placing tiles in clusters. These rules are likely CERRLA's best option for

placing tiles near *open* terrain that it controls, as it cannot represent more complex rules with the current language bias of only using action-related conditions. A rule with a more focused strategy could be:

$$currentPlayer(X), controls(X, V), nextTo(Z, ?, V)\ validLoc(Y, Z, W), city(V) \rightarrow placeTile(X, Y, Z, W)$$

This rule states that the current player X should place tile Y next to a city V that X controls (thereby expanding the city). The key conditions are those concerning V. CERRLA's current language bias does not allow non-action related variables to be shared among conditions or use specialisation conditions that do not contain any action variables, therefore it can never explicitly create such a rule. Allowing CERRLA to use non-action specialisation conditions and share non-action variables would result in many more rules created per specialisation operation, increasing the number of training episodes required for learning, but it could also increase CERRLA's performance on CARCASSONNE and other environments.

Due to the nature of CERRLA vs. CERRLA experiments, the average performance is bound to the average reward received by all players, as every player's policy and score is used to advance CERRLA's learned behaviour. To test if CERRLA is simply learning locally effective policies (only good at outperforming itself), the CERRLA vs. CERRLA policies are also tested against Static AI opponents, but the performance of the policies remain roughly the same, indicating the learned behaviour is equally effective against itself and Static AI opponents. However, these policies still perform worse than those learned for the CERRLA *vs. Static AI* goal, so the decrease in training time is counterbalanced by a decrease in performance.

6.6 Summary and Discussion

This chapter has shown that CERRLA is able to effectively learn relational policies for solving different problems within a range of environments when provided with only the environment's relational specification and the observations made upon each state.

In the BLOCKS WORLD environment, CERRLA consistently learns optimal policies for each of the four goals within BLOCKS WORLD environments of ten blocks or more, achieving better results than all comparison algorithms

and learning the behaviour in a relatively short amount of time. In the Ms. Pac-Man environment, Cerrla achieves a similar performance to the specialised Ms. Pac-Man algorithm it was based upon, but there is much room for improvement in performance. The complex Mario environment proves to be more challenging for Cerrla, as it only performs roughly equal to a 'forward-jumping' agent, but it is still able to create effective policies using Mario's relational representation. Carcassonne is also a difficult environment in which Cerrla does not perform at the same level as a static min-max AI, but it performs much better than random. It is also able to learn effective behaviour for a range of different Carcassonne scenarios involving competing players of different skill levels, where the final policies produced by Cerrla reflect the different strategies required for the different problems.

In general, Cerrla does not achieve better performance than specialised algorithms for the Ms. Pac-Man, Mario, and Carcassonne environments, but in all environments it does identify behaviours for achieving a reasonable performance. If the goal is to achieve the highest performance in a single environment, a specialised, domain-dependent solution would be a better choice than Cerrla. If the goal is to perform well across a range of environments for a range of different goals (approximately at the level of a non-expert human), Cerrla has been shown to be a capable algorithm.

This chapter also investigated the effect language bias has on Cerrla's performance (in the Blocks World and Ms. Pac-Man environments). Cerrla achieves a similar level of performance when using alternative representations of the environment, though these alternatives follow the same general structure as the original representations. Section 6.2.4 showed the benefit of recording *agent observations* for assisting rule simplification (in Blocks World). There is a clear advantage to simplifying the rules, both in terms of performance and evaluation time. Section 6.2.6 investigated how a stochastic environment affect's Cerrla's performance (compared to deterministic Blocks World). Because there are fewer elite samples, Cerrla is more likely to converge to policies that are created earlier in the experiment, which may be sub-optimal solutions.

In the Blocks World environment, Cerrla is quite fast (in training time), even though it processes more episodes than most other algorithms. Be-

cause CERRLA has a fixed strategy per episode, the Rete network representation of the policy's rules efficiently evaluates the relational rules against each state's facts. Other computation includes policy sampling, updating and rule specialisation, but these only occur every three episodes at most. There is also an additional overhead for observing states when performing agent observations, but this typically only occurs in the first few episodes of learning until it has a stable model of the environment. The policies shown throughout this chapter are all reasonably comprehensive and mostly contain non-redundant rules. There are some cases where simplification fails to remove redundant or illegal rule conditions, but in the latter case the CEM decreases the probability of sampling the illegal rule.

Transfer of existing behaviour for smaller goals to larger problems is beneficial in some cases, or no worse than unseeded learning in other cases. The seeded rules provide the CERRLA algorithm with a base strategy to build upon, testing new rules in combination with the seeded rules. However, the additional rules can also slow learning down, as each additional rule increases the complexity of the learning process.

CERRLA's primary drawback is the number of episodes it requires to learn an effective policy. In BLOCKS WORLD this is not a big issue, as the run time is quite fast, but in the larger environments, experiments can take quite some time to complete. Another aspect of large environments is that CERRLA usually needs a fixed episode limit for experiments to complete within an acceptable amount of time. Greedy policies *can* be produced at any point, but in order to measure the algorithm's utility, experimentation requires a termination point.

7

Conclusions and Future Work

This dissertation has described and evaluated the CERRLA algorithm, a direct policy-search RRL algorithm for learning effective goal-achieving behaviour within a range of environments. This chapter summarises the research presented (Section 7.1) and presents conclusions made on the algorithm (Section 7.2). Some of the choices made during this research impose limitations on the settings in which CERRLA is useful, detailed in Section 7.3. Section 7.4 discusses potential future work for CERRLA and the RRL field in general. Finally, Section 7.5 presents a list of the research's contributions.

7.1 Summary

The first three chapters of this thesis 'set the stage' for the remainder of the thesis by presenting an overview of the RRL field, the existing approaches towards solving problems in RRL and related problems, and the language and environments that were used to evaluate the algorithm.

The BLOCKS WORLD environment was an obvious choice for a testing environment, as it is the benchmark testing environment used throughout the RRL field, primarily due to its simplicity and ability to demonstrate core learning concepts. Ms. PAC-MAN was selected as a testing environment because the research presented in this thesis started as an extension to the CEM Ms. Pac-Man playing agent by Szita and Lörincz (2007) and the environment also presents several new problems to the algorithm, such

as numerical attributes, non-deterministic behaviour, and no fixed reward limit. Furthermore, the environment also presents additional problems for a learning algorithm to deal with, such as competing agents (the *ghosts*), numerical values, and non-immediate action resolution. The MARIO and CARCASSONNE environments present much more complex environments, introducing elements such as partial observability of the level, a large number of object types and relations, and an increased set of actions per state.

In Chapter 4 and 5 we looked at the details of how CERRLA creates, optimises, and specialises relational rules for creating effective relational policies for solving goals within relational environments. Chapter 4 focused on the higher level creation of policies by utilising an online modified CEM to rule and distribution probabilities and to guide the exploration process of rule specialisation. Chapter 4 also looked at seeding CERRLA with rules at the beginning of learning to provide a potential boost to CERRLA's performance. This was also easily integrated by using the branching (and seeding) mechanic introduced in Section 4.7.2. Chapter 5 described the agent observations model and how it is used to create the initial RLGG rules, determine the set of all useful specialisation operators, and infer simplification rules by observing the relations between facts within the environment. The inferred model only uses the current state observation and valid actions to incrementally determine the dynamics of both environment-wide and goal-related observations.

Chapter 6 presented the results CERRLA achieves for the four environments, with results for the environmental goals, the effects of seeding rules, and investigating other aspects of the algorithm. CERRLA was shown to be able to learn optimal policies for all four of the BLOCKS WORLD environment goals. The BLOCKS WORLD environment was also used to demonstrate the benefits of learning and applying simplification rules to the algorithm's learned policy rules. In the three game environments, CERRLA learned effective policies for achieving high reward, demonstrating the algorithm's ability to learn behaviour across a range of environments. Each learned policy conveys the agent's behaviour in a simple, rule-by-rule, format with few redundant conditions. Seeding CERRLA with prior behaviour was also shown to be helpful or in the worst case, had no effect on performance.

7.2 Conclusions

The CERRLA algorithm developed throughout this research has been shown to be an effective policy search algorithm, capable of learning optimal policies in BLOCKS WORLD and learning reasonably effective behaviour in the game environments. While it may not perform as well as specialised algorithms, it exhibits good scalability to problems and the ability to be generally applied to a range of different environments. CERRLA's emphasis on minimally complex rules and policies not only results in comprehensible behaviour (relative to say, a neural network or ensemble of models), but also minimises rule evaluation time. The algorithm does not require any human input or other forms of guidance to locate optimal policies because it creates rules and policies in a hypothesis-based manner: create the rule/policy then test it, rather than extract the rule/policy from previous episodes.

CERRLA creates relational policies by creating, optimising, and specialising relational rules in a methodical general-to-specific fashion, using a modified online CEM as the optimisation framework. The use of the CEM confers several benefits: Firstly, compared to value-based methods, no values need to be recorded, as the utility of a policy (and the rules of the policy) within the CEM is dependent on the *relative ranking* of the policy to other sampled policies. Hence, the algorithm only needs to produce policies that have a higher relative ranking to other policies in order to begin updating the distributions. This benefit also guides CERRLA's rule exploration strategy: CERRLA typically begins learning with a small number of general (in terms of rule conditions) rules that are unlikely to be optimal. However, some will invariably be better than others, allowing the algorithm to specialise those rules, creating better rules, and learning a new relative ranking of policy samples, continuing the exploration until no better rules are found.

Secondly, because unused rules and distributions are implicitly negatively updated during the update step of the CEM, the final policies produced by CERRLA will only contain empirically useful rules and distributions. This also allows the algorithm to eliminate rules with illegal conditions that could never subsume the state which were not simplified by the agent observation simplification rules. The opposite case is also beneficial: useful

rules and distributions are positively updated. This results in more samples and, if a rule's probability is high enough, specialisation of the rule. By concentrating on high probability rules, specialisation is restricted to rules that have been shown to be useful.

Finally, the CEM has been shown to be a fast method, at least in Blocks World, with respect to the number of episodes it processes. Except for when Cerrla is scanning the state to update the agent observations model (which generally only occurs in the first few episodes of training), Cerrla does not need to perform significant computation during policy evaluation; most of Cerrla's computation occurs during the update, specialisation and policy sampling processes. Jess's Rete network is ideal for representing Cerrla's static policies because the network only needs to be recreated when a policy is sampled and immediately calculates matches to the policy's rules when the state of the environment is asserted into the network.

The other half of the Cerrla algorithm is the rule creation and specialisation process. Cerrla takes a principled approach towards exploration of the rule space by beginning with the least general generalisation rules (RLGG), then exploring incremental specialisations of interesting rules. These specialisations are guided with the use of agent observations to reduce the number of possible specialisations and simplify any specialisations that result in redundant or illegal rule conditions, resulting in fewer redundant rules to optimise in the CEM aspect of the algorithm. Additionally, the simplification process results in fewer rule conditions, which makes the rule easy-to-comprehend for a human viewer. Without simplification rules, Cerrla's performance and learning speed are significantly lower, and the comprehensibility of the rules decreases.

As shown in Section 6.2.2, Cerrla's relational rule representation of behaviour results in scale-free learning, where the number of episodes required remains relatively constant regardless of the number of objects in the environment. In small-scale problems, this is a disadvantage because Cerrla represents more than it needs to, but in large-scale problems, the state space is abstracted into generalised rules. Scale-free learning is evident in Ms. Pac-Man and Carcassonne too, as the algorithm begins with nearly the same number of rules and distributions (subject to random initial states) even though the scale or nature of the problem changes. In

MARIO, the number of rules and distributions increases with difficulty because higher difficulties introduce new *types* of objects.

An additional advantage of using rule-based behaviour is that rules can be provided to CERRLA at the beginning of learning as something to use for potentially improved performance. Seeding a new set of distributions with previously learned rules is beneficial to the final performance or at least no worse than unseeded. Although seeding rules does increase the number of distributions, the added rules provide an improved starting point that CERRLA can base its learning upon. The simple representation of the rules also makes it easy for a human to manually input rules as seeds, which CERRLA will test and possibly specialise to create more useful rules.

A key component of CERRLA's language bias is to restrict the specialisation conditions to those that contain action variables. This reduces the number of possible rules to evaluate per distribution, but it does also constrain what sort of rules CERRLA can create. In many cases, the arguments of an action are all that need to be defined in a rule's conditions, but there may be cases where defining relations on non-action arguments create useful rules. The problem with allowing such specialisations is that the number of rules created per specialisation operation increases, increasing the number of rules CERRLA needs to examine and decreasing the rate of learning.

CERRLA's main disadvantage is the amount of training episodes required to converge to a final policy. Compared to other policy search algorithms, CERRLA is among the fastest, but it is typically slower than value-based algorithms. However, as a result of the rule-based representation CERRLA uses, the size of the search space for solutions remains constant, regardless of the scale of the environment (unless a change in scale also changes the environment dynamics).

7.3 Limitations

Although CERRLA was designed to be generally applicable across a range of environments, there are some known limitations that may restrict its utility. Some of the design choices made in Chapter 4 and 5 were often a trade-off between minimising the number rules created and the complexity of created rules.

An obvious limitation of CERRLA is the general limitations of RRL itself. An environment must specify a numerical reward which the learning agent uses to guide its learning. However, in many tasks a reward is not obvious (e.g. for a human, reward is often internally represented in the form of pleasure/displeasure) or does not full encapsulate the problem goal (e.g. in Ms. PAC-MAN, there is no explicit reward for avoiding ghosts; the agent must implicitly learn this). The other intrinsic limitation is the *reactiveness* of RRL agents. While it is possible to learn a model of the environment and use plan-based solutions (e.g. MARLIE Croonenborghs et al. (2007)), this can be very difficult in complex environments such as MARIO and CARCASSONNE. CERRLA uses a purely reactive strategy (if *state*, then *action*) and so it will have trouble learning effective behaviour in environments requiring complex plans.

Although policy-search based learning methods have their advantages over value-based methods, there are some limitations. As seen in Section 6.2.3, CERRLA requires significantly more episodes than most value-based methods to converge to a solution in BLOCKS WORLD, though this can also be an advantage — CERRLA's rate of episodic convergence is scale-free with respect to the number of blocks. An additional problem with episodic, policy-search is the problem of environments with 'easily-attainable' goals, that is, goals which can be achieved with a wide range of different behaviours. As evidenced in 3 and 5 block BLOCKS WORLD (Section 6.2.2), the easily attainable goal resulted in an increased number of useful rules, slowing convergence to a single solution and ultimately resulting in a poorer performance than the 10-block variant.

As previously noted in Section 6.2.1, the method in which CERRLA identifies useful/non-useful distributions can be detrimental when a clearly useful rule is not always applicable in the environment. If a rule is rarely applicable, the distribution it is contained within will gradually become less used, even if the rule is useful when it *is* applicable. This creates a bias towards learning policies that only contain the absolutely necessary rules for achieving high performance, preventing CERRLA from taking advantage of lucrative rare situations.

The restriction to only use action and goal-related variables in CERRLA's rules was primarily motivated to reduce the set of possible specialisations,

while defining enough information about the rule's action to make informed decisions. While this does result in a comparatively low number of possible specialisations, it also makes some assumptions about the environment:

1. That each action specifies at least one argument. CERRLA constructs rules by identifying the RLGG for an action where the set of potential conditions each contain at least one action argument. E.g. CERRLA could not learn any behaviour in a low-level, directional-based representation of Ms. PAC-MAN (e.g. *up*, *down*, etc.); it is better suited to learning high-level strategies using abstract actions (the representation seen in Section 3.4.

2. A related assumption is that each action in an environment defines *all* the necessary arguments for making informed decisions. It is assumed that conditions which do not directly reference an action-related argument are not required for making decisions about taking that action, though in Section 6.5.3 this assumption is violated as adding non-action-related specialisation conditions would allow CERRLA to create more powerful rules. But this would also result in a much larger set of possible specialisation conditions, slowing the rate of learning.

3. Individual named objects hold no special significance. CERRLA assumes the relations acting on an object are enough to denote significance and so CERRLA will not be able to directly act on significant, non-goal related objects (except with relations defined with variable arguments).

In general, CERRLA is best at learning high-level behaviour in environments with object-based actions and relationally-described objects.

While not technically a limitation of CERRLA per se, the representation of the environments can be a major factor in the effectiveness of CERRLA's learned behaviour. As noted above, actions should be defined with the objects they act upon but this can be difficult to determine. The relations used to express the state of the environment should also define the connections between objects but to what extent should this go to? For example, in Ms. PAC-MAN and MARIO, the distance predicate defines the distance

between the agent and another object, but what about defining distances between any two given objects? The 'optimal' representation for an environment is unlikely to be answered (or even answerable), but the representation directly affects how CERRLA can represent behaviour.

A related point is *how much* of a state should be described? While CERRLA should theoretically be able to continue to make decisions in POMDPs where only a portion of the state is shown, the rate of learning is likely to be much slower, as the RLGG for an action will be much more general, resulting in more specialisation conditions and therefore more rules to optimise. Furthermore, there would be fewer simplification rules to simplify the rule conditions. If only a portion of the valid actions were given per state, CERRLA should continue to function normally, as the RLGG and specialisation conditions would not change.

7.4 Future Work

CERRLA's limitations described in the previous section leave many different areas open for expansion of the algorithm. We first look at extending CERRLA by adding hierarchical 'modular' learning to the algorithm, which divides a problem into several smaller sub-problems (Section 7.4.1). We then discuss other future work for the CERRLA algorithm (Section 7.4.2) and more broadly, future work for RRL testing environments (Section 7.4.3).

7.4.1 Modular Learning

A prototype extension to CERRLA that was investigated throughout the course of this research was adding the ability to learn and apply *modules*: policies that can be used to achieve sub-goals throughout the course of an episode. By splitting a problem into multiple sub-goals, the curse of dimensionality is reduced, resulting in less work for the algorithm, and complex problems can be broken down into modular sub-problems. CERRLA already produces relational, parameterisable policies (by using goal replacements), so applying modular policies is simply a matter of inserting the appropriate policy into the agent's current policy when the sub-goal needs to be achieved. Sub-goals can be automatically identified as achieving a specific fact (e.g. *clear*(G_0)), or the (non-)existence of an object (e.g. *edible(?)*,

not(coin(?))).

In order to utilise a module, the agent must first learn it. A naive approach is to learn the modules *offline*: learn a module for every possible sub-goal (using parameterisable goal variables where appropriate) before learning the main environmental goal, but this may result in the agent learning modules for useless sub-goals and wastes training time. A less wasteful approach is to simultaneously learn and utilise the module only when it is required *online*. This approach involves maintaining a separate distribution for each goal/sub-goal and learning behaviour using the standard CERRLA algorithm. In either case, the algorithm needs to identify when the sub-goal is achieved and define an internal reward function.

Preliminary results indicate that online modular learning is ineffective and slow, even in the relatively simple BLOCKS WORLD environment (OnG_0G_1 goal). Often the module learned is not optimal, because the internal reward function of -1 per step is not accurate enough to identify truly optimal policies. In BLOCKS WORLD, CERRLA often learns the optimal policy in the main distribution, rather than utilising the module. Future work will focus primarily on defining an effective internal reward, as well as other methods for speeding up the modular learning process.

Modules are automatically inserted instead of presented as *options* (see Sutton et al. (1999b), Croonenborghs et al. (2008), and Section 2.1.2) to avoid increasing the number of actions available to the agent. However, automatic insertion of modular behaviour may not always be necessary. For example, in the MARIO environment, achieving the existence of a *goomba* could be a potential module sub-goal (though ultimately one which the agent has little control over). Whenever the *goomba* is present in a rule, the module for achieving the existence of a *goomba* is automatically inserted.

A possible solution for unwanted module use is to assign each modular sub-goal a probability of use, like distribution usage $p(D)$. Another alternative is to treat modules as actions and incorporate them into CERRLA's rule learning process (i.e. transform them into relational options). Like other rules in CERRLA, each module rule starts with the RLGG conditions for use and CERRLA explores specialisations of the rule, identifying the most useful rules. E.g. for the *clear(G_0)* sub-goal, the RLGG rule would be *not clear(G_0)* $\rightarrow$ *achieve_clear(G_0)* and specialisations add extra conditions

relating to the G_0 argument.

7.4.2 CERRLA-Related Future Work

Other directions for future research regarding the CERRLA algorithm are described below. Each direction considers individual aspects of CERRLA that could be improved or proposes major changes to the core algorithm.

- As mentioned throughout the thesis, CERRLA uses a particular language bias that only uses action-related conditions as specialisation conditions. This restricts the number of possible rules CERRLA can create, reducing the number of episodes required to test all rules, but also restricting what CERRLA can represent. A future modification to CERRLA is to allow any condition to be added as a specialisation. This will result in more rules being created, and therefore longer training times, but the expressivity of the rules increases as well. Preliminary results indicate that the additional conditions dramatically increase the training time and have little effect on performance in BLOCKS WORLD and CARCASSONNE. Future work should focus on expanding the agent observations model to identify which non action-related conditions are potentially useful for each action (rather than add any conditions to rules) and also to broaden the scope of the simplification rules so they can identify redundant or illegal conditions that do not share an argument.

- CERRLA's current method of handling numerical values naively assumes that the values observed within a range are uniformly distributed. The range splitting operation (Section 5.5.2) does not accommodate the *distribution* of observed values. A possible solution is to record the distribution of observed numerical values for the policies in the elite samples. That is, if a certain subset of numerical values frequently occur for the elite policies, then future specialisations should focus on those subranges.

- Currently CERRLA evaluates its policies deterministically (top to bottom), but an alternative is to evaluate them probabilistically by evaluating every rule of the policy and probabilistically selecting an action to perform. Probabilistic action selection could also be weighted by using each distribution's $p(D)$ as a weight for selecting the rule's

action. Probabilistic action selection simplifies policy creation by removing the need for distribution position ($q(D)$) but because *every* rule in a policy is evaluated, policy evaluation time will increase. Probabilistic action selection may also negatively affect the agent's performance by occasionally selecting useless rules. Experiments performed in van Otterlo (2009) show that probabilistic action selection does not perform as well as deterministic action selection for its relational rule policies, but CERRLA may have different results when using weighted probabilistic action selection.

- As stated in Section 4.9, CERRLA's training time could be potentially reduced by using a halting heuristic that prematurely terminates evaluation of a policy if it is unlikely to be an elite (originally performed in Tak (2010)). Because CEM only uses the samples with a value $\geq \gamma$ (elite samples), the algorithm can ignore samples that are unlikely to be within the elite samples.

  It would be difficult to apply this method to environments that only provide a single reward at the end of the episode (such as BLOCKS WORLD), but environments that provide reward throughout evaluation (e.g. Ms. PAC-MAN, MARIO, and CARCASSONNE) can probabilistically terminate policies that are not achieving a reward similar to that of elite policies. Care should be taken not to be too strict in removing policies (else potential elite samples could be removed).

- An extension of CERRLA's transfer learning mechanism is to modify the algorithm to be represented in a similar form as the NPPG algorithm (Kersting and Driessens, 2008). Behaviour is learned by iterating through a shortened CERRLA process of quickly identifying the most useful policy (e.g. reduce the population size N and/or increase α) and using it as a starting point for a new rule distribution via the seeding procedure described in Section 4.8. The rough starting point should make it easier for the agent to create high-reward policies. However, this strategy requires a much faster version of CERRLA to be able to learn behaviour in a reasonable amount of time.

7.4.3 Environment-Related Future Work

Future work regarding the environments defined in this research:

- The relational 'wrappers' for each of the four environments presented in this thesis were created manually by defining and extracting the features, relations and actions from the raw game-state and defining action-resolution procedures for the relational actions. This is a biased and time-consuming task and usually requires direct access to the low-level non-relational model of each environment. The relational representation selected by the person who created the environment wrapper would not necessarily be the same if a different person created it.

  A helpful future preprocessing tool would be the ability to automatically extract a (useful) relational representation from an environment (such as Ms. Pac-Man, etc.), such that an agent utilising this tool could be applied to any task without the need for a relational 'wrapper' to be defined beforehand. The representation would be deterministically created, such that multiple applications of the tool produce the same representation. The tool would need to identify the relevant objects, discern the relations between them (such as distance between), and identify/create an appropriate reward function. Such a tool could accept input directly from the model of an environment (using the environment's variables as sources of information), or take a human-like approach of using the visualisation of the environment as input. An additional benefit of this tool is the ability to extract higher-level features from an existing representation, allowing a learning agent to perform high-level strategies. Creating a tool such as this would be no easy task and would require several years of work, but a successful implementation would be extremely helpful not just for the field of RRL, but also for other fields such as machine learning and computer vision.

- An interesting study into the effects of language bias on learning algorithms could be achieved by reversing the behaviour being learned in a typical RL setup. By treating the environment representation as the target 'behaviour' to be learned, where the representation can change within a certain set of parameters, the most effective environment representation can be identified as the representation in which Cerrla (or any RRL learner) achieves the greatest reward. The results of this experiment would identify the types of environment represen-

tation that the learning algorithm performs best in.

7.5 Contributions

An itemised summary of the research's contributions is presented below:

- In Chapter 3, I provided a relational specification for the Ms. Pac-Man, Mario, and Carcassonne environments. Future RRL algorithms can use these implementations (directly or as an inspiration to an alternative representation) as large scale testing environments.
- In Chapter 4, I presented the direct policy search algorithm named Cerrla that uses a modified version of the Cross-Entropy Method (CEM) to identify the most useful combination of rules for solving goals within relational environments. This algorithm has been shown to be fast, effective and produce human-comprehensible behaviour.
- In Chapter 5, I presented the agent observations model which is able to automatically infer simplification rules for an environment and identify the minimal preconditions and potential specialisation operations required for creating useful relational rules. Cerrla uses this model to remove unnecessary conditions from rules and guide its specialisation process.
- In Chapter 6, I demonstrated Cerrla's utility in four separate environments, where each environment contains multiple goals.

  In the RRL benchmark environment Blocks World, Cerrla performs as well as state-of-the-art existing RRL algorithms for all four goals. It does require more episodes than most approaches, but the number of episodes remains roughly fixed regardless of environment scale.

  In the Ms. Pac-Man environment, Cerrla achieved results similar to the propositional CEM algorithm it was based on, as well as learning behaviour for alternative Ms. Pac-Man goals. Seeding a Cerrla agent with initial behaviour for a simpler goal was also shown to significantly improve performance on more complex goals.

  Experiments on the Mario environment show that the complex state

space and imprecise action resolution make this a difficult environment for CERRLA, though it is able to create a basic strategy for score maximisation.

The CARCASSONNE environment provides a large number of different goals, testing solo and multiplayer performance against a number of different opponents at varying skill levels. In each scenario CERRLA performs well, but never learns a strategy that would be better than the specialised AI.

Additionally, on all environments, I investigated the utility of 'seeding' CERRLA with previously learned or manually defined rules. In all environments seeding increases or, in the worst case, has no negative effect on final performance.

A list of publications regarding the research presented in this thesis can be found in Publications at the beginning of the thesis.

References

Aloupis, G., Demaine, E. D., Guo, A. (2012). Classic nintendo games are (NP-)hard. Tech. rep., arXiv 1203.1895.

Aslam, J. A., Popa, R. A., Rivest, R. L. (2007). On estimating the size and confidence of a statistical audit. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, EVT'07, pp. 8–8. Berkeley, CA, USA: USENIX Association.
http://dl.acm.org/citation.cfm?id=1323111.1323119

Baird, L., Moore, A. (1999). Gradient descent for general reinforcement learning. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pp. 968–974. Cambridge, MA, USA: MIT Press. ISBN 0-262-11245-0.
http://dl.acm.org/citation.cfm?id=340534.340892

Barto, A., Dietterich, T. (2004). Reinforcement learning and its relationship to supervised learning. *Handbook of learning and approximate dynamic programming*, *2*, 47–64.

Barto, A. G., Sutton, R. S., Anderson, C. W. (1990). Neuronlike adaptive elements that can solve difficult learning control problems. In Diederich, J. (Ed.), *Artificial neural networks*, pp. 81–93. Piscataway, NJ, USA: IEEE Press. ISBN 0-8186-2015-3.
http://dl.acm.org/citation.cfm?id=104134.104143

Baum, E. B. (1999). Toward a model of intelligence as an economy of agents.

Mach. Learn., *35*(2), 155–185. doi:10.1023/A:1007593124513.
http://dx.doi.org/10.1023/A:1007593124513

Baxter, J., IVidgell, A., Weaver, L. (1998). KnightCap: A chess program that learns by combining TD(lambda) with game-tree search. In *Machine learning: proceedings of the fifteenth international conference (ICML'98)*, p. 28. Morgan Kaufmann Pub.

Belew, R. K., McInerney, J., Schraudolph, N. N. (1992). Evolving networks: Using the genetic algorithm with connectionist learning. In Langton, C. G., Taylor, C., Farmer, D. J., Rasmussen, S. (Eds.), *Artificial Life II*, pp. 511–547. Redwood City, CA: Addison-Wesley.

Bellman, R. (1956). Dynamic programming and Lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, *42*(10), 767.

Bertsekas, D. P., Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, 1st edn. ISBN 1886529108.

Blockeel, H., De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artif. Intell.*, *101*(1-2), 285–297. doi:10.1016/S0004-3702(98)00034-4.
http://dx.doi.org/10.1016/S0004-3702(98)00034-4

Böhm, N., Kókai, G., Mandl, S. (2005). An evolutionary approach to Tetris. In *The Sixth Metaheuristics International Conference (MIC2005)*.

Boutilier, C., Dearden, R. (1994). Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, AAAI'94, pp. 1016–1022. Menlo Park, CA, USA: American Association for Artificial Intelligence. ISBN 0-262-61102-3.
http://dl.acm.org/citation.cfm?id=199480.199519

Boyan, J. (1992). *Modular neural networks for learning context-dependent game strategies*. Ph.D. thesis, Citeseer.

Boyan, J., Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in neural information processing systems*, *7*, 369–376.

Branavan, S. R. K., Silver, D., Barzilay, R. (2011). Non-linear Monte-Carlo search in Civilization II. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pp. 2404–2410. AAAI Press. ISBN 978-1-57735-515-1. doi: 10.5591/978-1-57735-516-8/IJCAI11-401.
http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-401

Buşoniu, L., Babuška, R., De Schutter, B., Ernst, D. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, Florida: CRC Press.

Chapman, D., Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2*, IJCAI'91, pp. 726–731. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1-55860-160-0.
http://dl.acm.org/citation.cfm?id=1631552.1631569

Chaslot, G., Winands, M. H. M., Szita, I., van den Herik, H. J. (2008). Cross-entropy for Monte-Carlo tree search. *ICGA Journal*, *31*(3), 145–156.

Costa, A., Jones, O. D., Kroese, D. (2007). Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters*, *35*(5), 573 – 580. doi:10.1016/j.orl.2006.11.005.
http://www.sciencedirect.com/science/article/pii/S0167637706001313

Croonenborghs, T., Driessens, K., Bruynooghe, M. (2008). Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the 17th international conference on Inductive logic programming*, ILP'07, pp. 88–97. Berlin, Heidelberg: Springer-Verlag. ISBN 3-540-78468-3, 978-3-540-78468-5.
http://dl.acm.org/citation.cfm?id=1793494.1793509

Croonenborghs, T., Ramon, J., Blockeel, H., Bruynooghe, M. (2007). Online learning and exploiting relational models in reinforcement learning. In *Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, pp. 726–731.

Croonenborghs, T., Ramon, J., Bruynooghe, M. (2004). Towards informed reinforcement learning. In *Proceedings of the ICML2004 workshop on relational reinforcement learning*, pp. 21–26. Citeseer.

Dabney, W., McGovern, A. (2007). Utile distinctions for relational reinforcement learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pp. 738–743. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
http://dl.acm.org/citation.cfm?id=1625275.1625394

De Boer, P., Kroese, D., Mannor, S., Rubinstein, R. (2004). A tutorial on the cross-entropy method. *Annals of Operations Research*, *134*(1), 19–67.

Demaine, E. D., Hohenberger, S., Liben-Nowell, D. (2003). Tetris is hard, even to approximate. In *Proceedings of the 9th annual international conference on Computing and combinatorics*, COCOON'03, pp. 351–363. Berlin, Heidelberg: Springer-Verlag. ISBN 3-540-40534-8.
http://dl.acm.org/citation.cfm?id=1756869.1756918

Dietterich, T., Wang, X. (2001). Support vectors for reinforcement learning. In De Raedt, L., Flach, P. (Eds.), *Machine Learning: ECML 2001*, vol. 2167 of *Lecture Notes in Computer Science*, pp. 600–600. Springer Berlin / Heidelberg. ISBN 978-3-540-42536-6.

Dorigo, M., Colombetti, M. (1998). *Robot shaping: an experiment in behavior engineering*. The MIT Press.

Driessens, K. (2004). *Relational reinforcement learning*. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

Driessens, K., Džeroski, S. (2004). Integrating guidance into relational reinforcement learning. *Machine Learning*, *57*(3), 271–304.

Driessens, K., Džeroski, S. (2005). Combining model-based and instance-based learning for first order regression. In *Proceedings of the 22nd international conference on Machine learning*, pp. 193–200. ACM. ISBN 1595931805.

Driessens, K., Ramon, J. (2003). Relational instance based regression for relational reinforcement learning. pp. 123–130.

Driessens, K., Ramon, J., Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, pp. 97–108. London, UK, UK: Springer-Verlag. ISBN

3-540-42536-5.
http://dl.acm.org/citation.cfm?id=645328.650008

Driessens, K., Ramon, J., Gärtner, T. (2006). Graph kernels and gaussian processes for relational reinforcement learning. *Mach. Learn.*, *64*(1-3), 91–119. doi:10.1007/s10994-006-8258-y.
http://dx.doi.org/10.1007/s10994-006-8258-y

Dzeroski, S. (2001). *Relational Data Mining*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1st edn. ISBN 3540422897.

Džeroski, S., De Raedt, L., Blockeel, H. (1998). Relational reinforcement learning. In Page, D. (Ed.), *Inductive Logic Programming*, vol. 1446 of *Lecture Notes in Computer Science*, pp. 11–22. Springer Berlin Heidelberg. ISBN 978-3-540-64738-6. doi:10.1007/BFb0027307.
http://dx.doi.org/10.1007/BFb0027307

Džeroski, S., De Raedt, L., Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7–52. doi:10.1023/A:1007694015589.
http://dx.doi.org/10.1023/A%3A1007694015589

Fern, A., Yoon, S., Givan, R. (2006). Approximate policy iteration with a policy language bias: solving relational markov decision processes. *J. Artif. Int. Res.*, *25*(1), 75–118.
http://dl.acm.org/citation.cfm?id=1622543.1622546

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17 – 37. doi:10.1016/0004-3702(82)90020-0.
http://www.sciencedirect.com/science/article/pii/0004370282900200

Gallagher, M., Ryan, A. (2003). Learning to play Pac-Man: An evolutionary, rule-based approach. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 4, pp. 2462–2469. IEEE.

Galván-López, E., Swafford, J. M., O'Neill, M., Brabazon, A. (2010). Evolving a ms. pacman controller using grammatical evolution. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pp. 161–170. Berlin, Heidelberg: Springer-Verlag. ISBN 3-642-12238-8, 978-3-642-12238-5. doi:

10.1007/978-3-642-12239-2_17.
http://dx.doi.org/10.1007/978-3-642-12239-2_17

Genesereth, M., Love, N., Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI Magazine*, *26*(2), 62.

Genesereth, M. R., Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 0-934613-31-1.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st edn. ISBN 0201157675.

Grefenstette, J. J., Ramsey, C. L., Schultz, A. C. (1990). Learning sequential decision rules using simulation models and competition. *Mach. Learn.*, *5*(4), 355–381. doi:10.1023/A:1022677607120.
http://dx.doi.org/10.1023/A:1022677607120

Guestrin, C., Koller, D., Gearhart, C., Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. In *Proceedings of the 18th international joint conference on Artificial intelligence*, IJCAI'03, pp. 1003–1010. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
http://dl.acm.org/citation.cfm?id=1630659.1630803

Gupta, N., Nau, D. S. (1992). On the complexity of blocks-world planning. *Artif. Intell.*, *56*(2-3), 223–254. doi:10.1016/0004-3702(92)90028-V.
http://dx.doi.org/10.1016/0004-3702(92)90028-V

Helvik, B. E., Wittner, O. (2001). Using the cross-entropy method to guide/-govern mobile agent's path finding in networks. In *Proceedings of the Third International Workshop on Mobile Agents for Telecommunication Applications*, MATA '01, pp. 255–268. London, UK, UK: Springer-Verlag. ISBN 3-540-42460-1.
http://dl.acm.org/citation.cfm?id=645701.663366

Heyden, C. (2009). *Implementing a computer player for Carcassonne*. Master's thesis, Maastricht University.

Hill, E. F. (2003). *Jess in Action: Java Rule-Based Systems*. Greenwich, CT, USA: Manning Publications Co. ISBN 1930110898.

Holland, J. H. (1992). *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press. ISBN 0-262-58111-6.

Holland, J. H. (1995). Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Computation & Intelligence*, pp. 275–304. Menlo Park, CA, USA: American Association for Artificial Intelligence. ISBN 0-262-62101-0.
http://dl.acm.org/citation.cfm?id=216000.216016

van Hoorn, N., Togelius, J., Schmidhuber, J. (2009). Hierarchical controller learning in a first-person shooter. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, pp. 294–301. Piscataway, NJ, USA: IEEE Press. ISBN 978-1-4244-4814-2.
http://dl.acm.org/citation.cfm?id=1719293.1719344

Howard, R. A. (1960). *Dynamic programming and Markov Processes*. MIT Press, Cambridge, MA.

Hui, K.-P., Bean, N., Kraetzl, M., Kroese, D. (2005). The cross-entropy method for network reliability estimation. *Annals of Operations Research*, *134*(1), 101–118. doi:10.1007/s10479-005-5726-x.
http://dx.doi.org/10.1007/s10479-005-5726-x

Ikehata, N., Ito, T. (2011). Monte-Carlo tree search in Ms. Pac-Man. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pp. 39–46. IEEE.

Jacobs, S., Ferrein, A., Lakemeyer, G. (2005). Unreal Golog bots. In *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pp. 31–36.

Kaelbling, L. P., Littman, M. L., Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artif. Intell.*, *101*(1-2), 99–134. doi:10.1016/S0004-3702(98)00023-X.
http://dx.doi.org/10.1016/S0004-3702(98)00023-X

Kaelbling, L. P., Littman, M. L., Moore, A. W. (1996). Reinforcement learning: a survey. *J. Artif. Int. Res.*, *4*(1), 237–285.
http://dl.acm.org/citation.cfm?id=1622737.1622748

Keith, J., Kroese, D. P. (2002). Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation. In *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, WSC '02, pp. 320–327. Winter Simulation Conference. ISBN 0-7803-7615-3.
http://dl.acm.org/citation.cfm?id=1030453.1030500

Kendall, G., Parkes, A., Spoerer, K. (2008). A survey of NP-complete puzzles. *ICGA Journal*, *31*(1), 13–34.

Kersting, K., Driessens, K. (2008). Non-parametric policy gradients: a unified treatment of propositional and relational domains. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pp. 456–463. New York, NY, USA: ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390214.
http://doi.acm.org/10.1145/1390156.1390214

Kersting, K., Raedt, L. D. (2004). Logical markov decision programs and the convergence of logical TD(lambda). In *ILP*, pp. 180–197.

Khardon, R. (1999). Learning to take actions. *Mach. Learn.*, *35*(1), 57–90. doi:10.1023/A:1007571119753.
http://dx.doi.org/10.1023/A:1007571119753

Kistemaker, S. (2008). Cross-entropy method for reinforcement learning.

Konda, V. R., Tsitsiklis, J. N. (2003). On actor-critic algorithms. *SIAM J. Control Optim.*, *42*(4), 1143–1166. doi:10.1137/S0363012901385691.
http://dx.doi.org/10.1137/S0363012901385691

Kroese, D., Porotsky, S., Rubinstein, R. (2006). The cross-entropy method for continuous multi-extremal optimization. *Methodology and Computing in Applied Probability*, *8*, 383–407. 10.1007/s11009-006-9753-0.
http://dx.doi.org/10.1007/s11009-006-9753-0

Kroese, D., Rubinstein, R., Taimre, T. (2007). Application of the cross-entropy method to clustering and vector quantization. *Journal of Global Optimization*, *37*, 137–157. 10.1007/s10898-006-9041-0.
http://dx.doi.org/10.1007/s10898-006-9041-0

Kullback, S., Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, *22*, 49–86.

Lagoudakis, M. G., Parr, R. (2003). Least-squares policy iteration. *J. Mach. Learn. Res.*, *4*, 1107–1149.
http://dl.acm.org/citation.cfm?id=945365.964290

Langford, J., Zadrozny, B. (2005). Relating reinforcement learning performance to classification performance. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pp. 473–480. New York, NY, USA: ACM. ISBN 1-59593-180-5. doi:10.1145/1102351.1102411.
http://doi.acm.org/10.1145/1102351.1102411

Lanzi, P. L., Stolzmann, W., Wilson, S. W. (Eds.) (2000). *Learning Classifier Systems, From Foundations to Applications*. London, UK, UK: Springer-Verlag. ISBN 3-540-67729-1.

Lavrac, N., Dzeroski, S. (1993). *Inductive Logic Programming: Techniques and Applications*. New York, NY, 10001: Routledge. ISBN 0134578708.

Lloyd, J. W. (1993). *Foundations of Logic Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN 0387181997.

Lucas, S. (2005). Evolving a neural network location evaluator to play Ms. Pac-Man. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pp. 203–210. Citeseer.

Mannor, S., Rubinstein, R., Gat, Y. (2003). The cross entropy method for fast policy search. In *In International Conference on Machine Learning*, pp. 512–519. Morgan Kaufmann.

Martín, M., Geffner, H. (2004). Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, *20*(1), 9–19. doi:10.1023/B:APIN.0000011138.20292.dd.
http://dx.doi.org/10.1023/B:APIN.0000011138.20292.dd

Mayer, H. (2007). Board representations for neural go players learning by temporal difference. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pp. 183–188. IEEE.

Mellor, D. (2008a). *A Learning Classifier System Approach to Relational Reinforcement Learning*. Ph.D. thesis, School of Electrical Engineering and Computer Science, The University of Newcastle, Australia.

Mellor, D. (2008b). A learning classifier system approach to relational reinforcement learning. In Bacardit, J., Bernadó-Mansilla, E., Butz, M. V., Kovacs, T., Llorà, X., Takadama, K. (Eds.), *Learning Classifier Systems*, pp. 169–188. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-540-88137-7. doi:10.1007/978-3-540-88138-4_10.
http://dx.doi.org/10.1007/978-3-540-88138-4_10

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill International Editions. ISBN 0-07-042807-7.

Mohan, S., Laird, J. (2009). Learning to play Mario. Tech. rep., Center for Cognitive Architecture, University of Michigan.

Moore, A. W., Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130. 10.1007/BF00993104.
http://dx.doi.org/10.1007/BF00993104

Morales, E. (2003). Scaling up reinforcement learning with a relational representation. In *Proc. of the Workshop on Adaptability in Multi-agent Systems*, pp. 15–26.

Moriarty, D., Schultz, A., Grefenstette, J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, *11*, 241–276.

Moriarty, D. E., Mikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Mach. Learn.*, *22*(1-3), 11–32. doi:10.1007/BF00114722.
http://dx.doi.org/10.1007/BF00114722

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, *8*(4), 295–318. doi:10.1007/BF03037089.
http://dx.doi.org/10.1007/BF03037089

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing, Special issue on Inductive Logic Programming*, *13*(3-4), 245–286. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1630

Muggleton, S., Feng, C. (1992). Efficient induction of logic programs. *Inductive logic programming*, *38*, 281–298.

Muller, T., van Otterlo, M. (2005). Evolutionary reinforcement learning in relational domains. In *Proceedings of the 7th European Workshop on Reinforcement Learning*. Citeseer.

Natarajan, S., Joshi, S., Tadepalli, P., Kersting, K., Shavlik, J. W. (2011). Imitation learning in relational domains: A functional-gradient boosting approach. In *IJCAI*, pp. 1414–1420.

Nilsson, N. J. (1980). *Principles of artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 0-934613-10-9.

Olson, D. (1993). *Learning to play games from experience: An application of artificial neural networks and temporal difference learning*. Master's thesis, Pacific Lutheran University.

Ormoneit, D., Sen, S. (2002). Kernel-based reinforcement learning. *Mach. Learn.*, *49*(2-3), 161–178. doi:10.1023/A:1017928328829. http://dx.doi.org/10.1023/A:1017928328829

van Otterlo, M. (2004). Reinforcement learning for relational MDPs. In Nowe, A., Lenaerts, T., Steenhaut, K. (Eds.), *Proceedings of the Machine Learning Conference of Belgium and the Netherlands, BeNeLearn '04*, pp. 138–145. Brussels: Brussels. http://doc.utwente.nl/64849/

van Otterlo, M. (2005). A survey of reinforcement learning in relational domains. CTIT Technical Report series TR-CTIT-05-31, Centre for Telematics and Information Technology University of Twente, Enschede.

van Otterlo, M. (2009). *The Logic of Adaptive Behaviour: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*. IOS Press, Amsterdam.

van Otterlo, M., De Vuyst, T. (2009). Evolving and transferring probabilistic policies for relational reinforcement learning. In *BNAIC 2009: Benelux Conference on Artificial Intelligence*.

van Otterlo, M., Kersting, K. (2004). Challenges for relational reinforcement learning. In Tadepalli, P., Givan, R., Driessens, K. (Eds.), *Proceedings of the Workshop on Relational Reinforcement Learning of the International Conference on Machine Learning, ICML '04*, pp. 74–80. Corvallis: Oregon State University.
http://doc.utwente.nl/64887/

Perez, D., Nicolau, M., O'Neill, M., Brabazon, A. (2011). Evolving behaviour trees for the Mario AI competition using grammatical evolution. In *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part I*, EvoApplications'11, pp. 123–132. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-642-20524-8.
http://dl.acm.org/citation.cfm?id=2008402.2008417

Pfeiffer, M. (2004). Reinforcement learning of strategies for Settlers of Catan. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education, Reading, UK.*
http://eprints.pascal-network.org/archive/00000425/

Pittman, J. (2011). The Pac-Man dossier. http://home.comcast.net/~jpittman2/pacman/pacmandossier.html. [Online; accessed 15-April-2012].

Plotkin, G. D. (1970). A note on inductive generalization. *Machine Intelligence*, *5*, 153–163.

Ponsen, M., Muñoz-Avila, H., Spronck, P., Aha, D. (2006). Automatically generating game tactics through evolutionary learning. *AI Magazine*, *27*(3), 75.

Potter, M. A., De Jong, K. A. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evol. Comput.*, *8*(1), 1–29. doi:10.1162/106365600568086.
http://dx.doi.org/10.1162/106365600568086

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1st edn. ISBN 0471619779.

Ramon, J., Driessens, K., Croonenborghs, T. (2007). Transfer learning in reinforcement learning problems through partial policy recycling. In *Proceedings of the 18th European conference on Machine Learning*, ECML '07, pp. 699–707. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-540-74957-8. doi:10.1007/978-3-540-74958-5_70.
http://dx.doi.org/10.1007/978-3-540-74958-5_70

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, *12*(1), 23–41. doi:10.1145/321250.321253.
http://doi.acm.org/10.1145/321250.321253

Rubinstein, R. Y. (1997). Optimization of computer simulation models with rare events. *European Journal of Operational Research*, *99*(1), 89 – 112. doi:10.1016/S0377-2217(96)00385-2.
http://www.sciencedirect.com/science/article/pii/S0377221796003852

Russell, S. J., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edn. ISBN 0137903952.

Sammut, C. (1998). Prolog, refinements and RLGG's. In *Proceedings of the 8th International Workshop on Inductive Logic Programming*, ILP '98, pp. 225–234. London, UK, UK: Springer-Verlag. ISBN 3-540-64738-4.
http://dl.acm.org/citation.cfm?id=647998.742776

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II: recent progress. *IBM J. Res. Dev.*, *11*(6), 601–617. doi:10.1147/rd.116.0601.
http://dx.doi.org/10.1147/rd.116.0601

Sanner, S. (2005). Simultaneous learning of structure and value in relational reinforcement learning. In *Proceedings of the ICML 2005 Workshop on Rich Representations for Reinforcement Learning*.

Sharma, M., Holmes, M., Santamaria, J., Irani, A., Isbell, C., Ram, A. (2007). Transfer learning in real-time strategy games using hybrid CBR/RL. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pp. 1041–1046. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
http://dl.acm.org/citation.cfm?id=1625275.1625444

Silver, D., Sutton, R., Müller, M. (2007). Reinforcement learning of local shape in the game of go. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pp. 1053–1058. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
http://dl.acm.org/citation.cfm?id=1625275.1625446

Slaney, J., Thiébaux, S. (2001). Blocks World revisited. *Artificial Intelligence*, *125*(1-2), 119 – 153. doi:DOI:10.1016/S0004-3702(00)00079-5.

Smith, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 1*, IJCAI'83, pp. 422–425. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
http://dl.acm.org/citation.cfm?id=1623373.1623474

Stanley, K. O., Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evol. Comput.*, *10*(2), 99–127. doi:10.1162/106365602320169811.
http://dx.doi.org/10.1162/106365602320169811

Stone, P., Kuhlmann, G., Taylor, M. E., Liu, Y. (2005a). Keepaway soccer: From machine learning testbed to benchmark. In *RoboCup*, pp. 93–105.

Stone, P., Sutton, R., Kuhlmann, G. (2005b). Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, *13*(3), 165–188.

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bull.*, *2*(4), 160–163. doi:10.1145/122344.122377.
http://doi.acm.org/10.1145/122344.122377

Sutton, R. S., Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press. ISBN 0262193981.

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y. (1999a). Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, pp. 1057–1063.

Sutton, R. S., Precup, D., Singh, S. (1999b). Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artif.*

Intell., *112*(1-2), 181–211. doi:10.1016/S0004-3702(99)00052-1.
http://dx.doi.org/10.1016/S0004-3702(99)00052-1

Szepesvári, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *4*(1), 1–103.

Szita, I., Chaslot, G., Spronck, P. (2009). Monte-carlo tree search in Settlers of Catan. In *ACG*, pp. 21–32.

Szita, I., Lörincz, A. (2006). Learning Tetris using the noisy cross-entropy method. *Neural Comput.*, *18*(12), 2936–2941. doi:10.1162/neco.2006.18.12.2936.
http://dx.doi.org/10.1162/neco.2006.18.12.2936

Szita, I., Lörincz, A. (2007). Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. *J. Artif. Int. Res.*, *30*(1), 659–684.
http://dl.acm.org/citation.cfm?id=1622637.1622654

Szita, I., Lörincz, A. (2008). Online variants of the cross-entropy method. Tech. rep., arXiv:0801.1988.

Szita, I., Ponsen, M., Spronck, P. (2008). Keeping adaptive game AI interesting. In *CGAMES 2008*, pp. 70–74.

Tadepalli, P., Givan, R., Driessens, K. (2004). Relational reinforcement learning: An overview. In *Proceedings of the ICML-2004 Workshop on Relational Reinforcement Learning*, pp. 1–9.

Tak, M. (2010). The cross-entropy method applied to samegame.
http://www.unimaas.nl/games/files/bsc/Tak_Bsc-paper.pdf

Taylor, M. E., Stone, P. (2005). Behavior transfer for value-function-based reinforcement learning. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pp. 53–59. New York, NY, USA: ACM. ISBN 1-59593-093-0. doi:10.1145/1082473.1082482.
http://doi.acm.org/10.1145/1082473.1082482

Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, *6*(2), 215–219. doi:10.1162/

neco.1994.6.2.215.
http://dx.doi.org/10.1162/neco.1994.6.2.215

Thiery, C., Scherrer, B. (2009). Improvements on Learning Tetris with Cross Entropy. *International Computer Games Association Journal*, *32*, 23–33.
http://hal.inria.fr/inria-00418930

Thrun, S. (1995). Learning to play the game of Chess. In *Advances in Neural Information Processing Systems 7*, pp. 1069–1076. The MIT Press.

Togelius, J., Karakovskiy, S., Koutník, J., Schmidhuber, J. (2009). Super mario evolution. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pp. 156–161. IEEE.

Togelius, J., Lucas, S. (2006). Evolving robust and specialized car racing skills. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pp. 1187–1194. IEEE.

Utgoff, P., Precup, D. (1998). *Constructive Function Approximation*, vol. 453, chap. 14, p. 219. Kluwer Academic Publishers.

Viglietta, G. (2012). Gaming is a hard job, but someone has to do it! Tech. rep., arXiv:1201.4995.

Walker, T., Shavlik, J., Maclin, R. (2004). Relational reinforcement learning via sampling the space of first-order conjunctive features. In *Proceedings of the ICML Workshop on Relational Reinforcement Learning, Banff, Canada*.

Wang, C., Joshi, S., Khardon, R. (2008). First order decision diagrams for relational MDPs. *J. Artif. Int. Res.*, *31*(1), 431–472.
http://dl.acm.org/citation.cfm?id=1622655.1622668

Wang, X., Dietterich, T. G. (1999). Efficient value function approximation using regression trees. In *In Proceedings of the IJCAI Workshop on Statistical Machine Learning for Large-Scale Optimization*.

Watkins, C. J. C. H., Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292. 10.1007/BF00992698.
http://dx.doi.org/10.1007/BF00992698

Whiteson, S., Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *J. Mach. Learn. Res.*, *7*, 877–917.
http://dl.acm.org/citation.cfm?id=1248547.1248578

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., Kohl, N. (2005). Automatic feature selection in neuroevolution. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pp. 1225–1232. New York, NY, USA: ACM. ISBN 1-59593-010-8. doi: 10.1145/1068009.1068210.
http://doi.acm.org/10.1145/1068009.1068210

Whitley, D., Dominic, S., Das, R., Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Mach. Learn.*, *13*(2-3), 259–284. doi:10.1007/BF00993045.
http://dx.doi.org/10.1007/BF00993045

Wiering, M., van Otterlo, M. (Eds.) (2012). *Reinforcement Learning: State-Of-The-Art*, vol. 12. Springer-Verlag New York Incorporated.

Wierstra, D., Schmidhuber, J. (2007). Policy gradient critics. In *Proceedings of the 18th European conference on Machine Learning*, ECML '07, pp. 466–477. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-540-74957-8. doi: 10.1007/978-3-540-74958-5_43.
http://dx.doi.org/10.1007/978-3-540-74958-5_43

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, *8*(3-4), 229–256. doi: 10.1007/BF00992696.
http://dx.doi.org/10.1007/BF00992696

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evol. Comput.*, *3*(2), 149–175. doi:10.1162/evco.1995.3.2.149.
http://dx.doi.org/10.1162/evco.1995.3.2.149

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, *34*(4), 286 – 295. doi: 10.1016/S0019-9958(77)90354-0.
http://www.sciencedirect.com/science/article/pii/S0019995877903540

Yoon, S., Fern, A., Givan, R. (2002). Inductive policy selection for first-order MDPs. In *Proceedings of the Eighteenth conference on Uncertainty in artificial*

intelligence, UAI'02, pp. 568–576. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1-55860-897-4.
http://dl.acm.org/citation.cfm?id=2073876.2073944