Computationally Efficient Relational Reinforcement Learning

by

Mitchell Keith Bloch

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2018

Doctoral Committee:

Professor John E. Laird, Chair
Professor Satinder Singh Baveja
Professor Edmund H. Durfee
Professor Richard L. Lewis

Baby at Play by Thomas Eakins, 1876, is currently on display at the National Gallery of Art, West Main Floor Gallery 65. [Commons, 2016]

Mitchell Keith Bloch

bazald@umich.edu

ORCID iD: 0000-0002-7219-4786

This thesis is dedicated to compassion. It is only through the compassion of others that I have made it this far. It is my intention that I too will always have compassion for others.

"Be pitiful, for every man [*sic*] is fighting a hard battle."

-Ian Maclaren (1897)

ACKNOWLEDGMENTS

I wish to thank the Computer Science and Engineering department and the University of Michigan at large for providing a stimulating and supportive environment for me to pursue my studies. I'm very grateful to my adviser and committee chair, John E. Laird, for his many years of kindness, patience, and support, and to my committee members, Satinder S. Baveja, Edmund H. Durfee, and Richard L. Lewis, for their help and guidance. Additionally, thank you to my friends and family who have supported me over the years. I appreciate you all.

TABLE OF CONTENTS

LIST OF APPENDICES

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

LIST OF RESEARCH QUESTIONS

LIST OF HYPOTHESES

LIST OF ABBREVIATIONS

ϵ-greedy Epsilon-greedy

***k*-d tree** *k*-dimensional tree

***k*-d trie** *k*-dimensional trie

LIST OF ACRONYMS

aHTC adaptive Hierarchical Tile Coding

AI Artificial Intelligence

ARgPE Average Regret Per Episode

ARtPE Average Return Per Episode

ATC Adaptive Tile Coding

Carli-Prop Carli for Propositional Representations

Carli-RRL Carli for Relational Reinforcement Learning

CATDE Cumulative Absolute Temporal Difference Error

CMAC Cerebellar Model Arithmetic Computer

CPU Central Processing Unit

DAG Directed Acyclic Graph

FELSC Feature Encoded in the Last Scope Convention

FOL First Order Logic

FOLDT First Order Logical Decision Tree

GVF Generalized Value Function

HAM Hierarchies of Abstract Machines

HOG Higher Order Grammar

HRL Hierarchical Reinforcement Learning

HTC Hierarchical Tile Coding

IOC Identical Ordering Convention

LALR Look-Ahead Left-to-Right

LR Left-to-Right

MDP Markov Decision Process

MSPBE Mean-Square Projected Bellman Error

naHTC non-adaptive Hierarchical Tile Coding

POMDP Partially Observable Markov Decision Process

RAM Random Access Memory

RBF Radial Basis Function

RL Reinforcement Learning

RRL Relational Reinforcement Learning

TD Temporal Difference

TDIDT Top-Down Induction of Decision Trees

TDU Temporal Difference Updates

TILDE Top-down Induction of Logical DEcision trees

VEC Value Estimate Contributions

WCTPS Wall-Clock Time Per Step

WME Working Memory Element

XOR Exclusive OR

LIST OF SYMBOLS

α learning rate

a current action

a^* best action

$\mathcal{A}$ action space

δ Temporal Difference (TD) Error

γ discount rate

$P_a(s, s')$ state transition function

r immediate reward

$R_a(s, s')$ reward function

s current state

s' successor state

$\mathcal{S}$ state space

$\mathcal{Q}(s, a)$ Q-function

ABSTRACT

Relational Reinforcement Learning (RRL) is a technique that enables Reinforcement Learning (RL) agents to generalize from their experience, allowing them to learn over large or potentially infinite state spaces, to learn context sensitive behaviors, and to learn to solve variable goals and to transfer knowledge between similar situations. Prior RRL architectures are not sufficiently computationally efficient to see use outside of small, niche roles within larger Artificial Intelligence (AI) architectures. I present a novel online, incremental RRL architecture and an implementation that is orders of magnitude faster than its predecessors. The first aspect of this architecture that I explore is a computationally efficient implementation of an adaptive Hierarchical Tile Coding (aHTC), a kind of Adaptive Tile Coding (ATC) in which more general tiles which cover larger portions of the state-action space are kept as ones that cover smaller portions of the state-action space are introduced, using k-dimensional tries (k-d tries) to implement the value function for non-relational Temporal Difference (TD) methods. In order to achieve comparable performance for RRL, I implement the Rete algorithm to replace my k-d tries due to its efficient handling of both the variable binding problem and variable numbers of actions. Tying aHTCs and Rete together, I present a rule grammar that both maps aHTCs onto Rete and allows the architecture to automatically extract relational features in order to support adaptation of the value function over time. I experiment with several refinement criteria and additional functionality with which my agents attempt to determine if rerefinement using different features might allow them to better learn a near optimal policy. I present optimal results using a value criterion for several variants of Blocks World. I provide transfer results for Blocks World and a scalable Taxicab domain. I additionally introduce a Higher Order Grammar (HOG) that grants online, incremental RRL agents additional flexibility to introduce additional variables and corresponding relations as needed in order to learn effective value functions. I evaluate agents that use the HOG on a version of Blocks World and on an Adventure task. In summary, I present a new online, incremental RRL architecture, a grammar to map aHTCs onto the Rete, and an implementation that is orders of magnitude faster than its predecessors.

CHAPTER 1

Introduction

An important challenge in the area of Artificial Intelligence (AI) can be cast as the combined problem of describing problems, tasks, or environments [Bellman, 1957a; Sondik, 1971; Aeronautiques *et al.*, 1998] and enabling agents to learn policies to solve these problems, perform these tasks, or behave optimally (or at least very well) in these environments [Bellman, 1957b; Sutton, 1988]. The problem of learning policies that define an agent's behavior as a function of the state of the environment and a reward signal defines a Reinforcement Learning (RL) problem. The RL problem is generally applicable to everything from simple games, to industrial process automation, to modeling of human thought processes.

I concern myself with RL agents that learn to predict value estimates for actions by updating a value function over time and that derive their policies from those value estimates by choosing greedily (selecting actions that have the highest value estimates) or greedily with some chance of exploration. I am additionally interested in agents that learn online (meaning that one cares about performance during the learning process) and efficiently (meaning that one cares about how much time or processing is required) and that are capable of pursuing goals that differ from episode to episode. These interests inform many of the choices I make, including my evaluation criteria (Section 1.2), my choice of value function representation (Section 1.3), and my approaches to manipulating the value function representations of my agents over time (Section 1.3.4.3).

Different value function representations ranging from tabular representations (Section 1.3.2), to linear combinations of features (Section 1.3.3), to tile codings (Section 1.3.4), to relational representations (Section 1.3.5) vary significantly in learning efficiency and computational efficiency (Section 1.2). Moving from tabular representations to more sophisticated representations involving linear function approximation, tiling coding, and/or relational representations enables RL agents to learn policies that generalize, allowing them to learn over larger or potentially infinite state spaces, which is important for many environments that are infinite, simply large, or continuously valued and difficult or impossible to perfectly discretize.

Relational Reinforcement Learning (RRL), in which an agent's value function or policy is defined over a set of conjunctions of relations about objects in the environment (Section 1.3.5),

enables agents to learn policies that are context sensitive. This allows them to learn to address variable goals and to transfer knowledge between similar situations.

Unfortunately, preceding RRL architectures, such as the one pioneered by Džeroski *et al.* [2001], are not sufficiently computationally efficient to see use outside of small, niche roles in larger AI architectures [Lang *et al.*, 2012a; Sridharan *et al.*, 2016a; Sridharan and Meadows, 2016a; Sridharan *et al.*, 2017; Martínez *et al.*, 2017]. I present a novel RRL implementation that is orders of magnitude faster than its predecessors, potentially paving the way for more widespread adoption of RRL.

My computationally efficient RRL architecture supports not only relational representations but also adaptive (and non-adaptive) value function representations (Section 1.3.4) and the ability for agents to refine and unrefine (Section 1.3.4.3) the value function online and incrementally. These extensions further enhance the generality and speed of learning of RRL architectures. It additionally supports the automatic introduction of additional variables as part of a Higher Order Grammar (HOG) (Chapter 6), allowing agents to solve problems where the agent must be capable of attending to varying numbers of objects and relations in the world. I evaluate its efficacy and efficiency when functioning both non-adaptively and adaptively using several value function refinement and unrefinement criteria.

In the remainder of this introduction, I present some fundamentals of Reinforcement Learning (RL) in Section 1.1. I present my evaluation criteria in Section 1.2. I then discuss different value functions in Section 1.3, starting with my primary testbed, the Blocks World environment, in Section 1.3.1 and tabular RL in Section 1.3.2. I present more advanced forms of RL: linear function approximation in Section 1.3.3, tile codings in Section 1.3.4, and Relational Reinforcement Learning (RRL) in Section 1.3.5. I discuss additional related work in Section 1.4. And I conclude the introduction with discussion of my goals, what I will cover in the rest of the thesis, and a summary of my contributions in Section 1.5.

1.1 Reinforcement Learning Fundamentals

Here I present the fundamental RL concepts that are invariant regardless of the kind of knowledge (or value function) representation I use.

1.1.1 Markov Decision Process

A Markov Decision Process (MDP) [Bellman, 1957b] is a standard model of a sequential decision-making problem that involves a state space ($\mathcal{S}$) and an action space ($\mathcal{A}$). I discuss the number of different states and actions for the Blocks World environment in detail in Section 1.3.2.

An MDP is additionally defined by:

1. A state transition function ($P_a(s, s')$) which provides the probability of transitioning from any state $s \in \mathcal{S}$ to any successor state $s' \in \mathcal{S}$ for any possible action $a \in \mathcal{A}$ available in s

2. A reward function ($R_a(s, s')$) which provides the expected value of the reward for transitioning from any state s to any successor state s' for any possible action a available in s.

Finally, following from the definition above, an MDP satisfies the Markov property, that the current state (s) provides sufficient knowledge for an agent to predict the successor state (s') and immediate reward (r) without requiring it to have any memory of the sequence of states ($s_0 \dots s_{t-1}$) that led up to that state. [Sutton, 1988]

1.1.2 Temporal Difference Methods

Temporal Difference (TD) methods allow agents to learn value functions (and in turn greedy policies) model-free, without a model of either $P_a(s, s')$ or $R_a(s, s')$. They do this by learning a value function that I refer to as $Q(s, a)$ – an estimate of a the expected return, or cumulative (possibly discounted) reward for taking the best action from s, then the best action from the successor state s', and so on. Model-free RL agents are faced with a temporal credit assignment problem – attributing credit for reward to the actions responsible for that reward, even after arbitrarily many steps of delay between the action and the corresponding reward. Let us examine how they are able to learn without a transition model.

Our first TD methods are Sarsa (equation 1.1) and Q-learning (equation 1.2):

$$Q(s, a) \overset{\alpha}{\leftarrow} r + \gamma Q(s', a') \tag{1.1}$$

$$Q(s, a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') \tag{1.2}$$

Here r is the immediate reward, γ is a discount rate $[0, 1]$, and α is the learning rate $(0, 1]$.

Sarsa [Sutton, 1988] and Q-learning [Watkins, 1989] are two basic TD methods designed to solve RL problems by backing up reward to the actions responsible for them in an online, incremental fashion. The use of a learning rate, α, if sufficiently small, is enough to allow an agent to learn return estimates for taking actions from different states. The differences in expected return for different successor states are ultimately incorporated into the value function. This is not sufficient to model the environment, but it is sufficient to allow an agent to derive a greedy policy. $Q(s, a)$ could be represented with a simple number called a Q-value, or it could be a sum of a number of different weights that may contribute to $Q(s, a)$ for multiple different state-action pairs,

allowing an agent to share knowledge between similar states or actions using linear function approximation (Section 1.3.3). For information regarding extensions Sarsa(λ) and Q(λ) which can accelerate learning and Greedy-GQ(λ) which additionally guarantees convergence when using linear function approximation (Section 1.3.3), see Appendix A on page 136. I use these extensions throughout this work but they are not necessary for understanding the contributions.

1.1.3 Exploration

When learning from experience in the world, there is a question of how best to explore different actions in the environment given what the agent currently knows. An agent must sometimes try actions that seem to be worse than the best possible choice according to $Q(s, a)$ in order to discover alternatives that improve its value estimate (and in turn its policy) over time and to maximize expected return in the future.

Epsilon-greedy (ϵ-greedy) exploration is one of the simplest methods of exploration[1]. With $\epsilon = 0.1$, an agent will take a greedy action with probability $0.9 + 0.1\frac{|G|}{|A|}$ where G is the set of greedy actions and A is all candidate actions. With the remaining $0.1\frac{|A-G|}{|A|}$ probability, the agent will choose one of the non-greedy actions.

That ϵ-greedy exploration is equally willing to explore all non-greedy choices, even if some are more promising than others, might lead one to prefer an alternative exploration method such as Boltzmann action selection which prefers exploration of slightly suboptimal actions over significantly suboptimal ones, as used in related work [Džeroski *et al.*, 2001; Irodova and Sloan, 2005]. I generally use ϵ-greedy exploration in my work, but I use Boltzmann when comparing against related work as needed.

An agent learning on-policy will learn a value function that, when followed greedily, will cause the agent to incorporate penalties that result from exploration into its value function, and to thereby take actions that are optimal taking exploration into account. An agent learning off-policy will learn a value function that, when followed greedily, will cause the agent to take actions that are optimal for the target policy (typically strictly greedy) without regard to the additional costs of exploration. Therefore, an agent learning on-policy should be expected to perform better during exploration and an agent learning off-policy should be expected to have a more optimal policy when evaluated offline. When one is concerned with online performance, learning on-policy makes sense since the agent is going to incorporate penalties resulting from exploration into its policy. If one is concerned solely with the optimality of the terminal policy, learning off-policy can make more sense. In this thesis, most of my agents learn on-policy since I am concerned with online

[1] Sutton and Barto [1998] states that Watkins [1989] may have been the first to implement ϵ-greedy exploration, but they state that "the idea is so simple that some earlier use seems likely."

learning. However, for some comparisons to related work where policy optimality is evaluated offline, learning off-policy is more suitable.

1.2 Evaluation Criteria

I evaluate my work on Relational Reinforcement Learning (RRL) in two different dimensions.

In this work, I emphasize the problem of developing algorithms for efficiently evaluating and updating value functions at each step. For that reason, the paramount focus of this work is the problem of computational efficiency in terms of Wall-Clock Time Per Step (WCTPS) as described in Section 1.2.1.

Computational efficiency is interesting, of course, only if the methods are actually accomplishing something. As I explain in Section 1.2.2, I am concerned with learning efficiency in terms of either Average Regret Per Episode (ARgPE) or Average Return Per Episode (ARtPE).

1.2.1 Computational Efficiency: WCTPS

I am concerned with problems of computational efficiency, which I evaluate simply by looking at Wall-Clock Time Per Step (WCTPS). One step refers specifically to one cycle of the agent's execution – taking input from the environment, choosing an action, and executing that action in the environment. (Since my agents and environments proceed in lockstep, with the environment waiting for the agent between steps, one step of the agent's execution correlates to one step of execution of my environments, but this would not be the case if my agents were executing in a continuous-time environment instead.) One can restrict time measurements to code that implements agent logic only, factoring out time spent running environment simulations, but my wall-clock time measurements are nonetheless dependent on the real world constants that affect performance, such as Central Processing Unit (CPU) and Random Access Memory (RAM) speeds, code efficiency, and computational complexity. In some cases, I can give some analysis of computational complexity as well, but generally speaking I am concerned directly with WCTPS only.

I concern myself to some degree with memory efficiency as well. This concern is secondary however, as I am interested in memory only to the extent that the time taken to use it affects WCTPS.

1.2.2 Learning Efficiency: ARgPE and ARtPE

Computational efficiency or WCTPS is important, of course, only if the methods are actually accomplishing something from step to step. Both the amount of learning achieved and the cost of

exploration to achieve it can be assessed in a number of different ways. Here I discuss possible approaches, including those used in related work, and explain the metrics that I use in this thesis: Average Regret Per Episode (ARgPE) and Average Return Per Episode (ARtPE).

One might simply plot return (cumulative reward) per episode, evaluating whether it increases as an agent gains experience or perhaps how much faster it increases. If optimal return can be calculated for all initial conditions being tested, one could instead plot regret (the discrepancy between the optimal return and actual return) per episode, evaluating whether regret decreases or goes to zero as an agent gains experience or how quickly regret decreases. Further, one might evaluate the performance of a snapshot of an agent's policy on a number of different starting conditions and provide a percent of the number of those starting conditions for which the agent achieves precisely the optimal return. This percent-optimality is the approach taken in related work [Džeroski *et al.*, 2001; Irodova and Sloan, 2005].

Approaches that evaluate per-episode fail to take into account episode length, potentially masking the true cost of learning. Agents that explore chaotically with an off-policy TD method can be expected to learn more from the same number of episodes, or potentially from even fewer episodes since their exploration is more exhaustive within each episode. This may be desirable if the number of episodes that can be experienced is the primary bound on an agent over the course of learning, but I do not believe this to be a desirable way for an agent to learn under most circumstances. In fact, this metric rewards an agent for doing worse on early episodes and masks the total cost of learning when reported per episode.

Evaluating percent-optimality of a snapshot of an agent's policy additionally fails to capture the degree of suboptimality of all suboptimal episodes. In many cases, one may not care about strict optimality. An agent that performs optimally on 30% of episodes and near-optimally on the other 70% may be preferable to an agent that performs optimally on 90% of episodes and fails catastrophically on the other 10%.

In order to take into account episode length and degree of suboptimality, I argue that a better metric is to examine either return or regret at step granularity, since the number of steps provides a more direct correlation with the amount of work and the amount of experience that have gone into learning the task. With known optimal return, one can plot Average Regret Per Episode (ARgPE) at step granularity (instead of plotting or otherwise examining it at episode granularity). For example, once an agent has completed the second episode, the ARgPE at time t halfway through the second episode can be evaluated to be the average of the regret experienced over the course of the first two episodes. ARgPE provides a measure of suboptimality per episode which can be evaluated per step and plotted on the y-axis only as episodes are completed. Evaluating it per step and using total step counts on the x-axis gives a measure of total effort that exposes the impact of varied episode lengths on learning. Wall-clock time could be used for similar effect if a combined analysis of

learning efficiency and computational efficiency is the sole objective.

See Figure 1.1 for a visual demonstration of how ARgPE (in Figure 1.1c) can reveal a difference in learning that is masked when plotting regret or average regret against the number of episodes. White is doing better based on the number of steps even if it is doing worse in terms of the number of episodes.

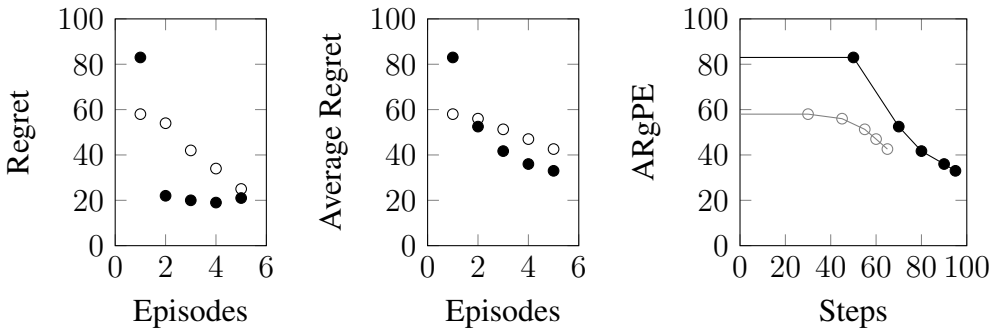
I present Algorithm 1 for ARgPE.[2] It directly sums regret (the difference between the expected return for the actions taken and the expected optimal) for all episodes and divides that sum by the number of episodes. ARgPE plotted per step (or per unit time) gives one a sense of amortized cost of learning. There are two important observations to be made here.

First, ARgPE is well defined only at steps that represent episode boundaries. Observe that Algorithm 1 does not depend on the number of steps. This necessitates that some method be devised to fill in the non-terminal steps for doing comparative analyses where episode lengths differ or when generating plots. As a simple approach, I choose to linearly interpolate between steps that represent episode boundaries. Algorithm 2 provides the regret value for the first episode until the second begins. From that point forward, all points at episode boundaries return values equivalent to those provided by Algorithm 1 and values between are linear interpolations instead.

Second, comparing two agents that have taken the same number of steps involves a comparison between agents that may have experienced different numbers of episodes. This is by design, since it is potentially unfair to directly compare an agent that takes $10,000$ steps to complete its first 10 episodes to an agent that completes its first 10 episodes in only 500 steps. The former has more experience to draw on after the same number of episodes, and it is not obvious from data plotted per episode that the latter has achieved its level of performance doing one twentieth the work. This interpolation allows one to additionally compute an average across multiple runs where episode boundaries might occur after different numbers of steps.

If optimal returns are unknown, instead of ARgPE, one can instead evaluate agents on the basis of return instead of regret. See Figure 1.3 for a visual demonstration of how ARtPE (in Figure 1.3c) can reveal the same difference in learning that is masked when plotting return or average return against the number of episodes. A plot of ARtPE is essentially equivalent to a plot of ARgPE, just with a sign inversion and a change in y-intercept.

[2]One might ask why I do not look at some form of normalized return. Normalization only makes sense if there exists a default, non-zero amount of suboptimality against which to compare an agent's performance. Since no such number exists, normalization is impossible in principle.



(a) Regret (b) Average Regret (c) ARgPE

Figure 1.1: Several regret-based plots

Algorithm 1 Exact ARgPE, valid at episode boundaries only.

Require: $|e|$ provides a count of the number of episodes
Require: r provides return experienced by the agent
Require: r^* provides return experienced by an agent following an optimal policy

function ARGPE(e, r, r^*)
    return $\frac{1}{|e|}\sum_{i=1}^{|e|}(r_i^* - r_i)$

Algorithm 2 ARgPE linearly interpolated for all steps.

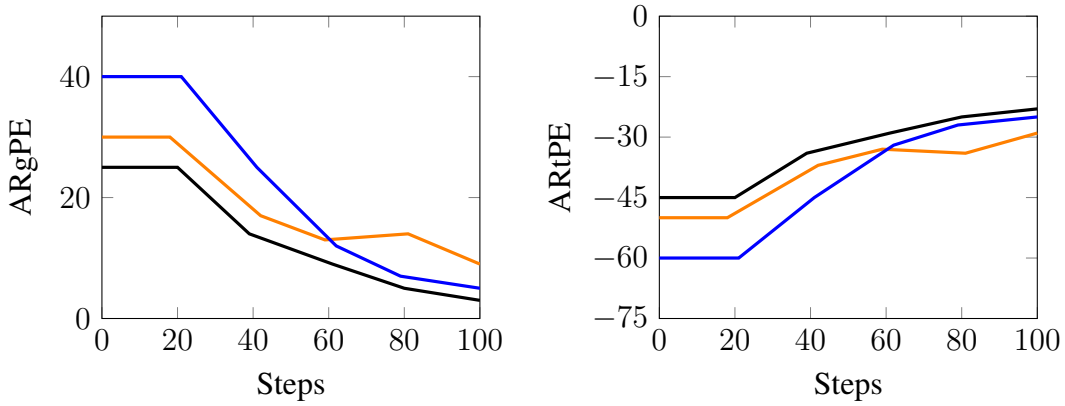
Require: e provides the lengths of all episodes
Require: r provides return experienced by the agent
Require: r^* provides return experienced by an agent following an optimal policy
Require: s provides a step between 0 and the maximum number of steps experienced

function ARGPE$\approx$(e, r, r^*, s)
    $i, c \leftarrow$ EPWITHSTEP(e, s)
    $r \leftarrow$ ARGPE($e|_1^i$, r, r^*)
    if $i = 1$ **then**
        return r
    else
        $l \leftarrow$ ARGPE($e|_1^{i-1}$, r, r^*)
        return $\frac{e_i - c}{e_i} l + \frac{c}{e_i} r$
function EPWITHSTEP(e, s)
    $i, c \leftarrow \langle 0, 0 \rangle$
    loop
        if $c >= s$ **then**
            return $\langle i, s + e_i - c \rangle$
        $i \leftarrow i + 1$
        $c \leftarrow c + e_i$

Average Return Per Episode (ARtPE) can be calculated per step as described in Algorithm 3 and Algorithm 4. Both algorithms are direct simplifications of their predecessors, structurally identical, but with regret calculations replaced with values of return. As the number of steps or episodes increases, the difference between ARgPE and ARtPE will converge to a constant factor, allowing maximal ARtPE to serve as a proxy for minimal ARgPE. ARtPE is the measure I use in most of my experiments, since I have not historically evaluated regret for my agents and return is an adequate proxy for my purposes. Note that if I attempted to evaluate average reward per step directly instead of plotting ARtPE per step, for any domain that provides -1 reward per step, the plot would be a flat line from start to finish regardless of what learning is achieved. ARtPE is meaningful for such domains and correlated to ARgPE even in the absence of knowledge of the degree of suboptimality of any given decision.

Consider the plots of equivalent ARgPE and ARtPE in Figure 1.2. One might ask what one can conclude about the performance of the black, blue, and orange agents on the basis of these plots. It is correct to say that orange performs better than blue until their crossover point and that blue performs better after that. If their performance at step 100 represents their asymptotic performance, one can conclude that blue is to be preferred with ample training time. With harsh limits on training time, one should instead prefer orange due to its better performance early on. However, this is of little importance in this scenario since the performance of black dominates that of both blue and orange, thereby making it the best of the three regardless of the amount of training time available. In summary, one can say in this instance that black has the best learning efficiency since it dominates blue and orange, but in lieu of a dominant agent, learning efficiency must be compared at specific points or intervals in time.

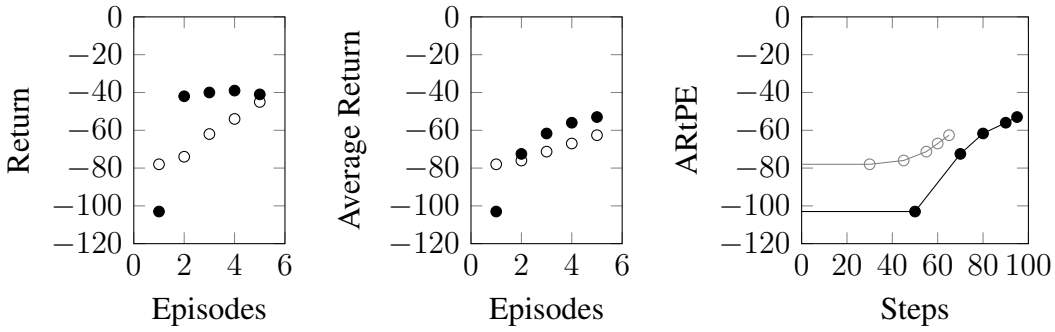
I will do a brief empirical evaluation of the differences that result from plotting ARtPE episodically or stepwise in Section 5.5.



(a) ARgPE

(b) ARtPE

Figure 1.2: ARgPE and ARtPE plots for analysis



(a) Return (b) Average Return (c) ARtPE

Figure 1.3: Several return-based plots

Algorithm 3 Exact ARtPE, valid at episode boundaries only.

Require: $|e|$ provides a count of the number of episodes
Require: r provides return experienced by the agent

function ARTPE(e, r)
    return $\frac{1}{|e|}\sum_{i=1}^{|e|} r_i$

Algorithm 4 ARtPE linearly interpolated for all steps.

Require: e provides the lengths of all episodes
Require: r provides return experienced by the agent
Require: s provides a step between 0 and the maximum number of steps experienced

function ARTPE$\approx$(e, r, s)
    $i, c \leftarrow$ EPWITHSTEP(e, s)
    $r \leftarrow$ ARTPE($e|_1^i$, r)
    if $i = 1$ **then**
        return r
    else
        $l \leftarrow$ ARGPE($e|_1^{i-1}$, r)
        return $\frac{e_i - c}{e_i} l + \frac{c}{e_i} r$
function EPWITHSTEP(e, s)
    $i, c \leftarrow \langle 0, 0 \rangle$
    loop
        if $c >= s$ **then**
            return $\langle i, s + e_i - c \rangle$
        $i \leftarrow i + 1$
        $c \leftarrow c + e_i$

1.3 Value Function Representations

Here I describe the different kinds of knowledge representation that I concern myself with in this thesis. I first present the Blocks World domain in Section 1.3.1 that I will later use to discuss how these representations relate to WCTPS and ARtPE. I then discuss tabular representations in Section 1.3.2. I present linear function approximation in Section 1.3.3 and tile codings in Section 1.3.4. And I introduce relational representations in Section 1.3.5.

1.3.1 Blocks World

Blocks World is a classic domain and one that I will use to illustrate tradeoffs between value function representations throughout this thesis. It presents n labeled but otherwise identical blocks in stacks where internal stack ordering can be important but placement of stacks relative to one another on the table is ignored.[3] The features an agent has available include [Džeroski *et al.*, 2001]:

1. `Higher-than(a, b)` – True if and only if a has a higher elevation than b, regardless of whether blocks a and b are in the same stack or different stacks
2. `Above(a, b)` – True if and only if a is higher than b and in the same stack
3. `On(a, b)` – True if and only if a is above b and $\neg\exists$ a block x such that (`Above(x, b)` $\wedge \neg$`Above(x, a)`)
4. `Clear(a)` – True if and only if $\neg\exists$ a block x such that `On(x, a)`

Each block can be in many `Higher-than(a, b)` and `Above(a, b)` relations, but in at most one `Clear(a)` and at most two `On(a, b)` relations – as the first argument in one and the second argument in the other. The table is special in that it can never be higher than or above any blocks, it can have as many blocks directly on top of it as there are blocks in the environment, and it is always clear.

An agent is tasked with rearranging blocks, by moving one clear block onto another clear block or onto the table at each step, in order to achieve some objective. Once the objective is achieved, the environment automatically terminates the episode. Many different classes of objectives can be specified. The different possible objectives I will discuss include:

[3] There are many other variants that could be explored. Block placement on the table could be considered to be relevant. Or block placement could be restricted in a number of ways. However, the version I describe here is what was explored in related work [Džeroski *et al.*, 2001; Irodova and Sloan, 2005].

1. The **stack** objective[4] requires that all the blocks be rearranged into a single stack. The order of blocks in that stack is irrelevant for this objective.
   More formally: $\exists!x$ such that `Clear(x)`

2. The **unstack** objective requires that all the blocks be rearranged to be on the table.
   More formally: $\forall x$ `Clear(x)`

3. The **on(a,b)** objective requires that a specified block, e.g. block A, be placed directly on top of another specified block, e.g. block B. The positions of other blocks are irrelevant for this objective.

4. The **exact** objective requires that the blocks be rearranged into a fully specified configuration of blocks. The goal configuration can be specified as a conjunction of `on(a,b)` relations that ultimately connect all blocks to the table.

Objectives **stack**, **unstack**, and **on(a,b)** (depicted alongside an **initial** configuration in Figure 1.4) were introduced and explored by Džeroski *et al.* [2001] and explored in later work as well [Irodova and Sloan, 2005], but objective **exact** (Figure 1.4e) has not been explored in the context of RRL prior to this point.

Taking Figure 1.4 as an example, let us explore example solutions to these objectives:

1. **Stack** can be reached optimally treating either stack AC or stack EB as the base since there exists no other block higher than either block B or block C. In either case, the remaining 3 blocks must be moved in any legal order (moving only clear blocks) onto whichever stack is selected. (3 moves)

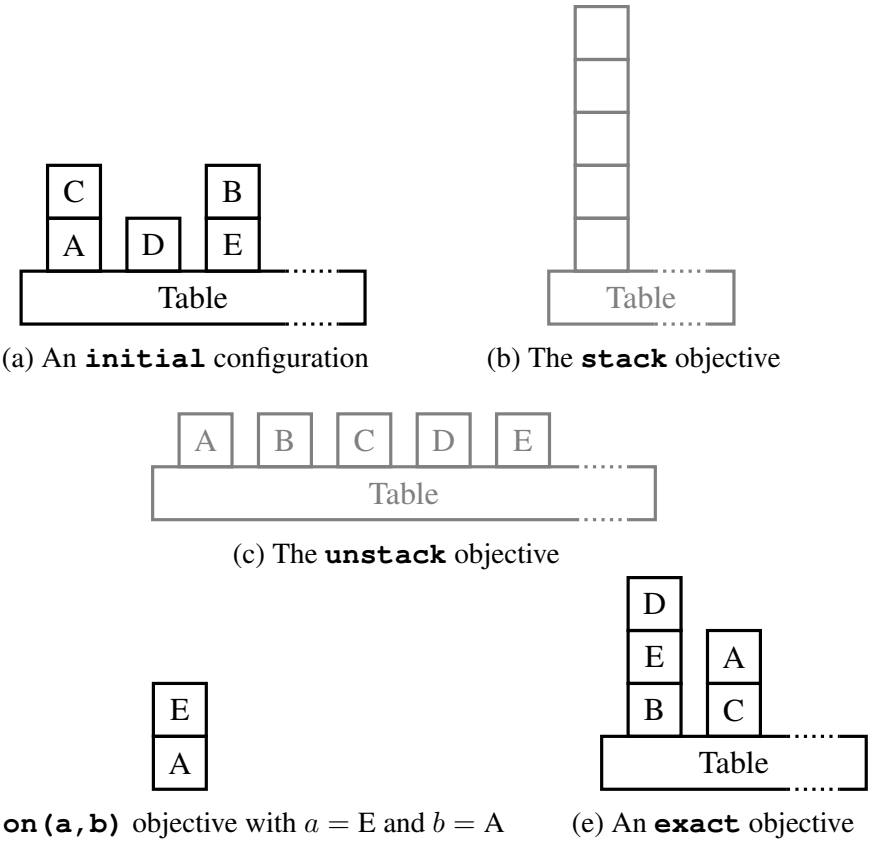
2. **Unstack** can be reached optimally by moving blocks B and C to the table in any legal order. (2 moves)

3. **On(a,b)** with $a =$ E and $b =$ A can be reached optimally by moving both blocks B and C either above block D, to the table, or onto one another on the table (anywhere that is not above either block A or block E) and then moving block E onto block A. (3 moves)

4. **Exact** can be reached by moving blocks B and C to the table in any order and then moving block A onto block C, block E onto block B, and block D onto block E. (5 moves)

Stack and **unstack** have fixed (implicit) objectives, but stack has $n!$ states that satisfy the objective while **unstack** has only one. **On(a,b)** and **exact** both represent the objective to

[4]This objective might be better named "single-stack" since it requires that all blocks be placed in a single stack, but "stack" is the name used in prior literature. [Džeroski *et al.*, 2001; Irodova and Sloan, 2005]



(a) An **`initial`** configuration (b) The **`stack`** objective

(c) The **`unstack`** objective

(d) An **`on(a,b)`** objective with $a = \text{E}$ and $b = \text{A}$ (e) An **`exact`** objective

Figure 1.4: Blocks World configurations illustrated for an instance with $n = 5$.

the agent explicitly and can change the goal **`on(a,b)`** relation or **`exact`** goal configuration from episode to episode.

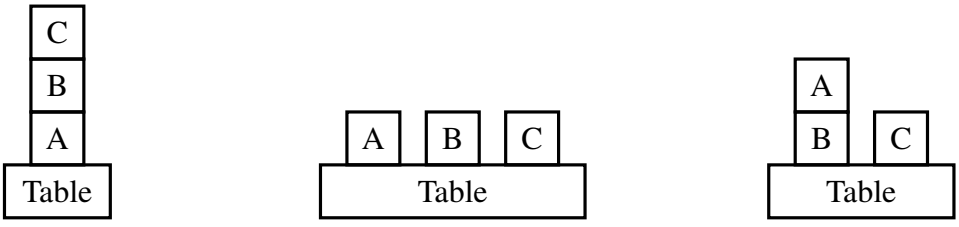
Using the features I described at the beginning of this section, **`unstack`** and **`stack`** are the simplest to express in terms of the number of features that an agent must incorporate into its value function. **`Unstack`** is slightly simpler than **`stack`** since whether a block is being moved to the table can be expressed with a single relation, while whether a block is being moved to one of the tallest stacks requires a conjunction of two relational features. **`On(a,b)`** is more complex to express, requiring a larger number of features and more complex conjunctions of those features in order to allow an agent to converge on an optimal policy, since an agent must be aware of whether blocks it is considering moving are above either of the blocks in the goal configuration in addition to being aware of whether one of the blocks being moved is one of the blocks in the goal configuration. **`Exact`** is more complex still, requiring either primitive features that encode more background knowledge or a drastically larger number of features for it to be solvable due to the number of `on(a,b)` relations that must be incorporated into the value function, both in the current state and in the goal configuration.

1.3.2 Tabular Reinforcement Learning

A tabular representation assigns a unique label, or index into a table, to each and every state of the world (or to each and every state-action pair). Regardless of how many details define the state of the world, the entirety of it is reduced to its label. Any change to the state results in a different label and there is no way for an agent to determine the similarity of states based on their labels. Therefore, there is no way for an agent naively using a tabular representation to identify or take advantage of similarities or differences between states which it has experienced.

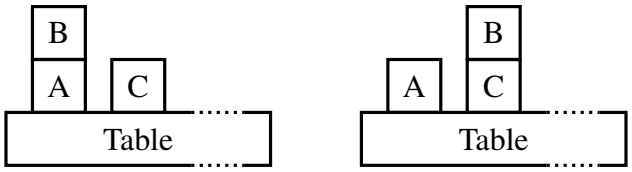
Let us first consider the knowledge representation problem for the current state of the world only in Blocks World without regard for the objectives (Figure 1.4a). Given that only the arrangement of blocks within stacks but not about where stacks of blocks are placed relative to one another, there are 3 possible states of the world for 2 blocks, 13 for 3 blocks, 73 for 4 blocks, and 501 for 5 blocks. (There is 1 possible state for 0 blocks and for 1 block, but no possible actions, so pursuing objectives in that case is not possible. It is for this reason that I start with 2 blocks.) Not all of these states are reachable for all objectives since for some states, passing through a state that satisfies the objective and terminates the episode would be required (e.g. as depicted in Figure 1.5, `On(A,B)`, `on(B,TABLE)`, and `on(C,TABLE)` cannot be reached from `on(C,B)`, `on(B,A)`, and `on(A,TABLE)` if the objective state is `on(A,TABLE)`, `on(B,TABLE)`, and `on(C,TABLE)` since there is no way to transition between those two states without passing through the objective state), but these numbers still give some sense of the structural complexity of the world.

I now analyze how the number of states grows as the number of blocks increases in order to illustrate the intractability of this approach as the number of blocks increases. The problem of calculating the possible number of states in Blocks World for a given number of blocks, n, can be reformulated as the problem of calculating the "Number of 'sets of lists': number of partitions of 1, . . . , n into any number of lists, where a list means an ordered subset." This can be calculated in a number of ways, but Wieder [2005] presents an equation that gives one the number of "sets of



(a) **`Initial`** configuration (b) **`Unstack`** objective configuration (c) Unreachable configuration

Figure 1.5: A state or configuration is unreachable if it requires passing through a state that satisfies the objective.



(a) State 8 (b) State 11

Figure 1.6: Two labeled states in a tabular representation.

lists" as a function of the number of different partitionings, $p(n)$, the number of differently-sized partitions, $d(i)$, and the counts for each size of partition, $m(i, j)$:

$$\sum_{i=1}^{p(n)} \frac{n!}{\prod_{j=1}^{d(i)} m(i,j)!} \tag{1.3}$$

Simplifying it slightly, one can consider the numbers of partitions of all sizes 1 to n for partitioning i, $k(i, 1) \ldots k(i, n)$, instead of considering $d(i)$ or $m(i, j)$

$$\sum_{i=1}^{p(n)} \frac{n!}{\prod_{j=1}^{n} k(i,j)!} \tag{1.4}$$

Dividing by the product of the factorials of the numbers of lists of each size is necessary because one does not care about the ordering between these different lists. In the case of Blocks World, each stack of blocks is a partition and one is unconcerned with where stacks of blocks are relative to one another.

One way of representing knowledge of the state of the world is to index these possible states. The agent might then know that it is in state 8 or in state 11. (See Figure 1.6.) Such an agent would not know how many blocks there are or that there even are blocks. This kind of representation is opaque, but can be learned over regardless.

Note that use of a tabular representation does not entail that a problem meets the criteria for an MDP. However, so long as the numbers of states and actions are finite and enumerable, any MDP can theoretically be solved using a tabular representation.

The number of possible actions, or more precisely state-action pairs is worth considering as well. There are between 1 and $n(n-1)$ move actions available to an agent at any given time. This can be calculated precisely with:

$$(\sum_{j=1}^{n} k(i,j))^2 - k(i,1) \tag{1.5}$$

The subtrahend k_1 in equation 1.5 is necessary to eliminate moves from the table to the table—

n	$p(n)$	States	State-Action Pairs
2	2	3	4
3	3	13	30
4	5	73	240
5	7	501	2,140
6	11	4,051	21,300
7	15	37,633	235,074
8	22	394,353	2,853,760
9	30	4,596,553	37,819,800
10	42	58,941,091	543,445,380
20	5,604	$3 * 10^{20}$	$6 * 10^{21}$
30	37,338	$2 * 10^{35}$	$6 * 10^{36}$
40	204,226	$3 * 10^{51}$	$1 * 10^{53}$

Table 1.1: A list of the number of different partitionings ($p(n)$), states, and state-action pairs for Blocks World instances with between 2 and 40 blocks. Here I ignore how different objectives can increase the complexity of the state space and/or decrease the accessibility of certain states.

moves that are disallowed for my agents and no-ops in terms of the state description. Assuming actions are represented in the form of block-destination pairs, there exist 4 state-action pairs for 2 blocks, 30 for 3 blocks, 240 for 4 blocks, and 2,140 for 5 blocks. This can be calculated precisely with an addition to equation 1.4, resulting in the novel equation [Bloch, 2018]:

$$\sum_{i=1}^{p(n)} \left\{ \frac{n!}{\prod_{j=1}^{n} k(i,j)!} \left(\left(\sum_{j=1}^{n} k(i,j) \right)^2 - k(i,1) \right) \right\} \tag{1.6}$$

I present values for the numbers of action (equation 1.5), the numbers of states (equation 1.4), and the number of state-action pairs (equation 1.6) in Table 1.1.

1.3.2.1 Efficiency

The idea of representing all the states and state-action pairs as numeric indices is simple. The WCTPS of mapping states to indices is low, so the computational efficiency of a value function using this approach is high.

However, it becomes clear from Table 1.1 that the number of state-action pairs in Blocks World (Section 1.3.1) grows quite large as n grows. The number of state-action pairs grows from 30 for 3 blocks to 240 for 4 blocks. There are comparable order of magnitude increase for each additional block added to the environment.

It is clear that the naive approach of using a tabular value function with a value for each state-action pair will make the problem of learning a value function sufficient for deriving a near optimal

policy intractable. Given the lack of generalization when learning over such tabular representations, and given the growth of state-action pairs as n increases in Table 1.1, some alternative approach with the ability to generalize from experience or to transfer learning between different states would be preferable even with a more costly approach to implementing the value function.

1.3.3 Linear Function Approximation

Having a simple one-to-one correspondence between indices and states as in a tabular representation (Section 1.3.2) does not allow for any generalization. One can instead represent states not as single indices into a table that map to Q-values, but as sets of features that describe the state-action space.

A feature is a statement about a state-action pair. The i^{th} feature can be described by a basis function, $\phi_i(s, a)$ that provides a degree of activity (or truth) between 0 and 1, where 1 is fully active (or true) and 0 is not active at all (or false). In this thesis, all features are Boolean and all basis functions are Boolean as well. (There are alternatives to Boolean basis functions such as Radial Basis Functions (RBFs) [Broomhead and Lowe, 1988], but they are outside the scope of this dissertation.)

A feature dimension is a set of features of which exactly one must be active for any state-action pair. Following from this definition, a feature dimension provides a partitioning of the state-action space. A feature dimension need not directly correspond to an actual dimension of the state space or state action space. An eight-dimensional state-action space could be described using any number of feature dimensions – possibly fewer than eight or possibly more. In the context of this thesis, features are strictly organized into feature dimensions.

A feature dimension describing the state of the environment for Blocks World might include the feature `block-a-is-clear` and the feature `block-a-is-not-clear`, or `block-a-is-on-block-b` and `block-a-is-not-on-block-b`, or `height-of-stack-s-is-greater-than-or-equal-to-3` and `height-of-stack-s-is-less-than-3`. See Figure 1.7 for more examples. A feature dimension could also have a higher arity and include features corresponding to several destinations an agent must reach in a navigation problem (e.g. `dest-is-red`, `dest-blue`, `dest-is-yellow`, ...). One could also include a discretization of a continuous feature such as a measure of temperature or an amount of fuel (e.g. `fuel-is-1`, `fuel-is-2`, `fuel-is-3`, ...).

`b-on-a`	`a-not-clear`	`c-height-is-3`
(a) Feature 18	(b) Feature 22	(c) Feature 42

Figure 1.7: Three features in a linear function approximation representation.

Given n total features, $\phi_1 \ldots \phi_n$, and a weight w_i associated with each feature, the Q-function can be calculated by

$$Q(s,a) = \sum_{i=1}^{n} \phi_i(s,a) w_i \tag{1.7}$$

where $Q(s,a)$ is semantically equivalent to the Q-values used with tabular representations. When doing TD updates using equation 1.1 or equation 1.2, all TD updates (δ) to individual weights must be divided by the number of active features or contributing weights:

$$w_i \leftarrow w_i + \alpha \frac{1}{\sum_{i=1}^{n} \phi_i(s,a)} \delta \tag{1.8}$$

For a more complex TD method than Sarsa (equation 1.1, Q-learning (equation 1.2), Sarsa(λ) (Appendix A.1), and Q(λ) (Appendix A.1) that provides asymptotic convergence guarantees when learning off-policy, see Greedy-GQ(λ) in Appendix A.2. The majority of my agents in Chapters 5 and 6 use Greedy-GQ(λ).

Linear function approximation provides abstraction and the ability to generalize. The number of values (weights or Q-values) in an agent's value function is no longer directly tied to the number of different states in the world since features, such as destination-block-is-clear, no longer depend on the size of the state space. An agent using this kind of knowledge representation has an effective state-action space size that can be calculated as a product of the number of features in each feature dimension, for example `b-on-a` or `b-not-on-a`, or `height-of-a-is-1`, `height-of-a-is-2`, or
`height-of-a-is-3`:

$$\prod_{i=1}^{D} D_i \tag{1.9}$$

If these features perfectly describe all state-action pairs, the effective number of state-action pairs is the same as when using a tabular representation. However, now the number of features to learn over scales with a sum instead of a product:

$$\sum_{i=1}^{D} D_i \tag{1.10}$$

This reduces the effective size of the state-action space even if the features perfectly describe the state-action space since an agent can generalize from its experience with a given feature even as features in other feature dimensions change their degrees of activity. With carefully chosen features, the number of features to be learned over (equation 1.10) can be significantly less than the total number of states (equation 1.9). For example, in a grid world, if it is possible to decompose a

10x10 grid into distinct x and y feature dimensions of size 10 instead of addressing all 100 positions individually, the number of features (and weights) is reduced from 100 to 20 – from multiplicative 10x10 to additive 10+10. As the number of feature dimensions increases, these savings can result in orders of magnitude fewer features and weights and more generalization from past experience as a result.

1.3.3.1 Deictic Representations

Linear combinations of features can be enhanced through the use of deictic representations. Deictic representations are similar to propositional ones, but rather than having a feature such as `block-a-on-block-b` and other such features for every pair of blocks, an agent might instead have a feature `block-being-moved-on-block-b` [Agre and Chapman, 1987].

In a deictic representation, there is a pointer (e.g. "`block-being-moved`") present in the propositional feature, requiring that some context-sensitive processing occur in order to evaluate the feature from step to step. These pointers are typically (but not necessarily) relative to the agent. Given that the agent need concern itself with only one labeled block in this case, this can provide a significant increase in the generality of the feature, potentially increasing ARtPE. However, these kinds of pointers are more limited than full fledged variables in that it is impossible to compare the `block-being-moved` to any references in other features, or to see if the `block-being-moved` is the same as one referred to in any other features. (Of course, since only one block can be moved at a time, they must be the same in this particular example, but that is not generally true.) In Section 1.4, I compare my work to related work that uses deictic representations. The ability to capture relationships between objects using variables, and to represent more of the structure of the task as a result, forms the basis of relational representations in Section 1.3.5.

1.3.3.2 Efficiency and Discussion

The features chosen reflect the lens through which whomever creates them views the problem. If features are chosen poorly, problematic state aliasing can result in which states that the agent can distinguish between in its value function cannot be separated.[5] This can mean that the best policy that an agent can achieve in the limit may be suboptimal. Further, Exclusive OR (XOR) relationships, in which certain features are situationally good or bad, depending on the values of other features, can also limit the optimality of the policy resulting from the value function that can be learned by an agent.

[5]State aliasing can result not only from knowledge representation issues but from real world sensory limitations as well.

WCTPS is higher for linear function approximation than when using a simple tabular representation, as D weights have to be processed for each state-action pair. However, this additional computation allows the agent to decouple its learning problem from the dimensionality of the state-action space, to share knowledge between different states, and to potentially apply that knowledge to states not yet visited. These improvements can lead to a significant improvement in ARtPE, although perhaps only for finite time horizons if optimality is bounded by the representation in the limit.

1.3.4 Tile Codings

RL algorithms (such as both Sarsa and Q-learning) suffer from the curse of dimensionality, that the amount of data or experience required for learning increases exponentially as the size of the state space increases [Bellman, 1957b], when learning over tabular representations. Sarsa(λ) and Q(λ) (see Appendix A.1), help speed learning, but do not reduce the dimensionality of the state space of a problem.

Linear function approximation (Section 1.3.3) is one approach to solving the curse of dimensionality. A related approach to improving generalization – to decouple learning and computational complexity from the raw size of the state space – is to use a tiling [Albus *et al.*, 1971; Albus, 1981]. As if placing a 2-dimensional continuously-valued world on grid paper and treating each square as a state, a tiling can reduce an infinitely large continuous state-space to a finite number of states. Each tile has a weight associated with it (making these weights Q-values in the case that there exists only the one tiling). An 8x8 tiling (see Figure 1.8) results in 64 distinguishable states since all states within a given tile share the same weight. A tiling can be applied to domains with more than two dimensions or with discrete features as well.

If discretization of continuous features is involved, this kind of generalization results in an

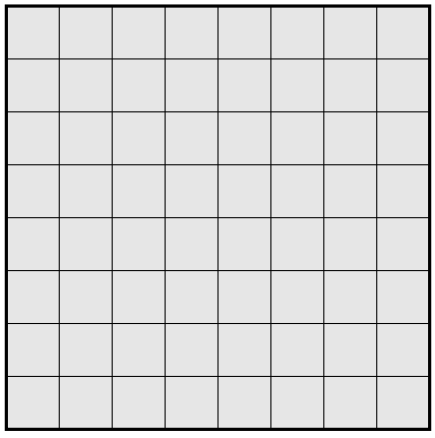


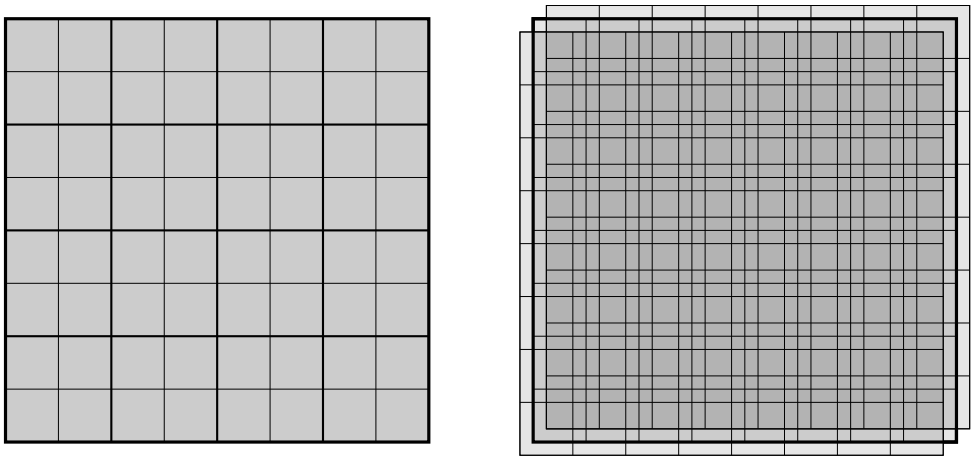
Figure 1.8: An 8x8, 2-dimensional tiling

agent's treating all states within a given tile identically. If value estimates need to change more gradually throughout the state space, one option is to use a more refined tiling at the expense of losing some generalization.[6] More refined tilings potentially result in lower ARtPE early on but can continue to have non-negligible growth in ARtPE for much longer (i.e. a long tail). Therefore, a coarse (e.g. 4x4) tiling might have high ARtPE in the beginning of an agent's lifetime but fall short of optimal in the limit, while a very refined (e.g. 64x64) tiling might have low ARtPE early on but achieve a nearer optimal ARtPE in the limit.

1.3.4.1 Multiple Tilings or CMACs

Using a single tiling is simple but does not provide an agent with any knowledge sharing (or generalization) in the value function for adjacent tiles. Using multiple overlapping tilings is an approach that was originally proposed was based on a theory about how a Cerebellar Model Arithmetic Computer (CMAC) might biologically process input [Albus, 1981; Albus *et al.*, 1971]. It might involve individual tilings of different sizes or with different offsets overlapping one another to form a tile coding (or CMAC) as depicted in Figure 1.9.

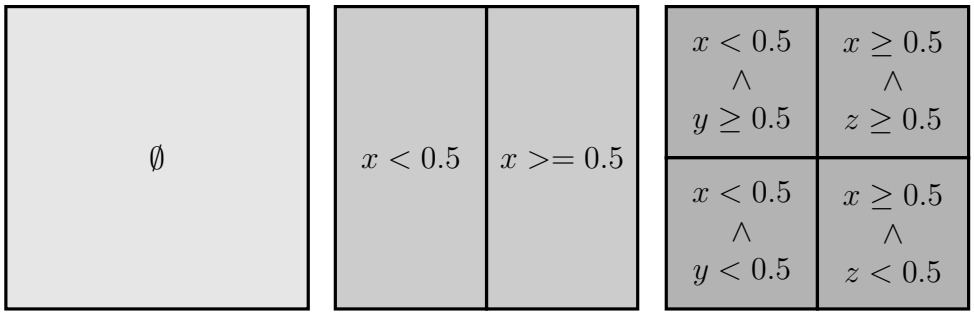
Tile codings with multiple tilings have the potential to provide a combination of high ARtPE in the beginning thanks to the coarsest tiling(s) and the ability to achieve a nearer optimal ARtPE in the limit than could be achieved with the coarsest tiling(s) alone. The weights of a tile coding consisting of multiple tilings can be learned over using linear function approximation as described in Section 1.3.3.



(a) An 8x8 tiling and a 4x4 tiling (b) 3 8x8 tile codings with different offsets

Figure 1.9: Two tile codings or CMACs consisting of different overlapping tilings, from which the Q-function can be computed as a sum of the weights for tiles that cover a given state

[6] Another possibility would be to use something more akin to a nearest neighbor or prototype approach, as is attempted by Kanerva encoding ([Kanerva, 1988]) but is beyond the scope of this research.



(a) 1x1 (b) 2x1 (c) 2x2

Figure 1.10: A naHTC consisting of 3 tilings.

1.3.4.2 Non-Adaptive Hierarchical Tile Coding

A non-adaptive Hierarchical Tile Coding (naHTC) imposes structural limitations on CMAC. While similar to what is depicted in Figure 1.9a, instead of having arbitrary flexibility to overlay tilings on top of one another:

1. There must exist a maximally general 1x1x... tiling that covers the entire state space with a single tile.

2. Each tiling after the first must represent a partitioning of one or more existing tiles, using exactly one unused feature dimension for each tile's partitioning.

When adding a tiling that represents a partitioning of more than one existing tile, it is not necessary that each tile use the same feature dimension for its partitioning. For example, if the most general tiling tiles over feature dimension x, dividing it into $x < 0.5$ and $x \geq 0.5$, the next most general tiling could consist of a tiling over y for $x < 0.5$ and over z for $x \geq 0.5$, resulting in the tiles $x < 0.5 \wedge y < 0.5$, $x < 0.5 \wedge y \geq 0.5$, $x \geq 0.5 \wedge z < 0.5$, $x \geq 0.5 \wedge z \geq 0.5$. (See Figure 1.10.) Ultimately, if there are a finite number of features, each region of the state space could have tiles that correspond to each active feature in a full naHTC but have those features at different levels of generality in different regions of the state-space. e.g. $x < 0.5 \wedge y < 0.5 \wedge z < 0.5$ and $x \geq 0.5 \wedge z < 0.5 \wedge y < 0.5$

This approach to building a CMAC results in the values for different tilings being learned at different levels of generality. The weights for larger, coarser tiles can converge more quickly, then allowing the weights of more refined tiles to converge on more specific knowledge about the value function.

1.3.4.3 Adaptive Tile Coding

Another approach to tile coding is to start with a coarse tile coding, and to dynamically break the tiles into smaller, more refined tiles over time using an Adaptive Tile Coding (ATC) [Whiteson *et al.*, 2007; Sherstov and Stone, 2005; Reynolds, 1999; Munos and Moore, 1999b; Moore and Atkeson, 1995]. This is a secondary learning mechanism that functions in tandem with TD methods. While TD learning is taking place for the set of weights for existing tiles, this secondary mechanism is modifying the set of tiles and corresponding weights for TD learning to work on.

An ATC is related to a naHTC (depicted in Figure 1.10), but instead of having multiple overlapping tilings of different levels of generality defined from the start, tiles that appear to require a more refined value function are replaced by more refined tiles as learning progresses. See Figure 1.11 for an abstract example and Figure 1.12 for an example of how an ATC can be used for a relational Blocks World domain, as we will explore in Chapter 5.

As the agent refines the tile coding in different regions of the state space, it becomes capable of learning a correspondingly more refined value function and a correspondingly better policy, but without the full cost associated with having started with a significantly refined tiling from the beginning. Overall, this can result in good early ARtPE but without the consistent overhead of using linear function approximation.

Munos and Moore [Munos and Moore, 1999a; Munos and Moore, 1999b] discuss a number of different criteria for choosing how to refine their tile codings. Ones that informed my work include:

1. **Influence**, formally defined as $I(\xi_i|\xi) = \sum_{k=0}^{\infty} p_k(\xi_i|\xi)$ where $p_k(\xi_i|\xi)$ is defined as "the discounted cumulative k-chained probabilities" which "represent the sum of the discounted probabilities of all sequences of k states from ξ to ξ_i" [Munos and Moore, 1999a], is a measure of the degree to which a Q-value associated with one tile influences other Q-values (associated with individual tiles) within the value function. It can be calculated by doing TD backups without the reward component. An agent should choose to refine tiles with high **influence** in order to maximize impact on other Q-values for each refinement.

2. **Variance** is a measure of the degree to which the discounted return experienced by an agent differs from run to run. The average difference between the expected value, i.e. the Q-function ($\mathcal{Q}(s, a)$), and the actual discounted return experienced after s (or state-action pair s-a) forms the basis for the **variance** measure. An agent should choose to refine tiles with high **variance** in order to best approximate the true value function.

3. **Stdev_Inf** is the product of both influence and variance, $I(s)\sigma(s)$ or $I(s, a)\sigma(s, a)$.

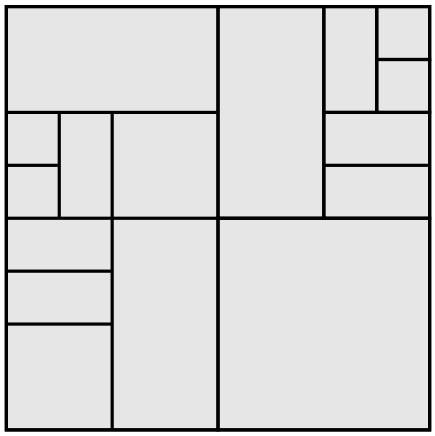


Figure 1.11: A two-dimensional Adaptive Tile Coding (ATC)

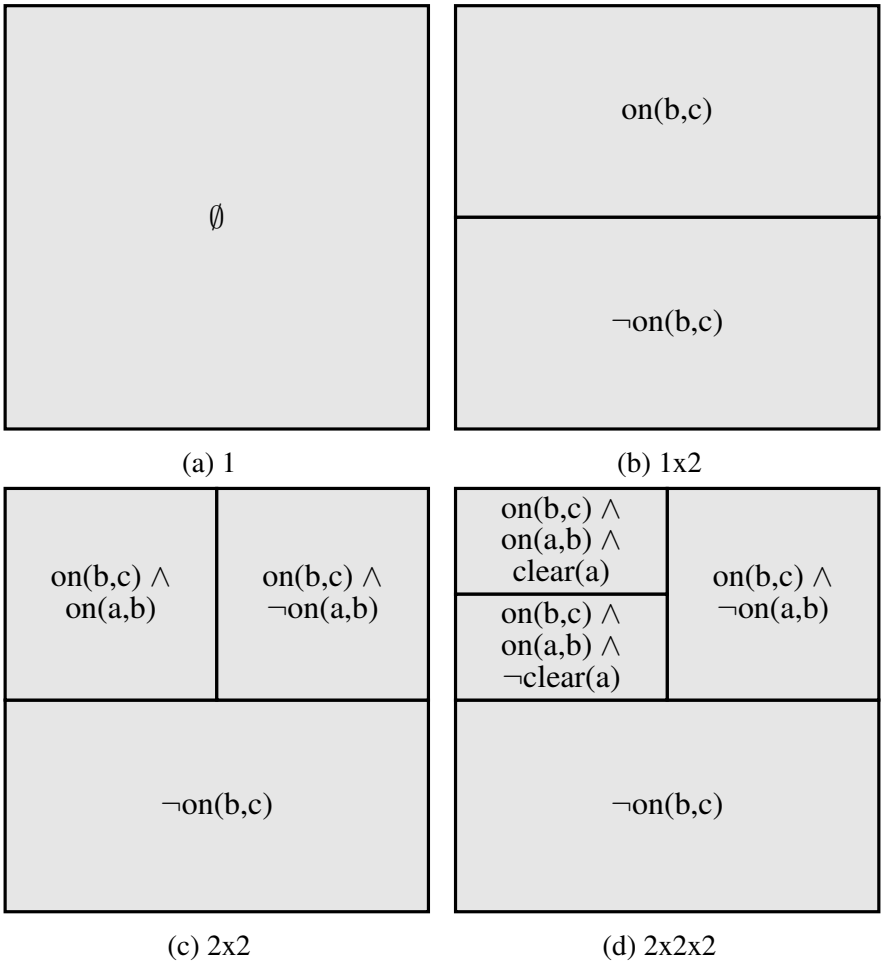


Figure 1.12: An ATC for Blocks World refined stage by stage, incorporating RRL (Section 1.3.5) using what I engineer in Chapter 4.

Their metrics are computed incrementally,[7] but they provide no incremental method for determining which tiles should be refined. Instead they refine tiles periodically in batches, splitting the top $f\%$ of the highest scoring tiles. Note that these metrics help an agent to decide *where* to refine the value function but offer no assistance in determining *when* to do so. That is, no values of these metrics directly entail that refinement ought to occur – they are purely relative.

Munos and Moore [1999b] use a *k*-dimensional trie (*k*-d trie) to store their tile codings, which forms the basis for the first implementation of the value function in my architecture (to be introduced in Section 3.1.1). A *k*-d trie is a variation of a *k*-dimensional tree (*k*-d tree) (which is itself a generalization of a binary tree) in which the index into the *k*-d trie is comprised of the entire sequence of nodes leading to the leaf. Tries are commonly used for dictionaries, but are also suitable for conjunctions of different features, or for more and more specific features.

Whiteson *et al.* [2007] explore other criteria that I built on:

4. The time since the most recent minimal **Bellman error**[8] is used as their criterion for determining *when* to refine the value function. The underlying idea is that improvement may have stalled when the minimal **Bellman error** stops declining.

5. Their **value criterion** for determining *where* to refine the value function, as a batch operation, selects the tiles to split that maximize the resulting value difference between the resulting subtiles. The objective of the **value criterion** is to do refinements to allow closer approximation of the value function.

6. Their **policy criterion** for determining *where* to refine the value function, as a batch operation, selects the tiles to split that maximizes the likelihood that the resulting subtiles will result in different policy decisions, using a running counter of expected changes to the policy that is updated for each active tile at each time step. The rationale underlying the **policy criterion** is that the policy is what one really cares about, so closely approximating the value function may needlessly refine parts of the value function where the gradient is high but the policy is consistent nonetheless.

If the agents are to make interesting decisions about *how* to refine the value function once the questions of *when* and *where* are answered (or as part of the process of making those decisions), some information about the candidate subtiles must either be built up over time or calculated at the time of refinement. The agents developed by McCallum [1996] store sets of examplars in the fringe

[7] Note, however, that in my work I settled on a different method for calculating variance as described in Appendix B.

[8] More correctly, this should have been called **TD error** since it represents the immediate update experienced by $\mathcal{Q}(s, a)$ and is not a differential resulting from an incomplete solution to the Bellman equations (i.e. incompletely run value iteration).

to allow them to make calculations about which subtiles to create. On the other hand, the agents developed by Whiteson *et al.* [2007] store more compact statistics about possible refinements to the value function and update them as they explore.

1.3.4.4 Adaptive Hierarchical Tile Coding

Combining naHTCs and ATCs together, I develop ATCs that, instead of removing coarser tiles as more refined ones are added, preserve them and continue to use them for learning. An adaptive Hierarchical Tile Coding (aHTC) must still follow my rules for the structure of a naHTC (listed in Section 1.3.4.2), but the structure of the value function is built up dynamically using approaches that are compatible with ATCs. See figure 1.13 for an example of how an aHTC can be used for a relational Blocks World domain, as I will explore in Chapter 5. The darker regions in Figure 1.13 (but not present in Figure 1.12 for a non-hierarchical ATC) indicate regions where features of

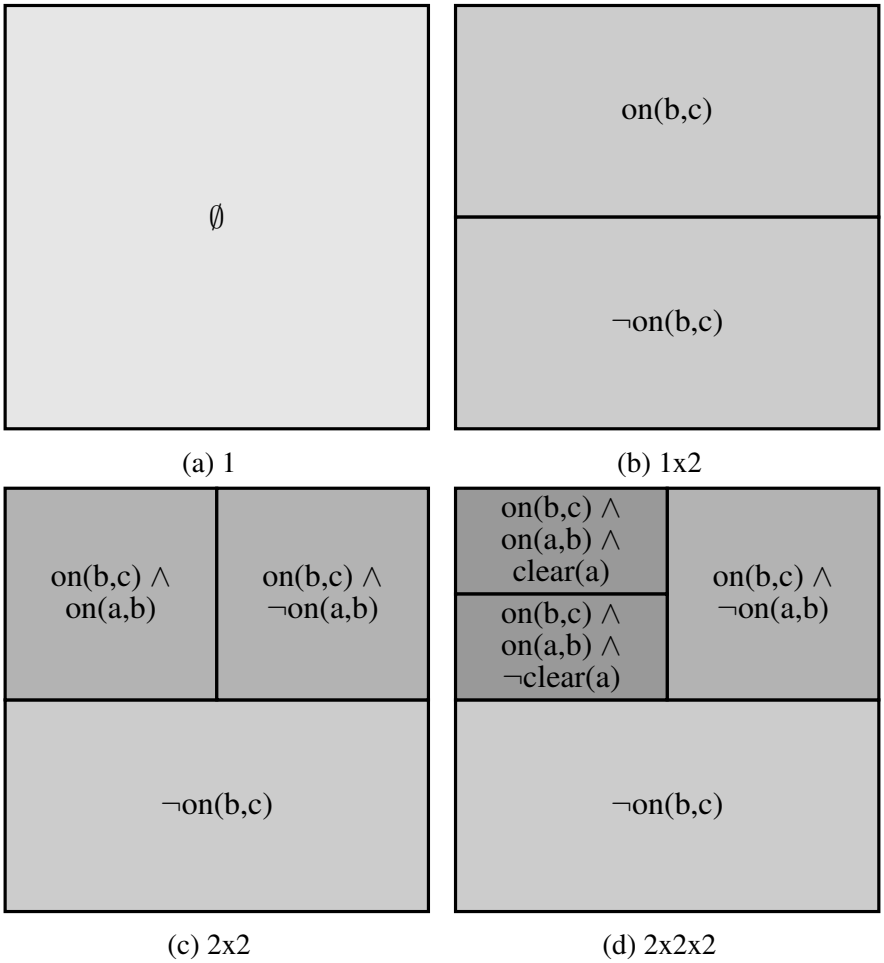


Figure 1.13: An adaptive Hierarchical Tile Coding (aHTC) for Blocks World refined stage by stage, incorporating RRL (Section 1.3.5) using what I engineer in Chapter 4.

different resolutions overlap, as defined by a Hierarchical Tile Coding (HTC). I describe this contribution in greater detail in Chapter 3.

aHTCs potentially allow an agent to combine the advantages of naHTCs with the flexibility of being able to learn about the world as refinement decisions are being made. For more complex domains, this may allow agents using HTCs to make better decisions about how to structure their value functions to maximize ARtPE. I test this hypothesis in Section 2.2.2.2 and throughout Chapter 5.

1.3.4.5 Efficiency and Discussion

Single tilings are the most computationally efficient form of a tile coding for computing the value function, mapping different portions of the state space (or state action space) onto a fixed number of tiles (or weights). However, for some domains, there may be no single tiling that allows for an optimal policy. Even when an optimal policy is possible, it is possible that such a tiling is so refined that the learning problem is once again intractable and ARtPE early in an agent's lifetime is low, just like with a tabular representation. Additionally, single tilings are the most dependent on the agent designer to choose a good tiling.

Multiple overlapping tilings can provide a faster increase in ARtPE early on while still allowing an agent to achieve a nearer optimal policy in the limit, but they also increase computational costs by a factor linear in the number of tilings. For low dimensional problems, multiple overlapping tilings can be sufficient to get effective generalization [Sutton, 1996].

Adaptive tile codings might superficially appear to require strictly less WCTPS than multiple overlapping tilings. However, the computational costs of deciding *when*, *where*, and *how* to refine the value function could exceed the computational cost savings of having only a single tiling rather than multiple fixed tilings. On the other hand, multiple overlapping tilings might be insufficient for learning high dimensional problems. I will consider these possibilities and evaluate the efficacy and efficiency of naHTCs, ATCs, and aHTCs throughout this work.

1.3.5 Relational Reinforcement Learning

Relational representations are another approach to value function representation. Rather than representing states as either opaque indices or as independent feature dimensions, in RRL the state is decomposed into objects and relations (relational features). Relational features contain variables which can match arbitrary objects in the environment. An RRL agent can compare variables across different features and build up conjunctions of features where the agent requires that variables match. For example, a conjunction of `clear(`a`)` and `on(`a`, `b`)`[9] will necessitate that a

[9] In this context, the variable a does not refer to the current action as defined in the list of symbols.

refers to the same block in both relational features, but not a specific block that must be the same each time the conjunction is evaluated.

Conceived as a merging of RL and relational learning or inductive logic programming, Džeroski *et al.* [2001] imagined that the more expressive representation of RRL could allow RL to be applied to new learning tasks, beginning with Blocks World (which I introduced in Section 1.3.1). The solution they explored, Q-RRL, is to use TILDE-RT [Blockeel and De Raedt, 1998] to generate a decision tree over the relational features representing state action pairs, storing Q-values at the leaves of the tree. They demonstrated that RRL is a powerful method for learning to solve certain Blocks World tasks. However, as they noted, TILDE-RT is not an incremental algorithm, thus Q-RRL requires extra processing to update Q-values between expensive executions of TILDE-RT. Their representation included predicates `clear(`a`)`, `on(`a`, `b`)`, and `above(`a`, `b`)` (in the same stack), as well as predicates allowing an agent to combine relations to calculate `higher-than(`a`, `b`)` and `lower-than(`a`, `b`)` (in any stack).

Džeroski *et al.* [2001] describes a number of limitations of an earlier approach to RRL by Langley [1995] in which Langley propositionalized the representation by treating each set of values within the relations as a proposition. Firstly, Langley's solution breaks down for large or infinite state spaces – a problem Langley resolves using an inductive learning algorithm (a neural network). Secondly, it fails to capture the structural aspects of the task. This limitation provides contrast with an RRL agent's value function that develops conjunctions of relations which correspond to the structure of the task. For example, an RRL agent can learn the value of a conjunction of both whether a block being considered for a move is one of the blocks in the objective `on(`a`, `b`)` relation and whether the destination block under consideration is above a block in the objective `on(`a`, `b`)` relation. See Figure 1.14 for a simple presentation of other example RRL features. Thirdly, any change of goal requires complete retraining even if the goal is explicitly represented as part of the state space since non-relational features can be conditioned only on fixed, individual goals once propositionalized. Finally, some transfer of previously learned knowledge might be expected going from 3 blocks to 4, and this solution does not facilitate that kind of transfer, once again due to the limitations imposed by propositionalization. Given that Langley presented a solution to only the first of these four problems, Džeroski *et al.* emphasized the need for a learning algorithm a relational learning algorithm, RRL, to take full advantage of relational representations.

Relational representations allow an agent to achieve a kind of generality that is not possible with the tabular representation described in Section 1.3.2 or with a linear combination of features

on(b,a)	clear(c)	clear(c) $\wedge$ $\neg$clear(a)
(a) Feature 2	(b) Feature 5	(c) Feature 8

Figure 1.14: Three features in a RRL representation.

n	$p(n)$	States w/out Labels	State-Action Pairs w/out Labels
2	2	2	3
3	3	3	10
4	5	5	27
5	7	7	56
6	11	11	114
7	15	15	202
8	22	22	357
9	30	30	585
10	42	42	951
20	5,604	5,604	38,307
30	37,338	37,338	599,292
40	204,226	204,226	5,899,292

Table 1.2: A list of the number of different partitionings ($p(n)$), non-isomorphic states, and corresponding state-action pairs for Blocks World instances with between 2 and 40 blocks. Unlike in Table 1.1 on page 16, here I ignore states that differ in block labels but are structurally identical. These numbers result from equation 1.11 instead of equation 1.6.

as described in Section 1.3.3. The use of variables in the hand-coded features provides one form of generalization.[10] For example, turning our attention back to Blocks World, isomorphic states could be treated identically in the idealized case. As I present in Table 1.2, this reduces the number of states in the third column to the number of partitions possible for n blocks, $p(n)$, and equation 1.6 for state-action pairs in the fourth column to:

$$\sum_{i=1}^{p(n)} \left\{ \left(\sum_{j=1}^{n} k_j \right)^2 - k_1 \right\} \tag{1.11}$$

Note how much smaller these numbers are in Table 1.2 when compared to those in Table 1.1.

1.3.5.1 Efficiency and Discussion

The WCTPS cost of computing a value function based on relational representations can be higher than the cost for tabular representations due to the multitude of features and higher than the cost for linear combinations of features due to evaluations of variables and the use of conjunctions of features.

Using relational representations (Section 1.3.5), conjunctions of features that share variables

[10] To use these features as a simpler linear combination of features requires that the variables be replaced with all possible values, exploding the number of features in a process called propositionalization. e.g. The feature `clear(`a`)` would be replaced by as many features as there are blocks: `clear(A)`, `clear(B)`, `clear(C)`, ...

result in the need to solve 3-satisfiability problems [Karp, 1972] to determine whether a feature is active or inactive (in the basis function for linear function approximation), resulting in NP-hard complexity as conjunction lengths grow. Without doing something to reduce computational complexity, as I do in Chapter 4, this could make RRL intractable.

On the other hand, it may be possible to learn from significantly less experience than is necessary with tabular representations since variables support a great deal of generalization. Additionally conjunctions of features allow an RRL agent to handle XOR relationships between features since a feature, such as `on(`a`,` b`)` can take on a different meaning in conjunction with different features, such as either `clear(`a`)` or $\neg$`clear(`a`)`.

With poor WCTPS but potentially good ARtPE, we are presented with a tantalizing representation that has been underutilized in practice. Providing efficient RRL methods to make RRL more useful is the primary focus of this work.

1.4 Related Work and Computational Efficiency

Džeroski *et al.* [2001] is the baseline, full-fledged RRL implementation (implemented using TILDE-RT – not aHTCs). Motivated by a desire to potentially avoid the complexity of RRL, Finney *et al.* [2002] investigated deictic representations (Section 1.3.3.1) to see if they might be a satisfactory alternative to Džeroski *et al.*'s approach for Blocks World (Section 1.3.1). Ultimately, they concluded that "none of the approaches for converting an inherently relational problem into a propositional one seems like it can be successful in the long run. [...] The deictic approach has a seemingly fatal flaw: the inherent dramatic partial observability[11] poses problems for model-free value-based reinforcement learning algorithms."

Regardless, Irodova and Sloan [2005] looked at Džeroski *et al.*'s RRL agent training times for Blocks World (**`stack`**, **`unstack`**, and **`on(a,b)`**) and decided that they could do better. They dropped RRL in favor of linear function approximation over propositionalized representations, effectively creating highly tuned, hand-coded agents and thus losing the generality of the RRL approach. As a further optimization, they were able to hide the true numbers of actions from their agents and present up to 11 categories of action (or examplar actions) to enhance scalability.

The end result is that the agents by Irodova and Sloan can learn to solve **`stack`**, **`unstack`**, or **`on(a,b)`** in four orders of magnitude less time than the RRL implementations existing at that time while providing an optimal policy that is able to be efficiently executed on problem instances of up to at least 800 blocks. See Table 1.3. For comparison, the Q-tree policies from [Džeroski *et al.*, 2001] converged to only 50% optimality for **`stack`** and **`unstack`** and 60% optimality for

[11] Partial observability refers to the problem of an agent that cannot distinguish between different states that must be treated differently for a policy to be effective.

Task	[Džeroski *et al.*, 2001]	[Irodova and Sloan, 2005]
Stack	13,900s*	1.7s
Unstack	36,400s*	1.7s
On(a,b)	44,600s*	2.3s

Table 1.3: Runtimes for 100 episodes of training for Blocks World (Section 2.2.1) with 3-5 blocks and only 45 episodes for [Džeroski *et al.*, 2001]

`on(a,b)`. Irodova and Sloan attributed perhaps a single order of magnitude speedup from hardware advances, leaving them with a three order of magnitude improvement from their approach.

One might ask how Irodova and Sloan [2005] succeeded with propositional representations when Finney *et al.* [2002] were unsuccessful. Irodova and Sloan's approach was simpler in many respects, avoiding the need for deliberate focusing actions and eliminating any hazardous partial observability as a result. **Moreover, in their approach, they manually assigned a separate value function to each category of action.** While their features would be inadequate for deciding which actions to take if shared between all actions, when separated by action, they provided a sufficient signal for learning the task.

In fact, any features would be sufficient for both **`stack`** and **`unstack`**. The action categories are themselves sufficient for making good choices for these tasks. For **`unstack`**, all an agent must learn is that the actions that move a block to the table give the highest reward. Similarly, for **`stack`**, all an agent must learn is that the actions that move a block to one of the highest stacks (of which there should be more than one for only the first decision) give the highest reward. **Since these classes of actions have their own value functions, it is irrelevant what features are contained within them. Put simply, they built the intelligence into the action selection procedure itself**[12]**.**

While the agents by Irodova and Sloan are fast and effective, their custom mapping of value functions onto categories of actions is a level of hand-coding that goes beyond the feature design that is required by RRL agents. Moreover, this extra level of manual action categorization is necessary to allow an agent using linear combinations of features in lieu of RRL to learn these Blocks World tasks. RRL is sufficiently powerful to allow an agent designer to implement agents that are capable of discovering the necessary generalizations for themselves as long as the features themselves are sufficient to distinguish between them. What RRL allows an agent to learn on its own with respect to value function structure will be functionally equivalent to what Irodova and Sloan hand-coded for their action categories.

[12] I assume they did this for **`on(a,b)`** as well, but details are omitted from their paper.

1.5 Discussion

In this chapter, I introduced the evaluation criteria for computational efficiency, WCTPS (Section 1.2.1), and learning efficiency (Section 1.2.2), ARgPE and ARtPE both evaluated or plotted per step. I additionally introduced MDPs (Section 1.1.1), illustrated with the Blocks World domain (Section 1.3.1), and discussed a number of RL methods for learning value functions from which optimal or near optimal policies can be derived. I additionally presented related work and previewed the kind of computational performance I will present in chapter 5 for RRL agents that learn online and incrementally and can pursue objectives that change from episode to episode (Section 1.4).

My objectives over the course of this researched included:

1. Evaluate HTCs as a method that could be made to work incrementally for computationally efficient, online RL.

2. Develop an architecture with implementations of HTCs and TD methods that can be shared between agents for different environments.

3. Optimize my architecture for aHTC value function representations.

4. Apply my RL architectural methods to RRL problems.

5. Increase the expressive power of my architecture RRL to enable it to tackle problems with either fixed or variable numbers of variables and relations.

Over the course of this thesis, I will cover the following material:

1. I explore more traditional Adaptive Tile Coding (ATC) methods, particularly a multilevel tile coding that I refer to as an adaptive Hierarchical Tile Coding (aHTC). An aHTC stores weights not only for the smallest tiles in the architecture but for the more general tiles that were refined along the way. As a result, it implements linear function approximation. I introduced the concept in sections 1.3.4.2 and 1.3.4.4 and begin my evaluation with non-relational RL tasks in chapter 2 and continue it throughout the rest of the thesis.

2. I develop an efficient k-d trie implementation of aHTCs in which one stores weights at every node along the way to the leaves, achieving a two orders of magnitude speedup on my prototype architecture in chapter 3.

3. In order to support RRL, I shift to an implementation of the Rete algorithm [Forgy and McDermott, 1977b] in Chapter 4 that implements a novel RRL grammar for HTCs, allowing for automatic feature extraction and efficient value function refinement and unrefinement.

4. I explore several of the Adaptive Tile Coding (ATC) refinement criteria from Section 1.3.4.3 in the context of this new architecture both to demonstrate the flexibility of the architecture and to verify that it is possible to achieve good ARtPE for **`exact`**, a more complex Blocks World objective, and a taxicab domain as well. The experiments with Blocks World, presented in Chapter 5, demonstrate computational performance that is orders of magnitude greater than that achieved by Džeroski *et al.* [2001].

5. Finally, I extend the capabilities of my agents to allow them to optionally create versions of existing relations with new variables. This effectively allows them to attend to multiple versions of a relation with new variables in order to solve more complex problems for which a known, fixed number of variables relations may be inadequate. I present this functionality and demonstrate its efficacy for **`exact`** and an adventure task in chapter 6.

The contributions of this thesis include are as follows.

1. The primary contribution of this dissertation is a theory and corresponding algorithms for connecting the rule-matching power of the Rete with the needs of RRL to arrive at computationally efficient methods for implementing the first online, incremental RRL architecture.

   (a) I develop aHTCs and demonstrate the efficacy of using hierarchically organized conjunctions of features in terms of WCTPS and ARtPE.

   (b) I develop a novel approach for embedding an aHTC in a data structure for rule-matching to enable it to take advantage of the features that make it suitable for efficient RRL.

2. I provide an RRL implementation that achieves a greater than two orders of magnitude WCTPS reduction in Blocks World tasks over prior work by Džeroski *et al.* [2001].

3. To support making my implementation online and incremental, I modify existing ATCs refinement criteria to make them fully incremental and create a new one as well.

4. I design, implement, and evaluate further extensions to aHTC refinement criteria to further increase ARtPE when allowing unrefinement while minimizing the WCTPS cost of doing so

5. I additionally provide an advance in the theory of RRL. It allows agents that learn online and incrementally to introduce new features with new variables while also being able to continue learning about situations in which no objects satisfy the new variables.

CHAPTER 2

Exploration of Hierarchical Tile Coding

If my ultimate goal were to develop an efficient Relational Reinforcement Learning (RRL) implementation, there are two approaches that I could take. I could begin with the goal of implementing a functional RRL architecture and attempt to see how fast I could make it run. Alternatively, I could begin with simpler non-relational Reinforcement Learning (RL) tasks and to see what kinds of techniques I might be able to make efficient and effective in that space first, leaving the problem of solving RRL tasks for later[1]. This should work, since the underlying Temporal Difference (TD) methods are the same regardless of whether features are propositional or relational.

2.1 Prototype Architecture (2012)

Here I describe the basic execution cycle of my agents. Additionally, I describe my prototype architecture that I implemented to evaluate the use of Adaptive Tile Codings (ATCs) to solve RL problems.

2.1.1 Execution Cycle

Any agent must have some kind of execution cycle. I am concerned solely with agents that base their decisions on a value function that they learn in an online, incremental fashion. Here I assume that the value function consists of weights associated with conjunctions of 0 or more binary features and where features can be grouped by the feature dimension of the state-action space that they describe. (i.e. $x < 0.5$ and $x >= 0.5$ might be two features in one feature dimension, or action $=$ move and action $=$ pickup two features in another feature dimension, ...) Given that I am developing agents that can adapt their value functions over time based on data collected at the fringe (Section 1.3.4.3), for each state one of my agents finds itself in, it will essentially (value function representation steps in **bold**, RL in *italics*):

[1] I take this latter approach since the decision to pursue RRL comes at the end of Chapter 3, at which point I have already explored efficient approaches to non-relational RL.

1. Generate a list of possible actions.
2. For each action, generate a set of features corresponding to the current state of the environment.
3. **Look up value estimates for each action using its set of features as an index into its value function.**
4. **For each action that the refinement criterion requests refinement:**
   (a) **Add an unused feature dimension and new weights as extensions of the largest conjunction of features currently applicable to that action.**
   (b) **Create new fringe nodes by extracting features from the old fringe and adding them to the new conjunctions.**
5. Decide between actions using their value estimates.
6. Execute the selected action.
7. Observe the resultant reward.
8. *Update the value function for the previously selected action (or actions in the case that eligibility traces are being used) accordingly.*

The focus of my work with respect to this cycle is how value estimate lookups and refinements are performed, as listed in **bold** (#3 and #4). For a visual of how this applies to Blocks World (Chapter 5), see Figure 1.13 on page 26.

2.1.2 Refinement Functionality

I extended an existing RL-capable architecture [Laird, 2012; Nason and Laird, 2004] to collect additional metadata in order to determine *when* refining a tile could be expected to help the agent learn more effectively. This must minimally answer the question of *when* to refine the value function but not necessarily *where* or *how* the value function must be refined. Given the design of the base architecture, mechanisms that imply that the current tile requires refinement are the most natural fit.

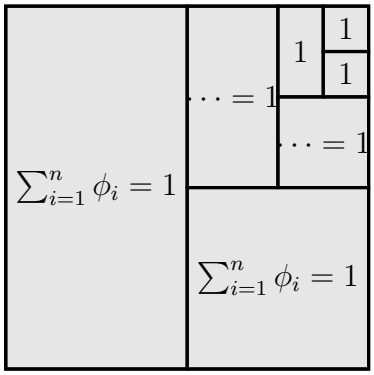
I modified the architecture to generate interrupts when, according to a refinement criterion, the metadata suggested that refinement might be helpful for the current most-refined tile. I took advantage of functionality in the base architecture to do the actual value function refinements.

Given that the functionality in the base architecture results in permanent additions to its agents, coarser tiles are preserved when more specific tiles are added to an agent, resulting in a multilevel ATC. This kind of ATC I shall call an adaptive Hierarchical Tile Coding (aHTC)[2]. Note the increased number of active weights per region when using an aHTC in Figure 2.1b that correspond to overlapping tiles as opposed to only one weight per region when using ATC in Figure 2.1a. The use of a Hierarchical Tile Coding (HTC) results in the need to use linear function approximation (Section 1.3.3), as with any tile coding (Section 1.3.4.1) that consists of multiple overlapping tilings. However, it is worth noting that these aHTCs are imperfectly hierarchical since the base architecture can only create tiles that apply to the current situation and there is no guarantee that same feature will be selected for refinement in the other portions of the state space. See Figure 2.2c for an example of an imperfectly hierarchical aHTC. Since the refinement decision in the lower left quadrant of the state space is performed independently of the earlier refinement decision (Figure 2.2b), it is possible for a feature to be selected that fails to perfectly partition the state space for that level of refinement (Figure 2.2c).

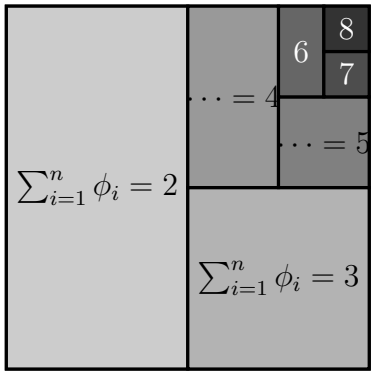
My use of aHTCs raises the question:

Research Question 1 (ARtPE of aHTC vs Flat ATC): *How do adaptive Hierarchical Tile Codings compare to flat Adaptive Tile Codings in terms of Average Return Per Episode (ARtPE)?*

Given that I cannot not remove larger tiles from this prototype architecture as more refined tiles are added, Research Question 1 cannot be addressed with this architecture. The architecture that I develop in Chapter 3 will enable me to address this question.



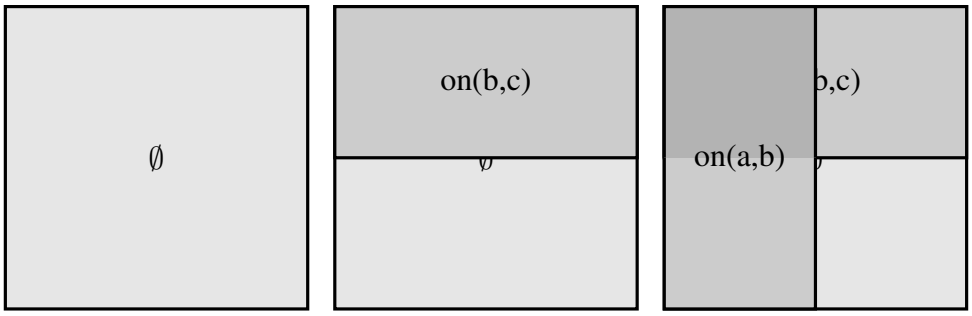
(a) Adaptive Tile Coding (ATC) with one weight per region



(b) adaptive Hierarchical Tile Coding (aHTC) with increasing numbers of weights in more refined regions

Figure 2.1: Flat and Hierarchical Adaptive Tile Codings

[2]I first introduced aHTCs in Section 1.3.4.4. I reintroduce them here since I developed them at this point in my research process.



(a) 1 (b) 1x2 (c) 2x2

Figure 2.2: An imperfectly hierarchical aHTC for Blocks World refined stage by stage, incorporating RRL (Section 1.3.5) using what I engineer in Chapter 4.

2.1.3 Refinement Criteria

In order to finish my prototype, it was necessary to implement at least one criterion to allow my agents to decide *when* to refine a given tile.

2.1.3.1 Influence and Variance

The first criteria I explored were the **influence**, **variance**, and **Stdev_Inf** criteria presented by Munos and Moore [1999a] and described in Section 1.3.4.3. I converted their Bellman equations to incremental TD formulations, but influence can only be approximated roughly when solving with value iteration is not an option.

My proxy for a local influence measure is **TD updates per value estimate contribution**. If we define a weight as active when the basis function ϕ_i (from equation 1.7 on page 18) is 1 rather than 0 for a given action, dividing the number of times that a particular weight is active for an action that is selected by the target policy (for my purposes, the greedy policy) by the number of times that same weight is active for any action at all provides an estimate of the likelihood that that weight influences those that precede it. A weight that is active for the selected action 100% of the time will have a local value of 1, while a weight that contributes only to actions that are not selected will have a local value of 0. This is certainly not identical to the Bellman formulation of **influence** which can have values greater than 1 to indicate probability of repeated visits to a successor state. On the other hand, in the presence of high state aliasing (which an agent is likely to encounter in early states of learning with ATCs and aHTCs) this measure will require less correction over time when using TD methods.

Storing and incrementally updating a global mean and variance for the value of interest (influence, variance, ...) allows an agent to perform a *Z*-test

$$c_i > \mu_c + z\sigma_c^2 \tag{2.1}$$

to determine if a tile is in the top $f\%$, assuming a normal distribution of value estimates[3]. Whenever a weight contributes to a value estimate for an action, I run the corresponding *Z*-test to determine if refinement is necessary. I cover the incremental mean and variance algorithms that I developed in some detail in Appendix B on page 138.

2.1.3.2 Cumulative Absolute Temporal Difference Error

Whiteson *et al.* [2007] used shrinking per-state Bellman error as their measure for determining when to refine the value function. The idea is that once Bellman error stops decreasing, the amount of learning that the agent is capable of for the given set of tiles has plateaued.

Since I am concerned with model-free approaches and do not have access to Bellman error, I work with a proxy: TD error for a state-action pair. The TD error, δ, is the difference between the expected return $Q(s, a)$ and the sum of the immediate reward and discounted future return, $r+\gamma Q(s', a')$ (on-policy) or $r+\gamma \max_{a' \in \mathcal{A}} Q(s', a')$ (off-policy). One way of looking at TD error is that it might be indicative of incorrect expectations. It can result from state aliasing due to an inadequately refined state space but, unfortunately, it can also result from stochasticity of the environment. Without knowledge of baseline stochasticity, I know of no way to distinguish between the two, so my agents assume that TD error is indicative of the former since it is something they can attempt to resolve.

Cumulative Absolute Temporal Difference Error (CATDE) can be defined formally for feature ϕ_i:

$$\text{CATDE}_{t_n}(\phi_i) = \sum_{t_0}^{t_n} \left| \frac{\delta_{i,t}}{\sum_{i=1}^{n} \phi_{i,t}(s_t, a_t)} \right| \tag{2.2}$$

Tracing the lowest Bellman error for a tile or TD error for a state-action pair is sensible as a global measure of *when* to refine the value function, but if one wishes to be able to say something about *where* to refine the value function using TD error, a different approach is required. Incrementally tracking the mean and variance for CATDE for all tiles in my architecture allows my agents to again use a *Z*-test to choose tiles in the top $f\%$ for refinement. Due to the cumulative nature of CATDE, it ends up giving a combined measure of poor expectations and importance to a critical path for the agent. I use CATDE for the majority of the experiments in this chapter and in Chapter 3 because I observed that it performed better than my TD updates refinement criterion once I began evaluating my agents in terms of ARtPE. The results that resulted in this observation are lost, but I will do a brief empirical evaluation of the differences that result from plotting ARtPE episodically or stepwise in Section 5.5.

[3] If the distribution is not normal, it may not be exactly $f\%$, but it will then be some $g\%$ instead. It will still work in essentially the same manner.

2.2 Exploratory Experiments

I test this prototype system on Blocks World and Puddle World.

2.2.1 Blocks World – Proof of Concept

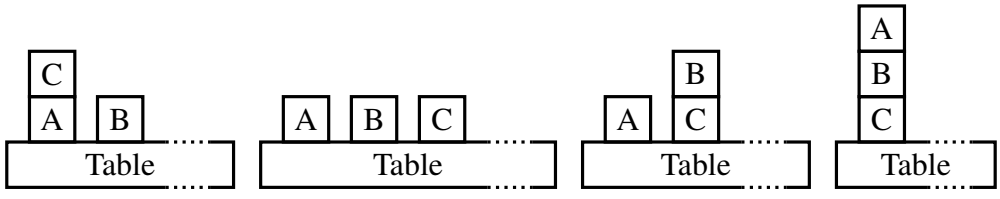
The initial version of Blocks World – the only version of Blocks World that I explored with this architecture – was rigidly defined. It consisted of a fixed initial configuration of 3 blocks and an **`exact`** objective as described in Section 1.3.1 on page 11, but unchanging from episode to episode. This objective is depicted in Figure 2.3d and the optimal sequence of actions is depicted in Figure 2.3. Since this objective is not included among the agent's features, it must be learned and encoded in the value function as a result.

I used Blocks World primarily as a proof of concept and testbed while implementing value function refinement. As depicted in Figure 2.4, my prototype agent converges to approximately -6 ARtPE by 500 steps. Since an optimal solution results in asymptotic -3 ARtPE, this result is not particularly impressive, but it suffices for purposes of demonstrating that the agent is capable of learning to solve this Blocks World task.

Limitations of the architecture include the following:

1. No fringe of possible next steps receives TD updates, making it very difficult to address the problem of *how* to refine the value function. This is a minor problem for environments that present small numbers of features, but a major problem for more complex environments.

2. Though possible features (in this case `clear(a)` and `in-place(a)` relations) are described without reference to any specific constants, the refinement mechanism as implemented ultimately replaces the variables relevant to my features with the constants that they point to at the time of refinement. This limits the ability of my architecture to implement RRL. Notably, this limitation results in new conditions being tied to the specific action in question, resulting in separate aHTCs for each action (i.e. moving `A` to `B`, moving `C` to the `TABLE`, ...).



(a) Initial (b) Optimal step 1 (c) Optimal step 2 (d) Final

Figure 2.3: Optimal solution to fixed Blocks World (Section 2.2.1)

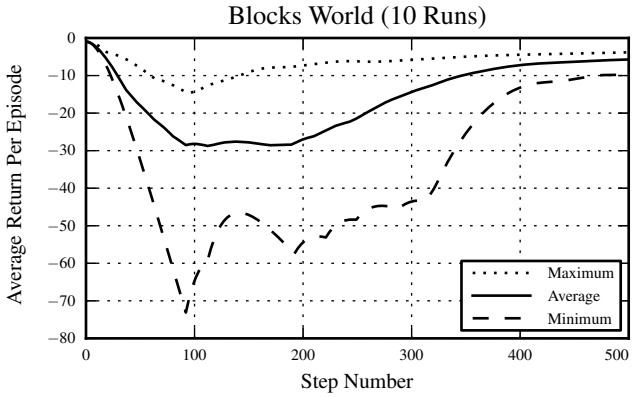


Figure 2.4: ARtPE of a prototype agent in Blocks World using an aHTC with CATDE – agents were trained with $z = 0.84155$ and tile refinement inhibited until no updates have been experienced for 3 steps within a single episode in this early result.

3. Additionally, this architecture can only add rules that apply to the situation that the agent is currently experiencing. As implemented, there is no guarantee that once `on(A,B)`, for example, is selected as the first feature for value refinement that $\neg$`on(A,B)` will be selected as the first feature for refinement for the other half of the state-space. This can result in an ATC that is not strictly hierarchical and can have overlaps for a given number of features. It might be possible to work around this, but it would be non-trivial. This limitation also means that were my agents capable of easily eliminating more general tiles as they add more specific tiles, that it would be unwise for them to do so regardless due to their inability to generate all subtiles at once.

4. Finally, I did not develop a mechanism to ensure that refinement requests would cease to be generated once all features had been exhausted. While a minor technical point, it actually makes it difficult to generate empirical results for Blocks World averaged across many runs since all features can be exhausted by step t with some probability, causing agents to loop infinitely after the next request for refinement. That being said, I was able to generate results as depicted in Figure 2.4

It is worth noting that the lack of a mechanism to address the problem of *how* to refine the value function can be used as a motivation for implementing aHTCs specifically, rather than allowing arbitrary conjunctions of features to be added to the value function without regard for hierarchy and overlap as explored in iFDD [Geramifard *et al.*, 2011]. Without said mechanism, the only guarantee that adding a conjunction of features might result in a value function with increased learning capabilities is to ensure that the conjunction is larger than all that preceded it. Computational and memory efficiency arguments can be made for aHTCs as well, but this was another early motivation for pursuing value function representations that were strictly hierarchical.

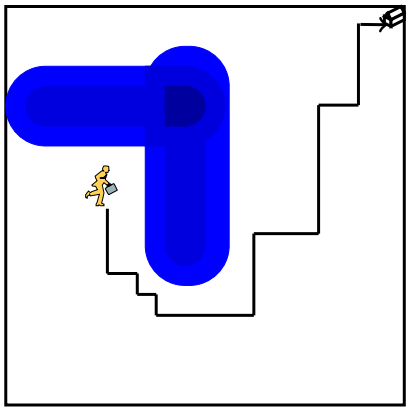


Figure 2.5: Puddle World

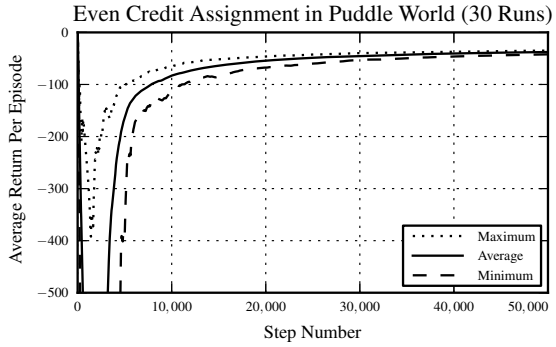
2.2.2 Puddle World – Proof of Efficacy

Puddle World [Sutton, 1996] is an environment in which the agent's state is described using continuous values $[0, 1]$ in two dimensions. The environment contains two "puddles" that are capsule shaped regions the depth of which increases from their edges to their centers. The agent can move North, South, East, or West from its current position. The environment is fully observable, but the agent's steps are stochastic, resulting in real-valued step sizes between 0.04 and 0.06 units. As the x and y positions are real-valued, the state-space is infinitely divisible, and therefore not discretizable. The agent's objective is to arrive at a goal region in the upper right corner ($x + y > 1.9$) starting from a fixed starting position (0.15, 0.45) with two puddles of radius 0.1 defined by line segments between (0.1,0.75)-(0.45,0.75) and (0.45,0.4)-(0.45-0.8) preventing the agent from moving directly toward the goal. The agent receives a penalty of -1 for each step along the way, an additional penalty of up to 400 per puddle (proportional to the depth of each puddle at its current location in order to encourage the agent to go around if possible), and no additional reward for reaching the goal.

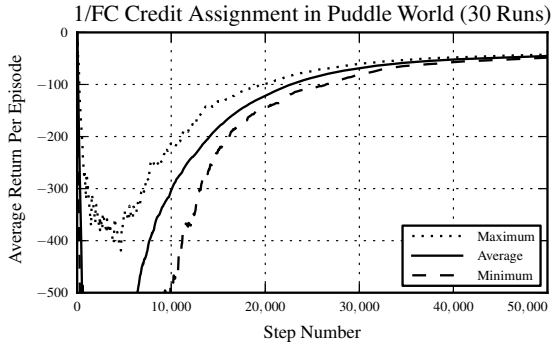
While this environment is more complex in some respects than the version of Blocks World described in Section 2.2.1 on page 39, some of the limitations described therein are easier to address:

1. An agent can always refine along whichever dimension is longest (i.e. has been selected less frequently) as a heuristic to decide between refining along the x or y dimension [Moore and Atkeson, 1995; Reynolds, 1999].

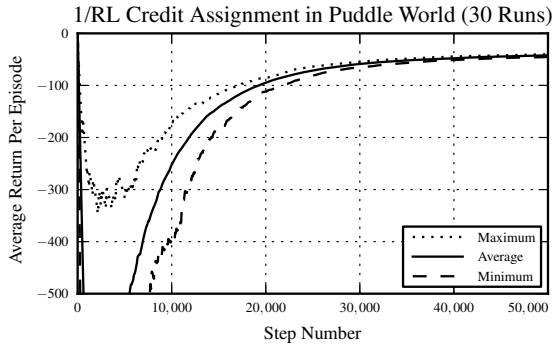
2. There are only the x and y variables to consider. Puddle World is a non-relational environment, because there is never a need to evaluate whether a variable shared between different relations refers to the same object in the environment. Refinement is still done independently for actions associated with each cardinal direction.



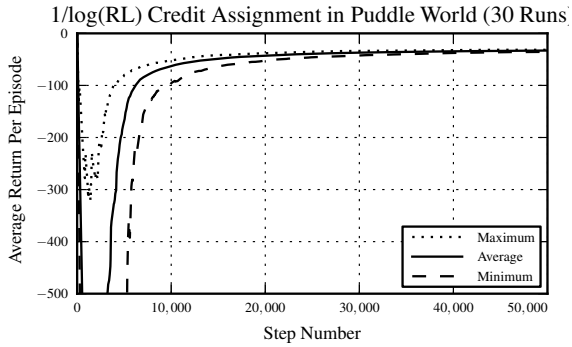
(a) Even



(b) Inverse Count of VEC



(c) Inverse Count of TDU



(d) Inverse Logarithm Count of TDU

Figure 2.6: ARtPE of my prototype agents in Puddle World with non-adaptive Hierarchical Tile Codings (naHTCs) with all tilings from 1×1 to 16×16 (i.e. all tilings present from step 1)

3. Limitation 3 is not resolved.

4. The agent will essentially never run out of features since the features are infinitely divisible up until the limitations of floating point numbers.

2.2.2.1 Credit Assignment Between Overlapping Tiles

An additional question I explored was whether credit assignment between overlapping tiles ought to be completely even, or whether smaller tiles ought to receive more credit than larger ones. The intuition was that giving overly large, more general tiles equal credit to smaller, more refined tiles might inhibit an agent's ability to converge on near optimal policies in the limit and that another credit assignment strategy might improve convergence rates.

Research Question 2 (Credit Assignment for HTC): *Can one improve the ARtPE of HTC with an alternative to even credit assignment?*

Strategies I included for this architecture are:

1. **Even** – $\frac{1}{\sum_{i=1}^{n} \phi_i}$
   This is the standard credit assignment strategy for linear function approximation (Section 1.3.3).

2. **Inverse Count of Value Estimate Contributions (VEC)** – $\frac{\text{VEC}_i^{-1}}{\sum_{i=1}^{n} \phi_i \text{VEC}_i^{-1}}$
   This gives more credit to weights which are used in decision-making.

3. **Inverse Count of Temporal Difference Updates (TDU)** – $\frac{\text{TDU}_i^{-1}}{\sum_{i=1}^{n} \phi_i \text{TDU}_i}$
   This gives more credit to weights on the actual critical path, and less to actions that are one off the critical path.

4. **Inverse Logarithm TDU** – $\frac{\ln(\text{TDU}_i)}{\sum_{i=1}^{n} \phi_i \ln(\text{TDU}_i)}$
   This reduces the rate at which more credit is given to the weights of more refined tiles compared to the previous credit assignment strategy, keeping it closer to **even** credit assignment during earlier stages of learning.

Learning to solve Puddle World (Section 2.2.2) with non-adaptive Hierarchical Tile Codings (naHTCs) and these different credit assignment strategies, I take the ARtPE and plot it per step in Figure 2.6.

I had some success with speeding up learning using **Inverse Logarithm TDU** credit assignment (Figure 2.6d), but the two other credit assignment strategies offered performance that is clearly inferior to the baseline performance of **Even** credit assignment (Figure 2.6a). Since **Inverse Logarithm TDU** is the credit assignment strategy that assigns credit the second most evenly, it seems unlikely that significant deviation from **Even** credit assignment can be beneficial. Given these results we cease exploration into alternative credit assignment strategies for the remainder of this thesis.

2.2.2.2 Puddle World with aHTCs

Learning to solve Puddle World with aHTCs instead, performance and value function structures are depicted in Figure 2.7. The tile codings for each action, North, South, East, and West, are depicted in figures 2.8b-2.8e with a logical OR of all four depicted in Figure 2.8a. I present these figures primarily as evidence that my proof of concept was functional, but one can see that the aHTC for each cardinal direction is most refined along the critical paths that the agent is most likely to take. Therefore, the aHTC for East (Figure 2.8c) has undergone extensive refinement while the aHTC for West (Figure 2.8d) has been minimally refined, which is the result I had expected.

Performance of agents using the **TD updates per value estimate contribution** criterion is somewhat inconsistent, as depicted in Figure 2.7. Although all seem to converge in the limit, there is a significant gap between the convergence rates of the best and worst runs. While the best convergence rate is in line with the best of my agents using non-adaptive Hierarchical Tile

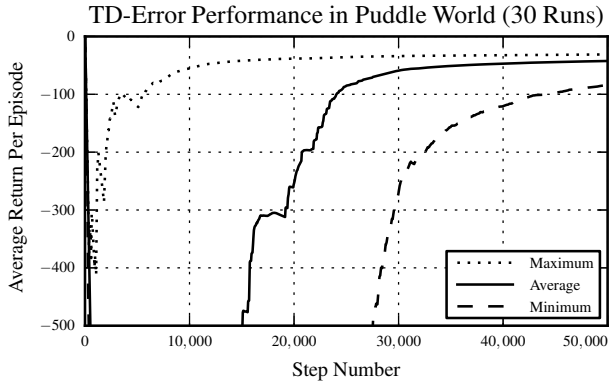


Figure 2.7: ARtPE of my prototype agents in Puddle World with aHTC with tilings from 1×1 up to 16×16 – trained using $z = 0.84155$, $\alpha = 0.2$, TDU

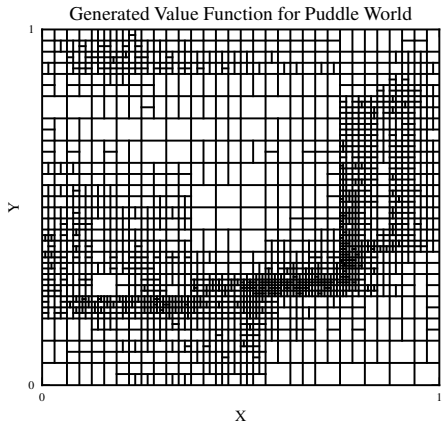
Codings (naHTCs), the worst is delayed by almost $30,000$ steps. This criterion focuses on refining the critical path, but until the critical path points to the true end goal, it is capable of refining parts of the state space that are unimportant. Given these results, I decided to abandon influence-based criteria and to focus on CATDE in Chapter 3.

2.3 Discussion

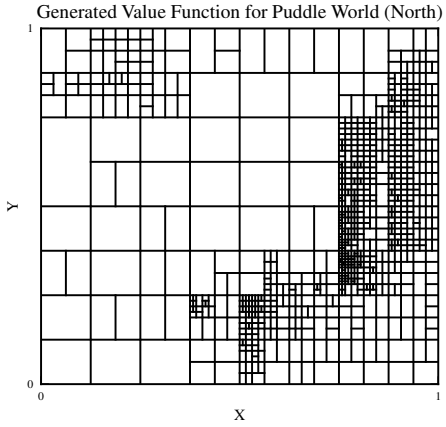
While I described Average Regret Per Episode (ARgPE) and ARtPE in detail in Section 1.2.2, it was not actually obvious from the beginning to use these measures to evaluate my agents. The major contribution of this chapter is devising ARgPE and ARtPE for the reasons described in the introduction and settling on their use for evaluation of my agents.

A minor contribution of this chapter is developing two measures for deciding *when* to refine the value function, TD updates per value estimate contribution and CATDE, and settling on CATDE as a baseline for the rest of this thesis. This choice resulted from observation that CATDE appeared to perform better once I switched from a purely episodic evaluation to one based on ARtPE evaluated per step.

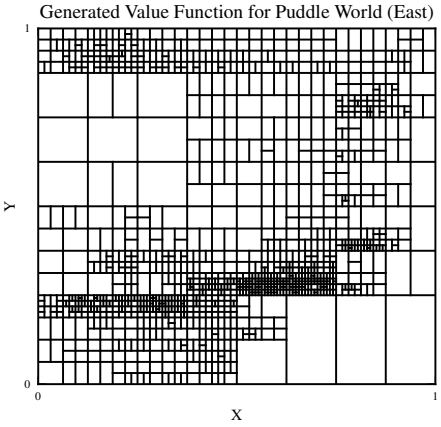
Additionally, the prototype described in this chapter served as a proof of concept for a RL architecture that uses aHTCs to represent the value function. Prior work [Reynolds, 1999; Munos and Moore, 1999a; McCallum, 1996; Davies, 1996] was based on ATCs and not aHTCs. Therefore, it is another minor contribution of this chapter to have verified that using aHTCs for value function representation does not undermine the basic strategy for value function refinement that they explored.



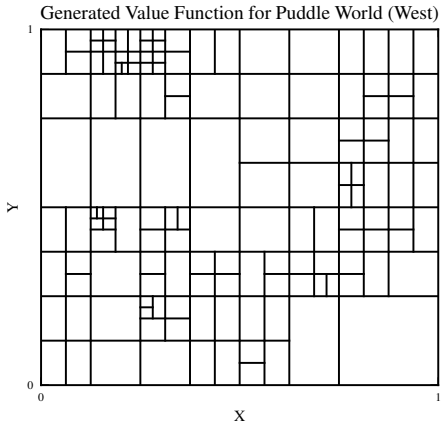
(a) All Tilings



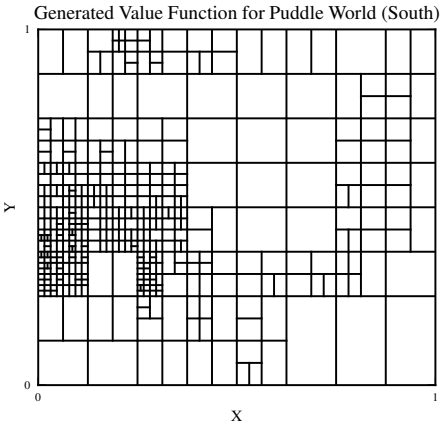
(b) North Tilings



(c) East Tilings



(d) West Tilings



(e) South Tilings

Figure 2.8: Tile refinement inhibited until no updates have been experienced for 20 steps within a single episode

CHAPTER 3

Computationally Efficient Adaptive Hierarchical Tile Coding

The prototype architecture that I developed in chapter 2 had a number of limitations as discussed in Section 2.3. I wanted to continue to investigate refinement criteria (Section 1.3.4.3) and adaptive Hierarchical Tile Codings (aHTCs) (Section 1.3.4.4), but experiments were slow to execute.

I had previously done some work on Hierarchical Reinforcement Learning (HRL) in a standalone architecture [Bloch, 2011] and believed that experimenting with designs in a more specialized architecture would be more efficient in terms of development time than attempting to optimize my prototype. Therefore I set the development of a new, more efficient architecture as a goal. I call this architecture Carli[1] (https://github.com/bazald/carli). The version I discuss in this chapter is Carli for Propositional Representations (Carli-Prop).

My objective in this chapter is not to resolve the functional limitations of my prototype, but simply to speed up future work by advancing my architecture's computational efficiency (minimizing Wall-Clock Time Per Step (WCTPS)) while maintaining existing functionality. The changes I introduce in Chapter 4 will address the functional limitations. At that point, I will be able to revisit Blocks world and Relational Reinforcement Learning (RRL).

3.1 Carli-Prop

It is important to remember that the efficiency of Adaptive Tile Codings (ATCs) and aHTCs depends on the efficiency of weight lookups for any given state description, fringe weight (or subtile) tracking, refinement criteria, and value function adaptation including fringe maintenance. Some sort of hierarchical, tree-like data structure is a good fit for these tile codings since more refined tiles depend on conjunctions of features that can correspond to nodes in a decision tree. Addition-

[1]The letters of Carli are taken from **C++**, Artificial Intelligence (**AI**), and Reinforcement Learning (**RL**) and jumbled a bit. The architecture is cleverer than its name.

ally, manipulation of nodes at or near the fringe (or leaf) nodes of a tree are simple to manipulate without directly modifying other parts of the data structure, allowing for small WCTPS.

I start by implementing a simpler, standalone version of the execution cycle described in Section 2.1.1 on page 34 with the expectation that it will be more computationally efficient. Then I proceed to develop a more efficient replacement for value function lookups and refinements, given the high complexity of that process in my prototype architecture.

3.1.1 A *k*-Dimensional Trie Value Function Implementation

Using a *k*-dimensional trie (*k*-d trie) is not a novel contribution of Carli-Prop. Munos and Moore [1999b] used a *k*-d trie in their ATC implementation. A *k*-d trie is a perfect fit for Carli-Prop since I care about storing weights at each node along the way to the leaves (corresponding to conjunctions of features of increasing complexity) in my implementation of an aHTC.

To index into my *k*-d trie-based value function, I take my features for the corresponding action and find the one that matches the root node. I then index into the next level of the *k*-d trie using the remaining features, and so on until the depth of the *k*-d trie is exhausted.

In Carli-Prop, I am able to better control the ordering in which features are refined, significantly decreasing the likelihood that an imperfectly hierarchical aHTC (as depicted in Figure 1.3.5 on page 27) will result. A *k*-d trie in several stages of refinement is depicted in Figure 3.1.

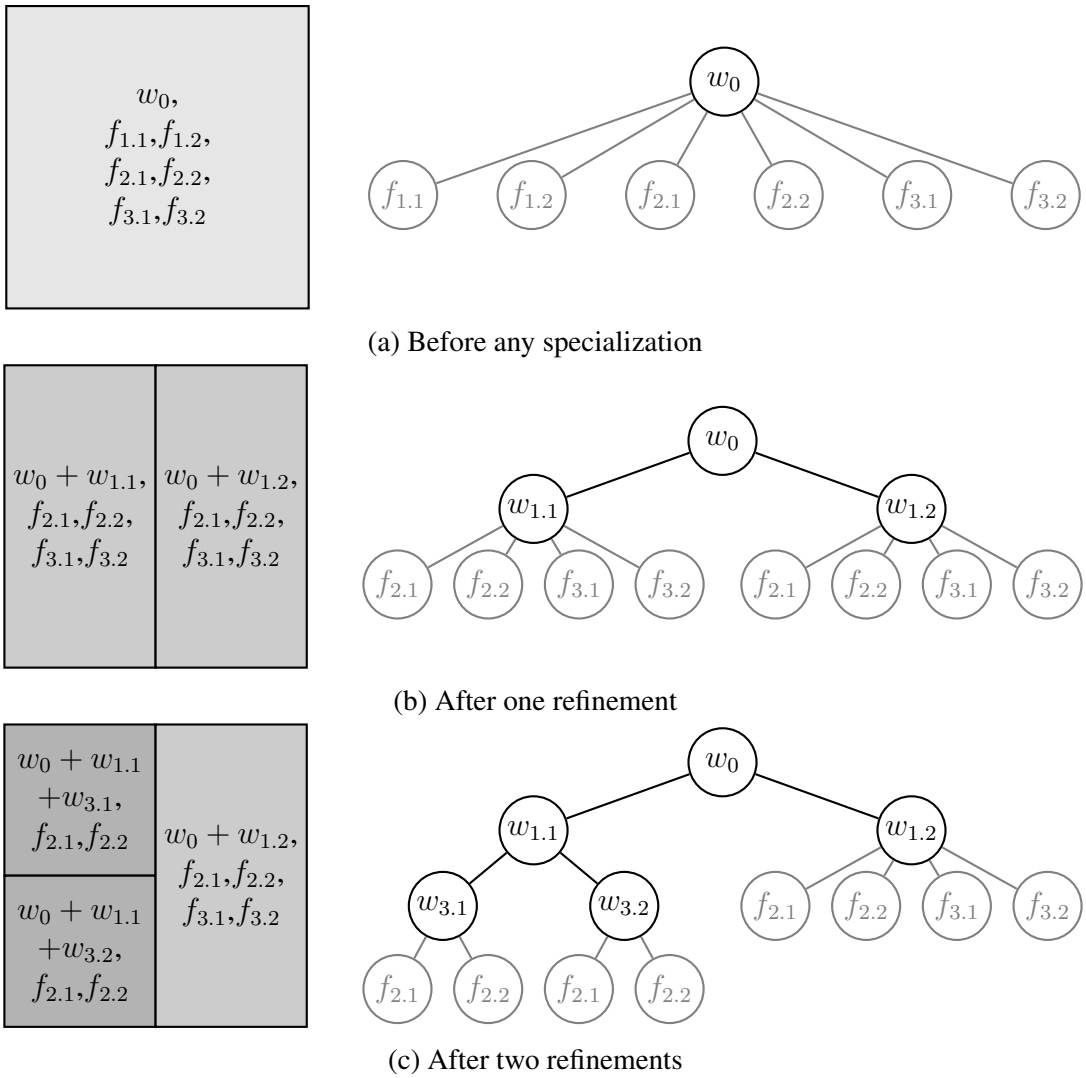
3.1.2 Fringe Nodes

Candidate features are stored in fringe nodes one step past the largest conjunction or, alternatively, one step past the smallest or most refined tile for each part of the state space. The weights stored in fringe nodes must be treated as full Q-values, independent of linear function approximation, and they must not contribute to the value estimates used for decision-making. They allow my agents to track metadata for each candidate successor conjunction or subtile in order to inform the decision of *how* to refine the value function.

The only kinds of features supported by Carli-Prop were Boolean features and ranged, continuous-valued features. When it came time to use fringe nodes for decision-making, I focused on ranged, continuous-valued features only. The resultant decision procedure is:

1. Always choose the least refined feature dimension. (i.e. In Puddle World, x must be selected if y has been selected once more than x in the feature conjunction for the most refined tile.)

2. In the case that there exist two or more features that had been selected the minimum number of times, choose the feature dimension that gives the largest separation between the successor values – a **value** criterion (Section 1.3.4.3).



(a) Before any specialization

(b) After one refinement

(c) After two refinements

Figure 3.1: *k*-d trie value function refinement with fringe nodes

3.2 Evaluation of Carli-Prop

Here I do my evaluation of my Hierarchical Tile Coding (HTC) implementation in Carli-Prop. Since the introduction of Carli-Prop marks my departure from my prototype architecture to one that is more rigorously designed, I now evaluate non-adaptive Hierarchical Tile Codings (naHTCs) against an alternative, traditional approach to tile coding with linear function approximation (Section 3.2.1). Building on that result, I can compare aHTCs to naHTCs and evaluate the utility of adaptivity (Section 3.2.2). Circling back, I can evaluate how aHTCs compare to non-hierarchical or flat ATCs as well, completing my evaluation of the utility of both hierarchy and adaptivity (Section 3.2.3). Finally, I begin my computational efficiency analysis, comparing Carli-Prop agents to that of agents implemented using my prototype architecture in Section 3.2.4.

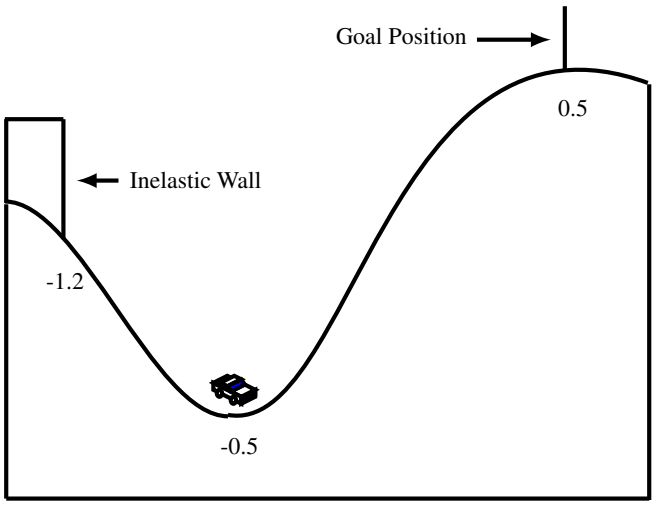


Figure 3.2: Mountain Car

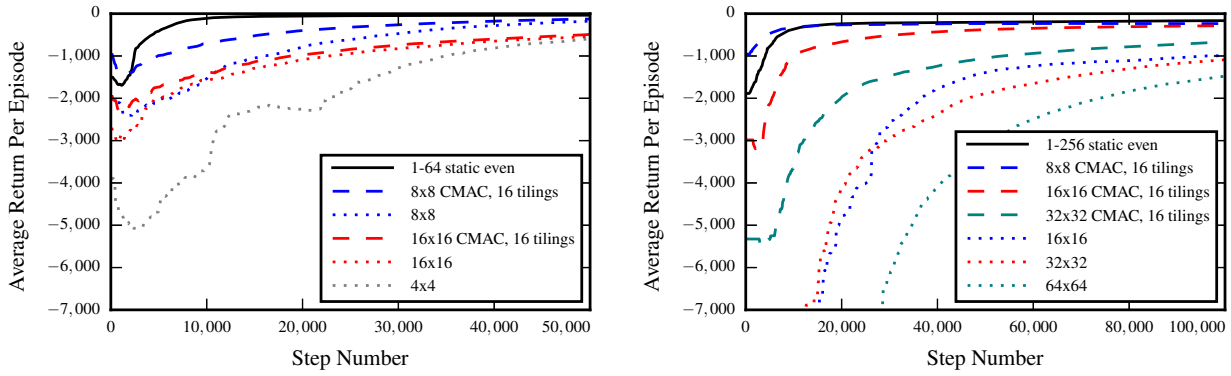
3.2.1 Non-Adaptive Hierarchical Tile Coding vs Traditional CMACs

The first new experiment I conducted with Carli-Prop was to compare the performance of agents using naHTCs to that of agents using individual tilings and Cerebellar Model Arithmetic Computers (CMACs).[2] My CMACs consist of 16 identical tilings of a specified resolution (16x16 or 32x32) where the first tiling exactly covers the state space and each subsequent tiling is offset by a percent of the state space ($1/16$ or $1/32$ respectively) in both the positive x and y directions (see Figure 1.9b on page 21). I wish to compare the efficacy of HTCs to that of more traditional tile codings that also take advantage of linear function approximation such as CMACs.

Puddle World (Section 2.2.2) is a useful task on which to evaluate my agents since it is impossible to perfectly tile the state space given the circular designs of the puddles. In addition to Puddle World, I evaluate my agents in the Mountain Car domain (Figure 3.2), which makes it similarly impossible to perfectly tile the state space, in order to demonstrate that the efficacy of Carli-Prop is not limited to Puddle World. The canonical Mountain Car [Moore, 1991], is a two-dimensional, continuous valued world, where an agent can control the car's motor (left, idle, right), and where the goal is for the agent to move the car from the basin, at rest, to the top of the mountain on the right. Given gravity $-0.0025\cos(3x)$, and a car with power 0.001, the car is incapable of climbing the mountain starting from rest. It is necessary to build up potential energy by backing up the hill on the left before moving to the right. The agent receives a penalty of -1 for each step with no penalty for colliding with the wall and no additional reward for achieving the goal.

In these experiments I used an Epsilon-greedy (ϵ-greedy) exploration strategy ($\epsilon = 0.1$ for

[2] I had previously compared naHTC only to one another using different credit assignment strategies. (See Section 2.2.2.1.)



(a) Puddle World

(b) Mountain Car

Figure 3.3: ARtPE averages over 20 runs with single tilings, traditional CMACs, and a naHTC (labeled "static even" for even credit assignment between the different levels of the hierarchy).

Puddle World and $\epsilon = 0.01$ for Mountain Car) with a random tiebreak, Q-learning with a learning rate of 0.1 in Puddle World and 1.0 in Mountain Car, and a discount rate of 0.999, with weights initialized to 0.

The results in Figure 3.3 represent an average of 20 runs. The Average Return Per Episode (ARtPE) improves the fastest for the agents that use naHTCs (labeled "static even" for even credit assignment between the different levels of the hierarchy) when compared to individual tilings and most CMACs. Additionally, terminal policies are nearer optimal in the limit for naHTCs when compared to individual tilings and CMACs.

I additionally ran experiments with agents using single 32x32 and 64x64 tilings for Puddle World. However, they did not begin to converge until more than $50,000$ steps had passed, so only smaller tilings and CMACs are included for comparison with my naHTCs in Figure 3.3a. For Mountain Car, single 4x4 and 8x8 tilings are insufficient to learn near optimal policies and single 128x128 and 256x256 do not converge in a reasonable amount of time, so only single 16x16, 32x32, and 64x64 tilings are included alongside the CMACs for comparison with my naHTCs in Figure 3.3b.

Using only two tilings instead of full naHTCs closely matches related work [Zheng *et al.*, 2006; Grzes and Kudenko, 2008; Grzes, 2010] in which their CMACs appear to be strictly hierarchical. I additionally tested naHTCs with varying subsets of the tilings, 1x1 through 64x64, including omitting the most specific tilings or the most general tilings, and none achieved better performance than using the complete hierarchy. This is evidence that full naHTCs are better than these more limited hierarchical CMAC approaches. From this I conclude that using HTCs as the basis for the value functions in my architecture has merit.

naHTCs do significantly better than any individual tiling in both Puddle World and in Mountain Car. This result dispels any possibility that the advantage of using a naHTC is just in hedging the

bet as to which level of generalization is best. **From this I conclude that naHTCs are an effective form of CMACs.** This also hints at a possible answer to Research Question 1: *"How do adaptive Hierarchical Tile Codings compare to flat Adaptive Tile Codings in terms of ARtPE?"*

Hypothesis 1 (aHTC Faster Than ATC): *An adaptive Hierarchical Tile Coding (aHTC) will result in better Average Regret Per Episode (ARgPE) or Average Return Per Episode (ARtPE) than an Adaptive Tile Coding (ATC) since a non-adaptive Hierarchical Tile Coding (naHTC) learns faster than any individual tiling (such as 8x8 or 16x16).*

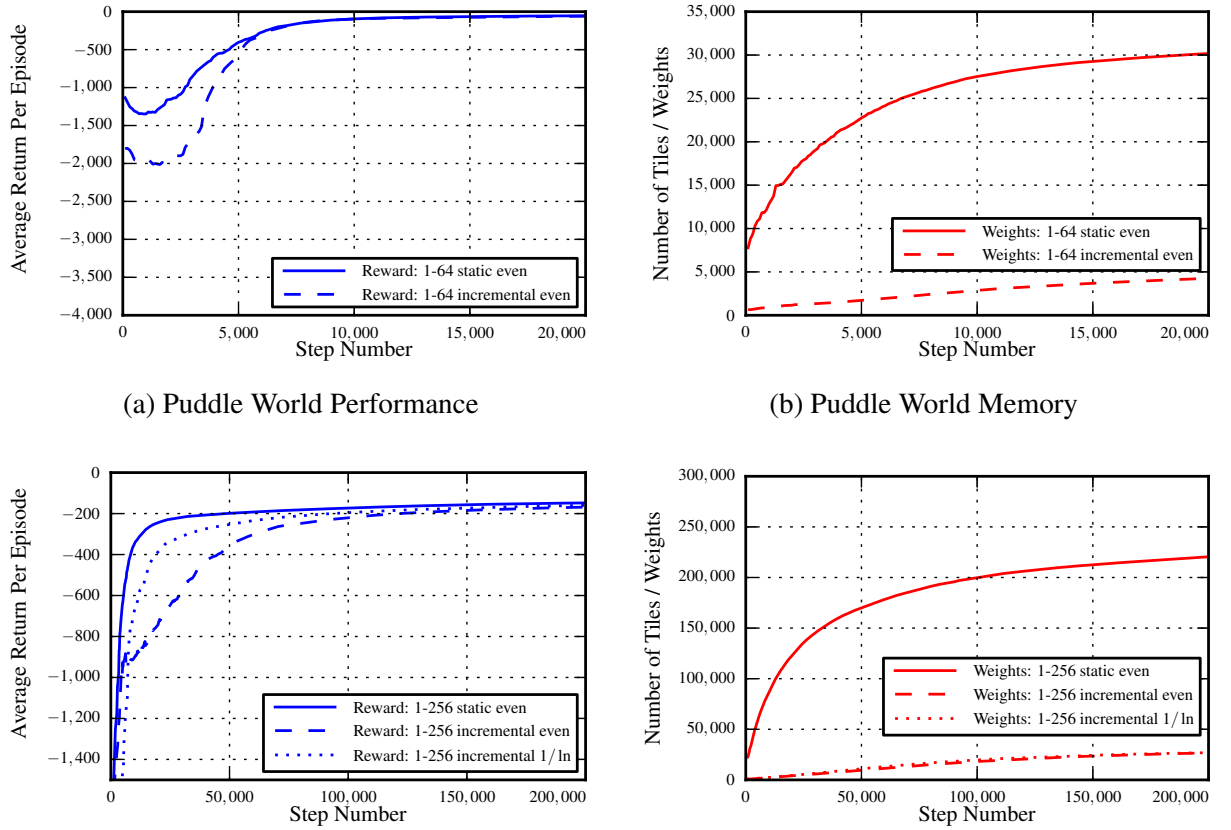
That Hypothesis 1 holds true is not obvious, since it could be the case that there exists an individual tiling that could allow the best possible ARtPE for any given point in time. However, in lieu of finding such a tiling if it is possible at all, it seems likely *a priori* that continuing generalization at coarser levels of the aHTC should help learning because it will allow an agent to refine aggressively while still benefiting from broader generalization that would otherwise be terminated by premature refinement. I will provide a comparison of the performance of agents using aHTCs and ATCs in Section 3.2.3.

3.2.2 Adaptive Hierarchical Tile Coding vs Non-Adaptive HTC

Now I move on to aHTCs which use the Cumulative Absolute Temporal Difference Error (CATDE) criterion (Section 2.1.3.2). Figures 3.4a and 3.4b show performance and memory usage data for the Puddle World domain. I present data for naHTCs (static in the figures) with tilings of resolutions 1x1 through 64x64 and for aHTCs (incremental in the figures) with only tilings of resolutions between 1x1 through 2x2 in the beginning. By 10,000 steps, the ARtPE of the aHTCs has caught up to within 2.6% of the ARtPE of the naHTCs. However, the aHTCs use only 10.5% as many weights as the non-incremental naHTCs.

Figures 3.4c and 3.4d present corresponding data for Mountain Car. I present data for naHTCs with tilings of resolutions 1x1 through 256x256 and for aHTCs with only tilings of resolutions between 1x1 through 2x2 in the beginning. At 100,000 steps the ARtPE of the aHTCs is still 27% worse than that of the naHTCs given the initial delay in learning. However, the aHTCs use only 9.1% as many weights as the naHTCs. aHTC agents using **Inverse Logarithm TDU** credit assignment (Section 2.2.2.1 on page 42) performed a bit more consistently than those using **Even** credit assignment while using the same amount of memory.

From this I conclude that it is possible to build aHTCs that have ARgPE or ARtPE nearly in line with what can be achieved using naHTCs even in domains where the problem of feature selection is relatively unimportant. Therefore I posit:



(a) Puddle World Performance

(b) Puddle World Memory

(c) Mountain Car Performance

(d) Mountain Car Memory

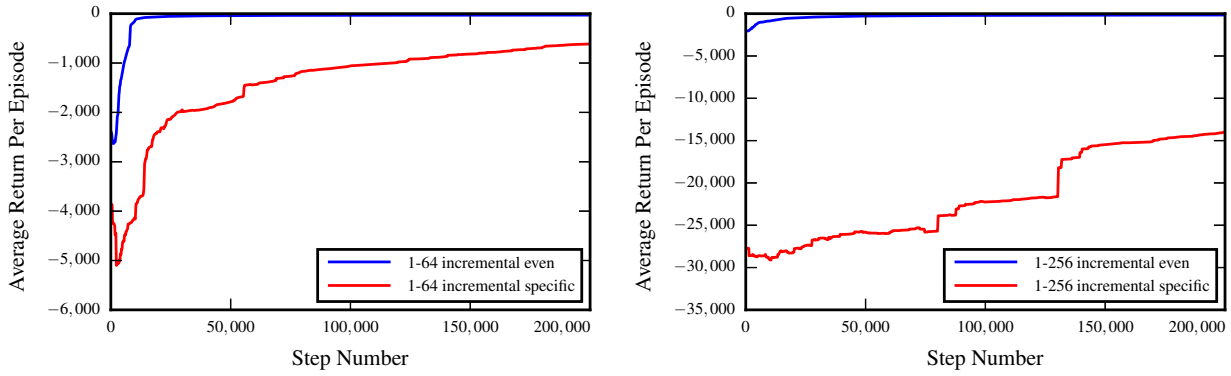
Figure 3.4: ARtPE averages for 20 runs of agents using aHTCs (incremental) with various credit assignment strategies and naHTCs (static)

Hypothesis 2 (Adaptive HTC Superior to Non-Adaptive HTC): *For problems with more difficult feature selection problems, adaptive Hierarchical Tile Codings (aHTCs) will result in better ARgPE or ARtPE than naHTCs since they will allow better choices to be made about which features result in earlier refinements.*

aHTCs should surpass naHTCs in domains where random ordering of features will perform worse and in which *a priori* good orderings are unavailable. I additionally hypothesize:

Hypothesis 3 (Rerefinement Potentially Useful with Adaptive HTC): *For complex problems, not only will Hypothesis 2 hold, but the agent may be able to do even better by undoing refinements in favor of ones that, in retrospect, may allow more efficient Temporal Difference (TD) learning.*

Considering these hypothetical advances in ARgPE or ARtPE in conjunction with the significantly more compact value functions that result from aHTCs, it seems likely that aHTCs will be a valuable technique as I move forward with problems with more difficult feature selection problems. It will allow my agents to achieve good ARgPE or ARtPE while minimizing WCTPS, since WCTPS at



(a) Puddle World

(b) Mountain Car

Figure 3.5: ARtPE averages for 20 runs of agents using aHTCs and aHTCs using "specific" credit assignment to simulate non-hierarchical ATCs

least partially depends on the number of weights that must be evaluated each step. I will both evaluate these hypotheses and demonstrate the connection between the number of weights and WCTPS in my results in Chapter 5.

3.2.3 Adaptive Hierarchical Tile Coding vs Adaptive Tile Coding

I additionally compared my aHTCs against ones that give all credit to the most refined tiles. The "specific" credit assignment strategy simulates an ATC implementation by assigning all credit to only the most refined tiles and freezing weights associated with more general tiles as a result. This allows us to address Research Question 1 (i.e. do aHTCs speed learning merely because they are adaptive, or do the more general tiles help) without having to complicate my implementation by enabling the outright removal of coarser tiles as more refined tiles are added. The results in Figure 3.5 demonstrate that the ARtPE of the agents using aHTCs is greater than that of the agents using ATCs.

3.2.4 Computational Efficiency

In order to compare the computational performance of my original prototype agents to that of my agents implemented in Carli-Prop, I ran them with parameters set as close to one another as possible.

In Section 2.3, I observed that my prototype agents took 5.926 seconds on average to execute Blocks World for 1000 steps on a i7-7700HQ CPU. For comparison, the Carli-Prop agents take 0.049 seconds on average – a speedup factor of 120.

For Puddle World, I chose to run 50,000 steps instead. This results in an average runtime of 51.426 seconds for my prototype agents and 0.738 seconds for Carli-Prop. This is a speedup factor

Environment	Prototype	Carli-Prop	Speedup
Blocks World 2.3.1	5926µs	49µs	120
Puddle World 2.3.2	1028µs	15µs	70

Table 3.1: Time per step for Blocks World (Section 2.2.1) and Puddle World (Section 2.2.2)

of 70 for Carli-Prop relative to Blocks World.

These results, presented as time per step in Table 3.1, reveal a two order of magnitude speedup over my prototype architecture in terms of WCTPS.

3.3 Discussion

In this chapter, I introduced Carli-Prop, a new architecture that implements aHTCs using k-d tries. Carli-Prop is able to provide guarantees of correctness for aHTCs that the prototype architecture in Chapter 2 was unable to provide. Thanks to this advance, I presented strong evidence that aHTCs perform better than non-hierarchical ATCs in Puddle World and Mountain Car, confirming Hypothesis 1, that *"an adaptive Hierarchical Tile Coding (aHTC) will result in better Average Regret Per Episode (ARgPE) or Average Return Per Episode (ARtPE) than an Adaptive Tile Coding (ATC) since a non-adaptive Hierarchical Tile Coding (naHTC) learns faster than any individual tiling (such as 8x8 or 16x16)."*

While it is only a minor contribution of this chapter since we have yet to incorporate RRL, I provided evidence that Carli-Prop provides a two order of magnitude performance improvement over my prototype architecture. Attempting to maintain this performance advance while incorporating RRL forms the basis for the ideas described in Chapter 4 – the major contribution of this dissertation.

CHAPTER 4

Computationally Efficient Relational Reinforcement Learning

I introduced a prototype architecture that implements Hierarchical Tile Codings (HTCs) in chapter 2. Identifying a computational efficiency limitation of that architecture, I hypothesized and confirmed that I could do better with an architecture built from the ground up using a *k*-dimensional trie (*k*-d trie) implementation of HTCs in Chapter 3. Moving on from non-relational Reinforcement Learning (RL), my next goal was to bring performant HTCs to Relational Reinforcement Learning (RRL). *k*-d tries are less suitable for RRL than they are for propositional (or deictic) representations, leaving open a problem of finding a more suitable algorithm. I will explain the problems with attempting to use *k*-d tries for RRL in Section 4.1 and introduce its replacement in Section 4.2.

4.1 Limitations of Carli for Propositional Representations

Let us again consider Blocks World (Section 1.3.1). I introduced four objectives: **`stack`**, **`unstack`**, **`on(a,b)`**, and **`exact`** (depicted in figures 1.4b-1.4e). **`Stack`**, **`unstack`**, and **`on(a,b)`** were explored in related work [Džeroski *et al.*, 2001; Irodova and Sloan, 2005] but I focused in on **`exact`**, a task that would have been more difficult for Džeroski *et al.* and impossible for Irodova and Sloan.

I could not see a way to allow agents implemented using my existing architecture, Carli for Propositional Representations (Carli-Prop), to solve the **`exact`** objective with variable goal configurations. Owing to my original prototype implementation and its lack of support for variables, each action definition included block labels and was associated with its own HTC. This prevented the implementation of RRL (Section 1.3.5) and the generalization that can be achieved through the use of variables. If I wanted to be able to handle variable numbers of blocks, different goal configurations from episode to episode, and have any chance of taking advantage of the isomorphisms that relational representations can capture, I needed to combine my HTCs for different actions and

eliminate my dependence on fixed block labels in my encoding. This would have required challenging alterations to my prototype architecture, but it may have been impossible to proceed with an implementation using a *k*-d trie.

My *k*-d trie-based value function relied on tests for each edge of the *k*-d trie. (Alternatively, each node would determine which edge passes, assuming the tiles properly partition the state space, and the value function lookup would continue down that trajectory through the *k*-d trie.) What allowed these tests to be efficient (complexity linear in the number of remaining features) was the ability to do comparisons of relations involving fixed constants. No actual variables were present in either the set of features being used as the index into the *k*-d trie or in the tests themselves.

The introduction of variables results in two immediate problems for a *k*-d trie. First, it is possible for a test to be multiply passed. For example, `clear(a)` may be true regardless of the choice of action and test true for each candidate action. Second, it is possible for multiple different tests to pass simultaneously. For example, `higher-than(block,dest)` and `not-higher-than(block,dest)` will both test true for different choices of action. Both of these results stem from the ability to match different constants with different variables. Some tests will pass for multiple different actions and different actions will pass different tests.

You might ask, "why not simply process the same *k*-d trie separately for each action?" Unfortunately, this resolves neither problem unless the only tests that result in these issues directly test the features that distinguish one action from another. This leaves us with:

Research Question 3 (Efficient Algorithm for RRL): *What algorithm will give an agent the kind of efficiency provided by k-dimensional tries (k-d tries) but support Relational Reinforcement Learning (RRL)?*

A value function for RRL must support:

1. Features that include variables. Such features enable agents to concern themselves with relations between objects without encoding specific labels or identities of objects. For example, an agent working in Blocks World may need to test that the block it is moving is above one of the two blocks in the goal configuration when solving **`on(a,b)`**, but it need not concern itself with whether the block has the letter "B" on it. This aspect of First Order Logic (FOL) allows learning without encoding unnecessary details and can result in more effective generalization. Without variables, an agent is limited to propositionalized features.

2. Comparisons across features involving variables such as equality, inequality, less than, ... It is these comparisons that allow an agent to build up combinations of relational features, which in turn allows an agent to solve problems without necessarily encoding labels that limit or eliminate generality, thereby reducing Average Return Per Episode (ARtPE).

3. Variable and unpredictable numbers of actions and observations about the world. This necessitates that actions share a value function. In Chapter 3, my agents for Puddle World had separate value functions for each cardinal direction. With actions defined relationally through the use of variables, it is unclear how to allocate separate value functions for different actions. Treating all actions of a given type identically regardless of variables would be unwise under many circumstances. e.g. `move(north)` should probably not be treated identically to `move(south)`. Giving all actions of a given type completely separate value functions results in an absolute absence of generalization between actions, which is also undesirable. e.g. Perhaps `move(north)` and `move(south)` are equivalent under some circumstances, such as when moving deeper into a puddle. Value function sharing in such cases is desirable to achieve greater generality and increased ARtPE.

4. Computationally efficient refinement and unrefinement, including support for fringe-based decision-making about refinements. This adaptation of the value function goes beyond mere partitioning of the state space in the context of RRL. A conjunction of relational features can form the basis of the inclusion of an important, implicit concept in the value function. For example, once an agent is aware that `block-0`, the block being moved, is the lower block in the goal of **`on(a,b)`**, awareness of whether the destination block is above the upper block in the goal or not becomes very important. An agent can then learn that placing the lower block above the upper block is a costly move to make.

Thinking back to my prototype architecture, which used Rete internally, the only issue for supporting RRL from a functionality standpoint was that my implementation of value function refinement ultimately propositionalized my representation. Thus I speculated:

Hypothesis 4 (Rete for RRL): *Rete can be used as an efficient algorithm for Relational Reinforcement Learning (RRL) given sufficient architectural support.*

Next I shall explain how Rete can accomplish this.

4.2 Rete

There are important, novel aspects of my Rete implementation, but the Rete algorithm [Forgy, 1979] is certainly not a novel contribution of this thesis in and of itself. In this section, I take care to explain not only the aspects of my Rete that are novel but to explain the Rete as a whole. I do this since limiting the understandability of this thesis to a reader in the RL community who already has even a passing familiarity with the Rete would greatly restrict its target audience. The novel aspects of this Rete implementation that are critical for achieving an embedding of an adaptive Hierarchical

Tile Coding (aHTC) in a Rete (which I will describe how to do in Section 4.3) are `Existential Join Nodes` (Section 4.2.3.5) and that this Rete implementation supports analysis and synthesis of rules at the node level. If you are a Rete expert, you may wish to skim Sections 4.2.1 through 4.2.3.4 and 4.2.3.6 through 4.2.3.7.

The Rete algorithm comes from the rule-based systems community. It is designed to solve the problem of how to efficiently match a large number of rules that may need to be executed at any given time. There are several features of Rete that are appealing from either an efficiency or a functionality perspective:

1. Rete uses memory to cache intermediate results to save computation time by processing only changes from step to step.

2. It shares computational effort between rules that share conditions.

3. And most importantly, it is designed to handle multiple matches for different sets of variables for any given condition, exactly solving the problem that it was unclear how to resolve when using k-d tries.

Figures 4.1 and 4.2a present two different abstract depictions of value function representations for Puddle World – one in Carli-Prop and one in Carli for Relational Reinforcement Learning (Carli-RRL). In both cases, each black circle represents one feature conjunction, one corresponding tile in a HTC, and the weight associated with that tile. Observe that the differences occur primarily where new variables are introduced into the value function. Figure 4.2b depicts a more complex value function representation for the **`exact`** objective in Blocks World.

So what is different about Rete when compared to a k-d trie? First, it is a Directed Acyclic Graph (DAG) rather than a tree. That is not to say that working memory cannot contain cycles. More precisely, the Rete implementation that processes working memory is a DAG. That the Rete is a DAG means that nodes can have multiple paths in and multiple paths out, rather than being

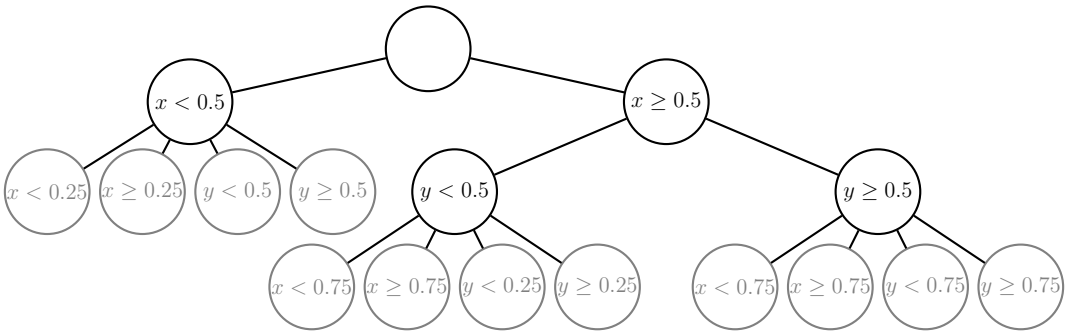
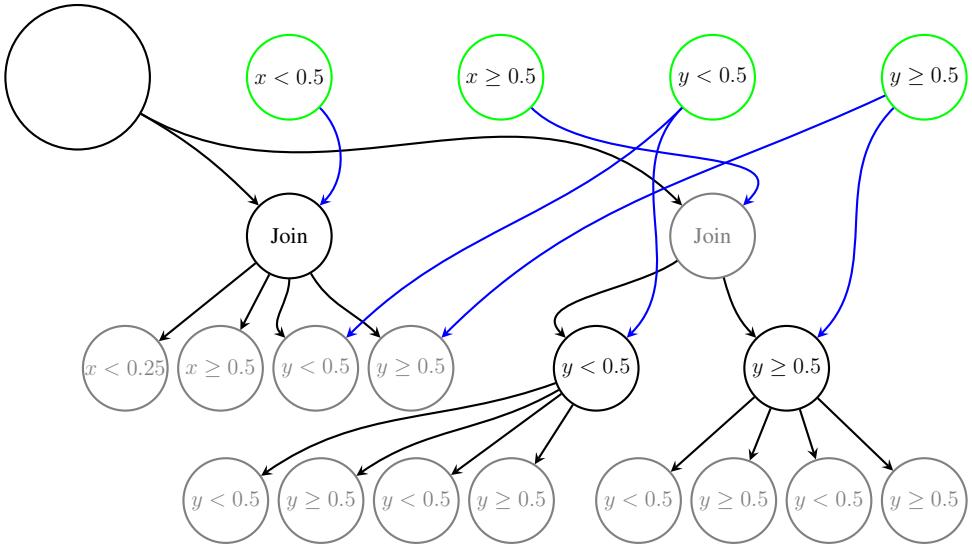
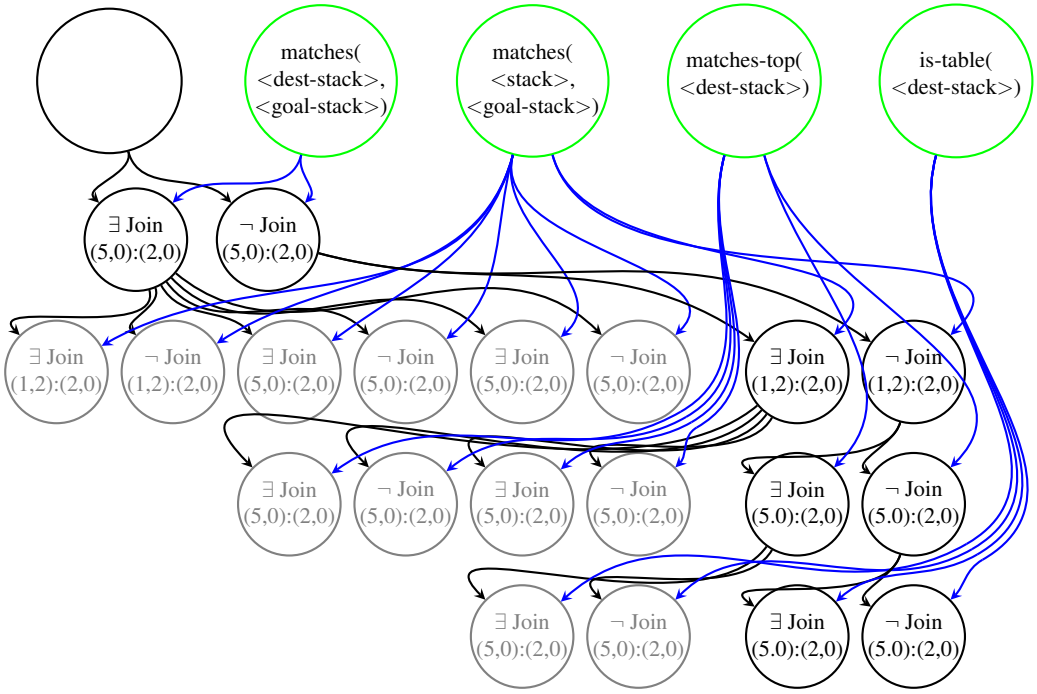


Figure 4.1: An RL trie value function representation for Puddle World, with weights that contribute to the value function in black and weights in the fringe in gray



(a) A Rete value function representation for Puddle World



(b) An RRL Rete value function representation for the **`exact`** objective of Blocks World

Figure 4.2: Rete value function representations with filter nodes at the top, left inputs in black, and right inputs in blue, with weights that contribute to the value function in black and weights in the fringe in gray

constrained to have only one path in but potentially multiple paths out. In the case of Rete, each kind of node is restricted to one or two edges in.[1]

Second, nodes in a Rete do not simply implement tests that are passed or failed. The Rete algorithm implements a token passing network, allowing it to handle variable and unpredictable numbers of actions and observations about the world. A distinction is sometimes made between an alpha network consisting of alpha nodes and a beta network consisting of beta nodes [Forgy and McDermott, 1977b; Doorenbos, 1995]. Alpha nodes (the top rows in Figure 4.2) can be viewed as implementing tests, more or less like nodes in a decision tree or *k*-d trie, but with support for variables. Beta nodes (all the nodes below the top rows in Figure 4.2) on the other hand must be designed to take in one or two tokens and to decide when to combine them (in the case that there are two), in some cases doing comparisons involving variables, and to pass on the resultant combined token. Alpha nodes feed forward into beta nodes but beta nodes never feed forward into alpha nodes due to the increased expressive power required by the tokens in the beta network. That these nodes support variables for generalization in the alpha network and variable comparisons as constraints in the beta network gives Rete the power needed to support RRL.

Finally, in order to do comparisons involving variables, certain beta nodes are designed to solve the variable binding problem that is introduced by supporting conjunctions of relations between variables. Observe that the beta nodes in Figure 4.2 do not generally contain the tests themselves, but rather contain variable bindings in the form of indices into the left and right tokens. (Tests on variables that have already been incorporated into the value function are included directly in the beta nodes for Puddle World in Figure 4.2a.) If a rule tests both the conditions `(<block-1> ^on-top <block-2>)` and `(<block-2> ^on-top <block-3>)`, care must be taken to ensure that the variable `<block-2>` refers to the same constant in all tokens that make it all the way to the leaf node that fires the rule. What happens when a rule is fired in Carli-RRL will be explained in Section 4.2.3.7. Broadly speaking, the value function for a given action is derived from the sum of the weights, top to bottom, for the nodes that fire for that action, using equation 1.7 on page 18.

4.2.1 Carli-RRL Representations and Syntax

I base the structure of my rules on that of the Soar cognitive architecture [Laird, 2012]. Soar defines a rule syntax that enables agent designers to implement different agents without directly tinkering with the underlying architecture. In fact, it encapsulates the underlying architecture to ensure that it remains fixed while the rules and memories are allowed to differ from one agent to the next. This

[1] Two edges in is effectively sufficient to implement anything, as more than two edges in could be simulated by the implementation of a node that implements an OR operation.

divide between architecture and rules can be traced back to Ops [Forgy and McDermott, 1977a]. Like Ops, Soar was originally implemented in Lisp and owes much of its syntax to its predecessor [Laird and Newell, 1983]. While Carli-RRL was never implemented in Lisp, given this history, it made sense to use a similar syntax for the rules implementing my weights to that used by Soar for its RL-rules [Nason and Laird, 2004]. The syntax does not represent any theoretical commitment, however.

The most basic memory elements of my working memory – the agent's representation of the state of the environment – are `Symbol`s which can represent constant values (`Float`ing point numbers, `Int`egers, and `String`s), `Identifier`s, and `Variable`s. `Float`s and `Int`s can be compared to each other, but otherwise any comparisons between different types of `Symbol`s will test false. These `Symbol`s are combined into ordered triples or 3-tuples which I refer to as state triples. Each `Symbol` in a state triple can be any of the above types, but only the constant values are universally appropriate. The set of state triples that make up the agent's state description can include `Identifier`s but must not include `Variable`s. On the other hand, the state triples that form the rules and the Rete should include `Variable`s but must not include `Identifier`s. For that reason, the only way for a rule to refer to an `Identifier` is with a `Variable`. By convention, an agent's state description is organized hierarchically by placing new object `Identifier`s on the right and by placing existing object `Identifier`s on the left, but any state triples without `Variable`s could constitute a legal state description. See Figure 4.3 for an example of this hierarchical memory organization.

Figure 4.4a presents a rule to associate a weight with a tile with one feature. `sp` stands for "source production."[2] The braces enclose the production, which follows the format: *rule-name conditions* --> = *value*.[3] The *value* at the end represents the initial value of the weight associated with the tile that is associated with the rule. Comparing against working memory (Figure 4.3), you can see that `Identifier`s are absent. `Variable`s are in the place of all `Identifier`s and some constants as well.[4] Each condition is capable of matching one or more of the Working Memory Elements (WMEs) represented in Figure 4.3.

[2]Carli-RRL supports a small number of other commands.

[3]The --> separates conditions from actions in Soar and can be traced back to Ops. Since there are no actual action in Carli-RRL, it is included only to maintain similarity of form with Soar RL-rules.

[4]You might observe that I put `Identifier`s in all upper case letter and use all lower case letters for all other `Symbol`s. This is purely convention on my part. `Identifier`s are never exposed to users of Carli-RRL in typical operation.

```
(S1 ^action move-2-0)
(S1 ^action move-2-3)
(S1 ^action move-3-0)
(S1 ^action move-3-2)
(S1 ^block A)
(S1 ^block B)
(S1 ^block C)
(S1 ^block TABLE)
(move-2-0 ^block B)
(move-2-0 ^dest TABLE)
(move-2-3 ^block B)
(move-2-3 ^dest C)
(move-3-0 ^block C)
(move-3-0 ^dest TABLE)
(move-3-2 ^block C)
(move-3-2 ^dest B)
(A ^name 1)
(B ^name 2)
(C ^name 3)
(TABLE ^in-place true)
(TABLE ^name 0)
```

Figure 4.3: A minimal Carli-RRL description of an initial state of the version of Blocks World from Section 2.2.1 and the actions possible from that state (move block B to the table, block B to block C, . . .)

```
sp {bw*dest-in-place
   (<s> ^action <action>)
   (<action> ^block <block>)
   (<action> ^dest <dest>)
   (<block> ^name <block-name>)
   (<dest> ^name <dest-name>)
  +(<dest> ^in-place true)
-->
   = 0.0
}
```

(a) Destination in place

```
sp {bw*dest-not-in-place
   (<s> ^action <action>)
   (<action> ^block <block>)
   (<action> ^dest <dest>)
   (<block> ^name <block-name>)
   (<dest> ^name <dest-name>)
  -(<dest> ^in-place true)
-->
   = 0.0
}
```

(b) Destination out of place

Figure 4.4: Paired Carli-RRL features for the version of Blocks World from Section 2.2.1

For example, `(<s> ^action <action>)` matches `(S1 ^action move-2-3)`, `(S1 ^action move-3-0)`, and `(S1 ^action move-3-2)`. The rule as a whole, however, only matches for `(S1 ^action move-3-0)` since the only destination that is in the correct place is the table. The counterpart rule in Figure 4.4b is associated with the other tile and matches both `(S1 ^action move-2-3)` and `(S1 ^action move-3-2)`. Together both rules tile the state space.

When these rules are fired, the `<block-name>` and `<dest-name>` can be extracted from the final token by the Carli-RRL agent to determine which action the corresponding weight should be added to. When all rules have finished firing, the list of weights associated with each action are summed to give a value estimate for linear function approximation (Section 1.3.3), just as the path through the *k*-d trie provided that set of weights in the previous version of this architecture (Section 3.1.1).

Reading the rules in Figure 4.4, it is probably difficult for you as the reader not to assume that the `<action>` in the first condition is the same as the `<action>` in the second and third conditions. However, ensuring that is a non-trivial feat of the Rete algorithm. The beta nodes are the Rete's solution to this problem.

4.2.2 Alpha Nodes

The purpose of the alpha network is to ensure that individual conditions match exactly the WMEs that they should.

Doorenbos [1995] implements an alpha network that has structural complexity comparable to that of its beta network (Section 4.2.3). It is possible to implement alpha nodes that test aspects of individual `Symbol`s in each incoming WME before ultimately generating a `Token` for the beta network to achieve greater computational efficiency within the alpha network.

However, my architecture implements a simplified alpha network where each unique condition directly corresponds to a unique alpha node. For my use cases, passing all WMEs to each of these alpha nodes is such a small part of the computational cost of my architecture that anything more sophisticated is unnecessary. However, for a significantly larger architecture with many unique conditions and a large working memory, this would need to be addressed.

4.2.2.1 Filter Node

I implement a single `Filter Node` to fully test a WME. One `Filter Node` must exist per unique condition in the Rete. Each `Filter Node` passes new `Token`s corresponding to incoming WMEs on to the beta network.

A `Filter Node` in my architecture has the following functionality:

1. Symbols in the incoming WME that are being tested against a variable that appears only once in the condition are not subjected to any tests.

   i.e. A Filter consisting of three non-repeated Variables, such as (<a> ^<b> <c>) will match every WME.

2. Symbols that are tested against variables that are repeated within the condition are compared against each other for equality. If any of them are not equal, they do not pass the Filter and no Token will be generated.

   i.e. (<a> ^<b> <a>) requires that the first and third Symbols in the WME test equal to each other. They must be the same type of Symbol unless one is a Float and the other an Int.

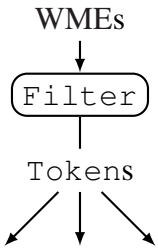
3. Symbols that are tested against constants must be equal to those constants or they do not pass and no Token will be generated.

   i.e. (<a> ^height 3) requires that the second Symbol be a String containing "height" and that the third Symbol be either a Float or an Int containing 3 as its value.

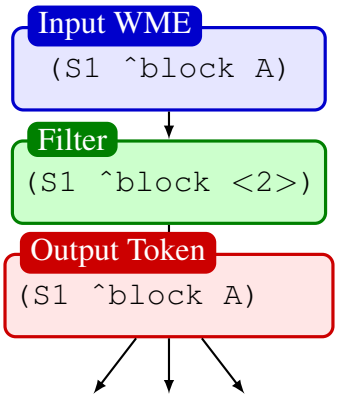
This results in an $O(mn)$ computation time for testing all WMEs (m) against all Filter Nodes (n).[5] This does not appear to be a significant cost for any workloads in my architecture, but you can imagine how this could be sped up by doing a hash lookup by Symbol type for the first Symbol, proceeding to the second Symbol in a successor Node if the first passes, and again for the third. This can get complexity down to something on the order of $O(m)$ for testing all WMEs. However, given the limited number of unique conditions in my agents and the significantly higher cost of executing the beta network, the complexity of this approach did not merit its implementation.

Figure 4.5 presents a Filter Node diagram and both a positive example and a negative example. Observe that each Filter Node will examine each input WME as walked through in the examples. In a more complex case, it could be necessary to verify whether two objects in the WME are one and the same. The Filter would then be represented as (<0> ^attr <0>) with the wildcard <0> in two different places.

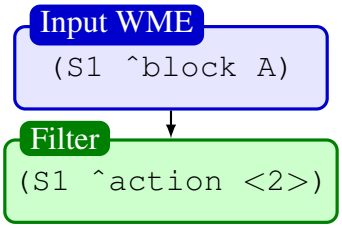
[5]Only changes are processed from step to step, so this cost is amortized over the duration of an agent's lifetime.



(a) A `Filter Node` diagram



(b) Positive `Filter` Example: "`A`" matches the wildcard <2> (associated with some `Variable`) in the `Filter Node`, so a new `Token` containing the WME is passed to all children.



(c) Negative `Filter` Example: "`block`" does not match "`action`" in the `Filter Node`, so no `Token` is passed.

Figure 4.5: `Filter Node` Functioning

4.2.3 Beta Nodes

The alpha network is restricted to testing individual WMEs. To detect whether all of the conditions of a rule match, rather than just one, is the purpose of the beta network. In other words, the beta network is responsible for everything that cannot be tested at the level of individual conditions. One important responsibility of the beta nodes is ensure that variables repeated between different conditions match one another. Additionally, tests of individual variables can exist at any point in a rule after the first condition.

4.2.3.1 Predicate Node

A `Predicate` is a single-input `Node` that implements one of six tests:

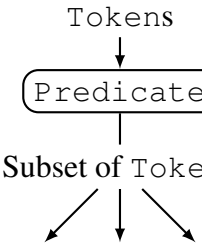
1. Equality ==
2. Inequality !=
3. Greater than >
4. Greater than or equal to >=
5. Less than <
6. Less than or equal to <=

The test can compare a `Variable` on the left side of the `Predicate` to either a `Variable` or a constant on the right side of the `Predicate`. All `Variables` must have appeared in preceding conditions – be included in the existing `Token` – but the relative order of appearance is irrelevant and both `Variables` are likely to point to different rows of the `Token`.

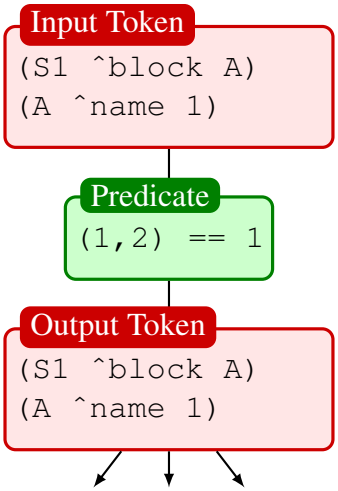
`Predicate Nodes` can be implemented as part of both the alpha network and the beta network. Carli-RRL implements them in the beta network only since the only time they can be applied in the alpha network is when they test variables that were introduced in the preceding condition and it would force my code to handle WMEs that pass the `Filter Nodes` differently in the case that they are followed by alpha `Predicate Nodes` for marginal performance gains.

Figure 4.6 presents a `Predicate Node` diagram and both a positive example and a negative example. Observe that each `Predicate Node` can implement only one test, but that they can be chained together to implement multiple tests. It is also possible to test two `Variables` against one another, in which case a `Predicate` test might be represented as `(0,2) == (1,2)`, where the first value in the pair represents the row in the `Token` and the second value represents whether it is the 0^{th}, 1^{st}, or 2^{nd} value in the row.[6]

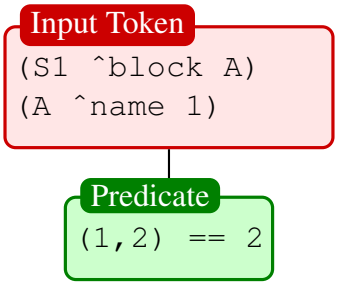
[6]Note that the first value represents the row in the `Token` and not the output of a particular `Filter Node`. Multiple rows in the `Token` can correspond to output from the same `Filter Node` with different `Variables` matching in different parts of the rule.



(a) A `Predicate Node` diagram



(b) Positive `Predicate` Example: "1" (stored at index `(1,2)` in the input `Token`) is equal to "1" in the `Predicate Node`, so the `Token` is passed to all children.



(c) Negative `Predicate` Example: "1" (stored at index `(1,2)` in the input `Token`) is not equal to "2" in the `Predicate Node`, so no `Token` is passed.

Figure 4.6: `Predicate Node` Functioning

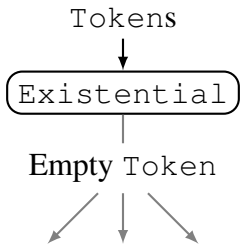


Figure 4.7: An Existential Node

4.2.3.2 Existential Node

An agent might care that certain conditions are satisfied, but not be concerned with either how they are satisfied or how many times they are satisfied. The ability to verify that they are satisfied without doing a Join (or by doing a Join with an empty Token) both reduces the number of Tokens passing through the Rete due to multiple matches and prevents thrashing due to changes in which WMEs satisfy the existential conditions.

Support for Existentials is one of the features of the Carli-RRL Rete implementation that distinguishes it from the Soar Rete implementation. An Existential is a single-input Node that passes on an empty Token – a Token that has 0 rows and which cannot be indexed into – in the case that it receives one or more input Tokens. Empty Tokens can be joined with other Tokens to generate output that is more meaningful than whether a rule fires or not.

An Existential can apply to one or more conditions. In the case that it applies to more than one condition, they must be combined into a single scope (i.e. enclosed in braces): $+\{(\textit{condition 1})(\textit{condition 2})\}$ as opposed to simply $+(\textit{condition 1})$. Furthermore, an Existential must be first in any given scope, or it is technically an Existential_Join which will be described in Section 4.2.3.5. Support for first scope Existentials is included primarily for the sake of grammatical completeness, but Existential_Join Nodes will be very important for the the implementation of HTCs in Section 4.3.

4.2.3.3 Negation Node

The ability to test conditions but not Negations potentially leaves open gaps in an agent's ability to reason. It is desirable for an agent to be able to respond in situations in which certain conditions do not match at all. The ability to verify that they are not satisfied allows an agent to do logic that hinges on the absence of certain WMEs or of certain combinations of WMEs.

Negations do the opposite of what Existentials do. A Negation is a single-input Node that passes on an empty Token in the case that it receives no input Tokens.

A Negation can apply to one or more conditions. In the case that it applies to more than one

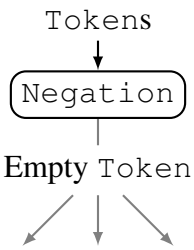


Figure 4.8: A Negation Node

condition, they must be combined into a single scope (i.e. enclosed in braces): $+\{(\textit{condition 1})(\textit{condition 2})\}$ as opposed to simply $+(\textit{condition 1})$. Furthermore, a Negation must be first in any given scope, or it is technically an Negation_Join which will be described in Section 4.2.3.6.

4.2.3.4 Join Node

The central purpose of the beta network is to evaluate which combinations of WMEs that satisfy individual conditions are compatible with one another and to combine them together into a Token to be passed to an Action. Join nodes are necessary to do tests across multiple conditions and to combine Tokens together to form larger Tokens.

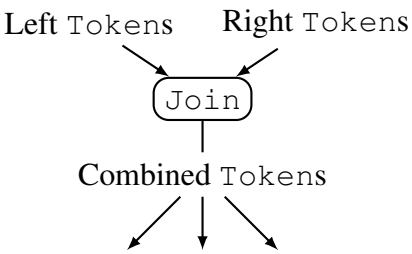
A Join is a two-input Node that passes on combined Tokens in the case that the Variables on the left match the corresponding Variables on the right. Join nodes implement the Rete's solution to the Variable binding problem across multiple conditions. Due to the use of hashing as an optimization, Floats and Ints are not comparable across Joins.[7]

Joins are necessary wherever a condition or scope follows another condition or scope. A Join between the first two conditions in a scope will Join the Tokens from the first condition on the left with the Tokens from the second condition on the right. For each condition that follows, a Join will take Tokens from the preceding Join on the left and Tokens from the new condition on the right.

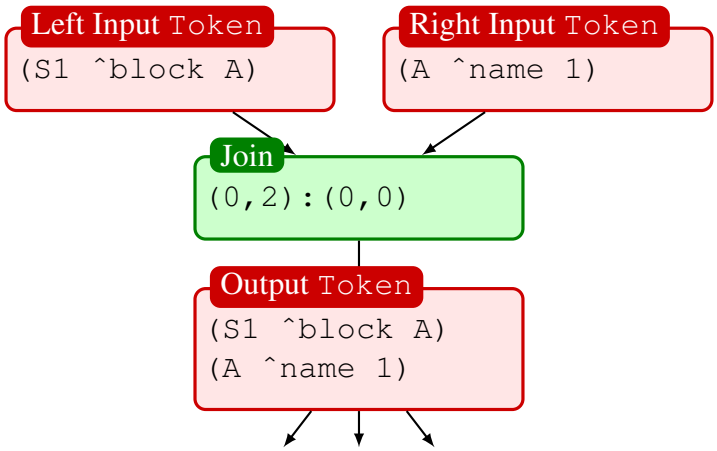
Figure 4.9 presents a Join Node diagram and both a positive example and a negative example. Observe that each Join Node can combine only two Tokens, but that they can be chained together to combine many Tokens together. It common to include multiple bindings in a Join Node or none at all.

A Join can result in a mn nodes in the event that all Tokens on the left match all Tokens on the right. Repeated Joins can result in a combinatorial explosion as a result, hurting computational performance and using a great deal of memory. It is sometimes necessary to be careful in

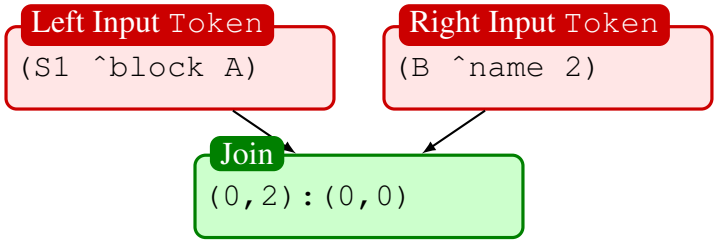
[7]This could be rectified by using the Int hash function on any Floats that could be stored in Ints, but this is unnecessary for any of my use cases.



(a) A `Join Node` diagram



(b) Positive `Join` Example: "A" (stored at index `(0,2)` in the left input `Token`) matches "A" (stored at index `(0,0)` in the right input `Token`) as demanded by the binding `(0,2):(0,0)` in the `Join Node`, so a combined `Token` is passed to all children.



(c) Negative `Join` Example: "A" (stored at index `(0,2)` in the left input `Token`) does not match "B" (stored at index `(0,0)` in the right input `Token`) as demanded by the binding `(0,2):(0,0)` in the `Join Node`, so no `Token` is passed.

Figure 4.9: `Join Node` Functioning

designing rules and corresponding working memory structures to avoid such explosions.

4.2.3.5 Existential Join Node

`Existential Node`s output empty `Token`s. As a result it is impossible to do any comparisons between the conditions within an `Existential` and those outside the `Existential`. The `Existential_Join` allows an agent designer to have existential conditions that depend on values in conditions outside the existential by combining functionality from `Existential` and `Join` into one `Node`.

An `Existential_Join` is a two-input `Node` that passes on `Token`s from the left in the case that the `Variable`s on the left match the corresponding `Variable`s on the right for at least one `Token` from the right. `Existential_Join`s, like `Existential`s, are a feature of the Carli-RRL Rete implementation and not of the Soar Rete implementation. I developed them specifically for the purpose of enabling the embedding of HTCs in a Rete.

An `Existential_Join` can apply to arbitrarily many conditions on both the left and the right. In the case that it applies to more than one condition, they must be combined into a single scope (i.e. enclosed in braces): $+\{(\textit{condition 1})(\textit{condition 2})\}$ as opposed to simply $+(\textit{condition 1})$. Furthermore, an `Existential_Join` cannot be first in any given scope, or it is technically an `Existential` which was described in Section 4.2.3.2.

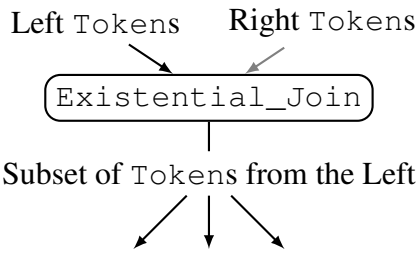
Figure 4.10 presents an `Existential_Join Node` diagram and both a positive example and a negative example. Each `Existential_Join` will pass the left `Token` exactly once for the totally of all matches on the right.

4.2.3.6 Negation Join Node

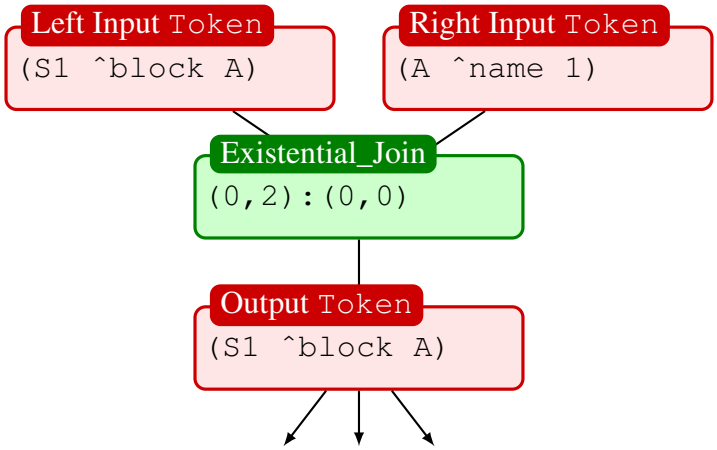
`Negation Node`s output empty `Token`s. As a result it is impossible to do any comparisons between the conditions within an `Negation` and those outside the `Negation`. The `Negation_Join` allows an agent designer to have negated conditions that depend on values in conditions outside the negation by combining functionality from `Negation` and `Join` into one `Node`.

A `Negation_Join` is a two-input `Node` that passes on `Token`s from the left in the case that the `Variable`s on the left do not match the corresponding `Variable`s on the right for even one `Token` from the right.

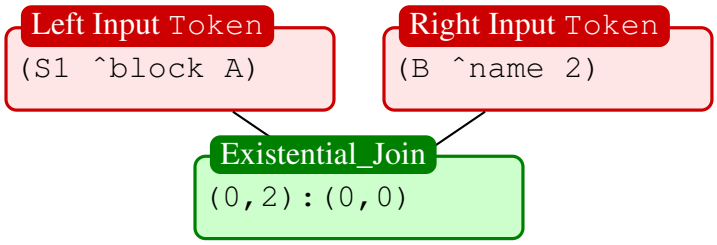
A `Negation_Join` can apply to arbitrarily many conditions on both the left and the right. In the case that it applies to more than one condition, they must be combined into a single scope (i.e. enclosed in braces): $+\{(\textit{condition 1})(\textit{condition 2})\}$ as opposed to simply $+(\textit{condition 1})$. Furthermore, a `Negation_Join` cannot be first in any given scope, or it is technically an `Negation` which was described in Section 4.2.3.3.



(a) An Existential_Join Node diagram

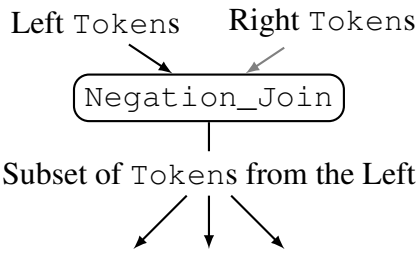


(b) Positive Existential_Join Example: "A" (stored at index (0,2) in the left input Token) matches "A" (stored at index (0,0) in the right input Token) as demanded by the binding (0,2):(0,0) in the Join Node, so if and only if this is the first match for the given left input Token, the left input Token is passed to all children.

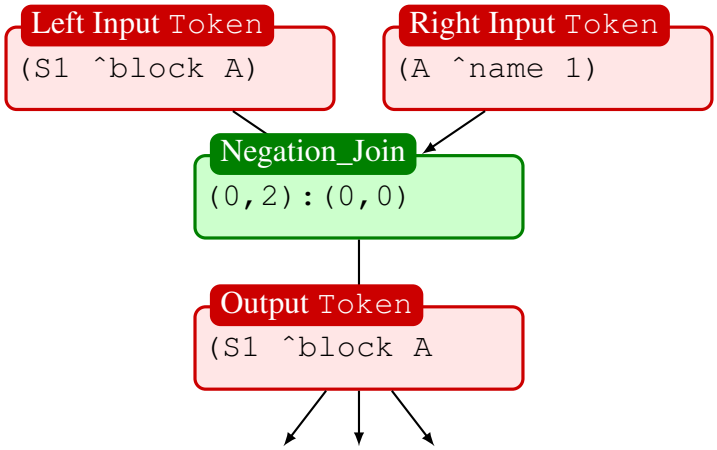


(c) Negative Existential_Join Example: "A" (stored at index (0,2) in the left input Token) does not match "B" (stored at index (0,0) in the right input Token) as demanded by the binding (0,2):(0,0) in the Join Node, so no Token is passed.

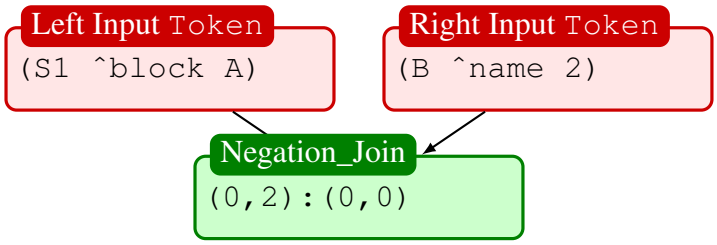
Figure 4.10: Existential_Join Node Functioning



(a) A Negation_Join Node diagram



(b) Positive Negation_Join Example: "A" (stored at index (0,2) in the left input Token) matches "A" (stored at index (0,0) in the right input Token) as demanded by the binding (0,2):(0,0) in the Join Node, so if and only if this is the first match for the given left input Token, the left input Token is retracted from all children.



(c) Negative Negation_Join Example: "A" (stored at index (0,2) in the left input Token) does not match "B" (stored at index (0,0) in the right input Token) as demanded by the binding (0,2):(0,0) in the Join Node, so no Token is retracted.

Figure 4.11: Join Negation_Node Functioning

Figure 4.11 presents an Negation_Join Node diagram and both a positive example and a negative example. To understand the examples included here, it is important to recognize that in the absence of any matching right input Tokens, a left input Token is automatically passed to all children. The positive example in Figure 4.10b demonstrates that what happens when a right input Token matches is the opposite of what happens for an Existential_Join Node. The negative example in Figure 4.10c, however, demonstrates that failure to match has no effect, which is the same as for the other Join Nodes.

4.2.3.7 Action Node

An Action corresponds to full rule. An Action can execute arbitrary code, but only one kind of Action is implemented in Carli-RRL. When it fires for a given Token, several things will occur:

1. It extracts any Variables it needs (i.e. block and destination names in Blocks World, cardinal direction in Puddle World, ...) in order to be able to generate an index for an action in the environment.

2. It adds its weight to the list corresponding to the value estimate for that action for purposes of linear function approximation (Section 1.3.3).

3. It adds its tile to an active set that must be evaluated for refinement from step to step. For a tile that applies to many different actions, this is added to the active set only once.

An Action can fire for any number of different Tokens, but once it has fired, nothing changes until the Token is retracted. Once the retraction of a WME causes a Token to be retracted the Action undoes its work for that Token.

Variable names are actually stored with the Actions and not with the Filter Nodes that correspond to the conditions. This may be counterintuitive, but consider the possibility of two rules sharing the same condition structure but use different Variable names. I want to be able to share work between them, and it turns out that the Variable binding problem can be solved using numeric indices into the Tokens, so Variable names are unnecessary for the internal functioning of the Rete. The Variable names are used only when printing rules and for other textual output to users.

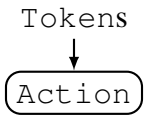


Figure 4.12: An Action Node

4.2.4 Discussion

Recognizing that the Rete algorithm would make a suitable replacement for k-d tries in order to support RRL is a major contribution of this dissertation. A Rete is comparable to a k-d trie in computation efficiency of its implementation, but token passing allows it to handle variable numbers of objects and relations in the environment and corresponding state description.

The Rete algorithm's support for `Join Node`s and variables can be taken advantage of for implementation of relational features and FOL. The `Existential Join Node`s that I introduce as a counterpart to `Negation Join Node`s are essential for the implementation of Boolean relational feature dimensions. Additionally, that the implementation allows node-level analysis and synthesis of rules, as opposed to having to source new rules in full from a file or string buffer, is critical for the enabling of efficient fringe-based value function refinement. I will describe how these aspects of Carli-RRL combine to support the embedding of an aHTC in a Rete in Section 4.3.

4.3 Hierarchical Tile Coding Grammar for Rete

The Rete data structure itself and the `Action` I described in Section 4.2.3.7 do not fully specify an aHTC implementation. Nor does the syntax described in Section 4.2.1 fully specify what an agent needs to know to treat a given Rete as an aHTC.

My architecture additionally requires:

1. A standard rule structure. This structure enables the architecture to both automatically determine what feature conjunction the rule represents and to extract the outermost, most refined, feature for possible value function refinement and fringe expansion.

2. Ordering conditions identically for as long as possible from the start of any scope so that divergence occurs only as new features are introduced. This shared structure allows agents to take maximal advantage of the computational efficiency of the Rete algorithm.

3. The storing of additional metadata in its nodes. This allows the architecture to directly traverse the HTC in order to allow refinement and unrefinement without necessarily referencing the entirety of any given rule of interest. This additionally enables the implementation of my refinement and unrefinement criteria. Ideally it will be possible to write out that data with the Rete and to read it back in alongside the rules.

4. The capability to read in the Rete from a set of rules and to write it back out again. This allows an agent designer to implement more complex agents than is feasible when directly creating a Rete in code. It has the additional benefits of allowing what an agent has learned to be preserved for successor experiments or for offline analysis.

It is possible to manually build the Rete using C++ function calls, but it also provides a parser implemented using using Flex[8], a lexical analyzer, and Bison[9], a tool for implementing Left-to-Right (LR) and Look-Ahead Left-to-Right (LALR) parsers. I used these tools to assist in writing C++ code to load rules corresponding to tiles and features in my HTCs. However, when I discuss the HTC grammar for Rete, the kind of grammar used for parsing rules is not what I mean to discuss. The grammar that defines how aHTCs map onto the Rete is far more important and was part of the design of Carli before I ever chose to implement a rule parser.

Let us return to the root rule for Blocks World (Section 2.2.1), as described in part in Section 4.2.1, and finish making it a complete Carli rule. See Figure 4.13 for the root rule for Blocks World that gives the most general tile for an agent – the one that applies for the entire state-action space. I have now partially made up for skipping the conditions in Section 4.2.3.4. The `:feature` directive, which provides additional metadata (see Section 4.3.2), and my conventions for structuring the conditions of the rule together form a grammar for mapping a `HTC` onto the `Rete` data structure as described in the remainder of this section. This grammar enables one to implement architectural mechanisms to refine the value function independent of both the environment and the relational representation choices made for the agent.

4.3.1 Feature Extraction

Feature extraction is one problem that must be solved in order to treat a Rete as a HTC. The architecture must be able to determine the **value** of the feature as well as the feature **dimension**, where the feature **dimension** groups features that together provide one partitioning of the state space (e.g. all values of x). Given the essentially unlimited expressive power of the conditions and predicates that can make up a rule, this is where convention comes in.

```
sp {bw*general
  :feature 1 unsplit nil
  (<s> ^action <action>)
  (<action> ^block <block>)
  (<action> ^dest <dest>)
  (<block> ^name <block-name>)
  (<dest> ^name <dest-name>)
-->
  = 0.0
}
```

Figure 4.13: The root rule for the version of Blocks World from Section 2.2.1

[8] https://github.com/westes/flex

[9] https://www.gnu.org/software/bison/

```
sp {bw*block-clear
  :feature 2 fringe bw*general
  (<s> ^action <action>)
  (<action> ^block <block>)
  (<action> ^dest <dest>)
  (<block> ^name <block-name>)
  (<dest> ^name <dest-name>)
  +(<block> ^clear true)
-->
  = 0.0
}
```

(a) Long form

```
sp {bw*block-clear
  :feature 2 fringe bw*general
  &bw*general

  +(<block> ^clear true)
-->
  = 0.0
}
```

(b) Terse form

Figure 4.14: An initial fringe rule for the version of Blocks World from Section 2.2.1

The Identical Ordering Convention (IOC) is that the conditions and their ordering must be identical up to the current level of refinement. This means that the most general rule for an agent defines the first conditions and their ordering for all rules in the HTC. Observe that `bw*block-clear` in Figure 4.14a is a direct extension of `bw*general` in Figure 4.13.

Failure to ensure the IOC can be a source of errors, especially when making changes. For example, Carli-RRL depends on the IOC in order to collect and evaluate fringe nodes when considering refinement, and rules with scrambled condition orders would make it impossible for the architecture to correctly refine an aHTC. For that reason, I eventually added a bit of syntactic sugar and developed the syntax in Figure 4.14b. This syntax causes the parser to take the ancestors of `Action`, `bw*general`, in the Rete as its own and to use the same `Variable` names associated with `bw*general` when doing so. This may appear to be redundant with the `:feature` directive once you familiarize yourself with Section 4.3.2, but since it serves a different purpose that may not always be coupled to that directive, it made sense to implement it independently.[10] One important consequence of this ability to refer to another rule when writing one of its descendants is that the rules corresponding to a complex HTC take space linear in the number of tiles in the agent. Without this compression, rule sizes grow longer the more refined the tiles get. This comes at the expense of some clarity when reading rules later, but the option to output the full rules is still available. Reading the rules back into the architecture is more computationally efficient as a result as well.

The Feature Encoded in the Last Scope Convention (FELSC) requires a new feature to be encoded in the last scope of the rule. The root rule is a special case in that it has no feature

[10]Additionally, maintaining these features of the Carli-RRL parser independently allows one to support both syntaxes.

associated with it, but every other tile in the HTC must be distinguished from its ancestors by the addition of a new feature to the preceding feature conjunction. Several different kinds of features are supported:

1. **Boolean Features** – `Existential_Join` and `Negation_Join` `Node`s as last scopes are interpreted as **Boolean features**. Rules such as those in Figure 4.15 are assumed to be part of a pair, one `Existential_Join` to one `Negation_Join`, in which the internals of the last scope are identical. If the assumption holds, these rules are guaranteed to partition the preceding tile.

2. **Multivalued Features** – A `Predicate` Node or a scope ending with a `Predicate` Node that does an equality (==) test of a `Variable` against an `Int` constant is interpreted to be a **multivalued feature**.[11] Rules such as those in Figure 4.16 are assumed to be part of a set of two or more rules that are identical save for the `Int` constant. The set must collectively partition the preceding tile.

3. **Ranged Features** – A `Predicate` Node or a scope ending with a `Predicate` Node that does a less-than (<) or greater-than-or-equal-to (>=) test of a `Variable` against a `Float` or an `Int` constant is interpreted to be a **ranged feature**.[12] Rules such as those in 4.17 are assumed to be part of a pair of two rules that are identical save for the test. If the assumption holds, these rules are guaranteed to partition the preceding tile.

In all of these cases, the feature dimension is defined by the combination of the set of conditions in the last scope (or simply the last condition) and the set of `Variable` bindings that `Join` the last scope or condition to the preceding conditions.[13] The value of a **Boolean feature** is an interpretation of whether it is an `Existential_Join` (true) or a `Negation_Join` (false). The value of a **multivalued feature** is directly extracted from the `Int` constant. And the values of **ranged features** are distinguished on the basis of whether they implement less-than (<) or greater-than-or-equal-to (>=) tests.

[11] That multivalued features must be represented as integers is an arbitrary limitation of Carli-RRL. With a little more complexity in my Feature representation, support for other values would work just fine.

[12] Less-than-or-equal to (<=) and greater-than (>) are supported as well but not used in any of my experiments. I will ignore this functionality to keep my writing simple.

[13] There are no `Variable` bindings for a lone `Predicate` `Node` since the `Variable`(s) in the `Predicate` itself refer to specific `Symbol`s in the `Token`.

```
sp {bw*block-clear
  :feature 2 fringe bw*g
  &bw*g
  +(<block> ^clear true)
-->
  = 0.0
}
```

(a) `<block> ^clear` is true

```
sp {bw*block-not-clear
  :feature 2 fringe bw*g
  &bw*g
  -(<block> ^clear true)
-->
  = 0.0
}
```

(b) `<block> ^clear` is false

Figure 4.15: Paired **Boolean features** for Blocks World from Section 2.2.1

```
sp {pw*move-north
  :feature 2 fringe pw*g
  &pw*g
  (<move> == 0)
-->
  = 0.0
}
```

(a) Move north

```
sp {pw*move-south
  :feature 2 fringe pw*g
  &pw*g
  (<move> == 1)
-->
  = 0.0
}
```

(b) Move south

Figure 4.16: Two **multivalued features** for Puddle World from Section 2.2.2

```
sp {pw*x-lower
  :feature 2 fringe pw*g
           1 0.0 0.5
  &pw*g
  {(<s> ^x <x>)
   (<x> < 0.5)}
-->
  = 0.0
}
```

(a) Lower `x`

```
sp {pw*x-upper
  :feature 2 fringe pw*g
           1 0.5 1.0
  &pw*g
  {(<s> ^x <x>)
   (<x> >= 0.5)}
-->
  = 0.0
}
```

(b) Upper `x`

Figure 4.17: Two **ranged features** for Puddle World from Section 2.2.2

4.3.2 `:feature` Directives

Given that all rules in my Rete are part of a HTC, each comes with a `:feature` directive to provide extra metadata. This section is necessary for reproduction of my work, but not necessary for understanding the contributions of this dissertation.

A `:feature` directive has several parts:

1. `:feature` comes at the beginning. There are no other directives currently, but it is necessary regardless.

2. An integer indicates the depth of the feature in the Rete. This is equal to the number of features in the feature conjunction plus one. So the general tile which has no features and covers the entire state-action space has a value of 1. This could be calculated automatically, so this is a bit superfluous. It is nice to have when printing out terse rules, however, since they make it impossible to tell how complex the feature conjunction has become without tracing through the parent rules.

3. The type of node: `split`, `unsplit`, or `fringe`. `Split` nodes keep track of their parents, which must be `split` nodes, and their children, which must be either `split` or `unsplit` nodes. `Unsplit` nodes keep track of their parents and their children, which must be `fringe` nodes. All three node types collect metadata since I consider the possibility of rerefinement (unrefinement followed by refinement) of the value function.

4. The name of the parent. This must be "nil" for the root node, making "nil" a reserved name that cannot be used for a rule that is part of a HTC. This allows implementation of the pointers between nodes that I just described and allows for error checking to ensure a valid hierarchy. This requires rules to be loaded in increasing order of refinement.

*5-7 – For **ranged features** only since it is possible to subdivide them to generate more:*

5. An integer indicating the depth of the ranged feature. This must be 1 for the original features. Subsequently derived features that cover smaller parts of the range will increment this value.

6. An integer or a decimal number indicating the lower bound of the range covered by the feature. The value used by the predicate should match the lower bound if it is a $>=$ test.

7. An integer or a decimal number indicating the upper bound of the range covered by the feature. The value used by the predicate should match the upper bound if it is a $<$ test. The lower bound and upper bound must both be integers or both be decimal numbers. When generating new features, they will be accordingly integer locked or not as a result.

4.3.3 Refinement and Rerefinement

Part of the purpose of the grammar is to enable automatic feature extraction from rules (Section 4.3.1). However, what I described would be overengineered if that were the sole purpose of the HTC grammar for Rete. This section describes aspects of HTC encoding for the Rete that are critical for implementation of aHTCs using a Rete. In conjunction with a Rete implementation that enables efficient rule analysis and synthesis at the node level, the two conventions I describe here make efficient refinement of an aHTC possible. This is a major contribution of this dissertation.

A Rete typically has efficient support for removing rules and adding new ones dynamically at runtime. However, my Rete needs efficient methods for manipulating existing rules and parts thereof. If I have an `unsplit` tile that is 10 feature conjunctions deep with a dozen `fringe` nodes attached to it, it would be fairly inefficient to send a couple of dozen large, new rules through the standard parser processing pipeline. And as inefficient as that would be, even that still requires one to be able to get features out of arbitrarily complex rules and to know how to form new rules using them.

The **Identical Ordering Convention (IOC)**, the **Feature Encoded in the Last Scope Convention (FELSC)**, and `:feature` directives combine to give an agent knowledge sufficient for efficient refinement of any `unsplit` tile and efficient rerefinement of any `split` tile. In fact, the cost of refinement depends solely on the combination of the number of nodes being converted from `fringe` to `unsplit` and the number of `fringe` nodes being copied under the new `unsplit` nodes. For both refinement and rerefinement, cost is 100% independent of the complexity of whatever lies above the node being modified.

Boolean features are the simplest case. The **FELSC** guarantees not only that the architecture can extract the feature, but in this case that the architecture knows how to take the `Existential_Join` or `Negation_Join` of the last scope and essentially modify what the left input points to in order to move the feature to a different conjunction. Implicitly following the **IOC**, my architecture:

1. Converts the parental `unsplit` node to a `split` node.

2. Converts all `fringe` nodes for the feature dimension selected for refinement to `split` nodes.

3. Takes all remaining **Boolean features** and make copies of their `Existential_Join` or `Negation_Join` nodes modified to have the left input point to the new `split` nodes.

Take a moment to examine Figure 4.2 on page 59 and see how the features there were combined over time to make more complex and deeper conjunctions of relations.

```
sp {pw*s47
  :creation-time 278
  :feature 4 split pw*s5
           2 0.0 0.25
  &pw*s5
  (<x> < 0.25)
-->
  = 4.3 1336.1 1.5 0.0 890
}
```

(a) Lower x

```
sp {pw*s48
  :creation-time 278
  :feature 4 split pw*s5
           2 0.25 0.5
  &pw*s5
  (<x> >= 0.25)
-->
  = 5.3 5442.4 3.1 0.0 1753
}
```

(b) Upper x

Figure 4.18: Two refined **ranged features** for Puddle World from Section 2.2.2. The additional values to the right of the = and the weight of the tile are contributions to mean and variance calculations, a secondary weight for Greedy-GQ(λ), and an update count.

Multivalued features are a bit more complex. The **FELSC** guarantees either a `Join` or a `Predicate Node` as the parent of the `Action` for tiles corresponding to **multivalued features**. In either case, it is the same principle:

4. Take all remaining **multivalued features** and make copies of their `Join` nodes modified to have the left input point to the new `split` nodes or their `Predicate` nodes modified to have the sole input point to the new `split` nodes.

Ranged features are more complex still. The **FELSC** `Node` possibilities are essentially identical to those of **multivalued features**, but using the `:feature` directive, new nodes must generally be created that represent longer feature conjunctions. Let us modify step 4 and continue:

4. Take all remaining **multivalued features** and **ranged features** and make copies of their `Join` nodes modified to have the left input point to the new `split` nodes or their `Predicate` nodes modified to have the sole input point to the new `split` nodes.

5. If the feature being refined is a **ranged feature**, check the `:feature` directive to determine if and how new subtiles should be generated. If so, add new subtile features by adding simple `Predicate Nodes`, even if the original feature involved a `Join`. See Figure 4.18.

In all of these cases, `Variable` indices for any `Variables` introduced by the feature generally must be modified since intermediate `Nodes` alter the number of `Nodes` on the path to the feature and the size of the `Token` as well. `:feature` directives must be updated to implement Section 4.3.2. And old rules must be excised from the Rete.

Rerefinement involves unrefinement followed by refinement. Unrefinement depends heavily on the **IOC** to gather all features below the node to be converted back into a `unsplit` node. It

depends on both the **FELSC** and the `:feature` directives to determine dependencies between the gathered features and to discard any that were generated by subsequent refinement. Those issues taken care of, converting the gathered nodes to `fringe` nodes below the new `unsplit` node is virtually identical to the process of making new `fringe` nodes during refinement.

If immediate rerefinement is intended, it could be implemented as a more direct path than unrefinement and refinement. This would be more efficient. However, unrefinement was originally considered to be a decision entirely separate from the decision of rerefinement when considering the design of Carli-RRL. The implementation still reflects the possibility of deciding to unrefine without necessarily planning on refining again afterward. One potential motivation for doing so might be if an agent could determine that all refinement below a certain level was unnecessary and that the value function could be compressed with no loss of optimality.

4.4 Discussion

I began this chapter with the observation that the **`exact`** objective of Blocks World with goal configurations that change from episode to episode, possibly even varying the numbers of blocks in the world, was not feasible to pursue with Carli-Prop. It would have been difficult both because Carli-Prop implemented one aHTC per action, limiting its ability to generalize as the numbers of blocks is increased, and because k-d tries do not support the kind of variable tests required to ensure that relational features refer to the same objects where necessary. For these reasons, I outlined requirements for a value function for RRL and observed that the Rete algorithm would be a good fit.

I introduced Carli-RRL and described the representations over which it operates in Section 4.2. I presented the kinds of nodes in the Rete that enable it to process variable numbers of objects, relations, and actions. The existential joins and negation joins are of particular importance for implementing relational features that perform binary tests in my HTCs.

I developed a grammar for mapping aHTCs onto the Rete in Section 4.3. Using a standard rule structure, consistent ordering of features (or clauses) within each rule, and additional metadata associated with each rule, Carli-RRL is able to automatically extract features when reading in the rules and to synthesize new rules at the fringe as the value function is refined. Carli-RRL can start with a value function with a single weight that applies to the entire state space and a fringe of candidate weights corresponding to individual relational features, and build up complex value function structures, potentially far more complex than the one depicted in Figure 4.2 on page 59. Automatic generation of relational value function structures such as these are only possible as a result of the IOC and FELSC that I developed and described in Section 4.3.1. This method for mapping an aHTC onto a Rete and the insight that this mapping is valuable for computationally

efficient implementation of RRL is the primary contribution of this dissertation.

The value function embedded in the Rete ultimately corresponds closely to a First Order Logical Decision Tree (FOLDT) as induced by Top-down Induction of Logical DEcision trees (TILDE) [Džeroski *et al.*, 2001; Blockeel and Raedt, 1998] and learned over using equation 1.7 on page 18. In fact, if treated as an Adaptive Tile Coding (ATC) instead of an HTC, the output of TILDE could be encoded in my Rete implementation. However, Carli-RRL supports incremental refinement and unrefinement of its aHTCs without the use of a model, and without starting from scratch each time. What is novel here is its model-free, fringe-based approach to the online, incremental creation of FOLDTs in RRL, the use of HTCs rather than ATCs, and the use of the Rete algorithm and a grammar imposed upon it to implement RRL more efficiently. I will evaluate the ARtPE and Wall-Clock Time Per Step (WCTPS) of this RRL implementation and compare it to earlier systems in Chapter 5.

CHAPTER 5

Relational Reinforcement Learning Evaluation

Here I present experiments using agents implemented with Carli for Relational Reinforcement Learning (Carli-RRL), my Relational Reinforcement Learning (RRL) architecture implemented using Rete (Chapter 4). The goals of this thesis are to develop a computationally efficient architecture for online, incremental RRL and to additionally investigate the problem of improving learning efficiency through RRL.

To those ends, in Section 5.1 I explore the capabilities of my architecture, demonstrating its flexibility and computational efficiency with respect to a number of additional mechanisms that support variations in adaptive Hierarchical Tile Codings (aHTCs) as I work to improve learning efficiency with the **`exact`** objective with variable goal configurations. I then demonstrate a greater than two orders of magnitude computational efficiency gain for the **`stack`**, **`unstack`**, and **`on(a,b)`** objectives of Blocks World when comparing to the agents implemented by Džeroski *et al.* [2001] and I compare percent optimality between our agents in Section 5.2. I evaluate the scalability of my agents for **`exact`** in terms of both optimality and Wall-Clock Time Per Step (WCTPS) in Section 5.3. I do some additional evaluation of transfer in both Blocks World and a scalable Taxicab environment in Section 5.4.

5.1 Learning Efficiency Experiments

My goal here is to evaluate the flexibility of Carli-RRL as an architecture for developing techniques for improving learning, whether the evaluation metric is Average Regret Per Episode (ARgPE), Average Return Per Episode (ARtPE), or even percent optimality (which we use in comparisons in Section 5.2). Since my primary objective is to evaluate the WCTPS of Carli-RRL, this section demonstrates that it can achieve a small WCTPS for RRL. However, it additionally serves the purpose of demonstrating that Carli-RRL can be used to solve a more difficult Blocks World objective than the ones investigated by Džeroski *et al.* [2001].

For these experiments, I investigate the **`exact`** objective of Blocks World with 3-5 blocks

and variable goal configurations due to its higher complexity relative to the other Blocks World objectives (Section 4.1). My agents in this section are evaluated on an Intel Core i7-3612QM CPU executing at 2.10 GHz. The graphs reflect averages over 20 runs. And the agents uniformly use Epsilon-greedy (ϵ-greedy) exploration (Section 1.1.3) with a fixed value of $\epsilon = 0.1$.

My Blocks World implementation generates instances for Blocks World by placing the blocks in random order. The first block forms a stack of height 1. For each subsequent block placement, existing stacks and a new stack are given an equal likelihood of receiving the new block. Therefore, once there exist 3 stacks, the likelihood of creating a new stack is $1/4$ and each existing stack has a $1/4$ chance of receiving the new block. This method is followed for both the initial configuration and the target configuration. In the unlikely event that the generated initial configuration is identical to the target configuration for **`exact`** (or otherwise satisfies the objective for **`stack`**, **`unstack`**, or **`on(a,b)`** in subsequent sections) this procedure is repeated for the initial configuration until it does not satisfy the objective.

The binary relational features available to my agents for **`exact`** include:

1. `Matches(<src-stack>, <goal-stack>)`
   i.e. The stack a block is to be moved from matches some goal stack
   (up to the height of the stack the block is to be moved from).

2. `Matches(<dest-stack>, <goal-stack>)`
   i.e. The destination stack matches some goal stack
   (up to the height of the destination stack).

3. `Goes-on-top(<moving-block>, <goal-stack>)`
   i.e. The block to be moved goes on top of a goal stack.

4. `Move(<moving-block>, TABLE)`
   i.e. The destination is the table.

These features come from viewing the problem as a visual reasoner might, comparing stacks of blocks in the real world to stacks of blocks in a specified goal configuration. Additional distractor features that I optionally include are:

5. `Height-less-than(<moving-block>, 2)`
   i.e. The height of the block to be moved is less than 2.
   (Subsequently whether the height of the block is less than 1 or 3 in subtiles.)

6. `Height-less-than(<destination-block>, 2)`
   i.e. The height of the destination block is less than 2.
   (Subsequently whether the height of the block is less than 1 or 3 in subtiles.)

7. `Brightness-less-than(<moving-block>, 0.5)`
   i.e. The "brightness" (a number that randomly changes from step to step) of the block to be moved is less than 0.5.
   (Subsequently 0.25 and 0.75 in subtiles, and so on ad infinitum.)

8. `Brightness-less-than(<destination-block>, 0.5)`
   i.e. The "brightness" of the destination block is less than 0.5.
   (Subsequently 0.25 and 0.75 in subtiles, and so on ad infinitum.)

9. `Glowing(<moving-block>)`
   i.e. The block to be moved is glowing
   (equivalent to the brightness check, but not refineable).

10. `Glowing(<destination-block>)`
    i.e. The destination block is glowing.

11. `Name(<moving-block>, A)`, `Name(<moving-block>, B)`,
    `Name(<moving-block>, C)`, ...
    i.e. The name of the block to be moved.

12. `Name(<destination-block>, A)`, `Name(<destination-block>, B)`,
    `Name(<destination-block>, C)`, ...
    i.e. The name of the destination block.

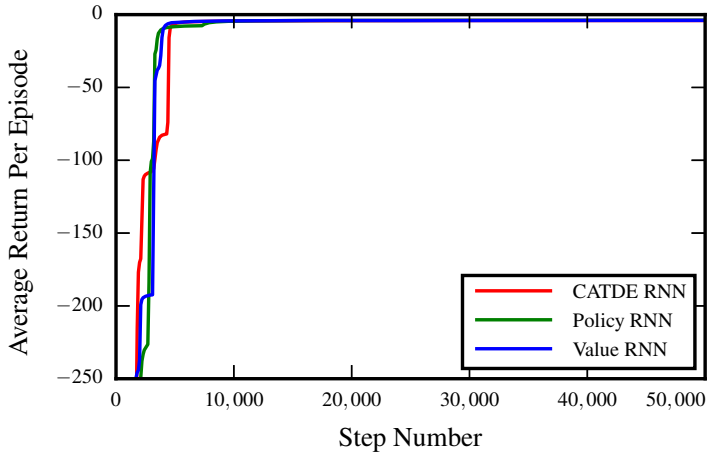
I first introduce several refinement criteria and evaluate my agents using them (Section 5.1.1). I then evaluate how my agents function with unrefinement and rerefinement enabled (Section 5.1.2). Then I will discuss further adjustments in the form of refinement blacklists (Section 5.1.3), a modification to rerefinement selection I call boost (Section 5.1.4), and an optimization of boost that I call concrete (Section 5.1.5).

5.1.1 Refinement Experiments

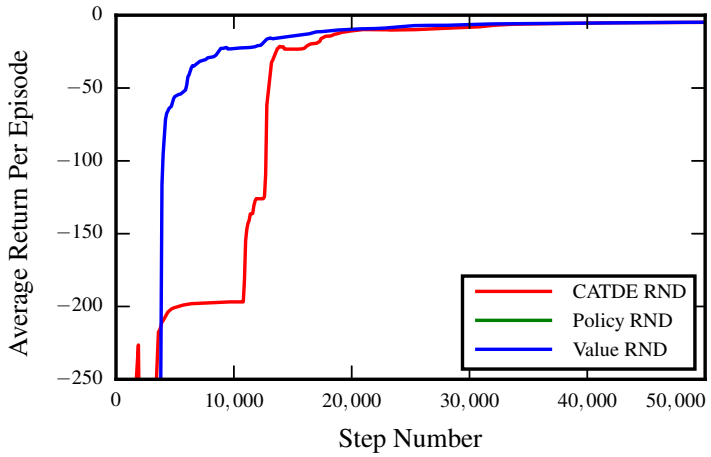
Value function refinement is the method by which Adaptive Tile Codings (ATCs) and aHTCs (which I use here) expand and specialize their value functions in order to learn general knowledge quickly and more specialized knowledge where necessary. Here I evaluate the learning efficiency of my agents with several refinement criteria.

When Carli-RRL refines the value function, it converts one `unsplit Node` to a `split Node`, it converts a small number of `fringe Nodes` to `unsplit Nodes`, and it creates many new `fringe Nodes` for these new `unsplit Nodes`.

The three refinement criteria I explore in Carli-RRL are:



(a) No Distractors



(b) Distractors

Figure 5.1: ARtPE for agents learning the **`exact`** objective in Blocks World. In the legend, "R" indicates that refinement is enabled, "N" indicates that unrefinement is not enabled, and "D" and "N" indicate whether distractors are enabled or not. The policy criterion performs too poorly to appear in the lower graph.

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−4.06	1.38ms	23.0
Policy Criterion	−3.84	0.91ms	24.7
Value Criterion	−3.97	1.33ms	25.6

Table 5.1: Agent performance solving **`exact`** Blocks World (Section 4.1)

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−4.93	18.8ms	1,487.8
Policy Criterion	−639	21.7ms	1,318.4
Value Criterion	−4.83	19.0ms	1,459.5

Table 5.2: Agent performance solving **`exact`** Blocks World (Section 4.1) with distractors

1. **Cumulative Absolute Temporal Difference Error (CATDE)** as described in Section 2.1.3.2 on page 38 – refine a tile if its CATDE is in the top $f\%$

2. A **policy criterion** akin to that in Section 1.3.4.3 on page 23 – refine a tile along the feature dimension that results in the largest change to the set of greedy actions

3. A **value criterion** akin to that in Section 1.3.4.3 on page 23 – refine a tile along the feature dimension with a maximal range of corresponding weights

I selected these criteria because I was able to conceive of methods to implement them incrementally using only summary statistics. That is not to say that other criteria, including ones that require more than summary statistics for implementation (such as U-trees [McCallum, 1996]) or that are more difficult to make incremental (such as **Stdev_Inf** [Munos and Moore, 1999a]), could not be supported by Carli-RRL. They are simply less suitable for our goals of minimizing WCTPS while pursuing high ARtPE. For all of these criteria, I use on-policy Greedy-GQ(λ) with $\alpha = 0.03$, $\rho = 0.01$, $\gamma = 0.9$, and $\lambda = 0.3$ and delay refinement decisions by 20 steps without distractors and by 30 steps with distractors in order to reduce the likelihood of premature refinement. These numbers are the result of a modest amount of parameter tuning on my part.

Figure 5.1 depicts ARtPE when learning the **`exact`** task using these three different refinement criteria for refining the aHTC. I show performance when learning with relevant features only in Figure 5.1a and with additional distractors in Figure 5.1b. Detailed results at $50,000$ steps are listed in tables 5.1 and 5.2. With no distractors, all criteria work well with minimal WCTPS. On the other hand, with distractors, WCTPS increases by more than an order of magnitude, the number of weights (and the number of feature conjunctions) increases by nearly two orders of magnitude, and learning performance suffers as well – significantly in the cases of the policy criterion. The number of weights grows with distractors because there is no criterion for stopping refinement and the agent lacks any ability to undo refinements when they appear to be unhelpful. Given that the "brightness" distractor is infinitely refineable, this is especially costly.

5.1.2 Rerefinement Experiments

My work up until this point has demonstrated results with non-adaptive tile codings, ATCs, and aHTCs. In the case of ATCs and aHTCs, refinement has been an irreversible process. This can result in suboptimal value function structure from the perspective of learning efficiency, and excessive numbers of weights as demonstrated in Table 5.2. From this point on, I explore the possibility of unrefining and rerefining the value function.

With unrefinement disabled, after converting the selected `fringe Nodes` to `unsplit Nodes` and using the remaining `fringe Nodes` as templates from which to build successor `fringe`

`Node`s, one could simply destroy the old `fringe Node`s. The process of refinement is described in Section 4.3.3 on page 81. With the possibility of unrefinement and rerefinement, it is helpful to maintain them as internal `fringe Node`s for two reasons. First, the architecture can use them to collect value estimates and metadata for comparison against those of the refined tiles. Second, it can use them as templates for regenerating the fringe should unrefinement be necessary, significantly simplifying the task of unrefining the value function.

My three unrefinement criteria directly correspond to my refinement criteria:

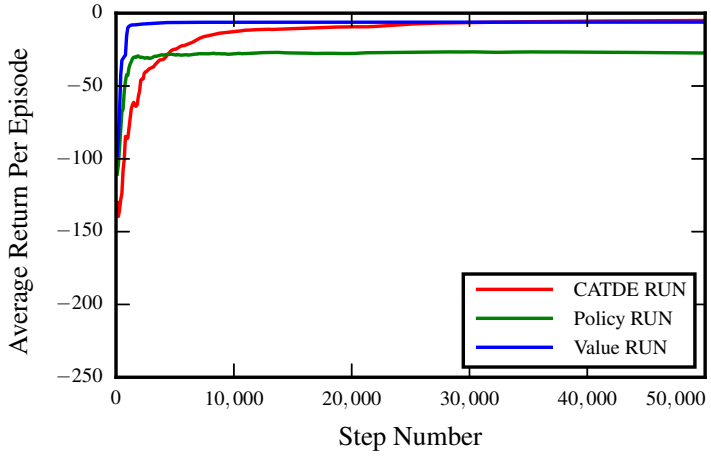
1. **CATDE** – Unrefine if the CATDE for the `split Node` is greater than that of the sum of the CATDEs of its child fringe nodes (which partition the tile). This indicates that its fringe nodes are doing a better job of estimating the value function than the current value function, and therefore the selected refinement was probably suboptimal. This is unlikely to make mistakes, and is likely to be reasonably good from a stability standpoint as unrefinement gets less and less likely the longer a refinement is kept.

2. A **policy criterion** – Unrefine if the change to the set of greedy actions is larger for an alternative feature dimension than for the extension to the value function stemming from the refined feature dimension. One might expect a larger change to the set of greedy actions to indicate a better option, so this could work well.

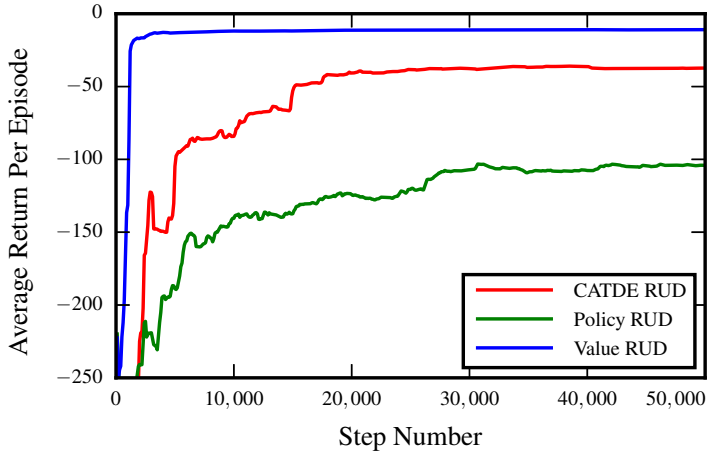
3. A **value criterion** – Unrefine if the maximal range of weights for an alternative feature dimension is greater than the range of summed weights for the extension to the value function stemming from the refined feature dimension. This would tend to indicate that this alternative feature dimension might allow the agent to better approximate the value function with less subsequent refinement and fewer weights, which will be good in the long run.

For all of these criteria, I delay unrefinement decisions by 50 steps without distractors and by 100 steps with distractors in order to reduce the likelihood of premature unrefinement. These numbers are also the result of a modest amount of parameter tuning on my part. However, it makes sense to take the values for refinement delays and to apply a small multiplier (2-4) to derive the values for unrefinement. Given that my refinement and unrefinement criteria do not offer statistical significance guarantees, some delay is required for decisions in both directions in order to allow sufficient data to be collected for decision-making.

Figure 5.2 depicts agent performance when learning the **`exact`** task using these three different rerefinement criteria. Detailed results at $50,000$ steps are listed in tables 5.3 and 5.4. In all cases, unrestricted unrefinement significantly impairs the agents' ability to converge to a stable value function. **CATDE** comes nearest to the optimal without distractors, which is consistent with my expectation that its unrefinement criterion is the least likely to be mistaken about choosing to



(a) No Distractors



(b) Distractors

Figure 5.2: ARtPE for **`exact`** using (unrefinement and) rerefinement. In the legend, "R" indicates that refinement is enabled, "U" indicates that unrefinement is enabled, and "D" and "N" indicate whether distractors are enabled or not.

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−5.03	1.65ms	16.3
Policy Criterion	−27.2	1.06ms	4.1
Value Criterion	−6.13	1.10ms	5.94

Table 5.3: Agent performance for **`exact`** using (unrefinement and) rerefinement

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−37.3	3.27ms	7.58
Policy Criterion	−104	1.79ms	3.15
Value Criterion	−10.9	2.20ms	4.75

Table 5.4: Agent performance for **`exact`** with distractors using rerefinement

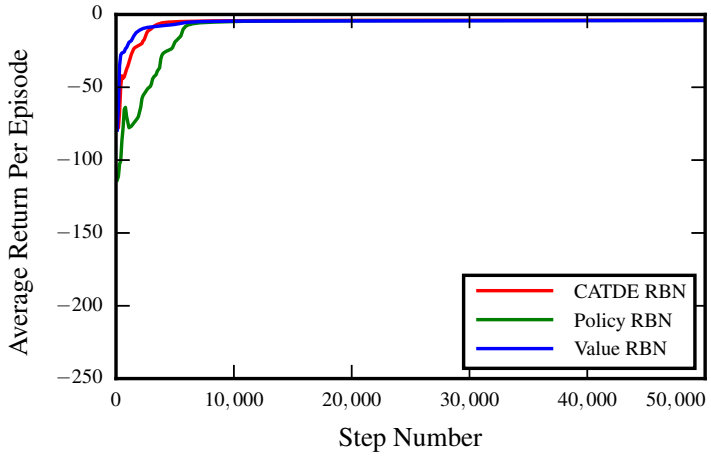
unrefine. Similarly, the **value criterion** performs better with distractors using a surprisingly small number of weights, which is consistent with my expectation that it makes better decisions about which refinements to do for the long run. The **policy criterion** does not do particularly well either with or without distractions, which perhaps refutes the idea that a larger change to the greedy set is a valuable measure of an effective value function refinement. In all cases, WCTPS decreases significantly due to the small number of weights that are maintained despite the overhead of unrefinement decisions.

Ultimately, it is clear that allowing unrestricted refinement is an ineffective strategy for allowing an agent to improve its learning efficiency, at least using the criteria I consider. This requires either refinement criteria which are guaranteed to settle on a fixed refinement at each level of the hierarchy or additional methods that inhibit an agent's ability to unrefine and rerefine, thereby forcing settling. I explore the latter approach in the rest of this section. This will test the architectural flexibility of Carli-RRL with respect to support for more complexity with respect to refinement and unrefinement decision-making.

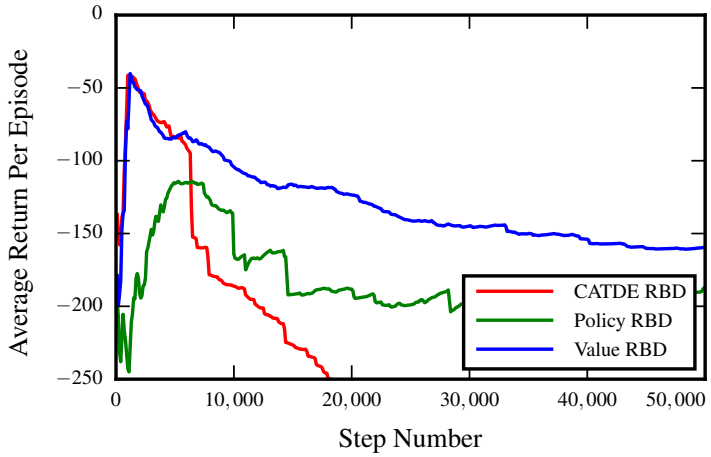
5.1.3 Rerefinement with Blacklisting Experiments

My first approach for limiting refinement in my agents is to implemented the simplest approach: blacklists for previous refinements. Simply put, whenever a feature dimension is selected for refinement at a given node, it will never again be selected for that same node.

Figure 5.3 depicts agent performance when learning the **`exact`** task using blacklists. Detailed results at $50,000$ steps are listed in tables 5.5 and 5.6. Without distractors, convergence is successfully achieved and performance actually improves for **CATDE**. However, catastrophic divergence is observed with the presence of distractor features. While perhaps difficult to anticipate, in hindsight it is reasonable to expect this behavior. Given that top feature axes may be difficult to choose between, unrefinement is expected. At the same time, those top feature axes are all good choices. When all of the features are good and the number of options is limited, maximal refinement in even a suboptimal order is not necessarily problematic. However, in the presence of distractor features, once unrefinement begins, the agent is restricted to worse and worse refinement options, resulting in a poorly structured value function. With many poor feature options, good features and their associated weights are inevitably attached to feature conjunctions that include distractors that present a random signal, making convergence very difficult. Clearly blacklists are not a successful approach for handling unrefinement with distractors.



(a) No Distractors



(b) Distractors

Figure 5.3: ARtPE for **`exact`** using rerefinement and blacklists. In the legend, "R" indicates that refinement is enabled, "B" indicates that blacklists are enabled for unrefinement, and "D" and "N" indicate whether distractors are enabled or not.

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−3.94	1.51ms	26.5
Policy Criterion	−4.04	1.20ms	25.9
Value Criterion	−4.13	1.41ms	26.8

Table 5.5: Agent performance for **`exact`** using rerefinement and blacklists

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−407	4.72ms	30.9
Policy Criterion	−187	8.55ms	87.1
Value Criterion	−159	5.55ms	44.7

Table 5.6: Agent performance for **`exact`** with distractors using rerefinement and blacklists

5.1.4 Boost

Given that blacklisting does not result in good outcomes when distractor features are present, what might be a better approach? Perhaps something that does almost the opposite?

Hypothesis 5 (Boost Instead of Blacklisting): *Since the early choices are likely to be good, boosting the likelihood of reselecting feature axes that have been selected in the past rather than reducing that likelihood to zero will result in improved performance.*

The mechanism I came up with in order to have this effect is to store a counter with each node. Starting from zero, this counter is incremented each time its direct descendants are refined. After it is incremented, it is added to a counter associated with the feature dimension being refined. Thus, the first feature dimension gets a value of 1 added to its counter, the second a value of 2, the third a value of 3, and so on. A feature dimension can be selected repeatedly, so if one were selected first and third, it would have a value of 4 associated with it, the sum of 1 and 3. This value is then used to augment the value of the **CATDE**, the change to the greedy set of actions for the **policy criterion**, or the maximal range for the **value criterion**. Since those baseline values should be fairly consistent over time while these boosts are likely to result in ever increasing separation between them, convergence to one feature dimension selection is likely.

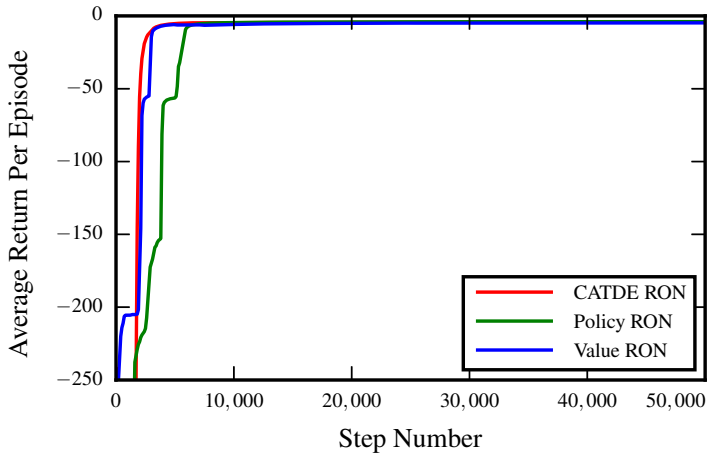
This suggests the introduction of a new parameter for how these boost values should scale relative to the value naturally associated with the refinement criterion. I set that parameter at 0.1 and have done little in the way of tuning since. While I make no claim of convergence guarantees from this strategy, it appears to work in practice in my experiments.

Figure 5.4 depicts agent performance when learning the **`exact`** task using boost. Detailed results at $50,000$ steps are listed in tables 5.7 and 5.8. Without distractors, performance generally worsens, although it actually improves for the **policy criterion**. The situation is not much better with distractors, where the **value criterion** does best by managing to thrash between different refinements despite the boost criterion. While longer runs going out to $200,000$ steps overcome these limitations, boost as implemented does not appear to be sufficient to allow the agent to effectively explore different refinements while converging quickly.

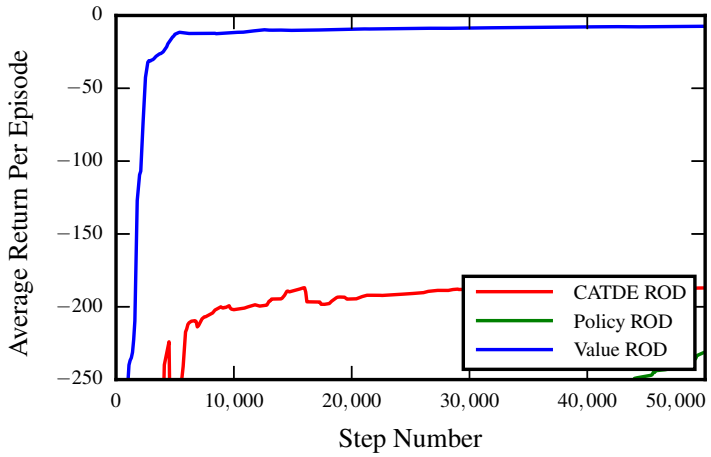
5.1.5 Rerefinement with Boost and Concrete Experiments

Looking at runs going out to $200,000$ steps (not shown) and seeing good learning performance but poor computational performance due to the ongoing reevaluations of whether rerefinements ought to be done, I sought to resolve the issue.

Hypothesis 6 (Make Refinements Concrete Eventually): *In order to avoid indefinite inefficient use of computational resources and to avoid discarding significant Temporal Difference (TD)*



(a) No Distractors



(b) Distractors

Figure 5.4: ARtPE for **`exact`** using rerefinement and boost. In the legend, "R" indicates that refinement is enabled, "O" indicates that boost is enabled for unrefinement, and "D" and "N" indicate whether distractors are enabled or not. The policy criterion performs too poorly to appear in the lower graph.

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−4.14	1.77ms	22.7
Policy Criterion	−3.95	2.22ms	24.8
Value Criterion	−4.78	1.56ms	15.1

Table 5.7: Agent performance for **`exact`** using rerefinement and boost

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−187	12.4 ms	73.2
Policy Criterion	−231	33.5 ms	218
Value Criterion	−7.42	6.91ms	26.2

Table 5.8: Agent performance for **`exact`** with distractors using rerefinement and boost

learning, once a refinement has been used for more than some fixed number of steps, make the refinement concrete and consider no others.

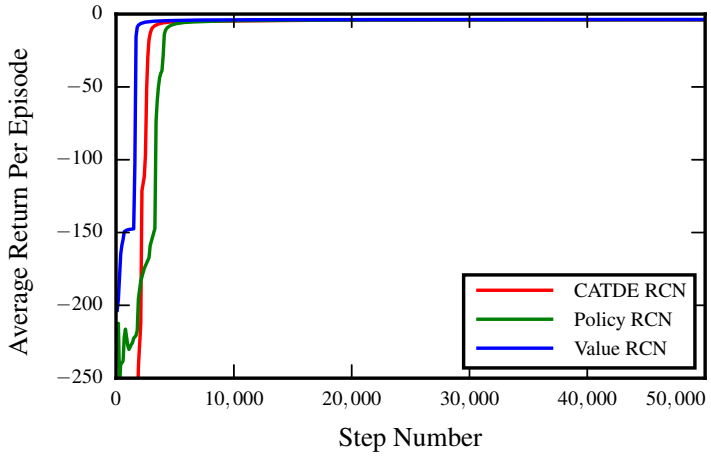
I implement this with another parameter, which I set to 500 steps for the agents without distractors and 900 steps for the agents with distractors after more extensive tuning. (The question of when an agent ought to stick to what it knows even if an alternative value function representation might be better in the long run is challenging when evaluating an agent in a finite number of steps.) Once the requisite number of steps have passed, when the agent would next consider unrefining a tile, it instead excises the child internal fringe nodes and stops considering rerefinement. This saves computational time in the limit and prevents loss of significant TD learning provided that boost is sufficient to stabilize the node for at least that many steps. Thus boost still has a significant role to play in achieving some measure of stability.

Figure 5.5 depicts agent performance when learning the **`exact`** task using boost and concrete. Detailed results at $50,000$ steps are listed in tables 5.9 and 5.10. Without distractors, performance improves for **CATDE** and the **value criterion** and computational performance improves across the board relative to the agents with no rerefinement. Distractors prove more difficult, but again the **value criterion** is improved in terms of learning, in terms of computational performance, and additionally in terms of the number of weights.

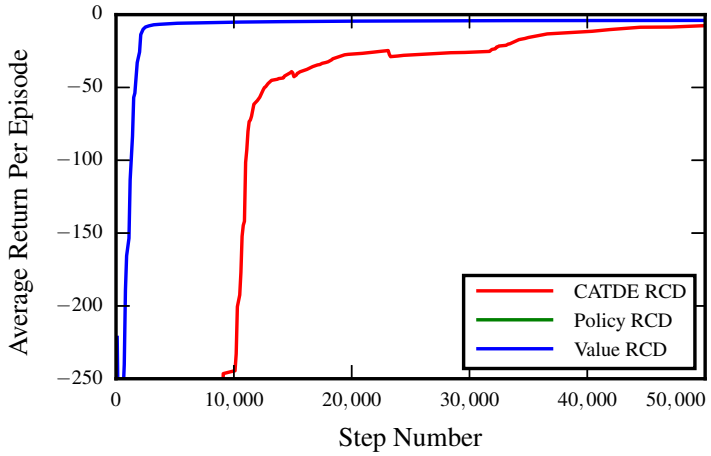
Thus the value criterion with rerefinement, boost, and concrete dominates the performance of the agents using either **CATDE** or the **policy criterion** as well as any of the preceding agents using the value criterion. This confirms Hypothesis 3, that *"for complex problems [...] the agent may be able to do [...] better by undoing refinements in favor of ones that, in retrospect, may allow more efficient TD learning."* This in turn entails that Hypothesis 2, that *"for problems with more difficult feature selection problems, adaptive Hierarchical Tile Codings (aHTCs) will result in better ARgPE or ARtPE than non-adaptive Hierarchical Tile Codings (naHTCs) since they will allow better choices to be made about which features result in earlier refinements,"* is likely to be correct. If rerefinement can be useful, then clearly refinement of an aHTC can result in better performance than a naHTC with structure of unknown optimality.

5.1.6 Discussion

Chapter 4 described the theory of how to support a computationally efficient embedding of an aHTC in a Rete to arrive at the Carli-RRL architecture. This section provides a demonstration that Carli-RRL can be used to learn to solve the **`exact`** objective of Blocks World with high ARtPE and low WCTPS. I have demonstrated that one can enable and disable refinement and rerefinement, modify the criteria for choosing to refine or unrefine, adjust the conditions under which those criteria are tested, and layer additional mechanisms on top of those criteria. I arrived



(a) No Distractors



(b) Distractors

Figure 5.5: ARtPE for **`exact`** using rerefinement and boost with concrete. In the legend, "R" indicates that refinement is enabled, "C" indicates that concrete is enabled for unrefinement with boost, and "D" and "N" indicate whether distractors are enabled or not. The policy criterion performs too poorly to appear in the lower graph.

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−3.99	0.605ms	23.6
Policy Criterion	−3.91	0.645ms	24.0
Value Criterion	−3.71	0.612ms	26.4

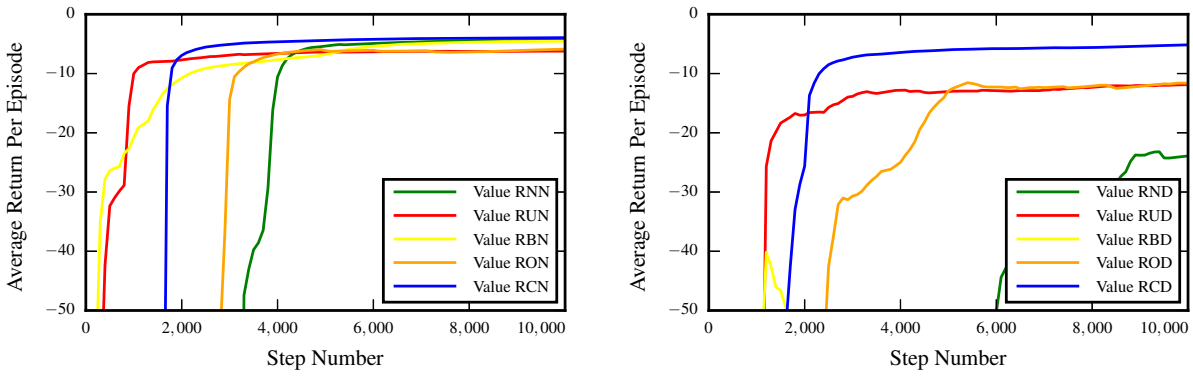
Table 5.9: Agent performance for **`exact`** using rerefinement and boost with concrete

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−7.60	19.6ms	596
Policy Criterion	−1,100	10.2ms	238
Value Criterion	−3.97	12.3ms	284

Table 5.10: Agent performance for **`exact`** with distractors using boost with concrete

at the best ARtPE for **`exact`** by using refinement and rerefinement with boost and concrete for the **value criterion**.

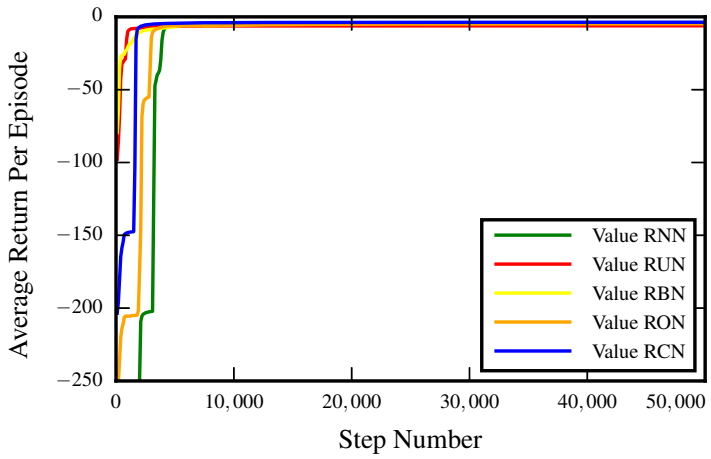
Looking at the **value criterion** in more detail, I take my earlier results and combine them in Figure 5.7 and tables 5.11 and 5.12 . I additionally zoom in on the first $10,000$ steps in Figure 5.6. Setting aside my agents that use blacklists, my agents that only refine and do not consider rerefinement of the value function are the slowest to converge in the beginning. This supports my decision to consider rerefinement in my agents. In fact, unrestricted unrefinement is the fastest to converge on its suboptimal policy, indicating that I have yet to find an optimal strategy for limiting rerefinement over time. Examining the numbers of weights for each agent is somewhat telling as to how this might be possible. With only 4.75 weights on average, the agent with unrestricted unrefinement is able to achieve -10.9 ARtPE in the presence of distractor features. My agent with rerefinement with boost and concrete provides my best ARtPE, but it requires 284 weights in the presence of distractor features. That my agents can come even close with unrestricted unrefinement and do so faster than my other agents is compelling evidence that getting the early refinements exactly right can significantly improve ARtPE.



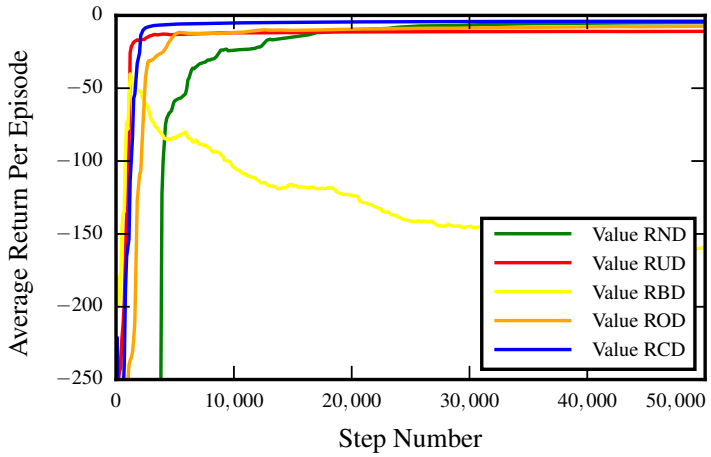
(a) No Distractors

(b) Distractors

Figure 5.6: Zoomed in ARtPE for **`exact`** using the **value criterion**. In the legend, "N" indicates that unrefinement is not enabled, "U" indicates that unrefinement is enabled, "B" indicates that blacklists are enabled for unrefinement, "O" indicates that boost is enabled for unrefinement, and "C" indicates that concrete is enabled for unrefinement with boost.



(a) No Distractors



(b) Distractors

Figure 5.7: ARtPE for **`exact`** using the **value criterion**. In the legend, "N" indicates that unrefinement is not enabled, "U" indicates that unrefinement is enabled, "B" indicates that blacklists are enabled for unrefinement, "O" indicates that boost is enabled for unrefinement, and "C" indicates that concrete is enabled for unrefinement with boost.

Value at $50,000$	Refinement	Unrefinement	Blacklists	Boost	Concrete
ARtPE	−3.97	−6.13	−4.13	−4.78	−3.71
WCTPS	1.33ms	1.10ms	1.41ms	1.56ms	0.612ms
Weights	25.6	5.94	26.8	15.1	26.4

Table 5.11: Agent statistics for **`exact`** with no distractors

Value at $50,000$	Refinement	Unrefinement	Blacklists	Boost	Concrete
ARtPE	−4.83	−10.9	−159	−7.42	−3.97
WCTPS	19.0 ms	2.20ms	5.55ms	6.91ms	12.3 ms
Weights	1,459.5	4.75	44.7	26.2	284

Table 5.12: Agent statistics for **`exact`** with distractors

5.2 Computational Efficiency Comparisons

Here I evaluate the overall computational efficiency of agents implemented using my RRL architecture for the **`stack`**, **`unstack`**, and **`on(a,b)`** objectives of Blocks World and compare the runtimes of my experiments to those of the agents implemented by Džeroski *et al.* [2001] (Section 5.2). Ideally I could compare the computational efficiency of just the value function implementations and not the other costs such as TD updates, agent implementation overhead, and environment execution. Unfortunately, only total execution time is available from related work without any specifics of the contribution of only the value function implementation. In Carli-RRL, overall runtimes are dominated by value function lookups and refinement, where they represent a combined 60%-80% of the WCTPS. Given the complexity of these processes in RRL architectures, as I described in Chapter 4, and that TD methods themselves and the environments my agents explore do not suffer from the same complexity issues, it is likely that slower implementations experience their increased costs primarily in these processes. This was certainly the case in my prototype system (Chapter 2), which was slowed down dramatically by my value function and refinement implementations. Given the unavailability of detailed WCTPS data for competing systems and my analysis that these are where the costs lie, it is reasonable to use overall runtimes as proxies for the evaluation of the computational efficiency of my value function and refinement implementations alone.

5.2.1 Computational Efficiency Compared to Soar and Carli-Prop

I believe one of the most important contributions of this work is the computational efficiency of my RRL architecture since it potentially paves the way for more widespread adoption of RRL. If it costs little more than linear function approximation, then it can be applied to a wider variety of tasks where it would not have seemed worthwhile to apply RRL before.

In Section 2.3, I observed that the Soar agents took 5.926 seconds on average to execute Blocks World for 1000 steps on a i7-7700HQ CPU. The Carli for Propositional Representations (Carli-Prop) agents using k-dimensional tries (k-d tries) took 0.049 seconds on average – a speedup factor of 120. Switching to Carli-RRL for full support of relational representations using Rete, execution times increase to 0.122 seconds on average, which is still a speedup factor of 49. Soar, Carli-Prop, and Carli-RRL are all executing with support for aHTCs for these benchmarks.

For Puddle World, executing for 50,000 steps I observed an average runtime of 51.426 seconds for Soar and 0.738 seconds for Carli-Prop. This is a speedup factor of 70 for Carli relative to Blocks World. Switching to Carli-RRL, execution times increase to 2.586 seconds on average, which is still a speedup of factor of 20.

These results, presented as time per step in Table 5.13, demonstrate the computational effi-

Environment	Soar	Carli-Prop	Carli-RRL
Blocks World 2.3.1	5926µs	49µs	122µs
Puddle World 2.3.2	1028µs	15µs	52µs

Table 5.13: Time per step for Blocks World (Section 2.2.1) and Puddle World (Section 2.2.2)

ciency of my RRL implementation. Using a Rete implementation, comparable to that used by Soar, I achieve a significant speedup for the same task. This speedup is larger for the environment with more relational structure. Additionally, my Carli-RRL agents have a great deal of relational power at their disposal that the Soar implementation did not provide, as I demonstrated with my experiments learning the **`exact`** objective with goal configurations that vary from episode to episode in Section 5.1 on page 85 – a task that neither my Soar implementation nor Carli-Prop could have attempted.

5.2.2 Computational Efficiency vs Džeroski et al.

I compare performance of the ultimate version of my online, incremental RRL implementation as described in chapter 4 to the offline RRL architecture implemented by Džeroski *et al.* [2001] for training on instances of 3-5 blocks for 45 episodes. Their architecture uses Top-down Induction of Logical DEcision trees (TILDE) to include First Order Logical Decision Trees (FOLDTs) that correspond to ATCs rather than aHTCs. They use a model that maps state-action pairs to their most recent Q-values in order to allow TILDE to induce a tree from scratch (non-incrementally) using standard information gain criteria.

For the experiments in this section, my agents use a variant of Boltzmann (or softmax) exploration, perhaps first used in the context of Reinforcement Learning (RL) by Watkins [1989], that involves selecting actions with probabilities based on a version of the Boltzmann (or Gibbs) distribution:

$$\Pr(a_i|s) = \frac{T^{-Q(s,a_i)}}{\sum_{i=0}^{n} T^{-Q(s,a_i)}} \tag{5.1}$$

from Džeroski *et al.* [2001] or alternatively

$$\Pr(a_i|s) = \frac{T^{Q(s,a_i)}}{\sum_{i=0}^{n} T^{Q(s,a_i)}} \tag{5.2}$$

from Irodova and Sloan [2005]. For equation 5.1, T represents a temperature that can be decayed to near 0 to reduce exploration over time. For equation 5.2, T acts in the opposite fashion, and increasing it over time reduces exploration. Equation 5.2 is the formulation that I implemented for my agents for these two sections.

I use off-policy Greedy-GQ(λ) with $\alpha = 0.05$, $\rho = 0.03$, $\gamma = 0.8$, and $\lambda = 0.1$ and the value

criterion (see Section 5.1.1 on page 87) with a 10 step delay for refinement, 20 for unrefinement with boost, and 40 for concrete (which I will describe in Section 5.1). I compare against their architecture using features that can learn any of these three tasks.

My total 14 binary relational features include:

1. `On(<moving-block>, TABLE)`
2. `On(<destination-block>, TABLE)`
3. `Higher-than(<moving-block>, TABLE)`
4. `Higher-than(<destination-block>, TABLE)`
5. `Clear(<block-3>)`
6. `On(<block-3>, TABLE)`
7. `On(<moving-block>, <block-3>)`
8. `On(<destination-block>, <block-3>)`
9. `Above(<moving-block>, <block-3>)`
10. `Above(<destination-block>, <block-3>)`
11. `Higher-than(<block-3>, <moving-block>)`
12. `Higher-than(<block-3>, <destination-block>)`
13. `Higher-than(<moving-block>, <block-3>)`
14. `Higher-than(<destination-block>, <block-3>)`

`On(a,b)` has two additional variables corresponding to the goals and allows for 8 additional binary relational features that include:

15. `Goal-on(<moving-block>, <goal-bottom>)`
    i.e. The block being moved is the top block in the goal.
16. `Goal-on(<goal-top>, <moving-block>)`
    i.e. The block being moved is the bottom block in the goal.
17. `Goal-on(<destination-block>, <goal-bottom>)`
    i.e. The destination block is the top block in the goal.

18. `Goal-on(<goal-top>, <destination-block>)`
    i.e. The destination block is the bottom block in the goal.

19. `Above(<moving-block>, <goal-top>)`

20. `Above(<moving-block>, <goal-bottom>)`

21. `Above(<destination-block>, <goal-top>)`

22. `Above(<destination-block>, <goal-bottom>)`

My more minimal agents include only features 4 and 14 for **`stack`**, features 4, 12, and 14 for **`unstack`**, and features 15-22 for **`on(a,b)`**. It is worth noting that there is no overlap between the features useful for **`on(a,b)`** and for the other two objectives they considered.

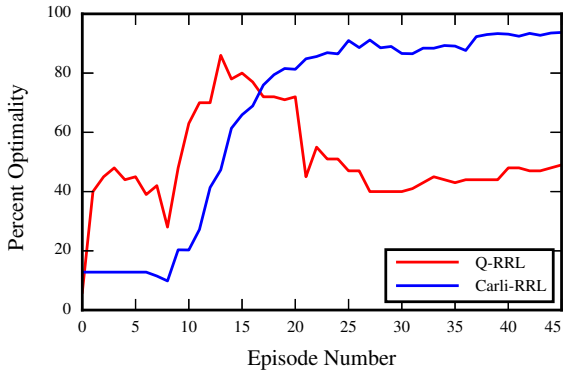
Džeroski *et al.* [2001] trained on 3 blocks for the first 5 episodes, for 4 blocks for the next 15, and 5 blocks for the final 25 for 45 episodes total, so I follow an identical approach. Comparing against their results, my agents achieve a speedup between 375 for **`on(a,b)`** and 543 for **`stack`**. **These numbers are from Carli-RRL compiled using GCC 5.5 for a version of Debian Sparc and run on a Sun Ultra 5/270 – the same hardware used by Džeroski *et al.*.** Language and compiler differences may play a role in our performance differences, but hardware gives my agents no advantage. These results, depicted in Table 5.14, make it clear that my agents represent a drastically more efficient RRL implementation than that implemented by Džeroski *et al.*. These results confirm Hypothesis 4 on page 57.

While the optimality of my agents does not match that of the Q-RRL agents by Džeroski *et al.* [2001] for **`on(a,b)`**, my agents do better than theirs for **`stack`** and **`unstack`** (Figure 5.8) despite storing only summary statistics and operating fully online and incrementally. This speaks to the efficacy of aHTCs and demonstrates that achieving good WCTPS does not necessarily demand more experience to achieve the same degree of optimality.

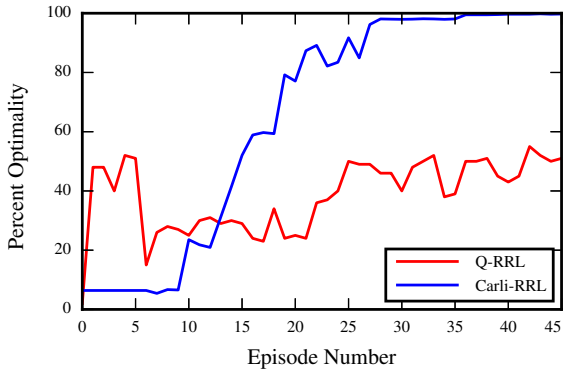
Most importantly, I demonstrated here that the Carli-RRL architecture as implemented using Rete for aHTCs is more computationally efficient, achieving a two orders of magnitude speed up over the canonical RRL architecture as implemented by Džeroski *et al.* [2001] in Section 5.2. These speedups are consistent for the **`stack`**, **`unstack`**, and **`on(a,b)`** objectives.

5.3 `Exact` Policy Scalability

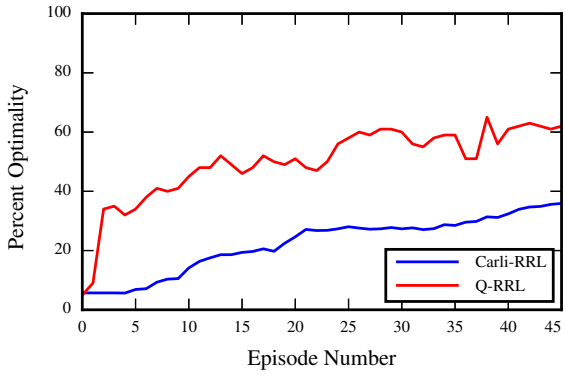
Here I evaluate the scalability of an approximately suboptimal but scalable policy that was pre-learned using Carli-RRL for Blocks World **`exact`**. (See Section 5.1.) The binary relational features that comprise the value function for this agent's policy include features 1-4 from Section 5.1. The agent was trained on 3-5 blocks as described in Section 5.1.5.



(a) Stack



(b) Unstack



(c) On(a,b)

Figure 5.8: Comparison of percent optimality over the course of training between Q-RRL [Džeroski *et al.*, 2001] and Carli-RRL.

Task	Džeroski *et al.*	Carli-RRL	Speedup
Stack	13,900s	25.6s	543
Unstack	36,400s	80.1s	454
On(a,b)	44,600s	119 s	375

Table 5.14: Runtimes for 45 episodes of training for Blocks World (Section 2.2.1) with 3-5 blocks averaged over 10 runs

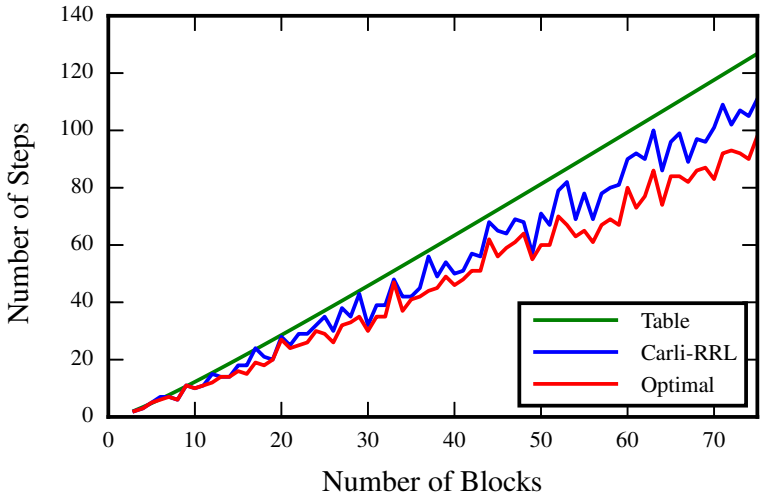
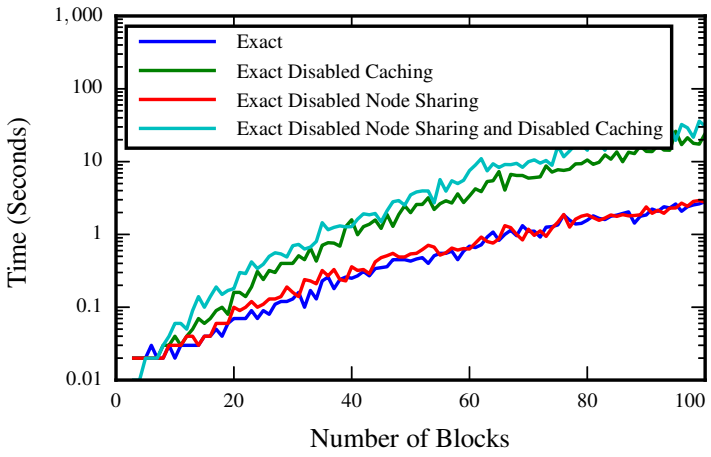


Figure 5.9: A graph of the number of steps taken by my agents with a general solution to **`exact`** with variable target configurations when compared to optimal. Each point in this graph represents one run. I additionally present an expected number of steps for a policy moving all blocks to the table and then into place for a rough upper bound.

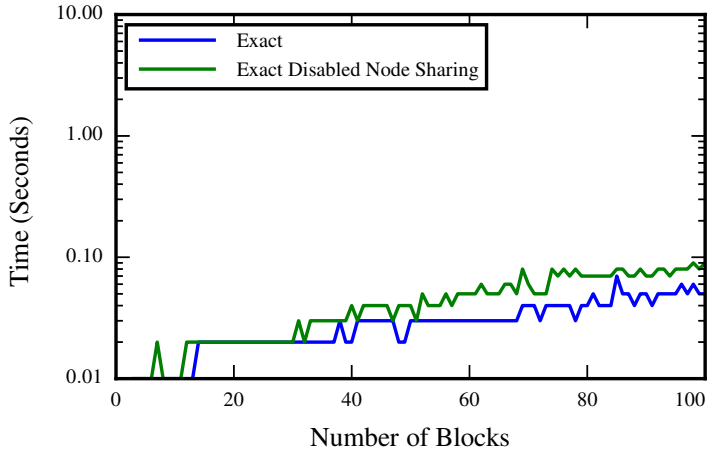
It is interesting to ask how an architecture scales as numbers of objects and relations increases. If scaling up results in catastrophic slowdowns, the computational complexity of the architecture is too high. Architectures that scale well are desirable.

While I am concerned primarily with scalability of this agent in terms of WCTPS as the number of blocks increases, it is worthwhile to investigate the degree of optimality that is achievable with its policy as well. Figure 5.9 presents both the number of steps it takes my 3-5 block pretrained Carli-RRL agent to solve a single, randomly generated **`exact`** objective and the number of steps that an optimal agent takes for that same scenario, for 3 blocks through 75 blocks. (Computation of optimal policies becomes prohibitively expensive past 75 blocks.) It additionally includes the expected number of steps for a given number of blocks for an agent that moves all blocks to the table and then moves them into place as efficiently as possible, providing an upper bound for the expected number of steps that an agent should take to solve an **`exact`** instance. Doing linear regressions of all three plots, the slope of the optimal runs is approximately 1.30, the slope of the Carli-RRL agent is approximately 1.51, and the slope of the "table" agent is approximately 1.75. This gives an expected overhead of 15% more steps for the Carli-RRL agent, which is significantly less than the expected 35% overhead of the "table" agent. Considering the high cost of calculating optimal using A*, and that **`exact`** is actually NP-hard [CHENOWETH, 1991], a policy learned using RRL that does not radically diverge as the number of blocks increases and regularly outperforms the "table" agent can be viewed as a success.

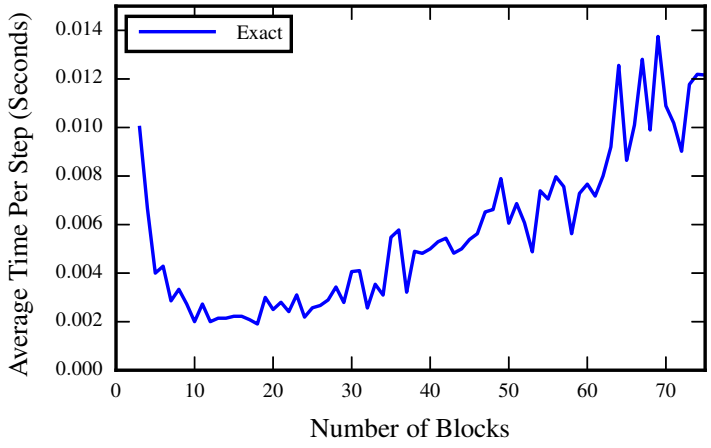
The numbers that can affect the performance of my agents are the numbers of blocks and the numbers of stacks of blocks. Figure 5.10 depicts agent computational performance for the **`exact`**



(a) For One Episode



(b) For The First Step



(c) For An Average Step

Figure 5.10: Computational performance for agents using aHTCs. Variance is a little high since each data point in these graphs represents only one run for each number of blocks rather than an average. For Figure 5.10b in particular, the number of stacks of blocks at the start of a run is somewhat random.

objective as the number of blocks in the environment increases.

One might observe, looking at Figure 5.10a, that processing only changes from step to step results in the vast majority of the computational savings in the **`exact`** environment. (Flushing Working Memory Elements (WMEs) ensures that full processing is done each step instead of processing only changes.) In fact, the benefits of node sharing in the Rete – sharing work between different rules, tiles, or queries (to use the terminology of Driessens *et al.* [2001]) where they share conditions and conjunctions of conditions – is negligible without sharing work from step to step.

5.4 Online Transfer Experiments

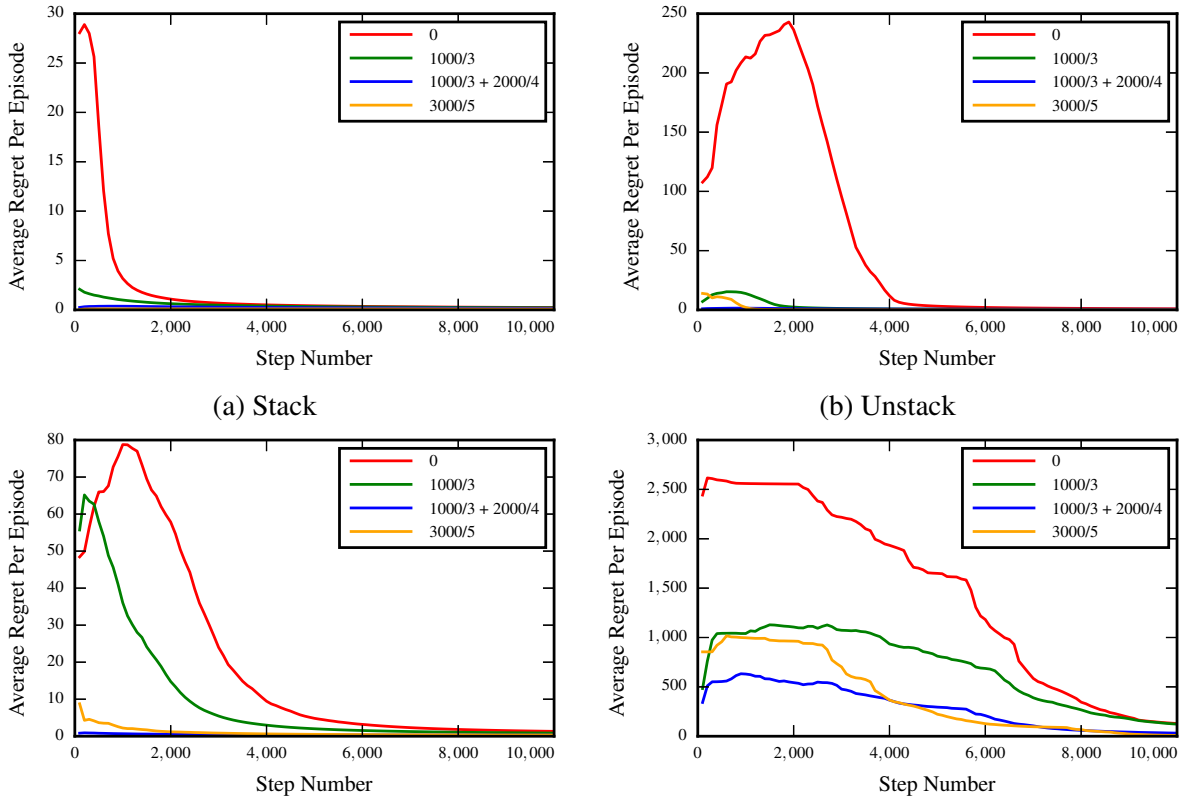
In this section, I explicitly evaluate the assumptions underlying the training schedules used in Section 5.2. I matched related work in training on 3 blocks, then 4, and so on. However, the related work did not demonstrate the benefit of doing so. Here I demonstrate that transfer occurs in Carli-RRL as a result of this kind of training.

5.4.1 Transfer for Blocks World

The agents in these experiments use the full sets of features described in Section 5.1 (including the distractors) and in Section 5.2.2. Figures 5.11a through 5.11d present the ARgPE for Blocks World agents acting on 5 blocks with no pretraining, with 1,000 steps of pretraining on 3 blocks, with 1,000 steps of pretraining on 3 blocks and an additional 2,000 steps of pretraining on 4 blocks, or with 3,000 steps of pretraining on 5 blocks.

I include agents which have pretrained on 5 blocks, the number of blocks used during the evaluation, in order to at least partially dispel the notion that any benefit of transfer simply results from having had more experience. Examining Figures 5.11b through 5.11d, it is clear that in the beginning, the agents which transfer experience from having pretrained on 3 blocks and then 4 blocks have a distinct advantage over even the agents which have pretrained on the 5 block case. Whatever the agent learns from the simpler cases results in better initial ARgPE in the 5 block scenario. For some intuition as to why, consider that the odds of randomly solving instances in early exploration are higher when the number of blocks is small, that the lengths of episodes is shorter in general, allowing an agent training on fewer blocks to experience more episodes, and that the relational features function independently of the number of blocks.

However, with the exception of **`on(a,b)`**, by the end of 10,000 steps, the agents that have trained on 5 blocks for 3,000 steps are doing better with respect to ARgPE than the agents that pretrained on 3 blocks and then on 4 blocks, which would seem to indicate that those agents actually converge on a more optimal policy more quickly. The overall regret experienced may be



(a) Stack

(b) Unstack

(c) On(a,b)

(d) Exact

Figure 5.11: Transfer experiments with agents with (red) no pretraining, (green) 1000 steps of pretraining on 3 blocks, (blue) 1000 steps of pretraining on 3 blocks and an additional 2000 steps of pretraining on 4 blocks, or (orange) 3000 steps of pretraining on 5 blocks – the number of blocks I evaluate the agents on in these graphs.

higher if one considers the pretraining, but if achieving an optimal policy as quickly as possible is your objective, transfer may not be best. I actually had more difficulty achieving good percent optimality when comparing against [Džeroski *et al.*, 2001] in Figure 5.8 once I copied their training schedule, so this conclusion is consistent with that experience.

Additionally, it is worth noting that for **`stack`**, pretraining on 5 blocks seems to be sufficient to actually start off with better ARgPE than the transfer agents in Figure 5.11a. This illustrates the simplicity of learning the optimal policy for **`stack`**.

5.4.2 Transfer for Taxicab

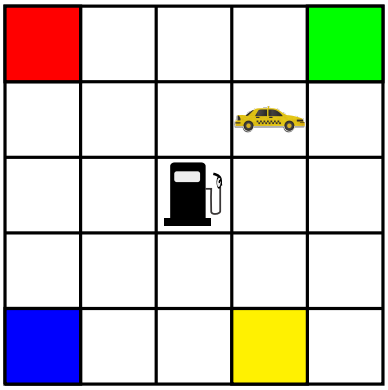
I first encountered the Taxicab problem in [Dietterich, 1998]. I present the canonical layout, minus the walls, in Figure 5.12a. There is a taxicab with 14 maximum fuel that expends 1 fuel per move in any cardinal direction. The agent is aware of both which possible destination, represented with a color, a passenger is waiting at and which destination that passenger wishes to be taken to. It

must navigate to the passenger, execute a pickup action to collect the passenger, navigate to the destination, and execute a drop off action to deliver the passenger. Along the way, the agent must potentially stop at filling stations to execute a refuel action that refills the taxicab to its maximum fuel in order to avoid running out of fuel and thereby being unable to complete its delivery. Running out of fuel at the destination with the passenger on board is not a failure condition, nor is running out of fuel exactly at a filling station. With -1 reward per step and a significant penalty of -100 for failure, an agent must learn to accomplish this task as efficiently as possible. For purposes of simplicity, I do not incorporate walls, as were present in the original version.

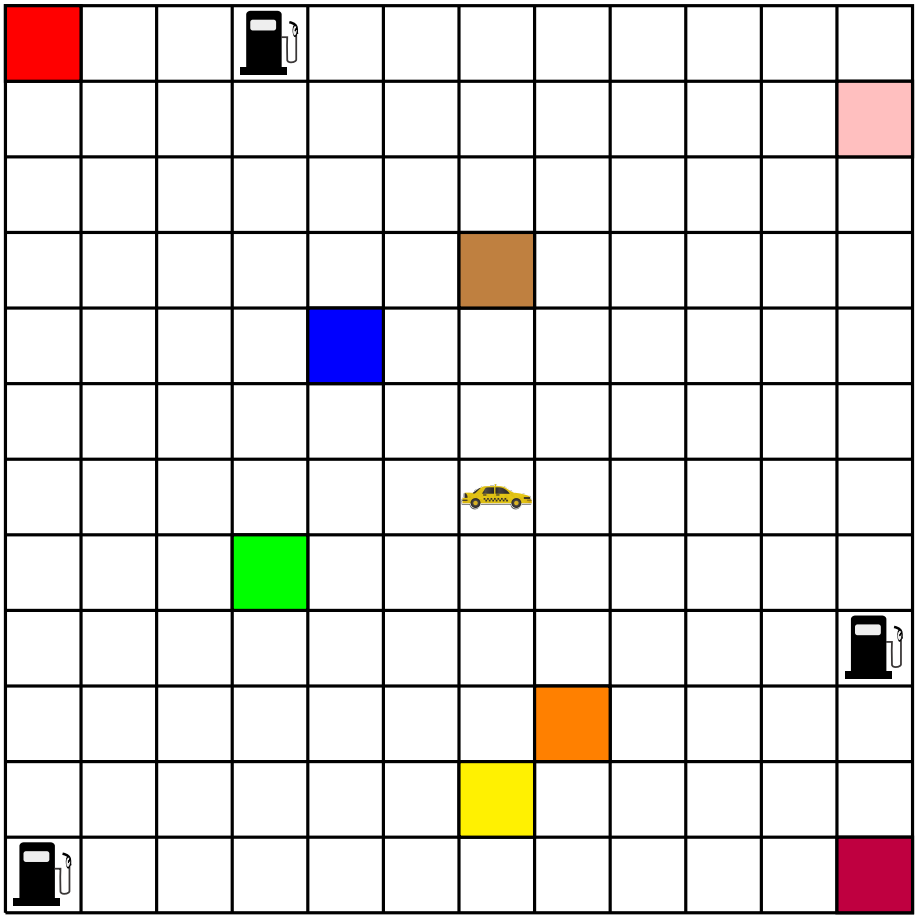
I developed a scalable, parameterized version of the Taxicab environment. For a specified grid size, amount of fuel, number of filling stations, and number of destinations, an instance can be randomly generated. To ensure solvability, I enforce that all filling stations are reachable from one another using a full tank, that all destinations are within range of at least one filling station using only half a tank of fuel, and that the taxicab always starts with sufficient fuel to reach at least one filling station. For large grids relative to the amount of fuel, it may be the case that multiple refueling stops are necessary in order to move from one destination to another.

For a difficult example, let us refer to the layout in Figure 5.12b. If an agent begins with $7/14$ fuel and must deliver a passenger from red to pink, it has a difficult task ahead of it. It must:

1. Navigate to the filling station in the lower right
2. Refuel
3. Navigate to the filling station in the lower left
4. Refuel
5. Navigate to either red or the refilling station in the upper right
6. Either collect the passenger or refuel
7. Navigate to the other location from Step 5
8. Either refuel or collect the passenger
9. Navigate back to the filling station in the lower left
10. Refuel
11. Navigate back to the filling station in the lower right
12. Refuel



(a) Canonical 5x5



(b) A 12x12 Instance

Figure 5.12: Taxicab

13. Navigate to pink

14. Drop off the passenger

An agent capable of succeeding at this task for arbitrary, variable instances must have capabilities beyond those of agents that solve only a single fixed instance with a single filling station.

The relational features available to my agents include:

1. `Action(MOVE)`, `Action(REFUEL)`, `Action(PICKUP)`, or `Action(DROPOFF)`

2. `Move-direction(NORTH)`, `Move-direction(SOUTH)`, `Move-direction(EAST)`, `Move-direction(WEST)`, or `Move-direction(NONE)` (for non-move actions)

3. `Passenger-at(SOURCE)`, `Passenger-at(DESTINATION)`, or `Passenger-at(TAXICAB)`

4. `Fuel-for-next-stop(INSUFFICIENT)`, `Fuel-for-next-stop(SUFFICIENT_ONEWAY)`, or `Fuel-for-next-stop(SUFFICIENT_ROUNDTRIP)`

5. `Toward-next-stop(TRUE)` or `Toward-next-stop(FALSE)`

6. `Toward-fuel(TRUE)` or `Toward-fuel(FALSE)`

7. `Toward-fuel-for-next-stop(TRUE)` or `Toward-fuel-for-next-stop(FALSE)`

Features 4 through 7 might be provided by any modern navigation system. I would characterize my agents as learning the top level problem of deciding what to do next, but not having to learn the details of how to navigate to a particular location, or how to find filling stations. Since a move action can be toward the next stop but still result in failure given fuel constraints and an action can be toward fuel but in the wrong direction, there still exists some complexity to the policy that an agent must learn to be successful in the general case.

For Taxicab, my agents use on-policy Greedy-GQ(λ) with $\alpha = 0.01$, $\rho = 0.1$, $\gamma = 0.99$, and $\lambda = 0.3$ and the value criterion (see Section 5.1.1 on page 87) with a 20 step delay for refinement, 50 for unrefinement with boost, and 100 for concrete (which I will describe in Section 5.1). My agents use Boltzmann exploration with temperature initially set to $1/10$ with the denominator being incremented by 0.1 after each episode. Additionally, my agents have a 100 step cutoff to prevent

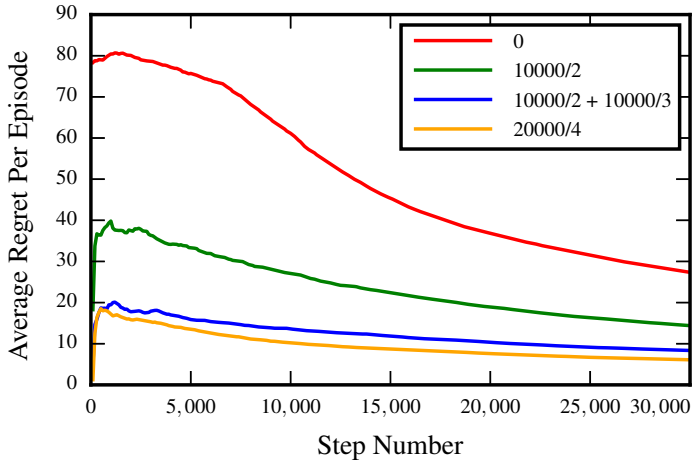


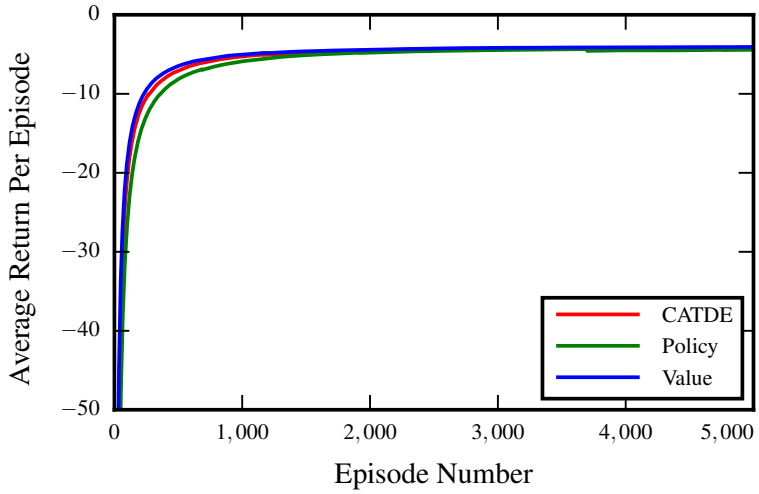
Figure 5.13: A transfer experiment for Taxicab with agents with (red) no pretraining, (green) 10000 steps of pretraining on 2 filling stations with 6 destinations, (blue) 10000 steps of pretraining on 2 filling stations with 6 destinations and an additional 10000 steps of pretraining on 3 filling stations with 8 destinations, or (orange) 20000 steps of pretraining on 4 filling stations with 10 destinations – the scenario I evaluate the agents on in these graphs.

them from cycling indefinitely to prevent failure, which can result in the lesson that failure is better than trying forever.

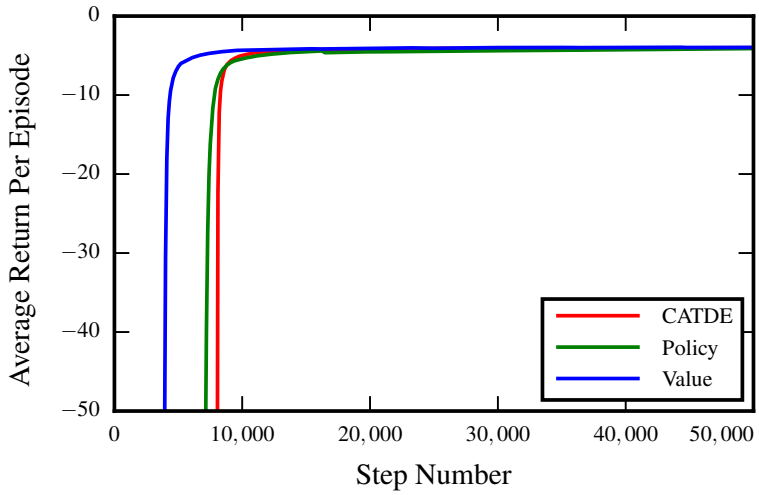
For purposes of this transfer experiment, the objective for my agents is to operate with up to 14 fuel in a 20x20 grid with 4 filling stations and 10 destinations. I hold fuel and the grid size constant but decrease the number of filling stations and destinations to 2 or 3 and 6 or 8 respectively. Transfer results are presented in Figure 5.13. The agents that have $10,000$ steps of pretraining on only 2 filling stations with 6 destinations and then $10,000$ more steps on 3 filling stations with 8 destinations perform nearly as well as the agents that trained on 4 filling stations from the beginning. Again, this provides further evidence that achieving a more optimal policy more quickly through transfer is a challenging problem, but that reducing overall ARgPE through transfer is attainable.

5.5 Average Return Per Episode (ARtPE) Evaluation

I introduced ARgPE and ARtPE early in this thesis (Section 1.2.2) and used ARtPE extensively throughout. Let us revisit my agents for the **`exact`** objective of Blocks World and plot ARtPE against the number of episodes (Figure 5.14a) in addition to plotting ARtPE against the number of steps (Figure 5.14b) to allow evaluation of whether my decision to use plots of ARtPE against steps was well informed. I plot $5,000$ episodes or $50,000$ steps since episode lengths on average are in the neighborhood of 5 steps long.



(a) Episodic



(b) Stepwise

Figure 5.14: ARtPE for agents learning the **`exact`** objective in Blocks World plotted against episodes and against steps.

The episodic plot results in all three agents looking very similar. The value criterion appears to be marginally better than CATDE, and both appear to be better than the policy criterion, but the patterns of improvement appear to be identical. The stepwise plot however, such as the ones I include throughout this thesis, illustrates the characteristic differences in the improvement of the ARtPE of these agents. The agents using the value criterion clearly arrive at a near optimal policy in approximately half the number of steps as those using either CATDE or the policy criterion. Additionally, it is possible to observe that CATDE overtakes the policy criterion at around $8,500$ steps. These details are lost when plotting episodically since early episodes that are quite long are treated as equivalent to the shorter ones experienced by the more successful agents. Plotting stepwise makes the costs of longer episodes more explicit. This affirms that my decision to plot ARtPE stepwise throughout this thesis was useful for doing my evaluations.

5.6 Discussion

I opened this chapter with an evaluation of the ARtPE and WCTPS achievable with Carli-RRL for the **`exact`** objective of Blocks World (Section 5.1). As part of that process, I additionally evaluated the efficacy of using CATDE, a policy criterion, and a value criterion for value function refinement, and found the value criterion to be generally the most effective in terms of ARtPE. Additionally, I explored the utility of doing unrefinement in conjunction with either blacklists, a novel boost mechanism, or the boost mechanism in conjunction with a concrete mechanism for eventually finalizing refinements. These are all minor contributions of this dissertation.

I then compared the WCTPS of my prototype architecture from Chapter 2, Carli-Prop, and Carli-RRL and found Carli-RRL to be far better than the prototype and within a factor of 3-4 of Carli-Prop even with the added functionality to support RRL. I additionally compared its performance to that of Džeroski *et al.*'s [2001] Q-RRL and found significantly better percent optimality past a small number of episodes for **`stack`** and **`unstack`** with a two orders of magnitude reduction in computation time. This demonstrates that Carli-RRL has been successful in significantly reducing the costs of RRL without significantly reducing its learning capabilities.

Investigating the ability of my agents to transfer learning, I first evaluated the degree of optimality achieved by my Carli-RRL agent for the **`exact`** objective as the number of blocks scales up. The policy consistently achieves a level of optimality greater than that of an agent that simply moves everything onto the table and then into place as the number of blocks scales up. I additionally evaluated transfer achieved by learning first on smaller numbers of blocks and working up to bigger problem instances, and showed that transfer is effective in reducing ARgPE.

CHAPTER 6

Higher Complexity, Higher Order Grammar

The grammar as defined in Section 4.3 provides methods for manipulating existing features and for generating subtiles for **ranged features**. However, there is a limitation implicit in it that might be difficult to notice without extensive consideration. That limitation is that the number of `Variables` is fixed from the start, even if **ranged features** allow for potentially unlimited refinement.

This limitation is less onerous that it might sound. Given that `Variables` can match unlimited numbers of objects in the world, it could be considered a kind of attentional limitation for an agent rather than a limit on the number of things that can be considered. However, if an agent must attend to a variable or unknown number of some class of object (i.e. must include a variable or unknown number of them in a feature conjunction) in order to be able to succeed in a task, then this limitation poses a problem.

Consider the kinds of features used by Džeroski *et al.* [2001] in solving tasks in Blocks World (Section 1.3.1). Their features, listed in Section 1.3.5 on page 27, are actually sufficient to solve **`stack`**, **`unstack`**, and **`on(a,b)`** with a fixed number of `Variables`. Their agents appear to refer to at most 3 blocks in any feature conjunction. One can consider only the block to be moved, the destination block, and some third block and know enough to decide the next action. The existentials and existential negations in Carli for Relational Reinforcement Learning (Carli-RRL) provide a great deal of representational power, allowing an agent to learn a lot, even without the ability to attend to a fourth block. For example, the questions of whether the block being moved is higher than the destination block and whether there exists a third block that the destination block is `higher-than` are sufficient to make good decisions for the **`stack`** task. One need never consider a fourth block at the same time.

The **`exact`** task is more demanding. Without a memory or a planning method in addition to Reinforcement Learning (RL), an RL agent is forced to attend to the entire state at once to make good decisions using Džeroski *et al.*'s features. For an unknown number of blocks, the ability to add more and more `Variables` is necessary if an agent is to have any chance at learning

to achieve all of the possible goal configurations of **exact**. Even then, the first time an agent encounters a case in which it must attend to a fourth block when it has only learned to handle three, it will have more learning to do.

Carli-RRL solved **exact** in Chapter 5 using a set of features that is stack-oriented rather than block-oriented. That is, features considered whether a stack in the world matched a stack in the goal configuration rather than considering whether individual blocks were clear or on top of one another. Here Carli-RRL attempts to solve **exact** using only on-top relations between individual blocks in the environment and in the goal configuration – a much more difficult task than any of the ones the Relational Reinforcement Learning (RRL) agents implemented by Džeroski *et al.* [2001] solved. To do this, the method I pursue is to extend Carli-RRL's grammar for mapping adaptive Hierarchical Tile Codings (aHTCs) onto Rete to support the derivation of new features that test additional variables that were previously not addressed by any existing features of the agent. At the time of refinement, this Higher Order Grammar (HOG) will require that new features are created that correspond to existing relations but with a new variable in the place of one that already exists. This requirement is in addition to the standard refinement requirements that features for other feature dimensions are copied into the fringes of the new features and that new features are created corresponding to further refinement of existing continuous variables.

This will enable an agent to consider only one relation, such as `on(a,b)`, and to then decide to consider `on(a,c)` and `on(c,b)` as well. This will eventually allow it to solve higher complexity problems. This is something that Top-down Induction of Logical DEcision trees (TILDE) by Džeroski *et al.* [2001] was capable of doing, but it depended on the presence of a model in the form of a map of state-action pairs onto most recent Q-values in order to allow TILDE to generate new First Order Logical Decision Trees (FOLDTs) offline from scratch. My goal in implementing my HOG is to continue to allow my agents to incrementally refine its aHTCs, and without the use of a model.

I describe the implementation of this HOG in Section 6.1. I describe an additional extension in Section 6.2 to support the case in which the number of relevant objects in the world dips below the number of variables used by the agent to refer to that class of object. I test its ability to solve the **exact** objective in Blocks World in Section 6.3. Then I evaluate its efficacy on a new adventure task in Section 6.4.

6.1 Implementation

My method of extending my rule grammar has one objective: to allow the creation of features in fringe nodes that are like existing features, but have new variables in the place of old ones. Requirements include:

1. A guarantee that the new variable is truly distinct from existing variables. It is easy to write rules in which different variables can refer to the same object, but I want to avoid this to ensure that `on(A,B)` tests something different from `on(C,D)`.

2. The ability to distinguish the arity of the HOG feature. That is, should a successor to a feature comparing `<block-2>` to `<block-3>` consider generating features comparing both `<block-2>` to `<block-4>` and `<block-4>` to `<block-3>` (binary) or just `<block-2>` to `<block-3>` (unary)?

3. The ability to distinguish features that should be part of this HOG (unary or binary) from ones that should not be (nullary).

Consider the HOG rules in Figure 6.1. They introduce a third block `Variable` and ensure that it refers to a block different from the one referred to by either of the block `Variable`s that preceded it. The introduction of a fourth block `Variable` would require an additional inequality comparison predicate node to compare against `<block-3>` for a total of 3, a fifth would require another still to compare against `<block-4>`, and so on. The HOG rules in Figure 6.1 can be simplified if those features are part of a feature conjunction that already includes `<block-3>`, as depicted in Figure 6.2.

It is not apparent from just the conditions whether a feature such as that depicted in Figure 6.1b is a unary or binary HOG feature or a nullary, non-HOG feature. For that reason, I make a critical change for these rules in the `:feature` directive. Between the type of node and the name of the parent, I insert an integer specifying the arity of the HOG rule. If the feature itself refers to only one instance of the HOG `Variable`, as in Figure 6.1a, then it is a unary HOG rule. If it refers to two instances of the HOG `Variable`, as in Figure 6.1b, then it is a binary HOG rule.[1]

[1] `<block-1>` and `<block-2>` do not count as instances of the HOG `Variable` for purposes of being marked binary since they are in the original `bw*general` and were not introduced by rules that are part of the HOG. Ordinary Hierarchical Tile Coding (HTC) rules handle the cases involving only `<block-1>` and `<block-2>` separately.

```
sp {bw*block-unary-3on0-t
  :feature 2 fringe 1 bw*...
  &bw*general
  (<blocks> ^block <block-3>)
  (<block-3> != <block-1>)
  (<block-3> != <block-2>)
  +(<block-3> ^on <table>)
-->
  = 0.0
}
```

(a) Unary HOG rule

```
sp {bw*block-binary-2on3-t
  :feature 2 fringe 2 bw*...
  &bw*general
  (<blocks> ^block <block-3>)
  (<block-3> != <block-1>)
  (<block-3> != <block-2>)
  +(<block-2> ^on <block-3>)
-->
  = 0.0
}
```

(b) Binary HOG rule

Figure 6.1: Higher Order Grammar (HOG) rules for Blocks World from Section 1.3.1

```
sp {bw*f86
  :feature 2 fringe 1 bw*...
  &bw*u58
  +(<block-3> ^on <table>)
-->
  = 0.0
}
```

(a) Unary HOG rule

```
sp {bw*f88
  :feature 2 fringe 2 bw*...
  &bw*u63
  +(<block-2> ^on <block-3>)
-->
  = 0.0
}
```

(b) Binary HOG rule

Figure 6.2: Rules implementing Higher Order Grammar (HOG) features from Figure 6.1 can be simplified once <block-3> has been introduced.

```
sp {blocks-world*f273
  :creation-time 103
  :feature 4 fringe 1 bw*u123
  &bw*u123
  (<blocks> ^block <block-4>)
  (<block-4> != <block-1>)
  (<block-4> != <block-2>)
  (<block-4> != <block-3>)
  +(<block-4> ^on <table>)
-->
  = -2.9 63.5 0.9 0.0 0
}
```

Figure 6.3: Unary Higher Order Grammar (HOG) successor for 6.1a

```
sp {bw*f270
  :creation-time 103
  :feature 4 fringe 2 bw*u123
  &bw*u123
  (<blocks> ^block <block-4>)
  (<block-4> != <block-1>)
  (<block-4> != <block-2>)
  (<block-4> != <block-3>)
  +(<block-2> ^on <block-4>)
-->
  = -2.9 63.5 0.9 0.0 0
}
```

(a) Binary HOG successor #1

```
sp {bw*f271
  :creation-time 103
  :feature 4 fringe 2 bw*u123
  &bw*u123
  (<blocks> ^block <block-4>)
  (<block-4> != <block-1>)
  (<block-4> != <block-2>)
  (<block-4> != <block-3>)
  +(<block-3> ^on <block-4>)
-->
  = -2.9 63.5 0.9 0.0 0
}
```

(b) Binary HOG successor #2

Figure 6.4: Binary Higher Order Grammar (HOG) successors for 6.1b

A unary successor to the rule depicted in Figure 6.1a can be seen in Figure 6.3. It extends the rule with the introduction of `<block-4>` and adds a string of negation `Predicate Node`s one longer than that required by `<block-3>`. The binary HOG rule in Figure 6.1b actually has two successors, as depicted in Figure 6.4. The rule in Figure 6.4a changes the relation between `<block-2>` and `<block-3>` to refer to `<block-2>` and `<block-4>`. The rule in Figure 6.4b instead changes the relation between `<block-2>` and `<block-3>` to refer to `<block-3>` and `<block-4>`. This kind of double successor is necessary for any binary HOG feature that refers to the most recent two HOG `Variable`s. Since any features referring to only `<block-1>` through `<block-3>` that remain as part of the fringe post-refinement must be copied down as successors as well. This means that with a grammar including binary HOG features, one fringe node at a given level of refinement can translate to as many as three at the next level of refinement.

6.1.1 Tractability

While the number of features in the fringe must monotonically decrease for non-HOG features, the HOG changes that picture drastically and allows for explosions in the number of possible features to consider. This cost of a HOG quickly becomes prohibitive, as each new `Variable` has a cost complexity per token of

$$\sum_{i=n-k}^{n} i \tag{6.1}$$

where n refers to the number of blocks in the environment and k refers to the k^{th} block `Variable`. The number of tokens matching for the k^{th} block `Variable` corresponds to the binomial coefficient, $\binom{n}{k}$, resulting in the combined complexity for `Variables` 1 through k:

$$\sum_{j=1}^{k} \left\{ \binom{n}{j} \sum_{i=n-j}^{n} i \right\} \tag{6.2}$$

Since that is $O(n!n^2)$ in the limit as $k \to n$, clearly this is tractable only for small values of k and n. The jump from 3 blocks to 4 blocks increases complexity by a factor of 7 even before new features using those `Variables` are actually taken into consideration. Thanks to the Rete algorithm (Section 4.2), `fringe` nodes for a given `unsplit` node can share this computational cost but, as the number of new `Variables` grows, the $O(n!n^2)$ growth will eventually kill performance.

That being said, the HOG, even in its current form, is still potentially useful for small numbers of `Variables`. For example, for an adventure environment (introduced in Section 6.4) in which an enemy may or may not be present, rather than writing rules to awkwardly handle a null enemy, I can use a HOG `Variable` for the enemy to handle whether there may be an enemy or not, and it will automatically generalize to handle additional enemies as needed.

6.2 Null Higher Order Grammar Rules

At this point an astute reader might inquire as to how I actually handle the case of whether `<block-4>` is present or not. Up until this point in my explanation of the HOG, I have actually skipped that test. More precisely, it comes bundled with tests of features dependent on the new `Variable`. This is desirable for my refinement criteria, since the mere presence or absence of a feature is unlikely to have much signal to determine whether refinement ought to occur or not, and this is particularly true in cases where absence never occurs.

However, this is potentially quite problematic. Should the agent ever refine over a feature referencing `<block-4>`, it will be impossible to incorporate new features into its value function for cases in which no fourth block is present without unrefining past the point of inclusion of `<block-4>`. By bundling the introduction of the `Variable` with the new features testing it, I have inadvertently failed to fully partition the state space. The solution is to include a null path for the case in which the `Variable` is not present. A null-HOG rule can be seen in Figure 6.5. With the introduction of the null-HOG `Node`, testing the presence or absence of `<block-4>` works as expected.

This solution is not ideal since, assuming generalization is possible from the 3-block case to the 4-block case, once `<block-4>` is introduced, any further features used for learning about

```
sp {bw*block-nullhog
  :feature 2 fringe 0 bw*general
  &bw*general
  -{(<s> ^blocks <blocks>)
  (<blocks> ^block <block-1>)
  (<blocks> ^block <block-2>)
  (<block-2> != <block-1>)
  (<blocks> ^block <block-3>)
  (<block-3> != <block-1>)
  (<block-3> != <block-2>)}
-->
  = 0.0
}
```

Figure 6.5: Null-HOG rule for <block-3>

the 3-block case will have to be learned over separately for the case in which there are 4 or more blocks. In the extreme case where a feature testing <block-4> is selected immediately after the first feature testing <block-3>, almost all learning specifically for the 3-block case will be redundant with learning for cases involving 4 or more blocks. However, the ability to do this redundant learning fixes the incomplete partitioning problem introduced by the introduction of new features and allows the 3-block case to be learned out of order with the 4-block case. It is generally preferable to learn the 3-block case, then the 4-block case, and finally the 5-block case to reduce the likelihood of feature refinement being necessary below null-HOG Nodes.[2]

6.3 Blocks World, Exact Objective – A HOG Stress Test

With a higher order grammar implemented, I chose to evaluate its capabilities using the **exact** objective for Blocks World.

Here I present a stress test for my HOG, by attempting to solve Blocks World for 3-5 blocks. As most features Džeroski *et al.* [2001] described would not be useful for a fully specified objective, I restricted the features for my HOG agent to **on(a,b)** relations. I have two types of on(a,b) relations: ones that describe the current state of the world (i.e. on(a,b)) and ones that describe the objective (i.e. goal-on(a,b)). Given such a meager representation, this effectively stress tests my architecture to see if it can learn to solve the problem.

Training my HOG agents for **exact**, I used off-policy Greedy-GQ(λ) with $\alpha = 0.03$, $\rho = 0.01$, $\gamma = 0.9$, and $\lambda = 0.3$ and the value criterion with a 20 step delay for refinement, 50 for unrefinement with boost, and 100 for concrete. The values for unrefinement delays and concrete

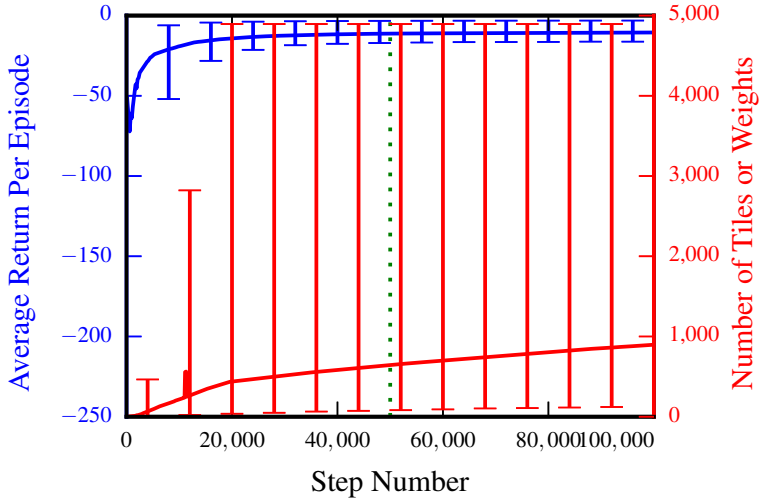
[2]Originally training strictly with this pattern, I did not initially observe the need for null-HOG Nodes.

were minimally tuned under the assumption that higher values would be necessary when using the HOG, but the other values are copied straight from the non-HOG agents for **`exact`**. I additionally impose a cutoff of 100 steps before any given episode is terminated, preventing agents from getting stuck in situations they are incapable of solving in a reasonable about of time.

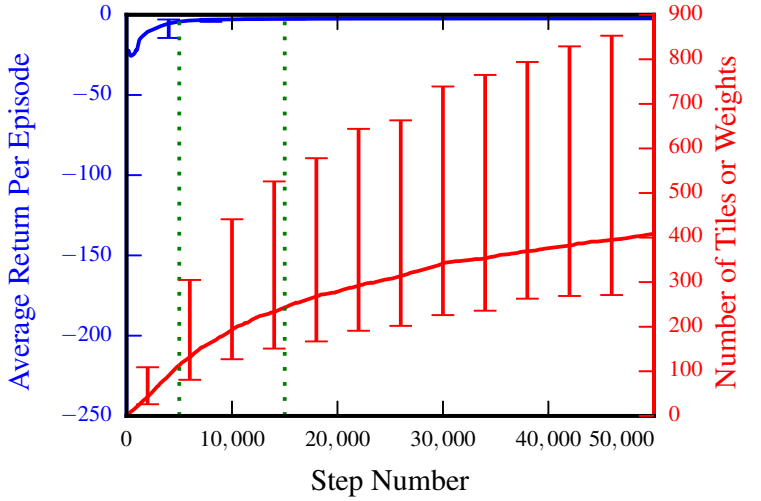
In Section 5.1, I trained my agents on 3-5 blocks from the start. Early experimentation with the HOG suggested that this would be too difficult, so my first attempt at easing the agent into 5 blocks was to train on 3-4 blocks for 50,000 steps and to transfer that learning to another phase of learning on 3-5 blocks for 50,000 additional steps. This training is depicted in Figure 6.6a, but it does not tell the whole story. Ultimately, agents suffer in one of two ways under this training schedule. Either they are not particularly successful and drag down the average Average Return Per Episode (ARtPE), or they are successful but ultimately overrefine the value function to the point that the number of features being tested slows down the agent too much. For one seed out of the 20 I tested, I ultimately cut off the agent after 20,000 steps and a week of computation.

Training instead on 3 blocks for 5,000 steps, for 3-4 blocks for the following 10,000 steps, and finally 3-5 blocks for another 35,000 steps, we arrive at a very different picture in Figure 6.6b. Overrefinement is still an issue, but the error bars for the ARtPE of the agents are virtually invisible by the end of 50,000 steps. They all converge to near optimal policies for 3-5 blocks. Furthermore, as listed in Table 6.1, the average Wall-Clock Time Per Step (WCTPS) drops from 913ms (excluding the terminated agent) to 43.7ms and the average number of weights drops from 8,972 to only 817. It is safe to conclude that it is essential to train over simpler problems before more complex ones when using a HOG.

Comparing to my approximate **`exact`** agents from Section 5.1.5 which had a WCTPS of 12.3ms, the HOG agents which have a WCTPS of 43.7ms take approximately 3.55 times as long per step. However, given that 817 is approximately 2.88 times the 284 weights that were required for the earlier agents and that these agents are additionally responsible for creating and evaluating new `Variables` and corresponding relations, one can see that the system is scaling reasonably well. It impossible however to directly compare the ARtPE between these agents, since the different conditions under which they are trained and evaluated affect expected episode lengths and costs.



(a) ARtPE and weights for 20 HOG agents training on 3-4 blocks for 50,000 steps and followed by 3-5 blocks for 100,000 steps.



(b) ARtPE and weights for 20 HOG agents training on 3 blocks for 5,000 steps, followed by 4 blocks 10,000 steps, and finally 5 blocks for 35,000 steps.

Figure 6.6: ARtPE and weights for 20 HOG agents trained on 3-5 blocks under different transfer scenarios.

Training Schedule	ARtPE	WCTPS	# Weights
3-4 $\rightarrow$ 3-5	−10.5	913 ms	8,972
3 $\rightarrow$ 3-4 $\rightarrow$ 3-5	−2.29	43.7ms	817

Table 6.1: HOG agent performance for **`exact`** using different training schedules.

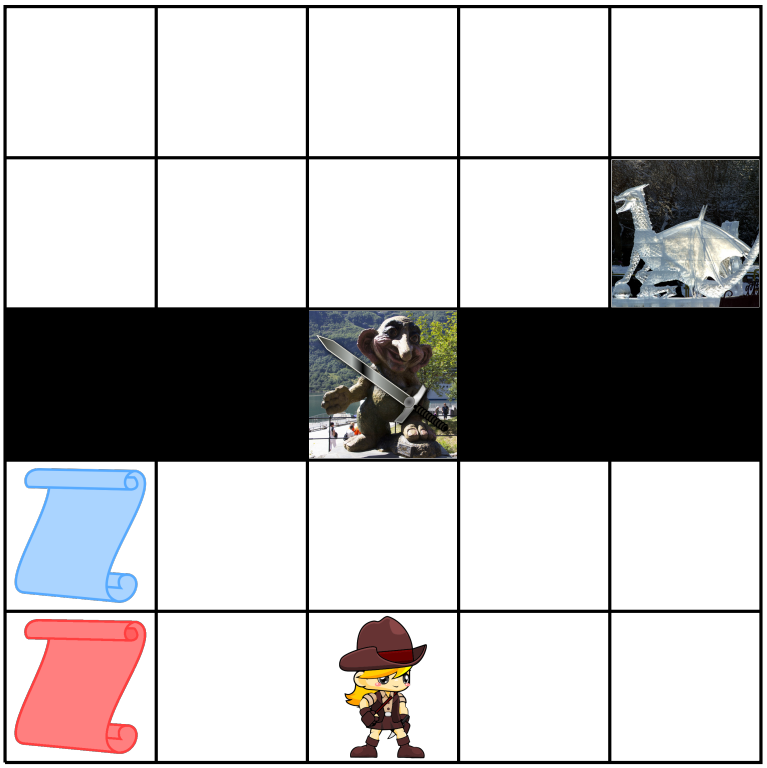


Figure 6.7: Advent

6.4 Advent – A Challenging Task

Here I present an adventure game I call Advent[3], both to present a non-pathological scenario for HOG and to reinforce that it is not a feature that is only theoretically useful, as demonstrated in my **`exact`** Blocks World stress test in Section 6.3. Recall that for **`exact`**, not only did the size of feature conjunctions grow to nearly 50 features, but for each rule searching for a `<block-6>`, Carli-RRL had to verify that there exists no block not already address by `<block-0>` through `<block-5>` (something that could possibly be sped up using an optimization described in Section 7.3.3). In Advent, there are different enemies present in the world that the agent must learn to deal with differently based on their attributes, but there is not always an enemy present in the tile occupied by the agent. The HOG allows the agent to refer to attributes of an enemy when it is present and to learn how to act when there is no enemy present as well. As the number of enemies is restricted to 0-1 and not all features are part of the HOG, this is a much easier HOG scenario for Carli-RRL.

The rules (or features) that enable an agent to refer to attributes of an enemy are implemented using the HOG. In Advent as currently implemented, the HOG grants the agent the ability to

[3] Advent is, of course, a nod to Colossal Cave Adventure.

incorporate multiple attributes of an enemy into its value function. The null-HOG features enable the agent to continue to concern itself with additional features in the absence of an enemy if it has already incorporated the existence of an enemy into its value function. Further, were I to extend Advent to allow multiple enemies in the same room, the HOG would automatically support the introduction of new features for the second enemy with no changes on my part.

Advent presents a 5x5 grid world with a bridge between the North and South halves of the environment and walls on either side of the bridge. The environment presents infinite sources of both fire and ice scrolls that replenish when the player character casts the corresponding spell, removing the scroll from their inventory. There is a troll on the bridge that will drop a magic sword if killed, but can only be reduced below 1 health using the firebolt spell. And there is a water dragon that can only be slain using the magic sword or if it is turned to ice using the icebolt spell. Slaying the dragon terminates the episode.

The player character, troll, and dragon each have 10 health. The player character can move North, South, East, or West as long as there is no enemy present with health greater than 1. They can pick up an item if one is present in the current room. They can attack an enemy with fists (5 damage to the troll, 0 damage to the water dragon, or 2 damage to the ice dragon) or with whatever weapon is currently equipped. They can equip a magic sword (capable of dealing 3 damage to the water dragon or 10 damage to the ice dragon) if it is present in the player character's inventory. And they can cast a healing spell (restoring 5 health) or a firebolt spell or icebolt spell if they have the corresponding fire or ice scroll. The firebolt spell does 5 damage to the troll and is capable of reducing its health to 0 or does 2 damage to the dragon. The icebolt spell does 5 damage to the troll or turns the water dragon to ice.

The player character and the enemy (if present) take turns taking actions with the player character going first. The troll always attacks the player for 2 damage. The dragon always attacks the player for 3 damage.

The actual features available to the agent (which controls the player character) include:

1. `Action(MOVE)`, `Action(ATTACK)`, `Action(TAKE)`, `Action(EQUIP)`, or `Action(CAST)`

2. `Move-direction(NORTH)`, `Move-direction(SOUTH)`, `Move-direction(EAST)`, `Move-direction(WEST)`, or `Move-direction(NONE)` (i.e. a wait or no-op).

3. `Less-than(<player-x>, 3)` (and subsequent refinements of $[0, 3)$ and $[3, 5]$).

4. `Less-than(<player-y>, 3)` (and subsequent refinements of $[0, 3)$ and $[3, 5]$).

5. `Less-than(<player-health>, 5)`
   (and subsequent refinements of $[0, 5)$ and $[5, 10]$).

6. `Has(<player>, MAGIC_SWORD)`

7. `Has(<player>, FIREBOLT_SCROLL)`

8. `Has(<player>, ICEBOLT_SCROLL)`

9. `Equipped(<player>, FISTS)` or `Equipped(<player>, MAGIC_SWORD)`

   And if an enemy is present in the current room:

10. `Less-than(<enemy-health>, 5)`
    (and subsequent refinements of $[0, 5)$ and $[5, 10]$).

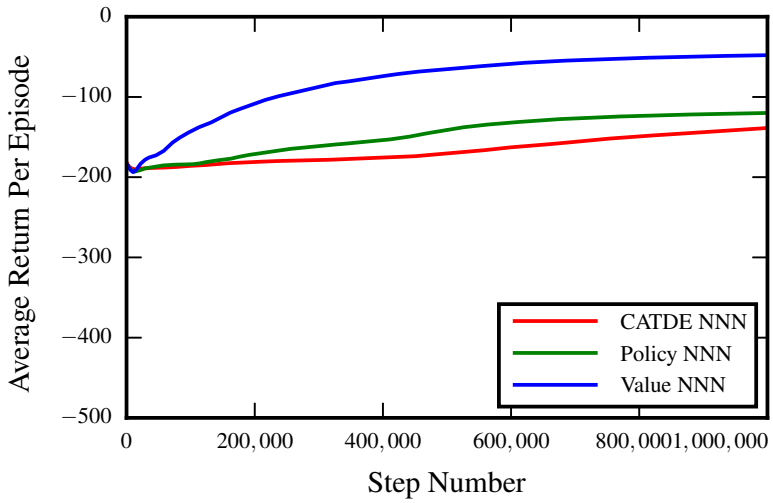
11. `Type(<enemy>, SOLID)` (including ice),
    `Type(<enemy>, TROLL)`, or `Type(<enemy>, WATER)`

This environment is much simpler than Blocks World in terms of the connectivity of the relations. However, it is interesting due to the diversity of relationships and the varied structure of the task. The complex conjunctions of relations describing the environment are essential to allow the agent to understand which actions are suitable for which conditions. A version of the agent that I lesioned to use only a linear combination of features is unable to converge on a policy to solve the task at all. Additionally, I can use my HOG to implement features that depend on the existence of an enemy. Given the low bound of one enemy per room, the computational efficiency issues of the HOG are essentially absent. The HOG enables an agent to reason about scenarios in which an enemy is present and to include those features in its value function while allowing it to continue to improve its value function for cases in which no enemy is present as well.

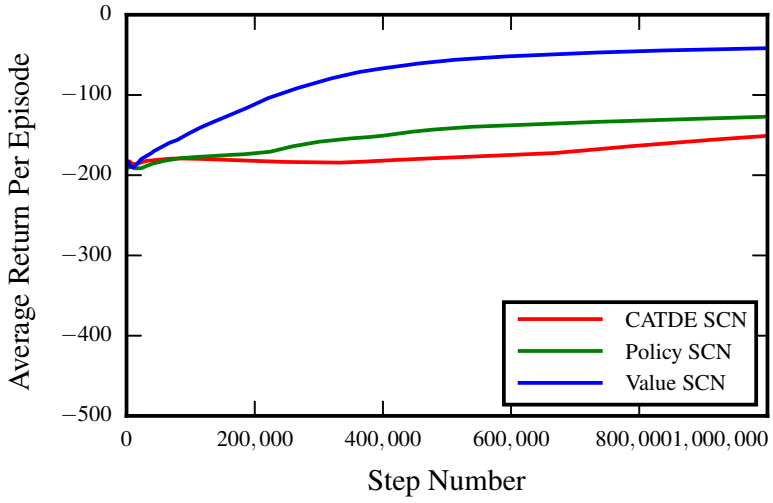
One way of interpreting the success of an RRL in solving this task is that it is able to make up for the lack of Hierarchical Reinforcement Learning (HRL) since it can understand features in different contexts differently. In conjunction with HRL, my agents would probably have had an easier time figuring out that fire spells are effective against trolls and that ice scrolls are effective against water creatures.

Incorporating HRL would be a relatively simple extension to Carli-RRL given its support for independent HTCs in the Rete, but it nevertheless lies outside of the scope of this research. I present success with Advent to demonstrate the efficacy of RRL when compared to linear function approximation, rather than to provide the best possible Advent learner I could implement.

I train for Advent using on-policy Greedy-GQ(λ) with $\alpha = 0.03$, $\rho = 0.01$, $\gamma = 0.99$, and $\lambda = 0.3$ and the value criterion with a 50 step delay for refinement, 100 for unrefinement with boost, and



(a) No Rerefinement



(b) Boost with Concrete

Figure 6.8: ARtPE for Advent

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−139	1.16ms	1,601
Policy Criterion	−120	1.65ms	3,013
Value Criterion	−48.0	1.80ms	2,745

Table 6.2: Agent performance for Advent with no rerefinement

Criterion at $50,000$	ARtPE	WCTPS	# Weights
CATDE	−151	1.47ms	1,757
Policy Criterion	−127	1.95ms	2,657
Value Criterion	−41.7	2.03ms	2,105

Table 6.3: Agent performance for Advent using boost with concrete

200 for concrete. As you can see in Figure 6.8a, of the agents without rerefinement enabled, the one with the value criterion does the best. With boost and concrete to possibly allow the agent to make better decisions with respect to feature selection, depicted in Figure 6.8b, the agent with the value criterion actually learns a bit more slowly with the parameters used. With a parameter sweep on delays for unrefinement and concrete, perhaps this result could be improved. Nevertheless, the agent using Cumulative Absolute Temporal Difference Error (CATDE) improves its performance with boost and concrete while the agent using the policy criterion shows no improvement.

A strictly optimal policy is 18 actions total: collect both scrolls (5 actions), move to encounter the troll (3 actions), attack twice and cast firebolt to kill the troll (3 actions), take and equip the magic sword (2 actions), move to encounter the water dragon (3 steps), and finally cast icebolt and attack to kill the water dragon (2 actions). My RRL agents rarely accomplish this level of performance. Since long term planning and navigation problems are intertwined, my agents have a great deal of difficulty recognizing the value of the ice scroll. By the time one of my agents is capable of solving the task reliably, it is incapable of effectively incorporating the presence of the ice scroll in the player character's inventory into its value function. However, my agents are capable of converging to a policy that achieves the optimal policy were no ice scroll present in the environment.

In summary, my HOG implementation enables these agents to learn to solve this task even in the presence of features that are conditional on a variable that refers to an enemy that may or may not be present in any given room. When an enemy is not present, even after an agent has begun to incorporate attributes of an enemy into its value function, null-HOG features enable my agents to continue to refine its value function in an online, incremental fashion. Additionally, were I to modify Advent to present two enemies at once, my HOG would enable my agents to automatically create a new variable to refer to it along with new relations that refer to it. These capabilities would have previously required complete reconstruction of the value function, as done offline using TILDE [Džeroski *et al.*, 2001].

CHAPTER 7

Summary and Future Work

In this chapter I summarize this thesis (Section 7.1) and its contributions (Section 7.2). I additionally provide some possible directions of future work (Section 7.3).

7.1 Summary

This thesis began with the introduction of the Relational Reinforcement Learning (RRL) domain, Blocks World, and several different objectives. Some of those objectives, **`stack`**, **`unstack`**, and **`on(a,b)`**, had been explored by related work [Džeroski *et al.*, 2001; Irodova and Sloan, 2005]. However, **`exact`** had not yet been explored in the context of RRL due to the significantly higher complexity of the task using traditional relational representations.

I discussed different kinds of knowledge representation including tabular representations, linear combinations of features, tile codings, and of course relational representations.

I initially prototyped my adaptive Hierarchical Tile Coding (aHTC) for Temporal Difference (TD) learning in Soar and used that prototype to explore the basic efficacy of aHTCs. With an eye toward improving computational performance, I shifted to an independent architecture, Carli for Propositional Representations (Carli-Prop), in which I used k-dimensional tries (k-d tries) to reduce the Wall-Clock Time Per Step (WCTPS) of aHTC by two orders of magnitude over my prototype architecture. Along the way I improved the Average Return Per Episode (ARtPE) in Puddle World with better refinement criteria and a stricter aHTC implementation that appears to be superior to a non-hierarchical Adaptive Tile Coding (ATC), confirming Hypothesis 1 and answering Research Question 1, posed earlier in chapter 3.

Upgrading to Carli for Relational Reinforcement Learning (Carli-RRL), I examined the Blocks World domain more closely and determined that Rete could solve many of the problems that using RRL in this domain raised. While similar in some respects to using a k-d trie, using Rete additionally ensured that variables in different conditions matched as needed and solved problems regarding variable numbers of actions without the need for either separate passes per action or

separate value functions for each action. Implementing aHTCs in Rete was significantly more complex due to the higher complexity of the Rete algorithm itself, but also due to the need to provide a rule-based grammar for a Hierarchical Tile Coding (HTC), the need for mechanisms to extract features from the rules, and the significantly higher complexity of extending the value function when refining an aHTC over time. To enable these mechanisms, I devised both the Feature Encoded in the Last Scope Convention (FELSC) and and the Identical Ordering Convention (IOC) for all rules and their corresponding feature conjunctions.

Exploring ARtPE, I investigated Cumulative Absolute Temporal Difference Error (CATDE), a policy criterion, and a value criterion for value function refinement in my relational blocks world domain. These experiments involved the development of approximately optimal policies for the **`exact`** objective and were done with fixed Epsilon-greedy (ϵ-greedy) exploration. Under these conditions, I demonstrated that all three criteria did quite well with aHTCs in the absence of distractors, but learning was slowed significantly by the distractor features. Attempting unrestricted rerefinement, I observed good ARtPE considering the low number of weights but ultimately the agents did not achieve a near optimal policy. Blacklists turned out to be a poor solution to the convergence problem with rerefinement, and I hypothesized that a boost mechanism could be more effective (Hypothesis 5). Boost achieved better ARtPE, but both WCTPS and ARtPE could still be improved, resulting in Hypothesis 6 that my agents could benefit from eventually ceasing rerefinement. Using a combination of unrefinement and rerefinement, a boost mechanism for previously selected features, and a concrete mechanism to eventually disable unrefinement, I arrived at my best results with or without distractors using the value criterion. That this agent dominates the ARtPE of the agents any of the preceding agents confirms both 3 and Hypothesis 6.

It would be fair to criticize the unrefinement, boost, and concrete mechanisms for introducing additional hyperparameters into Carli-RRL, but their utility is not restricted to just the **`exact`** objective of Blocks World. Using these additional techniques has improved the performance of my agents for other tasks in this thesis as well. A goal for future research is to develop a sound method for analyzing both the utility of the learning that would be discarded by doing unrefinement and of the improvement that would result from a rerefinement so that they can be compared. Then an agent could choose to do a rerefinement only in the case that the utility makes it worthwhile. It is unfortunate that the concrete mechanism in particular requires tuning an additional parameter for an agent to maximally benefit from it, but in lieu of a more advanced method as I've described, it is worth using.

I demonstrated a several orders of magnitude WCTPS reduction over Q-RRL by Džeroski *et al.* [2001] for **`stack`**, **`unstack`**, and **`on(a,b)`** objectives in Blocks World using RRL on an actual Sun Ultra 5/270 – the same hardware used by Džeroski *et al.* [2001] – (a speedup factor between 375 and 543) confirming Hypothesis 4 that Rete could be an efficient algorithm for RRL

implementation.

I evaluated the optimality of my approximately optimal policies for the **`exact`** objective of Blocks World as the number of blocks scales up. Then I did a deeper evaluation of transfer in Blocks World and on a Taxicab environment. Using both environments, I demonstrated that learning with simpler problem instances results in effective transfer to more complex problem instances.

Then I delved into a Higher Order Grammar (HOG) for RRL, in which I explored the problem of requiring an unknown number of variables and relations to refer to a class of object. In previous architectures employing Top-down Induction of Logical DEcision trees (TILDE) [Džeroski *et al.*, 2001; Blockeel and Raedt, 1998], this problem was solved implicitly by fully constructing its First Order Logical Decision Trees (FOLDTs) from scratch offline, using a model to map state-action pairs onto most recent Q-values. Making the problem explicit led to my implementation of null-HOG features, enabling my agents to continue to learn about scenarios in which fewer objects are present in the world than the maximum number of variables tested by the architecture. This allowed Carli-RRL to refine its aHTCs in an incremental, model-free fashion even with the HOG. I showed results for **`exact`** objectives in Blocks World using only `on(a,b)` and `goal-on(a,b)` relations, as well as for an adventure game.

7.2 Contributions

In summary, the contributions of this thesis are as follows.

1. The primary contribution of this dissertation is a theory and corresponding algorithms for connecting the rule-matching power of the Rete with the needs of RRL to arrive at computationally efficient methods for implementing the first online, incremental RRL architecture.

   (a) I developed aHTCs, an extension of
   ATCs in which weights are stored in non-leaf nodes in addition to the leaf nodes, and demonstrated the efficacy of using hierarchically organized conjunctions of features in terms of WCTPS and ARtPE.

   (b) I developed a novel approach for embedding an aHTC in a Rete to enable it to take advantage of the features of a Rete that make it suitable for efficient RRL.

2. I provide an implementation of Carli-RRL that achieves a greater than two orders of magnitude WCTPS reduction in Blocks World tasks over prior work by Džeroski *et al.* [2001].

3. To support making Carli-RRL online and incremental, I modified existing ATCs refinement criteria to make them fully incremental and created a new one as well (Cumulative Absolute Temporal Difference Error (CATDE)).

4. I designed, implemented, and evaluated further extensions to aHTC refinement criteria to further increase ARtPE when allowing unrefinement while minimizing the WCTPS cost of doing so

5. I additionally provided an advance in the theory of RRL with my Higher Order Grammar (HOG). It allows agents that learn online and incrementally to introduce new features with new variables while also being able to continue learning about situations in which no objects satisfy the new variables. This is done through the introduction of null-HOG features that represent the concept that it is possible that no i^{th} object exists to satisfy the i^{th} variable.

7.3 Future Work

There are many directions one could go in if one wishes to build on what has been achieved with Carli-RRL. Here I consider several possibilities. First, my HOG implementation in Carli is currently imperfect. I discuss correcting that in Section 7.3.1. Second, incorporating Hierarchical Reinforcement Learning (HRL) into my RRL agents would be interesting and could certainly be considered low hanging fruit. I consider this possibility in Section 7.3.2. Third, I have some ideas about possibly improving the performance of the Rete for RRL, particularly when using a HOG, and I discuss this potential optimization in Section 7.3.3. Fourth, the refinement criteria I evaluate in this thesis are certainly not exhaustive. I consider delving into refinement criteria more deeply in Section 7.3.4. And finally, the problems of architectural and agent evaluation and experimentation in other tasks are unbounded. I consider additional domains in Section 7.3.5.

7.3.1 Higher Order Grammar

The HOG implementation as presented is actually imperfect in that multiple tokens will be passed if multiple combinations of `Symbol`s match the HOG `Variable`s. The fix would be to enclose all features depending on a new variable within its corresponding existential test. This is a simple change to the grammar itself, but modifying the methods for extending conjunctions of features over time to unwrap some number of existentials and rewrap was one correction too many to be implemented in time for this thesis.

7.3.2 Hierarchical Reinforcement Learning

As I mentioned when discussing Advent (Section 6.4), one way of interpreting the success of my agent is that it was able to use RRL to compensate for the lack of HRL – a technique that might be more suitable for this task. Incorporating HRL into Carli-RRL would allow an agent designer

to ensure that RRL generalizes knowledge appropriately, such as fire spells being effective against trolls and ice scrolls being effective against water creatures.

Incorporating HRL would be a relatively simple extension to Carli-RRL given its support for independent HTCs in the Rete. my previous research [Bloch, 2009] employed the MAXQ approach to HRL [Dietterich, 2000; Dietterich, 1998], but another framework such as Options [Sutton *et al.*, 1999] or Hierarchies of Abstract Machiness (HAMs) [Parr and Russell, 1997] would work just as well.

The kind of induction of FOLDTs performed by Q-RRL using TILDE and by Carli-RRL likely has some connection to the problem of automatically inducing hierarchies. There has been some work already in the area of automatic hierarchy induction [Mehta *et al.*, 2008; Ryan, 2002]. Investigating this problem and its possible connections with the automatic induction of FOLDTs would be another interesting research problem to pursue.

7.3.3 Lazy Join Subnetworks for Rete

I made the case for using Rete to implement HTC for RRL in Section 4.2. However, it may be the case that I can do better than using the traditional Rete algorithm.

The standard Rete structure involves alpha nodes that test only one Working Memory Element (WME). The alpha nodes feed into the beta network that consists of join nodes that may be followed by additional nodes that test the set of WMEs that comprise the tokens generated by the join node.

Doorenbos [1995] is credited with having devised an optimization in which join nodes are unlinked from one of their parents (left or right) in the case that one input or the other is emptied. This can result in a significant reduction in traffic through the Rete in the case that there exist many nodes that receive traffic from one side or the other but rarely both. This tends to be a common scenario, and this optimization can result in substantial reductions in computation time.

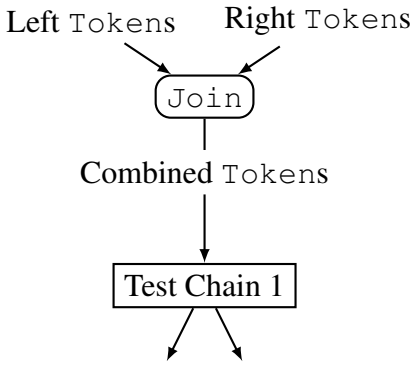


Figure 7.1: A (standard) Join Node

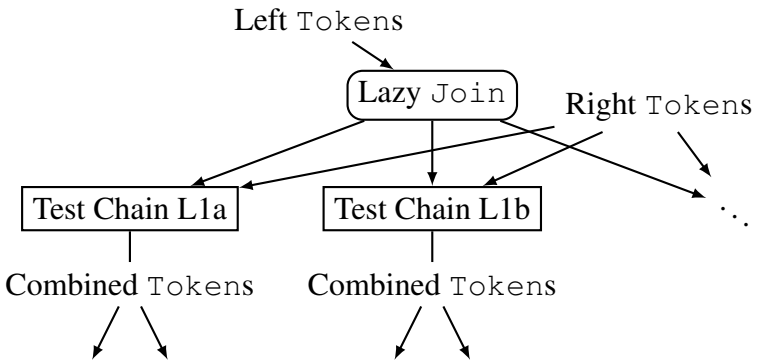


Figure 7.2: A lazy `Join` subnetwork, right unlinking equivalent

I suggest a modification of Rete that was inspired by my analysis of my HOG implementation. Consider the problem of selecting two unique blocks from 10. The Rete and rules as implemented will generate 100 tokens that will then have 10 tokens filtered out by a subsequent test. Selecting a third unique block will result in 900 tokens that will then have 180 tokens filtered out by subsequent tests. This subsequent filtering is necessary since the only thing the join nodes only ensure that variables line up correctly. However, it would be ideal if the join only occurred if the symbols on both sides were compatible given subsequent tests.

Therefore, I propose a kind of reordering of tests to prevent this dual wasted effort of making unnecessary tokens and then filtering them back out. What if, instead of doing any tests between this join and the next, one could do them before? Rather than doing the tests afterward, one could use values in the tokens on the left to generate tests that filter tokens coming from the right in advance of a lazy join. This is analogous to right unlinking but offers the additional advantage of reducing the number of tokens flowing through the Rete. Left unlinking could be done similarly.

While part of the beta network in the traditional view of the Rete, these intermediate nodes would function much like an alpha network, determining which tokens are passed to the join at all. To maximize the value of this intermediate network, their output tokens must be passed to a node that does the join with only the matching tokens from the other input, avoiding any duplicate checking of variable bindings.

There would be some cost to constructing test chains L1a, L2a, and so on on the basis of inputs to the join from the left. However, as the number of them would be bounded by the number of left tokens, the reduction in unnecessary joins can still be a win. When selecting two unique blocks from 10, test chains L1a-L1j would filter duplicate blocks and directly pass on 90 tokens. When selecting a third unique block, test chains L1a-L1cl would directly pass on 720 tokens. As described, one would still have 900 tests, but no unnecessary tokens would be produced.

Collection match [Acharya and Tambe, 1993] could further improve this picture. When selecting two unique blocks from 10 the lone test chain, L1a, could simply pass on 10 collections

of size 9. When selecting a third unique block, each of the ten nodes, L1a-L1j, could pass on 90 collections of size 8. With an ideal implementation of lazy joins with collection match, it would appear that one could reduce the creation of $1,000$ tokens and the filtering out of 190 tokens to the creation of 100 collections. It is still a pathological case since all HOG variables except for one would have size 1 in each collection, but less work would be wasted at each point in the process. With the use of the flyweight pattern, each range could be represented very efficiently, significantly reducing memory usage as well.

7.3.4 Refinement Criteria

Over the course of this thesis, I investigated value function refinement criteria based on **influence** and **variance** [Munos and Moore, 1999a; Munos and Moore, 1999b], value function refinement criteria based on **Bellman error** and **TD error**, a **value criterion**, and a **policy criterion** Whiteson *et al.* [2007]. These criteria were suitable for our agents due to my ability to integrate them into online, incremental algorithms.

However, earlier, model-based work such as Q-RRL using TILDE [Džeroski *et al.*, 2001; Blockeel and Raedt, 1998] and U-trees [McCallum, 1996] use statistical tests such as F-tests and the Kolmogorov-Smirnov test to determine whether there is sufficient statistical significance in the model to merit refinements. These tests are not suitable for a model-free architecture such as Carli-RRL, but they do provide the ability for refinement criteria to determine that refinement is unnecessary (with some probability) – functionality not offered by the criteria I explored in Carli-RRL in this thesis.

Online, incremental refinement criteria that can offer statistical claims about the likelihood of refinement being beneficial would be an interesting direction for future research. Such criteria could be used to implement methods for comparing the utilities of both unrefinement and rerefinement for a more advanced, parameter-free replacement of my concrete mechanism as well.

7.3.5 Domains

It would be interesting to evaluate Carli-RRL in domains that have been explored with other agents that use RRL such as a robotic assembly task [Lang *et al.*, 2012b], Robot Butler [Sridharan and Meadows, 2016b], Simple Mario [Sridharan *et al.*, 2016b], Infinite Mario [Mohan and Laird, 2010], and Tetris [Driessens and Džeroski, 2004]. The agents developed in this related work were not mere RRL agents but used RRL in the context of a larger Artificial Intelligence (AI) architecture, depended on human guidance, or used HRL as well. Regardless, Carli-RRL could be made to learn these tasks and it would be worthwhile to see how it performs, in terms of both WCTPS and ARtPE.

APPENDIX A

Temporal Difference Methods

A.1 Eligibility Traces–Q(λ)

For longer, more complex problems, the single step updates performed by Q-learning can result in extraordinarily long learning times. It is essential to backup temporal difference error multiple steps in order to make these kinds of problems tractable.

Sarsa(λ) [Watkins, 1989] and Q(λ) [Peng and Williams, 1996] incorporate eligibility traces into Sarsa and Q-learning respectively.

Algorithm 5 Q(λ) and Sarsa(λ).

function Q-LAMBDA(s,a,r,s',a')
  $\delta_t \leftarrow r_t + \max_{a^*} \sum_{i=1}^{n} (\gamma \phi_{s',a*}(i)\theta_t(i) - \phi_{s,a}(i)\theta_t(i))$ ▷ Calculate Temporal Difference (TD) error
  UPDATE(δ_t)
  if $a' \neq a^*$ **then** CLEAR-TRACE
function SARSA-LAMBDA(s,a,r,s',a')
  $\delta_t \leftarrow r_t + \sum_{i=1}^{n} (\gamma \phi_{s',a'}(i)\theta_t(i) - \phi_{s,a}(i)\theta_t(i))$ ▷ Calculate TD error
  UPDATE(δ_t)
function UPDATE(δ_t)
  for $s \in \mathcal{S}, a \in \mathcal{A}$ **do** ▷ Actual implementations are more efficient
    $e_t(i) \leftarrow \lambda e_{t-1}(i) + \frac{\phi_{s,a}(i)}{\sum_{i=1}^{n} \phi_{s,a}(i)}$ ▷ Update eligibility
    $\theta_{t+1}(i) \leftarrow \theta_t(i) + \alpha \delta_t e_t(i)$ ▷ Update weights
function CLEAR-TRACE
  for $s \in \mathcal{S}, a \in \mathcal{A}$ **do** ▷ Actual implementations are more efficient
    $e_t(s,a) \leftarrow 0$

Algorithm 6 Greedy-GQ(λ).

function GREEDY-GQ-LAMBDA(s,a,r,s',a')
  $\rho_t \leftarrow \frac{\pi(S_t,A_t)}{b(S_t,A_t)}$ ▷ Importance sampling ratio
  $I(s,a) \leftarrow 1$ ▷ Interest function is 1 for flat Reinforcement Learning (RL) and initiating states when using HRL
  for $i \in 1..n$ **do** ▷ "Clear" the trace and increase eligibility for current features.
    $e_t(i) \leftarrow \rho_t e_{t-1}(i) + I\phi_t(i)$
  $u \leftarrow \sum_{i=1}^{n} w_t(i)e_t(i)$
  $v \leftarrow \sum_{i=1}^{n} w_t(i)\phi_t(i)$
  $\delta_t \leftarrow r_t + \max_{a^*} \sum_{i=1}^{n}(\gamma\phi_{s',a*}(i)\theta_t(i) - \phi_{s,a}(i)\theta_t(i))$ ▷ Calculate TD error
  for $i \in 1..n$ **do** ▷ Update
    $\theta_{t+1}(i) \leftarrow \theta_t(i) + \alpha(\delta e_t(i) - \gamma(1-\lambda)u\phi_{t+1}(i))$
    $w_{t+1}(i) \leftarrow w_t(i) + \alpha\eta(\delta e_t(i) - v\phi_t(i))$
    $e_t(i) \leftarrow \gamma\lambda e_t(i)$

A.2 Greedy-GQ(λ)

Maei and Sutton [2010] introduced a more modern version of Q(λ) called Greedy-GQ(λ). Unlike Q(λ), Greedy-GQ(λ) provides convergence guarantees when using linear function approximation. The computational cost of Greedy-GQ(λ) is still small compared to the total cost of a reinforcement learning agent, and the additional tuning required for good (but not optimal performance) is minor. In practice, it is not guaranteed to perform better than Q(λ), but it is worth using when in doubt.

The key idea of GQ(λ) is to guarantee convergence using a second weight vector in order to minimize Mean-Square Projected Bellman Error (MSPBE) in the limit. It introduces a secondary set of learned weights, $w(i)$, a secondary learning rate or step-size parameter, η, an importance sampling ratio, ρ, and an interest function for Hierarchical Reinforcement Learning (HRL), $I(s,a)$.

APPENDIX B

Incremental Calculation of a Mean and Standard Deviation

Given a set of numbers, calculating a sample mean and standard deviation as a batch is straightforward. The equation for a sample mean is:

$$\mu_N = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{B.1}$$

The equation for sample variance is

$$\sigma_N^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\mu_N - x_i)^2 \tag{B.2}$$

The sample standard deviation is simply a square root of the sample variance.

Incremental calculation is more involved.

B.1 Incremental Mean Calculation

The mean of an empty set is defined, $\mu_0 = 0$. To insert the n^{th} value, x_i:

$$\mu' \leftarrow \frac{n-1}{n}\mu + \frac{1}{n}x_i \tag{B.3}$$

To update the mean when x_i is updated to x_i':

$$\mu' \leftarrow \mu + \frac{1}{n}(x_i' - x_i) \tag{B.4}$$

To remove x_i from the n values:

$$\mu' \leftarrow \frac{n}{n-1}\mu - \frac{1}{n-1}x_i \tag{B.5}$$

B.2 Incremental Variance Calculation

Welford [1962] and Knuth [1997] provide a method to incrementally calculate a sample variance as values are added:

Algorithm 7 Incremental variance insertion.

Ensure: $\nu_0 = 0$ and $\sigma_0^2 = 0$
Require: A new value x_i to be inserted
1: **Apply** Equation B.3 for μ' ▷ Mean insertion
2: $V_i' \leftarrow (\mu' - x_i)^2$ ▷ My modification for algorithms 8 and 9
3: $\nu' \leftarrow \nu + V_i'$
4: **if** $n > 1$ **then**
5: $\quad \sigma'^2 = \frac{\nu'}{n-1}$

I provide an update method:

Algorithm 8 Incremental variance update.

Require: A new value x_i' to replace x_i
1: **Apply** Equation B.4 for μ' ▷ Mean update
2: $V_i' \leftarrow (\mu' - x_i')^2$
3: $\nu' \leftarrow \nu + V_i' - V_i$
4: **if** $n > 1$ **then**
5: $\quad \sigma'^2 = \frac{\nu'}{n-1}$

I provide a reverse method as well:

Algorithm 9 Incremental variance removal.

Require: An existing value x_i to be removed
1: **Apply** Equation B.5 for μ' ▷ Mean removal – actually unused below
2: $\nu' = \nu - V_i$
3: **if** $n > 1$ **then**
4: $\quad \sigma'^2 = \frac{\nu'}{n-1}$
5: **else**
6: $\quad \sigma'^2 = 0$

Note that when using these algorithms to estimate variance for weights in an architecture using linear function approximation (Section 1.3.3 on page 17), one may wish to replace line 2 of

Algorithm 7 and line 2 of Algorithm 8 with $V_i' \leftarrow \frac{(\mu' - x_i')^2}{c}$. c should refer to an average or moving average of the credit assignment for the weight. Without this modification, variance estimates will incorrectly shrink as the credit assigned to a given tile decreases. This is potentially a significant problem for an architecture employing Hierarchical Tile Coding (HTC) if it uses this variance calculation as part of a refinement criterion.

BIBLIOGRAPHY

[Acharya and Tambe, 1993] Anurag Acharya and Milind Tambe. Collection oriented match. In *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, pages 516–526, New York, NY, USA, 1993. ACM. 134

[Aeronautiques *et al.*, 1998] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl| the planning domain definition language. 1998. 1

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 268–272. AAAI Press, 1987. 19

[Albus *et al.*, 1971] James S. Albus, Datu Techltiques Branch, Communicated Donald, and H. Perkel. A theory of cerebellar function, 1971. 20, 21

[Albus, 1981] James Sacra Albus. *Brains, Behavior and Robotics*. McGraw-Hill, Inc., New York, NY, USA, 1981. 20, 21

[Bellman, 1957a] Richard Bellman. A markovian decision process. Technical report, DTIC Document, 1957. 1

[Bellman, 1957b] Richard Ernest Bellman. Dynamic programming, 1957. 1, 2, 20

[Bloch, 2009] Mitchell Keith Bloch. Hierarchical reinforcement learning in the taxicab domain. Technical Report CCA-TR-2009-02, Ann Arbor, MI: Center for Cognitive Architecture, University of Michigan, 2260 Hayward Street, 2009. 133

[Bloch, 2011] Mitchell Keith Bloch. Off-Policy hierarchical reinforcement learning. arXiv:cs.LG/1104.5059, 2011. 46

[Bloch, 2018] Mitchell Keith Bloch. Number of ways of converting one set of lists containing n elements to another set of lists containing n elements by removing the last element from one of the lists and either appending it to an existing list or treating it as a new list., 2018. https://oeis.org/A300159 accessed on 2018-3-5. 16

[Blockeel and De Raedt, 1998] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, May 1998. 28

[Blockeel and Raedt, 1998] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1):285 – 297, 1998. 84, 131, 135

[Broomhead and Lowe, 1988] David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988. 17

[CHENOWETH, 1991] SV CHENOWETH. On the np-hardness of blocks world. In *Proc. of AAAI-91*, 1991. 105

[Commons, 2016] Wikimedia Commons. File:thomas eakins - baby at play.jpg — wikimedia commons, the free media repository, 2016. [Online; accessed 24-October-2017].

[Davies, 1996] Scott Davies. Multidimensional triangulation and interpolation for reinforcement learning, 1996. 44

[Dietterich, 1998] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *ICML*, pages 118–126, 1998. 108, 133

[Dietterich, 2000] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Intell. Res. (JAIR)*, 13:227–303, 2000. 133

[Doorenbos, 1995] Robert B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Pittsburgh, PA, USA, 1995. UMI Order No. GAX95-22942. 60, 63, 133

[Driessens and Džeroski, 2004] Kurt Driessens and Sašo Džeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304, Dec 2004. 135

[Driessens *et al.*, 2001] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Machine Learning: ECML 2001*, pages 97–108. Springer, 2001. 107

[Džeroski *et al.*, 2001] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001. 2, 4, 6, 11, 12, 28, 30, 31, 33, 55, 84, 85, 100, 101, 103, 104, 108, 114, 115, 116, 121, 128, 129, 130, 131, 135

[Finney *et al.*, 2002] Sarah Finney, Natalia H. Gardiol, Leslie Pack Kaelbling, and Tim Oates. The thing that we tried didn't work very well: Deictic representation in reinforcement learning. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 154–161, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 30, 31

[Forgy and McDermott, 1977a] C. Forgy and J. McDermott. Ops: A domain-independent production system language. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'77, pages 933–939, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc. 61

[Forgy and McDermott, 1977b] Charles Forgy and John P McDermott. Ops, a domain-independent production system language. In *IJCAI*, volume 5, pages 933–939, 1977. 32, 60

[Forgy, 1979] Charles Lanny Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Pittsburgh, PA, USA, 1979. AAI7919143. 57

[Geramifard *et al.*, 2011] Alborz Geramifard, Finale Doshi, Josh Redding, Nicholas Roy, and Jonathan P. How. Online discovery of feature dependencies. In Lise Getoor and Tobias Scheffer, editors, *ICML*, pages 881–888. Omnipress, 2011. 40

[Grzes and Kudenko, 2008] Marek Grzes and Daniel Kudenko. Multigrid reinforcement learning with reward shaping. In *ICANN (1)*, pages 357–366, 2008. 50

[Grzes, 2010] M. Grzes. *Improving exploration in reinforcement learning through domain knowledge and parameter analysis*. PhD thesis, University of York, 2010. 50

[Irodova and Sloan, 2005] Marina Irodova and Robert H Sloan. Reinforcement learning and function approximation. In *FLAIRS Conference*, pages 455–460, 2005. 4, 6, 11, 12, 30, 31, 55, 101, 129

[Kanerva, 1988] Pentti Kanerva. *Sparse distributed memory*. MIT Press, 1988. 21

[Karp, 1972] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. 30

[Knuth, 1997] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, November 1997. 139

[Laird and Newell, 1983] John E. Laird and Allen Newell. A universal weak method: Summary of results. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'83, pages 771–773, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc. 61

[Laird, 2012] John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, 2012. 35, 60

[Lang *et al.*, 2012a] Tobias Lang, Marc Toussaint, and Kristian Kersting. Exploration in relational domains for model-based reinforcement learning. *J. Mach. Learn. Res.*, 13(1):3725–3768, December 2012. 2

[Lang *et al.*, 2012b] Tobias Lang, Marc Toussaint, and Kristian Kersting. Exploration in relational domains for model-based reinforcement learning. *Journal of Machine Learning Research*, 13(Dec):3725–3768, 2012. 135

[Langley, 1995] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. 28

[Maei and Sutton, 2010] Hamid Reza Maei and Richard S Sutton. Gq (λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, volume 1, pages 91–96, 2010. 137

[Martínez *et al.*, 2017] David Martínez, Guillem Alenyà, and Carme Torras. Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, 247(Supplement C):295 – 312, 2017. Special Issue on AI and Robotics. 2

[McCallum, 1996] Andrew Kachites McCallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, 1996. AAI9618237. 25, 44, 89, 135

[Mehta *et al.*, 2008] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas G. Dietterich. Automatic discovery and transfer of MAXQ hierarchies. In *ICML*, pages 648–655, 2008. 133

[Mohan and Laird, 2010] Shiwali Mohan and J Laird. Relational reinforcement learning in infinite mario. *Ann Arbor*, 1001:48109–2121, 2010. 135

[Moore and Atkeson, 1995] Andrew W. Moore and Christopher G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Mach. Learn.*, 21(3):199–233, December 1995. 23, 41

[Moore, 1991] Andrew Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, Robotics Institute, Carnegie Mellon University, March 1991. 49

[Munos and Moore, 1999a] Remi Munos and Andrew Moore. Influence and variance of a markov chain: Application to adaptive discretization in optimal control. In *IEEE Conference on Decision and Control*, volume 2, pages 1464 – 1469, December 1999. 23, 37, 44, 89, 135

[Munos and Moore, 1999b] Remi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning Journal*, 1999. 23, 25, 47, 135

[Nason and Laird, 2004] Shelley Nason and John E. Laird. Integrating reinforcement learning with soar. In *ICCM*, pages 208–213, 2004. 35, 61

[Parr and Russell, 1997] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, 1997. 133

[Peng and Williams, 1996] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In *Machine Learning*, pages 226–232. Morgan Kaufmann, 1996. 136

[Reynolds, 1999] Stuart I. Reynolds. Decision boundary partitioning: Variable resolution model-free reinforcement learning, 1999. 23, 41, 44

[Ryan, 2002] Malcolm R. K. Ryan. Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *Proceedings of The 19th International Conference on Machine Learning*, pages 522–529. Morgan Kaufmann, 2002. 133

[Sherstov and Stone, 2005] Alexander A. Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In J.-D. Zucker and I. Saitta, editors, *SARA 2005*, volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 194–205. Springer Verlag, Berlin, 2005. 23

[Sondik, 1971] Edward Jay Sondik. The optimal control of partially observable markov processes. Technical report, DTIC Document, 1971. 1

[Sridharan and Meadows, 2016a] M. Sridharan and B. Meadows. Should i do that? using relational reinforcement learning and declarative programming to discover domain axioms. In *2016 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 252–259, Sept 2016. 2

[Sridharan and Meadows, 2016b] Mohan Sridharan and Ben Meadows. Should i do that? using relational reinforcement learning and declarative programming to discover domain axioms. pages 252–259, 09 2016. 135

[Sridharan *et al.*, 2016a] Mohan Sridharan, Prashanth Devarakonda, and Rashmica Gupta. *Can I Do That? Discovering Domain Axioms Using Declarative Programming and Relational Reinforcement Learning*, pages 34–49. Springer International Publishing, Cham, 2016. 2

[Sridharan *et al.*, 2016b] Mohan Sridharan, Prashanth Devarakonda, and Rashmica Gupta. *Can I Do That? Discovering Domain Axioms Using Declarative Programming and Relational Reinforcement Learning*, pages 34–49. Springer International Publishing, Cham, 2016. 135

[Sridharan *et al.*, 2017] Mohan Sridharan, Ben Meadows, and Rocío Gómez. What can I not do? towards an architecture for reasoning about and learning affordances. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, pages 461–470, 2017. 2

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998. 4

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999. 133

[Sutton, 1988] Richard S. Sutton. Learning to predict by the methods of temporal differences. In *MACHINE LEARNING*, pages 9–44. Kluwer Academic Publishers, 1988. 1, 3

[Sutton, 1996] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press, 1996. 27, 41

[Watkins, 1989] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989. 3, 4, 101, 136

[Welford, 1962] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962. 139

[Whiteson *et al.*, 2007] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Adaptive tile coding for value function approximation, 2007. 23, 25, 26, 38, 135

[Wieder, 2005] Thomas Wieder. Number of "sets of lists": number of partitions of 1,...,n into any number of lists, where a list means an ordered subset, 2005. https://oeis.org/A000262 accessed on 2017-8-24. 14

[Zheng *et al.*, 2006] Yu Zheng, Siwei Luo, and Ziang Lv. Control double inverted pendulum by reinforcement learning with double cmac network. *Pattern Recognition, International Conference on*, 4:639–642, 2006. 50