

Preface

This thesis documents the work done by Carl Christian Sloth Andersen at Aalborg University, Department of Computer Science, during the spring semester of 2005.

I would like to thank Uffe Kjærulff for supervising the project. I would also like to thank Hendrik Blockeel for letting me use the ACE data mining system, as well as Jan Ramon, Tom Croonenborghs and Jan Struyf for taking their time to answer my questions regarding the system.

A summary of the thesis can be found in Appendix A.

Aalborg, June 17, 2005

Carl Christian Sloth Andersen

Contents

1	Introduction	1
1.1	Problem Representation	1
1.2	Outline of the Report	2
1.3	Summary of Results	2
2	Reinforcement Learning	3
2.1	Reinforcement Learning	3
2.1.1	Ordered Policies	5
2.2	Learning the Q -Function	5
2.3	Problem Domains	7
2.3.1	The Taxi Domain	7
2.3.2	The Blocks World Domain	8
2.4	Scaling of Reinforcement learning	9
2.5	Value Prediction	10
2.6	Generalizing Policies to New Environments	11
2.7	Summary	12
3	Relational Reinforcement Learning	13
3.1	First-Order Predicate Logic	13
3.2	Relational Domains	14
3.3	Logical Representation of Policies	16
3.3.1	Background Knowledge	17
3.3.2	The Policy Function	18
3.4	Learning Logical Policies	19
3.4.1	Induction of Logical Decision Trees	23
3.4.2	Finding Test Candidates	24

3.4.3	Example Testing	25
3.4.4	Quality Heuristics	26
3.5	Experimental Evaluation of Relational Reinforcement Learning	26
3.6	Recent Work	31
3.7	Summary	31
4	Hierarchical Reinforcement Learning	33
4.1	Task Decomposition	34
4.1.1	Semi-Markov Decision Process	35
4.1.2	Definition of a Subtask	36
4.1.3	Hierarchical Policies	37
4.2	Hierarchical Semi-Markov Q -Learning	38
4.3	MAXQ Value Function Decomposition	39
4.3.1	Definition of the Value Function Decomposition	41
4.3.2	MAXQ Graphs	42
4.3.3	Different Kinds of Optimality	44
4.3.4	The MAXQ-Q Learning Algorithm	46
4.4	State Abstractions	50
4.4.1	Irrelevant Variable Elimination	51
4.4.2	Funnel Actions	52
4.4.3	Structural Constraints	53
4.4.4	Overview of State Abstractions in the Taxi Domain	53
4.5	Non-Hierarchical Execution of a Hierarchical Policy	54
4.6	Hierarchical Exploration Problem	55
4.7	Experimental Evaluation of the MAXQ Method	56
4.7.1	Performance of MAXQ Learning	57
4.7.2	Encoding of Knowledge	59
4.8	Related Work	60
4.9	Summary	60
5	Combining Hierarchical and Relational Reinforcement Learning	61
5.1	MAXQ Hierarchy for Blocks World	61
5.1.1	Hierarchical Exploration Problem	62
5.2	Value and Completion Trees	63
5.3	State Abstractions	64

5.3.1	Manual and Semi-Automatic State Abstraction	67
5.4	The Policy Function	68
5.4.1	Local P -Trees	68
5.4.2	Global P -Tree	70
5.5	Experiments	71
5.5.1	Hand-Coded Logical State Abstractions	72
5.5.2	Flat Relational Reinforcement Learning	73
5.5.3	MAXQ Hierarchy with Logical Decision Trees	74
5.6	Automatically Constructed Hierarchies	75
5.7	Related Work	77
5.8	Summary	77
6	Conclusion	79
6.1	Relational Reinforcement Learning	79
6.2	Hierarchical Reinforcement Learning	79
6.3	Combining Relational and Hierarchical Reinforcement Learning	80
6.4	Summary of Contributions	80
6.5	Future Work	81
A	Summary	83
B	ACE Blocks World Specification	85
B.1	Background Knowledge	85
B.2	TILDE-RT Settings for Inducing Q -trees	85
B.3	TILDE Settings for Inducing P -trees	86
C	Relational MAXQ-Q Learning Algorithm	89
C.1	Relational MAXQ-Q	89
C.2	Learning Local P -trees	89

Chapter 1

Introduction

Reinforcement Learning is the task of teaching an agent optimal behavior in its environment by reinforcing good actions with rewards and poor actions with penalties. At any point in time, the environment is in a specific state, and the agent is given a selection of actions to choose from. The chosen action moves the environment from its current state to a new state dictated by a transition probability distribution. Depending on the chosen action, the agent is rewarded or penalized. Since the choice of action alters the environment, it also directly affects all subsequent rewards. The primary characteristics of reinforcement learning are trial-and-error and delayed rewards.

Currently, there are three main approaches utilized in solving reinforcement learning problems. These are dynamic programming, Monte Carlo methods and temporal-difference learning. This work focuses on Q-learning (Watkins, 1989) and SARSA (Rummery and Niranjan, 1994), which are two temporal-difference learning algorithms. It is a well-known fact that both algorithms produce optimal control policies given the restriction of infinite exploration.

1.1 Problem Representation

As problems grow larger, representation becomes an increasingly important issue. Many real-world problems and their solutions (i.e. a control policies) are often impossible to represent directly in a conventional table-based manner. This has given rise to various approaches to ease the problem of a large state space. In general, the state space is either reduced by the use of state abstractions, or the agent is guided on the right path (thus avoiding a possibly large part of the state space). In this report we explore two of these approaches, namely relational reinforcement learning (Džeroski, Raedt and Driessens, 2001) and hierarchical reinforcement learning using the MAXQ value function decomposition (Dietterich, 2000).

The idea behind relational reinforcement learning is to combine traditional Q -learning with inductive logic programming and relational state descriptions. This combination makes it possible to obtain state abstractions through generalization of the state space. At the time of writing, there exists no proofs of convergence for relational reinforcement learning, but the topic has been receiving an increasingly amount of attention. There are, however, empirical results (although primarily on toy problems) indicating the feasibility of the approach (Džeroski et al., 2001; Driessens, Ramon and Blockeel, 2001; Driessens and Džeroski, 2004).

Hierarchical reinforcement learning using the MAXQ value function builds upon the principle of earlier hierarchical approaches (Hauskrecht, Meuleau, Kaelbling, Dean and Boutilier, 1998; Parr, 1998). Besides being able to learn a control policy for a procedural decomposition of a primary task, the method also decomposes the representation of the learned control policy (i.e. the value function). The decomposition of the value function creates the opportunity for further state abstractions that would otherwise be impossible. The method comes with theoretical guarantees of convergence also proven by Dietterich (2000).

Besides the re-exploration of these two existing methods, the major contribution of this work is to explore the advantages of combining the methods. That is, we investigate the possibilities of integrating inductive logic programming into hierarchical reinforcement learning. We do so within the boundaries of the already existing theory for the two methods.

1.2 Outline of the Report

The outline of the report is as follows: Chapter 2 covers the basics of traditional reinforcement learning using the Q -learning and SARSA algorithms. The two example domains used throughout the report, the Blocks World domain and the Taxi domain, are furthermore introduced in this chapter. Chapter 3 explores the method of relational reinforcement learning and concludes on the advantages of the method through experiments. In Chapter 4 we describe hierarchical reinforcement learning using the MAXQ value function decomposition. This method is also evaluated through experiments. Finally, Chapter 5 introduces the possibilities of combining relational and hierarchical reinforcement learning. We introduce various approaches towards the combination and concludes on their performance through a series of experiments.

1.3 Summary of Results

Combining inductive logic programming with hierarchical reinforcement learning creates the opportunity for applying logical state abstractions to a task hierarchy. These abstractions can be applied manually or can be found semi-automatically through the induction of logical decision trees. Since subtasks in a hierarchy are simpler than their ancestor tasks, patterns of optimality are more easily found by relational reinforcement learning. This results in both faster convergence and smaller space requirements for the learned control policy. This is the first result of this work.

The second result is the observation that automatically constructed task hierarchies also need some means of automatically detecting possible state abstractions. We show that logical decision trees are indeed a powerful tool for this purpose.

Chapter 2

Reinforcement Learning

Reinforcement Learning is the task of teaching an agent optimal behavior in its environment by reinforcing good actions with rewards and poor actions with penalties. At any point in time, the environment is in a specific state, and the agent is given a selection of actions to choose from. The chosen action moves the environment from its current state to a new state dictated by a transition probability distribution. Depending on the chosen action, the agent is rewarded or penalized. The environment of the agent is most often represented as a Markov decision process.

In Section 2.1, we will setup a notation for describing an environment as a Markov decision process. We will furthermore describe a value and action-value function that assigns a numerical value to each state and state/action pair in the environment. Given any of these two functions, an optimal policy for a domain can easily be derived. Section 2.2 describes how the action-value function can be learned using Q -learning or SARSA. Following, Section 2.3 introduces two domains commonly used in reinforcement learning: the Taxi domain and the Blocks World domain. Using the Blocks World domain, Section 2.4 and 2.5 discusses limitations of reinforcement learning, including issues regarding scaling and value prediction of unobserved state/action pairs.

2.1 Reinforcement Learning

Reinforcement learning is teaching an agent optimal behavior in its environment simply by reinforcing its actions with rewards and penalties. The general components of a reinforcement learning problem are an agent and its environment. The agent interacts with the environment in a sequence of discrete time steps $t = \{0, 1, 2, 3, \dots\}$. In each time step, the agent observes the state of the environment and chooses an action to perform. As a result of the chosen action, the state of the environment is updated, and the agent receives a numerical reward (or penalty) stating the quality of its choice of action. The environment is most often represented as a Markov decision process:

Definition 1 (MDP). *A Markov Decision Process (MDP) is a process defined by a 5-tuple $\langle S, A, T, R, T_0 \rangle$:*

- S : the set of states of the environment. A state $s \in S$ is a value assignment to all existing state variables.

- A : the set of actions. $A(s)$ denotes the set of available actions in state $s \in S$.
- P : the transition probability distribution, where $p(s'|s, a)$ is the probability of observing state $s' \in S$ after performing action $a \in A(s)$ in state $s \in S$.
- R : the reward function, where $R(s'|s, a)$ is the real-valued reward given to an agent when observing state $s' \in S$ after performing action $a \in A(s)$ in state $s \in S$.
- P_0 : the initial state probability distribution. $P_0(s)$ denotes the probability of starting in state $s \in S$.

A solution to an MDP is a policy $\pi(s, a)$ that maps each state $s \in S$ to a corresponding probability distribution of the possible actions $a \in A(s)$. The optimal solution to an MDP, denoted π^* , is a policy that maximizes the expected cumulative reward given a horizon. There may exist several optimal policies for an MDP.

Given a policy π , each state can be assigned a number representing the numerical value of starting in that state, and thereafter following policy π . This is achieved by the value function $V^\pi : S \rightarrow \mathcal{R}$, which can be defined as

$$V^\pi(s_0) = E \left[\sum_{i=0}^H \gamma^i R(s_i, a_i) \right] \quad (2.1)$$

where H is the number of steps in the horizon, and γ is the discount factor, which determines the weight put on future rewards. A distinction is normally made between episodic and continuous tasks. An episodic task is restarted every time a terminating state is encountered, while a continuous task runs forever. So, for a continuous task with a infinite horizon, $H = \infty$ and $0 \leq \gamma < 1$. For episodic tasks with a finite horizon and at least one absorbing reward-free state, H is known and γ is usually set to 1. An absorbing reward-free state is a state in which all transitions lead back to the same state with a reward of zero. These states are a way of unifying the notation of episodic and continuous tasks, since it theoretically makes episodic tasks continuous (Sutton and Barto, 1998).

The value function satisfies the Bellman equation for a fixed policy:

$$V^\pi(s) = \sum_{s'} P(s'|s, \pi(a)) [R(s'|s, \pi(a)) + \gamma V^\pi(s')] \quad (2.2)$$

which states that the value of a state s , given a policy π , is the sum of the immediate reward of performing the action $\pi(s)$ and the discounted value of the following state s' . Since there may exist several s' given the specific action, the expected value is calculated by weighting $R(s'|s, \pi(a))$ and $\gamma V^\pi(s')$ with the probability of observing each possible s' . The optimal value function V^* is the value function that maximizes the expected cumulative reward for all states in S . The optimal value function is the fixed point of the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s, a) [R(s'|s, a) + \gamma V^\pi(s')] \quad (2.3)$$

Similar to the value function, an action-value function $Q(s, a)$ can be defined. This function also satisfies the Bellman equation and denotes the value of performing action a in state s . The optimal Q -function, written $Q^*(s, a)$, is the fixed point of the equation:

$$Q^*(s, a) = \sum_{s'} P(s'|s, a) \left[R(s'|s, a) + \gamma \max_{a'} Q^*(s', a') \right] \quad (2.4)$$

Using the Q -function, an optimal action a in state s is an action that maximizes $Q(s, a)$:

$$\pi^*(s) = \arg \max_a Q(s, a) \quad (2.5)$$

This is an important equation, because it illustrates that if an agent learns the Q -function, it does not need to learn neither the reward function R or the transition probability distribution P . Algorithms following this principle are therefore referred to as model-free learning techniques.

2.1.1 Ordered Policies

For a given MDP, there exists only one optimal action-value function. However, as mentioned, there might exist several optimal policies for an MDP. These policies differ in the actions chosen, when several actions in a state have the same highest Q value. If two actions have the same value in state s , i.e. $Q(s, a_1) = Q(s, a_2)$, then neither of them is preferred over the other. To solve this issue, an anti-symmetric transitive action relation ω can be defined as

$\omega(a_1, a_2)$ is **true** iff action a_1 is preferred to action a_2 in all states

This establishes an action ordering such that, if $\omega(a_1, a_2)$, then action a_1 is chosen when $Q(s, a_1) = Q(s, a_2)$. A policy following an ordering ω is denoted π_ω , and is said to be ordered. There exists only one optimal ordered policy π_ω for any MDP.

2.2 Learning the Q -Function

The idea behind temporal-difference (TD) learning is to continuously create approximations of V or Q based on earlier approximations until convergence is achieved. This is very similar to dynamic programming (DP), but where DP needs a perfect model of the environment, TD learning does not, and is therefore a model-free learning technique. In practice, the approximations are most often created over a single time step, but can in theory be made over any number of steps. In fact, Monte-Carlo methods are actually a special case of TD learning, where the approximations are created over all observed steps.

In this report, we will make use of the two very similar TD algorithms Q -learning (Watkins, 1989) and SARSA (Rummery and Niranjan, 1994). The objective of both algorithms is to learn the Q -function by continuously making new approximations. The algorithms are shown in Table 2.1 and Table 2.2 respectively.

\hat{Q} denotes the current approximation of the real Q -function. The learning factor $0 \leq \alpha_t \leq 1$ is a number that indicates how much weight should be put on new observations. It is often a function of the current state and action at time step t :

$$\alpha_t(s_t, a_t) = \frac{1}{1 + \text{numberOfVisits}(s_t, a_t)} \quad (2.6)$$

```
1: For each  $s, a$  initialize the table entry  $\hat{Q}_t(s, a)$  to zero
2: Observe the current state  $s_t$ 
3: while ( $s_t$  is not an absorbing reward-free state) do
4:   Select an action  $a_t$  in state  $s_t$  using exploration policy  $\pi_e$  and execute it
5:   Receive immediate reward  $r$ 
6:   Observe the new state  $s_{t+1}$ 
7:   Update the table entry for  $\hat{Q}_{t+1}(s_t, a_t)$  as follows:
8:      $\hat{Q}_{t+1}(s_t, a_t) := (1 - \alpha_t)\hat{Q}_t(s_t, a_t) + \alpha_t [r + \gamma \max_{a_{t+1}} \hat{Q}_t(s_{t+1}, a_{t+1})]$ 
9:      $s_t \leftarrow s_{t+1}$ 
10: end while
```

Table 2.1: The Q -learning algorithm.

```
1: For each  $s, a$  initialize the table entry  $\hat{Q}_t(s, a)$  to zero
2: Observe the current state  $s_t$ 
3: Select an action  $a$  in state  $s_t$  using exploration policy  $\pi_e$ 
4: while ( $s_t$  is not an absorbing reward-free state) do
5:   Execute action  $a_t$ 
6:   Receive immediate reward  $r$ 
7:   Observe the new state  $s_{t+1}$ 
8:   Select an action  $a_{t+1}$  in state  $s_{t+1}$  using exploration policy  $\pi_e$ 
9:   Update the table entry for  $\hat{Q}_{t+1}(s_t, a_t)$  as follows:
10:     $\hat{Q}_{t+1}(s_t, a_t) := (1 - \alpha_t)\hat{Q}_t(s_t, a_t) + \alpha_t [r + \gamma \hat{Q}_t(s_{t+1}, a_{t+1})]$ 
11:     $s_t := s_{t+1}$ 
12:     $a_t := a_{t+1}$ 
13: end while
```

Table 2.2: The SARSA algorithm.

To enhance readability, the dependency on the s_t and a_t are often omitted in the notation.

At each time step, an exploration policy π_e can be derived from the approximated Q -function combined with a exploration technique. A widely used technique is Boltzmann exploration which assigns a probability to each possible action in a state based on a so-called temperature variable. Boltzmann exploration is defined as

$$P(a_i|s) = \frac{T^{-\hat{Q}(s,a_i)}}{\sum_j T^{-\hat{Q}(s,a_j)}} \quad (2.7)$$

where $P(a_i|s)$ is the probability of selecting action a_i given state s , and $T > 0$ is the temperature stating the weight put on exploration. As T approaches 1, the exploration policy becomes more and more random. As T approaches 0, the policy becomes greedy with respect to the Q values of the respective actions.

The difference between Q -learning and SARSA lies in how the current policy is used. Q -learning is said to be off-policy because it separates the current policy from the update of the approximation \hat{Q} . When the approximation of Q is updated, the action a_{t+1} in the next time step is predicted to be the action that maximizes the approximated Q -function. SARSA instead chooses a_{t+1} using the current exploration policy, and is therefore an on-policy algorithm¹.

Both Q -learning and SARSA will converge to the optimal action-value function if the agent follows an exploration policy that performs every action in every state infinitely often, and if the sequence of α_t values satisfy

$$\lim_{T \rightarrow \infty} \sum_{t=1}^T \alpha_t = \infty \quad \text{and} \quad \lim_{T \rightarrow \infty} \sum_{t=1}^T \alpha_t^2 < \infty \quad (2.8)$$

Furthermore, if a fixed exploration policy is used to select actions, SARSA will converge to the action-value function of that policy (Jaakkola, Jordan and Singh, 1994; Jaakkola et al., 1994).

2.3 Problem Domains

Throughout the report we will use two domains: the highly hierarchical Taxi domain, and the highly relational Blocks World domain. The Taxi domain was used in the introduction of the hierarchical MAXQ value function decomposition by Dietterich (2000), and is a good example of the benefits of this approach. Similarly, the Blocks World domain was used in the introduction of relational reinforcement learning by Džeroski et al. (2001) because of its relational qualities. Both domains are episodic.

2.3.1 The Taxi Domain

The Taxi domain consists of a 5-by-5 grid with four specially-designated locations marked as R(red), B(blue), G(green) and Y(yellow). Initially, a taxi is placed in a randomly chosen

¹The name SARSA comes from the one-step update tuple $(s_t, a_t, r, s_{t+1}, a_{t+1})$

square. One of the four locations is chosen randomly to contain a passenger, and another as the destination. The taxi must go to the location of the passenger, pick up the passenger, go to the destination, and then put down the passenger in the fewest possible steps. Figure 2.1 illustrates the domain.

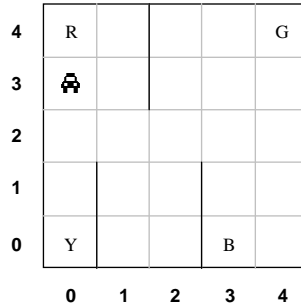


Figure 2.1: The Taxi Domain.

At each time step, the taxi can choose to navigate either north, south, east or west. It can also choose to pick up or to put down the passenger. If the taxi attempts to navigate through a wall, or if it attempts to pickup or putdown a passenger illegally, it will stay in the same square. Each such “illegal” action yields a penalty of -10 , while other legal actions yields a penalty of -1 . The final **putdown** action yields a reward of 20 .

There are 25 squares, 4 destinations and 5 locations of the passenger (also counting inside the taxi, which we will denote **T**). The size of the state space is calculated as a function of the grid size g and the number of specially-designated locations n : $size(g, n) = gn(n + 1)$. We will describe a state in the Taxi domain as a vector (x, y, p, d) where x and y denotes the location of the taxi, p denotes the location of the passenger, and d denotes the destination of the passenger. The location of the taxi is expressed in coordinates, while the location and destination of the passenger are expressed using the specially-designated locations.

2.3.2 The Blocks World Domain

The Blocks World domain consists of a number of unique blocks. Each block has a name, and can either be on the floor or on top of another block. We will write **on(a, b)** if block **a** is on top of block **b**. A block **a** is clear if no other blocks is on top of it, denoted by **clear(a)**. At each time step, a single block can either be moved to the floor (if not already there), or onto any clear block. The task in Blocks World is to reach a specified goal state in fewest possible steps. Figure 2.2 shows an example of an initial state and a goal state. A goal state specification might be partial, meaning that the goal is achieved if a subset of the blocks are at the correct place. For instance, **on(a, b)** is a partial goal-state specification.

Assuming a domain with three blocks $\{a, b, c\}$, the available actions are **move** (x, y) where $x \neq y, x \in \{a, b, c\}, y \in \{a, b, c, \text{floor}\}$. The size of the state space increases rapidly as more blocks are added. The size can be calculated as:

$$size(n) = \sum_{i=1}^n \frac{i(n-1)!size(n-1)}{(n-1)!}$$

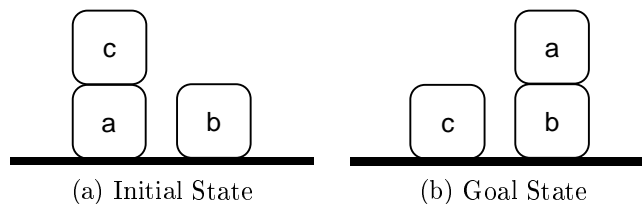


Figure 2.2: The Blocks World domain.

where n is the number of blocks. Table 2.3 shows the number of states for some small values of n .

n	1	2	3	4	5
$size(n)$	1	3	13	73	501
n	6	7	8	9	10
$size(n)$	4 051	37 633	394 353	4 596 553	58 941 091

Table 2.3: The size of the Blocks World domain, where n is the number of blocks, and $size(n)$ is the number of states.

Optimal planning in Blocks World was proven to be NP-hard by (Gupta and Nau, 1991). Furthermore, the domain exhibits so-called deleted-condition interactions, which has made it a popular domain in planning literature. As an example of a deleted-condition interaction, consider again the initial state in Figure 2.2. Given the task of achieving `on(a,b)`, we would first need to achieve two conditions: `clear(a)` and `clear(b)`. Since `b` is already clear, we clear `a` with the action `move(c,b)`. But now `b` is no longer clear, because we accidentally deleted one condition in order to achieve a second condition. The key observation here is that in the subtask of achieving `clear(a)`, the action `move(c,b)` is just as optimal as `move(c,floor)`. The presence of deleted-condition interactions makes Blocks World an interesting domain to investigate when combined with hierarchical reinforcement learning. Furthermore, the complexity of the domain is easily increased simply by adding more blocks.

2.4 Scaling of Reinforcement learning

As we saw in the previous section, the size of the state space in the Blocks World domain quickly grows as more blocks are added. Another effect of adding more blocks, is that the number of possible actions in almost every possible state also increases. Together, a large state space and many available actions create two primary concerns: the increasing space requirements of the action-value function, and the increasing time requirements of performing every action in every state sufficiently often. Imagine a scenario with 10 blocks and a fully specified goal state. The probability of reaching that particular goal state with initial random exploration is very low. Not only is the goal state only 1 state out of 58 941 091 states, the agent must also continuously chose actions that takes it closer to the goal state—amongst possible many actions that will take it further away. The reward function may give rewards for reaching other states than the goal state, so in general this problem occurs when the rewards are too sparsely distributed. The techniques presented in this chapter alone are simply not able to handle such problems in any reasonable time.

To illustrate the impact of increasing the number of blocks, we performed an experiment using the Q -learning algorithm. The task of the agent was to reach the goal `on(a,b)` from any initial state. The agent was trained for an increasing number of primitive steps, and was, with certain intervals, evaluated through 5 trials. Each evaluation resulted in a mean error per trial, which is the mean difference between the optimal solution and the solution chosen by the agent. The reason that we measured performance as a function of primitive training steps, instead of complete episodes, is that an agent explores more (and thus learns more) during an episode as a domain becomes increasingly complex—simply because it will take it a higher number of primitive steps to reach the goal. To avoid looping behavior, a trial was interrupted if the agent used more than the maximum number of steps for any initial state using the particular number of blocks—e.g. for 3 blocks, the maximum number of steps required to reach `on(a,b)` from any state is 4. In that case, the number of steps used by the agent was set to this maximum number. Figure 2.3 shows the results of the experiment.

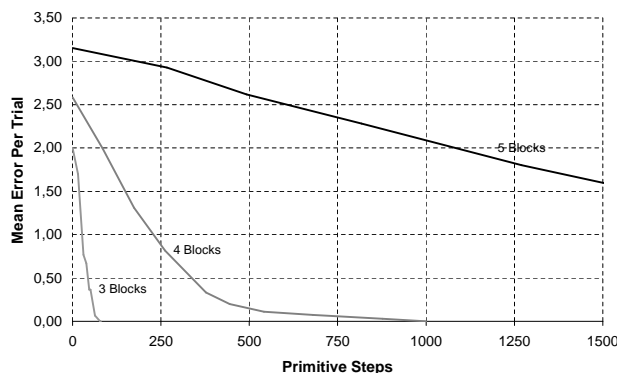


Figure 2.3: Performance of traditional Q -learning in Blocks World using 3 to 5 blocks.

For 3 blocks, optimal behavior is reached before 100 primitive training steps. Using 4 blocks, optimal behavior is not reached before after approximately 1000 steps, and for 5 blocks, approximately 15 000 training steps was needed before completely optimal behavior. Clearly, finding an optimal policy using traditional Q -learning will become infeasible very quickly as the number of blocks increase.

2.5 Value Prediction

It is unreasonable to expect an agent to explore the entire state space of a very large domain sufficiently for full convergence. It is even unreasonable to expect it to visit every state once given some realistic time constraint. Nevertheless, we would like to be able to predict the value of state/actions pairs that the agent has never visited. Unfortunately, the tabular representation of the action-value function discussed so far does not allow value prediction for unobserved state/action pairs (at least not directly).

An alternative to a tabular representation is to create a structure, which performs inductive inference on the components of a state/action pair. For instance, given the goal `on(a,b)` and a state/action pair with the action `move(a,b)`, the sum of all future rewards is going to be 0, because the goal will be achieved in the next time step. If such rules are learned, then

predictions can be made for all state/action pairs. Of course, the quality of the predictions will depend on the amount of training, as well as the uniformity of the state/action space.

The most common method for inductive inference is decision tree learning. Because the action-value function maps state/action pairs to real-valued numbers, it can be represented as a regression tree, where the nodes are tests on the state and action, and the leafs are the numerical values. Figure 2.4 shows a regression tree representing an action-value function for the goal $\text{on}(\mathbf{a}, \mathbf{b})$ using 3 blocks. Chapter 3 discusses the induction of regression trees further.

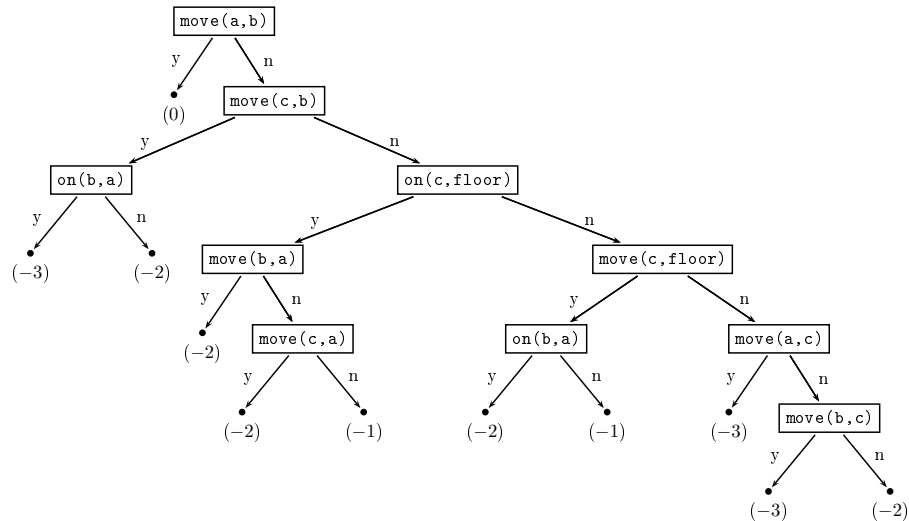


Figure 2.4: An action-value function for the goal $\text{on}(\mathbf{a}, \mathbf{b})$ using 3 blocks represented as a regression tree.

2.6 Generalizing Policies to New Environments

Policies using a tabular or regression tree representation, as discussed in this chapter, are very specific to the domain in which they are learned. For example, if a policy is learned in a domain with three blocks and the goal $\text{on}(\mathbf{a}, \mathbf{b})$, then it is not directly useable if the goal is changed to $\text{on}(\mathbf{a}, \mathbf{c})$ or if another block is added. In fact, these two issues were used by Džeroski et al. (2001) as part of the motivation in their introduction of relational reinforcement learning, which we will describe in Chapter 3. While this technique certainly solves these issues, some progress can be made without adding the same amount of overhead during learning.

The first issue regarding a change of goal state (e.g. from $\text{on}(\mathbf{a}, \mathbf{b})$ to $\text{on}(\mathbf{a}, \mathbf{c})$) can be solved by renaming blocks throughout the policy representation. Consider the regression tree in Figure 2.4 for the goal $\text{on}(\mathbf{a}, \mathbf{b})$. If we switch the names of each \mathbf{b} and \mathbf{c} in all nodes, then the tree represents a policy for the goal $\text{on}(\mathbf{a}, \mathbf{c})$. Although the time complexity of such a renaming mechanism is linear in the number of nodes in the tree, it can still be a time consuming task because the number of nodes for most domains will be high.

Adding another block to a domain will render a learned tabular representation of Q useless. A regression tree can, however, be used as a reasonable policy to speed up learning in the new

domain. Since the Q -function basically encodes the distance to the goal, which is obviously dependent on the number of blocks, the old policy will of course only be somewhat reasonable, and certainly not optimal. The reusability of a regression tree is enhanced significantly in Chapter 3.

2.7 Summary

This chapter introduced reinforcement learning as a learning technique that uses rewards and penalties to reinforce the actions of an agent. The environment in a reinforcement learning problem is represented as a Markov decision process, which, given a specific ordering of actions, has a unique optimal policy. A policy can be derived from the action-value function, which maps state/action pairs to their expected cumulative reward for some horizon.

The two most common temporal-difference algorithms for learning the action-value function are the off-policy Q -learning and the on-policy SARSA. The two algorithms differ only in their prediction of the action chosen in the next time step, and they both converge given infinite exploration and certain restrictions on the learning factor.

The Taxi domain, which is well suited for hierarchical decomposition, was introduced and will be the primary example in Chapter 4 for showing the advantages of hierarchical reinforcement learning.

The relational domain Blocks World was also introduced, and was used to demonstrate the limitations of traditional reinforcement learning when using a tabular representation of the action-value function. The problem of predicting unobserved values and generalizing to similar domains can be handled, to some extent, by the use of regression trees, but other limitations still exist. These include learning anything reasonable in very large domains but also learning more general policies without the need for renaming objects.

Chapter 3

Relational Reinforcement Learning

The previous chapter introduced reinforcement learning, and described the difficulties that the technique must overcome. This chapter describes Relational Reinforcement Learning (RRL), which was introduced by Džeroski et al. (2001). The idea behind RRL is to combine reinforcement learning with inductive first order predicate logic with variables. In its current state, this combination takes a step towards generalizing agent policies to similar domains using structural properties. Furthermore, the use of inductive logic also creates the possibility of applying state abstractions, which results in a more compact policy representation.

Section 3.1 introduces first-order predicate logic as a representation language for relational reinforcement problems. In Section 3.2, we describe relational MDPs and give a complete specification of the Blocks World domain. In RRL, policies are represented using logical decision trees. These are described in Section 3.3, while Section 3.4 explains how they can be learned using modified versions of the Q -learning algorithm. In Section 3.5, experiments are performed to illustrate the performance of RRL compared to reinforcement learning using a propositional tabular representation. The results of the experiments also show the extend to which policies can be generalized to similar domains. Finally, in Section 3.6 we discuss recent work in the field of relational reinforcement learning.

3.1 First-Order Predicate Logic

Learning in any domain requires the use of an appropriate representation language. An appropriate language should have enough expressive power to represent a domain and a given problem completely. In practice, the language should also allow acceptable performance while learning.

When we introduced the Taxi domain in Chapter 2, we used a propositional representation to describe the state space. This representation is very suitable for the Taxi domain, primarily because a grid world is very naturally represented using coordinates.

When we introduced Blocks World, however, we described a state using first-order predicate logic with predicates such as `on(a,b)` and `clear(a)`. Blocks World can easily be represented using a propositional language and coordinates such as the Taxi domain, but it would seem somewhat non-intuitive. This is because the high-level logical language directly encapsulates the concepts that are important in Blocks World. For instance, the answers to questions such

as “which block is on top of block a?” and “is block a clear?” are directly part of the state description.

First-order logic allows the use of universal and existential quantifications, which make the specification of appropriate domains much easier. For instance, to specify the available `move` actions in a Blocks World state, we can set up a premise using the following rule (in Prolog notation):

$$\text{move}(X,Y) \text{ :- clear}(X), \text{ clear}(Y), \text{ not}(X=Y)$$

which states that the conclusion `move(X,Y)` holds when any two different blocks `X` and `Y` are clear (ignoring the floor for now). Notice that here, `X` and `Y` are variables, and executing the query `?-move(X,Y)` (with `X` and `Y` un-instantiated), we are in fact asking if there exists any two blocks for which the premise of the rule holds. Given a state where both `clear(a)` and `clear(b)` holds, the conclusions `move(a,b)` and `move(b,a)` can be automatically inferred.

For further information on first order predicate logic refer to e.g. Russell and Norvig (2003).

3.2 Relational Domains

A relational domain is often described as the existing relations between objects. In general, a relational domain can be defined as a relational Markov decision process:

Definition 2 (RMDP). *A Relational Markov Decision Process (RMDP) is a process defined by a 7-tuple $\langle O, F, S, A, P, R, P_0 \rangle$:*

- O : the set of objects.
- F : the set of predicate relations over O .
- S : the set of all legal states over O and F .
- A : the set of all possible instantiated actions.
- P, R, P_0 : Unchanged from Definition 1.

The above definition includes the sets F and A , which are both exponential in the number of objects in O . These sets can be represented compactly by using logic. For Blocks World with three blocks $\{a, b, c\}$, we have that

$$\begin{aligned} O &= \{a, b, c\}, \\ F &= \{\text{on}(X,Y), \text{clear}(X) \mid X, Y \in O, X \neq Y\}, \text{ and} \\ A &= \{\text{move}(X,Y) \mid X, Y \in O, X \neq Y\}. \end{aligned}$$

Notice that the contents of the sets have not been changed, and their sizes are still exponential in the number of objects. It is only their representation that has been minimized.

In practice, as described in Chapter 2, we need to know the available actions $A(s)$ in any state s . We also need to represent the transition probability distribution P compactly. To achieve this, the actions available in a state, and the transitions they invoke, can be represented in

a STRIPS like manner (Fikes and Nilsson, 1990). STRIPS is a planning system that uses logical formulas to represent action preconditions and the transitions invoked by actions. Each action has a delete list and an add list, and its transition is performed by deleting all information in the delete list from the current state, and then adding all information from the add list. For Blocks World, we define the action preconditions as the predicate `pre`, and the transition probability distribution using the predicate `delta`. Table 3.1 illustrates the definition used by (Džeroski et al., 2001).

```

pre(S,move(X,Y)) :-
    holds(S,[clear(X), clear(Y), not(X=Y), not(on(X,floor))]).
pre(S,move(X,Y)) :-
    holds(S,[clear(X), clear(Y), not(X=Y), on(X,floor)]).
pre(S,move(X,floor)) :-
    holds(S,[clear(X), not(on(X,floor))]).

holds(S, []).
holds(S,[not X=Y | R]) :-
    not X=Y, !, holds(S,R).
holds(S,[not A | R]) :-
    not member(A,S), holds(S,R).
holds(S,[A | R]) :-
    member(A,S), holds(S,R).

delta(S,move(X,Y),NextS) :-
    holds(S, [clear(X), clear(Y), not(X=Y), not(on(X,floor))]),
    delete([clear(Y),on(X,Z)], S, S1),
    add([clear(Z),on(X,Y)], S1, NextS).
delta(S, move(X,Y), NextS) :-
    holds(S, [clear(X), clear(Y), not(X=Y), on(X,floor)]),
    delete([clear(Y),on(X,floor)], S, S1),
    add([on(X,Y)], S1, NextS).
delta(S, move(X,floor), NextS) :-
    holds(S, [clear(X),not on(X,floor)]),
    delete([on(X,Z)], S, S1),
    add([clear(Z),on(X,floor)], S1, NextS).

```

Table 3.1: Specification of the transition system and action preconditions for Blocks World

The auxiliary predicate `holds` takes a state `S` and a list of relations¹. If all relations in the list holds in the state, then the predicate succeeds, otherwise it fails. The precondition predicate `pre` takes as input a state `S` and an action `move(X,Y)`. If the action is allowed in `S` then `pre` succeeds, otherwise it fails. Finally, the input to `delta` is also a state `S` and an action `move(X,Y)`. The last parameter `NextS` is the output of the predicate and must be an un-instantiated variable when the predicate is called. If the action is legal according to `pre`, then `NextS` will be unified with the result of executing the action (i.e. `NextS` is instantiated as a side-effect).

A state $s \in S$, which we will represent in list notation, is any legal state over O and F . For instance, the state

¹While it is standard to use upper-case letters for sets and lower-case letters for set elements, Prolog notation unfortunately requires variables to be upper-case.

$$[\text{clear}(\mathbf{a}), \text{on}(\mathbf{a},\mathbf{b}), \text{on}(\mathbf{b},\mathbf{c}), \text{on}(\mathbf{c},\text{floor})]$$

is legal because it represents a scenario that is possible to build using three blocks. However, the state $[\text{on}(\mathbf{a},\mathbf{b}), \text{on}(\mathbf{b},\mathbf{c}), \text{on}(\mathbf{c},\mathbf{a})]$ is not legal, because no such scenario can be built. For some applications, it might be necessary to include a more formal notion of legal states in the RMDP definition.

To complete the formal specification of Blocks World, the penalty for any action is set to -1 . This will make the agent complete goals in the fewest possible steps to avoid more penalties in future time steps (since there are no more penalties after reaching the goal state). It will also make the Q values easier to read (e.g. $Q(s, a) = -4$ means that performing action a in state s will result in the goal state being reached in 4 steps). The probability of being an initial state is equal for all states.

3.3 Logical Representation of Policies

In itself, a relational MDP does not solve any of the problems discussed in Chapter 2. In this section, we will define a logical representation of the action-value function called Q -trees (Džeroski et al., 2001). We will also show how background knowledge can be incorporated into Q -trees, thereby achieving a higher level of abstraction.

In Chapter 2, the Q function was represented both as a table and as a regression tree to obtain the possibility of value prediction. A problem with both of these representations is that any changes to the environment or goal specification requires re-learning from scratch. Although the latter can be achieved by the use of a renaming mechanism (as described in Section 2.6), using variables will result in far more elegant solution. A special kind of decision tree, which uses first order logic, was formally introduced by Blockeel and Raedt (1998). These trees make use of variables that makes it possible to avoid direct references to objects.

Definition 3 (FOLDT). *A first order logical decision tree (FOLDT) is a binary decision tree in which the following applies:*

- The nodes of the tree contain a conjunction of predicates.
- Different nodes may share variables, under the following restriction: a variable that is introduced in a node (which means that it does not occur in higher nodes) must not occur in the right branch of that node (i.e. the “no” branch).

Each node in an FOLDT contains a predicate or a conjunction of predicates (which is also just a predicate). A predicate may contain one or more variables that will be instantiated with different values as the predicate is applied to various examples. Here, an example is simply the state, action and goal of a specific time step. Given a node and an example, a node predicate either holds or does not hold. If it does not hold, then the example is sorted down the right branch (the no branch) and no variables are instantiated. Otherwise, the example is sorted down the left branch (the yes branch), and any existing variables are instantiated to the values that allowed the predicate to hold. Figure 3.1 illustrates an FOLDT where the variable C is shared between two nodes. Each leaf is a numerical value—in this case a (random) Q -value. A logical regression tree representing a Q -function is referred to as a Q -tree.

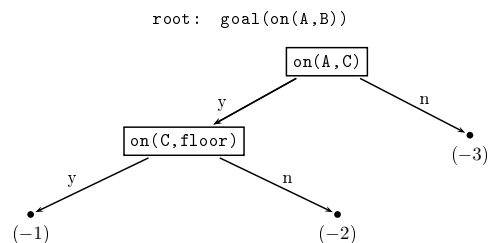


Figure 3.1: A FOLDT illustrating variable sharing.

The illustrated Q -tree contains an extra root with the predicate $\text{goal}(\text{on}(A,B))$. This extra root is used for predicates that always hold given the state/action space and the goal of the agent. The point of the extra root is to instantiate relevant variables. In this case, the goal is to always have a specific block on top of another specific block, so the predicate $\text{goal}(\text{on}(A,B))$ will always hold (because the variables A and B can be any blocks). However, in the rest of the tree, A and B will be instantiated to the blocks in the actual goal state. For example, given the goal $\text{goal}(\text{on}(b,c))$, A will be instantiated to b and B to c . The same way as the goal is wrapped in a goal predicate, we will also wrap actions in an action predicate.

The restriction of not referring to a variable introduced by a particular node in its right branch makes sense when observing Figure 3.1. The predicate $\text{on}(A,C)$ in the tree introduces the variable C . It tests if A is on top of any block C . If the predicate does not hold, then it makes no sense to reference C again², since there is no such block (i.e. C will remain un-instantiated).

A Q -tree can be encoded using a Prolog rule structure, where each leaf is encoded by exactly one rule. The premise of a leaf rule is the predicates encountered on the path from the root of the Q -tree to the particular leaf. The predicates that do not hold on the path can safely be ignored by using the cut-operator ($!$). The cut-operator denotes that if the rule in question holds, then no other rules are considered. Table 3.2 shows the rule structure representing the Q -tree of Figure 3.1.

```

q(-1)  :- goal(on(A,B)), on(A,C), on(C,floor), !.
q(-2)  :- goal(on(A,B)), on(A,C), !.
q(-3)  :- goal(on(A,B)), !.
  
```

Table 3.2: Prolog rule structure representing the Q -tree of Figure 3.1.

Predicates that do not hold on the path to a specific leaf are encoded in the ordering of rules. For instance, the rightmost leaf in Figure 3.1 with a value of -3 is represented by the last rule above. If this rule is considered, then none of the rules above have succeeded.

3.3.1 Background Knowledge

A predicate used inside the node of a Q -tree can be any predicate from the set of predicate relations F (see Definition 2). It can also be a predicate present in some specified background knowledge. In RRL, background knowledge is simply predicates which induce facts and

²Technically, the name C could be reused, but semantically it would be a different variable.

relations on a higher level of abstraction using the predicates present in F . For Blocks World, an example of a background knowledge predicate is the `above(X,Y)` predicate:

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

The predicate `above(X,Y)` holds if either X is on Y , or if X is on some other block Z , and Z is above Y . The use of background knowledge has two effects: first, the representation of a Q -tree will most likely be more compact, and second, it can make a policy less specific for the environment in which it was learned. While the first effect is obvious, the second is easily illustrated with an example. Consider an environment with 4 blocks (a , b , c and d) and the goal `on(a,b)`. Before being able to perform the action `move(a,b)` which completes the goal, a and b must first be clear. While the agent clears a , it should obviously not move blocks onto the stack in which b is located (or vice versa). Doing so would not bring it any closer to the ultimate goal. Thus, the agent should not perform any action `move(X,Y)` if either `above(Y,a)` or `above(Y,b)` is true. It turns out that an optimal policy for achieving `on(A,B)` for any A and B with any number of blocks can be specified using the `above` predicate:

```
optimal(goal(on(A,B)),move(A,B)) :- !.
optimal(goal(on(A,B)),move(X,Y)) :- above(X,A), not(above(Y,B)), !.
optimal(goal(on(A,B)),move(X,Y)) :- above(X,B), not(above(Y,A)), !.
```

Two other common goals of the Blocks World domain is `unstack` and `stack`. The goal `unstack` is achieved if all blocks are on the floor, and `stack` is achieved if all blocks are in the same stack. While an optimal policy for `unstack` is straightforward to define without background knowledge, this is not possible for `stack` if the independence of the number of blocks is to be maintained. From any given initial state, optimal behavior is to locate the highest stack and then keep moving blocks onto that stack. The optimal policies for both `stack` and `unstack` can be specified as

```
optimal(unstack,move(X,floor)) :- on(X,Y), not(Y=floor).
optimal(stack,move(X,Y)) :- height(Y,HY), not(height(Z,HZ), HZ > HY).
```

where `height(X,H)` is background knowledge that instantiates the variable H with a number indicating the height of block X . The `unstack` rule is read as: the action `move(X,floor)` is optimal if X is not already on the floor. The `stack` rule is read as: the action `move(X,Y)` is optimal if Y is in a stack of height HY , and no block Z in a higher stack exists.

3.3.2 The Policy Function

A major part of the motivation behind RRL is to enable generalization of learned policies to other similar domains. As briefly mentioned in the previous Chapter, the Q -function, in principle, encodes the distance to the goal after performing a state/action pair. In Blocks World, this distance is partly determined by the number of blocks in the domain. In effect, if a new block is added, then the distance for many state/action pairs is changed with the consequence of making the old policy perform worse.

To avoid the direct encoding of distance, Džeroski et al. (2001) introduced the policy function P , which encodes the optimality of each action a in each state s :

$$P(s, a) = \begin{cases} 1, & \text{if } a \in \pi^*(s) \\ 0, & \text{otherwise} \end{cases} \quad (3.1)$$

In general, the P -function can be represented more compactly than the Q -function. Since both the Q and P function can be defined in terms of the optimal policy π^* , the definition of P can be rewritten in terms of Q :

$$P(s, a) = \begin{cases} 1, & \text{if } a \in \arg \max_a Q(s, a) \\ 0, & \text{otherwise} \end{cases} \quad (3.2)$$

This definition of P means that it is still sufficient to learn the Q function, since P can then be directly derived. The P function can be represented as a logical classification tree denoted as a P -tree. Figure 3.2 shows the optimal P -tree for the goal $\text{on}(A,B)$ using three blocks.

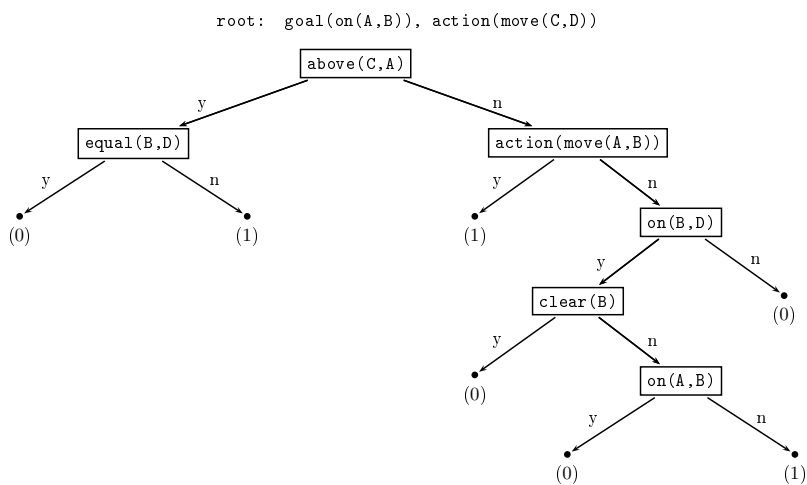


Figure 3.2: The optimal P -tree for the goal $\text{on}(A,B)$ using three blocks.

The illustrated P -tree also shows that sufficient learning is needed to achieve a truly general policy. Clearly, the left branch of the tree is not optimal if more blocks are added. It states that, given the goal $\text{on}(A,B)$ and the action $\text{move}(C,D)$, an action is optimal if C is above A and D is not equal to B . In other words, it is optimal to move blocks away from A as long as they are not moved directly onto B . Obviously, a more general tree should encode that blocks must not be moved onto B or any block above B . However, an agent trained using only three blocks would never have encountered a scenario where such a rule was necessary. This means that true generality in similar domains (as Blocks World with a varying number of blocks) can only be achieved if training is done in sufficiently complex instances of the domains. For Blocks World and the goal $\text{on}(A,B)$, the example just discussed suggests that the minimum number of blocks needed in training to obtain a general policy is four.

3.4 Learning Logical Policies

The Q -learning and SARSA algorithms described in Chapter 2 are both online algorithms meaning that the agent policy is updated at each time step. This is straightforward since

they both use a tabular representation for the Q function. When changing the representation to a (logical) decision tree, Q is often not updated before at the end of each episode making it off-line learning (the agent does not learn while being online). This is done to avoid the overhead of updating the Q -tree at each time step.

A Q -tree is induced by using examples generated over the state/action pairs encountered over previous episodes. An example is created for each state/action pair, the goal, and the estimated Q value. As in Chapter 2, this estimation is made by predicting the action taken in the next time step. Table 3.3 shows four examples from Blocks World.

Example 1	Example 2
goal(on(a,b)).	goal(on(a,b)).
action(move(b,floor)).	action(move(b,a)).
on(a,floor).	on(a,floor).
on(b,c).	on(b,c).
on(c,floor).	on(c,floor).
clear(a).	clear(a).
clear(b).	clear(b).
q(-3).	q(-2).
Example 3	Example 4
goal(on(a,b)).	goal(on(a,b)).
action(move(a,b)).	action(move(a,floor)).
on(a,floor).	on(a,b).
on(b,c).	on(b,c).
on(c,floor).	on(c,floor).
clear(a).	clear(a).
clear(b).	q(0).
q(-1).	

Table 3.3: Blocks World examples.

In example 4 in the table, the goal state is already reached and its Q value is therefore set to 0. By generating such an example per episode, the generated Q -trees will quickly converge towards returning 0 for goal states.

The Q -RRL algorithm for learning Q -trees is illustrated in Table 3.4, and it is very similar to the traditional Q -learning algorithm (see Table 2.1). Instead of updating \hat{Q} during an episode, examples are generated at the end of each episode and a new Q -tree is induced using TILDE-RT (Blokeel and Raedt, 1998) and all the examples observed so far. The TILDE/TILDE-RT algorithms are described in Section 3.4.1.

The Q -RRL algorithm learns the Q -function, but we would also like to learn the more general policy function P . In Section 3.3.2 it was described how P can be defined from Q . This definition can be directly used to extend the Q -RRL algorithm to produce P -RRL. The two algorithms are identical except that the pseudo-code in Table 3.5 is appended to the end of P -RRL.

While the P -function is obviously more compact than the Q -function, experiments conducted by Džeroski et al. (2001) show that it does in fact also perform better in most cases. Since P is derived from Q , this seems strange at first. The first observation to make is that the optimality of an action does not always depend on the distance to the goal. The second observation is that using a logical decision tree representation of P (as in P -RRL algorithm),

```

1: Initialize  $\hat{Q}_0$  to assign 0 to all  $(s, a)$  pairs
2: Initialize Examples to the empty set
3:  $e := 1$ 
4: while ( $e < \text{EpisodeCount}$ ) do
5:    $e := e + 1$ 
6:    $i := 0$ 
7:   Generate a random state  $s_0$ 
8:   while not(goal( $s_t$ )) do
9:     Select action  $a_t$  in state  $s_t$  using exploration policy  $\pi_e$  and execute it
10:    Receive immediate reward  $r$ 
11:    Observe the new state  $s_{t+1}$ 
12:     $i := i + 1$ 
13:  end while
14:  for ( $j = i - 1$  to 0) do
15:    Generate example  $x = (s_j, a_j, \hat{q}_j)$  where
16:     $\hat{q}_j := (1 - \alpha_e)\hat{Q}_{e-1}(s_{j+1}, a) + \alpha_e [r_j + \gamma \max_a \hat{Q}_e(s_{j+1}, a)]$ 
17:    if ( $x_{old} = (s_j, a_j, \hat{q}_{old})$  exists in Examples) then
18:      Replace  $x_{old}$  with  $x$  in Examples
19:    else
20:      Add  $x$  to Examples
21:    end if
22:  Update  $\hat{Q}_e$  using TILDE-RT to produce  $\hat{Q}_{e+1}$  using Examples
23:  end for
24: end while

```

Table 3.4: The Q -RRL algorithm.

```

1: for (all observed states  $s$ ) do
2:   for (all actions  $a_k$  possible in state  $s$ ) do
3:     if (state/action pair  $(s, a_k)$  is optimal according to  $\hat{Q}_{e+1}$ ) then
4:       Generate example  $(s, a_k, c)$  where  $c = 1$ 
5:     else
6:       Generate example  $(s, a_k, c)$  where  $c = 0$ 
7:     end if
8:   end for
9: end for
10: Update  $\hat{P}_e$  using TILDE to produce  $\hat{P}_{e+1}$  using these examples  $(s, a_k, c)$ 

```

Table 3.5: Learning P -trees from Q -trees within the P -RRL algorithm.

it is really only the examples that are derived directly from the Q -function. These examples contain information about the optimality of actions, but not about the distance to the goal, which has been abstracted away. The induction of the P -tree is therefore a task of generalizing over the optimality of actions. There are, in principle, two ways that a P -tree can outperform a Q -tree:

- **The examples do not cover the entire state/action space, however all Q -values are correct:**

This scenario is equivalent to moving the learned policy to a larger and more complex domain. The key observation here is that, using a Q -tree, the optimality of a state/action pair is determined not only by the Q value of the pair, but also by the values of all other available actions in the same state. This means that a single error in the Q values of any of these actions can change what is considered the optimal action. Furthermore, although a Q -tree partitions a limited part of the state/action space correctly (into pairs with various distances to the goal), this might not be so for the entire state/action space. Un-observed state/action pairs might even have Q -values outside the scope of the learned Q -tree. Together, these issues will affect the performance of a Q -tree for un-observed state/action pairs. Figure 3.3 illustrates how the incorrect Q value of a previously unobserved action can shift what is considered optimal when using a Q -tree. In this case, the goal is on (a, b) and we assume that the current Q -tree is optimal for a domain with 3 blocks. The action $\text{move}(d, c)$ is previously unobserved because it is introduced by adding the new block d to the domain.

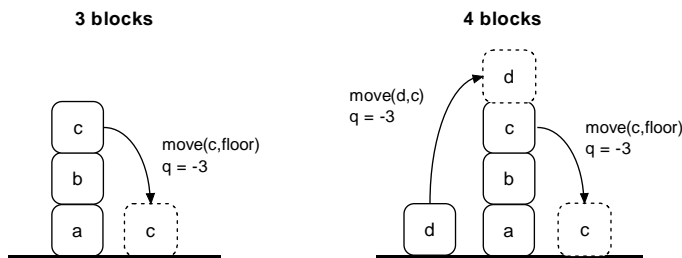


Figure 3.3: Previously unobserved actions can only be assigned Q -values within the limit of the current approximation of Q . Introducing new blocks will therefore result in actions that are assigned wrong Q -values.

The optimal (and only possible) action, when using 3 blocks in the illustrated state, is $\text{move}(c, \text{floor})$. This action yields a Q value of -3 . Adding the new block d introduces the action $\text{move}(d, c)$, which is of course not optimal since it is actually a step further away from the goal state. However, the Q -tree would possibly assign this unobserved action the same value as $\text{move}(c, \text{floor})$ because of the similarity of the state. Worse than that, it could happen that $\text{move}(d, c)$ is assigned an even lower Q -value because of similarity with some other state/action pair. Notice that when using 3 blocks, the maximal value in an optimal Q -tree for any state/action pair is -4 . However, the real Q -value of the action $\text{move}(d, c)$ in Figure 3.3 is -5 . Thus, in some states, the notion of optimality *will* change. To which extend depends on the Q -tree.

In the examples used during the induction of a P -tree, an action is either optimal or not optimal—thus the distance to the goal has been abstracted away. Given that the known state/action space is sufficiently large, a reasonable set of rules, separating

optimal actions from non-optimal actions, can most likely be found. The performance of such rules is only affected by whether or not there exists structural similarities between optimal actions. In the scenario illustrated in Figure 3.3, block d is moved on top of a stack containing a and b (blocks in the goal state). Such an action is very unlikely to be classified as optimal in a P -tree, simply because it does not show any structural similarities with real optimal actions in Blocks World.

- **The examples cover the entire state/action space, but some Q -values are wrong:**

Depending on the tests and pruning heuristics available during induction of a P -tree, actions that are not optimal according to the examples, but which are in fact optimal, will be grouped with other optimal actions because of their structural similarities (given that such errors are limited). During induction, the tests chosen for the tree will partition presumably optimal and non-optimal actions in the best possible way. But at some point, nodes will be reached where the set of examples cannot be partitioned any further using the available tests. Because of their similarities, true optimal actions tend to end up together in such leaves.

Since a better general performance can be expected from using the P -function, it is also feasible to change the Boltzmann exploration technique to utilize the P -function (refer to Section 2.2):

$$P(a_i|s) = \frac{T^{-\hat{P}(s,a_i)}}{\sum_j T^{-\hat{P}(s,a_j)}} \quad (3.3)$$

After the P -function has been learned as a P -tree, it can be queried to find the optimal action in any given state. The changes and extensions made in this section have only been applied to the Q -learning algorithm. SARSA can be updated similarly for relational reinforcement learning.

3.4.1 Induction of Logical Decision Trees

In this work, the TILDE and TILDE-RT algorithms developed by Blokkeel and Raedt (1998) have been used to grow P -trees and Q -trees. The algorithms are now part of the ACE data mining system (Blokkeel, Raedt, Dehaspe, Ramon, Struyf and Laer, 2004). TILDE and TILDE-RT differs only in that they induce classification trees and regression trees, respectively. As will be explained, this difference boils down to the heuristics used to determine the quality of single tests. Table 3.6 shows the basic pseudo-code for both algorithms.

TILDE and TILDE-RT are very similar to classical decision tree algorithms such as ID3 and C4.5. The task of computing the possible tests in a node (line 8) is, however, new and non-trivial. It is described further in Section 3.4.2. Furthermore, an example satisfies the test in a node only if it also satisfies the tests in nodes higher in the tree when following the yes-branch. This is used when determining the quality of a test (line 9 and 14), and is further described in Section 3.4.3 and Section 3.4.4. Finally, since the outcome of a logical test is either yes or no, the induced trees are always binary.

The TILDE/TILDE-RT algorithms are non-incremental algorithms. This means that all observed examples must be stored in some database, and a new tree must be induced from

```

1: function INDUCETREE(Examples  $E$ ) : Tree
2:   Create a root node  $n$  for the tree  $t$ 
3:   SPLIT( $n, E, t$ )
4:   return  $t$ 
5: end

6: procedure SPLIT(Node  $n$ , Examples  $E$ , Tree  $t$ )
7:    $best := false$ 
8:   for (all possible tests  $q$  in node  $n$ ) do
9:     Compute quality( $q$ )
10:    if (quality( $q$ ) is better than quality( $best$ )) then
11:       $best := q$ 
12:    end if
13:  end for
14:  if ( $best$  yields improvements) then
15:     $test(n) := best$ 
16:    Create two sub-nodes  $n_{\oplus}, n_{\ominus}$  of  $n$  in  $t$ 
17:     $E_{\oplus} := \{e \in E \mid e \text{ satisfies } best \text{ in } t\}$ 
18:     $E_{\ominus} := \{e \in E \mid e \text{ does not satisfy } best \text{ in } t\}$ 
19:    SPLIT( $n_{\oplus}, E_{\oplus}, t$ )
20:    SPLIT( $n_{\ominus}, E_{\ominus}, t$ )
21:  else
22:    Turn  $n$  into a leaf
23:  end if
24: end

```

Table 3.6: The TILDE/TILDE-RT algorithm.

scratch after each episode. Every time an example is observed, the database must be searched for previous observations of the same example, such that the estimated value can be updated. For all but small domains, this overhead slows down the learning process considerable. Fortunately, research has shown that the process can be made much faster with the use of an incremental tree learner such as the TG-Algorithm (Driessens et al., 2001)³. For our purpose, however, TILDE/TILDE-RT will suffice.

3.4.2 Finding Test Candidates

The set of predicate relations in a domain F (see Definition 2) contains all the relations that can be present in a state (e.g. $on(a, b)$). It is important to note that these predicates do not contain variables (although a compact specification of them might). In a classical decision tree learner, this set of relations would be used as tests, where each test would only be allowed once in any given subtree.

For logical decision trees, we now also have to consider background knowledge and variables. Furthermore, we might not even be interested in allowing tests with constants such as $on(a, b)$. As specified in Definition 3, if a variable is introduced in a node, then it can be referenced by nodes in the yes-branch of the subtree of that node. Together this means that a parameter in a test can either introduce a variable, reference an existing variable or be a constant. This is called the mode of the parameter. TILDE/TILDE-RT supports restriction of the mode, so that some tests might only be allowed to have parameters with existing variables and so on. Also, a parameter can be assigned a type, which means that only variables of that type can be used. Mode and type restrictions are specified using a so-called declarative

³An implementation of the TG-algorithm is also available in the ACE data mining system.

bias. For the predicates `on` and `clear`, the declarative bias (using TILDE notation) might look like the following:

```

type(on(block,block)).
type(clear(block)).
rmode(5: on(+X,+Y)).
rmode(5: on(+X, floor)).
rmode(5: clear(+X)).

```

The `type` predicates state that only variables or constants of the type `block` can be used. The `rmode` predicate is a little more complex. A `+` means that an already introduced variable can be used, and a `-` means that a new variable can be introduced. If needed, `#` means that any observed constant (in the set of examples) can be inserted. The number 5 denoted inside the `rmode` predicates indicates how many times tests created over this predicate are allowed to occur in the tree.

Consider the Q -tree illustrated in Figure 3.1 and the first node with the test `on(A,C)`. Using the declarative bias above, this test was chosen among the test candidates

$$\{\text{on}(A,B), \text{on}(A,C), \text{on}(B,A), \text{on}(B,C), \text{on}(C,A), \text{on}(C,B), \text{on}(A, \text{floor}), \\ \text{on}(B, \text{floor}), \text{on}(C, \text{floor}), \text{clear}(A), \text{clear}(B), \text{clear}(C)\}$$

where `A` and `B` were introduced by the root and `C` is a new un-instantiated variable. An un-instantiated variable should be read as “any block”. For this small example, there are already 12 possible tests in the first node in the tree. Since more and more variables are introduced, there will often be many more tests possible in nodes further down the tree. Fortunately, the possible restriction on the number of times a test can occur helps to keep the number of possible tests reasonable. This requires careful specification though, as one more test could mean the difference between a good and a poor tree.

3.4.3 Example Testing

When the set of test candidates are determined for a node, each candidate must be applied to the set of examples sorted down to that node. The goal is to find the test that yields the best split (Section 3.4.4 describes what is meant by the best split). In classical decision tree learning, each candidate is simply applied to each example, but because of variable sharing, this does not work for logical decision trees. A variable used in a test might have been introduced in a node higher in the tree. This means that the value of that particular variable is dependent on (possible) all the tests from the root to the given node. Since variables are not instantiated in failing tests (see Section 3.3), only positive tests must be considered. Again using the Q -tree of Figure 3.1, the node containing `on(C,floor)` in fact represents the test

$$\text{goal}(\text{on}(A,B)), \text{on}(A,C), \text{on}(C, \text{floor})$$

While processing this node, TILDE will put any example that makes the test succeed into the set E_{\oplus} , and all other examples into E_{\ominus} . These sets are then used to determine the quality of the test.

3.4.4 Quality Heuristics

The best test in a node is chosen among the possible test candidates by using an appropriate quality heuristic. For classification trees, TILDE uses a measure of information gain (Quinlan, 1993) which must be maximized. Because only Boolean tests are considered, the entropy of a set of examples E can be defined as

$$\text{entropy}(E) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \quad (3.4)$$

where p_{\oplus} is the proportion of positive examples in E and p_{\ominus} is the proportion of negative examples in E . If $p_{\oplus} = 0$ then we will define $p_{\oplus} \log_2 p_{\oplus}$ to be 0 (the same applies to p_{\ominus}). The information gained for applying a boolean test T on the set of examples E can be defined as

$$\text{gain}(E, T) = \text{entropy}(E) - (\text{entropy}(E_{\oplus}) + \text{entropy}(E_{\ominus})) \quad (3.5)$$

where $E_{\oplus} \in E$ is the set of examples that satisfy T and $E_{\ominus} \in E$ is the set of examples that do not satisfy T . The value calculated by $\text{gain}(E, T)$ is the expected reduction in entropy caused by knowing the outcome of test T .

For regression trees, TILDE-RT uses the intra-subset variance quality criterion (Breiman, Friedman, Olshen and Stone, 1984) which must be minimized. The variance of a set of examples E can be defined as

$$\text{variance}(E) = \sum_{i=0}^{|E|} (t_i - t_{\text{mean}})^2 \quad (3.6)$$

where t_i is the target-value in example i and t_{mean} is the mean of all the target variables in E :

$$t_{\text{mean}} = \frac{\sum_{i=0}^{|E|} t_i}{|E|} \quad (3.7)$$

Using the intra-subset variance, the quality of a test T can be defined as the relative improvement of variance:

$$\text{variance-improvement}(E, T) = \frac{\text{variance}(E_{\oplus}) + \text{variance}(E_{\ominus})}{\text{variance}(E)} \quad (3.8)$$

The variance improvement is always a number between 0 and 1 because the summed variance of E_{\oplus} and E_{\ominus} is never greater than the variance of E .

3.5 Experimental Evaluation of Relational Reinforcement Learning

In Chapter 2, the performance of tabular Q -learning was evaluated using experiments. In this section, we will experiment with the performance of relational Q -learning. We will try to clarify the answers to the following questions:

- What is the performance of relational Q -learning compared to tabular Q -learning?
- How does P -trees perform compared to Q -trees?
- How do the state abstractions possible in RRL affect the size of the learned policies?

These questions are difficult to answer theoretically, since they depend greatly on the examples observed during training episodes. The experiments were conducted the exact same way as in Section 2.4. The data obtained for tabular Q -learning in that section was reused for comparison to data obtained by using Q -trees and P -trees. Settings and background knowledge used with TILDE/TILDE-RT can be found in Appendix B. The performance was compared for 3 to 5 blocks. Figure 3.4 shows the results of the experiment. Each diagram shows a graph for tabular Q -learning, a graph for relational Q -learning using Q -trees, and a graph for relational Q -learning using P -trees. The graphs map the number of primitive steps during training to the mean error per trial observed during testing. The mean error per trial is the mean difference between the steps used by an optimal policy, and the steps used by the evaluated policy. The mean was taken over 10 trials.

The results show that relational Q -learning outperforms its tabular counterpart in every case. The only exception to this rule is for 3 blocks where the tabular representation reaches optimal behavior before the Q -tree, although not before the P -tree. A noticeable observation is that both Q -trees and P -trees reach a reasonable performance after very little training compared to tabular Q -learning. For readability, the diagram for 5 blocks does not show when tabular Q -learning reaches optimal behavior. This happens after approximately 15000 primitive steps during training (refer to Section 2.4).

As expected, by using P -trees the agent reaches reasonable behavior faster than when only using Q -trees. Optimal behavior is also reached faster, but only slightly. Of course, for a P -tree to perform optimally, it requires an almost optimal Q -tree such that the notion of optimality is not biased in the wrong direction. Otherwise, a P -tree would not be able to find a good pattern of optimality.

To determine if P -trees also perform better in more complex domains, we used the optimal trees from the previous experiment and applied them to domains with an increasing number of blocks. For each domain, 50 trial states were randomly chosen, whereafter the error per trial were recorded. Figure 3.5 shows the results of this experiment. The graph shows the mean error per trial as a function of the number of blocks in the domain.

The optimal P -tree learned in a 3 blocks domain performs reasonably well in more complex domains, and much better than the corresponding Q -tree. It does not perform optimal because a 3 block training domain is not complex enough. The Q -tree learned in a domain with 4 blocks performs slightly better than the one learned using 3 blocks. However, the P -tree learned for 4 blocks is optimal for any number of blocks. As suggested in Section 3.3.2, this makes sense since Blocks World with the goal $\text{on}(A,B)$, on an abstract level, does not become any more complex when using more than 4 blocks. The optimal policy remains to clear both blocks in the goal state without ever moving blocks onto stacks containing the other, and then moving A onto B . The optimal P -tree learned in the experiment is illustrated in Figure 3.6, and it clearly follows this principle.

The optimal P -tree for the goal $\text{on}(A,B)$ classifies the final action $\text{move}(A,B)$ as optimal. Furthermore, if a block is moved to the floor, then it must have been above one of the blocks in the goal state for the action to be optimal. In other words, it is optimal to clear the two blocks in the goal state. Finally, if neither of these two statements hold, then the following

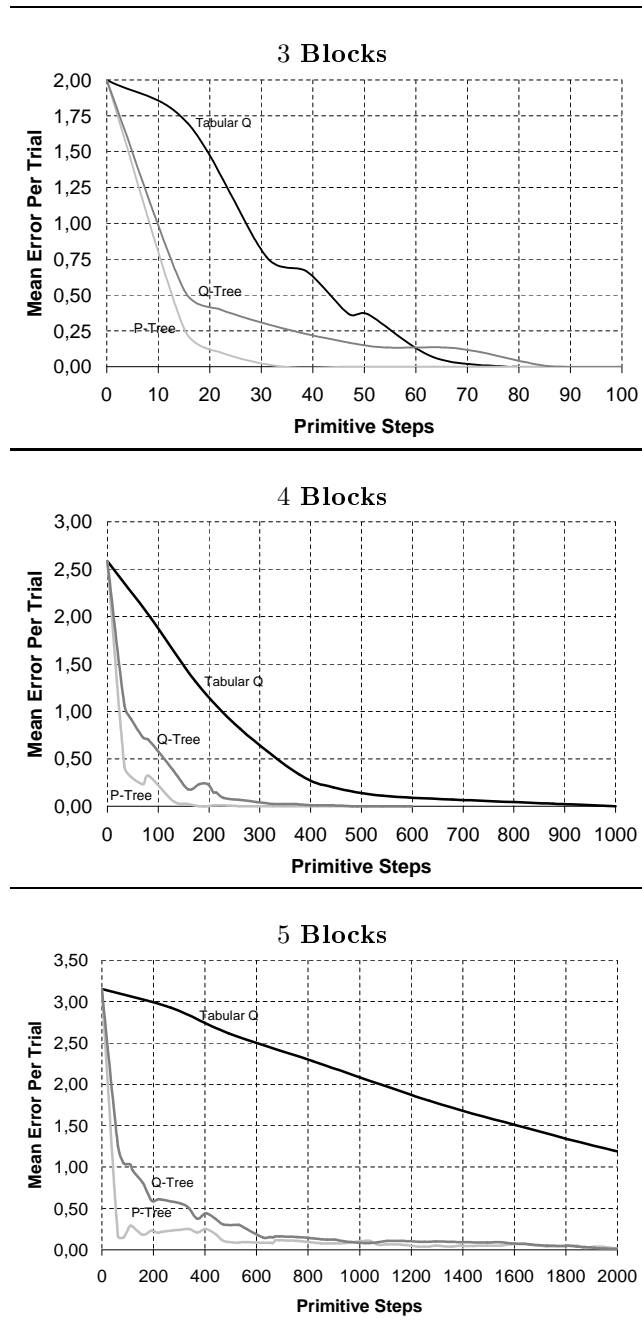


Figure 3.4: Performance of traditional Q -learning compared to relational Q -learning and P -learning in Blocks World using 3 to 5 blocks. The graphs map the number of primitive steps during training to the mean error per trial observed over 10 trials.

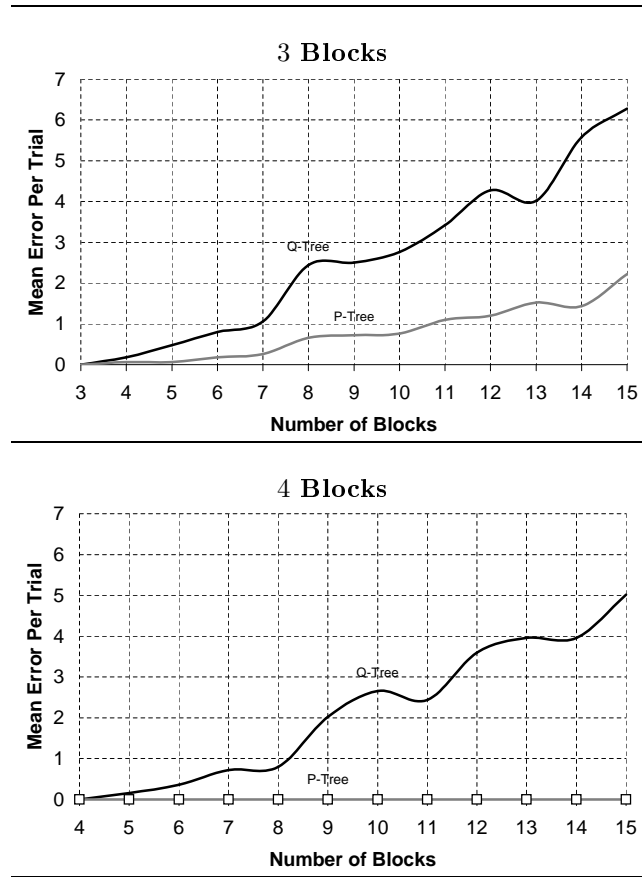


Figure 3.5: Performance of *Q*-trees and *P*-trees learned for 3 and 4 blocks when applied to domains with more blocks.

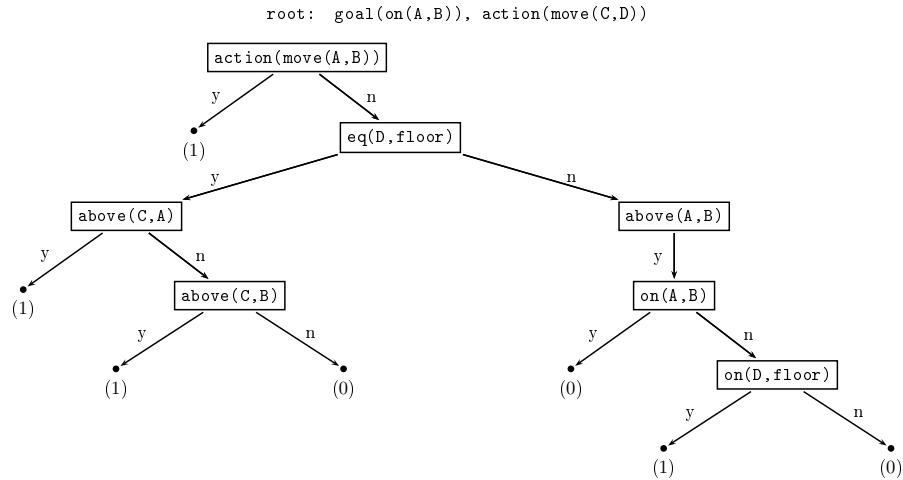


Figure 3.6: The optimal *P*-tree for any number of blocks learned using 4 blocks.

Blocks	Tabular Q	Q -Tree	P -Tree
3	25	14	5
4	209	56	7
5	1887	169	9

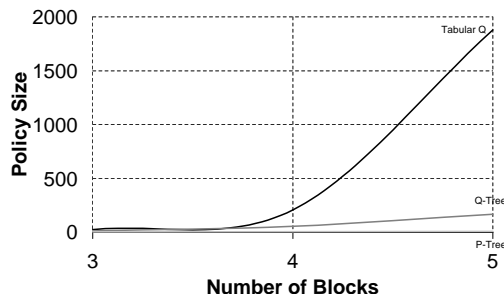


Table 3.7: The size of policies when using tabular and relational Q -learning.

must hold for an action to optimal: A must be above B but not directly on it, and the block A is being moved onto must be on the floor. This is specified in the right part of the tree, and seems strange at first, but is in fact a side-effect of having learned using only 4 blocks. During clearing of the two blocks in the goal state, there can be at most one clear block on the floor that is not part of the goal state. Any block not moved to the floor must be moved onto this irrelevant block. In a more general setting, a block can be moved onto any block that is not above a block in the goal state.

The experiments have now answered two of the questions stated in the beginning of the section. Relational Q -learning performs better than tabular Q -learning, both by reaching a reasonable behavior very quickly, but also by reaching optimal behavior with less training needed. Q -trees will perform optimally in the domain in which they were trained, but do not generalize well when more blocks are added. A P -tree derived from an optimal Q -tree will also perform optimally in the training domain. However, given a training domain with enough blocks, a P -tree will also perform optimally in domains with any number of blocks.

During the experiments, policies for domains with 3, 4 and 5 were learned both for relational and tabular reinforcement learning. The use of logic and background knowledge in RRL automatically enables abstractions over the state/action space of a domain. It is therefore interesting to investigate to which extent these abstractions affect the space used by the learned policies. The size of a tabular policy is simply the number of cells in the table, while the size of a tree-based policy can be defined as the number of leaves. The latter makes sense because using the Prolog-based rule notation, there will be exactly one rule for each leaf. Table 3.7 shows the size used by the learned policies.

The numbers presented in the table should not be read as the only possible sizes of the policies. They are the sizes of the optimal policies observed during the performed experiments. Depending on the available tests and background knowledge, the size of the policies may vary to each side. The numbers are, however, a good indication of the abstraction possibilities of RRL. The graphs, also illustrated in Table 3.7, pictures the exponential growth when using a tabular representation compared to logical decision trees.

3.6 Recent Work

As mentioned, the research applied to relational reinforcement learning since its introduction has moved it past learning algorithms such as TILDE/TILDE-RT. The incremental TG-algorithm (Driessens et al., 2001) was the first obvious step as it combines TILDE with the incremental G-algorithm (Chapman and Kaelbling, 1991).

Following, two other regression algorithms has been developed. The first is an instance based algorithm named RIB (Driessens and Ramon, 2003). It computes a weighted average of the Q values of examples where the weight is inversely proportional to the distance between the examples. The second algorithm is called KBR (Gärtner, Driessens and Ramon, n.d.) and uses Gaussian processes as the regression technique. Because Gaussian processes are a Bayesian technique, the KBR algorithm offers both basic prediction of the Q value, but also indication of the expected accuracy of the prediction. This indication can be used by the Q -learning algorithm to guide exploration.

Relational reinforcement learning has only been sparsely tested in more realistic domains. One such test was conducted on a simplified version of the multi-agent board game Risk (Andersen, Boesen and Pedersen, 2005). The results of that work indicates that reasonable policies can be learned even in semi-complex multi-agent environments.

Furthermore, the integration of guidance into relational reinforcement learning has been discussed by Driessens and Džeroski (2004). Their work evaluates the advantages of supplying an agent with optimal and reasonable examples during training. The advantages are evaluated using both the TG and RIB algorithms.

3.7 Summary

This chapter introduced relational reinforcement learning, which combines traditional reinforcement learning with inductive logic. The environment is represented as a relational Markov decision process that can be compactly represented using first order predicate logic with variables. Policies learned are represented as logical decision trees. Trees that map examples to Q -values are denoted Q -trees, and trees that encode the optimality of examples are denoted P -trees.

Q -trees are learned using a modified version of the Q -learning algorithm from Chapter 2. Instead of updating the Q -function continuously during an episode, the algorithm instead generates examples. At the end of an episode, these examples are used to induce a Q -tree. This makes relational Q -learning an off-line learning technique.

While Q -trees can be trained to produce optimal behavior, they do not generalize well to domains similar to the training domain. For Blocks World, this means adding more blocks to the domain. This is because the Q -function in principle encodes the distance to the goal, and this distance may change when the domain is changed. Instead, the relational Q -learning algorithm can be extended to also learn P -trees. P -trees are induced over examples denoting the optimality of actions, which means that a structural pattern of optimality is found. For this reason, P -trees will perform better in other similar domains, and in some cases even produce general optimal behavior. Experiments performed supported this statement.

More experiments were conducted to compare the performance of tabular and relational Q -learning. As expected, relational Q -learning produces both reasonable and optimal behavior

using less training than tabular Q -learning. Furthermore, P -trees also outperform Q -trees in the domain in which they were learned, although only to a minor extent. Finally, the size of policies learned in a relational setting were much smaller than when using a tabular representation. This is because of the very noticeable abstractions, which the use of logic and background knowledge introduce when inducing Q -trees and P -trees.

Chapter 4

Hierarchical Reinforcement Learning

The previous chapter presented relational reinforcement learning as a technique for achieving state abstractions and generalizing policies to similar domains. It was shown that for relational domains, the learning rate of an agent can be significantly improved. The success of relational reinforcement learning depends, however, on the existence of structural similarities throughout the state/action space of a domain. It is easy to find a domain for which this is not the case. Consider the task of navigating through a maze. The reason that escaping a maze can be difficult is that seemingly similar scenarios requires different actions. For example, the optimal action when being in a corner with two paths leading east and west depends on the entire maze. The optimal action for another similar corner might be very different. Applying relational reinforcement learning to such a domain will only add the overhead of inducing logical decision trees at the end of an episode.

Unlike relational reinforcement learning, Hierarchical Reinforcement Learning (HRL) is not about generalizing policies to similar environments. Instead, the idea of HRL is to decompose the primary task of an agent into a hierarchy of subtasks. The benefits of such a decomposition can be summarized as

- achieving a better initial performance, and
- achieving state abstractions by eliminating irrelevant information and using “funnel” actions.

A task hierarchy restricts the actions of an agent at any time step. To some degree, this guides the agent towards its goal resulting in a better initial performance. As we will see, this kind of guidance can have the side-effect of slowing down exploration of some parts of the state space considerably. State abstractions are achieved by identifying relevant and irrelevant information for each individual subtask in the hierarchy. Furthermore, some tasks might move the environment from some large number of states to a small number of resulting states. Such tasks are denoted “funnel” actions.

Currently, the most popular method for hierarchical reinforcement learning seems to be the MAXQ value function decomposition (Dietterich, 2000). This method stands out because it

does not only provide a framework for procedural decomposition of a given task. It also provides a framework for decomposition of the value function, which leads to new opportunities for state abstraction.

Section 4.1 describes the motivation for hierarchical decomposition of a task and introduces the semi-Markov decision process that allows temporally extended actions. An intuitive approach to hierarchical reinforcement learning called hierarchical semi-Markov Q -learning is explained in Section 4.2. Following, Section 4.3 describes the MAXQ decomposition of the value function and explains how the decomposed value function can be learned. The most important part of hierarchical reinforcement learning, namely state abstractions, is described in Section 4.4, while the possibility of non-hierarchical execution follows in Section 4.5. A problem with some task hierarchies is the inability of exploring all states sufficiently often. This problem is described in Section 4.6. An overview of experiments performed to illustrate the performance of the MAXQ method is presented in Section 4.7. Section 4.8 describes other approaches to hierarchical reinforcement learning.

4.1 Task Decomposition

The Taxi domain introduced in Section 2.3 is well suited for hierarchical decomposition. In each episode, the taxi must navigate to the passengers location, pick up the passenger, navigate to the destination and put down the passenger. Decomposing this task displays the need for

- temporal abstraction,
- state abstraction, and
- subtask sharing.

Temporal abstraction covers that some tasks may be temporally extended, which means that they can take a different number of time steps to complete. For instance, the task of navigating to a specific location in the Taxi-grid can be viewed as a temporally extended task. Using temporal abstraction, the top-level of a hierarchical decomposition can often be expressed very simple.

State Abstractions can be achieved by eliminating irrelevant state variables inside a subtask. For instance, while the taxi is getting a passenger, the destination of the passenger is irrelevant, and when navigating to a specific destination, the only relevant information is the destination and the position of the taxi.

The taxi needs to navigate both to the passenger's location and to the passenger's destination. Thus, if the subtask of navigating is learned once, then this solution can be shared by both tasks. This illustrates the need for subtask sharing.

The set of individual subtasks in the Taxi domain can be defined as

- **Navigate** (t): move the taxi from its current position to one of the four target locations. The target location is indicated by the formal parameter t .
- **Get**: move the taxi to the passengers location and pick up the passenger.

- **Put**: move the taxi to the destination and put down the passenger.
- **Root**: the whole taxi task.

Each subtask is defined by its own subgoal and terminates when this subgoal is reached. A subtask is also defined by the possible actions (which might be other non-primitive subtasks) that it can perform. Such a definition is best illustrated with a task graph as shown in Figure 4.1¹.

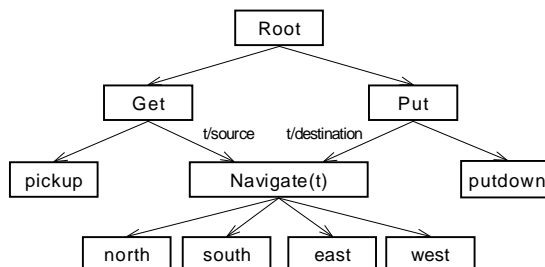


Figure 4.1: A task hierarchy for the Taxi domain.

The **Root** task of completing one episode is decomposed into the two subtasks **Get** and **Put**. **Get** is further decomposed into **Navigate(t)** and the primitive action **pickup**. **Put** is decomposed into **Navigate(t)** and the primitive action **putdown**. Finally, **Navigate(t)** is decomposed into the four primitive actions **north**, **south**, **east** and **west**. The execution of subtasks is similar to calling procedures in a programming language. When a subtask is invoked, control is simply shifted to its policy. The collection of individual policies is denoted a hierarchical policy.

4.1.1 Semi-Markov Decision Process

A traditional MDP cannot express temporal extended actions. In particular, each primitive action in an MDP takes exactly 1 time step to perform. When imposing a task hierarchy, subtasks might cover several time steps. A Semi-Markov Decision Process (SMDP) is an MDP in which actions can take a variable amount of time steps to complete. This change affects the transition probability distribution, as well as the definition of the value and action-value functions. Let the random variable N denote the number of time steps it takes to complete a particular action. The transition probability distribution can then be extended to a joint distribution over the resulting state and N , where $P(s', N|s, a)$ denotes the probability of observing state s' after N steps when performing action a in state s . A similar change can be made to the reward function where $R(s', N|s, a)$ denotes the reward received when s' is observed in N steps after performing action a in state s . The value function for a policy π can now be defined as the Bellman equation

$$V^\pi(s) = \sum_{s', N} P(s', N|s, \pi(s)) [R(s', N|s, \pi(s)) + \gamma^N V^\pi(s')] \quad (4.1)$$

¹The task decomposition of the Taxi domain used throughout this chapter is the one defined by Dieterich (2000).

Note that the discount factor (applied to the value of the resulting state s') is raised to the power of N . This discounts actions that take more than one time step to complete appropriately. The Bellman equation denoting the action-value function is defined in a similar fashion:

$$Q^\pi(s, a) = \sum_{s', N} P(s', N|s, a) [R(s', N|s, a) + \gamma^N Q^\pi(s', \pi(s'))] \quad (4.2)$$

Both $V^\pi(s)$ and $Q^\pi(s, a)$ can be rewritten as the sum of the expected reward for performing action $\pi(s)$ and the expected value of the resulting state s' :

$$V^\pi(s) = \bar{R}(s, \pi(s)) + \sum_{s', N} P(s', N|s, \pi(s)) \gamma^N V^\pi(s') \quad (4.3)$$

$$Q^\pi(s, a) = \bar{R}(s, a) + \sum_{s', N} P(s', N|s, a) \gamma^N Q^\pi(s', \pi(s')) \quad (4.4)$$

where $\bar{R}(s, a)$ is the expected reward with respect to s' and N for performing action a in state s .

For episodic tasks with $\gamma = 1$, an SMDP is equivalent to an MDP. In this case, future rewards are not discounted, which makes the number of steps used by an action irrelevant. Furthermore, for primitive actions where $N = 1$, we will suppress N in the notation when denoting the transition probability distribution and the reward function.

4.1.2 Definition of a Subtask

In general, an MDP M can be decomposed into a set of subtasks $\{M_0, \dots, M_n\}$ with the convention that M_0 is the root-task. For instance, in Figure 4.1, M_0 is `Root`, M_1 is `Get` and so on. Solving M_0 is equivalent to solving the original MDP M . To avoid cluttering the notation, we will sometimes denote a subtask M_i simply as i .

A subtask is defined by its own subtasks and a termination predicate. The termination predicate partitions the state space into a set of active states and a set of terminal states. Furthermore, each terminal state is assigned a numerical value indicating how desirable it is to terminate execution in that state.

Definition 4. *An unparameterized subtask M_i is a 3-tuple $\langle T_i, A_i, \tilde{R}_i \rangle$ defined as:*

- $T_i(s)$: the termination predicate over the set of states S . The predicate partitions S into a set of active states S_i , and a set of terminal states, which we will denote T_i (without parameters). Subtask M_i can only be executed if the current state s is in S_i .
- A_i : the set of actions available in subtask i . $A_i(s)$ denotes the actions available in state s .
- $\tilde{R}_i(s'|s, a)$: the pseudo-reward function, which specifies a pseudo-reward for each transition from a state $s \in S_i$ to a state $s' \in T_i$.

Each primitive action a from a subtask M is a primitive subtask in the decomposition such that a is always executable, it always terminates immediately after execution, and its pseudo-reward function is uniformly zero.

If a subtask has formal parameters, then each possible binding of actual values specifies a distinct subtask (i.e. the actual values are part of the name of the subtask). In practice, of course, parameterized subtasks are implemented by extending the definition of the termination predicate and reward function to also encompass the actual parameter values.

The need to specify pseudo-rewards is dependent on the real reward function. If rewards are only given to the agent when the final goal state is reached, then some intermediate subtasks might never receive any feedback. As an example, consider that our taxi only received a reward for putting down the passenger at the end of an episode. This reward would propagate up to the `Root` task, but not down to the `Get` subtask. On the other hand, if rewards or penalties are given for all primitive actions, then the specification of pseudo-rewards is not necessary. Pseudo-rewards can, however, be used to speed up learning or change the optimal behavior in a subtask. This is further explained in Section 4.3.

4.1.3 Hierarchical Policies

The collection of individual subtask policies for a hierarchy is denoted a hierarchical policy. A hierarchical policy π is thus defined as

$$\pi = \{\pi_0, \dots, \pi_n\} \quad (4.5)$$

where n is the number of subtasks in the hierarchy. As in the previous chapters, a policy takes a state and returns an action. If a subtask contains parameters, then its policy must also take these parameters as input. In such case, the definition of a policy is $\pi(s, f)$ where f is the bindings of actual parameters. A hierarchical policy can be executed using a stack that initially contains the root task. At each time step, the task at the top of the stack is examined. If it is a primitive subtask, then it is executed. If it is a composite subtask, then the task denoted by the composite subtask's policy is pushed onto the stack. If this is a primitive action, then it is executed, and so on. Table 4.1 shows the pseudo-code for executing a hierarchical policy.

After the execution of a primitive subtask, the algorithm checks if any tasks on the stack have reached a terminal state (lines 18-22). If a task M' has terminated, then it is popped off the stack together with all tasks above M' on the stack. As an example, consider a Taxi domain where the passenger can cancel a ride while navigating to the destination. If this happens, then the `Root` task has entered a terminal state. All subtasks invoked by `Root` or its descendants must therefore also terminate, which is why they are popped off the stack.

At any time step t , the choice of the next primitive action to be executed is affected by the current contents on the stack. This means that a hierarchical policy is non-Markovian with respect to the original MDP. Since actions are chosen with respect to both the current state s and the contents on the stack K , Dietterich (2000) defines a hierarchical value function $V^\pi(\langle s, K \rangle)$. This value function gives the expected cumulative reward of following the hierarchical policy π starting in state s with stack contents K .

To avoid the extra space requirements (and the consequence of increased learning difficulty), Dietterich also defines a so-called projected value function of a hierarchical policy.

```

1: procedure EXECUTEHIERARCHICALPOLICY( $\pi$ )
2:    $s_t$  is the state of the world at time  $t$ .
3:    $K_t$  is the state of the execution stack at time  $t$ .
4:   Let  $t = 0$ ;  $K_t =$  the empty stack; observe  $s_t$ .
5:   Push  $(0, nil)$  onto stack  $K_t$  (invoke the root task with no parameters).
6:   repeat
7:     while ( $top(K_t)$  is not a primitive action)
8:       Let  $(i, f_i) := top(K_t)$ , where
9:          $i$  is the name of the “current” subroutine, and
10:         $f_i$  gives the parameter bindings for  $i$ .
11:       Let  $(a, f_a) := \pi(s, f_i)$ , where
12:         $a$  is the action, and
13:         $f_a$  gives the parameter bindings chosen by policy  $\pi_i$ .
14:       Push  $(a, f_a)$  onto the stack  $K_t$ .
15:     end while
16:     Let  $(a, nil) := pop(K_t)$  be the primitive action on the top of the stack.
17:     Execute primitive action  $a$ , observe  $s_{t+1}$ , and receive  $R(s_{t+1}|s_t, a)$ .
18:     if (any subtask on  $K_t$  is terminated in  $(s_{t+1})$ ) then
19:       Let  $M'$  be the terminated subtask closest to the root on the stack.
20:       while ( $top(K_t) \neq M'$ ) do  $pop(K_t)$ 
21:        $pop(K_t)$ .
22:     end if
23:      $K_{t+1} := k_t$  is the resulting execution stack.
24:   until  $K_{t+1}$  is empty
25: end

```

Table 4.1: Pseudo-code for execution of a hierarchical policy.

Definition 5. *The projected value function of a hierarchical policy π_i on subtask M_i , denoted $V^\pi(i, s)$, is the expected cumulative reward of executing π_i (and the policies of all descendants of M_i) starting in state s with an empty stack until M_i terminates.*

The projected value function for a task disregards content pushed onto the stack by any of its ancestors. The value $V^\pi(i, s)$ can be thought of as the value of state s when following policy π given that execution stops when subtask i terminates.

We can also define the projected action-value function as $Q^\pi(i, s, a)$ where i is the current task, s is the current state and a is the subtask to be executed. Similarly, the value $Q^\pi(i, s, a)$ can be thought of as the value of performing action a in state s and then following policy π until subtask i terminates. We will formalize both the projected value function and the projected action-value function in Section 4.3.

4.2 Hierarchical Semi-Markov Q -Learning

A primary concept in the MAXQ method is the decomposition of the projected value function. In this section, however, we will look at a straight-forward way of solving a task hierarchy without decomposing the value function. This approach is called Hierarchical Semi-Markov Q -learning (HSMQ). We will do this to be able to illustrate the differences between this approach and MAXQ. Furthermore, the use of HSMQ follows more intuitively from flat (and relational) reinforcement learning.

In principle, there are two ways to solve a task hierarchy. The first way is to start by learning optimal policies for the subtasks at the bottom of the hierarchy. Afterwards, optimal policies for the parents of these tasks are learned. This continues until the root is reached. Doing so effectively reduces each subtask to a primitive action for its parent in the hierarchy. The parent will only observe one kind of behavior from its subtasks, namely optimal behavior.

The second approach is to simultaneously learn optimal policies for the entire hierarchy. This is the approach used by both HSMQ and MAXQ. In this case, parent tasks observe changing behavior from their subtasks as these explore the state space and eventually converge to an optimal policy. Of course, a parent task will only converge to an optimal policy when its subtasks have converged too. Simultaneously learning the entire hierarchy puts an extra requirement on the exploration policy used in each subtask. While traditional Q -learning “only” requires that all states are visited infinitely often, convergence now requires that the exploration policy used is Greedy in the Limit of Infinite Exploration (GLIE). A GLIE policy is a policy that, in the limit of infinite exploration, eventually becomes greedy with respect to Q . Only when a subtask policy is greedy will the parent task observe optimal behavior consistently. Boltzmann exploration (see Section 2.2) can be used to create a GLIE policy by continuously decreasing the temperature.

```

1: function HSMQ(State  $s$ , Subtask  $p$ )
2:   Let  $TotalReward := 0$ 
3:   while ( $p$  is not terminated)
4:     Choose subtask  $a := \pi_e(s)$  according to exploration policy  $\pi_e$ 
5:     Execute  $a$  and observe resulting state  $s'$ 
6:     if ( $a$  is primitive) then
7:       Observe one-step reward  $r := R(s'|s, a)$ 
8:     else
9:        $r := \text{HSMQ}(s, a)$ , which invokes subroutine  $a$  and
10:      returns the total reward received while  $a$  executed.
11:     end if
12:      $TotalReward := TotalReward + r$ 
13:     Update  $\hat{Q}(p, s, a) := (1 - \alpha)\hat{Q}(p, s, a) + \alpha[r + \max_{a'} Q(p, s', a')]$ 
14:   end while
15:   return  $TotalReward$ 
16: end

```

Table 4.2: Pseudo-code for execution of a hierarchical policy.

Table 4.2 shows the pseudo-code for the HSMQ algorithm. The algorithm covers one episode of learning. At first, the algorithm is called with the initial state and the `Root` task. The variable `TotalReward` is initialized to hold the sum of the rewards received during the `Root` task. While `Root` is not terminated, a subtask is repeatedly chosen using a GLIE exploration policy. If the subtask is primitive then it is executed and the immediate reward is observed. Otherwise, if a is non-primitive, then it is executed by calling HSMQ recursively (the same way `Root` was called), which returns the total reward received during the executing of the subtask. Afterwards, `TotalReward` is increased by the observed reward. The total reward is then used to update the current approximation of Q .

4.3 MAXQ Value Function Decomposition

When using the HSMQ algorithm, a task hierarchy is treated as a set of independent Q -learning problems. Each subtask contains all the values needed to completely specify its own

policy. In other words, HSMQ provides a procedural decomposition of the learned policy into policies for each subtask. However, there is bound to exist a dependency between the value function of a task and its subtasks. For instance, the value of performing the task `Get` in the Taxi domain must somehow be related to the value of performing its child tasks `Navigate` and `pickup`.

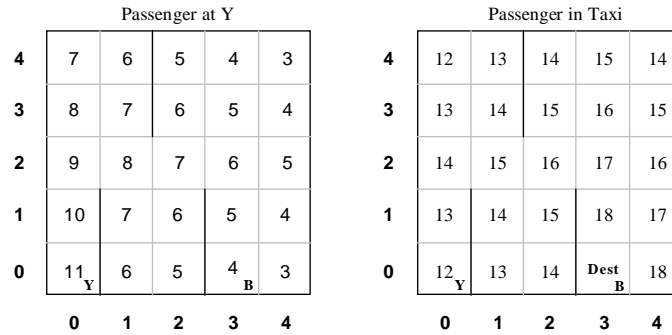


Figure 4.2: Value function for the case where the passenger is at (0, 0) (location Y) and wishes to get to (3, 0) (location B).

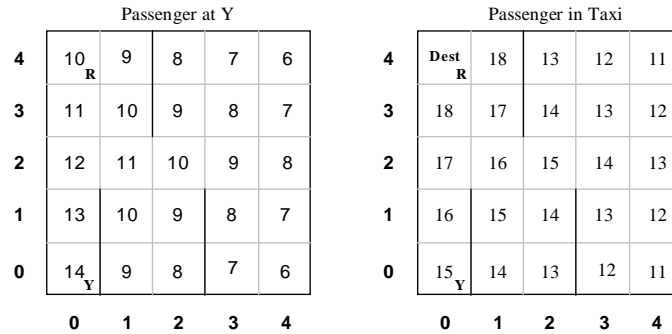


Figure 4.3: Value function for the case where the passenger is at (0, 0) (location Y) and wishes to get to (0, 4) (location R).

Figure 4.2 illustrates part of a projected value function for the Taxi hierarchy. In this case, the passenger is at location Y and wishes to get to location B. The left-side maze shows the state values while getting the passenger, and the right-side maze shows the state values while driving the passenger to the destination. For instance, the value of a state, where the taxi is at location (0, 2) and on its way to get the passenger is 9.

Figure 4.3 illustrates a similar value function, where the only difference is that the passenger wishes to get to location R. Comparing the value functions of these two scenarios, we see that there is no similarity between the values in the right-side mazes. However, the values in the left-side mazes are the same except for an offset of 3. This is because that the left-side mazes really reflect the same subgoal of moving to location R and picking up the passenger. They differ only in what happens after the passenger has been picked up. In Figure 4.2, the destination is 7 steps away, and in Figure 4.3 the destination is 4 steps away. The difference $7 - 4 = 3$ accounts for the difference between the values in the two mazes.

The motivation behind decomposing the value function is to exploit such regularities by representing the left-side value function only once. Notice that decomposing the value function does not enable a more compact representation in itself. Instead, the decomposition enables state abstractions over “funnel” actions, something which is not possible when using HSMQ. This is further explained in Section 4.4.

4.3.1 Definition of the Value Function Decomposition

In general, the MAXQ method decomposes the projected action-value function $Q(i, s, a)$ (where i is the current subtask, s is the state, and a is the action to be performed) into the sum of the following two components:

- the expected total reward received while executing subtask a in state s , and
- the expected total reward of following the hierarchical policy π *after a has returned* until parent task i terminates.

For a primitive action a , the first component is just the expected immediate reward of performing a in s . For a composite action, Dietterich (2000) shows that this component is instead the projected value function $V^\pi(a, s)$ by proving the following theorem:

Theorem 6. *Given a task hierarchy over tasks M_0, \dots, M_n and a hierarchical policy π , each subtask M_i defines an SMDP with states S_i , actions A_i and the transition probability distribution P_i . The expected reward function of M_i , denoted $\bar{R}_i(s, a)$, is defined as $\bar{R}_i(s, a) = V^\pi(a, s)$ where*

- $V^\pi(a, s)$ is the projected value function for child task a in state s , and
- If a is a primitive action then $V^\pi(a, s)$ is defined as the expected immediate reward of executing a in s : $V^\pi(a, s) = \sum_{s'} P(s'|s, a)R(s'|s, a)$.

The theorem states that the expected reward received by subtask M_i , when executing a composite subtask M_a , is the projected value function $V(a, s)$. If a is primitive, then the reward received is instead the expected immediate reward (as in flat Q -learning). As a consequence, we can define the action-value function of policy π when executing action a from subtask i in state s as

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s', N} P_i(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')) \quad (4.6)$$

which has the same form as the Bellman equation for an SMDP (see Equation 4.4). If a is primitive, then $V^\pi(a, s)$ equals the expected immediate reward, thereby making equations (4.6) and (4.4) identical.

Recall that Definition 5 defined a projected value function $V(a, s)$ to be the expected cumulative reward received *until subtask a terminates*. This means that the right-most term of Equation (4.6) denotes the value of completing task M_i after executing a in state s . This term can be encapsulated in a new function called the completion function:

$$C^\pi(i, s, a) = \sum_{s', N} P_i(s', N|s, a) \gamma^N Q^\pi(i, s', \pi(s')) \quad (4.7)$$

By substituting the completion function into Equation (4.6), we get the following definition of the action-value function:

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a) \quad (4.8)$$

Furthermore, the definition of the value function $V^\pi(i, s)$ can be re-expressed as

$$V^\pi(i, s) = \begin{cases} Q^\pi(i, s, \pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s, a) R(s'|s, a) & \text{if } i \text{ is primitive} \end{cases} \quad (4.9)$$

For a composite subtask i , this recursive definition states that the value of a state s can be computed as the following equation:

$$V^\pi(i, s) = Q^\pi(i, s, \pi_i(s)) = V^\pi(\pi_i(s), s) + C^\pi(i, s, \pi_i(s)) \quad (4.10)$$

In other words, to find the value of a state s in subtask i given a hierarchical policy π , we must simply 1) find the value of state s in the subtask denoted by $\pi_i(s)$, and 2) add the value of completing subtask i after subtask $\pi_i(s)$ has terminated. If $\pi_i(s)$ is also a composite action, then this term can be further decomposed in the same way. The recursiveness of V ends in the bottom of the hierarchy when a primitive action is encountered (as defined in Equation 4.9).

Dietterich refers to equations (4.7), (4.8) and (4.9) as the decomposition equations for the MAXQ hierarchy under a fixed hierarchical policy π . These equations recursively decompose the projected value function $V^\pi(0, s)$, for the root task M_0 , into the projected value functions for the subtasks M_1, \dots, M_n and the completion functions $C^\pi(j, s, a)$ for $j = 0, \dots, n$. This means that a complete specification of the decomposed Q and V functions requires exactly the storage of

- the completion value $C(i, s, a)$ for all composite subtasks i , states s and subtasks a , and
- the value $V(i, s)$ for all primitive subtasks i and states s .

By storing these values, the value of any combined state/action pair, of any subtask in the hierarchy, can be computed by the use of the decomposition equations.

4.3.2 MAXQ Graphs

To make it easier to understand the decomposition equations, a task hierarchy can be illustrated as a MAXQ graph. Figure 4.4 illustrates a MAXQ graph for the Taxi domain. The graph contains two kinds of nodes, Max nodes and Q nodes.

Max nodes corresponds to subtasks in the task hierarchy. There is one Max node for each composite and primitive subtask. Each primitive Max node i stores the value of $V^\pi(i, s)$ for all $s \in S_i$.

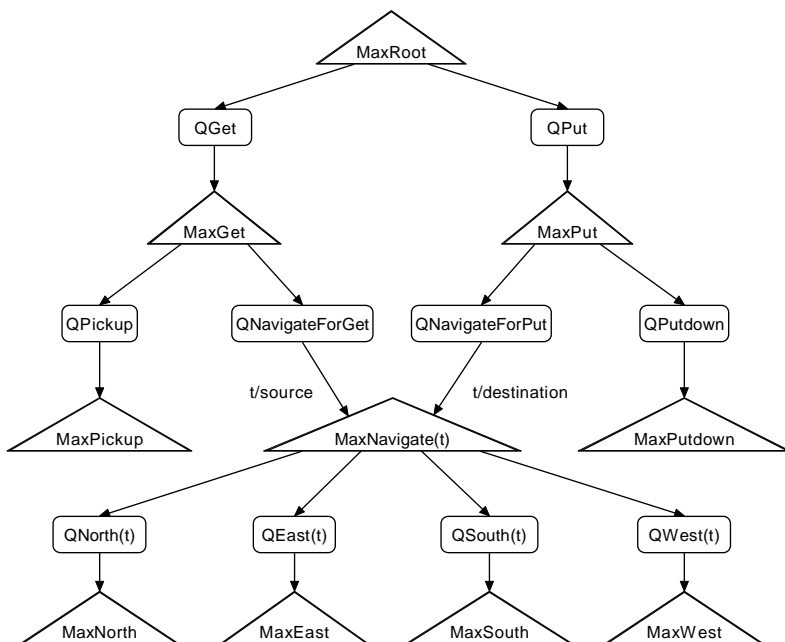


Figure 4.4: A MAXQ graph for the Taxi domain. Max nodes correspond to the subtasks in the domain, and Q nodes correspond to the actions available for each subtask.

Q nodes corresponds to the actions that are available for each subtask. For instance, the available actions from subtask `Get` are `Pickup` and `Navigate(t/source)`. These actions are modelled as the Q nodes `QPickup` and `QNavigateForGet` in the MAXQ graph. Each Q node for parent task i , state s and subtask a stores the value of $C^\pi(i, s, a)$. A parent task may execute a subtask multiple times before it terminates.

The purpose of each Max node i is to compute the projected value function $V^\pi(i, s)$ for all $s \in S_i$. For primitive Max nodes, such as `MaxPickup` and `MaxPutdown` in Figure 4.4, this information is stored directly in the node. For composite Max nodes, the information must be computed. To compute the value $V^\pi(i, s)$, the Max node i consults its policy π_i and finds that the next action is $\pi_i(s)$. It then queries the Q node corresponding to $\pi_i(s)$ for the value $Q^\pi(i, s, \pi_i(s))$.

The Q node does not directly store this value. It only stores $C^\pi(i, s, \pi_i(s))$, the value of completing subtask i after $\pi_i(s)$ has been executed. To find the value of actually executing $\pi_i(s)$, the Q node queries its child Max node for $V^\pi(\pi_i(s), s)$. If $\pi_i(s)$ corresponds to a primitive Max node, then $V^\pi(\pi_i(s), s)$ can be looked up. Otherwise $\pi_i(s)$ is composite and $V^\pi(\pi_i(s), s)$ must be computed by querying further down the MAXQ graph. Afterwards, the Q value

$$Q^\pi(i, s, \pi_i(s)) = V^\pi(\pi_i(s), s) + C^\pi(i, s, \pi_i(s)) \quad (4.11)$$

is returned to the Max node i . Since $V^\pi(i, s) = Q^\pi(i, s, \pi_i(s))$, the Max node has finished its computation.

For a more concrete example, let s be the state illustrated in Figure 4.5, and assume that an optimal policy π^* has been previously learned. From state s , it requires 1 step to reach the passenger, 1 step to pick up the passenger, 7 steps to reach the destination, and 1 step to put down the passenger. Since each of these steps has a penalty of -1 , and because the taxi receives a reward of 20 after delivering the passenger, the value of state s is 10. The complete set of recursive computations needed for this conclusion looks as follows:

$$\begin{aligned}
 V^*(\text{North}, s) &= -1 \\
 Q^{\pi^*}(\text{Nav}(\text{R}), s, \text{North}) &= V^*(\text{North}, s) + C^*(\text{Nav}(\text{R}), s, \text{North}) = -1 + 0 = -1 \\
 V^*(\text{Nav}(\text{R}), s) &= Q^*(\text{Nav}(\text{R}), s, \text{North}) = -1 \\
 Q^{\pi^*}(\text{Get}, s, \text{Nav}(\text{R})) &= V^*(\text{Nav}(\text{R}), s) + C^*(\text{Get}, s, \text{Nav}(\text{R})) = -1 + -1 = -2 \\
 V^*(\text{Get}, s) &= Q^*(\text{Get}, s, \text{Nav}(\text{R})) = -2 \\
 Q^*(\text{Root}, s, \text{Get}) &= V^*(\text{Get}, s) + C^*(\text{Root}, s, \text{Get}) = -2 + 12 = 10 \\
 V^*(\text{Root}, s) &= Q^*(\text{Root}, s, \text{Get}) = 10
 \end{aligned}$$

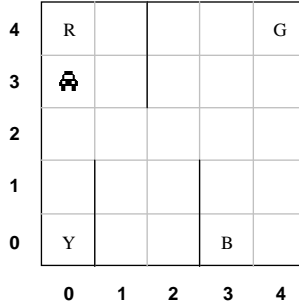


Figure 4.5: A Taxi domain scenario. The taxi is at $(0,3)$ and the passenger is at $(0,4)$ (location R). The destination is $(3,0)$ (location B).

In general, the MAXQ value function decomposition takes the form

$$V^\pi(0, s) = V^\pi(a_m, s) + C^\pi(a_{m-1}, s, a_m) + \dots + C^\pi(a_1, s, a_2) + C^\pi(0, s, a_1) \quad (4.12)$$

where a_1, \dots, a_m is the “path” of the Max nodes, from the root-node 0 to the primitive action a_m , chosen by the hierarchical policy π . This concludes the description of the representation of the value function when using the MAXQ value function decomposition.

4.3.3 Different Kinds of Optimality

Before proceeding to describe an algorithm for learning an optimal policy using the value function decomposition, we must first define the meaning of optimality given the introduction of a task hierarchy. Of course, without decomposing a task, a truly optimal policy can be learned using traditional flat Q -learning. However, imposing a hierarchy puts two constraints on the policies representable by the hierarchy:

- Within a subtask, some primitive actions may not be allowed. In the Taxi hierarchy, for instance, the taxi cannot perform the actions `pickup` or `putdown` during the `Navigate` subtask.
- The policy learned for task M_j must involve the policies learned for its child tasks $\{M_{j_0}, \dots, M_{j_k}\}$. When the policy for subtask M_{j_i} is invoked, it will run until a terminal state in T_{j_i} is encountered. This means that the policy for task M_j must pass through some subset of the terminal states of its subtasks $\{T_{j_1}, \dots, T_{j_k}\}$.

The impact of these two constraints depends entirely on the specification of the task hierarchy. The taxi hierarchy discussed so far is not affected by them, and is capable of representing a truly optimal policy (in Section 4.7, however, we will change this fact). A policy that is as optimal as possible, given the constraints of a hierarchy, is said to be hierarchical optimal.

A goal of the MAXQ method is subtask sharing (see Section 4.1). To achieve this, individual subtasks must be context-free. For instance, the task `Navigate(t)` is context-free because of its target location parameter `t`. The task would not be context-free if the target-location was implicit (given the parent task executing `Navigate`). To achieve total subtask independence, an even weaker form of optimality must be pursued. This form of optimality is called recursive optimality.

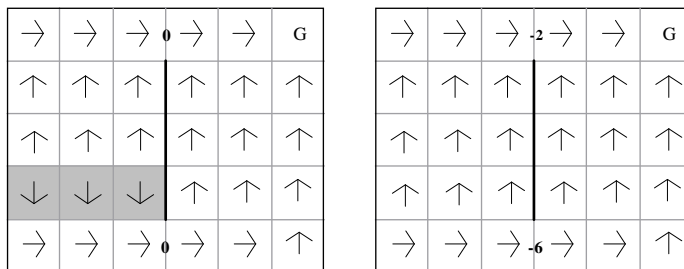


Figure 4.6: A domain illustrating recursive optimality. The agent must leave the left room and go to the goal square `G`. The policy illustrated in the left maze is recursively optimal but not hierarchically optimal. The shaded cells indicate points where the locally optimal policy is not globally optimal. The policy illustrated in the right maze is hierarchical optimal due to the specification of pseudo-rewards.

The left-side of Figure 4.6 illustrates the effects of pursuing only recursive optimality. The figure shows a grid world consisting of two rooms with two doors separating them. The task of an agent in this world is to reach the goal square `G`. The primitive actions in this world are `North`, `South`, `East` and `West`, and each give a penalty of -1 . A hierarchy is imposed such that the task of an agent is split into the subtasks `ExitLeftRoom` and `GotoGoal`. `ExitLeftRoom` is only available when the agent is in the left room, and terminates when the right room is entered. `GotoGoal` is only available in the right room, and terminates when `G` is reached. During each subtask, any primitive action can be executed.

The arrows in the figure represents the recursively optimal policy given the described hierarchy. The illustrated policy is locally optimal for both subtasks, but clearly not hierarchical optimal. The shaded cells indicate points where the locally optimal policy is not globally optimal.

In this case, it is possible to specify pseudo-rewards to reach hierarchical optimality. If the agent exits the left room through the lower door, we will assign it a pseudo-reward of -6 because this is the number of steps needed afterwards to reach the goal. Similarly, the agent will receive a pseudo-reward of -2 if it exits the left room through the upper door. For now, pseudo-rewards can be thought of simply as extra rewards besides the traditional immediate rewards. The right-side of Figure 4.6 shows a hierarchical optimal policy for the domain using the described pseudo-rewards.

4.3.4 The MAXQ-Q Learning Algorithm

Dietterich (2000) presents two learning algorithms for the decomposed value function. The first one, called MAXQ-0, can only be applied when the pseudo-reward function \tilde{R} is always zero. The second and more general algorithm is called MAXQ-Q, which works with any specification of the pseudo-reward function. We will skip the description of MAXQ-0, since this is just a special case of MAXQ-Q. The overall goal of MAXQ-Q is to learn a) the completion value $C(i, s, a)$ for each composite subtask i , state s and subtask a , and b) the value $V(i, s)$ for each primitive subtask i and state s .

As mentioned in Section 4.3.3, pseudo-rewards can be thought of as simply extra rewards. As such, an approach towards incorporating them into a learning algorithm would be to just add them to the corresponding immediate rewards whenever an update of either C or V is performed. However, this would have the effect of changing the original MDP to have a different reward function. Furthermore, the pseudo-rewards for a single subtask could “contaminate” the policies learned throughout the hierarchy. As an example, we will extend the domain from Figure 4.6 as illustrated in Figure 4.7. To clarify the point, let the pseudo-reward for exiting the middle room through the upper door be 100, and the pseudo-reward for exiting the middle room through the lower door be 0. The immediate reward for reaching **G** will remain 20. Now consider the lower left room. After exiting this room through the upper door, there will remain 11 steps yielding a penalty of -11 . Furthermore, adding the pseudo-reward of 100 and a goal-state reward of 20 to this value yields a total reward of 109. If the lower left room is exited through the lower door, then a penalty of -10 is received before the goal is reached. The consequence is that the pseudo-rewards, specified for the subtask of exiting the middle room, has changed the optimal policy for the subtask of exiting the lower left room. This policy is no longer optimal in the shaded cells. In fact it gets even worse, because the optimal policy is no longer to reach the goal state. Since the pseudo-reward is greater than the reward for entering the goal state, the optimal behavior for the agent will become to continuously enter and exit the right room.

This example clearly shows that pseudo-rewards cannot simply be added to immediate rewards. The problem can be solved by learning one completion function to be used “inside” each subtask, and a separate completion function to be used “outside” each subtask. The external completion function $C(i, s, a)$ is the one discussed so far. It is computed without reference to pseudo-rewards, and denotes the expected reward for completing task M_i after performing action a in state s , and then following the learned policy for M_i . It is used by parent tasks to compute $V(i, s)$, the expected value of performing task M_i in state s .

The internal value function, denoted $\tilde{C}(i, s, a)$, is computed by adding pseudo-rewards to the real rewards. It is used to find the locally optimal policy for each subtask M_i . In effect, MAXQ-Q should learn C and \tilde{C} such that

↓	↓	↓	↓	→	→	→	100	←	←	G
↓	↓	↓	↓	↑	↑	↑	↑	↑	↑	↑
→	→	→	→	↑	↑	↑	↑	↑	↑	↑
↑	←	←	←	↑	↑	↑	↑	↑	↑	↑
↑	↑	↑	↑	↑	↑	↑	0	↑	↑	↑
↑	↑	↑	↑	→	→	→	→	→	→	↑

Figure 4.7: Pseudo-rewards can “contaminate” the hierarchy if used simply as extra rewards. The pseudo-reward for exiting the middle room using the upper door is 100, while the immediate reward for reaching **G** remains 20. The result is that the hierarchical policy is non-optimal in the shaded cells.

- $\tilde{C}(i, s, a)$ denotes the pseudo-reward “contaminated” completion function over the locally optimal policy for task M_i .
- $C(i, s, a)$ denotes the “clean” completion function over the locally optimal policy for task M_i .

In other words, the locally optimal policy for subtask M_i is found using pseudo-rewards to contaminate $\tilde{C}(i, s, a)$. Then, $C(i, s, a)$ is learned to be the clean completion function over the found locally optimal policy. The policies for the parents of M_i is learned using the clean completion function to avoid cascading contamination. As a result, local optimality can be achieved with pseudo-rewards without worrying about changing the behavior of other tasks in the hierarchy.

Figure 4.3 shows the pseudo-code for the MAXQ-Q algorithm. Invoking $\text{MAXQ-Q}(i, s)$ returns the sequence of states visited by subtask i when being executed from state s . This sequence is maintained in the variable seq . The algorithm first checks if i is primitive. If this is the case, then i is directly executed. The observed immediate reward is used to update $V(i, s)$. Indeed, this part of the algorithm can be viewed as learning the immediate reward function for the original MDP.

If a subtask i is not primitive, then the algorithm enters a while loop which runs until i terminates. During an iteration inside the while loop, an action a is chosen according to the exploration policy $\pi_e(i, s)$. Action a is executed by calling MAXQ-Q recursively, which results in the sequence of states visited during the execution of a . The resulting state s' is then observed. Following, in line 14, the optimal action a^* in the next time step is predicted. *This prediction is made using the internal completion function \tilde{C} for subtask i .* We now have all the information needed to begin updating both the internal and external completion functions.

To speed up learning in nodes at the top of the task hierarchy, the completion functions are updated for each state s visited during the execution of the chosen action a . Dietterich refers to this as “all-states-updating”. The reasoning is that the execution of a will move the environment through a sequence of states s_1, \dots, s_n, S_{n+1} where S_{n+1} is equal to the

```

1: function MAXQ-Q(MaxNode  $i$ , State  $s$ )
2:   Let  $seq = ()$  be the sequence of states visited while executing  $i$ 
3:   if ( $i$  is a primitive MaxNode)
4:     Execute  $i$ , receive  $r_t = R(s'|s, a)$ , and observe result state  $s'$ 
5:      $V_{t+1}(i, s) := (1 - \alpha_t(i)) \cdot V_t(i, s) + \alpha_t(i) \cdot r_t$ 
6:     Push  $s$  onto the beginning of  $seq$ 
7:   else
8:     Let  $count = 0$ 
9:     while ( $T_i(s)$  is false)
10:      Choose an action  $a$  according to the current exploration policy  $\pi_\omega(i, s)$ 
11:      Let  $childSeq = \text{MAXQ-Q}(a, s)$  where  $childSeq$  is the sequence of states
12:      visited executing action  $a$  (in reverse order)
13:      Observe result state  $s'$ 
14:      Let  $a^* = \arg \max_{a'} [\tilde{C}_t(i, s', a') + V_t(a', s')]$ 
15:      Let  $N = 1$ 
16:      for (each  $s$  in  $childSeq$ ) do
17:         $C_{t+1}(i, s', a) := (1 - \alpha_t(i)) \cdot C_t(i, s', a) + \alpha_t(i) \cdot \gamma^N \text{externalValue}(s')$ 
18:         $\tilde{C}_{t+1}(i, s', a) := (1 - \alpha_t(i)) \cdot \tilde{C}_t(i, s', a) + \alpha_t(i) \cdot \gamma^N \text{internalValue}(s')$ 
19:        where
20:           $\text{externalValue}(s') = [C_t(i, s', a^*) + V_t(a^*, s')]$ , and
21:           $\text{internalValue}(s') = [\tilde{R}_i(s') + \tilde{C}_t(i, s', a^*) + V_t(a^*, s')]$ 
22:         $N := N + 1$ 
23:      end for
24:      Append  $childSeq$  onto the front of  $seq$ 
25:       $s := s'$ 
26:    end while
27:  end if
28:  Return  $seq$ 
29: end

30: //main program
31: Initialize  $V(i, s)$ ,  $C(i, s, a)$  and  $\tilde{C}(i, s, a)$  arbitrarily
32: MAXQ-Q(root node 0, starting state  $s_0$ )

```

Table 4.3: The MAXQ-Q learning algorithm.

resulting state s' . Since all subtasks are Markovian, executing a in s_2, s_3 , or any state up to (and including) s_n , would result in the same state s' .

Before updating the completion functions, the internal and external values of executing a^* in s' is first computed. The computation of the internal value includes possible values received by the pseudo-reward function \tilde{R} . Notice that the action a^* is also used in the computation of the external value—even though this action might not be the optimal action in the next time step according to the external completion function C . Both completion functions are then updated, discounting the computed values properly. The primary observation here is that the action a^* , which is optimal in the next time step according to \tilde{C} , might not be optimal according to C . Nevertheless, a^* is used to update C , which results in a SARSA like algorithm (see Table 2.2). In effect, C will converge to the non-contaminated completion function over the locally optimal policies learned by \tilde{C} .

The updates of C and \tilde{C} requires the computation of $V_t(i, s')$. In Section 4.3.2 we described how this value could be computed for a fixed hierarchical policy with the recursive decomposition functions. The problem is that, during learning, there exists no fixed hierarchical policy. Furthermore, because \tilde{C} should converge to the locally optimal policy for each subtask, actions should always be chosen greedily during the recursive computation of $V_t(i, s')$ (as apposed to be chosen by a fixed policy). This leads to the following modified definition of the decomposition functions:

$$V_t(i, s) = \begin{cases} \max_a Q_t(i, s, a), & \text{if } i \text{ is composite} \\ V_t(i, s) \text{ (lookup)}, & \text{if } i \text{ is primitive} \end{cases} \quad (4.13)$$

$$Q_t(i, s, a) = V_t(a, s) + C_t(i, s, a) \quad (4.14)$$

The computation of $V_t(i, s)$ using the above equations requires a complete search of all paths through the MAXQ graph starting at node i and ending at the leaf nodes. Fortunately, MAXQ graphs are normally small of size, so this does not affect the performance of MAXQ-Q noticeably. Table 4.4 shows the function EVALUATEMAXNODE(i, s), which, among other things, calculates $V_t(i, s)$. For composite tasks, the algorithm chooses the action a_{max} that maximizes $V_t(a, s) + \tilde{C}(i, s, a)$ for any $a \in A(i)$. The algorithm then uses this action to calculate the uncontaminated value $V_t(a_{max}, s) + C(i, s, a_{max})$. The uncontaminated value is returned together with the primitive action reached at the bottom of the MAXQ graph (corresponding the leaf). This action is returned to allow non-hierarchical execution, which will be further explained in Section 4.5. Again, \tilde{C} is used to select actions because it will eventually represent the locally optimal policy. The value returned to other subtasks are, however, based on the uncontaminated completion function C to avoid cascading pseudo-reward contamination.

To avoid cluttering the pseudo-code, “ancestor” termination (as described in Section 4.1.3) is not shown in the MAXQ-Q algorithm in Table 4.3. However, “Ancestor” termination should of course be included in any “real” implementation of the algorithm. Furthermore, MAXQ-Q requires that the exploration policy π_ω is not only a GLIE policy, but an ordered GLIE policy, where ω denotes the ordering of actions used to break ties. An ordered GLIE is required because, in general, each subtask M_i will have a choice between many different locally optimal policies. These different locally optimal policies will all achieve the same locally optimal value function, but they may result in different probability transition functions $P(s', N|s, i)$. As a consequence, the SMDP problems at the level above subtask M_i will differ depending on which of the different locally optimal policies is chosen by subtask M_i . An ordering of actions ω ensures that consistent behavior is observed from subtask M_i .

```

1: function EVALUATEMAXNODE(MaxNode  $i$ , State  $s$ )
2:   if ( $i$  is a primitive MaxNode)
3:     Return  $\langle V_t(i, s), i \rangle$ 
4:   else
5:     for ( $j \in A_i(s)$ )
6:       Let  $\langle V_t(j, s), a_j \rangle = \text{EVALUATEMAXNODE}(j, s)$ 
7:     end for
8:     Let  $a_{max} = \arg \max_j V_t(j, s) + \tilde{C}_t(i, s, j)$ 
9:     Return  $\langle V_t(a_{max}, s) + C_t(i, s, a_{max}), a_{max} \rangle$ 
10:  end if
11: end

```

Table 4.4: Pseudo-code for computing $V_t(i, s)$ for Max node i and state s .

4.4 State Abstractions

One of the reasons to introduce hierarchical reinforcement learning is to create opportunities for state abstractions. In general, for task hierarchies generated by hand, it can be a straightforward task to simply begin removing irrelevant state variables from different subtasks in the hierarchy. This is true because a hand-made hierarchy will often be specially designed to allow state abstractions. The task might, however, be far more complex for automatically generated hierarchies. To formalize the opportunities of state abstractions when using the MAXQ method, Dietterich (2000) specifies conditions that permit “safe” state abstraction. Furthermore, Dietterich proves that MAXQ-Q will converge to the same unique recursively optimal policy with or without “safe” state abstraction for any given task hierarchy. In this section, we will give a less formal description of the opportunities for state abstraction when using the MAXQ method.

The purpose of applying state abstractions is to minimize the number of needed values to represent the projected value function for a task hierarchy. When less values are needed, then less values must be learned, which in most cases will speed up learning. To be able to evaluate the effect of applying state abstractions, let us first compute the number of values needed for the Taxi domain without state abstraction. We will ignore the representation of the internal completion function \tilde{C} for now.

- To represent $V(i, s)$ for each of the six leaf nodes in the MAXQ graph, 500 values are required for each leaf because there are 500 states.
- `MaxRoot` has two children, which requires a total $500 \cdot 2 = 1\,000$ values.
- Both `MaxGet` and `MaxPut` has two children, so each one also requires 1 000 values giving a total of 2 000.
- `MaxNavigate` has four children and the target parameter t , which can take on 4 values. For each child $500 \cdot 4 = 2\,000$ values are needed giving a total of 8 000.

The total number of values needed for the MAXQ representation is therefore 14 000. To place this number in perspective, consider that, using flat Q -learning, the number of needed values is 3 000.

The conditions for state abstraction specified by Dietterich all assume that a state s can be represented as a vector of values of existing state variables. At each Max node i , the vector

can be partitioned into two sets, relevant variables X_i and irrelevant variables Y_i . \mathcal{X}_i is a function that projects a state s into only the variables in X_i :

$$\text{if } s_i = \{x_0, \dots, x_n, y_0, \dots, y_m\} \text{ then } \mathcal{X}_i(s_i) = \{x_0, \dots, x_n\} \quad (4.15)$$

where n is the number of variables in X_i and m is the number of variables in Y_i . State abstractions are achieved by, for any state s , using $\mathcal{X}_i(s)$ instead of s to represent the projected value function. An abstraction is safe, when for all states s and subtasks i , we have that $V(i, \mathcal{X}(s)) = V(i, s)$.

It is furthermore required that the exploration policy used during learning is a so-called abstract hierarchical policy. This means that actions in a subtask i must be chosen using only information specified by \mathcal{X}_i , i.e. :

$$\text{if } \mathcal{X}_i(s_1) = \mathcal{X}_i(s_2) \text{ then } \pi_i(s_1) = \pi_i(s_2) \quad (4.16)$$

Failing to do so will result in unexplainable behavior given \mathcal{X}_i . Boltzmann exploration used throughout this report can easily be modified to be an abstract hierarchical exploration policy.

There are three kinds of conditions under which state abstractions can be introduced. The first condition involves eliminating irrelevant variables from subtasks in the MAXQ graph. This kind of abstraction is mostly useful in the lower part of the MAXQ graph, since subtasks near the leaf tend to have only few relevant variables. The second kind arises from so-called funnel actions that move the environment from a large number of current states to a small number of resulting states. Funnel actions normally appear in the top of the MAXQ graph. Finally, the third kind of state abstraction arises from the structure of the MAXQ graph itself. In effect, a large part of the state space may not be reachable for certain subtasks.

In the following, Y will always denote the set of irrelevant variables, while X will denote the set of relevant variables. After discussing the conditions that allow state abstractions, the total reduction of needed values to represent the projected value function for the task hierarchy is summarized.

4.4.1 Irrelevant Variable Elimination

Irrelevant variables can be eliminated both in leaves and composite subtasks in the MAXQ graph. The former condition is referred to as Leaf Irrelevance, while the latter is referred to as MaxNode Irrelevance (or Subtask Irrelevance).

MaxNode Irrelevance

A set of state variables Y is MaxNode irrelevant for subtask i if the following properties hold for any stationary abstract hierarchical policy π :

- No variable $y \in Y$ affects the value of any variable $x \in X$ in subtask i .
- No variable $y \in Y$ affects the value function $V^\pi(a, s)$ or pseudo-reward function $\tilde{R}_i(s)$ for any child action a and any state s .

In other words, the child actions chosen by subtask i must not depend on any variable in Y . Furthermore, the outcome of any executed child action must not depend on any variable in Y either. If these conditions hold, then the variables in Y are irrelevant for subtask i .

In the Taxi domain, two nodes in the MAXQ graph can benefit from state abstractions using the MaxNode irrelevance condition. First of all, during subtask `Get`, the destination of the passenger is irrelevant, because it does not affect which subtasks `Get` chooses to execute, nor does it affect the outcome of these subtasks. This means that the variable can be excluded from the completion functions stored in `QNavigateForGet` and `QPickup`.

Secondly, during the subtask `Put`, the variable denoting the passenger location is irrelevant, and can be eliminated from the completion functions in `QNavigateForPut` and `QPutdown`.

Leaf Irrelevance

While the MaxNode Irrelevance condition eliminates variables in the completion functions for composite subtasks, the Leaf Irrelevance condition eliminates variables in the value function for primitive actions. A set of variables Y is leaf irrelevant to primitive action a if, for any two states s_1 and s_2 that differ only in their values of variables in Y , we have that $V^\pi(a, s_1) = V^\pi(a, s_2)$. Remember that $V^\pi(a, s)$, for a primitive action a and state s , is defined simply as the expected immediate reward for performing a in s .

The primitive actions `North`, `South`, `West` and `East` have a constant immediate reward of -1 . This means that all state variables can be eliminated in their respective primitive Max nodes. Furthermore, the immediate rewards of `Pickup` and `Putdown` only depend on whether or not the actions are performed legally. For instance, `Putdown` is illegal if the taxi is not at the destination and holding the passenger. Thus, the value functions for each of these two actions require 2 values each.

In the elimination of variables in `Pickup` and `Putdown`, Dietterich introduces a new variable that denotes the legality of the actions—something which is in fact not directly possible using his proposed framework. Indeed, this form of abstraction requires logic and is similar to abstractions achieved using relational reinforcement learning. We will return to this subject in Chapter 5.

4.4.2 Funnel Actions

Funnel actions are composite subtasks that move the environment from a large number of current states to a small number of resulting states. Irrelevant variables in funnel actions must satisfy the condition of Result Distribution Irrelevance. Furthermore, if the termination of some subtask is guaranteed to make its parent task terminate too, then further abstraction can be applied. This condition is referred to as Termination.

Result Distribution Irrelevance

A set of variables Y is result distribution irrelevant for subtask i if, for all pairs of states s_1 and s_2 that only differ in their values for state variables in Y , we have that

$$P^\pi(s', N|s_1, i) = P^\pi(s', N|s_2, i) \tag{4.17}$$

for all s' and N . Thus, to be irrelevant for subtask i , a variable $y \in Y$ must not have any effect on the distribution of resulting states.

Consider the `Get` subtask. No matter what location the taxi has is in state s , it will be at the passenger's starting location in state s' when `Get` finishes executing. This makes the location of the taxi result distribution irrelevant, and the corresponding state variable can be eliminated in `QGet` and `QNavigateForGet`. Notice that the taxi location cannot be eliminated in `QPickup` because, when `Pickup` is executed illegally, the completion cost is dependent on the number of steps needed to complete `Get`.

Similarly, the taxi location is irrelevant for the `Put` subtask and can be eliminated from `QPut` and `QNavigateForPut`. For `QPut`, however, a stronger form of abstraction can be achieved using Termination and the structural constraints of the task hierarchy, as described in the following.

Termination

The Termination condition is very intuitive. If a subtask a is guaranteed to terminate in a goal state where its parent task i also terminates, then the completion cost of i after a has terminated must be uniformly zero for all states where a has not terminated.

This condition holds for `Put` in the Taxi domain. For all states where the passenger is in the taxi, `Put` will succeed and result in a goal terminal state for `Root`. This happens because the goals of `Put` and `Root` are identical.

4.4.3 Structural Constraints

The last form of state abstraction arises from the structural constraints introduced by the task hierarchy itself. A task a can only be executed in a state s if there exists a path from the root down to M_i consisting of un-terminated tasks. For any state s' where this is not the case, a cannot be reached. This means that it is unnecessary to represent the completion function $C(i, s', a)$ for any such state s' and parent task i .

The `Put` subtask also satisfies this condition, which is known as Shielding. `Put` is terminated in all states where the passenger is not in the taxi. Thus, `QRoot` does not need to represent completion values $C(\text{Root}, s, \text{Put})$ for these states. Together with the Termination condition above, this means that the entire completion function represented in `Put` is uniformly zero.

Furthermore, during the subtask `Get`, the passenger cannot be in the taxi in any non-terminal state. Therefore, any state s where this is the case can be disregarded.

4.4.4 Overview of State Abstractions in the Taxi Domain

The opportunities for state abstraction described in the previous section can be summarized as the following list:

- `MaxNorth`, `MaxSouth`, `MaxWest`, and `MaxEast`: each require 1 value (Leaf Irrelevance).
- `Pickup` and `Putdown`: each require 2 values (Leaf Irrelevance).

- $Q_{\text{North}}(t)$, $Q_{\text{South}}(t)$, $Q_{\text{West}}(t)$, and $Q_{\text{East}}(t)$: each requires 100 values (the passenger’s location and destination are MaxNode Irrelevant).
- $Q_{\text{NavigateForGet}}$: requires 4 values (the passenger’s destination is MaxNode Irrelevant, and the taxi starting location is Result Distribution Irrelevant).
- Q_{Pickup} : requires 100 values (the passenger’s destination is MaxNode Irrelevant).
- Q_{Get} : requires 16 values (the taxi’s location is Result Distribution Irrelevant, and the passenger’s location is limited to the four target locations because of Shielding).
- $Q_{\text{NavigateForPut}}$: requires 4 values (the passenger’s location is MaxNode Irrelevant, and the taxi’s location is Result Distribution Irrelevant).
- Q_{Putdown} : requires 100 values (the passengers location is MaxNode Irrelevant).
- Q_{Put} : requires 0 values (Termination and Shielding).

In total, this results in 632 distinct values when using state abstractions. If pseudo-rewards are needed, then it becomes $2 \cdot 632 = 1\,264$ values. Compared to the 3 000 values needed for flat Q -learning, this is a fairly low number. Furthermore, if the size of the grid is increased, then the number of values also increase. This increase is much larger for flat Q -learning compared to the MAXQ hierarchy with state abstractions. The reason is that, while the projected value function for the task hierarchy as a whole still depends on all state variables, each of the individual terms, that make up the decomposition of the value function, only depends on a subset of the state variables.

MaxNode Irrelevance and Leaf Irrelevance can also be applied when using the HSMQ algorithm. However, Result Distribution Irrelevance, Termination and Shielding cannot be applied. It is only because the Q function is decomposed into the completion function and the child value function that it is possible to take advantage of state abstractions that only affect the completion function.

4.5 Non-Hierarchical Execution of a Hierarchical Policy

As described in Section 4.3.3, the optimal policy for a task may not be representable given a task hierarchy. Dietterich (2000) presents a very simple technique, which in many cases can derive an optimal non-hierarchical policy from a hierarchical optimal policy. The idea is to start at the top of the task hierarchy, and then choose the locally optimal action in every subtask until a primitive action is reached—exactly the same as the functionality of EVALUATEMAXNODE. The primitive action is then executed, and control is again directed to the top of the hierarchy.

Table 4.5 shows the pseudo-code for the procedure EXECUTEPOLICYNONHIERARCHICAL, which follows this idea. At line 3, the algorithm calls EVALUATEMAXNODE to conduct a complete search of all paths through the MAXQ graph. Remember that EVALUATEMAXNODE also returns the primitive action a found at the end of the path through the graph (see Table 4.4). Afterwards, action a is executed and the current state is updated. This continues until the root task of the MAXQ graph terminates.

Consider changing the Taxi domain such that the passenger, with some probability, changes destination after the taxi has started navigating to the original destination. Using a strict

```

1: procedure EXECUTEPOLICYNONHIERARCHICAL(State  $s$ )
2:   while ( $T_0(s)$  is false)
3:     Let  $\langle V(0, s), a \rangle = \text{EVALUATEMAXNODE}(0, s)$ 
4:     Execute primitive action  $a$ 
5:     Let  $s$  be the resulting state
6:   end while
7: end

```

Table 4.5: Pseudo-code for executing the one-step greedy policy.

hierarchical execution, the already invoked navigation task must be completed, which means that the taxi will drive all the way to the original destination. At that point, it will discover that the passenger has changed destination, and will then begin to navigate towards the new destination. If non-hierarchical execution is used instead, control is shifted to the top of the hierarchy after each primitive action. This means that the taxi will discover the destination change immediately. The difference between hierarchical and non-hierarchical execution of a hierarchical policy becomes very clear in the experiments illustrated in Section 4.7.

4.6 Hierarchical Exploration Problem

Decomposing a primary task into a task hierarchy can be viewed as supplying the agent with knowledge because, besides creating opportunities for state abstraction, any reasonable task hierarchy will also guide the agent towards its goal. While this has advantages in form of a better initial performance, it also introduces a problem regarding sufficient exploration of certain parts of the state/action space. We will refer to this problem as the Hierarchical Exploration Problem. To our knowledge, this problem has not previously been described.

In the task hierarchy of the Taxi domain used throughout this chapter, the agent can explore the entire state/action space with the exception of states and actions that are unreachable due to structural constraints (e.g. `putdown` when the passenger is not in the taxi). Even though the hierarchy as a whole directs the agent in its actions, within a single subtask such as `Navigate(t)`, the agent will explore each possible state with an almost equal frequency. The only exception is that only one terminal state can be visited during each invocation of a subtask.

Now, consider what would happen if we changed `Navigate(t)` to a primitive action (although still temporally abstracted). Being a primitive action, `Navigate(t)` will always perform optimally (even during early learning) and will move the agent directly towards its target t . Figure 4.8 illustrates the exploration frequency for states in this modified Taxi domain, where the shade of gray denotes the relative level of exploration among the states. For simplicity, we assume that the only target locations are `R` and `B`.

This uneven exploration frequency occurs because the agent, once on the “right” path, never visits the outer states. Since `Navigate(t)` is assumed to be optimal, the agent will simply take the shortest route. In fact, these outer states will only be visited when they are the starting location of the taxi (as illustrated in the figure). This also suggests a solution for the problem, namely always letting the taxi start in these outer locations—thereby increasing their exploration frequency. Indeed, it is difficult to see any other possible solutions for the hierarchical exploration problem. In practice, a task hierarchy should be carefully created

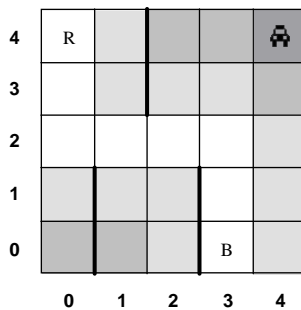


Figure 4.8: The hierarchical exploration problem with `Navigate(t)` as an optimal primitive action. White cells indicate often explored states, while darker shaded cells denote relatively less explored states. The darker the shade, the less the state is explored.

such that this problem is avoided. We will also discuss the hierarchical exploration problem for Blocks World in Chapter 5.

4.7 Experimental Evaluation of the MAXQ Method

So far, this chapter has described the theoretical advantages of using the MAXQ value function decomposition. In this section, we will perform two experiments to clarify the performance of MAXQ in practice. More specifically, the experiments are conducted to answer the following questions:

- How does the MAXQ method perform compared to flat Q -learning?
- How important are state abstractions?
- How does the encoding of knowledge in a task hierarchy influence performance?

Both experiments are based on a slightly modified Taxi domain called the Fickle Taxi domain². To make learning more challenging, the navigation actions `North`, `South`, `West` and `East` are made noisy. With probability 0.8, the taxi moves in the intended direction, but with probability 0.2, it instead moves to the right of the intended direction (e.g. if `East` is intended then with probability 0.2 the taxi will move south). Furthermore, after picking up the passenger and moving one square away from the passenger's source location, the passenger changes the destination with probability 0.3. The purpose of this change is to make the optimal policy non-hierarchical.

During training, the `Navigate(t)` subtask often exhibited looping behavior. In effect, testing a policy by choosing actions strictly greedily does not accurately show the improvement in performance as a function of primitive training steps. For instance, consider an almost optimal policy π . If this policy results in infinite looping behavior between states s_1 and s_2 (whenever one of these states are visited), then this behavior overshadows the performance of π in the rest of the state space. To avoid looping behavior, we applied Boltzmann exploration,

²The first experiment is similar to one conducted in Dietterich (2000).

and thereby a controllable level of randomness, to the evaluation of a policy. The exploration temperature was set to the initial value of 1 (total randomness) for all trials. It was thereafter decreased for each primitive training step such that it reached 0 at an estimated point of convergence. The estimated point of convergence was found by training the agent multiple times. The estimation was set to the latest convergence observed.

For all experiments, the learning factor α was set to 0.25. Furthermore, all initial V and C values were set to 0.

4.7.1 Performance of MAXQ Learning

In this experiment, we evaluated the performance of the following approaches when applied to the Fickle Taxi domain:

- Flat Q -learning,
- MAXQ without state abstractions (MAXQ),
- MAXQ with state abstractions (MAXQ-SA), and
- MAXQ with state abstractions and non-hierarchical execution (MAXQ-SA/NHE).

After each training episode, the current policy was tested by observing the error compared to an optimal policy. The mean of this error was computed over 10 training runs per approach. Figure 4.9 shows the mean error per trial (over the 10 runs) as a function of primitive training steps.

The results of the experiment shows that any form of MAXQ learning have better initial performance than flat Q -learning. This is due to the constraints introduced by the task hierarchy, which puts a restriction on the number of available primitive actions in any given state. Furthermore, it is interesting to see that MAXQ learning without state abstraction actually takes longer to converge than flat Q -learning. This is caused by the increased number of values needed to represent the decomposed value function without state abstractions.

MAXQ learning with state abstraction converges much faster than both flat Q -learning and MAXQ without state abstractions. It does not, however, reach true optimality. As mentioned, this is a result of allowing the passenger to change destination during an episode. Using the described task hierarchy in the Fickle Taxi domain, the taxi can simply not avoid taking a de-tour in 30 percent of the episodes.

Applying non-hierarchical execution to MAXQ with state abstractions solves this problem, and the same level of optimality as shown by flat Q -learning is reached. Non-hierarchical execution allows the taxi to react immediately to the change of destination. This advantage is also the reason that this approach reaches its potential optimal behavior a little faster than when not using non-hierarchical execution.

This experiment answers two of the questions stated in the beginning of the section. Clearly, MAXQ learning outperforms flat Q -learning, however, only when state abstractions are applied. If the optimal policy is non-hierarchical, then non-hierarchical execution must also be incorporated.

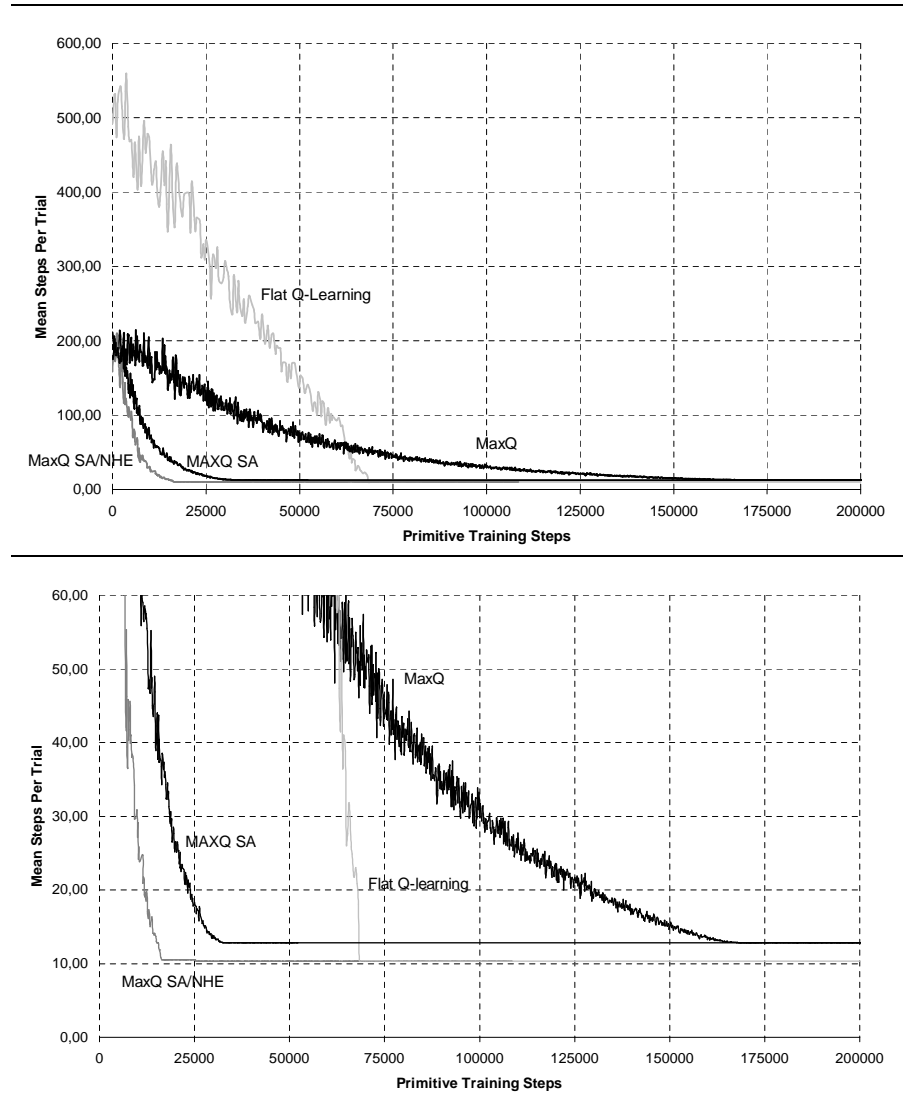


Figure 4.9: Performance of hierarchical MAXQ learning with state abstraction and non-hierarchical execution. The lower diagram shows a close-up view of the upper diagram.

4.7.2 Encoding of Knowledge

The task hierarchy used so far encodes a great deal of knowledge. For instance, the agent is automatically informed that during `Get`, it should only navigate to the passengers location—and not any of the other three locations. It is reasonable to supply the agent with this knowledge since it must be optimal. In general, as much knowledge as possible should be encoded into a task hierarchy. It is interesting however, to investigate the impact of this knowledge on the performance of MAXQ learning.

To perform the experiment, we changed the task hierarchy such that the taxi could navigate to any of the four target locations during both the `Get` and `Put` subtask. This increases the number of available actions in both these subtasks with 3 to a total of 5 (i.e. the hierarchy becomes less informed). The change also affects the number of needed values to represent the projected value function. The total number of values needed is increased from 632 to 656.

We tested the performance of the less informed hierarchy using both state abstractions and non-hierarchical execution. Figure 4.10 shows the results of the experiment. The policy over the previous task hierarchy is denoted “informed” and corresponds to MAXQ SA/NHE in Figure 4.9. The new hierarchy is denoted “less informed”.

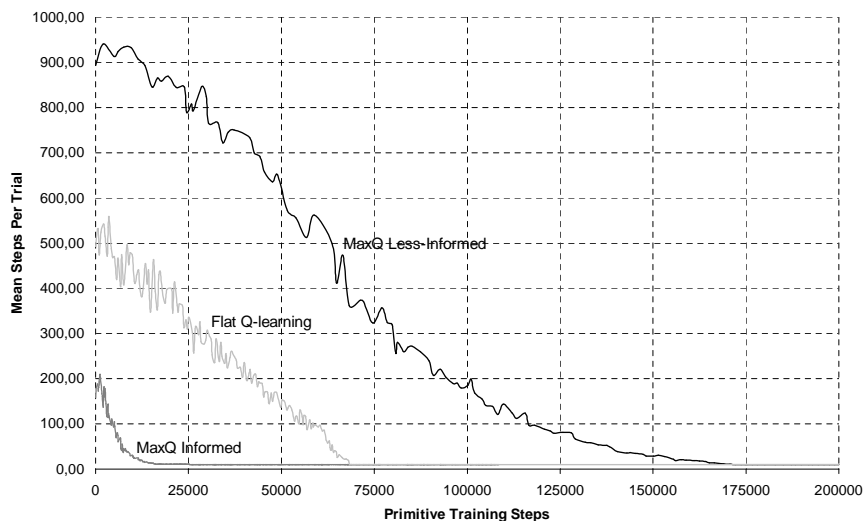


Figure 4.10: The performance of a less informed task hierarchy where the taxi can navigate to any of the four target locations during the `Get` and `Put` subtasks. The performance is compared to the task hierarchy in Figure 4.4 and flat Q -learning.

Surprisingly, the less informed hierarchy performs much worse than flat Q -learning. Even the initial performance is much worse. This observation should be seen in contrast to the fact that flat Q -learning requires the learning of almost twice as many values. Indeed, the problem is not the number of values but the pattern of exploration. The less informed hierarchy allows the agent to navigate to the wrong locations continuously during training. In effect, the number of primitive training steps increases fast without significantly leading the agent towards optimality.

This experiment answers the question of the effect of encoding knowledge into task hierarchies. Without sufficient knowledge, the agent risks wasting training steps performing subtasks that do not increase performance in any significant way. The experiment also shows that the number of needed values (to represent the projected value function) is not necessarily proportional with the learning rate.

4.8 Related Work

The MAXQ method is not the only hierarchical decomposition technique for reinforcement learning. Indeed, the hierarchical decomposition of domain, in order to make the search for solutions more effective, has been researched by several authors. Among the more recent decomposition techniques is the HAM method introduced by Parr (1998). The HAM method permits partial specification of hierarchical and temporally abstract actions.

A similar approach was introduced by Hauskrecht et al. (1998). This approach includes a hierarchical model for handling macro actions using periphery states to simplify the original environment. A goal of the model is the possibility of reusing macro actions in other similar environments.

4.9 Summary

In this chapter we described two approaches to hierarchical reinforcement learning. The first approach, Hierarchical Semi-Markov Q -learning (HSMQ), decomposes the primary task into a task hierarchy where each subtask completely encapsulates its own Q -function. This task decomposition allows some degree of state abstraction. The second approach, the MAXQ value function decomposition, goes further and also decomposes the projected value function of a task. This creates even further opportunities for state abstraction.

A task hierarchy guides the agent towards its goal, and can therefore make it difficult to explore the state/action space sufficiently. This problem can be made smaller by letting the agent start in less explored states, however there does not seem to exist any general solution.

Another inherent problem of hierarchical reinforcement learning is the inability to directly represent an optimal policy that is non-hierarchical. However, using the MAXQ method, non-hierarchical execution can easily be applied to a task hierarchy. This allows the agent to reach true optimal behavior—even when this is not hierarchical.

Two experiments were conducted to evaluate the performance of MAXQ learning. With state abstractions and non-hierarchical execution, MAXQ learning was shown to converge much faster than flat Q -learning in the Fickle Taxi domain. Furthermore, the importance of encoding knowledge into a task hierarchy was illustrated by creating a less informed hierarchy. The less informed hierarchy performed much worse than flat Q -learning, because it allowed the agent to waste training steps without increasing performance significantly.

Chapter 5

Combining Hierarchical and Relational Reinforcement Learning

In Chapter 3 and Chapter 4, we described two distinct approaches to reinforcement learning: relational reinforcement learning and hierarchical reinforcement learning using the MAXQ value function decomposition. The question now remains: can these two techniques be combined to achieve further advantages? In this chapter we will try to answer this question within the boundaries of the theory presented so far. That is, we will explore the possibilities of integrating logical state abstractions and logical decision trees into hierarchical reinforcement learning.

In Section 5.1 we define a MAXQ hierarchy for Blocks World and discuss deleted-condition interactions and the hierarchical exploration problem for the hierarchy. Section 5.2 introduces logical value and completion trees, while Section 5.3 introduces logical state abstraction into the MAXQ decomposition. In Section 5.4 we describe two approaches for deriving P -trees from a MAXQ hierarchy, and in Section 5.5 we discuss the results of a series of experiments conducted with the combination of relational and hierarchical reinforcement learning. Section 5.6 discusses the applicability of the combination in automatically constructed hierarchies, and finally we discuss related work in Section 5.7.

5.1 MAXQ Hierarchy for Blocks World

We will use Blocks World as the ongoing example domain throughout this chapter. To this end, we must first define a task hierarchy and a MAXQ graph for the domain. To keep things simple, we will once again only consider the task of stacking a specific block on top of another specific block. We will denote this root task as `Stack(A,B)`, where `A` and `B` are any two distinct blocks. The root task can be decomposed into the three subtasks: `MakeClear(A)`, `MakeClear(B)` and `Move(A,B)`. `MakeClear(A)` and `MakeClear(B)` are really denoting the same subtask `MakeClear(X)`—that of clearing a specific block `X`. This task can be further decomposed into `Move(Y,Z)` for any clear pair of blocks `Y` and `Z` in the domain. Figure 5.1 shows the described task hierarchy for Blocks World.

The task hierarchy describes the procedural decomposition of the domain. The value function decomposition is defined by the MAXQ graph illustrated in Figure 5.2. The primitive

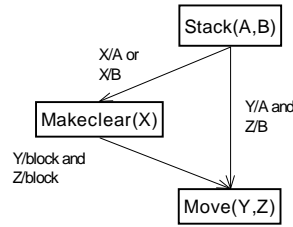


Figure 5.1: A task hierarchy for the Blocks World domain.

Max node $\text{MaxMove}(Y,Z)$ contains the values $V(\text{Move}(Y,Z), s)$ for all states s . The Q -nodes contain the completion cost of following the current policy after performing the particular action in the current state. Notice that the root Max node takes the parameters A and B . Thus, this hierarchy covers any binding of actual blocks such as, for instance, $\text{stack}(a,b)$ or $\text{stack}(c,a)$. In general, logical policies naturally allow parameterized subtasks in a more ad-hoc fashion than propositional approaches.

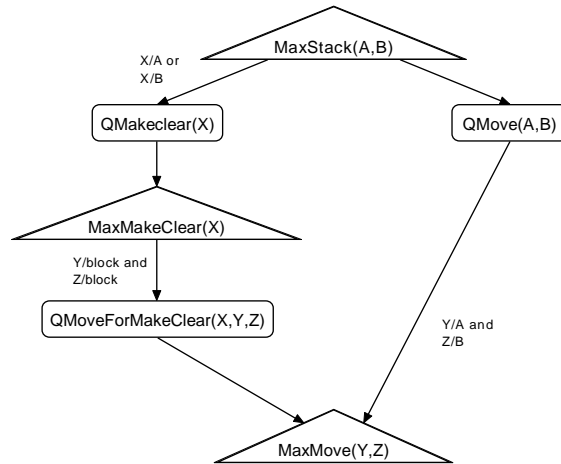


Figure 5.2: A MAXQ graph for the Blocks World domain. Again, Max nodes correspond to the subtasks in the domain, and Q nodes correspond to the actions available for each subtask.

5.1.1 Hierarchical Exploration Problem

The hierarchy in Figure 5.1 informs the agent that, to reach its goal, it must clear the two goal-state blocks A and B and then move A onto B . However, during the clearing of a block X , the agent is not guided in any way. Indeed, the agent can choose to move any clear block in the domain onto any other clear block. A valid question here is why we do not simply restrict the available actions during $\text{MakeClear}(X)$ to only include actions where a block is moved to the floor. Assuming unlimited floor space, there clearly exist an optimal policy using this restriction and we would furthermore effectively avoid deleted-condition interactions (see Section 2.3) where blocks are moved on top of an already cleared goal-state block. The reason we do not make this restriction is that it would introduce the hierarchical

exploration problem, as defined in Section 4.6, into the hierarchy. Consider applying the restriction; during each training episode, existing block stacks would “flatten” out until the goal was reached. Thus, the agent would explore “flat” states far more often than states with high stacks. The result of this would be poor performance in these states.

To avoid deleted-condition interactions without applying restrictions to the actions during `MakeClear(X)`, we will slightly change the reward function. As before, all actions will yield a penalty of -1 . Furthermore, actions that move a block on top of another block which is above either one of the goal-state blocks (A or B) will carry a second penalty of -1 yielding a total penalty of -2 . This change guides the agent towards solving `MakeClear(X)` without moving blocks onto A or B. Notice that we could achieve the same guidance using pseudo-rewards. For simplicity, however, we will ignore pseudo-rewards in this chapter.

5.2 Value and Completion Trees

As described in Chapter 4, the MAXQ decomposition of the value function opens up new opportunities for state abstractions. Combining MAXQ with relational reinforcement learning does not change this fact. First of all, the decomposition reduces the size of the individual functions, thereby making it easier for a relational learner to find suitable patterns. Secondly, if new state abstractions are made possible, then these will also be found by the relational learner. As a result, we will no longer consider Q -trees. Instead, the Q function is defined by a combination of V -trees (value trees) and C -trees (completion trees). These trees are, as Q -trees, logical regression trees. V -trees map a state s and a primitive subtask i to a numerical value. Similarly, C -trees map a state s , a parent task i , and a subtask a to a numerical value.

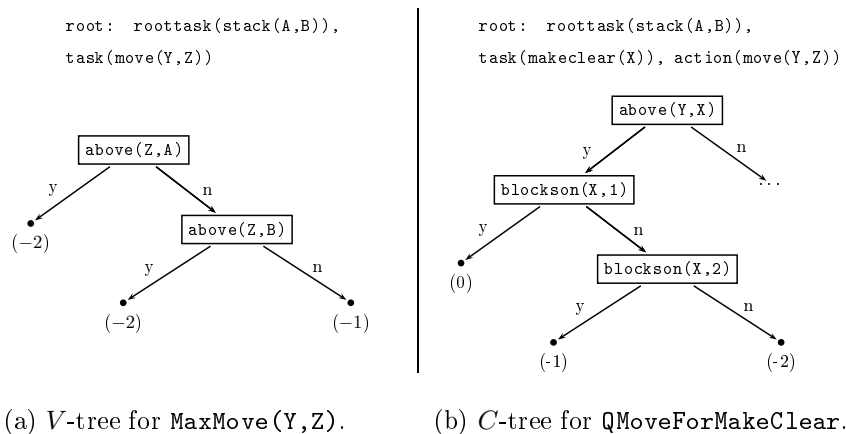


Figure 5.3: Example of an V -tree and a C -tree for subtasks in the Blocks World MAXQ graph.

Figure 5.3 shows an example V -tree and an example C -tree for Blocks World¹. The trees are queried in exactly the same way as Q -trees. Q values are computed by applying the decomposition equations defined in Chapter 4 to the results of querying the V -trees and C -trees. Updated trees are induced at the end of each episode over the base of examples created

¹The predicate `blockson(X,N)` holds if N is equal to the number of blocks on top of block X .

during training. Thus, the MAXQ-Q algorithm must be altered to generate examples and invoke the induction algorithm TILDE-RT. This change is very straightforward and similar to the change made to the regular Q -learning algorithm in Chapter 3. The complete relational MAXQ-Q learning algorithm can be found in Appendix C.

The MAXQ graph for Blocks World requires the learning of one V -tree for the Max node $\text{MaxMove}(Y,Z)$ and three C -trees for the Q -nodes $\text{QMakeClear}(X)$, $\text{QMove}(A,B)$ and $\text{QMoveForMakeClear}(X,Y,Z)$. Figure 5.4 shows a graphical overview of this.

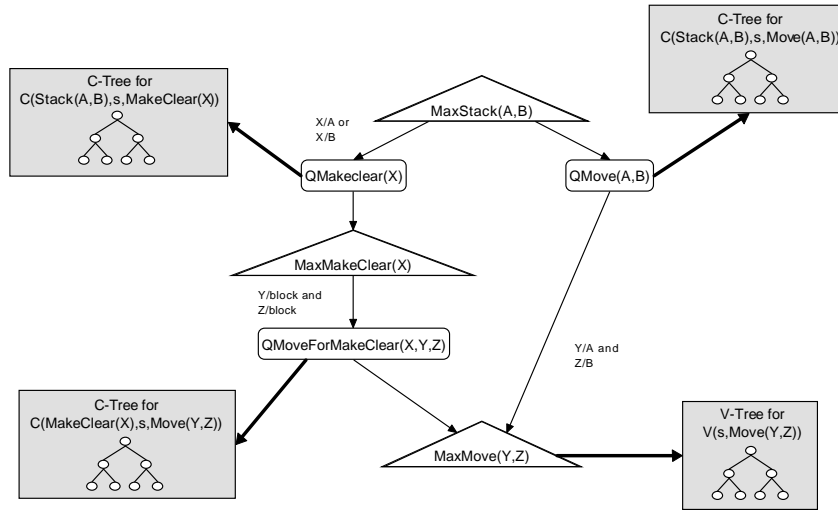


Figure 5.4: A graphical overview of the connection between the Blocks World MAXQ graph and logical decision trees representing the value and completion functions.

5.3 State Abstractions

Relational reinforcement learning makes use of first order logic both during the specification of a domain and for the induction of logical decision trees. In this chapter we will make a distinction between these two applications of logic. Doing so enables the possibility of manually applying logical state abstractions to hierarchical reinforcement learning without inducing logical decision trees. In fact, we have already seen a logical state abstraction manually applied to the Taxi domain in Chapter 4. This abstraction concluded on the legality of `Pickup` and `Putdown` actions and introduced a new state variable to contain the new information. As mentioned, the framework proposed by Dietterich (2000) does not really support this kind of abstraction. We will therefore introduce two new state abstraction conditions for the MAXQ decomposition. However, we first need to define precisely what we mean by a logical abstraction.

Definition 7. A logical abstraction function $\mathcal{L} : S \rightarrow S_{\mathcal{L}}$ is a mapping from the set of states S to the set of logical abstracted states $S_{\mathcal{L}}$ such that $size(S_{\mathcal{L}}) < size(S)$, where $size(S)$ is a function that returns the number of states in S . $\mathcal{L}(s)$ denotes the abstraction of state $s \in S$.

The definition of \mathcal{L} is left intensionally vague. It covers any function that reduces the number of distinct states in the state space. If an abstraction function is to be useful in a domain, it

must map each subset of similar states $\{s_0, \dots, s_n\}$ into a single abstract state $s_{\mathcal{L}}$. Sometimes further abstractions can be achieved by considering both a state and action together (and sometimes even a parent or ancestor task). In this case, \mathcal{L} can be changed to take this information as input (e.g. $\mathcal{L}(s, a)$ or $\mathcal{L}(i, s, a)$). The output still remains a single abstracted entity $s_{\mathcal{L}}$.

An abstraction function can be either safe or unsafe. A safe abstraction function only groups states (or state/action pairs) together that yield the exact same V and C values. The following are conditions under which an abstraction function \mathcal{L} is safe. We assume a hash-like lookup table to handle the mapping of multiple values to single abstracted entities.

Definition 8. *A logical abstraction function \mathcal{L} is safe for a primitive Max node i if, for the non-abstracted state $s \in S$ and action a , we have that*

$$V_i^\pi(a, s) = V_i^\pi(\mathcal{L}(a, s)) \quad (5.1)$$

Definition 9. *A logical abstraction function \mathcal{L} is safe for a Q-node j if, for the non-abstracted state s , action a and MaxNode i , we have that*

$$C_j^\pi(i, a, s) = C_j^\pi(\mathcal{L}(i, a, s)) \quad (5.2)$$

To illustrate these conditions by example, we will proceed to define a complete logical abstraction function for the Blocks World hierarchy defined in Section 5.1. We will define the function as a set of Prolog rules. The rules take the form

`logabstract(S, I, A, NextS)`

where S is the non-abstracted state, I is the parent task, A is the subtask, and $NextS$ is the logical abstracted state. As mentioned, the parent task I can be replaced by any ancestor task to A as this can sometimes allow for a higher level of abstraction. The first rules we define cover the logical abstractions possible in the Max node `MaxMove(Y,Z)`.

```
logabstract(S, stack(A,B), move(Y,Z), [illegal]) :- above(Z,A), !.
logabstract(S, stack(A,B), move(Y,Z), [illegal]) :- above(Z,B), !.
logabstract(S, stack(A,B), move(Y,Z), [legal]).
```

Remember that the agent receives an immediate penalty of -2 if a block is moved onto a stack with either of the goal-state blocks A and B , otherwise it receives -1 . The rules partition the state/action space into two abstracted states, `[legal]` and `[illegal]`, which exactly encompass these penalties. More specifically, for any non-abstracted state s and action a we have that

$$\text{if } V_i^\pi(a, s) = \begin{cases} -1 & \text{then } \mathcal{L}(s, \text{stack}(A,B), a) = [\text{legal}] \\ -2 & \text{then } \mathcal{L}(s, \text{stack}(A,B), a) = [\text{illegal}] \end{cases} \quad (5.3)$$

where i is the Max node `MaxMove(Y,Z)`. Thus, the abstraction satisfies Definition 8 and is safe. The result is that the number of needed values in `MaxMove(Y,Z)` is reduced to 2.

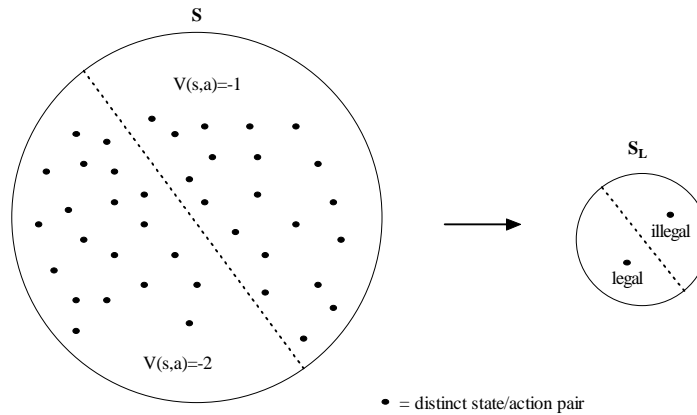


Figure 5.5: A graphical illustration of the logical abstraction applied to the Max node $\text{MaxMove}(Y, Z)$.

Furthermore, this reduction is completely independent of the number of blocks in the domain. A graphical way to observe the logical abstraction is illustrated in Figure 5.5.

A thing to notice is that abstracted entities are only used to determine the value of state/action pairs. They are *not* used in, for instance, the action precondition function or the transition function. These functions do therefore not need redefinitions even though we are sometimes changing the state variables dramatically. They still only need to depend on the original state variables present in the domain.

There are three Q -nodes in the Blocks World MAXQ graph to which state abstractions can be applied. We will start from the top and first define logical abstractions for $\text{QMakeClear}(X)$:

```

logabstract(S, stack(A,B), makeclear(A), NextS) :-
    blockson(S,B,N), NextS=[blocks_on_other(N)], !.

logabstract(S, stack(A,B), makeclear(B), NextS) :-
    blockson(S,A,N), NextS=[blocks_on_other(N)], !.

```

The important thing to notice here is that Q -nodes contain completion functions. As such, the information contained within an abstracted entity must only be exactly enough to conclude the completion cost of the parent task after performing the particular subtask—in this case $\text{MakeClear}(X)$. Indeed, this is the major advantage of using the MAXQ value function decomposition. After executing the subtask $\text{MakeClear}(A)$, the only relevant information to the completion of its parent task $\text{Stack}(A, B)$ is how many blocks that are above the other goal-state block B in state S . This number N is computed by the predicate $\text{blockson}(S, B, N)$. The resulting abstracted entity becomes the single fact $\text{blocks_on_other}(N)$. Similarly, for the subtask $\text{MakeClear}(B)$, the only important information for the completion function is how many blocks is above A . The reduction of needed values in this Q -node is *not* independent of the number of blocks in the domain. However, the number has a linear growth rate. For instance, using 3 blocks, only 3 values are needed, because a block can only have either one, two or three blocks above it.

The next Q -node is $QMove(A,B)$, which represents the final action of an episode. We do not need to define logical abstraction rules for this node, because all state variables are eliminated using the Termination and Shielding conditions from Chapter 4. Thus, the completion cost after performing this action is always zero.

Finally, we have the last Q -node $QMoveForMakeClear(X,Y,Z)$. The motivation behind the abstraction rules for this node is somewhat similar to the abstractions made for $MaxMove(Y,Z)$:

```

logabstract(S, makeclear(X), move(Y,Z), NextS) :-
    above(S,Y,A), blocksabove(S,X,N),
    N1 is N-1, NextS=[blocks_on(N1)], !

logabstract(S, makeclear(X), move(Y,Z), NextS) :-
    above(S,Z,A), blocksabove(S,X,N),
    N1 is N+1, NextS=[blocks_on(N1)], !

logabstract(S, makeclear(X), move(Y,Z), NextS) :-
    not(above(S,Y,X)), not(above(S,Z,X)),
    blocksabove(S,X,N), NextS=[blocks_on(N)], !

```

During the subtask of clearing a block X , the completion cost of moving a block depends on how many blocks are above X after the move. Thus two questions must be determined:

- Is the block Y being moved above X ?
- Is the destination block Z above X ?

The first rule covers the first question. In this case, the block Y being moved is above X . As a result, we can represent the abstracted state as `blocks_on(N1)` where $N1$ is the previous number of blocks above X minus one. The second rule covers the case where the destination block Z is above X . In this case, the number of blocks above X is increased by one. Finally, if the stack containing X is not involved in the move action, then the number of blocks above X remains the same.

The reduction in the number of needed values to represent the completion function for $QMoveForMakeClear(X,Y,Z)$ is again dependent on the number of blocks in the domain. Using three blocks, the number of needed values is reduced to two, since only one or two blocks can be above X without $MakeClear(X)$ being terminated. The number of needed values for this Q -node also has a linear growth rate.

5.3.1 Manual and Semi-Automatic State Abstraction

In the previous section we defined logical state abstractions for the $MAXQ$ value function decomposition. The question now is how these abstractions relate to the abstractions achieved by inducing logical decision trees. One might be tempted to think that first applying logical state abstractions manually and then proceeding to induce logical decision trees would somehow result in even further abstractions. Of course, this is not the case.

The logical abstractions that can be applied manually are indeed exactly the same abstractions that can be discovered during the induction of V -trees and C -trees. Whether or not they are actually discovered depends on the available tests defined by the background knowledge

(see Section 3.3). Finding state abstractions this way can be viewed as semi-automatic. It requires manual specification of the background knowledge, but the tedious task of searching for the actual possible abstractions becomes automatic. One advantage of applying logical decision trees to find state abstractions is therefore simply that it requires less manual specification. On the other hand, the induction of logical decision trees requires more computer power. We will compare the two approaches further in Section 5.5.

5.4 The Policy Function

The policy function P introduced in Chapter 3 encodes the optimality of state/action pairs. It is by definition dependent on the Q -function, but was shown to perform better in both its training domain and other similar domains (see Section 3.5). In this section, we will investigate how the policy function can be introduced into a relational MAXQ hierarchy. In general, the P -function cannot be part of the MAXQ value function decomposition because it does not express anything about observed rewards. Instead, the value function decomposition can be viewed simply as a representation of a hierarchical Q -function from which P can be derived. In particular, we will introduce two ways of deriving a P function from a MAXQ hierarchy. The first approach builds a local P -tree for each non-primitive Max node in the hierarchy. Each local tree denotes the optimality of executing subtasks of the Max node. The second approach builds a single global P -tree for the hierarchy. A global P -tree denotes the optimality of primitive actions and do not reference composite subtasks in the MAXQ hierarchy.

5.4.1 Local P -Trees

Remember that P takes a state s and an action a as input, and returns 1 if the pair is optimal and otherwise 0 if the pair is not optimal. P is defined using the Q -function, and since Q is defined by V and C in the MAXQ hierarchy, we can formulate the following definition of a hierarchical P function:

$$P(i, s, a) = \begin{cases} 1, & \text{if } a \in \arg \max_a (V(s, a) + C(i, s, a)) \\ 0, & \text{otherwise} \end{cases} \quad (5.4)$$

For instance, if `MakeClear(A)` is optimal in state s during the execution of subtask `Stack(A,B)`, then $P(\text{Stack}(A,B), s, \text{MakeClear}(A)) = 1$. As in Chapter 3, P is represented by a logical classification tree called a P -tree. In general, a P -tree can be induced for every non-primitive Max node in a MAXQ graph. Thus, using the Blocks World MAXQ graph from Figure 5.2, we can induce one P -tree for the root Max node `MaxStack(A,B)`, and one P -tree for `MaxMakeClear(X)`. This is illustrated in Figure 5.6.²

Using this approach, the relational MAXQ-Q algorithm must generate separate examples for each non-primitive Max node. As in Chapter 3, an example must be generated for all observed states combined with all possible actions. When using the non-incremental TILDE algorithm, these examples must be generated from scratch after each episode followed by the induction of a new P -tree. Assuming a local P -tree for each non-primitive Max node, the learned policy can be executed using the recursive algorithm displayed in Table 5.1.

²Notice that this is identical to inducing a P -tree for each non-primitive subtask in the task hierarchy illustrated in Figure 5.1.

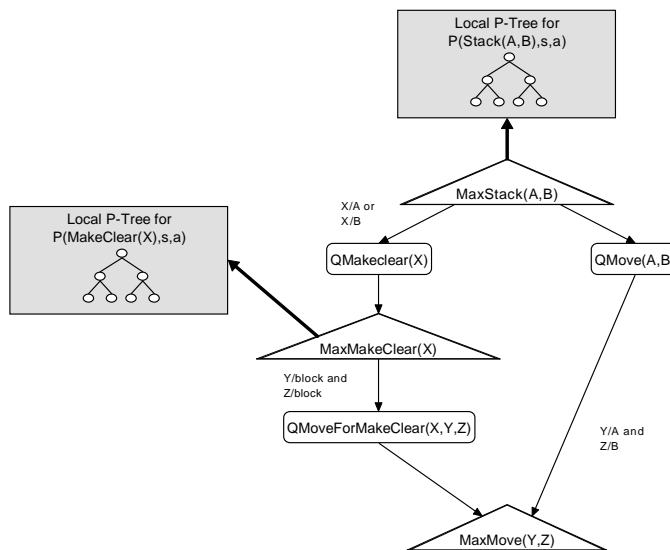


Figure 5.6: The Blocks World MAXQ graph with attached local P -trees for $\text{MaxStack}(A,B)$ and $\text{MaxMakeClear}(X)$.

```

1: function EXECUTEHIERARCHICALLOCALPOLICY(MAXNODE  $i$ , STATE  $s$ )
2:   if ( $i$  is a primitive Max node)
3:     Execute  $i$  and observe resulting state  $s'$ 
4:     Return  $s'$ .
5:   else
6:     while ( $T_i(s)$  is false)
7:       Find the available actions  $\{a_0, \dots, a_n\}$  in state  $s$ 
8:       Let  $m := 0$ 
9:       Let  $executed := false$ 
10:      while ( $m \leq n$  and  $executed = false$ )
11:        if ( $P(i, s, a_m) = 1$ )
12:           $s := \text{ExecuteLocalPolicy}(a_m, s)$ 
13:           $executed := true$ ;  $m := m + 1$ 
14:        end if
15:      end for
16:      if ( $executed = false$ )
17:         $s := \text{ExecuteLocalPolicy}(a_0, s)$ 
18:      end if
19:    end while
20:    Return  $s$ 
21:  end if
22: end function

```

Table 5.1: An algorithm for executing a policy represented by the hierarchical local policy function P .

The EXECUTEHIERARCHICALLOCALPOLICY algorithm takes a Max node i and a state s as input. If i is a primitive Max node, then the corresponding action is executed directly and the resulting state is returned. Otherwise, the algorithm first finds the set of available actions in s . It then cycles through these actions until an optimal action according to P is found. This action is executed by calling EXECUTEHIERARCHICALLOCALPOLICY recursively, which returns an updated state. If no actions are classified as optimal according to the P function (this can happen when P is not fully learned), then the first action a_0 is chosen.

For examples of local P -trees for the Blocks World hierarchy, see Section 5.5.

5.4.2 Global P -Tree

A local P -tree, for a task i in a MAXQ hierarchy, encodes the optimality of executing both composite and primitive subtasks of i . It is, however, also possible to use the MAXQ hierarchy to derive a global P -tree over only the primitive actions in the domain. Such a tree is similar to the P -trees described in Chapter 3 in all ways except how its examples are generated. Figure 5.7 illustrates the Blocks World MAXQ graph with an attached global P -tree.

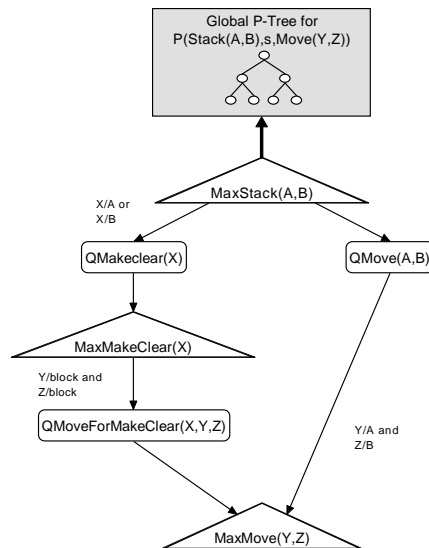


Figure 5.7: The Blocks World MAXQ graph with an attached global P -tree over all primitive actions in the domain.

Even though a MAXQ hierarchy, as a whole, can be viewed simply as any other representation of the Q -function for a domain, it also offers an opportunity for exploiting its internal structure. Recall the function EVALUATEMAXNODE defined in Chapter 4. This function performs a greedy search in the MAXQ hierarchy to find the path (from the root node to any possible leaf) that yields the highest expected cumulative reward. It returns both this expected reward as well as the primitive action at the leaf of the path. At the end of an episode, EVALUATEMAXNODE can be used to find the optimal primitive actions for all observed states to create optimal examples over these pairs. For all other actions, non-optimal examples are generated. Table 5.2 shows the pseudo-code for the algorithm GENERATEEX-

AMPLESFORGLOBALP that performs this functionality. After generating the examples, they are subsequently fed to TILDE to induce a global P -tree for the MAXQ hierarchy.

```

1: procedure GENERATEEXAMPLESFORGLOBALP (MAXNODE root)
2:   for (all observed states s)
3:     Find all available primitive actions  $A_p$  in state s
4:      $\langle v, a_{max} \rangle := \text{EVALUATEMAXNODE}(\text{root}, s)$ 
5:     for (all actions  $a \in A_p$ )
6:       if ( $a = a_{max}$ ) then
7:         Create optimal example  $x = \{s, a, 1\}$ 
8:       else
9:         Create non-optimal example  $x = \{s, a, 0\}$ 
10:      end if
11:    end for
12:  end for
13: end procedure

```

Table 5.2: An algorithm for generating examples for inducing a global P -tree over a MAXQ hierarchy.

This construction has a single advantage over regular P -trees learned from a flat representation of Q . An imposed hierarchy will often shield certain primitive actions from being executed in certain states. This means that EVALUATEMAXNODE will never return such a shielded action as optimal in these states. In effect, even during early learning, the agent will know that these shielded actions are never optimal.

The Blocks World hierarchy does not benefit from this advantage, since no primitive actions are shielded by the imposed hierarchy in any state. The hierarchy applied to the Taxi domain in Chapter 4, however, would benefit from it. In particular, the primitive actions `Pickup` and `Putdown` are shielded from execution in many states in the Taxi domain.

5.5 Experiments

To evaluate the various approaches for combining relational and hierarchical reinforcement learning, we performed a series of experiments. The purpose of these experiments was to answer the following questions:

- What is the performance gain for MAXQ hierarchical reinforcement learning when applying hand-coded logical state abstractions?
- How does MAXQ hierarchical reinforcement learning with hand-coded logical state abstractions perform compared to flat relational reinforcement learning?
- What is the performance of MAXQ hierarchical reinforcement learning using logical V -trees and C -trees (without hand-coded state abstractions), and to what extent is this performance improved by introducing local and global P -trees?
- How does the use of logical decision trees in MAXQ hierarchical reinforcement learning compare to the use of hand-coded logical state abstractions?

For each approach we measured the mean error per trial as a function of primitive training steps. As in the previous chapter, we once again computed the mean over 10 runs for each approach. For hierarchical approaches we defined a GLIE exploration policy using the Boltzmann exploration technique with a decreasing temperature. The temperature was initially set to 1, and was then decreased such that it reached 0 at the expected time convergence. The expected time of convergence was found by observing a series of test runs for each approach. All experiments were performed on a Blocks World domain with four blocks.

5.5.1 Hand-Coded Logical State Abstractions

In Chapter 4 we described five conditions that introduce state abstractions to a MAXQ hierarchy: Leaf Irrelevance, MaxNode Irrelevance, Result Distribution Irrelevance, Termination and Shielding. As described, these conditions can only be used to eliminate irrelevant state variables, and not to make logical conclusions over part of the state/action space. Using these conditions on a state in the Blocks World hierarchy, we can eliminate stacks of blocks not containing either of the goal-state blocks A or B. We furthermore do not need to represent any values for the Q -node $QMove(A,B)$ (see Section 5.3).

In this experiment, we compared the performance of a hierarchy with these non-logical state abstractions to the performance of a hierarchy with the logical state abstractions described in Section 5.3. Figure 5.8 shows the results of the experiment, where MAXQ-SA denotes the hierarchy with non-logical state abstractions and MAXQ-LSA denotes the hierarchy with logical state abstractions.

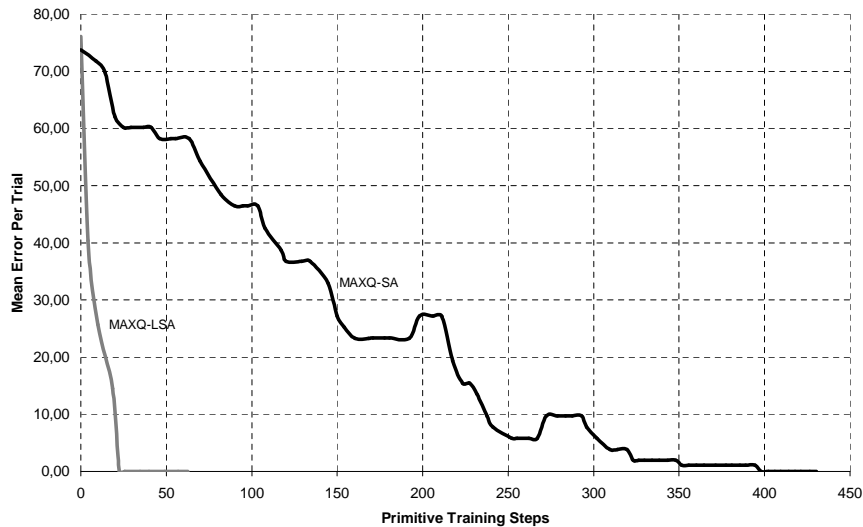


Figure 5.8: Comparison of the performance of MAXQ hierarchies when hand-coded logical and non-logical state abstractions are applied. MAXQ-LSA denotes a hierarchy with logical state abstractions, while MAX-SA denotes a hierarchy with only non-logical state abstractions

Since less values are needed to represent the hierarchy for MAXQ-LSA compared to MAX-SA, it reaches optimal behavior much faster. During the experiment, the agent learned 136 values for MAXQ-SA, while only 11 values were needed for MAXQ-LSA.

5.5.2 Flat Relational Reinforcement Learning

Having determined the performance of using logical and non-logical abstractions in Blocks World, it is interesting to compare these results to the performance of flat relational reinforcement learning. To achieve this, we let the agent learn both Q -trees and P -trees over the entire domain. We then compared the mean performance of these logical decision trees to the data obtained in the previous experiment. Figure 5.9 shows the results of this comparison. `Flat RRL-Q` denotes the mean performance of the learned Q -trees, while `Flat RRL-P` denotes the mean performance of the learned P -trees.

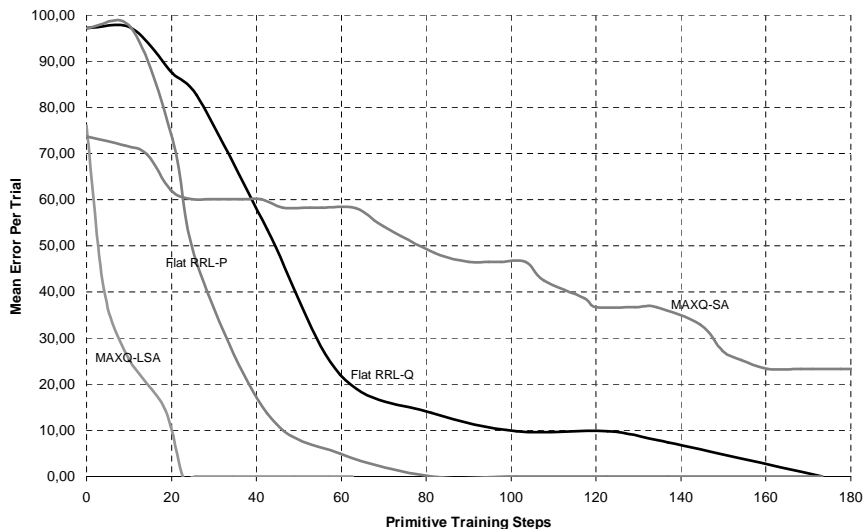


Figure 5.9: The performance of MAXQ hierarchies with hand-coded logical and non-logical state abstractions compared to flat reinforcement learning. `Flat RRL-Q` denotes the performance of learned Q -trees, while `Flat RRL-P` denotes the performance of learned P -trees.

The first thing to notice is that `MAXQ-SA` and `MAXQ-LSA` both have better initial performance. This is a consequence of the information implicitly encoded in the imposed hierarchy. The second thing to notice is that `MAXQ-LSA` reaches optimal behavior faster than both `Flat RRL-Q` and `Flat RRL-P`. This is not surprising since every possible logical state abstraction is hand-coded into the hierarchy of `MAXQ-LSA`. The relational approaches must instead search for these abstractions during learning. Thirdly, `MAXQ-SA` performs much worse than flat relational reinforcement learning. This was somewhat unexpected. It is a direct consequence of the powerful logical state abstractions possible in Blocks World that are unavailable to `MAXQ-SA`.

It is much more surprising that flat relational reinforcement learning performs so well compared to using hand-coded state abstractions. The most likely explanation for this behavior is that values of unobserved states can be predicted when using decision trees. This can result in reasonable or optimal behavior in unobserved parts of the state space. Using a tabular representation of the value functions, as done by `MAXQ-SA` and `MAXQ-LSA`, unobserved states are simply assigned a value of zero.

5.5.3 MAXQ Hierarchy with Logical Decision Trees

Before proceeding to compare the performance of hand-coded logical state abstractions to the use of logical decision trees in the MAXQ hierarchy, we first investigated the performance of three possible approaches to the latter. The first approach was to choose actions using the learned V -trees and C -trees. The second approach was to derive a global P -tree from the MAXQ hierarchy, while the third approach was to derive local P -trees for all non-primitive Max nodes. The results of the experiment are shown in Figure 5.10, where C/V , $P(\text{Global})$ and $P(\text{Local})$ refers to the three approaches respectively.

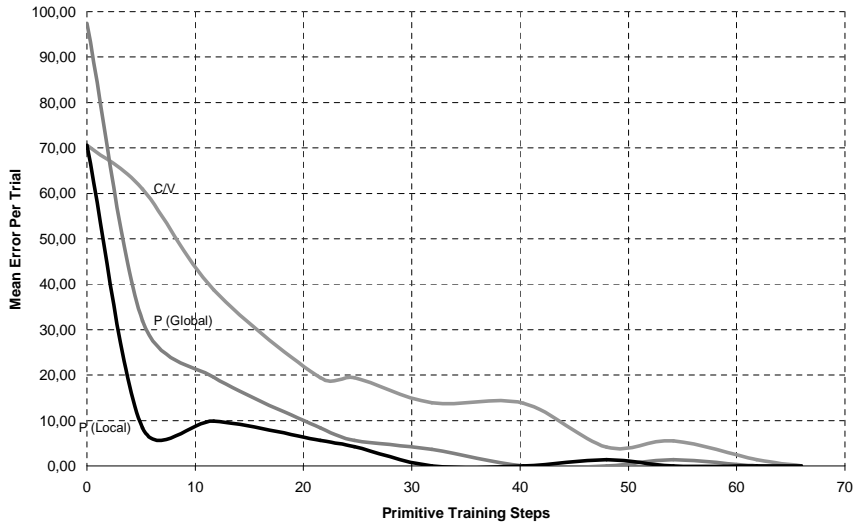


Figure 5.10: The performance of relational MAXQ hierarchies using logical decision trees. C/V denotes the performance of learned V -trees and C -trees. $P(\text{Local})$ denotes the performance of learned local P -trees, and $P(\text{Global})$ denotes the performance of learned global P -trees.

As expected, given the experiments performed in Chapter 3, both local and global P -trees perform better than using only V -trees and C -trees. Furthermore, $P(\text{Local})$ reaches both reasonable and optimal behavior slightly faster than $P(\text{Global})$. This happens because the patterns of optimality are simpler inside subtasks in the MAXQ hierarchy compared to the entire domain. For instance, an optimal action during the subtask $\text{MakeClear}(X)$ is simply an action that moves away a block from the stack containing X without moving it onto any goal-state block. This pattern is easier to learn than the optimal pattern for $\text{Stack}(A,B)$, which includes clearing both A and B and moving A onto B . It is reasonable to expect that the advantage of using local P -trees becomes even greater as the root task becomes more complex compared to its subtasks.

The optimal global P -tree learned in this experiment is similar to the optimal P -tree learned in Chapter 3 (illustrated in Figure 3.6), and has a total of 7 leaves.

The optimal local P -tree learned for the subtask $\text{MakeClear}(X)$ is shown in Figure 5.11. The tree classifies an action as optimal if a block is moved away from the stack containing X , otherwise false. It is clearly optimal both for a domain with four blocks, but also for a domain with any number of blocks. The optimal local P -tree for $\text{Stack}(A,B)$ turns out to be the constant 1. This encodes that every possible action in the subtask is optimal. Although this

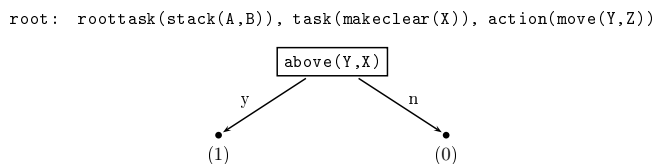


Figure 5.11: An optimal local P -tree for `MakeClear(X)` using any number of blocks.

sounds strange, it is a consequence of the applied action precondition function. `Stack(A,B)` can only invoke `MakeClear(A)` if `A` is not clear (and similarly with `B`). Furthermore, if both `A` and `B` are clear, then the only available action is `Move(A,B)`. As a result, the subtask can never execute a non-optimal action.

We can now compare the performance of hand-coded abstractions to the use of logical decision trees. To compare the approaches, we have plotted `MAXQ-SA`, `MAXQ-LSA` and `P(Local)` from the previous experiments into the diagram illustrated in Figure 5.12. The lower diagram shows a close-up view of the upper diagram.

An important thing to remember here is that `MAXQ-LSA` and `MAXQ-SA` are hand-coded with all possible logical and non-logical abstractions respectively. `P(Local)` must search for these abstractions during the induction of logical decision trees. Nevertheless, `P(Local)` performs much better than `MAX-SA`. This is again caused by the powerful logical state abstractions possible in `Blocks World` that is unavailable to `MAX-SA`. This statement is supported by the fact that `MAX-LSA` stabilizes with optimal behavior almost twice as fast as `P(Local)`. However, `P(Local)` actually outperforms `MAXQ-LSA` until after 22 primitive training steps. It does not stabilize with optimal behavior before after 55 primitive training steps. This is an effect of being derived directly from `C/V` which does not converge before after 65 primitive training steps.

The reason that `P(Local)` performs this well compared to `MAXQ-LSA` can again be credited to the possibility of value prediction for unobserved states. It can also be credited to the use of P -trees that, as explained in Chapter 3, outperforms techniques that encode the distance to the goal. Indeed, a pattern of optimality is often simpler than a pattern describing specific distances to the goal.

5.6 Automatically Constructed Hierarchies

The conducted experiments show that relational and hierarchical reinforcement learning can indeed be combined with advantages. By inducing local and global P -trees we obtained almost as good performance as manually hand-coding logical state abstractions into the `MAXQ` hierarchy. Both manual specification and the induction of logical decision trees can be time consuming, so the trade-off may seem simply to be the allocation of time to the two tasks.

Although logical state abstractions may be easy to manually locate in hand-coded hierarchies, such as the ones used in this report, this may not be the case in automatically constructed hierarchies. By inducing logical decision trees to automatically find the possible state abstractions, one needs only to construct a good global background knowledge. This background knowledge can then be used by all subtasks in their search for state abstractions. In essence,

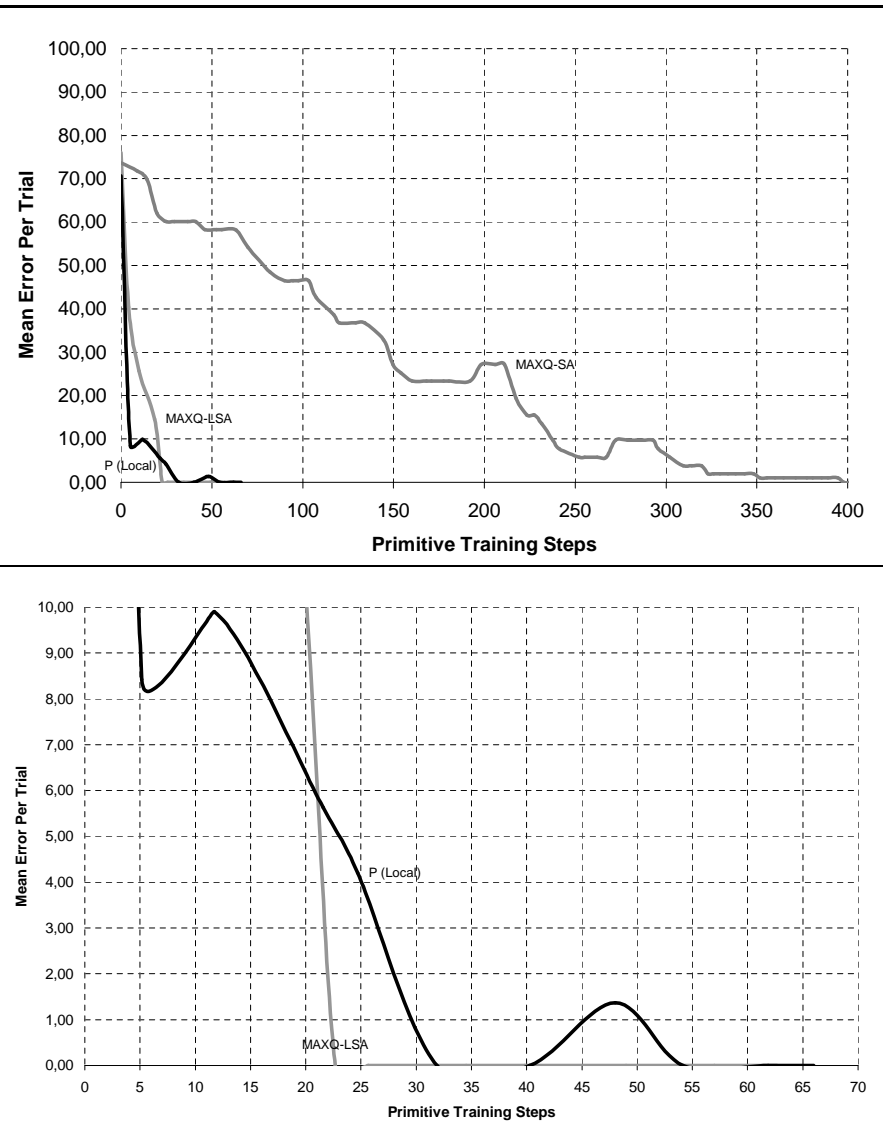


Figure 5.12: The performance of relational MAXQ hierarchies using logical decision trees compared to hand-coding state abstractions directly into a hierarchy. The lower diagram shows a close-up view of the upper diagram.

the combination of relational and hierarchical reinforcement learning may be very useful for agents that must automatically discover the hierarchical structures within its domain.

5.7 Related Work

Roncagliolo and Tadepalli (2004) also present a relational extension of hierarchical reinforcement learning using the MAXQ value function decomposition. Traditional Q -trees are piecewise constant in that they map state/action pairs to a constant value. This construction does not perform well when the complexity of an environment is slightly increased (as illustrated by the experiments performed in Chapter 3). Instead, Roncagliolo and Tadepalli propose a learning algorithm that learns a new form of hierarchical Q -tree as illustrated in Figure 5.3. The rules are on the form $q(\text{Task}, \text{Subtask}, \text{Value})$. The symbol $_$ denotes that any subtask can be inserted.

```

q(MakeClear(A),_,0)      :- clear(A).
q(MakeClear(A),MakeClear(B),V) :- on(B,A), q(MakeClear(B),_,V1), V is V1-1.
q(Stack(A,B),MakeClear(B),V) :- clear(A), q(MakeClear(B),_,V1), V is V1-1.

```

Table 5.3: Piecewise linear hierarchical Q -tree for the root task $\text{Stack}(A,B)$.

The illustrated hierarchical Q -tree is piecewise linear. For instance, the value V of executing $\text{MakeClear}(B)$ during the subtask $\text{Stack}(A,B)$ in a state where $\text{clear}(A)$ already holds, is computed as the value $V1$ of actually executing $\text{MakeClear}(B)$ minus one. Minus one denotes that only one action, namely $\text{Move}(A,B)$ remains when both A and B are clear. The work done by Roncagliolo and Tadepalli only includes preliminary experiments on the induction of such piecewise linear Q -trees.

5.8 Summary

In this chapter we have introduced various approaches for combining relational reinforcement learning with hierarchical reinforcement learning using the MAXQ value function decomposition. We have shown that logical state abstractions can be applied to a MAXQ hierarchy either manually or semi-automatically by inducing logical decision trees to approximate the value and completion functions.

Furthermore, we have introduced two approaches for deriving P -trees from a MAXQ hierarchy. One approach derives a local P -tree for each non-primitive Max node in the hierarchy, while the other derives a single global P -tree for the hierarchy.

A series of experiments showed that the performance of MAXQ hierarchy with logical decision trees comes very close to the performance of a hierarchy with manually hand-coded state abstractions. Even flat relational reinforcement learning performed fairly well compared to a hierarchy with hand-coded state abstractions. Notice that these conclusions are based solely on the experiments performed on Blocks World with four blocks. Future work should evaluate these results on other and more complex domains.

Chapter 6

Conclusion

Reinforcement learning denotes the process of teaching an agent optimal behavior in its environment by reinforcing its actions with rewards and penalties. Unfortunately, traditional reinforcement learning is inadequate for anything but very small problem domains. In this work we have re-explored two existing extensions of reinforcement learning, namely relational reinforcement learning and hierarchical reinforcement learning. We have furthermore investigated the possibilities of combining these two methods.

6.1 Relational Reinforcement Learning

In Chapter 3, we explored relational reinforcement learning and evaluated the method using the Blocks World domain. The conclusions of the work can be summarized as follows:

- Relational reinforcement learning exploits the structural constraints in relational domains. In such domains, relational reinforcement learning with proper background knowledge will reach both reasonable and optimal behavior faster than traditional reinforcement learning.
- The induction of P -trees to encode the optimality of actions enhances the performance of a Q -tree representation of a policy. Furthermore, a P -tree will often perform well in other similar domains (i.e. a P -tree has better generalization properties).

6.2 Hierarchical Reinforcement Learning

Similarly, in Chapter 4 we evaluated hierarchical reinforcement learning using the MAXQ value function decomposition. For this method we can make the following conclusions:

- A procedural hierarchical decomposition of a problem domain introduces the opportunity for state abstractions because some variables become irrelevant in individual subtasks.
- The MAXQ value function decomposition introduces further opportunities for state abstractions because of the separation of value and completion functions.

- Constraints created by an imposed hierarchy will sometimes prohibit an agent from reaching optimal behavior. This problem is solved by using pseudo-rewards and non-hierarchical execution of the learned hierarchical policy.
- A task hierarchy will often guide the agent such that it avoids executing poor actions in certain states. To some extent, this guidance can help an agent learn in domains with sparsely distributed rewards. However, less informed hierarchies can make the agent perform worse compared to flat reinforcement learning.
- Careful consideration must be employed in the construction of the task hierarchy, such that the hierarchical exploration problem can be avoided.

6.3 Combining Relational and Hierarchical Reinforcement Learning

In Chapter 5 we introduced the combination of relational and hierarchical reinforcement learning. The application of inductive logic in the MAXQ value function decomposition was split into manual specification of state abstractions, and semi-automatic detection of state abstractions through the induction of logical decision trees. The following conclusions can be made:

- In appropriate hierarchies, the introduction of logical state abstractions will greatly increase the rate in which optimal behavior is reached.
- If subtasks in a hierarchy are of limited size, the induction of logical decision trees will often help discover state abstractions fairly quickly. This is, of course, dependent on the quality of the existing background knowledge.
- The individual subtasks in a hierarchy will most often be less complex compared to the root task. As a result, patterns of optimality are more easily found by the induction of local P -trees.
- The semi-automatic detection of state abstractions, introduced by the induction of logical decision trees, can most likely be beneficially applied to automatically generated hierarchies.

6.4 Summary of Contributions

This work both re-explores existing material and introduces new knowledge. The new contributions of knowledge can be summarized as

- The construction and experimental evaluation of a relational MAXQ-Q algorithm.
- Introduction of two methods for deriving local and global P -trees from a MAXQ hierarchy.

The constructed relational MAXQ-Q algorithm is general and can be applied to any problem domain. The two methods for deriving P -trees from a MAXQ hierarchy supplement each other nicely. Local P -trees will most often converge faster than a global P -tree. However, a global P -tree abstracts away from any reference to the imposed hierarchy and is more directly executed.

Other minor contributions include the following:

- Formulation of the hierarchical exploration problem.
- Experimental evaluation of the advantages of logical state abstractions in relational domains.
- The idea that logical decision trees can be used to semi-automatically find state abstractions in automatically generated task hierarchies.

The hierarchical exploration problem and its effects on MAXQ hierarchies were described for both the Taxi and Blocks World domain. We furthermore evaluated the advantages of using logical state abstractions in a relational domain such as Blocks World. For such domains, logical state abstractions are vastly superior to any non-logical state abstractions. Finally, we introduced the idea that logical decision trees can be used to semi-automatically find state abstractions in automatically generated task hierarchies.

6.5 Future Work

The evaluation of the combination of relational and hierarchical reinforcement learning in Chapter 5 is based only on experiments conducted on the relational Blocks World domain using four blocks. It would be interesting to investigate the application of the method on a non-relational domain such as the Taxi domain, as well as other realistic (or semi-realistic) problems.

It seems likely that a task hierarchy can be used to separate relational and non-relational subtasks such that different learning algorithms can be applied to the hierarchy. For the Taxi domain, a relational learning algorithm could be applied to all subtasks except the non-relational subtask `Navigate`. This subtask could then be solved using a propositional representation language.

Learning techniques for agents are most often used by the computer game industry to make games more real and challenging. The problem with reinforcement learning, in this context, is that it is primarily suited for stationary environments. For example, the attempt to learn a navigation policy for anything but a small stationary domain (such as the Taxi domain) would require both considerable space and time. Indeed, reinforcement learning seems more suited for making decisions on a higher level of abstraction. For an action computer game, reinforcement learning could be used to decide when to attack, hide or apply other strategic actions. These decisions could be trained by observing the behavior of the opponent player, thereby customizing the agent's behavior to challenge individual human players. The actual low-level execution of a strategic decision could then be distributed to other more fitting techniques.

Finally, the hierarchies used in this work are a result of a procedural decomposition of the root task. The hierarchies encode a “subtask of” relationship between tasks. A relational

setting might create the opportunity for richer hierarchies using different relationships such as e.g. “more specific than”. For instance, a task in the top of such a hierarchy could supply a crude solution to a problem. This crude solution could then be refined by tasks lower in the hierarchy.

Appendix A

Summary

Reinforcement Learning is the task of teaching an agent optimal behavior in its environment by reinforcing good actions with rewards and poor actions with penalties. At any point in time, the environment is in a specific state, and the agent is given a selection of actions to choose from. The chosen action moves the environment from its current state to a new state dictated by a transition probability distribution. Depending on the chosen action, the agent is rewarded or penalized.

As problems grow larger, representation becomes an increasingly important issue. Many real-world problems and their solutions (i.e. a control policies) are often impossible to represent directly in a conventional table-based manner. This has given rise to various approaches to ease the problem of a large state space. In general, the state space is either reduced by the use of state abstractions, or the agent is guided on the right path (thus avoiding a possibly large part of the state space).

In this report we explore two of these approaches, namely relational reinforcement learning (Džeroski et al., 2001) and hierarchical reinforcement learning using the MAXQ value function decomposition (Dietterich, 2000). Relational reinforcement learning exploits the structural constraints in relational domains by combining reinforcement learning and inductive logic programming. In such domains, relational reinforcement learning with proper background knowledge will reach both reasonable and optimal behavior faster than traditional reinforcement learning. Inducing logical decision trees over the optimality of actions furthermore enables good generalization properties, such that learned policies can be applied to similar domains.

Hierarchical reinforcement learning imposes a hierarchical decomposition of a domain. This decomposition has the advantages of creating opportunities for state abstractions and guiding the agent towards its goal. The MAXQ value function decomposition creates the opportunity for even further state abstractions. As a result, hierarchical reinforcement learning with state abstractions outperforms traditional reinforcement learning given a reasonable informed hierarchy. Careful consideration must, however, be put into the the construction of the hierarchical decomposition to avoid exploration problems.

Besides the re-exploration of these two existing methods, the major contribution of this work is to explore the advantages of combining relational reinforcement learning and hierarchical reinforcement learning. That is, we investigate the possibilities of integrating inductive logic programming into hierarchical reinforcement learning. Logical state abstractions can be

introduced into a hierarchy either manually or semi-automatic by inducing logical decision trees. The experiments performed in this work shows that the latter requires less specification and performs almost as good as the former. The final result is a general learning algorithm that outperforms both relational reinforcement learning and hierarchical reinforcement learning. The algorithm is only evaluated in the Blocks World domain, and should be further tested in other more realistic domains.

Appendix B

ACE Blocks World Specification

This appendix shows the ACE configuration files used for the various Blocks World experiments throughout the report. We only show the configuration files used for flat relational reinforcement learning, since the others are almost identical. The only difference is background knowledge for testing on other actions than the primitive `Move(X,Y)` action.

B.1 Background Knowledge

```
eq(E,X,X).
above(E,X,Y) :- on(E,X,Y).
above(E,X,Y) :- on(E,X,Z), above(E,Z,Y).
action_move(E,X,Y) :- action(E,move(X,Y)).
goal_on(E,A,B) :- goal(E,stack(A,B)).
```

B.2 TILDE-RT Settings for Inducing Q-trees

```
tilde_version('3.0').
load_package(tilde).
load(key).
predict(qvalue(+ex,-value)).
heuristic(eucl).
euclid(qvalue(E,X), X).
tilde_mode(regression).
confidence_level(1).
minimal_cases(1).
output_options([prolog]).
ftest(1.0).
talking(0).
use_packs(0).
execute(tilde).
execute(quit).
```

```
root((goal_on(E,A,B),action_move(E,C,D))).
```

```
typed_language(yes).
type(clear(ex,block)).
type(on(ex,block,block)).
type(eq(ex,block,block)).
type(above(ex,block,block)).
type(action_move(ex,block,block)).
type(goal_on(ex,block,block)).
```

```
rmode(10: clear(+E,+X)).
rmode(10: on(+E,+X,+Y)).
rmode(10: on(+E,+X, floor)).
rmode(10: eq(+E,+X,+Y)).
rmode(10: eq(+E,+X,floor)).
rmode(10: above(+E,+X,+Y)).
rmode(10: action_move(+E,+X,+Y)).
rmode(10: action_move(+E,+X,floor)).
```

B.3 TILDE Settings for Inducing *P*-trees

```
tilde_version('3.0').
load_package(tilde).
load(key).
predict(pvalue(+ex,-value)).
confidence_level(1).
minimal_cases(1).
output_options([prolog]).
ftest(1.0).
talking(0).
use_packs(0).

root((goal_on(E,A,B),action_move(E,C,D))).
```

```
typed_language(yes).
type(on(ex,block,block)).
type(eq(ex,block,block)).
type(above(ex,block,block)).
type(clear(ex,block)).
type(action_move(ex,block,block)).
type(action_makeclear(ex,block)).
type(goal_on(ex,block,block)).
```

```
rmode(10: clear(+E,+X)).
rmode(10: on(+E,+X,+Y)).
rmode(10: on(+E,+X, floor)).
rmode(10: eq(+E,+X,+Y)).
```

```
rmode(10: eq(+E,+X,floor)).  
rmode(10: above(+E,+X,+Y)).  
rmode(10: action_move(+E,+X,+Y)).  
rmode(10: action_move(+E,+X,floor)).
```


Appendix C

Relational MAXQ-Q Learning Algorithm

This appendix shows pseudo-code for a relational MAXQ-Q learning algorithm. The algorithm is further extended to produce local P -trees. Pseudo-code for producing global P -trees is illustrated in Table 5.2 in Chapter 5.

C.1 Relational MAXQ-Q

Table C.2 shows the pseudo-code for the relational MAXQ-Q algorithm REL-MAXQ-Q. The algorithm approximates V , C and \tilde{C} using logical decision trees

C.2 Learning Local P -trees

To produce local P -trees, the relational MAXQ-Q algorithm must be extended with the pseudo-code illustrated in Table C.1. The pseudo-code should be executed at the end of each episode.

```
1: For (all non-primitive Max nodes  $i$ )
2:   For (all observed states  $s$  in Max node  $i$ )
3:     For (all possible subtasks  $a_k$  possible in state  $s$ )
4:       If (state/action pair  $(s, a_k)$  is optimal
5:         according to current approximation of  $Q$ ) Then
6:           Generate example  $x = (s, a_k, c)$  where  $c = 1$ 
7:         Else
8:           Generate example  $x = (s, a_k, c)$  where  $c = 0$ 
9:         End If
10:      End For
11:    End For
12:    Update  $P_e^i$  using TILDE to produce  $P_{e+1}^i$  using these examples  $(s, a_k, c)$ 
13:  End For
```

Table C.1: Learning local P -trees using a relational MAXQ hierarchy.

```

1: function REL-MAXQ-Q(MaxNode  $i$ , State  $s$ )
2:   Let  $seq = ()$  be the sequence of states visited while executing  $i$ 
3:   if ( $i$  is a primitive MaxNode)
4:     Execute  $i$ , receive  $r_t = R(s'|s, a)$ , and observe result state  $s'$ 
5:     Generate example  $x = (s, i, v_{t+1})$  in  $Examples_V$  where
6:      $V_{t+1} := (1 - \alpha_t(i)) \cdot V_t(i, s) + \alpha_t(i) \cdot r_t$ 
7:     Push  $s$  onto the beginning of  $seq$ 
8:   else
9:     Let  $count = 0$ 
10:    while ( $T_i(s)$  is false)
11:      Choose an action  $a$  according to the current exploration policy  $\pi_\omega(i, s)$ 
12:      Let  $childSeq = \text{REL-MAXQ-Q}(a, s)$  where  $childSeq$  is the
13:        sequence of states visited executing action  $a$  (in reverse order)
14:      Observe result state  $s'$ 
15:      Let  $a^* = \arg \max_{a'} [\tilde{C}_t(i, s', a') + V_t(a', s')]$ 
16:      Let  $N = 1$ 
17:      for (each  $s$  in  $childSeq$ ) do
18:        Generate example  $x = (i, s, a, c)$  in  $Examples_C$  where
19:         $c = (1 - \alpha_t(i)) \cdot C_t(i, s', a') + \alpha_t(i) \cdot \gamma^N \text{externalValue}(s')$ 
20:        Generate example  $\tilde{x} = (i, s, a, \tilde{c})$  in  $Examples_{\tilde{C}}$  where
21:         $\tilde{c} = (1 - \alpha_t(i)) \cdot \tilde{C}_t(i, s', a') + \alpha_t(i) \cdot \gamma^N \text{internalValue}(s')$ 
22:        with
23:         $\text{externalValue}(s') = [C_t(i, s', a^*) + V_t(a^*, s')]$ , and
24:         $\text{internalValue}(s') = [\tilde{R}_i(s') + \tilde{C}_t(i, s', a^*) + V_t(a^*, s')]$ 
25:         $N := N + 1$ 
26:      end for
27:      Append  $childSeq$  onto the front of  $seq$ 
28:       $s := s'$ 
29:    end while
30:  end if
31:  Return  $seq$ 
32: end

33: //main program
34: Initialize  $V(i, s)$ ,  $C(i, s, a)$  and  $\tilde{C}(i, s, a)$  to trees producing the value 0 for all inputs
35: Initialize  $Examples_C$ ,  $Examples_{\tilde{C}}$  and  $Examples_V$  to the empty set
36: MAXQ-Q(root node 0, starting state  $s_0$ )
37: Update  $V$  using TILDE-RT to produce  $V_{e+1}$  using  $Examples_V$ 
38: Update  $C$  using TILDE-RT to produce  $C_{e+1}$  using  $Examples_C$ 
39: Update  $\tilde{C}$  using TILDE-RT to produce  $\tilde{C}_{e+1}$  using  $Examples_{\tilde{C}}$ 

```

Table C.2: Relational MAXQ-Q algorithm.

Bibliography

- Andersen, C. C. S., Boesen, T. and Pedersen, D. K. (2005). Applying relational reinforcement learning to multi-agent environments. URL = <http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1105611153>.
- Blockeel, H. and Raedt, L. D. (1998). Top-Down Induction of First-Order Logical Decision Trees, *Artificial Intelligence* **101**(1-2): 285–297.
*citeseer.ist.psu.edu/blockeel98topdown.html
- Blockeel, H., Raedt, L. D., Dehaspe, L., Ramon, J., Struyf, J. and Laer, W. V. (2004). *The ACE Data Mining System User's Manual*.
- Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984). *Classification and Regression Trees.*, Wadsworth.
- Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons, *Proc. of the 12th IJCAI*, Sidney, Australia, pp. 726–731.
- Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, *J. Artif. Intell. Res. (JAIR)* **13**: 227–303.
- Driessens, K. and Džeroski, S. (2004). Integrating Guidance into Relational Reinforcement Learning, *Machine Learning* **57**: 271–304.
- Driessens, K. and Ramon, J. (2003). Relational instance based regression for relational reinforcement learning, *Proceedings of the Twentieth International Conference on Machine Learning*, AAAI Press, pp. 123–130. URL = http://www.cs.kuleuven.ac.be/cgi-bin/dtai/publ_info.pl?id=40845.
- Driessens, K., Ramon, J. and Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner, *Lecture Notes in Computer Science* **2167**.
*citeseer.ist.psu.edu/driessens01speeding.html
- Džeroski, S., Raedt, L. D. and Driessens, K. (2001). Relational Reinforcement Learning, *Machine Learning* **43**(1/2): 7–52.
- Fikes, R. E. and Nilsson, N. J. (1990). Strips: A new approach to the application of theorem proving to problem solving, in J. Allen, J. Hendler and A. Tate (eds), *Readings in Planning*, Kaufmann, San Mateo, CA, pp. 88–97.

- Gupta, N. and Nau, D. S. (1991). On the complexity of blocks-world planning., *Technical Report TR 1991-74*, The Institute for Systems Research.
- Gärtner, T., Driessens, K. and Ramon, J. (n.d.). Graph kernels and gaussian processes for relational reinforcement learning.
*citeseer.ist.psu.edu/644898.html
- Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T. and Boutilier, C. (1998). Hierarchical solution of markov decision processes using macro-actions., *UAI*, pp. 220–229.
- Jaakkola, T., Jordan, M. I. and Singh, S. P. (1994). Convergence of stochastic iterative dynamic programming algorithms, in J. D. Cowan, G. Tesauro and J. Alspector (eds), *Advances in Neural Information Processing Systems*, Vol. 6, Morgan Kaufmann Publishers, Inc., pp. 703–710.
*citeseer.ist.psu.edu/article/jaakkola93convergence.html
- Parr, R. E. (1998). Hierarchical control and learning for markov decision processes.
*citeseer.ist.psu.edu/parr98hierarchical.html
- Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Roncagliolo, S. and Tadepalli, P. (2004). Function Approximation in Hierarchical Relational Reinforcement Learning, *Proceedings of the ICML'04 workshop on Relational Reinforcement Learning*.
- Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems, *Technical Report CUED/F-INFENG/TR 166*, Engineering Department, Cambridge University.
- Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*, 2nd edition edn, Prentice-Hall, Englewood Cliffs, NJ.
- Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA. A Bradford Book.
*<http://www-anw.cs.umass.edu/rich/book/the-book.html>
- Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*, PhD thesis, Cambridge University, Cambridge, England.

References containing URLs are valid as of June 16, 2005.