MSc Thesis

Learning Recognizers from Experience

Harry Seip

McGill

TUDelft

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Learning Recognizers from Experience

by

Henricus Paulus Leendert Seip

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

presented at

Delft University of Technology,
Faculty of Electrical Engineering,
Mathematics, and Computer Science,
Man-Machine Interaction Group.

February 2007

Man-Machine Interaction Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Reasoning and Learning Lab
School of Computer Science
McGill University
3480 University Street
Montreal
H3A 2A7
Quebec, Canada

Keywords
Artificial intelligence, Reinforcement Learning, Markov decision processes, Recognizers, Off-policy learning, Decison making under uncertainty.

Learning recognizers from experience

Copyright © 2007 by Harry Seip (1015729)
Man-Machine Interaction Group
Faculty of EEMCS
Delft University of Technology

Members of the Supervising Committee

Abstract

Learning through interaction forms a foundational aspect of learning. To construct powerful autonomous agents we need to ground their knowledge in (verifiable) observations and predictions over these observations. We call these observations experience. Reinforcement learning is a computational approach that defines problems and solutions for learning through interaction by goal-oriented agents. We provide a survey of this field as well as the mathematical foundations of reinforcement learning, Markov Decision processes. A key step towards making reinforcement learning easier to use in real world applications is the ability to use data efficiently, as well as to use data sets generated off-line in the learning process. Off-policy learning algorithms aim to achieve this goal. Recently, [Precup et al., 2005] proposed a framework called recognizers, which facilitates off-policy learning. Recognizers can be viewed as filters on actions, which can be applied to a data set in order to learn expected returns for different policies. In the work so far, recognizers were hand specified. In this thesis, we present the first results on learning such recognizers from data. The main idea of the approach is based on eliminating from the recognition function actions that are deemed suboptimal. We provide theoretical results regarding the convergence in the limit to the optimal policy, as well as PAC-style convergence guarantees. We implemented our model and tested our system in different environments and under varying conditions. We provide empirical results illustrating the advantages of this approach.

Contents

List of Figures

Acknowledgements

The Master's thesis lying before you is the culmination of one year of graduate research and many years of study at Delft University of Technology. The bulk of the research presented here was done at the Reasoning and Learning Laboratory of the School of Computer Science at McGill University, Montreal, Canada. I started my research project in February 2006 and I have spend six months in total since May 2006 in Montreal at the Reasoning and Learning Laboratory. In this thesis I present my original research, but I would never have accomplished this without the great help and assistance of some people. I would like to take the opportunity here to express my gratitude and thank the following people:

- Leon Rothkrantz, for supporting me in going to Canada, for the many useful improvements that made this thesis possible and for always finding the time to discuss my work and my life.
- Doina Precup, for taking me into the lab at McGill, for making me feel welcome far away from home and for the great insights during our weekly meetings that always pushed me on.
- Anders Jonsson, for graciously providing me with his VISA code, and for patiently answering all my questions.
- My many friends in Montreal and at McGill, for the wonderful summer, the lively discussions and the good times.

Next to the contributors to my work, I would like to thank the following people for their support during this year of hard work, for making invaluable comments on my thesis, and for bearing with me when I had to work:

- Roujiar Manouchehri,
- Fam Seip,
- Joris Hulst, and
- Michel Meulpolder, Jelle Ten Hoeve and Jurriaan Verssendaal aka the 'VC'

Finally, my stay at McGill would not have been possible without the financial aid of the following parties:

- Malee Thai Restaurant,
- Huygens Scholarship Programme,
- Stichting Fundatie van de Vrijvrouwe van Renswoude, and
- Stimuleringsfonds Internationale Universitaire Samenwerkingsrelaties (STIR), Delft University of Technology.

Chapter 1 Introduction

This thesis is about reasoning under uncertainty and learning how to make decisions and take actions from experience. The area within artificial intelligence that studies this is called *reinforcement learning* (RL). We make decisions and take actions everyday, we decide what to eat for breakfast, wether we take the train or the car to work, we decide what stock to invest our money in. The decisions we make give rise to actions and the sequence of actions form our behavior. Reasoning under uncertainty is central to our lives and starts from an early age. The most interesting actions do not only influence our direct situation but will also have consequences in the long term. Decisions and their results, actions, are defining factors of behavior and most of our behavior is naturally learned from our own or others previous experience. We would like to investigate how we choose appropriate actions to reach goals and specifically how these choices are *learned* in a computational context.

In a broader sense this thesis deals with learning. Learning itself is a very broad concept so we need to define what learning is and demarcate the aspects of learning that we will concern ourselves with in this thesis to provide a focus. Learning is the process of gaining understanding that leads to the modification of attitudes and behaviors through the acquisition of knowledge, skills and values, through study and experience. We can discern and identify different modes of learning. We can learn from experience in a trial and error fashion, we can learn by imitating others or we can learn by studying knowledge and reflection. While all forms of learning are worthy of study, this thesis will focus on the first form of learning, learning through interaction. We will see in section 1.1 that psychologists also split learning in cognitive learning and associative learning.

As an example let us consider one of the most Dutch activities, bicycling. When we learn to ride a bicycle elements of reinforcement learning and sequential decision making can be seen. The goal is to ride the bicycle without falling over. In the beginning we simply try to ride a bicycle and after several actions we will most probably fall over. The experience and possibly the pain of falling over will lead us to reconsider the events (actions) causing the tumble and we will try again, this time perhaps turning the handle bar to the left instead of to the right. It is important to realize that no single action will cause the bicycle to fall over, but rather a series of actions taken after each other. With time and experience children all over the world learn to provide the right actions in each situation.

In this thesis we try to find principled ways of solving this and other such problems that are encountered every day in diverse fields.

1.1 Motivation

Learning by interacting with our environment is probably the first idea to occur to us when we think about the very nature of learning. When a baby plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exploring this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout human lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are profoundly aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is one of the foundational ideas underlying nearly all theories of learning and intelligence. In the most interesting cases our interaction at an earlier stage has an influence at a later stage.

In psychology a similar approach is sometimes called *behaviorism* or associative learning, which states that all behavior is a function of environmental histories of experiencing consequences. This approach was popularized by B.F. Skinner who argues that the only objective measurable aspects of learning are changes in behavior. This advocates a study of stimuli and responses and relegates the role of mental processes in learning as irrelevant or unnecessary at best since mental processes are not (objectively) observable. An implication of the behaviorist theory is that people are born as tabula rasas and must learn everything from birth.

This view of people as tabula rasas that learn only from past behavior was challenged by several psychologists, most notably Chomsky. Chomsky argued for the role of mental processes and beliefs in explaining behavior. This approach is known as cognitive learning. Our approach cannot be classified as one or the other. It departs from behaviorism because the learning algorithms we propose and their constructs such as recognizers are more akin to the mental processes found in the cognitive science approach, rather than simple summaries of past experiences. We do believe however that these mental processes must be formed by and grounded in experience. Because we wish to study computational models of reasoning we are merely inspired by psychological theories but can never confirm or falsify them.

Many problems in such diverse fields as economy, medicine, operations research and engineering can be formulated as sequential decision making problems, that is problems whose solutions require a sequence of interrelated actions. Examples of this are finding the optimal point in time to sell an option in financial markets, deciding what the right treatment is for a patient given certain symptoms in medicine, finding the optimal path for an assembly line at a factory in operations research or how to control a robotic arm to assemble machine parts for mechanical engineering. Finding the right behavior for often complex environments is becoming increasingly harder to do and a computational approach that assists and can even take over decision making is very valuable.

Ever since the advent of computers the earliest computer scientist have employed computers to assist with control and decision making. In world war II the first computers were used to detect enemy airplanes in the first, crude radars in Britain. With time both the technological complexity of computers and our understanding of decision making under uncertainty have greatly increased. At this point in time we have the critical mass to tackle a long standing goal of artificial intelligence, namely truly autonomous systems that can interact with the environment, extract meaningful knowledge from that environment and ultimately make and take decisions to control its environment. Another benefit that artificial agents bring to the table is the fact that they can often simulate the decision process when 'real' actions are too costly (sometimes even a matter of life and death).

Over the years many researchers have developed equally many approaches to intelligent systems and machine learning. We can roughly divide the machine learning field into supervised, unsupervised and reinforcement learning, each with their own algorithms and methods. Supervised learning is characterized by the use of labeled training data, whereas unsupervised methods use unlabeled training data to learn their structure. Reinforcement learning cannot be classified as either supervised or unsupervised learning. RL cannot be classified as supervised learning since it does not use labeled training data, nor can it be called an unsupervised learning method since it does use some kind of feedback (reinforcement) to learn its behavior. Figure 1.1 gives a non-exhaustive overview of the different learning methods. For an excellent overview of machine learning we refer to the standard textbook by Mitchell [Mitchell, 1997].

The scope of this thesis and the subject of our study will be reinforcement learning. Reinforcement learning is a computational approach that tries to find answers for the problem of sequential decision making by learning what decisions to make through interaction with the environment and it is inspired by ideas from psychology, neurology and animal learning. In reinforcement learning we learn what to do - how to map situations to actions - so as to maximize a numerical reward signal. We are not told which actions to take, as in most forms of machine learning, but we have to discover which actions yield the most rewards by trying them. In the most challenging and interesting cases, actions may affect not only the immediate reward but also the next situation and through that all subsequent rewards as is often the case in sequential decision making problems. Reinforcement learning provides us with the necessary tools to solve sequential decision making problems efficiently.

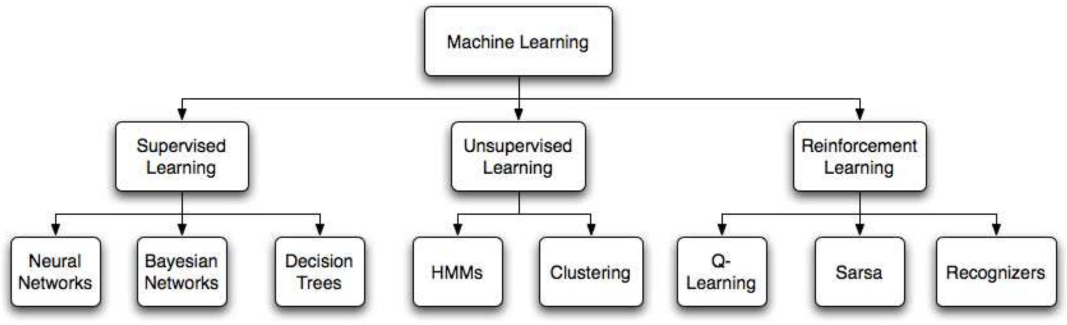

Figure 1.1: *Different elements of machine learning.*

Reinforcement learning itself is based upon a mathematical model of sequential decision making, called *Markov decision processes* that formalizes the problem of making sequential decisions and makes it amenable to rigorous mathematical analysis in order to objectively study different solution methods independently of specific problems that may bias the results.

While many promising results have been made with reinforcement learning and the applications of RL in industry are steadily increasing, we begin to encounter some limits of the current theory. In practice the techniques from reinforcement learning are difficult to apply because real world problems are so large that computing solutions for these problems would take too long. This is often called the curse of dimensionality since the problem grows exponentially larger in volume when adding extra dimensions. Recently the focus of research has shifted to higher level frameworks that extend and generalize the basic concepts of reinforcement learning in order to find solutions for larger problems. One of these developments are recognizers that enable computationally feasible solutions for larger problems.

Recognizers can best be described as a filter on the problem that tries to limit the problem space by considering only the relevant subspace that contains the solution. It effectively forms a corridor around the solution.

Until now recognizers had to be hand crafted and provided by the system designer which severely defeats its stated purpose of autonomously solving large scale problems. We need to answer questions such as: *How can we learn behavior from experience? In which way can we scale up these ideas to find solutions for large scale problems? Specifically, can we improve one of these ideas, recognizers, by making recognizers learnable autonomously from experience? How much experience (data) is needed before we can find a good solution?* These questions form the motivation and starting point for the research presented in this thesis.

This thesis adopts an engineering approach to experience, where effective multimodal interaction between artificial actors and an environment or between artificial actors and humans are central. The context of this interaction is always dynamic and may be stochastic. Instead of engineering experiences we are engineering with experience.

To clearly demarcate the scope and subject of this thesis we will summarize the focus of our thesis. We are interested in the way machines can learn and in this thesis we focus on learning through interaction. Of all the machine learning approaches that have been developed we focus on the reinforcement learning paradigm. We wish to study sequential decision making problems but limit ourselves to problems that can be formulated as a Markov decision process. Our aim is to investigate extensions of reinforcement learning that can solve larger problems that are currently unfeasible and focus on the study of recognizers. Within the scope of recognizers our primary concern is how we can learn the recognizer function relying solely on experience, explicitly trying to minimize the reliance on handcrafted recognizer functions that must be provided by human experts.

1.2 Applications

The impact of computational systems that learn from experience on society and industry is starting to be seen and will increase with the availability of principled methods and techniques. With the rising complexity of tasks that are candidates for automation, the need for adaptable dynamic systems increases. The lowering cost of computing power on the other hand enables the use of such systems. Our research contributes to the development of adaptive systems and makes the efficient use of such systems possible. To get a feeling for the implications of our work we will sketch two proposed applications of our research:

- Adaptive user interfaces.
- Mars rover.

1.2.1 Adaptive user interfaces

With the increasing functionality of applications and new possibilities opened up by technological advances, the user interface for these applications has become increasingly complex. While this in essence should make the application more powerful, most of the time users only need a subset of the functionality of a program. The added complexity can confuse or even hinder the effective use of the application. An application is only so powerful as its user interface allows it to be. An example of a contemporary application that provides a lot of functionality at an increase in complexity is seen in figure 1.2.

Some approaches to the problem of bloated user interfaces have been to allow the user to customize the interface or through static user modeling. These solutions do not recognize the dynamic nature of a user interface in relation to the user. Different users require different modes of communicating with the application but even a single user will change its preferences over time. An adaptive user interface that can learn from the user and evolves with a user can provide and maintain clear communication between user and application at all times. Recently a reinforcement learning approach to the problem has been proposed by [Jokinen et al., 2002]. The benefits of such an approach is that it is dynamic, the user interface will evolve with the user and it will adapt to specific users. In this scenario recognizers can be used to learn and focus the user interface as a function of user interaction with the program. In this case the recognizer

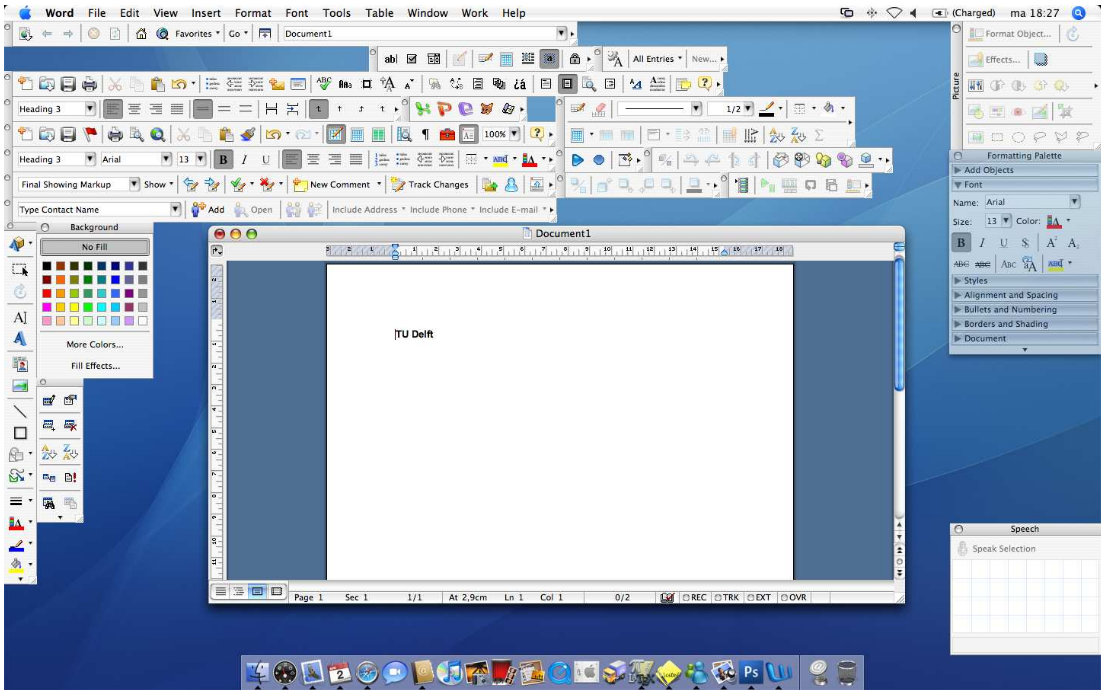

Figure 1.2: *Example of the user interface of MS Word.*

will highlight the actions it recognizes and hide the unused and unrecognized actions automatically without explicit input by the user. The experience is provided by the interaction of the user with the application.

1.2.2 Mars rover

Space exploration continues to drive technology and our understanding of the universe. In recent years the two largest agencies involved with space exploration, NASA and ESA have focussed on our closest neighbor, Mars, as the destination for its most ambitious projects. Although human missions are on the planning table, the near future will rely on robot missions on the surface of Mars. The difficulty of remote control from Earth due to communication lag and the complexity of the mission, require autonomous software for visual terrain navigation and independent maintenance.

The Canadian Space Agency is currently bidding for development of the Exomars rover which will be launched by ESA in 2011 as part of the Aurora program. The Exomars rover mission objectives is to study the biological environment of the martian surface, to search for possible martian life, past or present and to demonstrate the technologies necessary for such a mission. Figure 1.3 shows an artist rendition of the Exomars rover.

Our research will most likely contribute to the terrain navigation module of the Exomars rover. Using simulated data from earlier missions, recognizers can learn approximate policies for the various tasks of the Exomars rover which include establishing points of interest, navigating to those points and performing various scientific experiments. As this work is still in progress no results are yet available, chapter 9 provides results for a simplified model of a robot navigation task.

1.3 Problem Area

This thesis is part of the requirements for the Master of Science degree at the Man-Machine Interaction Group within the faculty of EEMCS of Delft University of Technology.

This thesis is a result of research done at the Reasoning and Learning Lab. The *Reasoning and Learning Lab* (RLL) is a research group within the School of Computer Science at McGill University, Montreal, Canada. The RLL is broadly concerned with the study of probabilistic systems. Areas of interest include Markov processes, decision making under uncertainty and reinforcement learning. The methods include the development of theoretical work and breakthroughs, the development of systems in collaboration with industry partners and empirical studies. Publications of interest from faculty of this group form the foundation of the options framework [Precup, 2000] and the introduction of the recognizer framework [Precup et al., 2005] which is the main subject of this thesis.

The assignment consists of three main parts:

1. A *theoretical* part that consists of investigating the existing reinforcement learning theory and its mathematical foundation, Markov decision processes, and investigating the current theory concerning recognizers.

2. A *proposed solution* which formulates the first ever learning algorithm that is able to learn the recognizer function autonomously from experience, and validating this proposed solution by a mathematical proof.

3. An *application* part that consists of implementing the proposed solution, applying it to benchmark problems, and performing a comparative analysis on the empirical results.

An elaboration of the different parts is given below.

1.3.1 Theoretical part

1. **Investigating existing RL theory** The state of the current research in reinforcement learning needed to be treated to provide a frame of reference for the novel research in this thesis. This part of the research is based on the standard works in RL [Sutton and Barto, 1998], [Kaelbling et al., 1996] and my previous research [Seip, 2006].

2. **Investigating the theory of Recognizers** Our main focus in this thesis lies in extensions of the recognizer framework so a thorough understanding of this framework is required before we can extend it. Recognizers have been introduced in [Precup et al., 2005] but since this is very recent, most work and knowledge of recognizers is still in active development. Most of the research presented in this thesis stems either from [Precup et al., 2005] or from direct collaboration on the subject between myself and professor Precup at the Reasoning and Learning Lab.

1.3.2 Proposed solution

3. **Developing learning algorithms for recognizers** The main contribution of this thesis is the development of the first ever learning algorithm that provides autonomous learning of the recognizer function solely from experience generated through interaction with the environment. This is is a radical departure from the current situation, where the recognizer function has to be provided to the agent by the (human) designer and no learning takes place. The impact of our approach is that the availability of the recognizer function does not rely on human experts anymore. The research in this part is entirely novel.

4. **Validating the proposed learning algorithm** To validate the correctness and the soundness of the proposed algorithm a mathematical proof has been developed that provides upper and lower bounds on the convergence properties of the algorithm. This proof guarantees that our proposed learning algorithm works.

1.3.3 Application part

5. **Applying the algorithm to benchmark problems** To verify the theoretical guarantees from the proof and empirically validate the algorithm, studies have been conducted with implementations of the algorithm for two benchmark problems in reinforcement learning. A comparative analysis is conducted with state-of-the-art existing algorithms in the domains.

1.4 Thesis Overview

The thesis is globally divided in five parts:

- Part I contains the theory needed to embed the research. Reinforcement learning is introduced in chapter 2; the mathematical framework of MDPs is treated in chapter 3; a treatment of Options, an extension of RL, is discussed in chapter 4; recognizers are formally introduced in chapter 5; chapter 6 surveys work related to our research.

- Part II contains our proposed algorithm in chapter 7; the proof is deferred to chapter 8. This part forms the main contribution of this thesis and is original research.

- Part III contains the empirical studies performed with the proposed algorithm on two benchmark problems; chapter 9 shows our results for the gridworld environment, a navigation task; chapter 10 applies our algorithm to a special case of MDPs and shows the synergy between recognizers and options.

- Part IV formulates the conclusions and indicates the next steps that can be taken to develop the research in this thesis further.

- Finally, part V contains the code used in the empirical studies of part III. Appendix A lists the documentation of the experiments.

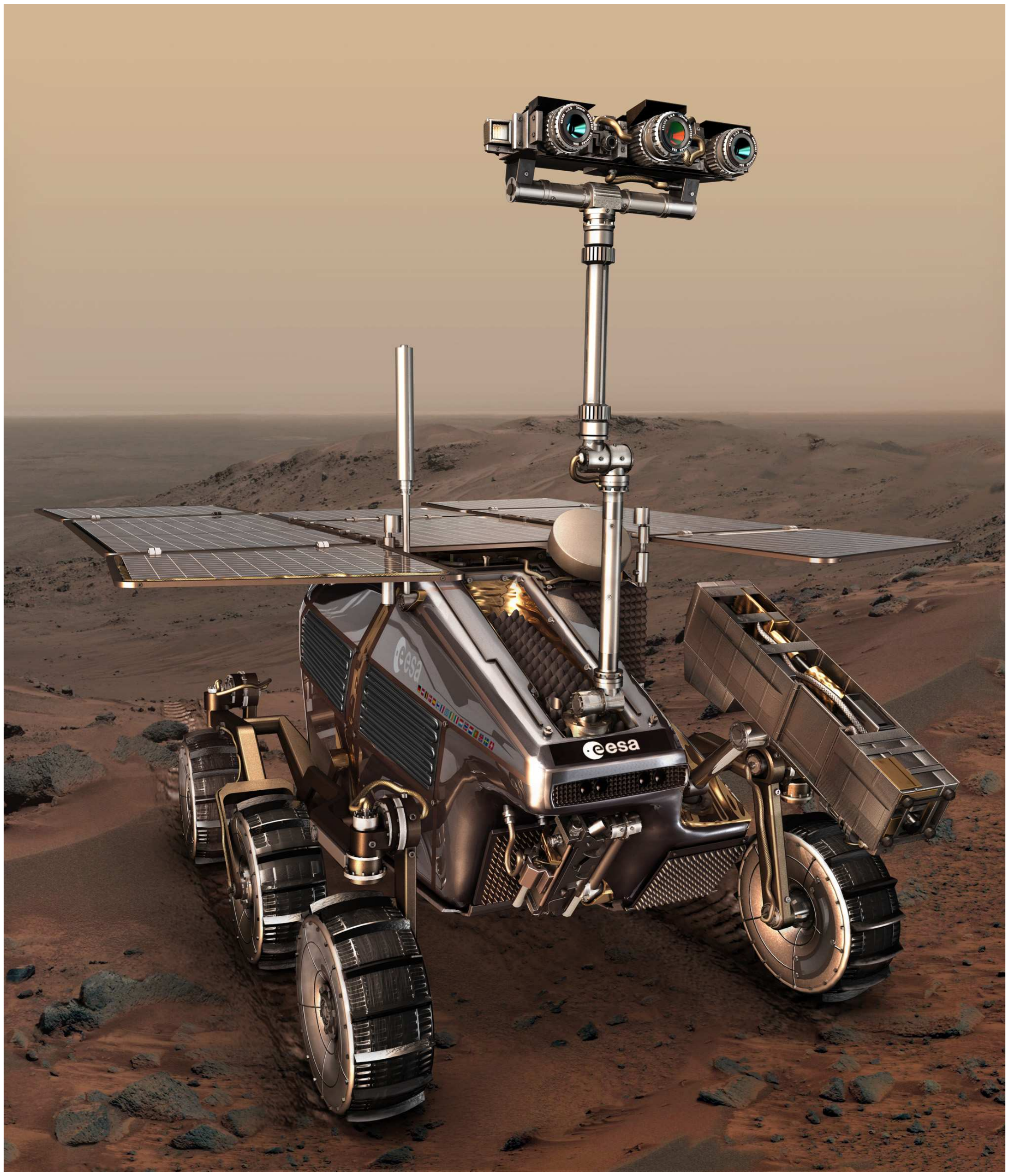

Figure 1.3: *Artist rendition of the Exomars rover.*

Part I

Theory

‘The past cannot remember the past.
The future can’t generate the future.
The cutting edge of this instant right here and now
is always nothing less than the totality of everything there is.”
- Robert M. Pirsig
- Zen and the art of mortorcycle maintenance.

Chapter 2

Reinforcement Learning

Reinforcement learning is currently one of the most active research areas in artificial intelligence. Reinforcement learning can be described as a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment.

In this chapter we give a clear and simple account of the key ideas and algorithms behind reinforcement learning. We describe the theoretical and mathematical background of reinforcement learning as it is essential to understanding the work in this thesis. This chapter deals with the intuitive notions underpinning reinforcement learning and this notion will be gradually refined until we arrive at the subject of the next chapter, Markov decision processes. Section 2.1 defines the subject of study in reinforcement learning. In Section 2.2 we will talk about the core concepts found in reinforcement learning. To demarcate the field Section 2.3 elaborates on the fundamental difference between reinforcement learning and supervised learning. We conclude with a short overview of the origins of the field in Section 2.4 and a summary in Section 2.5.

2.1 The Reinforcement Learning Problem

Reinforcement learning is learning what to do –how to map situations to actions– so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics –trial-and-error search and delayed rewards– are the two most important distinguishing features of reinforcement learning. Central to reinforcement learning is the concept of an agent. The agent is the learning system, able to reason and learn from its environment and able to exercise control over its environment through the actions at its disposal. As such we define reinforcement learning as a computational approach to understanding and automating goal-directed learning and decision-making. It emphasizes learning what to do by interacting with the environment and obtaining feedback from actions. The feedback, or the reward signal as it is called, guides the choice of actions. In reinforcement learning the learner must discover which actions yield the most reward by exploring actions, instead of observing known good (labeled) actions as is usual in supervised learning. In the RL context the agent is able to perceive (parts of) the environment and take actions to alter the state of its environment. As mentioned, in the most interesting and challenging cases, the actions of the agent may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. Therefore we are not only interested in maximizing the immediate rewards but instead we are interested in maximizing the return, which is the sum of all subsequent rewards. Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. A diagram of this view can be seen in Figure 2.1. Another key feature of reinforcement learning is that it explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment. This is in contrast with many approaches that consider subproblems without addressing how they might fit into a larger picture. For example, much of machine learning research is concerned with supervised learning without explicitly specifying how such an ability would finally be useful. Other researchers have developed theories of planning with general goals, but without considering planning's role in real-time decision-making, or the question of where the predictive models necessary for planning would come from. Although these approaches have yielded many useful results, their focus on isolated subproblems is a significant limitation.

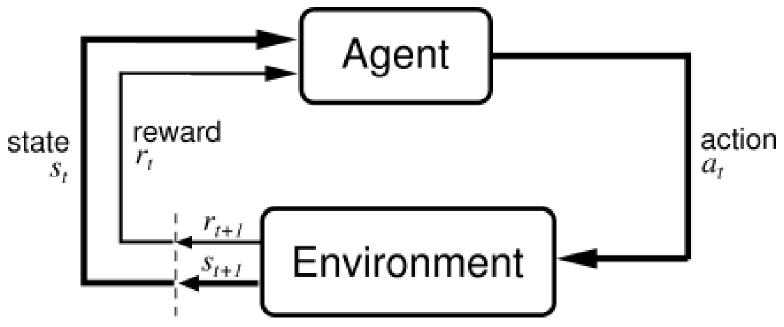

Figure 2.1: *Abstract view of the reinforcement learning model.*

2.2 Elements of Reinforcement Learning

The best way to illustrate the principles of reinforcement learning is to take a closer look at the example of the first chapter, learning to ride a bicycle without falling over. In this case the agent can control the bike by steering to the left or to the right and the agent receives a positive reward for every time-step that it bikes and large negative rewards if the bike falls. In the first trial the agent will perform actions randomly because it does not know how to ride a bicycle and will fall over fairly quick. During these trials the agent will associate its actions with negative rewards and try another action when faced with the same situation until after some time it learns how to keep the bicycle from falling over.

The example above is fairly simple but it already contains the basic elements of reinforcement learning. The agent has some perception of its *environment*, in the example this would be the tilt of the bicycle. The agent has to choose among available *actions*. After each choice of action it receives a *reward* from the environment and the state has changed. The reward signal enables the agent to evaluate its actions. We will see that the reinforcement learning agent effectively learns it behavior by generalizing over past experiences.

The reinforcement learning system can be formalized by identifying four sub-elements:

- the environment,
- reward function,
- policy,
- value function.

We will look at each element in more detail. Every RL system learns a mapping from situations to actions by trial and error interactions with a dynamic *environment*. This environment must at least be partially observable by the reinforcement learning system. The observations may be low level sensory readings, symbolic descriptions or even abstract mental situations. The actions in the environment can also vary from low level to abstract. The choice of states and actions provides the frame of reference for the RL system.

The *reward function* is a mapping from states or state-action pairs of the environment to a scalar reward that indicates the desirability of that state or state-action pair. The objective of any reinforcement learning agent is to maximize the total reward it receives over time. The reward function has some likeness to the concepts of pain and pleasure in a biological system. It is very important to realize that the reward function must be external and unalterable to the agent. The agent can however use the results to alter its policy. reward functions may be stochastic.

A *policy* defines the behavior of an agent. Essentially a policy is a mapping from perceived states of the environment to actions that should be taken in those states. Policies can be represented in a number of ways, from simple lookup tables to extensive search processes. The main characteristic that all policy representations have in common is that they are an input-output mapping from states to actions. The representations differ in the level of sophistication with which they perform this mapping. In the rest of this thesis we will encounter some examples of the more complex representations. The policy is the core of a reinforcement learning agent because a policy alone is sufficient to determine an agents' behavior. Our work on recognizers will deal with the efficient learning and representation of policies. In general, policies may be stochastic. We can think of the policy of an agent as the *strategy* that the agent employs to solve the task.

The reward function returns the immediate reward (which may be positive, negative or even zero) but the reward function alone does not provide the agent with enough information to formulate a good policy and solve its task. A myopic agent that greedily chooses the action with the highest one-step reward could never guarantee finding the optimal solution. The agent needs a persistent notion of long term reward and the *value function* provides this. The *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state and following some policy π (which must be provided to the value function). As such the value function takes into account states that lie further away. For example some states may always yield a low immediate reward as dictated by the reward function but still have a high value because it is often followed by other states that yield high rewards. The opposite can also be true, states with high immediate rewards may offer poor value in the long run. It must be clear that if only the reward function is used to guide action selection, a different policy will result than if the value function is used instead. In general using the value function will result in a better policy. It can be shown that the value function yields the optimal policy under certain conditions. Unfortunately it is much harder to determine the value function than it is to determine the reward function. Rewards are experienced directly while values must be estimated from sequences of observations. The value function is sometimes also called the *utility function* most notably in [Russel and Norvig, 1995].

A choice that has to be made is the definition of optimal behavior. This will specify how the agent takes the future into account. It turns out that we can actually have different definitions of optimal behavior. There are three models that are widely used in the machine learning community:

- finite-horizon model,
- infinite-horizon model,
- average-reward model.

The *finite-horizon* model optimizes the expected reward for the next h steps. This model is appropriate if the task at hand is *episodic* and h is bounded and known. Tasks that are episodic continue until some sort of an end-state is reached after which a new episode starts. Games like chess or backgammon are good examples of problems that are episodic. This model is also preferential when there is little long-term correlation in the sequences. Alternatively *continuous* tasks do not have a set end-point or no end-point at all. Equation 2.1 calculates the return for the finite-horizon model:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{t+h}. \tag{2.1}$$

The *infinite-horizon* discounted model in principle takes all future rewards into account but rewards that are received in the future are geometrically discounted with a discount factor γ, (where $0 \le \gamma < 1$). This is done because future rewards are uncertain. This practice is common in many financial asset valuation instruments such as a discounted cash flow, using a discount rate. The infinite-horizon model is more suitable for continuous tasks. Unless explicitly stated the infinite-horizon discounted model is assumed in this text. Equation 2.2 gives the return using the infinite-horizon discounted model:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

Thirdly an *average-reward* model can be used when we deal with cyclical tasks. In this case the agent tries to optimize the reward per time step based on the criterion of an average reward. The intuition is that higher than average rewards imply desirable actions. Average reward reinforcement learning is discussed in detail in [Mahadevan, 1996].

2.3 Reinforcement Learning vs. Supervised Learning

It is important to realize that reinforcement learning is fundamentally different from supervised learning. There is an essential distinction between instruction and evaluation. In a supervised learning task the agent is presented with labeled input-output pairs that *instruct* the learner which actions are good. Supervised learning thus depends on the availability of large pre-labeled data sets that can be costly to obtain. A corpus for speech recognition is an example of such a large and costly dataset. In reinforcement learning there are no samples of good actions available. An agent takes an action after which it receives a reward and moves to another state, but it is not told which action would have been the best in the long term. A reinforcement learning agent thus has to *evaluate* the actions it takes on the basis of reinforcements it receives from the environment. This distinction is sometimes also called the distinction between *selective* and *evaluative* instruction [Sutton and Barto, 1998].

The difference between supervised and reinforcement learning is also apparent in the data we need to train the learning systems. In supervised learning we need to label the data beforehand, in reinforcement learning the raw data is sufficient, as the feedback comes from the interaction with the environment.

Finally the difference between the two machine learning methods can be found in the way they are trained. Indeed the concept of training the learner has a different meaning in both methods. Supervised learning is mostly done offline where labeled training data is fed to the system. During operation the system does not usually learn. Reinforcement learning methods on the other hand use online learning. The RL agent basically learns from every step it takes, while it is taking them. Training in a reinforcement learning context often means the phase where the agent operates in a controlled environment, like a simulator, to learn an initial policy after which it is 'set loose' in the actual domain and most of the time continues to learn. Reinforcement learning always uses the experience it generates to learn from.

2.4 History of Reinforcement Learning

Reinforcement learning is a multidisciplinary field and encompasses research from many fields: Psychology, Control Engineering, Artificial Intelligence and Neuroscience are the most visible of these. The different viewpoints complement and challenge scientists from all these disciplines. A short overview of the evolution of reinforcement learning shows the various research areas that contributed to the development of reinforcement learning.

The oldest ideas that lie at the base of reinforcement learning were formulated in the field of animal psychology and centered around the idea of trial and error learning. One of the first crisp formulation of this idea may very well be Thorndike's *Law of Effect* [Thorndike, 1911]. This research established the key ideas of exploration and association in RL. The philosophy of *behaviorism* in psychology also played a role in popularizing the idea that behavior can be studied as a response function of experience. The most well known advocate was Skinner [Skinner, 1938].

Largely independent from the work of psychologists, researchers in control engineering investigated the problem of designing a controller to minimize a measure of a dynamical systems behavior over time. Bellman developed a mathematical approach called dynamic programming [Bellman, 1957] which provides much of the theoretical foundations of RL including the Bellman equation and Markov decision processes.

Dynamic programming, as an optimal solution, often requires complete knowledge of the world and a lot of computing time. A lot of the research in RL can be viewed as approximations to DP that relaxes some of its assumptions.

The idea of programming a computer to learn by trial and error dates back to the early days of computers and artificial intelligence. Minsky was one of the earliest researchers to discuss a computational model of reinforcement learning [Minsky, 1954]. Samuel was one of the first computer scientists to develop a reinforcement learning program in 1959. His checkers program tried to learn an evaluation function for checkers by using evaluations of later states to generate training values for earlier states. As such it can be thought of as a forerunner of the ubiquitous *backup* operation found in modern RL.

Thirty years later another game, backgammon, would become the subject of one of the most successful applications of reinforcement learning to date. Developed by Tesauro at IBM, TD Gammon is a backgammon program that plays on the level of the world's best human players [Tesauro, 1995].

Another early success for reinforcement learning is due to Michie and Chambers (1968). They employed reinforcement learning techniques to balance a pole on a moving cart.

After a relatively quiet period for RL in the 60's and 70's, Klopf is credited with reinvigorating the field by tying together the trial and error branch of RL with temporal difference learning, a technique that will be discussed in the next chapter.

Sutton and Barto continued to develop the ideas of Klopf in the eighties and introduced many of the current terms used in reinforcement learning. Most notably they established temporal difference learning as the main paradigm in RL. Sutton and Barto are also noted because they wrote the standard introduction to reinforcement learning [Sutton and Barto, 1998].

In 1989 Watkins introduced Q-learning, a model free approach to reinforcement learning and dynamic programming [Watkins, 1989]. Most current algorithms are based on the work developed by Watkins.

More recently the attention of the research community has shifted towards scaling the techniques of reinforcement learning to large state spaces. *Hierarchical* methods for reinforcement learning aim to aggregate states in order to arrive at more compact representations of the problem space. An excellent overview can be found in [Barto and Mahadevan, 2003] and a framework for hierarchical reinforcement learning is developed by Precup [Precup, 2000]. We will treat this framework in chapter 4.

2.5 Summary

In this chapter we introduced the basic concepts and ideas of reinforcement learning. We discussed the core elements of reinforcement learning (environment, reward function, policy and value function) and clarified the difference of the reinforcement learning approach versus the supervised machine learning approach. We also provided a background history of the major developments in the field.

Chapter 3

Markov Decision Processes

The mathematical foundation of reinforcement learning is provided by the study and theory of Markov decision processes (MDPs). Markov decision processes model sequential decision making under uncertainty and take into account both the outcomes of current decisions and future decision making opportunities.

Every day we make many decisions that have both immediate and long-term effects. Decisions are rarely made in isolation and we need to constantly make a trade-off between our short term objectives and long term goals. To decide what field of study to pursue for instance has an effect on the rest of a persons life and career. To achieve a good overall performance we need to uncover and take into account the relationship between present and future decisions. Without this our decisions will be myopic and our results can be suboptimal. When running a marathon for example, deciding to sprint at the beginning may deplete energy and result in a poor finish.

This chapter presents Markov decision processes and relates them to our subject of study, reinforcement learning. Section 3.1 describes the model we use throughout this thesis. After defining the model, section 3.2 provides the basic solution methods for fully specified MDPs. Section 3.3 generalizes the solution methods to stochastic environments where the model parameters are not assumed to be known in advance. The solution method presented in chapter 7 in this thesis is of this form. We will also discuss some advanced topics. Section 3.4 presents an alternative solution method, policy search and section 3.5 provides an extension to the original model to enable practical solutions for large and continuous state spaces. The chapter concludes with a summary in section 3.6. For the interested reader the excellent book on this subject by [Puterman, 1994] gives a complete treatment of MDPs.

3.1 The Sequential Decision Model

Most reinforcement learning research is based on the formalism of Markov Decision Processes. MDPs provide a general framework for sequential decision making under uncertainty. Before we give a formal definition of the model used throughout this thesis a short refresher of the Markov property is in order.

From probability theory we know that a stochastic process is said to have the Markov property when the conditional probability of future states of the process, given the present sate and all past states, depends only upon the present state and not on any past states, in other words it is conditionally independent of the past states (the path of the process) given the present state. Mathematically a stochastic process $X(t), t > 0$ has the Markov property if:

$$P[X(t+h) = y | X(s) = x(s), \forall s \leq t] = P[X(t+h) = y | X(t) = x(t)], \ \forall h > 0. \tag{3.1}$$

In the context of decision processes this means that if the current state of the MDP at time t is known, transitions to a new state at time t+1 are independent of all previous states.

We now define the model used throughout this chapter and thesis. A MDP consists of:

- a set of states $\mathcal{S}$,
- a set of actions $\mathcal{A}$,
- a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$, and
- a state transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$, where a member of $\mathcal{P}(\mathcal{S})$ is a probability distribution over the set $\mathcal{S}$ (it maps states to probabilities). $\mathcal{P}(s, a, s')$ is written to denote the probability of making a transition from state s to state s' under action a.

The state transition function incorporates a notion of uncertainty. A model is defined to have the Markov property if the current state sums up all information of previous states. In mathematical terms the Markov property applies if the state transitions are independent of any previous states or actions. The goal is to maximize a measure of the return, like the infinite-horizon discounted reward from section 2.2.

3.2 Dynamic Programming

Before considering approximate algorithms for learning to behave in MDP environments, in this section the optimal solution is presented for the ideal case when a correct model of the transition probabilities and reward function is available. The solution presented originates from dynamic programming (DP) [Bertsekas and Tsitsiklis, 1996]. Although many real world applications do not adhere to these assumptions it serves as the basis of many if not all reinforcement learning techniques as they try to relax some of the assumptions made in dynamic programming. Reinforcement learning can be viewed as trying to do dynamic programming with less computation and less knowledge. Before this solution is presented some additional concepts have to be introduced.

The solution takes the form of a policy π for the environment, where the goal is to find a *good* policy i.e. one that gives a good reward in the long run or simply a good return. Recall from Section 2.2 that a policy is defined as:

$$\pi : \mathcal{S} \to \mathcal{A} \qquad \forall s \in \mathcal{S}$$

To estimate returns we define the value function for MDP's, introduced in 2.2:

$$V^\pi = E_\pi\{R_t|s_t = s\} = E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\Big\} \tag{3.2}$$

This is called the state-value function and gives the expected return when starting from state s and following policy π thereafter. A model of the state transition probabilities is needed to compute the state-value function.

Analogously the action-value function for policy π can be defined:

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\Big\} \tag{3.3}$$

Q^π returns the value of taking action a in state s under policy π. The action-value function enables construction of model free methods since the state transition probabilities are no longer directly required.

The value function of a state depends on the value function of its successors. This recursive property is fundamental into solving the RL task. The value function is the link between the directly observable rewards and the policy. For any policy π and any state s, the following relationship holds:

$$\begin{aligned}
V^\pi &= E_\pi\{R_t|s_t = s\} \\
&= E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\Big\} \\
&= E_\pi\Big\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s\Big\} \\
&= \sum_a \pi(s,a) \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma E\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s'\Big\}\Big] \\
&= \sum_a \pi(s,a) \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma V^\pi(s')\Big],
\end{aligned} \tag{3.4}$$

where

$\pi(x, a)$	probability of taking action a in state x according to policy π
γ	discount rate $0 \leq \gamma < 1$

equation 3.4 is called the *Bellman equation* and relates the value of state s to the values of its successor states. The bellman equation gives the value of a state by recursively determining the rewards of its successor states and averaging over all the possibilities.

At this point the value for all states in a given policy can be evaluated. For finite MDP's an optimal policy can be defined with the help of value functions. Value functions define a partial ordering over the set of all possible policies. A policy π is defined to be better when its expected return (value) is greater or equal than that of policy π' for all states. So:

$$\pi \geq \pi' \iff V^{\pi}(s) \geq V^{\pi'} \quad \forall s \in S \tag{3.5}$$

Consequently, a policy is an optimal policy if its value function is greater than or equal to the value functions of all other policies for a given set of states. The optimal policy function is defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \forall s \in S \tag{3.6}$$

and satisfies Bellman's optimality equation:

$$V^*(s) = \max_{a} \sum_{s'} P(s, a, s') \Big[R(s, a, s') + \gamma V^*(s') \Big] \tag{3.7}$$

3.2.1 Policy iteration

The search for an optimal policy can now be structured as follows. Starting with a (random) policy π, a value function can be computed. This step is called *policy evaluation*. When the value function V^{π} is available it is possible to find a better policy π'. Now π' can be constructed using equation 3.3. When in a state s the best action is chosen on the basis of a one step look ahead and thereafter π is followed than the new policy π' will be equal or better than the old one. This can be extended to all states. This is called *policy improvement*. The π' is called the greedy policy with respect to the value function because we choose the actions greedily, based on a short term reward. Now this procedure can be repeated using π' as the policy. This process terminates when there is no improvement in the policy anymore. The resulting algorithm is called policy iteration in the DP literature. In this way a sequence of monotonically improving policies and value functions will be obtained:

$$\pi_0 \rightarrow^E V^{\pi_0} \rightarrow^I pi_1 \rightarrow^E V^{\pi_1} \rightarrow^I pi_2 \rightarrow^E \ldots \rightarrow^I pi^* \rightarrow^E V^*,$$

where $\rightarrow^E$ denotes a policy evaluation iteration and $\rightarrow^I$ denotes a policy improvement iteration. The value function at the end is the unique optimal value function and the greedy policy with respect to this value function will be the optimal policy.

This interaction between value function and policy lies at the very core of reinforcement learning. the value function of a policy can be thought of as looking ahead from one state to its possible successor states. The value for the state is then *backupped* from the future states to s. This backup can be illustrated using *backup diagrams* [Sutton and Barto, 1998]. They are called backup diagrams because they show the relationships of the update or backup operations that form the basis of reinforcement learning methods. Figure 3.1 shows the backup diagram for V^{π} and Q^{π}:

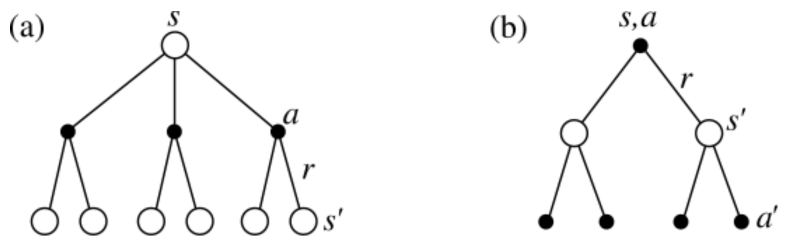

Figure 3.1: *Backup diagrams for (a) V^{π} and (b) Q^{π}.*

Each open circle represents a state and each solid circle represents an action. Starting from state s, the root node at the top, the agent could take any of some set of actions (three are shown in figure 3.1). From each of these, the environment could respond with one of several next states s', along with a reward, r. The Bellman equation 3.4 averages over all the possibilities, weighing each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward along the way.

3.2.2 Value iteration

A drawback of policy iteration is that each iteration involves policy evaluation, itself an iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence to V^{π} is only realized in the limit. For any but the most small state spaces this is not feasible. A key observation is to realize that it is not necessary to wait until V^{π} is reached exactly before terminating the policy evaluation step. It is possible to stop each policy evaluation after performing only one sweep (one backup for every state). The resulting algorithm is called *value iteration* and often achieves faster convergence than policy iteration. The equation for the backup operation combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} V_{k+1} &= \max_a E\{r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V_k(s')], \end{aligned} \tag{3.8}$$

for all $s \in \mathcal{S}$. equation 3.8 is derived from equation 3.7 simply by turning the Bellman optimality equation into an update rule. Algorithm 1 gives a complete value iteration algorithm. It terminates when successive improvements fall below some threshold θ.

Algorithm 1 Value Iteration.

```
Initialize $V$ arbitrarily, e.g., $V(s) = 0$, for all $s \in S^+$

repeat
  $\Delta \leftarrow 0$
  for each $s \in S$ do
    $v \leftarrow V(s)$
    $V(s) \leftarrow max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  end for
until $\delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that $\pi(s) = argmax_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
```

It is instructive to look at the backup diagram (figure 3.2) for value iteration and compare this to the diagram for policy iteration in figure 3.1. It is essentially the same backup structure except that in the value iteration case the maximum over all actions is taken instead of its expectation.

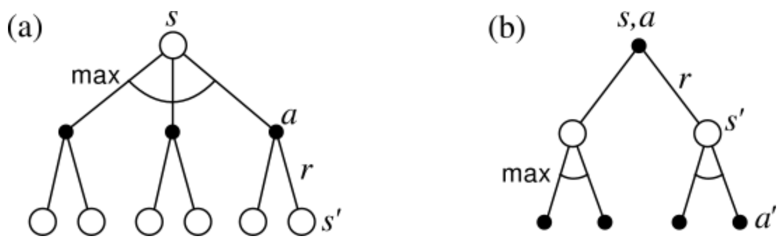

Figure 3.2: *Backup diagrams for (a) V^* and (b) Q^*.*

The methods used in DP and, in general, in all reinforcement methods follow a common strategy. There are two interacting processes at work the (approximate) policy function and the (approximate) value function. Each process uses the intermediate result of the other as its input to improve its estimate, which in turn will be used by the other process to improve its own estimate. This continues iteratively until a fixed point is reached where the policy and the value function do not change anymore and consequently are optimal. This concept is called *General Policy Iteration*. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the current value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution. For DP methods it has been proved to converge to the optimal solution. For other methods this proof has not yet been found but still the idea of GPI improves our understanding of the methods.

The algorithms presented so far require updating the value function or policy for all states at once. It is however not necessary to perform DP methods in complete sweeps through the state set. There exist asynchronous DP methods that back up states in an arbitrary order and do not have to visit each state. The ability to concentrate on a subset of states allows for much more efficient algorithms. The result is a locally good policy which can still converge to the optimal solution provided that the unvisited states are unlikely to occur. Asynchronous policy iteration makes sense in many real world applications: it naturally emphasizes states that are visited more often, and hence more important. If some states are unreachable it is not necessary to have an exact value for that state. Many of these methods can be viewed as fine-grained forms of GPI. Our work in this thesis uses this insight to arrive at our solution.

Dynamic programming methods allow for the computation of optimal policies given the correct model. All the above presented methods share the property that they update or backup estimates of the values of states based on estimates of the value of successor states. This is called *bootstrapping*. The DP methods propagate the information throughout the system using the backup operations. As stated earlier often this model is not available. In the next section methods will be developed that are able to learn policies from experience only, without requiring a model.

3.3 Temporal Difference Learning & Monte Carlo Methods

In many of the most interesting cases a complete model of the environment (state transition probabilities) is difficult or even impossible to obtain. Classical DP methods cannot be used when such a scenario is encountered. In this section we will present computational methods that are able to learn from experience by interacting with the environment. The algorithms presented in this section form the central core of reinforcement learning. Monte Carlo (MC) methods on the one hand require only sample trajectories to learn the value function. Temporal Difference (TD) learning methods provide a bridge between the Dynamic Programming methods from section 3.2 and Monte Carlo methods. Monte Carlo methods are also seen as a limiting case of Temporal Difference methods. This connection will be addressed in this section.

A reinforcement learning method has two integral processes as stated in 3.2, policy evaluation and policy improvement. First the problem of finding a value function given a policy will be treated (policy evaluation), then the policy improvement will be discussed. Recall that the value of any state is the expected return—expected sum of future discounted rewards. A simple way to obtain an estimate for a state is to average the *observed* return after visits to that state. A sample return can be easily generated by following a policy *on-line*, while interacting with the environment. Monte Carlo methods do exactly this, they keep a count of all rewards received after visiting each state and use the average of all rewards as the estimate for the value of that state. For continuous tasks that may continue indefinitely, defining returns can be complicated. Therefore Monte Carlo methods are only defined for episodic tasks (Section 2.2). MC methods are thus incremental on an episode by episode basis.

Temporal Difference methods do not have to wait until the end of an episode before improving the estimates of the state values. They combine learning from sample experiences as used in Monte Carlo methods with the propagation of intermediate results of DP (bootstrapping) to learn from every step taken. Temporal Difference methods are thus incremental on a step by step basis. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s)$ for all states encountered, because only at that time can the return be calculated, TD methods only have to wait until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward and the estimate of the value of the next state. The difference can be seen by looking at equation 3.4 from section 3.2 which we will repeat here for convenience:

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} &\quad (3.9)\\
&= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\}\\
&= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right\}\\
&= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \middle| s_t = s\right\}. &\quad (3.10)
\end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of (3.9) as a target, whereas DP methods use an estimate of (3.10) as a target. The MC target is an estimate because the expected value in 3.9 is not known, a sample return is used instead. The DP target is an estimate not because of the expected value (the correct model provides this) but because $V^{\pi}(s_{t+1})$ (the value of the next state) is not known and the current estimate, $V_t(s_{t+1})$, is used instead. The Temporal Difference target is an estimate for both reasons it uses a sample return for the expected value like MC *and* it uses the current estimate V_t instead of the true value function V^{π} like DP. Thus the TD methods combine the sampling of Monte Carlo with the bootstrapping of DP.

The simplest TD method, known as TD(0), is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \quad (3.11)$$

where α is a positive learning rate parameter and $[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is sometimes called the TD-error since it basically 'moves' the estimate towards its true value.

If we take a look at the backup diagrams for TD and Monte Carlo methods (figure 3.3) the difference between these methods and the DP methods as shown in figure 3.1 and figure 3.2 can be clearly seen. Temporal Difference and Monte Carlo methods use *sample backups* as they are based on a single sample successor state path whereas DP methods are based on the complete distribution of all possible state paths. The difference between Temporal Difference methods and Monte Carlo methods is then that TD requires only one step before making a backup while Monte Carlo methods have to wait until the terminal state before backing up the estimates.

The actual learning of the value functions for TD and MC methods can be done in two ways: *on-line* and *off-line*. During on-line learning the agent really interacts directly with the environment, taking actions and receiving rewards, as it tries to learn the value function. The great advantage is that no model of the environment is required. A disadvantage can be that the environment can be dangerous, costly or slow, hampering learning. In off-line learning the environment is simulated and thus a model is required. This model can be relatively simple though and does not even require that the transition

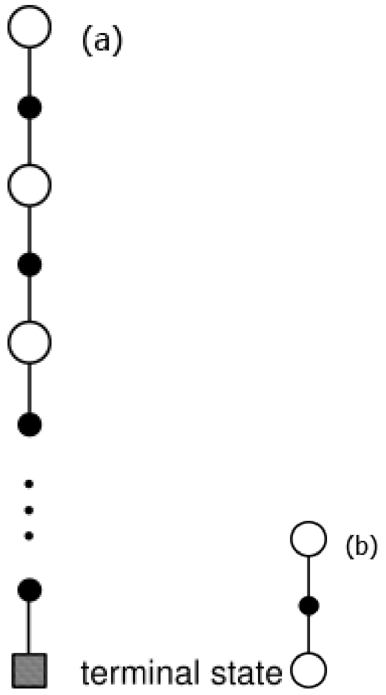

Figure 3.3: *Backup diagrams for (a) Monte Carlo methods and (b) TD(0).*

probabilities are known, the model only needs to generate sample transitions. The advantage of this method that it can be relatively cheaper and safer to learn than in the real environment. The agent can also learn faster than real time in a simulator. When considering a trading agent for instance, the goal is to learn a strategy to profitably trade some assets like stocks or houses. It is beneficial in this case let the agent learn from off-line experience first (where its mistakes do not cost millions) before allowing it to handle large sums of capital.

Policy iteration is relatively straightforward using TD and MC methods. After one or possibly multiple iterations of policy evaluation (finding the values of the value function) a new policy can be constructed by choosing the maximum over the state action pairs for each state using equation 3.12. This is called greedy action selection since we take the current best action in every state.

$$\pi'(s) \leftarrow arg \max_a Q(s,a) \tag{3.12}$$

Reinforcement learning problems can now be solved using the framework of General Policy Iteration (Section 3.2.2), only now using TD or MC methods for the evaluation part.

A common problem that the TD and MC methods suffer from is the problem of *maintaining exploration*. This occurs because greedy action selection may result in some states not being visited very often, which in turn can lead to suboptimal results. This tradeoff between exploration and exploitation was mentioned in 2.2 and forms a fundamental element of reinforcement learning. To ensure convergence we need to make sure that, in the limit, all actions and all states have been sampled infinitely often.

If we do not ensure this there will always be a chance that the value of the 'neglected' actions and states are maximal and therefore essential for any optimal policy. There are two approaches to ensure this: *on-policy methods* and *off-policy methods* (Not to be confused with on-line and off-line learning).

In on-policy methods exploration is maintained by following a policy that assigns a non-zero probability to each state-action pair, so that when the number of transitions approach infinity, all states will be visited a lot of times. Policies that have this property are called soft policies. An example of such policies are *ϵ-greedy* policies, meaning that most of the time they will select an action that has maximal estimated action value, but with probability ϵ they will choose any non-maximal action at random. On-policy methods learn such a soft policy. The policy that is being learned and the policy that is being used is the same one. In reinforcement learning we discern between the *target* policy and the *behavior* policy. For on-policy methods these two policies are the same.

Off-policy methods on the other hand separate the policy that is used for action selection from the policy that is learned. The policy that selects the actions, called the *behavior* policy is still soft so it ensures that all actions and all states are sampled. The policy that is learned, the *target* policy however can be different and in fact usually is completely greedy, In the sense that it always chooses the optimal action.

In the next sections two of the most widely used reinforcement learning algorithms will be presented, *Sarsa* is an example of on-policy methods while *Q-learning* is an example of off-policy methods.

3.3.1 Sarsa: on-policy

The name Sarsa is an acronym for *State - Action - Reward - State - Action*, we will shortly see why. It was first investigated by [Rummery and Niranjan, 1994] under the name *modified Q-learning*.

The basic ideas of GPI (Section 3.2.2) carry over for this algorithm but an important distinction is that instead of learning the state-value function, the action-value function from Section 3.2 is learned. As an on-policy method $Q^\pi(s,a)$ must be estimated for the current behavior policy π and for all states s and actions a. Essentially the same algorithm as used to compute V^π can be used because conceptually there is no difference: to learn V^π, transitions from state to state are considered, to learn Q^π, transitions from state–action pair to state–action pair. Both are Markov chains with a reward process. The update equation now becomes:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]. \tag{3.13}$$

Here α is a learning parameter. This update is called for every nonterminal state s_t. If it is a terminal state the update will be zero. The name sarsa stems from the fact that the update uses the five elements $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$. The complete algorithm is given in algorithm 2.

Sarsa converges with probability 1 to an optimal policy and optimal action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be accomplished, for example, by an *ϵ-greedy* policy in combination with a simulated annealing approach where the ϵ is made smaller over time).

Algorithm 2 Sarsa: An on-policy TD algorithm.

```
Initialize $Q(s,a)$ arbitrarily
**repeat**
  (for each episode):
  Initialize $s$
  Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
  **for** each step of episode **do**
    Take action $a$, observe $r, s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'; a \leftarrow a'$;
  **end for**
**until** $s$ is terminal
```

3.3.2 Q-Learning: off-policy

Q-learning is arguably the most important breakthrough in reinforcement learning in the last twenty years.
Developed by Watkins [Watkins, 1989], [Watkins and Dayan, 1992], Q-learning is independent of the policy being followed and directly approximates the optimal action-value function Q^*. The update equation for *one-step Q-learning* is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{3.14}$$

The difference with sarsa is that now the Temporal Difference error is calculated between the maximal Q-value of the next state and the current Q-value instead of with the Q-value of the next state. Q-learning will converge as long as all state-action pairs continue to be updated. The complete Q-learning algorithm is shown in algorithm 3.

Algorithm 3 Q-Learning: An off-policy TD algorithm.

```
Initialize $Q(s,a)$ arbitrarily
**repeat**
  (for each episode):
  Initialize $s$
  **for** each step of episode **do**
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $a$, observe $r, s'$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  **end for**
**until** $s$ is terminal
```

Q-learning and its derivations are the most widely used reinforcement learning algorithms. They can learn without the need of a model and are proven to converge to an optimal policy as long as all state-action pairs are updated. Note that the target policy learned by Q-learning does not have to explore and will usually be deterministic while the policy being learned in sarsa also has to take care of exploration.

As to the question of which method is best there is no easy answer. This depends on the problem and the environment. Q-learning is usually the best choice since it learns the optimal policy regardless of the policy being followed, while sarsa learns the optimal policy that still explores. However if we want to keep exploring the environment then sarsa can give better performance.

3.4 Policy Search

All of the techniques presented above center around the concept of values, either the state-value function V^{π} or the action-value function Q^{π}. This framework is developed to facilitate learning from experience and its sophistication allows for the efficient use of available information such as information about the state space, action space and rewards. It is however not the only way in which reinforcement can be used to learn behavior through interaction. When policies were introduced in Section 2.2 we stated that

> *The policy is the core of a reinforcement learning agent because a policy is sufficient to determine an agents' behavior.*

Following this line of thought policy search can be defined as changing the policy directly (without the use of value functions) to improve behavior. Two elements are necessary, a function to represent the policy: this can be as simple as an array or as complicated as a neural network parameterized by some parameters θ, and a metric to compare different policies. Now a search in policy space can be conducted, possibly using gradient ascent methods if the policy function is differentiable. Convergence to the optimal policy cannot be guaranteed due to local minima but applications often provide good results. The main advantage is its simplicity over value based reinforcement learning. Moody calls this form of RL 'Direct reinforcement learning" [Moody and Saffel, 2001]. In most applications of RL to trading this form of RL is used.

Another approach that is considered policy search can be found in the field of evolutionary methods such as genetic programming. Here different policies are evaluated at the same time, some randomly mutated into new policies. After some period only the 'fittest' according to some performance criterion may proceed to the next iteration of this process. This is repeated until a good policy is found. While this approach can be very usable, it does not take advantage of the information available from the environment and we will not consider them here. For a good introduction [Koza, 1992] provides a general overview.

3.5 Function Approximation

One of the biggest challenge in current reinforcement learning research is to deal with large state and action spaces. Reinforcement learning suffers from the *curse of dimensionality*, this means that it becomes computationally infeasible to obtain good solutions when the dimension of the problem increases. In this section, we discuss one extension that allows for larger problems; *function approximation*. Function approximation is in essence a generalization over states.

Most textbook RL algorithms are of a tabular form, which is basically a key-value pair or an array with an entry for each state. This tabular form makes the algorithms powerful enough to work and simple enough to be analyzed for convergency properties. However when there are many states a tabular algorithm becomes prohibitive. The challenge is to find a compact representation of the state space that is independent of the number of states and that generalizes well to previously unseen states. This is what function approximation tries to do: find a compact representation of the state-space. A compact representation is usually a function of a small number of parameters, ideally the number of parameters is independent of the state space but in practice any function approximation that grows sub-exponential in the size of the state space will suffice. The parameters can be coefficients in a linear function approximator or perceptron weights in neural networks. Good generalization is achieved when states that are in each others neighborhood have similar value functions. This is sometimes also called graceful degradation.

3.6 Summary

In this chapter we formalized the problems that we study in this thesis by introducing the model of Markov decision processes. We provided some elementary solution methods both for when the complete model is known (Dynamic programming) and when a complete model is not available (RL). Finally we looked at some advanced topics such as function approximation and alternative solution methods (policy search).

Chapter
4

Options

Learning, planning and representing knowledge at multiple levels of temporal abstraction are key, longstanding challenges of artificial intelligence. In this chapter we consider how temporal abstraction can be incorporated into the mathematical framework of reinforcement learning and Markov decision processes by extending the notion of actions to include *options* – closed loop policies for taking actions over a period of time.

This extension is an example of the current frontier in reinforcement learning research but its relevance to this thesis stems mainly from the interplay that exists between options and recognizers. Recognizers help to solve some challenges encountered with options and may form a pivotal role in option discovery as well as learning with options.

We will first discuss the idea of temporal abstraction reinforcement learning in general. In section 4.2 we will focus on the prevalent framework used in reinforcement learning, the option framework. We will introduce the notation used and provide examples of options in applications. To conclude the chapter the connection between options and recognizers is treated.

4.1 Temporal Abstraction in Reinforcement Learning

One of the problems that reinforcement learning has to deal with is what is commonly known as the curse of dimensionality. The curse of dimensionality occurs because the number of parameters to be learned grows exponentially with the size of the domain, rendering efficient computational solutions infeasible. The response from the research community has been to develop new methods and techniques to tackle the larger state and action spaces required in real world domains. One approach looks at ways to exploit *temporal abstraction*. Temporal abstraction means that decisions (actions in a MDP) do not have to be specified at each time step but can be grouped together on a higher level to form temporally extended actions. The learning algorithm can now choose from these temporally extended actions. The options framework that we discuss in this chapter gives us the mathematical tools to use and learn such temporal extended actions within the reinforcement learning framework. Sometimes we will use macro actions as a synonym for temporally abstract actions.

In many task-domains different levels of action-abstraction can be discerned. In a robot navigation task actions can be described on the motor-voltage level that move the different joints of a robot or on a higher level that guide the robot to different locations, such as actions that tell the robot to go to some coordinates. Now if all decisions are made on the motor-voltage level its should be clear that the state and action space will be very large. If temporal abstraction is introduced, it becomes possible to lump a set of actions together and form macro actions such as MOVE FORWARD and TURN AROUND. The task can now be solved using these macro actions, thereby greatly reducing the action space because there are less actions to choose from. Of course the macro actions themselves should also be learned but because they have to learn a specific task it is easier to solve. The resulting approach employs a *divide & conquer* approach where at each level the task is split into smaller sub-tasks until the individual sub-tasks consists of only 'atomic' actions i.e. the task cannot be split further.

There are currently three competing frameworks that have been introduced in recent years. These are: the *options* formalism introduced by Sutton, Precup and Singh [Sutton et al., 1999], the *hierarchies of abstract machines* (HAMs) approach of Parr and Russell [Parr and Russel, 1998] and the MAXQ framework of Dietterich [Dietterich, 1998]. The approaches differ in the assumptions made and the scope of their applicability but all are grounded in the theory of semi-Markov decision processes [Puterman, 1994] that provides the formal basis. In this thesis we will focus on the options framework as it is the most general and natural framework for temporal abstraction. It is possible to reformulate the other approaches in terms of the option framework.

4.2 Options

The options framework [Sutton et al., 1999] was developed to address the longstanding problem in AI of learning, planning and representing knowledge at multiple levels of temporal abstraction. The options framework is based on the same mathematical foundations that underpin 'flat' reinforcement learning, Markov decision processes. Options are conceptual generalizations of primitive actions, in the sense that options can extend over time. Put differently an option is a temporally extended course of action. Simply put an option can take multiple time-steps to complete. In this light primitive actions are a special case of options, where the option is only one time step long.

The simplest kind of options consist of three components: a policy $\pi : S \times A \rightarrow [0,1]$, a termination condition $\beta : S^+ \rightarrow [0,1]$, and an initiation set $I \subseteq S$. An option $\langle I, \pi, \beta \rangle$ is available in state s_t if and only if $s_t \in I$. If the option is executed, then actions are selected according to π until the option terminates stochastically according to β. An option can be considered a closed loop policy defined over a subset of the original state space.

The initiation set and termination condition of an option together restrict its range of application in a potentially useful way. In particular, they limit the range over which the optionÕs policy needs to be defined. For example, a handcrafted policy π for a mobile robot to dock with its battery charger might be defined only for states I in which the battery charger is within sight. The termination condition β could be defined to be 1 outside of I and when the robot is successfully docked.

Given a set of options, their initiation sets implicitly define a set of available options O_s for each state $s \in S$. These O_s are much like the sets of available actions, A_s. We can unify these two kinds of sets by noting that actions can be considered a special case of options. Each action a corresponds to an option that is available whenever a is available ($I = s : a \in A_s$), that always lasts exactly one step ($\beta(s) = 1, \forall s \in S$), and that selects a everywhere ($\pi(s,a) = 1, \forall s \in I$). Thus, we can consider the agentÕs choice at each time to be entirely among options, some of which persist for a single time step, others which are more temporally extended. The former we refer to as single-step or primitive options and the latter as multi-step options. Just as in the case of actions, it is convenient to notationally suppress the differences in available options across states. We let $O = \cup_{s \in S} O_s$ denote the set of all available options.

The definition of options is crafted to make them as much like actions as possible, except temporally extended. Because options terminate in a well defined way, we can consider sequences of them in much the same way as we consider sequences of actions. We can also consider policies that select options instead of primitive actions, and we can model the consequences of selecting an option much as we model the results of an action.

Options can be used as building blocks for planning and learning, for example by constructing a policy over options instead of primitive actions. Since options are extended over time there are usually fewer options and this translates in a reduced action space. Options also provide a natural way to learn the policies of subgoals that are the result of a decomposition of a complex task. In this scenario the often infeasible task of learning the complete task is broken down into learning separate simpler subtasks that may even be learned in parallel. Figure 4.1 gives an example of a possible task decomposition in a taxi domain, where we see that navigating the taxi is independent of picking up and dropping of passengers. Options for navigation and for passenger pick up can thus be combined to solve the overall task

We can also find support for the concept of options from a psychological standpoint. Options seem an attractive concept because humans employ temporal abstraction all the time when making complex decisions. For example, when planning a trip through Europe we tend to see the activity of driving from one city to another as one action while in reality it consists of many actions. By learning options for common tasks and reusing them we can solve complex problems.

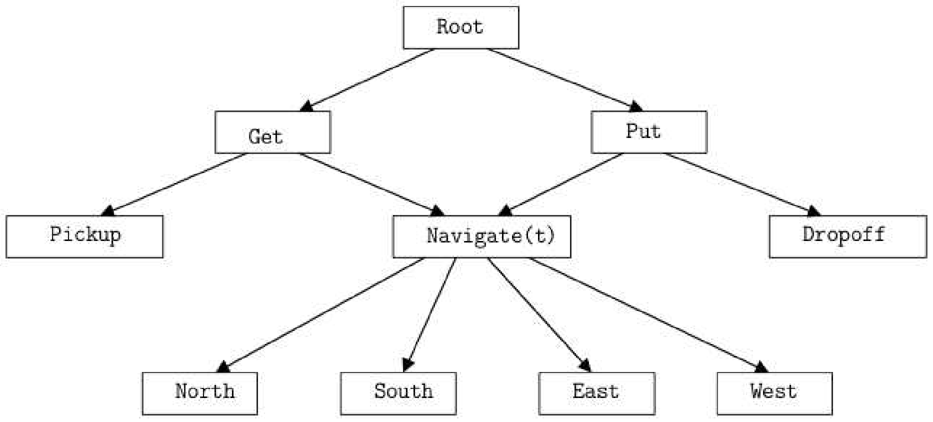

Figure 4.1: *Task Graph for a Taxi agent (after Dietterrich [Dietterich, 1998]).*

4.3 Open Problems

Some questions are still open to research in hierarchical reinforcement learning. One of the main challenges is how to find and learn options, i.e. how can we develop a system that discovers options when we start with only primitive actions. The current state-of-the-art uses prior knowledge to handcraft options in specific domains. To realistically apply temporal abstraction in across many different real world tasks, a dynamic option learning system is required. This is a very important but also very difficult question to answer.

Another useful avenue of research is looking into ways of learning multiple options from one stream of experience. By . This closely resembles the way how humans learn tasks and would greatly speed up learning time and generalization since information would effectively be reused. Finally the interplay between hierarchical methods and function approximation techniques is being investigated as a combination of these two techniques will be necessary to scale up the problems that can be studied with reinforcement learning in general.

4.4 Options and Recognizers

Our work with recognizers addresses two of the open problems in option research. First the algorithm that will be introduced in 7 can serve as an important module for discovering options as it can take care of learning the policy of an option as well as defining its initiation set and termination condition. Secondly as we will see in the next chapter recognizers enable off-policy learning so that an agent can learn about multiple options from only a single stream of experience.

4.5 Summary

Options are a powerful concept that make it possible to generalize the concept of actions. By using options decisions can be made on a higher level and the complexity per decision decreases. This results in both a speed up of learning as well as enabling efficient solutions for larger state and action spaces. Recognizers can form an integral part in the discovery and specification of options.

Chapter 5 Recognizers

In this chapter we introduce and look at recognizers, the main reinforcement learning concept that is studied in this thesis. First recognizers will be formally defined and related to the reinforcement learning techniques presented in chapters 2, 3 and 4. Following this the various uses of recognizers and their advantages compared to other methods will be explained. The current methods of acquiring recognizers and the problems that we face with this will also be featured in this chapter. Our proposed solution to these problems will be treated in chapter 7.

This chapter is organized as follows: in section 5.1 recognizers are introduced in an intuitive way. A more formal introduction is given in section 5.2. An important aspect of recognizers, composition, is treated in section 5.3. The current method of acquiring recognizers and its drawbacks are discussed in section 5.4. Finally we conclude the chapter with a summary of the covered material in section 5.5.

5.1 Introduction

What are recognizers? This question is central to this thesis and we will try to find an answer in this chapter. There are several ways to look at recognizers varying from how we define them to how we use them. To frame the discussion we will look at three aspects of recognizers. The sequence of these three is loosely ordered from most intuitive to most formal. The three aspects are:

- Intuitive notion of recognizers.
- Usage of recognizers in the RL framework.
- Definition of recognizers within the RL framework.

We will discuss the first two aspects in the remainder of this section. The third aspect, a definition of recognizers within the RL framework is the subject of the next section.

In general recognizers serve to restrict the space of possible solutions to a problem by excising uninteresting parts and directions. Recognizers form a filter on the stream of experience that usually floods the sensors, thereby allowing the agent to focus only on relevant data. Agents that use recognizers can work more efficiently since they can consider a smaller problem space than without the help of recognizers. In this general way recognizers work as templates. They provide the general direction of the solution but defer final computation of such a solution to the learning agent. By restricting the space of possible solutions, recognizers define a subspace of the problem. The idea is that this subspace still contains all relevant solutions. By reducing the problem space recognizers make effective solutions possible for problems that are otherwise too large to solve. It is in this way that recognizers provide answers for the curse of dimensionality.

For example a recognizer function to travel from San Francisco to Vancouver could recognize (and learn from) different modes of transport but would ignore directions that would take it south, west or east because they will in all likelihood not lead to its goal (Vancouver lies directly north of San Francisco).

When we apply this idea to the framework of reinforcement learning it becomes clear that recognizers should put restrictions on the space of policies, the solutions. Recognizers act as a filter on actions and restrict the policy space available to the learning system. This results in a speed up in learning since only a subset is considered by the learning algorithm.

Recognizers are primarily useful for off-policy learning since they can learn a target policy by selectively tuning in to the relevant parts of the behavior policy while tuning out irrelevant parts. Because recognizers still follow the behavior policy in the relevant parts, the off-policy updates have a low variance. The recognizer function specifies what part of the state action space is relevant and what part is not.

This aspect enables recognizers to learn from off-line data or to learn from data generated by an unknown behavior policy. By filtering the actions that are recognized (and therefore updated) it will always learn about its target policy, even when the policy actually generating the experience is completely different[1]

Recognizers are general enough to allow for different courses of actions as long as they are connected to the task that is being recognized. This makes recognizers very flexible in dynamic environments. The trade-off between specialization (restricting) and generalizing can be calibrated by the domain.

[1]To learn about the complete target policy however we must demand that the behavior policy visits the state-action pairs with a non zero probability.

5.2 Definition

A recognizer is defined as a function that assigns a recognition probability to each state and action pair:

$$c : S \times A \to [0,1],$$

where $c(s,a)$ indicates to what extent the recognizer allows action a in state s. While in general it is more powerful to see this function as a probability measure, in this thesis we only consider binary recognizers to enable a detailed analysis of the theoretical properties. In this special case, c becomes an indicator function, specifying a subset of actions that are allowed, or recognized, in a particular state.

The recognizer function alone is not sufficient to specify a target policy and perform off-policy learning, it needs a behavior policy that selects actions. A recognizer c together with a behavior policy b form a target policy π, where:

$$\pi(s,a) = \frac{b(s,a)c(s,a)}{\sum_x b(s,x)c(s,x)} = \frac{b(s,a)c(s,a)}{\mu(s)} \tag{5.1}$$

The denominator, $\mu(s) = \sum_x b(s,x)c(s,x)$, is the recognition probability at state s and can be estimated from the data as in [Precup et al., 2005]. The policy π is only defined at states for which $\mu(s) > 0$.

To aid the analysis of the algorithm we introduce the concept of an induced MDP:

Definition 5.2.1. *An Induced Markov Decision process (IMDP) I is a MDP defined by a 4-tuple (S,A,P,R) that is formed by combining an underlying ordinary MDP M and a policy π, where S is a set of states from M that are reachable from π, A is a set of actions from M where $\pi(s,a) > 0$ and P and R are equal to their counterparts in M. The induced MDP I is a subset of the underlying MDP M.*

An induced MDP can be seen as the union of an ordinary MDP with a policy π. If the policy π is defined for all state action pairs as is the case for ϵ-soft policies, the induced MDP is equal to the underlying MDP.

The induced MDP forms the input to our algorithm. Note that the dependancy on the target policy only requires that the policy must visit state action pairs that we wish to learn from without imposing any requirements on the frequency or order of these visits. Since it cannot be expected to learn about 'experience' that can never occur this restriction is quite reasonable.

5.3 Composing Recognizers

Recognizers can be combined and aggregated whereby the combined recognizer is the set union of the constituting recognizers. This characteristic is particularly useful since it allows specific recognizers to be learned for subproblems that can easily be combined to form an overall solution. Another property of recognizers is their portability across different instances in the same domain. Since recognizers are a generalization of a policy, they can accommodate multiple ways of behaving (policies) and effectively act as templates that can be used to quickly solve many instances of the same problem type.

5.4 Recognizer Acquisition

In the current research the recognizer function is either specified by experts or the existence of an oracle is presumed that will supply the correct values. An approach that learns a recognizer function from data or experience has never been developed. We are of the opinion that the availability of a learning algorithm next to the expert option is essential in making recognizers usable beyond a few test problems in the laboratory. A solution that can acquire the structure of the recognizer function without human intervention advances both the idea of creating intelligent systems that can learn autonomously and it will make the specification of a recognizer function faster, cheaper and more efficient.

Our intention is not to replace the current practice of specifying recognizer functions by hand by experts in total, we propose our solution as a viable alternative that can even work in conjunction with the existing method. We do believe that knowledge provided by experts is very costly to acquire, even harder to format in the right structure and prone to human error.

A successful hybrid approach can start with a rough estimate provided by human experts that is gradually refined by our proposes algorithm. Alternatively we can imagine that the output of our algorithm may be pruned and revised by human experts to arrive at the best possible recognizer function.

5.5 Summary

This chapter deals with one of the proposed solutions for the curse of dimensionality, recognizers. Research on the characteristics and properties of recognizers is currently a very active area. We defined what recognizers are and how they work. So far the recognizer function has been hand specified or assumed as prior knowledge. In this thesis we present the first results on learning the recognizer function when only experience in the form of sample transitions of a Markov decision process is available. The learned recognizer functions can then be used to perform off-policy learning for reinforcement learning systems.

Chapter 6
Related Work

In this chapter we discuss other research initiatives that deal with the same questions. The focus is on research within the artificial intelligence and machine learning community but also research from other fields such as psychology is treated when applicable.

Directly related work is not available since [Precup et al., 2005] is the first and currently only publication on recognizers and. There is some research within reinforcement learning and dynamic programming that revolves around the same ideas, *action elimination*. We discuss this in section 6.1. Work on action schemas, while strictly belonging to psychological research, shares some elements of our research. It is discussed in section 6.2. We also look at a well known theory developed by Schank called the scripts theory.

6.1 Action Elimination

Action elimination (AE) has been around since the late 60s [MacQueen, 1966] in the context of MDPs and dynamic programming. Action elimination identifies suboptimal actions and eliminates them from the MDP. AE serves two purposes:

1. Reducing the set of actions to be searched at each iteration, since non-optimal actions cannot be part of the optimal solution.

2. Identifying the optimal solution in models with an unique optimal policy. After finitely many iterations AE will have eliminated all but the optimal actions.

AE provides a great increase in efficiency since less actions have to be evaluated at each iteration, effectively diminishing the size of the problem. In dynamic programming the model and its parameters are assumed to be known but for reinforcement learning we cannot assume this knowledge which made AE initially not suitable for reinforcement learning. AE has been extended for reinforcement learning by [Even-Dar et al., 2003] and a model free algorithm is provided in the paper. The authors use AE to limit the number of samples needed from the environment by eliminating suboptimal actions early on. Our proposed algorithm shares the common idea of eliminating actions but the goals of AE and our work are not the same. Specifically, our work is concerned with finding recognizers, filters on actions that retain the salient features of a task that we wish to recognize, usually given in the form of an option. The resulting recognizer will then be used to perform off-policy learning of this task and can be reused across domains and instances. AE is more classical in the sense that its goal is to find an optimal policy for a given problem instead of a recipe to solve many such problems. AE allows an agent to converge faster to an optimal policy by eliminating actions.

6.2 Action Schemas

The work by [Cohen et al., 2007] on learning action schemas, while more abstract shares some elements that can be seen in our work. In the paper Cohen presents an agent called Jean, that uses actions schemas, structures that are similar to options. The action schemas consists of three components:

- controllers,
- maps,
- decision regions.

Controllers specify the agents behavior and closely resemble our concept of actions. The equivalent of recognizers in Cohen's model are maps. Like recognizers maps are associated with different tasks and specify a region of behavior that accomplishes these tasks. decision regions do not have a simple correspondence with reinforcement learning concepts. Maps are also learned from experience by sample transactions. The work of Cohen differs from our work in its scope and methods. First of all action schemas incorporate the action selection (policy) part whereas recognizers are independent of the policy followed. Maps are learned by adding sample trajectories and averaging them while recognizers are learned by eliminating undesired policies. Finally Cohen's work sets out to study a psychological model of learning, Piaget's theory, while our aim is to apply recognizers to solve problems.

6.3 Scripts

Schank and Abelson [Schank and Abelson, 1977] introduced a theory of knowledge representation claiming that people primarily learn not from rigid rules or structures but from specific past experiences called cases. Similar cases are refined, generalized and distilled into some sort of prototype case which they call *scripts*. So a script is some kind of template that provides us with the right actions if we encounter a similar situation. Scripts and their causal structure also allows us to understand knowledge by weaving a 'story' from scripts. The role of goals and plans in Schank's theorem is to relate the different scripts and form expectations about scripts. According to Schank scripts are stored in our memory and specific experiences trigger the recollection of it and influence our behavior accordingly i.e. we respond according to the appropriate script. This approach gave rise to *case based reasoning* systems. The key to understanding knowledge here is the way in which people structure past experiences into appropriate scripts and use scripts to explain behavior.

The classic example from Schank's book *Scripts, Plans, Goals and Understanding: an Inquiry into Human Knowledge Structures* [Schank and Abelson, 1977] is the restaurant script.

Our approach has some similarities with Schank's conceptual dependency framework. Experience plays an important role in both approaches but it is clear that Schank's theory is conceptually more abstract and allows for instance the synthesis of series of connected events into scripts. The concept of a script as a sort of template, one that gives us a default behavior for situations that can be adapted to suit specific instances, is somewhat similar to our stated goal for recognizers, namely to provide a general direction to solve various related problems. The main difference is that Schank's theory allows for the explicit representation of knowledge via scripts stored in memory, whereas our representation of knowledge is far more implicit and resides mainly in the values assigned to state-action pairs.

6.4 Summary

This chapter provides some background information on various subjects that are related to the work in this thesis. We highlighted three approaches, one within reinforcement learning (action elimination), one approach that deals with the same issues in a broader sense (action schemas) and an alternative approach. We provided an overview of the most important aspects and demonstrated how these approaches relate to our work. We also signaled the differences of the approaches and our work.

Part II
Model and algorithm

“A learning experience is one of those things that says, ’You know that thing you just did? Don’t do that.’ ”
- Douglas Adamsr.

Chapter 7 Proposed Algorithm

In this chapter we present our solution to solve the problem of acquiring recognizers. We propose a learning algorithm that can learn a recognizer from experience. The experience in this context is generated through interaction with a MDP.

In section 7.1 an intuitive overview of the algorithm is presented in order to understand the basic concepts. Section 7.2 discusses the necessary conditions that the algorithm requires. The formal algorithm is presented in section 7.3. An analysis and proof of the algorithm is reserved for the next chapter. In section 7.4 some extensions and variations of the basic algorithm are presented.

This chapter forms the main contribution to the body of research and consists of original research. The problem as specified in the thesis outline is to develop methods to learn the recognizer function on the basis of experience. Our proposed solution consists of an algorithm that is able to do exactly that: learn a recognizer function from experience. Our algorithm is the first departure from design by human experts for recognizers.

7.1 Introduction

Any algorithm is in essence a procedure or prescription comprised of simple steps that solves a problem or accomplishes a task. In the context of this thesis the problem we wish to solve is finding a recognizer function without the use of human intervention. An algorithm is then a good solution since the steps can be processed by any computing device. Chapter 5 defined what recognizers are exactly, we repeat it here for convenience.

Recognizers are filters on actions that pick out only those actions from a stream of experience that are useful for the task associated with the recognizer. Mathematically a recognizer is defined as a function

$$c : S \times A \to [0,1].$$

A recognizer c together with a behavior policy b forms a target policy π , where:

$$\pi(s,a) = \frac{b(s,a)c(s,a)}{\sum_x b(s,x)c(s,x)} = \frac{b(s,a)c(s,a)}{\mu(s)}, \tag{7.1}$$

the target policy $\pi(s,a)$ represents the policy we wish to learn and $b(s,a)$ represents the policy that generates the stream of experience. Note that if $c(s,a) = 1,\ \forall s,a$, then π is the same as b. As stated in chapter 5 we will only consider binary recognizers, that is, recognizers that can be either 1 or 0 for every state-action pair.

Now what we want the algorithm to do is to find a recognizer function c that assigns either 1 or 0 to every state-action pair in order to define a target policy π using only experience generated from an (external) behavior policy b.

On a high level the algorithm accomplishes this as follows:

1. Start by recognizing all actions: $c(s,a) = 1 \quad \forall s,a$.
2. Use experience generated by b to decide which actions do not have to be recognized.
3. Repeat step 2 until termination.

At all times during the execution of the algorithm, the recognizer function is legal in the sense that it satisfies the definition found in chapter 5. What is happening is that under influence of experience generated by the behavior policy b more insight is gained as to which actions should be recognized and which actions should not. By dropping the undesired actions we slim down the recognizer function from its starting pint where all actions are recognized until only the best actions are recognized. To discriminate between desired and undesired actions we use a sort of value function as referenced in section 2.2. We extend the value function however because we use sample transitions to estimate the values of state action pairs. The use of samples introduces uncertainty in the estimated value function. This in turn could lead to incorrect restrictions of good actions. To counter this we employ *Interval Estimation* on the estimates to minimize the probability of throwing out the wrong actions. Interval estimation basically replaces the single estimate of the value (usually the mean) with an interval wherein the value must lie (usually one standard deviation around the mean). The idea of interval estimation is similar to its use by [Strehl and Littman, 2005], however here it is not used to formulate an exploration strategy but to restrict actions. We will see in chapter 8 that we can make the probability of dropping the wrong actions arbitrarily small using interval estimation. In order to work with intervals we must define and maintain upper- and lowerbounds on the state action values and on the state values.

We can think of our algorithm as a net that is thrown over all possible (s,a) pairs, so that at the start all (s,a) pairs are 'caught' in the net. After assimilating experience over time, the algorithm gains more understanding of the value of the different state action pairs and will slowly tighten its net around the high valued pairs and dropping the lower value pairs until the net only captures the optimal state action pairs. At each iteration we reduce the problemspace so that fewer options remain. A conceptual visualization of the algorithm is given in figure 7.1. It shows the scope of the recognized state-action pairs over time and visualizes the convergence of the algorithm in a funnel like manner towards the optimal (s,a) pairs.

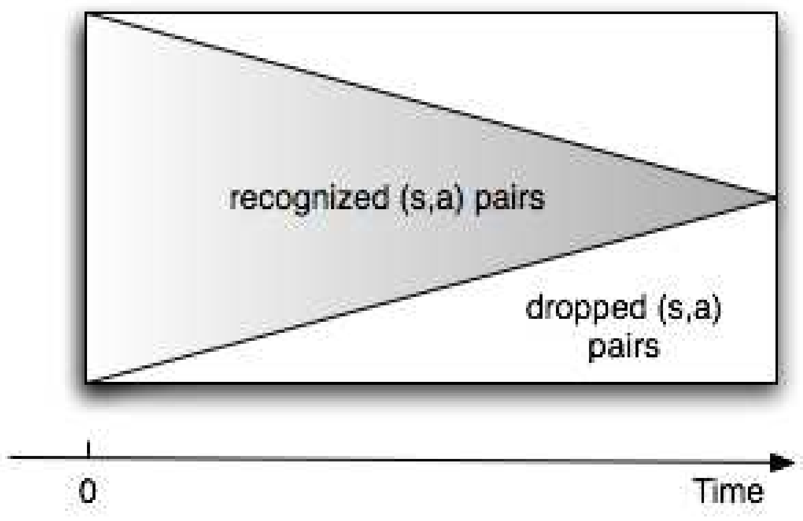

Figure 7.1: *A conceptual visualization of the proposed learning algorithm.*

The ideas behind this algorithm can be traced back to the ideas of action elimination found in the classical dynamic programming literature as found in MacQueen [MacQueen, 1966] and Puterman [Puterman, 1994] and also discussed in section 6.1 of chapter 6.

7.2 Pre-requisites

Every algorithm requires some externally supplied elements. Our learning algorithm has to have two elements available:

1. An (Induced) MDP.
2. A behavior policy.

These two elements together generate the experience or rather the stream of sample transitions and rewards that the algorithm will use to learn a recognizer. We do not put any constraints on the behavior policy and it specific working can be unknown to the algorithm. The algorithm only relies on the behavior policy to select actions that it will learn from.

The above requirements describe the on-line version of the algorithm where actions are selected at each iteration in real-time. Alternatively we can perform the algorithm with previously generated, off-line data. In this case a history of experience is taken. Learning takes place in the same manner as the on-line version.

In addition to the above requirements, there is a subtle an implicit demand made on the structure of the reward function. The reward function in a MDP is of the form $r(s, a)$ and specifies a reward for each state and action pair (see also 2.2). We wish to learn recognizers that are useful at recognizing a specific task, restricting actions that do not contribute to this task. Since the algorithm uses value functions to bas its restriction decision on and since values are derived from rewards, the algorithm ultimately relies on the reward function to guide its decisions. As rewards are defined as a numerical feedback that indicates the relative usefulness of behavior this is expected.

Definition 7.2.1. *The state action value is denoted by Q(s,a) and the state value with V(s). The upper confidence bound for Q(s,a) is denoted by* $\overline{Q}(s,a)$ *and for V(s) with* $\overline{V}(s)$*. Analogously the lower confidence bound for Q(s,a) is denoted by* $\underline{Q}(s,a)$ *and for V(s) with* $\underline{V}(s)$*.*

7.3 Algorithm

In this section we formally define the algorithm and look at the different steps that make up the algorithm. An overview of the algorithm is given in Algorithm 4 on page 57.

In essence the algorithm chips away at the induced MDP until a suitable recognizer function is learned. The metaphor of a sculptor chipping away at a piece of marble is striking since this algorithm also constructs by leaving out, in our context, actions. The resulting recognizer can then be used for off-policy temporal difference learning. There are three main parts of the algorithm. These are:

1. Initialization.
2. Learning.
3. Termination.

In the following subsections each part is described in detail.

7.3.1 Initialization

The initialization phase is used to construct a starting point that is in itself already a solution for the problem albeit a very suboptimal one. We initialize a trivial recognizer function, namely the recognizer function that recognizes all actions:

$$c(s, a) = 1 \quad \forall s, a.$$

We also have to initialize our estimates for the value function. More specifically we have to initialize our estimates for the upper and lower bounds of the interval. This is somewhat problematic since we have do not have any information about the environment at this stage. A naive but perfectly legal way of providing initial values is by using $-\infty$ and ∞ for these values. After seeing enough samples the upper and lower bounds would eventually converge to the correct value but this can take very long. Could we improve on these bounds without any additional experience? We submit that this is possible. By making use of the fact that the rewards themselves are bounded we can develop an upper and lower bound on the values. Recall that in an infinite-horizon discounted model the value is given by:

$$\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

(See also equation 2.2 in section 2.2.)

If we know that any individual reward r is bounded by R_{max} we can construct V_{max} by $V_{max} = \frac{R_{max}}{1-\gamma}$. Now this new upper and lower bound of the value estimates can be used:

$$\overline{Q} = V_{max} = \frac{r_{max}}{\gamma}.$$

Analogously for the $\underline{Q}$ values we can define a similar bound:

$$\underline{Q} = -V_{max}.$$

The values above provide the tightest bounds possible without any prior knowledge. When such knowledge is easily available however, for instance when we know that all rewards are positive, the incorporation of this information will further increase efficiency.

7.3.2 Learning

In the learning phase we iteratively throw out undesired actions on the basis of experience generated by the behavior policy. This throwing out of undesired actions is called the *restriction* step and effectively reduces the size of the recognizer and the size of the induced MDP.

It is important to precisely define what an undesired action is in this context. Intuitively we would like a recognizer to rank the actions for each state according to their *value* in solving the specific task(s) that we want to recognize. It is evident that we should compute some form of value function to realize this. Our approach tries to estimate the value of each state action pair on the basis of experience generated by the behavior policy much like Q-learning [Watkins, 1989]. We use these estimates to rank the different action pairs per state.

With enough experience we can divide the state-action space into desired actions with a high state-action value and undesired actions with low state-action values. As stated in 7.2 the algorithm uses the bounds on the interval rather than a point estimate to restrict actions. This process will greatly improve accuracy at the cost of a loss in efficiency.

To update the upper- and lowerbounds of the state action values we use a TD style update rule:

$$\overline{Q}(s,a) = (1-\alpha)\overline{Q}(s,a) + \alpha(\gamma(r(s,a) + \alpha\overline{V}(s') + \beta(\#(s,a)))), \tag{7.2}$$

where s' is the state reached from state s after performing action a and $\overline{V}^t(s) = max_\alpha \overline{Q}^t(s,a)$.

Replacing $\overline{Q}$ and $\overline{V}$ with $\underline{Q}$ and $\underline{V}$ respectively in Equation 7.2 and subtracting $\beta(\#(s,a))$ yields the corresponding formula for the lowerbound:

$$\underline{Q}(s,a) = (1-\alpha)\underline{Q}(s,a) + \alpha(\gamma(r(s,a) + \alpha\underline{V}(s') - \beta(\#(s,a))). \tag{7.3}$$

The $\beta(\#(s,a))$ used above is a function of the number of times a state-action pair is visited and must slowly diminish to 0. Its function here is to add an extra weight to the estimate further ensure that the estimate encapsulates the optimal value i.e.: $\overline{Q}(s,a) > Q^*(s,a)$.
In this thesis we follow [Even-Dar et al., 2003] and define β as:

$$\beta(\#(s,a)) = c\sqrt{\frac{ln(\#(s,a)}{\#(s,a)}}V_{max},$$

where $c = 4$ (as in [Even-Dar et al., 2003]) and $\#(s,a)$ denotes the number of times we visited state-action pair (s,a). We can see that this function goes to zero as $\#(s,a)$ goes to ∞ since $\lim_{x\to\infty}\frac{ln(x)}{x} = 0$. The significance of β is mainly theoretical since it guarantees convergence, in practice we can drop the β term if we know the value bounds. In chapter 8 we will use this term to arrive at a theoretical convergence result.

7.3.3 Termination

The algorithm ultimately terminates when the recognizer function is reduced to the optimal policy, i.e. no more suboptimal actions can be thrown out. We will see in chapter 8 that this policy is indeed equal to the optimal policy for the induced MDP with high probability. We can quantify the point of termination with:

$$\forall s \in S, \forall a \in U(s)\ |\overline{Q}(s,a) - \underline{Q}(s,a)| < \epsilon,$$

where $U(s)$ is the set of all actions that are still recognized: $U(s) = \{a|\overline{Q}(s,a) \geq \underline{V}(s)\}$. However for a recognizer to be able to generalize well it is preferable to terminate the algorithm before this time. To leverage another strong theoretical point of recognizers, their portability, it is also advisable to not overfit it. When to terminate the algorithm is a question not easily answered, because technically the algorithm can be stopped after any number of iterations (even 0) and it will still output a legal recognizer function. The trade off that must be made is one between generality (recognize as much actions as possible) and efficiency/speed (recognize as little actions as possible) This trade-off can even be dictated by resource constraints such as available data, computing time or memory allocation. In practice the available data is almost always the bottleneck. The complete Algorithm is found as Algorithm 4.

7.4 Variations

The algorithm presented above contains the basic elements to make recognizers learnable from experience and its specific structure is kept as simple as possible to make it

Algorithm 4 Recognizer

```
Input: MDP M, recognizer c, policy b
Initialize: $c(s,a) = 1 \ \forall s,a$
For every state action pair (s,a):
$\overline{Q}(s,a) = \quad V_{max}$
$\underline{Q}(s,a) = -V_{max}$
repeat
  for each transition (s,a,r,s') do
    update $\overline{Q}(s,a)$ and $\underline{Q}(s,a)$
    if $\overline{Q}(s,a) < \underline{V}(s,a)$ then
      c(s,a) = 0
    end if
  end for
until $\forall s \in S, \forall a \in U(s) \ |\overline{Q}(s,a) - \underline{Q}(s,a)| < \epsilon$
where $U(s) = \{a|\overline{Q}(s,a) \geq \underline{V}(s)\}$
```

easier to provide theoretical guarantees. There is certainly room to improve the performance of this algorithm. Some of these improvements have already been mentioned. The use of sharper upper and lower bounds for the initialization of the state-action values is an example. We can also apply this principle to the initial values of the recognizer function by discarding actions that are clearly not optimal or can never be reached in the first place. The improvement will be faster convergence since we start further down the road but the trade off is that we must now rely on prior knowledge, knowledge that is not always available. We essentially trade away universal applicability for improved performance in the target domain. When we want to develop an algorithm that is as universal as possible, as we do in this thesis, this does not seem like a good option. If however we are only interested in solutions for a specific domain then this is perfectly arguable.

Another element of the algorithm that can be altered to improve performance is the restriction step. In the current version an action is discarded if the upper limit of its value is lower than the lowest limit of the values of the other actions available in that state. While this procedure gives us strong guarantees on the optimality of the algorithm it can be unnecessary restrictive for practical purposes. Alternative setups could use a notion of an average value to test against or even a fixed value. This will allow the algorithm to converge much faster but the trade-off is the loss of guaranteed optimality. Ultimately the application and the system designer must decide how much guarantee it needs.

As we can see there are many elements of the algorithm that can be tailored to suit the individual needs of different domains and designers. All of these variations however rely on the basic mechanism of reinforcement learning updates, acquired through interaction with the environment, to learn the recognizer function.

7.5 Summary

This chapter presented the learning algorithm that we developed in order to learn recognizers from experience. The algorithm presented in this chapter for the first time breaks the reliance on human experts in acquiring the recognizer function. We presented the general idea, the pre-requisites and the step-by-step workings of the learning algorithm. We finally discussed some possible variations on the main algorithm.

Chapter 8 Proof

Before the proposed algorithm can be used in practice a thorough analysis of the theoretical properties and convergence assurances is required to provide a guaranteed level of performance which can be measured and verified in practice. Without such an analysis our approach cannot be justified as scientific in nature since it would be a claim supported only by empirical evidence, which have only the power to falsify claims, not corroborate them.

In this chapter we will provide a formal proof of the soundness of the algorithm presented in Chapter 7. We have to define which criteria we will use and look at the different metrics employed. The convergence guarantees will follow the Probably Approximate Correct (PAC) framework introduced by [Valiant, 1984]. A complete introduction to the PAC framework falls outside the scope of this work. For a good reference work see [Kearns and Vazirani, 1994]. Finally the implications of the proof will be discussed before moving on to the experimental results. A summary can be found in section 8.3.

8.1 Proof Formulation

We wish to provide theoretical guarantees that the algorithm converges to the optimal recognizer. As we have seen from 7.3.3 it is very difficult to define what an optimal recognizer looks like, since this depends on the specific task and goals. We can however look at the optimal policy π^* as a proxy for the optimal recognizer since a recognizer can never do better than the optimal policy in terms of performance (accumulated reward). In reality we contend ourselves with a slightly lower performance compared to the optimal policy because we gain generality. For the proof this is however irrelevant.

Now we can prove that the algorithm presented in chapter 7 is optimal by establishing two facts:

1. The recognizer always contains the optimal solution.
2. The algorithm converges with high probability to the optimal solution.

The following two propositions establish the fact that the algorithm will ultimately converge to the optimal policy π^* of the induced MDP.

Proposition 8.1.1. *If every state-action pair is performed infinitely often then the upper (lower) estimation process, $\overline{Q}^t_\delta(\underline{Q}^t_\delta)$, converges to Q^* with probability one.*

Proof of Proposition 8.1.1
In order to show the almost sure convergence of the upper and lower estimations, we consider a general type of iterative stochastic algorithms, which is performed as follows:

$$X_{t+1}(i) = (1 - \alpha_t(i))X_t(i) + \alpha_t(i)((HX_t)(i) + w_t(i) + u_t(i)), \tag{8.1}$$

where w_t is a bounded random variable with zero expectation and each H is a pseudo contraction mapping. A contraction mapping is a function that shortens the distance on each application. An elaborated proof for Q-learning following the same procedure can be found in [Bertsekas and Tsitsiklis, 1996].

Definition
An iterative stochastic algorithm converges if:

1. The step size $\alpha_t(i)$ satisfies (1) $\sum_{t=0}^{\infty} \alpha_t(i) = \infty$, (2) $\sum_{t=0}^{\infty} \alpha_t^2(i) = 0$ and (3) $\alpha_t(i) \in (0, 1)$.
2. There exists a constant A that bounds $w_t(i)$ for any history F_t , i.e., $\forall t, i : |w_t(i)| \leq A$.
3. There exists $\gamma \in [0, 1)$ and a vector X^* such that for any X we have $||HX - X^*|| \leq \gamma||X - X^*||$, where $|| \cdot ||$ is any norm.
4. There exists a nonnegative random sequence θ_t, that converges to zero with probability 1, and is such that

$$\forall i, t \quad |u_t(i)| \leq \theta_t(||X_t|| + 1). \tag{8.2}$$

We first note that the Q-learning algorithm from section 3.3.2 satisfies the first three criteria and the fourth criteria holds trivially for Q-learning since $u_t = 0$, thus its convergence is established. We now make use of the fact that our upper and lower Q estimation updates are very similar to Q-learning. We will provide the proof for the upper estimate, the proof for the lower estimate follows analogously. The upper estimate of our algorithm has an additional noise term, u_t . If we show that it satisfies the fourth requirement, then the convergence of our algorithm is also proved.

Proposition 8.1.2. *The upper estimation algorithm converges.*

Proof:
In the convergence proof of Q-learning, it was shown that requirements 1-3 are satisfied, this implies that the upper estimates satisfies them as well. Our algorithm has an additional noise term, $\beta(\#(s,a))$, which was introduced in section 7.3.2. Now we let $u_t = \theta_t = \beta(\#(s,a)) = c\sqrt{\frac{ln(\#(s,a,t)}{\#(s,a,t)}}V_{max}$. Since $\beta(\#(s,a))$ is always positive we do not need the absolute brackets in equation 8.2 anymore. Now θ_t clearly converges to zero because $\frac{ln(x)}{x}$ goes to zero with probability 1 when x goes to infinity and since we let $u_t = \theta_t$, equation 8.2 holds:

$$|u_t(i)| = \theta_t \leq \theta_t(||X_t|| + 1).$$

Similar result holds for the lower estimate as well.

Now we have to show that the optimal policy will be recognized by any recognizer resulting from our algorithm. To do this we make use of a proposition from Evan-Dar that bounds the optimal value between the interval spanned by the upper and lower estimates:

Proposition 8.1.3. *[Even-Dar et al., 2003] For every state-action pair s, a and time t with probability at least* $1 - \delta$ *we have that*

$$\overline{Q}^t_\delta(s) \geq Q^*(s) \geq \underline{Q}^t_\delta(s).$$

Proof:
For a proof of this proposition see [Even-Dar et al., 2003].

Proposition 8.1.4. *At any point the recognizer c contains the optimal policy with probability at least* $1 - \delta$

Proof:
We use a proof by induction on the discrete time-step t to establish that at any time during the algorithm the recognizer contains the optimal policy with probability at least $1 - \delta$. In other words we have to prove the relationship that if the recognizer contains the optimal policy at time t it must also hold for time $t + 1$ with high probability. In addition to this we have to establish that the recognizer contains the optimal policy for t=0, so that for every time-step t:

$$c(s,a) = 1 \quad \forall s,a \in \pi^*(s,a). \tag{8.3}$$

Upon initialization ($t = 0$) this is trivially true since we initialize with: $c(s,a) = 1 \quad \forall s,a$.
Let's assume that the recognizer contains the optimal policy at time step t. In this case equation 8.3 holds.
From the algorithm it is clear that a state-action pair is eliminated **if and only if** the upper bound of the state-action value is lower than the lower bound of the state value:

$$\overline{Q}^t_\delta < \underline{V}^t_\delta = max_a\underline{Q}_\delta(s,a). \tag{8.4}$$

using Proposition 8.1.3 we see that **for every time** t with probability at least $1 - \delta$ the optimal state-action value Q^* falls between the interval estimations for Q. The probability of eliminating an optimal action at time-step $t + 1$ is therefore also bounded by proposition 8.1.3 which is at least $1 - \delta$. This completes the proof.

8.2 Implications

The proof presented above implicates that the algorithm will always give a correct output and even stronger, that given enough experience the algorithm will give the best possible output, the optimal recognizer. This means that if we follow this algorithm we can guarantee with probability $1 - \delta$ that the optimal solution, in this context the optimal policy π^* is recognized by the output of the algorithm, namely the recognizer function. This probability $1 - \delta$ can be made arbitrarily small. The trade-off that is evident from the proof is that the smaller the probability, the longer it takes before the stopping condition from 7.3.3:

$$\forall s \in S, \forall a \in U(s)\ |\overline{Q}(s,a) - \underline{Q}(s,a)| < \epsilon,$$

is reached.

8.3 Summary

This chapter provides a formal proof of the algorithm proposed in 7 and grounds the claims in a logical proof. The proof shows that our reduction of the problem space will converge to the most useful state-action pairs. We showed that the probability of an error can be made arbitrarily small at the cost of slower convergence. Vice versa we can accept an increased error probability for a fast convergence.

Part III

Implementation and empirical studies

"In the computer field,
the moment of truth is a running program;
all else is prophecy."
- Herbert Simon.

Chapter 9

Gridworld Task

This chapter and the next chapter contain our empirical results, the first and only published results of learning with recognizers. Currently it will also be the largest study of the performance of recognizers published. Our main focus is to assess the quality of the learning algorithm and to study its behavior in experiments. Assessment of the resulting recognizer function is part of this study since little empirical data is present in the literature. One of the goals of this thesis is to develop principled ways of learning the recognizer function from experience. Ultimately we wish to apply the results of this work to solve the tasks in the different domains mentioned in chapter 1. While the theoretical results of chapter 8 give an indication of the usefulness of our algorithm the ultimate test for applications is still empirical results obtained in practice. This chapter and the next can thus be said to provide us with the necessary 'experience' to build our conclusions on.

In this thesis we will look at two controlled environments that serve as prototypes for larger real world applications. There are two main requirements that we look for in our experimental problems:

- Capture essential elements of real life problems,
- transparent enough to analyze and visualize effects.

The first requirement ensures that our results will be relevant for real applications, the second requirement helps us to understand and interpret these results. Problems that do not capture essential elements of real life problems give meaningless results since they are only reproducible in the laboratory, while problems that are not transparent enough give results that are hard to understand and even harder to generalize. Einstein summarized our requirements best when he said: "Everything should be made as simple as possible, but not simpler."

This chapter will deal with the gridworld environment, arguably the most famous and most used test environment in reinforcement learning, while chapter 10 will deal with the coffee robot environment, a prototype for more complex, factored problems.

This chapter is organized as follows: The objectives of our experiments are established in section 9.1. We will describe the gridworld environment in section 9.2. The set up of the experiments and the different experiments are described in section 9.3. An interpretation and discussion of the results is found in sections 9.3.2 and 9.3.4.

9.1 Objectives

Before we describe what happened in the experiments it is very important to clearly describe what we want to accomplish with the experiments. These objectives should be relevant and measurable. With this in mind several objectives for this experiment can be formulated:

1. Assess working of the algorithm.
2. Assess quality of the recognizer.
3. Compare performance to other approaches.
4. Study effects of learning parameters.

The first objective, assess the working of the algorithm, is very clear. If the algorithm does not work or does not work as intended, our proposed algorithm fails to provide the promised performance. To assess this we need to answer questions such as: *Does the algorithm restrict actions? Are the restricted actions clearly sub-optimal? Does the output, the recognizer function, fulfill the requirements as stated in chapter 5?* This brings us naturally to the next objective, assess the quality of the resulting recognizer. It is not only important that 'a' recognizer is found, but also that the resulting recognizer is good according to a meaningful criterion. The proof in chapter 8 provides us with a theoretical promise that the algorithm will yield the best recognizer function possible for the associated MDP. In our experiments we can verify if this claim holds in the face of empirical evidence. Since our algorithm formulates a new and different approach to existing problems it is fair to ask how our solution compares to the current best solutions for these tasks. A comparison of performance will provide us with a frame of reference for the quality of our approach. Finally we hope to gain insight into the impact of the parameters on our algorithm by controlled variation of the values of several learning parameters and uncover what the optimal settings for our algorithm will be.

9.2 Gridworld Environment

The gridworld environment is the de facto standard test environment in reinforcement learning. The gridworld environment is attractive because it is one of the simplest environments that still retains the most important aspects of sequential decision making problems. The popularity of the gridworld environment among reinforcement learning researchers also means that it is well studied and that many results and implementations are available for the environment. Its preeminence is emphasized by its inclusion as an RL-benchmark in the annual RL-benchmarking event of the Neural Information Processing Systems (NIPS) foundations conference.

The gridworld environment serves as a simplified model in which navigation tasks can be performed. Objectives and tasks in the gridworld environment usually involve finding the shortest path to locations. Locations are cells in a grid that can only be traversed horizontally or vertically. The distance measure in this environment is not euclidian but rather the L1 norm or manhattan distance measure. In this world an agent may move in only four directions. Figure 9.1 shows a gridworld environment.

The state of the environment is defined with just two location parameters: x and y. Some states can also have a specific role such as starting state goal state or obstacle. This are usually predicates that can be defined implicitly but are technically part of the state.

The actions available to the agent in the gridworld are actions that take the agent from state to state which translates in this context to "from location to location". The actions in each state are restricted to the four directions: North, East, South and West. In our experiment we consider both the situation where actions have a deterministic result (actions always have the same result) and the situation where action results are stochastic.

In our implementation of the gridworld environment the goal of an agent is to reach a fixed goal state in the shortest possible time while avoiding obstacles. The agent starts in a random location. This problem setting is a simplified version of the general problem of robot navigation. Examples of real world applications that are similar are:

- Returning to home base while avoiding hostile territory for Unmanned Aerial Vehicles (UAVs).
- Delivering packets over a network while avoiding congestion and malfunctioning nodes (Network routing).

The task that we have used for our experiments consists of a maze that the learning agent must navigate in order to find a goal state. The environment also contains mines that should be avoided and obstacles that have to be navigated around. Each episode starts with the agent in a random starting position with a fixed goal position and environment. An episode ends if the agent reaches the goal state or if the agent steps on a mine. There is a +10 reward associated with reaching the goal state, a -10 reward for stepping on a mine and a -1 reward for each action taken. The actions available to the agent are restricted to up, down, left and right. When an obstacle is encountered the action has no effect (the agent will bump into the obstacle). There were 90 states in the environment, resulting in 360 state-action pairs.

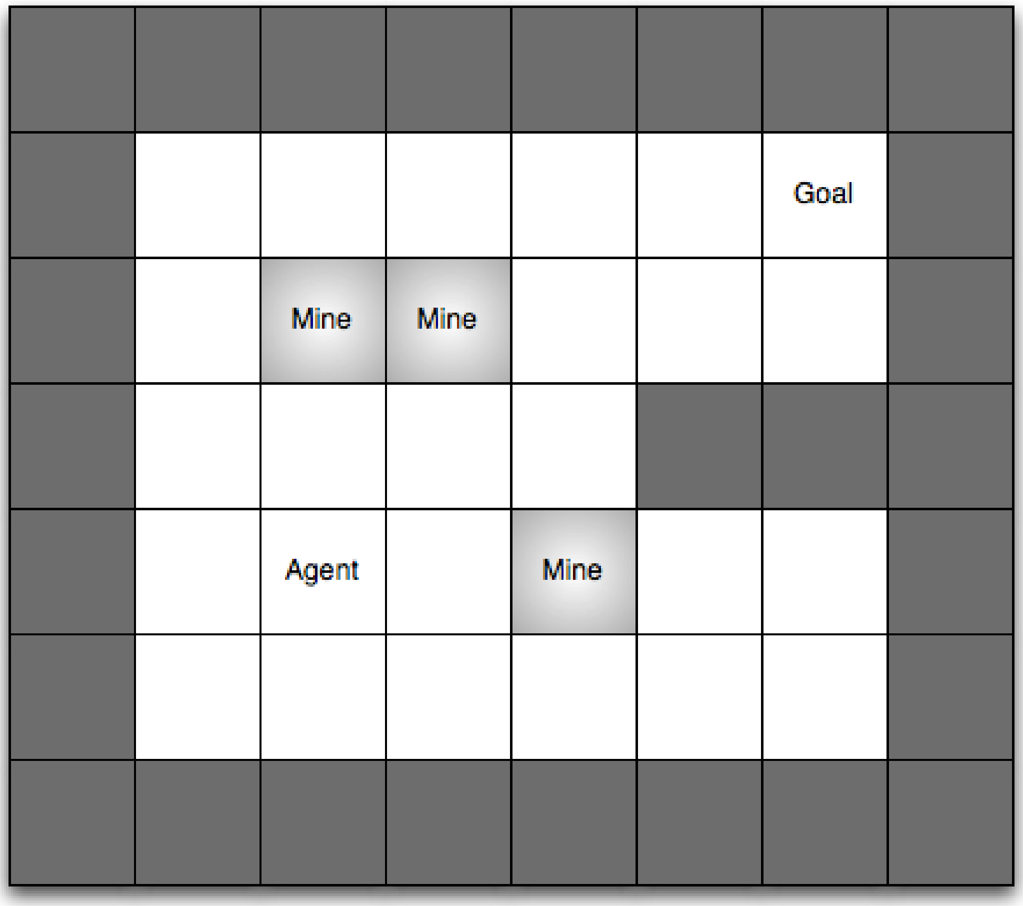

Figure 9.1: *An example of a gridworld with mines and obstacles.*

9.3 Experiment Design

A good experiment design must ensure that the results are applicable and that the results are reproducible. To enable learning and reduce the variance of our results the task consisted of 100 episodes, results are measured over this time-span.

We compared our algorithm to two existing algorithms:

- The Q-learning algorithm from section 3.3.2.
- The Sarsa algorithm from section 3.3.1.

In the following two sections we will describe the different experiments that have been performed.

9.3.1 Experiment 1: Performance comparison

In this experiment the accumulated reward per time step is measured. We compared our algorithm, that learns a recognizer function and restricts actions with an ϵ-greedy Q-learning algorithm and a Sarsa learning algorithm.

Our hypothesis is that the recognizer algorithm will converge to a recognizer function that will not allow actions that result in stepping on a mine since the value of this will be lower than the values of other actions in the same state.

We predict that if our hypothesis holds, an agent using our algorithm will obtain a higher average reward per time step than respectively the Q-learning agent and the Sarsa agent, because it will learn to never step on a mine, even as an exploration step.

The results can be seen in Figure 9.2. It is clear from the figure that the recognizer algorithm obtains higher rewards. The initial low average reward of our algorithm as compared with the other two algorithms is probably due to the high initial values of our algorithm and the fact that it has to learn the structure of the recognizer function. All three algorithms use the ϵ-greedy action selection strategy.

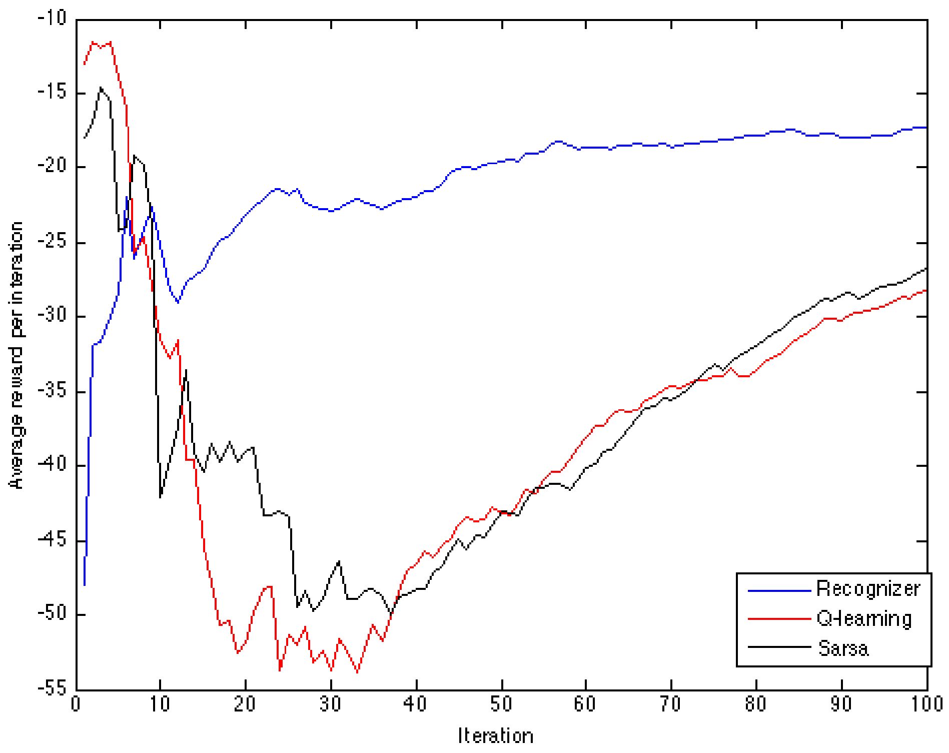

Figure 9.2: *The reward gained per iteration for the recognizer, Q-learning agent and Sarsa.*

9.3.2 Experiment 1: Results and discussion

Figure 9.2 clearly shows that the recognizer algorithm performs significantly better than its competitors, Q-learning and Sarsa. The probable cause of this performance is that after an initial period of exploration, the recognizer algorithm drops undesired actions. To investigate the probable cause we have visualized the restriction operator of the algorithm in figure 9.3. When we take a look at figure 9.3 we see that some actions are indeed dropped. Upon inspection of the recognizer function, it is revealed that the dropped state-action pairs correspond to actions that normally would result in stepping on a mine. Eliminating these actions improve the performance of an agent in this environment since it effectively is prohibited of going near a mine thereby avoiding the high negative rewards associated with that situation.

The results show that our algorithm compares favorably to existing algorithms in reinforcement learning. Our algorithm does a better job of capturing the relevant structure of a problem and then confines solutions to lie within this structure, the recognizer function. Learning the structure incurs some overhead but the overall performance is not affected by this overhead. The benefits outweigh the costs.

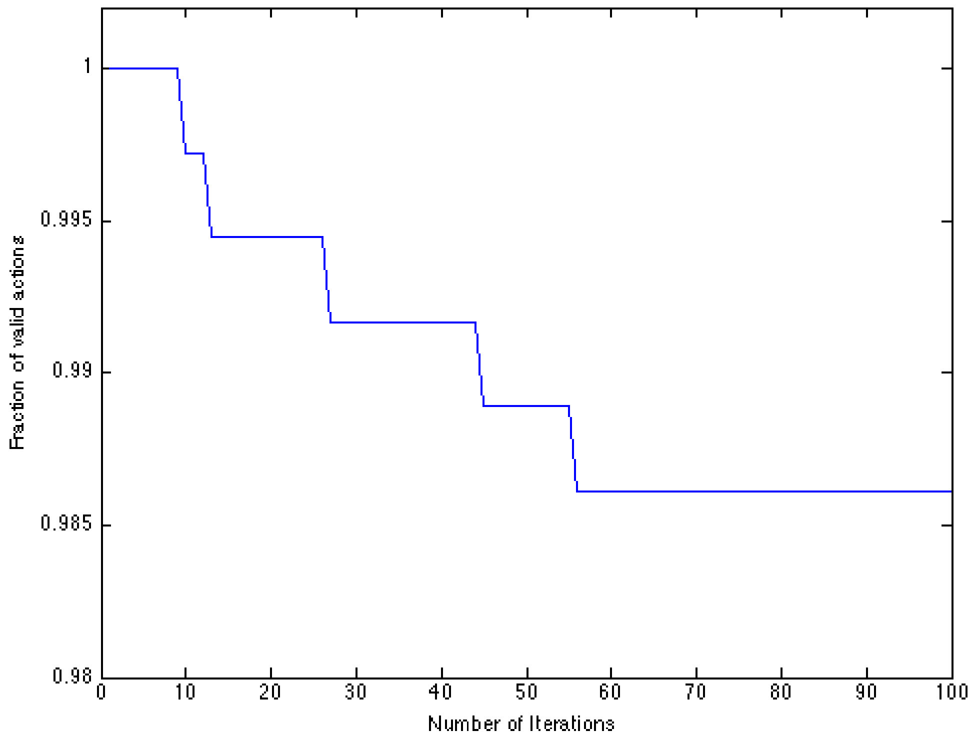

Figure 9.3: *The figure shows the fraction of the state action space that is recognized after each iteration.*

9.3.3 Experiment 2: Varying exploration parameter

The exploration parameter ϵ regulates the trade-off between exploration and exploitation of all three algorithms. Some exploration is always needed in order to find the optimal solution since the current estimates of the best actions to take (the greedy choice) is only guaranteed to be locally optimal. This would lead us to conclude that the exploration parameter should have a fairly large value. However the higher the exploration parameter is set, the higher the probability of taking sub-optimal actions in order too explore becomes. Setting the exploration parameter too high therefore results in a low performance.

It is clear that the amount of exploration has a profound impact on the performance of any learning algorithm. Our hypothesis is that our proposed algorithm performs structurally better than the other algorithms, independent of the exploration parameter.

We predict that if our hypothesis holds, an agent using our algorithm will obtain a higher average reward per time step than respectively the Q-learning agent and the Sarsa agent, even if effects on performance of the exploration parameter are taken into account

To assess the effects of the exploration parameter on the algorithms we have varied the exploration parameter for all three algorithms and plotted their performance.

First we look at the recognizer algorithm in isolation. Figure 9.4 shows the performance of the recognizer algorithm for various values of ϵ, the exploration parameter. The figure shows several things. First we can conclude that the exploration parameter has an effect on performance since the performance seems to change as a function of the exploration parameter. The figure indicates that there is also a (unique) optimum for the value of the exploration parameter. This hypothesis is grounded on the observation that very low and very high values of the exploration parameter perform suboptimal.

Now we investigate the effects of varying the exploration parameter for the other algorithms and comparing the results to verify if our algorithm does indeed provide a structural improvement in performance or that it should be attributed to the parameter settings of the agent. Figures 9.5, 9.6 ,9.7 and 9.8 give the performances of the three algorithms for respectively ϵ values of 0.05, 0.1,0.3 and 0.5. What the graphs show is that the recognizer learning algorithm dominates the other two methods, Q-learning and Sarsa, for every value of ϵ.

9.3.4 Experiment 2: Results and discussion

From the figures we may conclude that our algorithm brings benefits that are independent from the choice of the exploration parameters. So an added advantage of using our algorithm is its robustness to variation of ϵ. The explanation for this is twofold. One aspect is that the other two algorithms and especially Sarsa since it is an on-policy algorithm (see section 3.3.1) are very sensitive to the choice of the exploration parameter. The other aspect is that our algorithm continually reduces the number of states-action pairs that it recognizes and limits it exploration within this reduced subspace of the original problem space. Our algorithm also theoretically ensures that the recognized subspace contains the high value pairs so that exploration within this subspace will yield increasingly relevant (near optimal) behavior. This effect seems to be

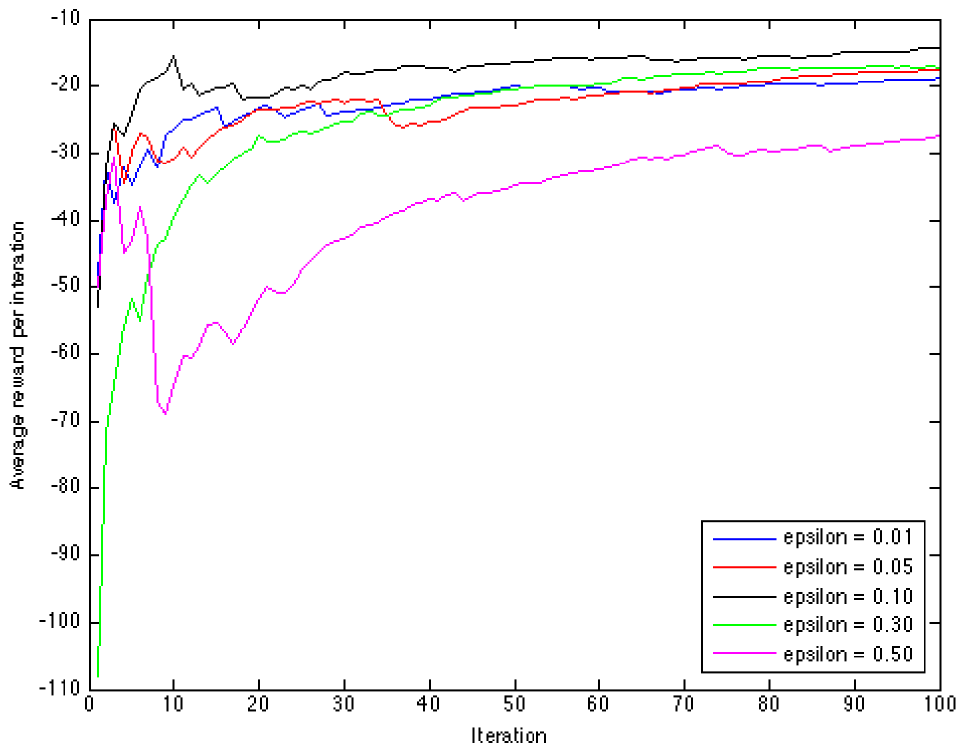

Figure 9.4: *The reward gained per iteration for the recognizer algorithm for different values of ϵ.*

corroborated by the results.

A critical note that we must make is that there is a possibility that the exploration in our algorithm degrades because dropped state-action pairs (where the recognizer function is 0) are not explored anymore. We proved that this possibility may be made arbitrarily small in chapter 8 but the possibility remains, especially when we tolerate higher error probabilities.

9.4 Summary

This chapter presented the first empirical results of our algorithm and recognizers using the gridworld environment a popular testbed with reinforcement learning researchers. We tested several hypotheses and compared our approach to existing learning algorithms. We also provide empirical evidence that support the theoretical claims of chapter 8. Finally we discussed the results of the experiments in light of the workings of the algorithm as defined in chapter 7.

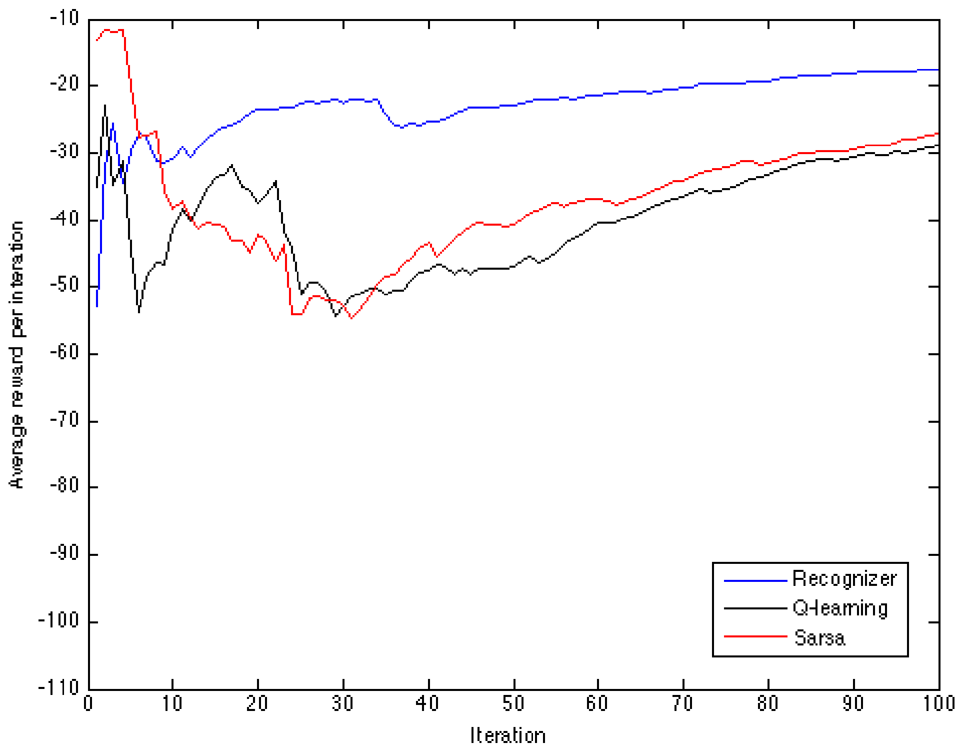

Figure 9.5: *The reward gained per iteration for the three learning algorithms for* $\epsilon = 0.05$.

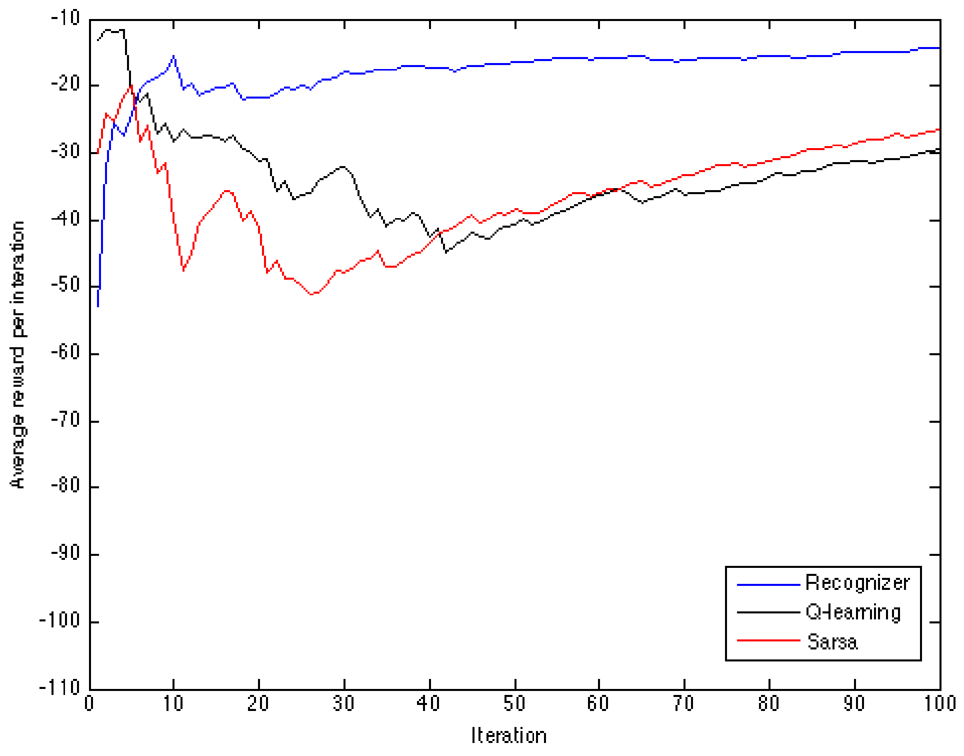

Figure 9.6: *The reward gained per iteration for the three learning algorithms for* $\epsilon = 0.10$.

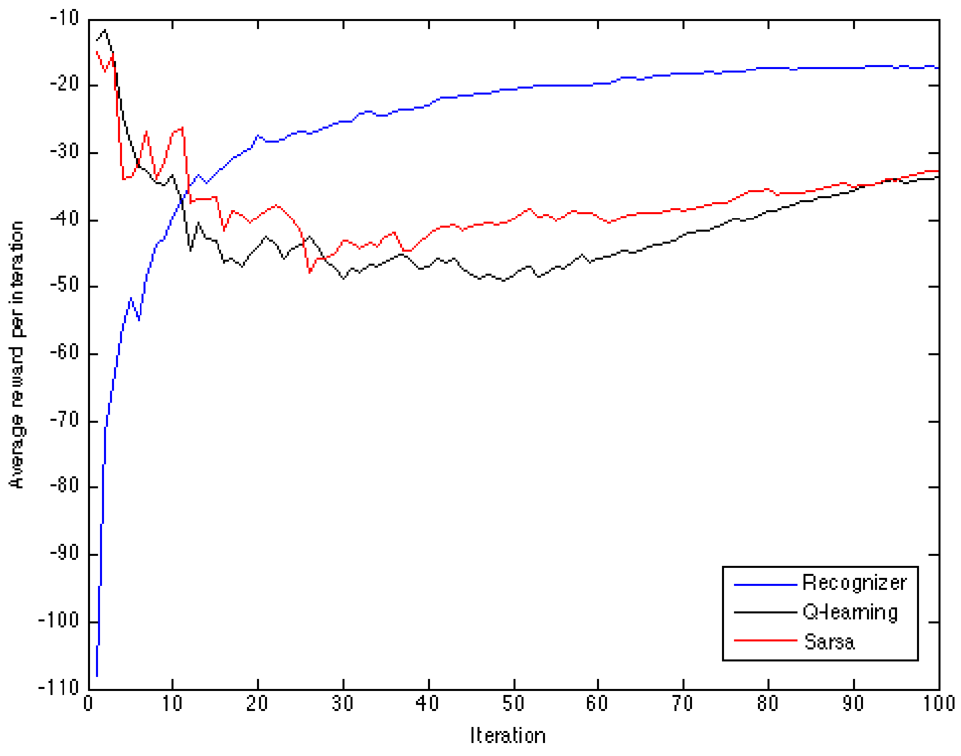

Figure 9.7: *The reward gained per iteration for the three learning algorithms for* $\epsilon = 0.30$.

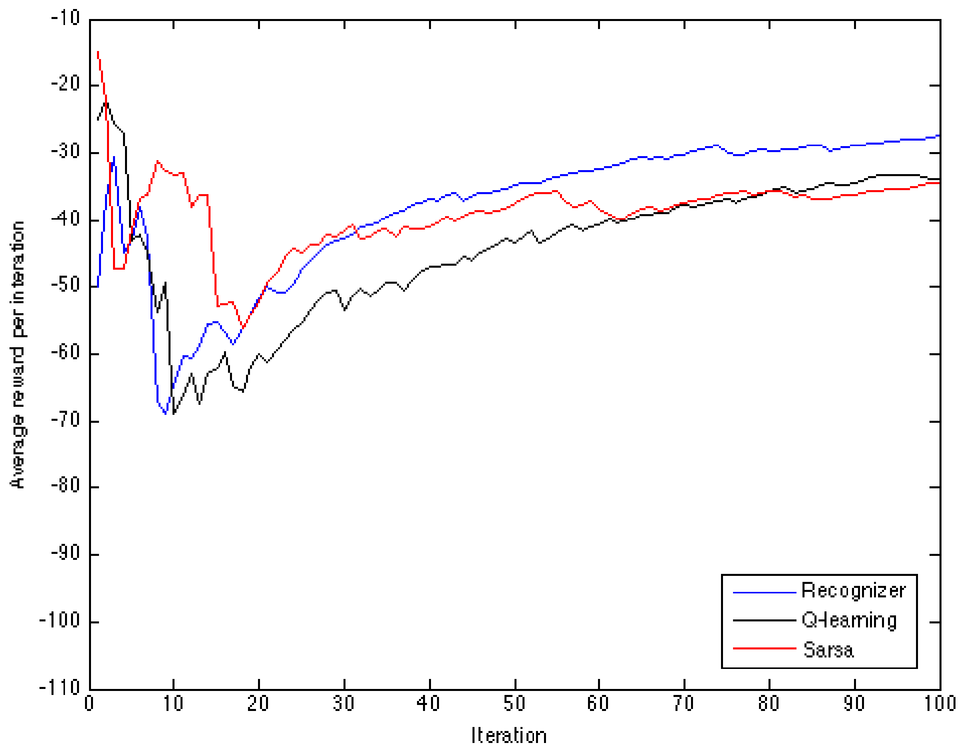

Figure 9.8: *The reward gained per iteration for the three learning algorithms for* $\epsilon = 0.50$.

Factored MDP Task

Chapter 10

In this chapter we will experiment with a special family of problems. The environment we will consider is a prototype of an important class of problems known as factored MDPs. This type of MDP allows a task to be decomposed in multiple smaller subtasks that can be solved individually. This approach enables efficient solutions for larger problems, one of the reasons recognizers have been developed. If recognizers can be applied to factored MDPs, the combination would scale very well and make problems feasible that are currently outside of the reach of reinforcement learning.

This chapter is separated in the following sections. In section 10.1 we will explain what precisely is meant by the term factored MDP. Some elaboration on their usefulness is also given. In section 10.2 we present the objectives for the experiments. Section 10.3 provides a detailed description of the environment that we use in the experiment, the set-up and results of the experiment are presented in section 10.4. An interpretation of the results is reserved for section 10.4.2.

10.1 Factored MDPs

We use this domain primarily to investigate one of the possible advantages of the use of recognizers. This particular advantage is the property that recognizers do not rely heavily on the behavior policy, requiring only that it visits salient states. Recognizers should therefore work exceptionally well in settings for which it is hard to define a good behavior policy. The domains that we study in this chapter are simple examples of factored MDPs (FMDP), MDPs described by several state variables. The usefulness of FMDPs lies in the fact that they can be decomposed into smaller subproblems that can be solved individually. The individual solutions can then be recombined to arrive at a global solution. One possible method to accomplish this would be to define options for each subproblem. A problem with FMDPs is that the resulting policy is itself a composition made up of the policies of the subproblems and is not directly available or even describable. The overall policy *emerges* out of the policies of the subproblems and could even be called a form of procedural knowledge. The subproblems themselves can also be FMDPs requiring decomposition on multiple levels. The decomposed nature of FMDPs make it very difficult for traditional reinforcement learning algorithms to learn from the emergent policies. It is our hypothesis that recognizers can work very well with such composed policies since they do not put any constraints on how the policy is formed.

A factored MDP is described by a set of state variables $\{S_i\}_{i \in D}$, where D is a set of indices. The set of states $S = \times_{i \in D} Val(S_i)$ is the cross-product of the value sets $Val(S_i)$ of each state variable S_i. A state $s \in S$ assigns a value $s_i \in Val(S_i)$ to each state variable S_i. Note that in this context Val means the range of values a state variable can attain and is not related to the concept of value functions used throughout this thesis.

The usefulness of this model lies in the conditional independence of state variables on the transition probabilities. when an action is executed, the resulting value of a state variable usually depends only on a subset of the state variables. The model takes advantage of this structure by introducing dynamic Bayesian networks, or DBNs [Dean and Kanazawa, 1990], approximating the transition probabilities and expected rewards associated with actions. An example of a DBN for our domain is given in figure 10.1, see also section 10.3.

10.2 Objectives

The main objective of this experiment is to provide a proof of concept for the use of recognizers in combination with factored MDPs. Secondary objectives deal with the properties that such solutions have.

First we want to answer the question: *Can recognizers work with FMDPs?* So we ask if there exists a solution at all. If solutions exist we would like to answer questions concerning the properties of such solutions. The questions involved are: *How does the absence of an overall behavior policy impact the learning algorithm? Are their any benefits of using our algorithm? Can we measure the benefits of our algorithm in combination with factored MDPs?*

The importance of these objectives lies in the potential of solving large scale problems if the combination of recognizers and factored MDPs turns out to be feasible. Solving large factored MDPs is currently very hard because the solution (and therefore the policy) is distributed. This experiment can establish results that can significantly simplify finding solutions for factored MDPs.

To conclude, the objectives of this experiment can be summarized as:

- Present a proof of concept of recognizer learning with factored MDPs.

- Investigate the results of recognizer learning without an overall behavior policy.
- Investigate possible improvements of applying our algorithm to factored MDPs.

10.3 Coffee Robot Environment

The environment used in the experiments is called the coffee robot task in the literature and was introduced by [Boutilier et al., 1995]. It is a simple task that serves as a prototype for tasks described by factored MDPs. As stated above factored MDPs show characteristics that make them very suitable for solutions involving recognizers and options. The lack of an overall behavior policy would be a problem for many learning algorithms but not for recognizers.

We looked at the **Coffee Robot** task from [Boutilier et al., 1995], in which a robot has to deliver coffee to its user. The coffee task is described by six binary variables: $\mathbf{S_L}$, the robot's location (office or coffeeshop); $\mathbf{S_U}$, wether the robot has an umbrella; $\mathbf{S_R}$, whether it is raining; $\mathbf{S_W}$, wether the robot is wet; $\mathbf{S_C}$, whether the robot has coffee; and finally $\mathbf{S_H}$, wether the user has coffee. To distinguish between variable values we use the notation $Val(\mathbf{S_i}) = \{i.\bar{i}\}$, where $\overline{L}$ = office and L = coffeeshop. An example state is $s = (\overline{L}, U, R, \overline{W}, C, \overline{H})$. The robot has four actions: *GO*, causing its location to change and the robot to get wet if it is raining and it does not have an umbrella; *BC* (buy coffee) causing it to hold coffee if it is in the coffee shop; *GU* (get umbrella) causing it to hold an umbrella if it is in the office; and *DC* (deliver coffee causing the user to hold coffee if the robot has coffee and is in the office. All actions have a chance of failing. The robot gets a reward of 0.9 when it successfully delivers the coffee to the user (H) plus a reward of 0.1 if it remains dry ($\overline{W}$).

10.4 Experiment Design

The following experiment has been conducted using the coffee robot domain as described in section 10.3. Our algorithm does not provide in ways to decompose a factored MDP by itself so we need to invoke another algorithm to do this. A state of the art algorithm that can automatically decompose factored MDPs is the VISA algorithm by Jonsson and Barto [Jonsson, 2006]. VISA is an acronym for *Variable Influence Structure Analysis* and dynamically performs hierarchical decomposition of factored MDPs. VISA can determine causal relationships between state variables and introduces options for these relationships. It uses the DBN model of the factored MDP to infer these causal relationships.

Our experiment uses the VISA algorithm for all decomposition steps. Learning takes place using recognizers. An implementation of the VISA algorithm was kindly provided by Anders Jonsson, author of the VISA algorithm. As a baseline we have taken the VISA algorithm by Jonsson and Barto [Jonsson, 2006] and applied our recognizer learning algorithm on the top level. The VISA algorithm can automatically decompose a factored MDP into subtasks and creates options that solve these tasks. Although we have restricted ourselves to learning a recognizer for the toplevel only of the overall task in this experiment, recognizer functions could be learned for every level of the decomposition.

10.4.1 Experiment 1: Performance improvement of recognizers

We compared the average reward per trial of the original VISA algorithm and the average reward of the target policy resulting from the learned recognizer. The original VISA algorithm uses a modified Q-learning algorithm to learn the policy. A trial ended when the robot completed its task or when the maximum number of actions per trial was reached (4000). The results of the first 1500 trials can be seen in Figure 10.2. What we see is that the average reward per trial improves by introducing the recognizer function. In this particular domain the recognizer function actually converges to the optimal policy as indicated by the proof in 8 resulting in the optimal average reward per trial. The variance is due to the fact that every action has a chance of failure. By only allowing the optimal action in every state the recognizer cuts away approximately 75% of the state-action space, a huge reduction, effectively making the problem four times smaller.

10.4.2 Experiment 1: Results and discussion

We presented the first results ever of learning recognizers in a factored MDP environment. The results show that our algorithm can work with this type of MDP and moreover, the results show that recognizers can improve the performance of existing algorithms. The expectation is that the combination recognizers and factored MDPs will perform better than current solutions and more importantly will bring larger problem domains within reach.

The improvement in performance is not very large in an absolute sense. In our experiment this is in all likelihood attributed to the high performance of our baseline, the VISA algorithm. There was simply not much room left for improvement. The recognizer function that our algorithm learned in this experiment was equal to the optimal policy of the task. In one way this is positive since this result is predicted by our proof in chapter 8. On the other hand it also shows that we lack a good stopping criterion since the optimal policy is not transferable to related instances of this task. A recognizer function that is a bit wider would be preferable. Perhaps this result is due to the relatively small size of the problem. More experiments with our algorithm and factored MDPs should confirm or falsify this hypothesis.

10.5 Summary

This chapter contains the second environment that we used to conduct our experiments in. We described the form of this special class of problems, namely factored MDPs, and expounded the advantages that they posses. We conducted some experiments that form a proof of concept for the successful fusion of recognizers and our algorithm with options and the VISA algorithm. The results were encouraging but the discussion also pointed out that more integration is required before all the kinks are ironed out.

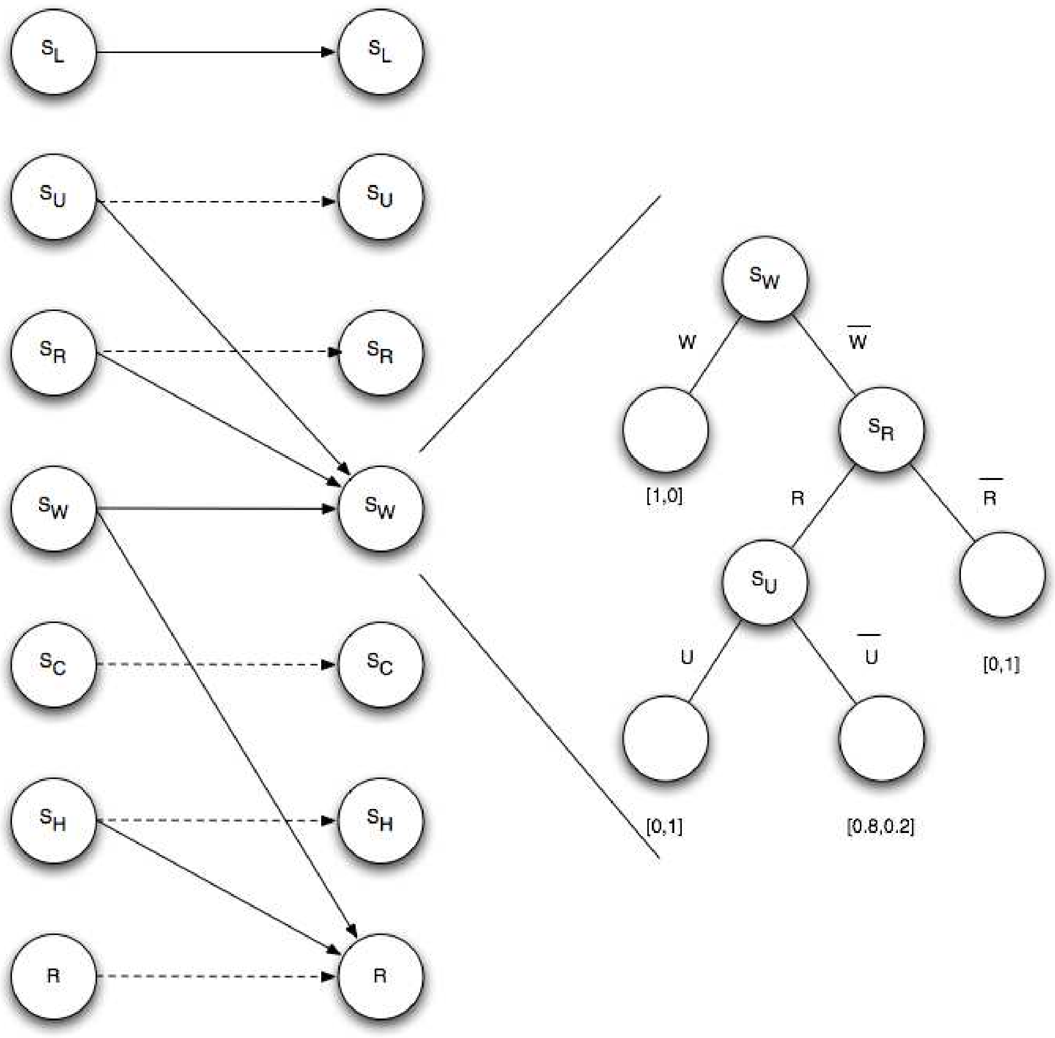

Figure 10.1: *The DBN for action GO in the coffee experiment.*

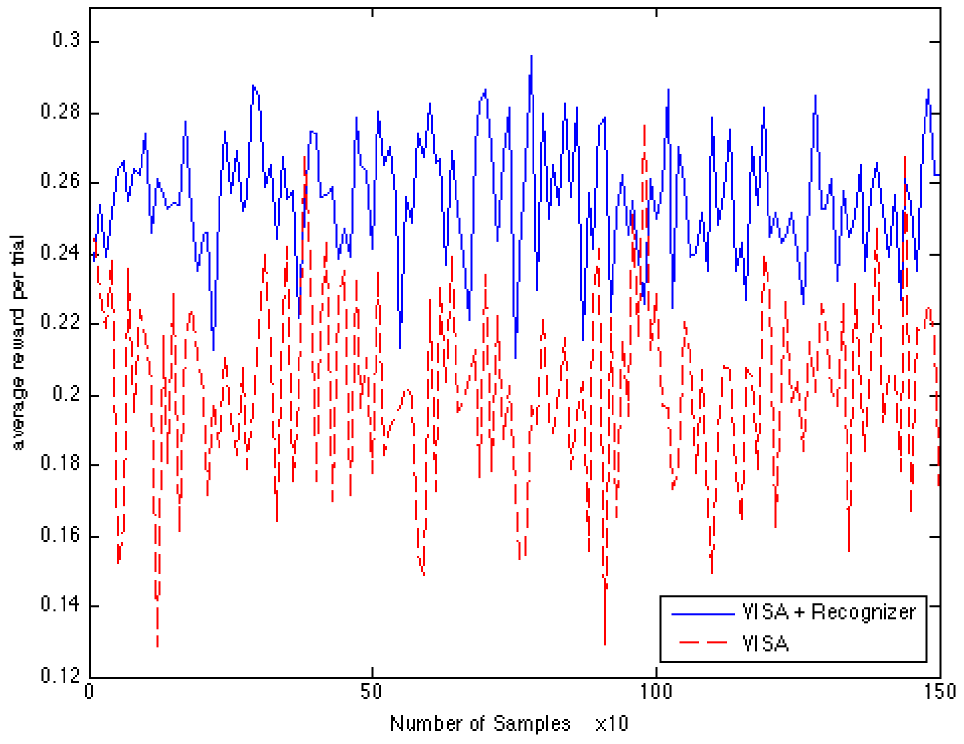

Figure 10.2: *The average reward per trial in the coffee domain.*

Part IV
Final Results

“Qoute”
- Unknown Author.

Chapter 11

Conclusions and Future Work

In this chapter we review and discuss the goals that were formulated and set out at the beginning of our research in chapter 1. We discuss what the contributions of this thesis are for the reinforcement learning field and beyond. Finally we look at the opportunities for future research, some of which are already in the process of being followed up.

11.1 Goals and Objectives

In the introduction we formulated the following research questions:

1. How can we learn behavior from experience?
2. In which way can we scale up these ideas to find solutions for large scale problems?
3. Can we improve one of these ideas, recognizers, by making recognizers learnable autonomously from experience?
4. How much experience (data) is needed before we can find a good solution?

Based on these questions, we formulated our assignment. The original assignment consisted of three parts: a theoretical part, a proposed solution, and an application part. In turn, these three parts consisted of our goals and objectives of the research presented in this thesis. We will consider each of these parts and we will look at how we addressed our assignment for each part.

11.1.1 Theoretical part

1. **Investigating existing RL theory**
   We presented a survey of the reinforcement learning field in chapters 2 , 3 and chapter 4. This served as both the introduction of the necessary tools and models used in our research as well as the frame of reference for the entire researchassignment. By surveying the existing theory insight was gained into the big picture. We presented our results and demonstrated that the current theory lacks methods that can efficiently scale up to solve large problems. The current research also provided us with leads that we could follow and approaches that aim to address the scaling challenge.

2. **Investigating the theory of Recognizers**
   We defined and presented the concept of recognizers, summarizing the current research. We have established links between recognizers and options and shown how recognizers can provide an answer to the curse of dimensionality, thereby bringing larger problems within reach. Where the first objective, investigating the existing RL theory, was mainly concerned with the big picture, this objective provided the focus of our research assignment and zoomed in on the solution that we found most promising and interesting, recognizers. This objective also made clear what the main bottleneck for the success of recognizers is. This is the enormous effort that goes into acquiring the recognizer function, something that has not been addressed in current research.

11.1.2 Proposed solution

3. **Developing learning algorithms for recognizers**
   In chapter 7 we presented a learning algorithm that removes the effort of defining recognizers by hand and can learn recognizers from experience. The algorithm is presented as an iterative process compiled of simple steps that performs the task of autonomously learning the recognizer function.

4. **Validating the proposed learning algorithm**
   A proof that guarantees the correctness of our algorithm was presented in chapter 8. This proof uses previous results in reinforcement learning to build on and the proof relates our algorithm to known correct algorithms for reinforcement learning. It also provides us with bounds on the amount of experience needed, thereby answering the question of how much data is needed.

11.1.3 Application part

5. **Applying the algorithm to benchmark problems**
   In chapters 9 and 10 an implementation of the proposed algorithm was tested against our hypotheses and against existing algorithms in different but controlled environments and with varying parameters. The results show that our solution is not only viable but outperforms current algorithms significantly

11.2 Contribution to the Field

This thesis presents the first steps towards making the recently discovered and powerful concept of recognizers learnable from experience. Recognizers play an important role because they enable low variance importance sampling off-policy learning. This aspect makes recognizers especially suitable in combination with options.

Our algorithm forms a solid starting point to develop more powerful implementations that use available data even more efficiently.

The theoretical claims and proof form a solid foundation of the analysis of learning recognizers and provide empirically verifiable guarantees on the performance of our algorithm.

The empirical study provides a proof of concept of the algorithm and gives the first results in common test environments so that comparison with other learning algorithms is facilitated.

Recognizers and similar methods rely on the concept that computation is cheap but data is expensive. Our work provides the first instance where the recognizer function is not provided beforehand by human experts and as such it is a departure from the existing theory. In our opinion this thesis forms a major contribution to the development and study of recognizers within reinforcement learning.

11.3 Future Work

While the main contribution of this thesis is self contained, our work provides a starting point to develop the theory of recognizers further. The future work and goals can be split along many dimensions, here we will treat two dimensions: long term vs short term and practical vs theoretical.

Short term:

Practical: Function Approximation
This is necessary in order to handle MDPs that have ex state and action spaces a characteristic of many real world problems. Work on this subject has been started and forms the subject of Jordan Franks thesis at McGill.

Theoretical: Stopping Criterion
What constitutes an optimal stopping criterion and is there one anyway falls outside the scope of this thesis but must be answered to unlock the capabilities of recognizers.

Long term:

Practical: Incorporation of Prior Knowledge
This forms a practical subject of study since the effective use of prior knowledge will increase performance. Theoretically this does not add any new overhead basically start with actions blotted out or guaranteed instead of a blank slate.

Theoretical: Transfer of Knowledge
Recognizers show great promise as enablers of knowledge transfer both between domains within one agent as between different agents in MAS settings.

Bibliography

[Barto and Mahadevan, 2003] Barto, A. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341 – 379.

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

[Bertsekas and Tsitsiklis, 1996] Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

[Boutilier et al., 1995] Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. In Mellish, C., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, San Francisco. Morgan Kaufmann.

[Cohen et al., 2007] Cohen, P., Chang, Y.-H., and Morrison, C. (2007). Learning and transferring action schemas. In *Twentieth International Conference on Artificial Intelligence*.

[Dean and Kanazawa, 1990] Dean, T. and Kanazawa, K. (1990). A model for reasoning about persistence and causation. *Comput. Intell.*, 5(3):142–150.

[Dietterich, 1998] Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conference on Machine Learning (ICML '98)*.

[Even-Dar et al., 2003] Even-Dar, E., Mannor, S., and Mansour, Y. (2003). Action elimination and stopping conditions for reinforcement learning. In *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*.

[Jokinen et al., 2002] Jokinen, K., Rissanen, J., Keränen, H., and Kanto, K. (2002). Learning interaction patterns for adaptive user interfaces. In *Proceedings of the 7th ERCIM Workshop User Interfaces*.

[Jonsson, 2006] Jonsson, A. (2006). Journal of machine learning research 7 (2006) 2259-2301 submitted 10/05; revised 7/06; published 11/06 causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Kearns and Vazirani, 1994] Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. The MIT Press.

[Koza, 1992] Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. MIT Press.

[MacQueen, 1966] MacQueen, J. (1966). A modified dynamic programming method for markov decision problems. *Math. Anal. Appl., 14, 38-43*.

[Mahadevan, 1996] Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1-3):159–195.

[Minsky, 1954] Minsky, M. L. (1954). *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem.* PhD thesis, Princeton University.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraww Hill.

[Moody and Saffel, 2001] Moody, J. and Saffel, M. (2001). Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12.

[Parr and Russel, 1998] Parr, R. and Russel, S. (1998). Reinforcement learning with hierarchies of machines. In *NIPS*.

[Precup, 2000] Precup, D. (2000). *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst.

[Precup et al., 2005] Precup, D., Sutton, R. S., Paduraru, C., Koop, A., and Singh, S. (2005). Off-policy learning with recognizers. In *Proceedings of NIPS '05 conference*.

[Puterman, 1994] Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA.

[Rummery and Niranjan, 1994] Rummery, G. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical report, Cambridge University.

[Russel and Norvig, 1995] Russel, S. and Norvig, P. (1995). *Artificial Intelligence: a Modern Approach*. Prentice-Hall.

[Schank and Abelson, 1977] Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding: an Inquiry into Human Knowledge Structures*. L. Erlbaum.

[Seip, 2006] Seip, H. (2006). Reinforcement learning and applications in finance. Technical report, Delft University of Technology.

[Skinner, 1938] Skinner, B. (1938). *The Behavior of Organisms: An Experimental Analysis*. Prentice Hall.

[Strehl and Littman, 2005] Strehl, A. L. and Littman, M. L. (2005). A theoretical analysis of model-based interval estimation. In *Proceedings of the 22nd international conference on Machine learning*, pages 856–863, New York, NY, USA. ACM Press.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An Introduction*. MIT Press.

[Sutton et al., 1999] Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68.

[Thorndike, 1911] Thorndike, E. L. (1911). *Animal Learning*. Hafner, Darien, Conn.

[Valiant, 1984] Valiant, L. G. (1984). A theory of the learnable. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445, New York, NY, USA. ACM Press.

[Watkins, 1989] Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College Cambridge, UK.

[Watkins and Dayan, 1992] Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*.

Part V

Appendices

Appendix A

Code of Experiments

A.1 Introduction

This appendix contains the source code of the experimental setups. The recognizer algorithm is implemented in C++ as well as in java. Portability of the coding should allow easy ports to other porgramming languages.

This code is for educational purposes only and may not be redistributed in any form without written consent and is not considered in the open source domain.

```
#include "mineAgentH.h"
#include <math.h>

void agent_init(Task_specification task_spec)
{

    if(task_spec != NULL)
    {
        //Parse task_spec...General case where I dont know in adavance
           number of state and actions dimensions
        //if you know use one call to sscanf
    int j = 0;
    double init;
    try
    {
      task_spec_struct ps;
      parse_task_spec(task_spec, &ps);
      version = ps.version;
      state_dim = (int)ps.obs_maxs[0]+1;
      action_dim = (int)ps.action_maxs[0]+1;

    } //done parsing ...not so bad!!
        catch(int e)
        {
            fprintf(stderr,"Error:␣\n");
            fprintf(stderr,"I␣did␣not␣understand␣your␣task␣spec.␣I␣need␣'
                numStates␣numActions'\n");
            exit(-1);
    }

    n = state_dim; //WE KNOW ONLY OE STATE DIMENSION
    m = action_dim;    //again we know number of action dim
    fprintf(stderr,"task_spec:␣%s,␣num_states:␣%d,␣num_actions:␣%d\n",
        task_spec,n,m);
    nm = n*m;

    actions = new int[m];//array of possible actions
    for(int i = 0; i < m; i++)
      actions[i] = i;


    init = 2*log(7218); // keeps track of uppervalue estimates; initialized
        to Vmax
    upperQ = new double*[n];
    for( int i = 0; i < n; i++)
      upperQ[i] = new double[m];

    for(int i = 0; i < n; i++)
      for(int j =0; j < m; j++)
        upperQ[i][j] = init;

    lowerQ = new double*[n];    // keeps track of lowervalue estimates;
        initialized to -Vmax
    for( int i = 0; i < n; i++)
      lowerQ[i] = new double[m];

    for(int i = 0; i < n; i++)
      for(int j =0; j < m; j++)
        lowerQ[i][j] = 0-init;  // keeps track of # times visited
    countSA = new int*[n];
    for( int i = 0; i < n; i++)
      countSA[i] = new int[m];

    for(int i = 0; i < n; i++)
      for(int j =0; j < m; j++)
        countSA[i][j] = 1;

    //keeps track of recognized actions (where upperQ[s][a] >= lowerQ[S][
        argmax_lower(S)])
    recognizer = new int*[n];
    for( int i = 0; i < n; i++)
      recognizer[i] = new int[m];

    for(int i = 0; i < n; i++)
      for(int j =0; j < m; j++)
        recognizer[i][j] = 1;


    alpha = 0.1;
    epsilon = 0.01;
    discount = 0.99;
    srand(999);
    T=0;
    }

}

Action agent_start(Observation o)
{
  //Choose and return the agent's first action
  A = egreedy(o);
  S = o;
  return actions[A];
}

Action agent_step(Reward r, Observation o){   //Update one step, return
    agent's action`
  double delta, temp;
  int a;
  temp = 0;

  alpha = 1.0/countSA[S][A];
  const double Tolerance = 0.0000001;
  if(alpha < Tolerance){
    alpha=0.0;
  }
  else{
    if (!stop_test()) { // stopping condition
      delta = r + discount*upperQ[o][argmax_upper(o)] - upperQ[S][A] + beta(
          countSA[S][A]);
      upperQ[S][A] = upperQ[S][A] + alpha*delta;

      delta = r + discount*lowerQ[o][argmax_lower(0)] - lowerQ[S][A] - beta(
          countSA[S][A]);
      lowerQ[S][A] = lowerQ[S][A] + alpha*delta;

    }
  }
  a = egreedy(o);
  countSA[S][A]++;        // update
  A = a;
  S = o;
  T++;
  return actions[a];
}

void agent_end(Reward r)
{
  //Update last step of current episode
  int i;
  double error;

  upperQ[S][A] = r; //upperQ[S][A] + alpha*error;

  lowerQ[S][A] = r; //lowerQ[S][A] + alpha*error;

  T = 0;

}

void agent_cleanup()
{
  double frac, tot, ans;
  frac = 0;
  tot = 0;
  for(int i = 0; i < n; i++){
    for(int j =0; j < m; j++){
      if(recognizer[i][j] == 1) frac++;
      tot++;
    }
  }
  ans = frac/tot;

  fprintf(stderr,"Fraction␣recognized␣is␣%g␣frac␣=␣%g␣tot␣%g␣", ans, frac,
      tot);

  for( int i = 0; i < n; i++){
    delete [] upperQ[i];
    delete [] lowerQ[i];
    delete [] countSA[i];
    delete [] recognizer[i];
  }

  delete [] upperQ;
  delete [] lowerQ;
  delete [] countSA;
  delete [] recognizer;
  delete [] actions;
}

int egreedy(int s)
{
  //chooses action epsilon-greedily but only from recognized state action
      pairs
  int ans;

  double ran = rand()/static_cast<double>(RAND_MAX);
  if( ran > epsilon ) // param epsilon
    ans = argmax_lower(s);
  else
    ans = (int)(rand() % m);

  while (1) {
    if(upperQ[S][ans] >= lowerQ[S][argmax_lower(S)]){
      return ans;
    }
    else{
      recognizer[S][ans] = 0;
      ans = (++ans)%m;
    }
  }
}

int argmax_upper(int s){
  //find the index of the maximal action, break ties randomly
  int bestIndex;
  double bestValue, value;
  int numBests = 1;
  bestIndex = 0;
  bestValue = upperQ[s][0];
  for(int i = 1; i < m; i++){
            value = upperQ[s][i];
            if( value > bestValue ){
                bestValue = value;
                bestIndex = i;
                numBests = 1;
            }
            else if (value == bestValue) {
                numBests = numBests + 1;
                if( (int)(rand() % numBests) == 0 )
                    bestIndex = i;
            }
  }
  return bestIndex;
}

int argmax_lower(int s){  //find the index of the maximal action, break ties
     randomly
  int bestIndex;
  double bestValue, value;
  int numBests = 1;
  bestIndex = 0;
  bestValue = lowerQ[s][0];
  for(int i = 1; i < m; i++){
            value = lowerQ[s][i];
            if( value > bestValue ){
                bestValue = value;
                bestIndex = i;
                numBests = 1;
            }
            else if (value == bestValue) {
                numBests = numBests + 1;
                if( (int)(rand() % numBests) == 0 )
                    bestIndex = i;
            }
  }
  return bestIndex;
}

double beta(double c){
  // computes the betas
  double d;
  d = 7218*c*c;
  d = log(d);
  d = d/c;
  d = sqrt(d);

  return d;
}

int stop_test(){
  // performs the stopping condition of Mannor:
  // returns true(=1) if the absolute distance between upperQ and lowerQ is
      smaller than epsilon for all recognized state action pairs
  // returns false otherwise
  int t;
  double e;

  e = epsilon*(1-discount);
  e = e/2;
  t = 1;
  for(int i = 0; i < n; i++){
    for(int j =0; j < m; j++){
      if(recognizer[i][j] == 1){
        if(abs(upperQ[i][j] -lowerQ[i][j]) > e){
          t = 0;
          return t;
        }
      }
    }
  }
  return t;
}
```

Listing A.1: *mineAgentH.cpp*

```
//
//  CoffeeRecognizer.java
//  VISA
//
//  Created by Harry Seip on 30/8/06.
//  Copyright 2006 __MyCompanyName__. All rights reserved.
//
package visa;

import java.io.*;

public class CoffeeRecognizer extends Recognizer{

  public static final boolean DEBUG = false;
  public static final int NOTRECOGNIZED = 0;
  public static final int RECOGNIZED = 1;
  public static final int UNSEEN = 2;


    public static double ALPHA = 0.1; // learning rate
  public static double GAMMA = 0.9; // discount factor
  public static double RO    = 0.05;  // recognition treshold (must be
      overwritten)

  // Function Approximation should replace this for general appl.
  // phase out arrays
  protected int[][] actions;      // array that specifies which actions are
```

```
      recognized per s,a pair
  protected double[][] valuesU;   // array that holds the Q values of each s
      ,a pair
  protected double[][] valuesL;   // array that holds the Q values of each s
      ,a pair
  protected double[][] counts;    // array that holds the counts of each s,a
       pair

  protected int states;        // number of states in the statespace
  protected int maxactions;     // max number of actions per state

  protected double rewardOld;     // keeps track of reward at t-1 to use in
      Interval estimation updates

  // constructor:
  // creates a new recognizer that can recognize (max) n actions
  public CoffeeRecognizer(int s, int n){
    states = s;
    maxactions = n;

    actions = new int[states][maxactions];
    valuesU = new double[states][maxactions];
    valuesL = new double[states][maxactions];
    counts = new double[states][maxactions];

    for (int i = 0; i < states ; i++) {
      for (int j = 0; j < maxactions; j++) {
        actions[i][j] = RECOGNIZED;
        valuesU[i][j] = 10;
        valuesL[i][j] = -10;
        counts[i][j] = 1;
      }
    }
    rewardOld = 0.0;
    System.out.println("CoffeeRecognizer␣online");
  }

  // isRecognized:
  // checks if the action is recognized
  protected boolean isRecognized(HashInt s, int a){
    int index = parseState(s);
    return actions[index][a] == RECOGNIZED;
  }

  // update:
  // updates the value of the s,a pair
  public void update(HashInt s, HashInt sOld, int a, double reward){
    double delta;

    int index = parseState(s);
    int indexOld = parseState(sOld);

    double alp = 1.0/counts[indexOld][a];
    if (alp < 0.0001) alp = 0.0;

    delta = reward + GAMMA*valuesU[index][argmaxUpper(index)] + beta(counts[
        indexOld][a]);
    valuesU[indexOld][a] = valuesU[indexOld][a] + alp*delta;
```

```

      delta = reward + GAMMA*valuesL[index][argmaxLower(index)] + beta(
          counts[indexOld][a]);
    valuesL[indexOld][a] = valuesL[indexOld][a] + alp*delta;

    counts[indexOld][a]++;        // update
  }

  public double beta(double c)
  { // beta from Mannor the journal version
    double constant, y;

    double z = 1.0/c;
    if( z < 0.000001) z=0.0;
    y = beta_root(c) - (1-z)*beta_root(c-1);
    y = y*c; // *10;

    return y;
  }

  private double beta_root(double c){ // helper function that computes the
      sqrt part of the betafunction
    double constant = 0;
    double x = 0;

    if(c==0){
      return 0;
    }
    else {
      constant = 7218;  // constant = 4+*n*m/0.20;

      x = constant*c*c;
      x = Math.log(x);
      x = c*x;
      x = Math.sqrt(x);

      return x;
    }
  }

  // wrapper class for restrict that sets the treshhold
  // cuts away (unrecognizes) bad actions
  public void restrict(double t){
    RO = t;
    restrict();
  }

  // restrict:
  // cuts away (unrecognizes) bad actions

  public void restrict(){
    double c = 0.0;           // count the percentage of actions that are
        cut away
    for (int i = 0; i < states; i++) {
      for (int j = 0; j < maxactions ; j++) {
        if(valuesU[i][j]== 0.0){
          actions[i][j] = NOTRECOGNIZED;
          c++;
        }else if(valuesU[i][j] <= 12){
          actions[i][j] = NOTRECOGNIZED;
          c++;
        }
      }
    }
    c = c/256*100;
    System.out.println("Cut␣away␣" + c + "␣%␣of␣all␣s,a␣pairs");
    System.out.println(printRecognized());
    System.out.println(printUpper());
  }

  protected int argmaxUpper(int s){ //find the index of the maximal action,
      break ties randomly
    int bestIndex;
    double bestValue, value;
    int numBests = 1;
    bestIndex = 0;
    bestValue = valuesU[s][0];
    for(int i = 1 ; i < maxactions; i++){
            value = valuesU[s][i];
            if( value > bestValue ){
                bestValue = value;
                bestIndex = i;
                numBests = 1;
            }
            else if (value == bestValue) {
                numBests = numBests + 1;
                if( (int)(Math.random() % numBests) == 0 )
                    bestIndex = i;
            }
    }
    return bestIndex;
  }

  protected int argmaxLower(int s){ //find the index of the maximal action,
      break ties randomly
    int bestIndex;
    double bestValue, value;
    int numBests = 1;
    bestIndex = 0;
    bestValue = valuesL[s][0];

    for(int i = 1; i < maxactions; i++){

      value = valuesL[s][i];
            if( value > bestValue ){
                bestValue = value;
                bestIndex = i;
                numBests = 1;
            }
            else if (value == bestValue) {
                numBests = numBests + 1;
                if( (int)(Math.random()%numBests) == 0 )
                    bestIndex = i;
            }
    }
    return bestIndex;
  }

  // parseState:
  // helper function to convert the state representation into an enumeration
  // variable dependant and only handles binary variables now
  public int parseState(HashInt s){
    String b = "";
    for (int i = 0; i < s.table.length-1; i++) {
      b = b + s.get(i);
    }
    return Integer.parseInt(b,2);
  }

  // getStates:
  // returns the number of distinct states ( statespace)
  public int getStates(){
    return states;
  }

  // getMaxActions:
  // returns the number of distinct actions
  public int getMaxActions(){
    return maxactions;
  }

  // toString:
  public String toString(){
    String s = "CoffeeRecognizer␣with:␣";
    s = s + states + "␣states␣and:␣";
    s = s + maxactions + "␣actions.";
    return s;
  }

  // printRecognized: Prints the actions array
  public String printRecognized(){
    String s = "\n";
    for (int i = 0; i < states; i++) {
      s = s + "state:␣" + i + "\tact:␣";
      for (int j = 0; j < maxactions ; j++) {
        s = s + actions[i][j] + ",␣";
      }
      s = s + "\n";
    }
    return s;
  }

  // printRecognized: Prints the actions array
  public String printUpper(){
    String s = "\n";
    for (int i = 0; i < states; i++) {
      s = s + "state:␣" + i + "\tact:␣";
      for (int j = 0; j < maxactions ; j++) {
        s = s + valuesU[i][j] + ",␣";
      }
      s = s + "\n";
    }
    return s;
  }
```

```

}
```

Listing A.2: *CoffeeRecognizer.java*

Appendix B
Papers

B.1 Introduction

This appendix contains two papers that are the result of the research from this thesis and are submitted to two international conferences. Both papers are still under review and may not be published before the conference date. The two papers deal with the same subject. The ICML paper emphasizes the theoretical accomplishments and establishes the algorithm, while the UAI paper is more technical and emphasizes the empirical results.

The international Conference on Machine Learning (ICML) will be held in Corvalis, Oregon, USA and is the 24th annual conference of its kind. The conference serves as a presentation platform for novel research in all areas of machine learning.

Uncertainty in Artificial Intelligence(UAI) will be held in Vancouver, British Columbia, Canada and deals with new and unpublished research within artificial intelligence.

The first paper is the ICML paper, the second paper is the UAI paper. They are printed here in their original camera ready lay-out which may differ from the lay-out in the rest of the thesis due to style requirements set by the conference program committee.