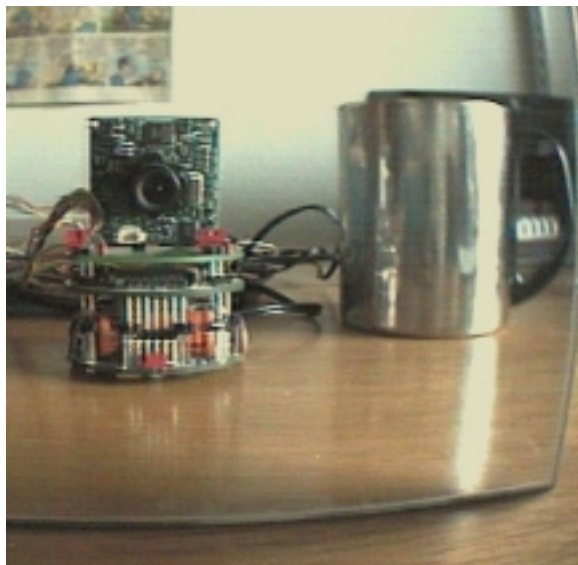

Reinforcement Learning on Real Robots

May 2000



by

Jesper Blynel

Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
8000 Aarhus C, Denmark

Abstract

The aim of this Master's Thesis is to investigate the usability of reinforcement learning in robotics. An investigation of different methods concludes that reinforcement learning is a promising method for visually guided robot control. The reinforcement learning methods of Q-learning, Dyna, and Prioritized Sweeping are compared on two different hardware setups. In the first setup the robot has to learn a task by the use of short-range proximity sensors, and in the second setup a long-range visual sensor is added. The conclusion is that it is possible to apply reinforcement learning to a real robot and make it learn a simple task within a few minutes. In addition, it is found that the method of Prioritized Sweeping performs better than Q-learning on both hardware setups.

Contents

Preface	vii
1 Introduction	1
2 Designing Autonomous Agents	3
2.1 Embodied Cognitive Science	3
2.2 The Fungus Eater Principle	5
2.3 Braitenberg Vehicles	7
2.4 Control Theory	8
2.5 Applications and Goals	10
3 Artificial Intelligence and Robotics	11
3.1 Classical AI and Robotics	11
3.1.1 The first Mobile Robots	12
3.2 Behaviour-Based Robotics	14
3.3 Evolutionary Robotics	15
3.3.1 Representation	16
3.3.2 Evaluation	17
3.3.3 Selection	17
3.3.4 Reproduction	18
3.3.5 Co-evolution	18
3.3.6 Evolution in Simulation	18
3.3.7 Evolution on Real Robots	20
3.3.8 Adding Vision to the Robots	20
3.3.9 Combining Evolution with Learning	21
3.4 Reinforcement Learning Robots	21
3.5 Discussion	22
4 Reinforcement Learning	25
4.1 Introduction	25
4.1.1 The Reinforcement Learning Model	26
4.1.2 Exploitation versus Exploration	27
4.1.3 Markov Decision Processes	28

4.2	Finding the Policy	29
4.3	Q-learning	30
4.4	The Dyna Algorithm	32
4.5	Prioritized Sweeping	33
4.6	Summary of the Algorithms	36
4.7	Putting RL onto Robots	37
4.7.1	The Markov Property	37
4.7.2	State-space and Action-space Construction	38
4.7.3	The Reward Function	38
4.7.4	Evaluation	38
4.8	Additional Issues in Reinforcement Learning	39
4.8.1	Partially Observable Markov Decision Processes	39
4.8.2	Eligibility Traces	39
4.8.3	Generalisation	39
5	Obstacle Avoidance Learning	41
5.1	Experimental Setup	41
5.1.1	The Agent	41
5.1.2	The Environment	42
5.1.3	The Task	42
5.1.4	Sensor-space Division	43
5.1.5	Action-set Construction	45
5.1.6	The Reward Function	45
5.1.7	Measures of Performance	48
5.1.8	Interfacing the Robot	49
5.2	Experiment 1: Setting the Learning Rate	49
5.2.1	Q-learning	51
5.2.2	Conclusion	52
5.3	Investigating the Effective State-space	52
5.3.1	Conclusion	54
5.4	Experiment 2: Different Timestep Lengths	54
5.4.1	Conclusion	55
5.5	Experiment 3: Effect of the Discount Factor	55
5.5.1	Conclusion	56
5.6	Experiment 4: Action Selection Methods	57
5.6.1	Softmax Action-selection	57
5.6.2	Optimism in the Face of Uncertainty	58
5.6.3	Results	59
5.6.4	Conclusion	60
5.7	Experiment 5: Model-based versus Model-free	60
5.7.1	Dyna	60
5.7.2	Prioritized Sweeping	61
5.7.3	Results	61

5.7.4	Conclusion	62
5.8	Experiment 6: Extending the Action-set	63
5.8.1	Results	64
5.8.2	Conclusion	65
5.9	Experiment 7: On-line Learning in a Dynamic Environment . . .	66
5.9.1	Setup	66
5.9.2	Results	67
5.9.3	Conclusion	68
6	Visually Guided Obstacle Avoidance	69
6.1	Experimental Setup	69
6.1.1	Adding a Camera	69
6.1.2	Image Processing	70
6.1.3	The Task	72
6.1.4	The Environment	72
6.1.5	Sensor-space Division	73
6.1.6	Action-set Construction	73
6.1.7	The Reward Function	73
6.1.8	A Rescue Behaviour	74
6.2	Experiment 8	74
6.2.1	Results	75
6.2.2	Conclusion	76
7	Final Discussion and Conclusion	79
7.1	Related Work	79
7.2	Discussion	81
7.3	Future Work	84
7.4	Conclusion	85
A	Notation	87
B	On-line Documentation	88

List of Figures

2.1	<i>Overview of the Framework for Embodied Cognitive Science.</i>	4
2.2	<i>The Nomad Robot in Antarctica.</i>	7
2.3	<i>Two Braitenberg Vehicles</i>	8
3.1	<i>The Classical Architecture of Functional Decomposition</i>	12
3.2	<i>Shakey the Robot.</i>	13
3.3	<i>The Behavioural Decomposition Architecture.</i>	14
3.4	<i>Charles Darwin.</i>	15
3.5	<i>An Evolutionary Algorithm.</i>	16
4.1	<i>A Reinforcement Learning Agent Acting in an Environment.</i>	26
4.2	<i>The Q-learning Algorithm.</i>	31
4.3	<i>The Model-based Learning / Planning Architecture used in Dyna.</i>	32
4.4	<i>The Dyna Algorithm.</i>	34
4.5	<i>The Prioritized Sweeping Architecture.</i>	35
4.6	<i>The Prioritized Sweeping Algorithm.</i>	36
4.7	<i>Summary of the Three Algorithms.</i>	37
5.1	<i>The Khepera Robot.</i>	42
5.2	<i>The Robot in the Field made of LEGO Duplo Bricks.</i>	43
5.3	<i>The action set.</i>	45
5.4	<i>KhepReal. The Graphical User Interface.</i>	50
5.5	<i>Q-learning: Different Learning Rates.</i>	50
5.6	<i>Sensor Activations in Different States</i>	52
5.7	<i>State Visit Frequencies.</i>	53
5.8	<i>Histogram of State Visit Frequencies.</i>	53
5.9	<i>Effect of the Timestep Length.</i>	54
5.10	<i>Effect of the Discount Factor.</i>	56
5.11	<i>A Part of a Q-table During an Experiment.</i>	58
5.12	<i>Softmax Action Selection Probabilities.</i>	58
5.13	<i>Performance of the Three Action Selection Methods.</i>	59
5.14	<i>Average Reward per Step in the Acting-phase of the Three Action Selection Methods.</i>	59
5.15	<i>Comparing the Different Learning Methods.</i>	61

5.16	<i>Average Reward per Step in the Three Learning Methods.</i>	62
5.17	<i>The extended Action-set.</i>	63
5.18	<i>Performance with the extended Action-set.</i>	64
5.19	<i>Average Reward per Step with the extended Action-set.</i>	65
5.20	<i>The Simplified Field.</i>	67
5.21	<i>On-line Learning.</i>	68
5.22	<i>On-line Learning.</i>	68
6.1	<i>The Khepera Robot with the Video Extension Module.</i>	70
6.2	<i>The Original and the Processed Video-image.</i>	72
6.3	<i>The Field for Visually Guided Obstacle Avoidance.</i>	72
6.4	<i>The Action-set set for the Visually Guided Task.</i>	73
6.5	<i>Performance in the Visually Guided Obstacle Avoidance Task.</i>	75
6.6	<i>Average Reward per Step in The Visually Guided Task.</i>	76
6.7	<i>Total Number of Bumps in each Method.</i>	76

Preface

My first introduction to the field of robotics was in the fall of 1997 when I attended the semester course “Adaptive Robots” by lecturer Henrik Hautop Lund. During the course different methodologies for controlling robots were introduced, and they were investigated both theoretically and practically. The course project was to programme an autonomous robot footballer and compete in the first *Danish Championship in Robot Football*. I entered the competition with the footballer *Static* (*Statisk Støj* in Danish) programmed in co-operation with Kasper Støy and Esben Østergaard and it went on to win the tournament. *Static* continued its ride of success and in the summer of 1998, it entered and won both the FIRA World Cup held in Paris and the Second Autonomous Robot Football Tournament held in London. The lessons learned during the work with *Static* was most of all insight in problematic issues of robot control. My first source of inspiration to the work done in this Master’s Thesis was a talk by Jeremy Wyatt entitled “Reinforcement Learning for Real Robots” given at the *Workshop on Artificial Life and Adaptive Robots* held in Aarhus in January of 1998. I immediately got fascinated by the idea of avoiding some of the problematic issues of evolutionary robotics by doing real-time learning on real robots.

Acknowledgements

I would like to thank my thesis supervisor Henrik Hautop Lund for introducing me to the field of robotics and guiding me through my work with this Master’s Thesis. I would also like to thank Kasper Støy and Esben Østergaard for the all fun we have had in pursuing the impossible goal of making a robot play football. If you had not convinced me to take “Adaptive Robots” course, I might have made a thesis in theoretical computer science and missed out on a lot of fun. I would also like to thank all the other people of *LEGO Lab* for making our robot study group a functional unit, both professionally and socially. Finally many thanks go to Christina Christensen for reading and correcting my written English. It was surely a big help.

Aarhus
May 2000

Jesper Blynel

Chapter 1

Introduction

The ultimate goal of research in robotics is to replicate human level intelligence on humanoid robots. At the same time a both fascinating and frightening thought if this goal is ever to be accomplished. Until now it has only been possible to recreate insect level intelligence in artificial creatures, and researcher are still struggling with seemingly basic tasks such as making two-legged robots walk. Robots having philosophical discussions or cracking jokes are still science fiction, and it is doubtful if that point is ever reached.

Robots behaving intelligently or in adaptive ways with respect to their surroundings will need to have some kinds of built in learning mechanisms. Instincts and reactive behaviours are not sufficient. The aim of this text is to investigate robot learning by the use of reinforcement learning. The basic idea of reinforcement learning is to tell a robotic agent when it is behaving *good* or *bad* and make it derive a suitable behaviour from these *reinforcement signals*. Recently, reinforcement learning has begun being used on simulated and real robots. This text will investigate difficulties in applying reinforcement learning to robots. In the spirit of embodied cognitive science, the investigations will include experiments on a real robot. Recently an understanding has emerged that it is not feasible to separately investigate the mind and body of humans, animal, or robots when the goal is to gain knowledge about intelligent behaviour. The things are interconnected and have to be treated as a whole.

The structure of the rest of this text is as follows. In chapter 2, an introduction to the fields autonomous agents, mobile robots and embodied cognitive science is given. Research goals, design principles and possible applications are described. Chapter 3 gives an overview of the history of artificial intelligence in robotics. Different approaches to the problem of designing control systems for mobile robots are presented and their strengths and weaknesses discussed. The methods included in the discussion are: traditional artificial intelligence, behaviour-based robotics, evolutionary robotics and learning robotics. On the basis of this discus-

sion, a deeper investigation of reinforcement learning is conducted in chapter 4. The formal theory of reinforcement learning is described with focus on the three learning methods Q-learning, Dyna, and Prioritized Sweeping. The chapter ends with an discussion of the usability of reinforcement learning for robot control. In chapter 5, the methods described in chapter 4 are implemented and used in a series of robot learning experiments. The experiments are made on a real Khepera robot using proximity sensors as input. The effect of the experimental setup and the setting of learning parameters is investigated and the three methods described in chapter 5 are compared. In chapter 6, a camera is added on top of the basic Khepera robot and the learning task is repeated with this new hardware setup. Finally in chapter 7, the text is concluded with a discussion of the results and an outline of possible future work.

Chapter 2

Designing Autonomous Agents

Traditional industrial robots as known from assembly lines of Japanese car factories, for instance, are to work in fixed and controlled environments doing high precision work on repeated predefined tasks of varying complexity. They move in fixed positions along axes and their movement is often a predefined motion sequence based on kinematic calculations. They are to work in highly shaped *closed* environments and their performance will fail if the boundary of this closed environment is broken either by a human control error or unexpected sensory signals [Gom98]. These industrial robots do not exhibit what can be referred to as intelligent behaviour, although some parts of them may have been developed using traditional artificial intelligence methods¹.

2.1 Embodied Cognitive Science

I want to study development, design, evolution and learning of mobile robots or autonomous agents as they are sometimes called. In contrast to the traditional industrial robots, these agents are to co-exist with humans in the world and are to be seen by those humans as intelligent beings in their own right [Bro91b]. This study falls into the field of embodied cognitive science. An overview of the framework for embodied cognitive science taken from [PS98] can be seen in figure 2.1. This field is interdisciplinary of nature, attracting researchers from biology, ethology, psychology, neuroscience, computer science, and engineering. These researchers often have different goals with their research [PS98]. I will follow the same goals as stated by Brooks in his investigations of so-called Creatures. He has put the following requirements on these Creatures:

- A Creature must cope appropriately and in a timely fashion with changes in its dynamic environment.

¹The difference between traditional and new methods in artificial intelligence will be explained in chapter 3.

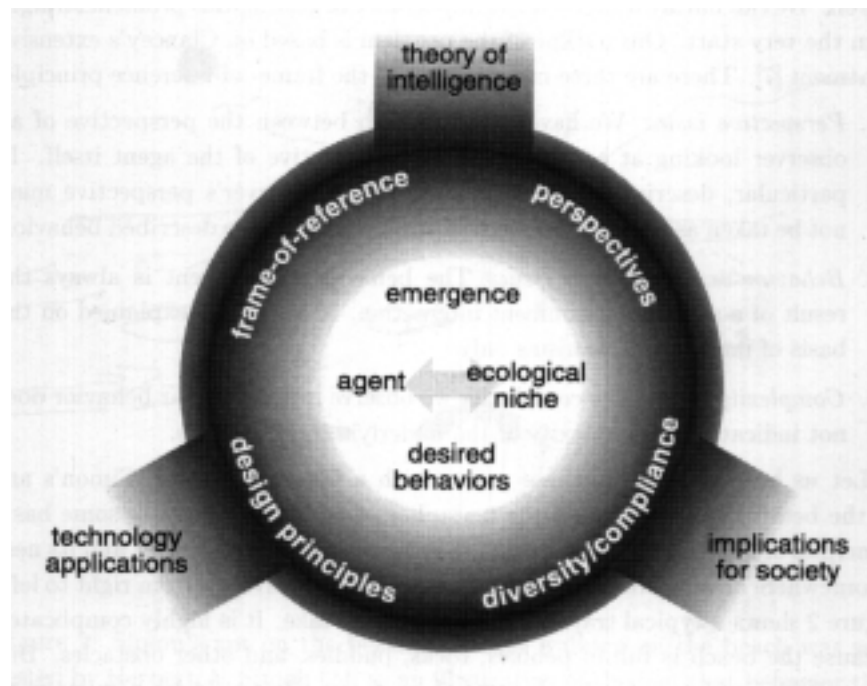


Figure 2.1: *Overview of the framework for embodied cognitive science.*

- A Creature should be robust with respect to its environment.
- A Creature should be able to maintain multiple goals and, depending on the circumstances it finds itself in, change which particular goal it is actively pursuing.
- A Creature should do *something* in the world; it should have some purpose in being.

The study of Creatures allows Brooks to take a computer scientific point of view on embodied cognitive science:

“I have no particular interest in demonstrating how human beings work, although humans like other animals, are interesting objects of study in this endeavour as they are successfully autonomous agents. I have no particular interest in applications. It seems clear to me that if my goals can be met then the range of applications for such Creatures will be limited only by our (or their) imagination. I have no particular interest in the philosophical implications of Creatures, although clearly there will be significant implications.” [Bro91b]

Brook focuses on four main aspects characterising his style of work in robotics and artificial intelligence:

- **Situatedness:** The robots are situated in the world - they do not deal with abstract descriptions.
- **Embodiment:** The robots have bodies and experience the world directly.
- **Intelligence:** They are observed to be intelligent - but the source of intelligence is not limited to just the computational engine.
- **Emergence:** The intelligence of the system emerges from system's interactions with the world and from sometimes indirect interactions between its components.

2.2 The Fungus Eater Principle

A “Fungus Eater”² is a complete autonomous creature sent to a distant planet to collect uranium ore. The concept of “Fungus Eaters” has been used by Rolf Pfeifer as a springboard to discuss issues in designing autonomous agents [Pfe96][PS98]. He proposes the following general design principles that have to be considered when designing autonomous agents or mobile robots:

- **The Complete Agent Principle:** The agents of interest (the “Fungus Eaters”) are *autonomous*, *self-sufficient*, *embodied* and *situated* in a physical environment. They have to be able to function without human intervention over extended periods of time. They have to be able to perform a set of tasks and at the same time maintain themselves, preventing a breakdown.
- **The Ecological Niche Principle:** The environment in which the agent has to operate in, the ecological niche, must be defined. There is no universality in the real world. The agents can exploit specific characteristics of their environment when solving a task. For example a robotic agent situated in office-environment can use the fact that floors of offices are flat, the so-called *ground plane constraint*, to estimate distance to objects based on their y-coordinate on the image-plane of a camera input [Hor93].
- **The Principle of Parallel, Loosely Coupled Processes:** This principle states that intelligence emerges from the agents' interactions with the environment. These interactions are based on loosely coupled processes run in parallel and connected directly to the sensory-motor apparatus of the agent. Seemingly complicated behaviour can be the effect of simple control rules acted out by an agent in an environment. This principle is inspired by the Braitenberg Vehicles [Bra84] and the subsumption architecture of Brooks [Bro86].

²Named after a creature invented by the Japanese psychologist Masanao Toda.

- **The Value Principle:** This principle states that the agent has to be embedded in a value system and that it must be based on self-supervised learning mechanisms. In other words, the agent has the means to “judge” what is good for it and what is not.
- **The Principle of Sensory-motor Coordination:** Interactions with the environment has to be conceived as a sensory-motor coordination. Active vision can be seen as an example of this principle. In active vision, instead of just using the visual input passively, the movement of the observer can be used in categorising objects and coordinating behaviours. Another example is a robot moving around a circular object maintaining the same distance to the object based on proximity sensor input. The wheel-velocities of the robot during this movement can be used as a measure of the size of the object.
- **The Principle of “Ecological Balance”:** This principle states that there has to be a balance between the “complexity” of the sensors and the actuators of the agent. When adding new sensors to an agent, new actuators should be added as well to maintain this balance. As an example Brooks has noted that “Elephants don’t play chess” [Bro90].
- **The Principle of Cheap Designs:** This principle states that good designs are “cheap”. If there are several models of design achieving the same performance, the simplest or “cheapest” one is considered the best. Complexity should be avoided if possible. Moreover, cheap means exploiting the constraints of the ecological niche.

The concept of “Fungus Eaters” has yet to be realised to its full extent. However, recently steps in the right direction have been taken. In January 2000, a Nomad robot from CMU succeeded in identifying and collecting a meteorite rock in Antarctica without human guidance. A picture of the Nomad robot can be seen in figure 2.2. This robot was by no means close to fulfilling all of the above-mentioned principles, but can be seen as a practical application of the somewhat artificial definitions.

If these design principles are kept in mind when designing an autonomous agent, some of the worst pitfalls can be avoided. In realistic applications, some of the principles have to be violated. A self-sufficient robot, that can detect sensor or motor breakdowns and repair itself, has yet to be seen in other places than the movies ³

It is good practice, however, to be aware of principle violations in order to keep

³In the movie RoboCop (1987) by director Paul Verhoeven, a cyborg policeman was able to repair “his” own arm.



Figure 2.2: *The Nomad robot in Antarctica.*

them to a minimum. In many real robotics applications, for example, power is supplied by a wire directly connected to the robot. This violates the principle of a “complete agent”, and battery problems are overlooked. Sensor and actuator characteristics are dependent of the energy-level of the power supplied. During my work with robot-footballers, I have experienced this problem. Often power was supplied to the robot by a wire during the development and testing of the control program. When this power-line was removed in a real match situation and the robots own batteries were used instead, the behaviour would be slightly different, showing the battery-level effect on sensing and actuation. Good arguments should be given when violating the principles, and it should be discussed how vital the violation is in practice.

2.3 Braitenberg Vehicles

The principle of parallel, loosely coupled processes resulting in emergent behaviour is elegantly described in the little book *Vehicles* by Valentino Braitenberg [Bra84]. This book contains a number of thought experiments where hypothetically self-operating machines exhibit increasingly complex behaviour on the basis of some very simple control rules.

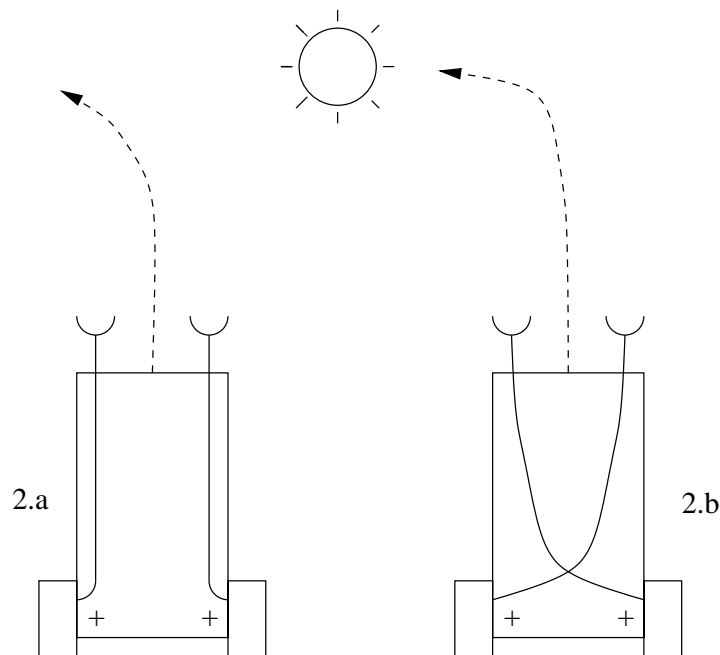


Figure 2.3: *Two Braitenberg Vehicles. Vehicle 2.a will drive away from the light source, and Vehicle 2.b with the connections crossed will drive towards it.*

An example taken from the book can be seen in figure 2.3. The vehicles in this figure are equipped with two sensors and two wheels each. The sensors can sense the lamp in the middle. The more a sensor senses, the faster the wheel goes. The effect of these simple internal mechanisms is that Vehicle 2.a where the sensor is connected to the wheel on the same side will drive away from the lamp. On the other hand if the sensors and the wheels are cross-connected the effect will be a motion towards the lamp as seen in Vehicle 2.b. Braitenberg calls vehicle 2.a for *Fear* and vehicle 2.b. for *Aggression*. In the eyes of an external observer, these are the behaviours the creatures seem to exhibit towards the light-source. An insect-like kind of intelligence. Originally the Braitenberg Vehicles were just science fiction, but with the development of the MIT Programmable Brick⁴ some of the creatures have been realised and seen to work in practice [HMR91].

2.4 Control Theory

Control theory is the theory of controlling dynamic processes. In the model of control theory, there is an agent (a control system) and an environment. The

⁴The MIT Programmable Brick is the predecessor of the LEGO Mindstorms RCX.

agent can sense and affect the environment using sensors and actuators. From a control theoretic point of view, the control of simple Braitenberg Vehicles is called *reactive control* [Mar94]. In a reactive control system there is a direct mapping between sensed input and affected output.

Control theory can roughly be divided into three different strategies [Hal][Smi90]:

- **Open Loop Control:** Some requirements are used as inputs to a decision procedure which selects which actions should be executed in the environment to achieve the required outcome. This strategy requires a model of action-outcome relationships in this particular environment.
- **Feedforward Control:** In a feedforward strategy, disturbances on the environment are measured and taken into account when selecting actions to achieve the required outcome. Again a model is required.
- **Feedback Control:** In feedback control, the disturbances on the environment are not measured, but rather the outcome of these disturbances. The outcome is compared to the requirements and the deviation or error is used as input to the decision procedure.

The feedback approach resembles the reactive control described above. Here are two examples illustrating the different approaches:

Example 1: Controlling room temperature. In an open loop strategy, the laws of thermodynamics would be used as a model to calculate how long a heater or a cooler should be turned on to achieve the required room temperature. The feedforward approach would be to measure the temperature outside the room and include this disturbance in the calculations. In the feedback control-strategy a temperature feeler would be placed inside the room. It would compare the actual temperature with the required temperature and turn on a heater or a cooler depending on the sign of the temperature difference.

A robotic control example:

Example 2: A robotic footballer that has to approach a ball. In the open loop strategy the distance and direction to the ball would be used to calculate a trajectory from the robot to the ball. The motor commands that would move the robot along this trajectory would be calculated. A model speed and acceleration of the robot motors and wheels would be included in Newtonian mechanical calculations. In a feedforward approach the friction between the wheels and floor would be measured and taken into account. In a reactive or feedback strategy, the current direction to the ball would be used to set the motor speeds. If the ball is to the left, the speed of the right motor is set a bit higher than the

speed of the left motor and vice versa.

The best control-strategy depends on the application. If a perfect model of agent and environment dynamics is given, and no external disturbances are present, the open loop method is usable. This is *not* the case in most robotic applications so here the feedback to close the loop is usually needed.

2.5 Applications and Goals

What are the long term goals in this effort to design autonomous agents? Brooks has changed his view a bit, not just focusing on Creatures, and started a humanoid project. In “The Cog Project”, as it is called, the aim is to build robots that can interact with humans. In the RoboCup challenge [KAK⁺98], a standard problem in robotics has been proposed. The project is to develop a robot soccer team that plays against human players and wins. The hope is that research in this landmark project will spin off a lot of useful technologies. Examples of current useful applications of autonomous agents include intelligent wheelchairs [GG98], space exploration robots, cleaning robots, land mine detection robots, and transportation robots for construction sites [Gom98].

Chapter 3

Artificial Intelligence and Robotics

In this chapter, an overview will be given of the history of artificial intelligence (AI) and robotics. Starting in traditional AI, the chapter moves on to Brooks *subsumption architecture* and behaviour-based robotics. Then the inspiration from Darwinian evolution that forms the base for evolutionary robotics will be described. Different approaches and design choices in evolutionary robotics will be investigated, including a discussion on the use of simulated robots. Finally, learning robots and especially reinforcement learning (RL) will be presented and the chapter ends with a discussion on usability of the different methods in the design of complex control systems.

3.1 Classical AI and Robotics

In classical artificial intelligence, the problems that were considered were logical problems that had natural symbolic representation. Examples of these are the geometric problems that appear in IQ tests, block worlds where symbolically represented objects can be manipulated, or game playing problems like checkers and chess. These are problems that are considered hard by humans to solve and that may be the reason why the computer programs, that did solve the problems, were considered intelligent. A nice example of the early AI work is the checker player developed by Arthur Samuel in 1952. By playing against itself, the program became better to play checker than Samuel himself and in fact it eventually learned to play tournament-level checkers [RN95].

Based on this and other initial successes of AI, the same methods were applied to a wide range of problems. For example, it was assumed that, problems of language translation could be solved in the same way. Grammars and the dictionaries of different languages were given as input, and the systems were to apply simple word substitutions in order to translate a text. A classical example of the failure of this approach was the machine translation project of Russian

scientific texts by the U.S. Government after the first Russian Sputnik launch in 1957. A historical example of the problems with automatic machine translation is the translation of the sentence “The spirit is willing, but the flesh is weak?”. The result of the translation into Russian and back was: “The vodka is good, but the meat is rotten”. Failures of this kind resulted in fundings cancellation for the machine translation project[RN95], and in general the initial expectations of the possible applications of artificial intelligence were lowered to a more realistic level.

3.1.1 The first Mobile Robots

The first attempts to build autonomous mobile robots were based on classical artificial intelligence research. A main assumption in this early robotic work was that the robot mind (the software) and the robot body (the hardware) could be developed independently of each other. The development of the hardware was considered an engineering problem and the focus of interest was on manipulating some symbolic representation of the world in the mind of the robot in the same way as it was done in classical AI. The whole system was thought of as an information processing system and the flow of control followed the cycle referred to as the *sense-think-act* cycle [MSH89]. The architecture of the robots had a vertical structure as can be seen in figure 3.1.

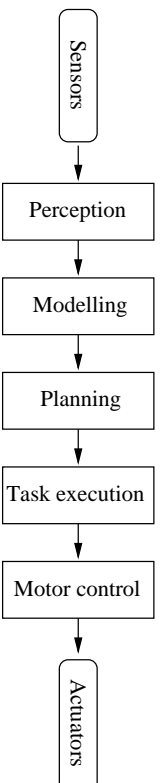


Figure 3.1: *The classical architecture of functional decomposition* [Bro86]

The sensory input, for example from a camera, was fed into the first functional unit, in this case the perception unit. Here relevant image features such position and orientation of objects in the image were extracted. These image features were then used to update the robot’s internal model of the world surrounding it, and this internal model was used to plan the future path of the robot in the planning unit. This path was then converted into the motor commands which were finally send to the motors, controlling the wheels of the robot.

One of the first examples of a mobile robot using this overall architecture was the robot *Shakey* developed at Stanford from 1966 to 1972 (see figure 3.2). Shakey had bump sensors and a TV-camera and was connected to two computers via radio and video links.

A serious limitation to the usability of the *sense-think-act* cycle methodology and the functional decomposition architecture is that the cycle must be fast enough to pre-empt significant changes in the world while doing the thinking [MSH89]. Furthermore in a physical world both sensing and acting will be imperfect at some level and thus introducing many additional problems:

- It turned out that the computer vision problem was much harder than first thought. It became clear that huge amounts of computing power were needed in order to extract relevant image features, and still the result was error-prone.
- It is difficult to maintain a world-model based on these imperfect informations from the sensing modules.
- Planning is time consuming even in perfect block-worlds. In real-time applications, a plan may be obsolete, when it has been computed because of the state changes that took place during the computation.

Although robots like Shakey did work at some level, the overall result was that the robots either had to stop and think in every cycle or that they would try to act out a plan that would often fail due to inaccurate acting abilities or errors in the internal world model.

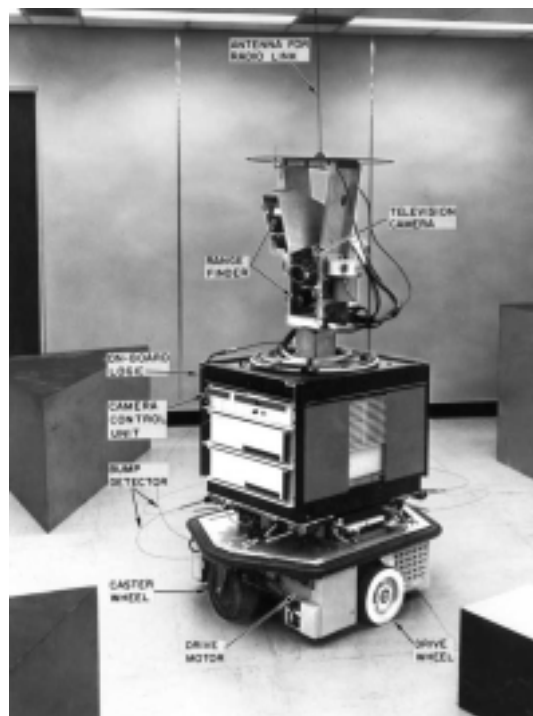


Figure 3.2: *Shakey the Robot.*

3.2 Behaviour-Based Robotics

Due to dissatisfaction with the traditional AI way of attacking robot problems by symbolic representation and manipulation, a new paradigm in robotics began to emerge in the 80s. This new approach was called *Behaviour-Based Robotics*, and one of its main advocates was Rodney Brooks from MIT. He argued that the old way of separate development of the mind and the body of the robots was not ideal, and he focused his research on embodied and situated physical robots where real world factors such as inaccurate sensing and acting have to be taken into account. The rather surprising and controversial conclusion of Brooks was that there need not be a representation of the outside world in the mind of the robot [Bro91b]. On the basis of this, he proposed a new robot architecture where the robot controller is divided into layered behaviours as can be seen in figure 3.3.

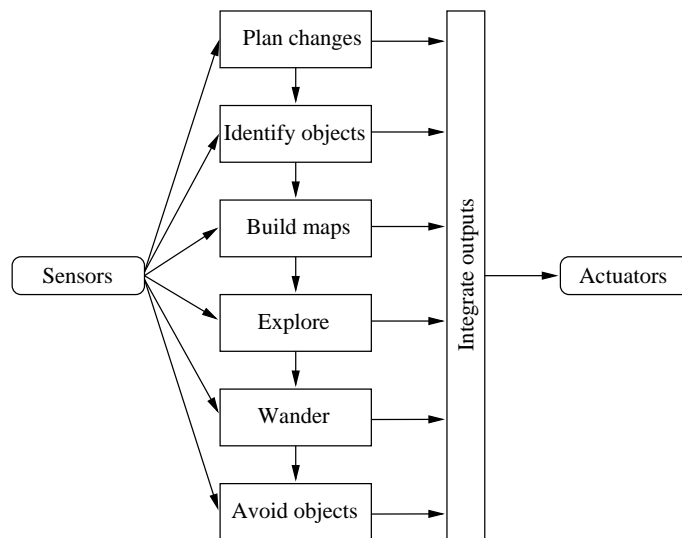


Figure 3.3: *The behavioural decomposition architecture* [Bro86].

All the layers of control are run in parallel and each layer can access the sensors reading and affect the actuators. The idea behind this layered structure is that the robot can try to maintain several goals simultaneously. A high-level goal, for example of reaching a certain place can sometimes be conflicting with the low level goal of staying clear of obstacles. The higher levels can examine and alter the data-flow of the lower levels, the lower levels being unaware of this. Brooks calls this architecture the *subsumption architecture* because the higher levels can *subsume* the lower levels in this way.

One of the issues, that Brooks has stressed in favour of his Behaviour-based

controllers, is the incremental nature of the design. The lowest levels can be implemented at first accounting for a simple behaviour. Then more diversity can be introduced by adding more layers. This design process has been called a “recapitulation of evolution”. However, one remaining problem with this design method is that a human designer is still needed. As put in [HHC⁺97]: “Each new layer is wired by hand design; and despite the heuristics used to minimise interactions between layers, it seems that unpredictable interactions become insuperable when the number of layers gets much beyond single figures”. It seems what is needed is some kind of method to automate the design process.

3.3 Evolutionary Robotics

In the Braitenberg book “Vehicles” mentioned in chapter 2 Vehicle 6 is of some interest at this point. When introducing Vehicle 6 Braitenberg says, “things get slightly out of hand”. A collection of the previous vehicles is put onto a large table together with some sources of light, sound and smell. Whenever a vehicle falls off the table, one of the remaining vehicles gets picked up and copied. In the copying process some of the wires get twisted and interchanged, and the overall architecture is altered a bit. In some cases the result will be malfunction of the new vehicle, but in other cases the copy will survive longer on the table than the original. The idea behind this process is the same that underlies evolutionary algorithms.

Evolutionary algorithms (EA) is a way of solving problems inspired by the way that nature successfully evolves robust organisms. The source of inspiration is the theory of Darwinian evolution originally proposed in the publication *On the Origin of Species* [Darwin 1859]. The idea in EA is to have a population or pool of individuals and to apply selection and reproduction to the population in order to evolve individuals that are successful with respect to some *fitness function*. When evolutionary algorithms are applied to the problem of developing robot controllers (and sometimes robot morphology), the method is called evolutionary robotics (ER). Figure 3.5 shows a skeleton of an evolutionary algorithm. There are a number of decisions that have to be made when using evolutionary algorithms. Each of the steps in the generic algorithm has to be specified.



Figure 3.4: *Charles Darwin.*

1. Initialise the population to a set of random individuals
2. Repeat for a number of generation:
 - Evaluate each individual using some fitness function
 - Select the parents on the basis of the evaluation-scores
 - Reproduce the parents to make the next population of individuals
 - Replace the old individuals with the newly produced ones

Figure 3.5: *An evolutionary algorithm.*

3.3.1 Representation

How are the individuals to be evolved (the robot controllers) represented? There are several possible options here.

Genetic Programming

In the genetic programming (GP) community, the individuals stored are high level programs such as LISP-code [KR92]. Brooks has used genetic programming to evolve programs in the behavioural language (BL) also used when implementing the subsumption architecture [Bro91a]. The main arguments against this approach is that when choosing the semantics of the high-level language used in GP, the human designer's prejudices are incorporated within their choice. It has been argued that the primitives manipulated by the evolutionary process should be at the lowest level possible, in contrast to the use of high-level languages [HHC92].

Artificial Neural Networks

In most work done in evolutionary robotics, artificial neural networks (NN) have been used to store the controller. The original motivation for research in neural computation was to model real neurons in the human brain. However, the models commonly used are extremely simple from a neurophysiological point of view [HKP91]. The encoding of the *phenotype* (the neural network) into the *genotype* manipulated by the evolutionary algorithm is usually just a bitstring. If the size of the network is fixed, *direct encoding* can be used. In simple feedforward networks, there are direct connections between the input neurons and output neurons, resulting in a *reactive* controller. Some experiments have shown that

the use of simple representation still can result in rather complex task solving [LH96][MLN95]. In other experiments the controllers are allowed to have recurrent connections [MDTP98][Nol97]. The effect of recurrent connections is that the network can remember past activations of certain neurons; a kind of memory. The Sussex approach has been to allow evolution of the physical structure of the network as well to avoid putting any restrictions on the evolutionary process [HHC⁺97].

3.3.2 Evaluation

When the initial population of random individuals has been generated, they are allowed to act out their behaviour either in a simulator or on a real robot. The metric used for evaluating the behaviour is called the *fitness function*. This function is applied to the behaviour of each individual, resulting in an *evaluation value* telling how *fit* the corresponding individual is. Hopefully, there is a correspondence between the evaluation value and the distance to the desired behaviour; the task the designer had in mind. An example of a fitness function reported in [FM96a] to evolve the behaviour of navigating while staying clear of obstacles is:

$$\Phi = V(1 - \sqrt{\Delta v})(1 - i), \quad (3.1)$$

where V is the average rotation of the two wheels, Δv is the difference of the two wheel-speeds, and i is the activation of the proximity sensor with the highest activation. If any of the terms were left out, the behaviour would differ from the desired behaviour. Often the designers' first attempt to construct the fitness function results in an unpredicted behaviour and several iterations are therefore needed. Unfortunately, most of this insight is lost since typically only the best fitness function is reported, but not the complex process that generated it [MC96].

3.3.3 Selection

The next step is to select the individuals that form the basis of the next generation of individuals. In *rank based selection*, the individuals are sorted on the basis of their fitness evaluation value. Their position in this ranking is then used to form their probability of being selected for reproduction. In *roulette wheel selection*, there is a linear correspondence between the evaluation value and the selection probability of an individual. In *tournament selection*, the individuals are compared one-on-one¹ and the winners of the tournaments are selected.

¹Like in football cup tournaments.

3.3.4 Reproduction

When the parents for the next generation have been selected, the new individuals have to be produced. For this purpose the two genetic operators *mutation* and *crossover* are used. The mutation operator makes mutations to a single individual producing a new individual. If the individuals are encoded as bitstrings for instance, a certain percentage of the bits is flipped by the mutation operator. This percentage is called the *mutation rate*. The crossover operator takes two individuals as input and produces two new individuals by crossing the two genotypes. If the genotypes are bitstrings, the two bitstrings are cut at some point (the crossover point) and parts from the two bitstrings are “glued” together. In GP implementing the crossover operator is even more easy. Pointers to subtrees of the high-level programs are just interchanged.

The job for the designer of the evolutionary algorithm is to choose appropriate mutation and crossover rates to ensure a search for good individuals through the fitness landscape. If the rates are too high, there will be no resemblance between parents and children and the algorithm has degenerated into a random search strategy. On the other hand if the rates are too low, progress is very slow because parents and children are almost identical.

3.3.5 Co-evolution

In [NF98] the effect of *competitive co-evolution* was investigated on a predator-prey task evolved in simulation and tested on real robots. The starting assumption in *co-evolutionary* experiments is that when two separate populations compete on a task they will push each other’s performance out of local minima solutions by dynamically altering the *fitness-landscape*.

3.3.6 Evolution in Simulation

In most evolutionary robotics research, the experiments are carried out using a simulator. This is somewhat in conflict with the *complete agent principle* of Pfeifer described in chapter 2. The main reason for the necessity of using simulations in ER is the huge amounts of experiments generally needed when applying an evolutionary algorithm to a problem. If the experiments are run on the physical robot, they have to be done real-time which can be problematic. A calculation underlining this fact is given in [LH96]. Some generally used settings of population size and evaluation time of each individual form the basis and the result is that a single experiment could easily take 17 days when performed on a real robot, and half a year with repeated testing for valid results. Although some of the parameters can be changed in order to reduce the overall time requirements, the argument is that this is an infeasible way to proceed and a simulator,

therefore, has to be used.

The Reality Gap

If something has to be said about on-line performance of the evolved controllers, the controllers have to be moved out on the physical robot at some point. The behaviour when moving a controller to the physical robot is often quite different from the behaviour observed in the simulator. This discrepancy of the observed behaviour is referred to as the *reality gap* [JHH95]. Several ways of constructing the simulator in order to reduce the reality gap have been proposed. One method is to construct a simulator where the characteristics of the sensors, the actuators and the environment are based on real world measurement stored in a table. This approach has been taken in [LH96][MLN95] to build a simulator for a Khepera robot. In Sussex the approach has been to use a mathematical description of real world physics in order to build a simulator. Parameters of this model were set using empirical information from real robot measurements[HHC⁺97]. Usually careful modelling of the sensors and actuator is not enough in order to “cross the gap”. Something else has to be done. The approach often taken is to add noise to the simulations in order to mirror the inaccurate sensing and acting of any physical robot. Discrepancies between the simulations and the real world, as long as they are not too big, can be treated as noise; the system can adapt to cope with this [HHC92]. In [MLN95] the effect of adding different kinds of noise has been investigated. The best method found was to add “conservative noise” where random replacements of the robot were used when calculating the sensor activations. The robot did not move, but sensed the world as if it had moved. The method outperformed adding random noise to the sensor-readings. In these experiments the first 200 generations of the evolution were run in the simulator and the last 20 on the real robot.

Minimal Simulation

A slightly different approach to the problem of transferring the robot controller from a simulated to a real robot has been taken by Nick Jakobi [Jak98]. In his *minimal simulation* methodology, he introduces the concept of a *base set*, which is the set of environmental features the robot needs to know about in order to perform its task. The rest of the environment inputs to the robot, he calls *implementation aspects*. The goal he has in mind, when building his simulator, is to make evolved controllers:

- **Base set exclusive:** The robot should only use features from the base set when performing its task.
- **Base set robust:** The robot should be robust to the differences in the base set features when moving from the simulated to the real world.

The idea when building the simulator is to add huge amounts of noise to everything else than the base set features, so the robot controller does not take advantage of implementation aspects of the simulator not present in the real world, making the evolved controllers *base set exclusive*. In addition, random noise is added to the base set features in order to make the controllers *base set robust*.

3.3.7 Evolution on Real Robots

Despite of the various experiments where robot controllers evolved in simulators have been successfully transferred to real robots, some researcher still feel that the use of simulations should be avoided in robotics. Dario Floreano et. al. has conducted a series of ER experiments where the entire evolutionary process has taken place real-time on a Khepera robot. In [FM96a] navigating clear of obstacles and homing behaviours were evolved. The navigation behaviour was evolved over 100 generations each taking 40 minutes, in all over 66 hours' experiment time. It is doubtful that this method will scale to more complex problems. As pointed out by Mataric [MC96], the main problems with on-line evolutions are:

- **Real Time on Real Hardware:** Evolution on physical systems takes prohibitively long.
- **Battery Lifetime:** The need to recharge robot batteries further slows down the experimental procedure².
- **Robot Lifetime:** The physical hardware of a robotic system cannot survive the necessary continuous testing without constant maintenance and repair.

3.3.8 Adding Vision to the Robots

When the goal is to evolve more complex robot behaviours in the future there is no doubt that some visual sensors have to added to the robots. Not much work has been done in this field within the evolutionary robotics community. Most commonly only proximal sensing devices such as whiskers, bumpers, infrared light or sonar ultrasound depth sensors are used. They only provide reliable data for the immediate surroundings of the robot. Such robots are thus forced to employ primitive navigation strategies such as wall-following[HHC92].

One example of evolution with real vision is the use of the specially developed gantry-robot in Sussex [HHC94]. A 64x64 pixels monochrome CCD camera has

²On the Khepera platform aerial lightweight cable is usually used for continuous power supply, but this solution does not generally apply to other platforms or to multi-robot setups.

been subsampled and used as input for an evolutionary algorithm for the task of moving towards stationary and moving targets or varying size. It is difficult and computationally heavy to make realistic simulations of visual input. A few attempts have been made by applying the minimal simulation approach described earlier. In the [Smi98], the transfer from simulation to reality of a robot football behaviour, using visual input, is described.

3.3.9 Combining Evolution with Learning

The general approach in evolutionary robotics has been to evolve the robot behaviours at the population level. This kind of *phylogenetic* evolution results in controllers with some innate behaviours that does not change during the lifetime of a single individual. However, a few attempts have been made to combine this kind of evolution with *ontogenetic learning*. In [FM96b], the single individual is allowed to change the synaptic weights of its neural controller during each run. The idea behind combining evolution and learning is that robot controllers made by pure evolutionary methods are not robust to environmental changes other than those present during the evolutionary process. The combination with learning methods tries to fix this problem.

3.4 Reinforcement Learning Robots

The combination of evolution and learning just described is the exception. Most commonly only one of the methods is used. The fundamental difference between evolution and learning is the different timescales they act on. Evolutionary methods evolve robot controllers at the population level whereas learning methods learn controllers at the individual level. In evolutionary methods, the bad controllers do not survive to the next generation. In learning methods, the hope is that every individual learns to perform well during its lifetime. In [KLM96] evolutionary methods are described as methods that search in the space of behaviours in order to find one that performs well in the environment. In contrast learning methods are described as statistical techniques and dynamic programming methods to estimate the utility of taking action in states of the world. When learning methods are applied to mobile robots, most commonly reinforcement learning (RL) is used. The basic assumption in learning robots is that nothing about the world is known in advance. In the machine learning community, supervised learning methods are often applied. In supervised learning, the right answer is given whenever the learning algorithm has made a decision. The learning then tries to adjust the future decision accordingly. In the development of autonomous robot controllers, the “correct” behaviour is not known in advance, so self-supervised learning methods like reinforcement learning have to be used. In reinforcement learning, an agent acting in an environment is given rewards and punishments

on the basis of its decisions. The job of the agent is to adjust its action-selection policy in a way that maximised the reward received.

3.5 Discussion

In this chapter four different methods for designing autonomous agents have been presented. The four methods are: *traditional AI methods*, *behaviour-based methods*, *evolutionary methods* and *reinforcement learning methods*. All of them have their strengths and weaknesses.

The *traditional AI methods* are applicable in controlled environments where the setup is fixed. When this is the case, a sufficient reliable model of the environment can be maintained inside in control program and the affected actions will have the predicted results. This approach to robotics has successfully been used in assembly line robots, but in autonomous agent problems the environment cannot in general be controlled to such an extent as to make the internal model reliable. The result is a break down of the method in this case.

The *behaviour-based methods* have been implemented with success on a variety of platforms and tasks. An example is the robot Polly [Hor93] which gave visitors at the MIT AI Lab tours in an unmodified office environment. Another application of a behaviour-based architecture is the intelligent wheelchairs described in [GG98]. In section 3.2 however, it was argued, that if the methods have to scale to even more complex applications, an automatic development of the controllers is necessary. This demand is not fulfilled in the standard behaviour-based framework, where the individual layers of the architectures are hand-coded.

In section 3.3 evolutionary approaches to robotics were described in some detail. In addition to allowing an automatic development of controllers, an advantage of evolution is that the evolutionary process may find very different and often better solutions to a problem than can be found by a human designer. An example of this can be seen in [Nol96]. Human designers are likely to use fixed thresholds when designing a control program, making the resulting controller vulnerable to changes in the environmental conditions and vulnerable to different characteristics of physical sensors and actuators. The control program may not work anymore when a sensor is replaced. A major strength of the evolutionary process is its ability to make the robot controllers *robust* with respect to these disturbances. This robustness can be achieved by exposing the robots to disturbances during the evolutionary process. On the downside is that the evolutionary robots developed so far most often have been evolved in a simulator, introducing the problem of *the reality gap* described in section 3.3.6. Recall that the use of simulators is a choice of necessity, because *on-line* evolution is very time consum-

ing. Furthermore in most work where simulators has been used, the robots have had a limited range of sensor inputs available. Short-range proximity-sensors and bumpers are most often used in stead of long range visual sensors like a camera. This reduced sensing-ability limits the number of possible navigation strategies for the process to evolve to. Often the tasks solved by these robots have been designed with specific robots or simulators in mind. The hard problem of simulating visual input has often been overlooked, and none of the robots have been evolved to work in unstructured environments like the above-mentioned Polly. Another drawback is that although the robots can adopt their behaviours to environmental changes on a long time-scale, the single individual is fixed and changes in the environment during the lifetime of an agent cannot be coped with.

The fourth method mentioned for developing autonomous agents was *reinforcement learning*. Reinforcement learning is, like evolution, an automatic method of developing controllers. The individual robot incrementally improves its behaviours in order to find the best strategy (or policy as it is called in RL methodology) to solve the task in question. In others words learning is done during the lifetime of the single agents, which makes is possible to adapt to changing environments on a shorter time-scale that in the evolutionary approach. Another feature of reinforcement learning is that when learning is done on a single individual, it is less time-consuming than evolution of a whole population, and therefore easier to apply *on-line* to physical robots. When learning in done on a real robot, the *reality gap* is bypassed and real visual sensors can be used. On the downside of reinforcement learning is that the single individual starts from scratch with a random behaviour which sometimes results in convergence problems, especially if the task is defined by very delayed rewards. Still, as with the evolutionary methods, automatic learning of complex behaviours in unstructured environments without explicit shaping of the problem is yet to be seen, but the problems encountered are slightly different than those encountered with the evolutionary approach.

On basis of the above discussion, the decision was to go on with the reinforcement learning approach. Looking back at the design principles described in chapter 2, reinforcement learning is closer to fulfilling *the complete agent principle* and *the value principle* than both behaviour-based and evolutionary approached to robotics. The method is automatic when the initial setup is done, and it is not necessary to use the simulators which have introduced a lot of problems in evolutionary robotics. This fact makes reinforcement learning a promising candidate for designing visually guided autonomous agents that are to perform their tasks in realistic environments. An important question, however, is how well the method will scale with increase in complexity of the task and the environment.

Chapter 4

Reinforcement Learning

In this chapter an introduction the theory of reinforcement learning will be given. The standard reinforcement learning model has a well founded mathematical background, but relies upon some assumptions that are seldom or never fulfilled in mobile robot applications. The emphasis will be on the three reinforcement learning methods *Q-learning*, *Dyna* and *Prioritized Sweeping* that will be used later on in some real robot¹ experiment.

4.1 Introduction

In short Reinforcement learning is a trial-and-error approach to learning in which an agent operating in an environment learns how to achieve a task in that environment [Wya95]. Original reinforcement learning came out of the *machine learning* community, but where machine learning normally is a sort of *supervised learning*, reinforcement learning is not. In supervised learning methods, whenever an action is taken, the learner tells the agent afterwards what the best action would have been. The agent can then adjust its behaviour according to this supervision. For example, if the behaviour is represented in an artificial neural network, the agent can adjust the weights of the network by back-propagating the correct answer. In contrast to this, the reinforcement learning agent is just given a scalar reinforcement signal as a response to its action, but no clue on what the best action would have been. It has to find a good strategy for increasing the sum of the reinforcement signals by *trial-and-error* interactions with the environment. That is why ordinary machine learning sometimes is referred to as *learning with a teacher* whereas reinforcement learning is characterised as *learning with a critic*. This important distinction indicates that reinforcement learning is well-suited for problems where the correct behaviour of the agent is not known in advance, but can be derived from the *trial-and-error* interactions

¹The term *real robot* is used to in order to emphasise the difference between a real physical robot and a simulated robot.

with a perhaps dynamic environment. Many robotic control problems certainly fall into this category.

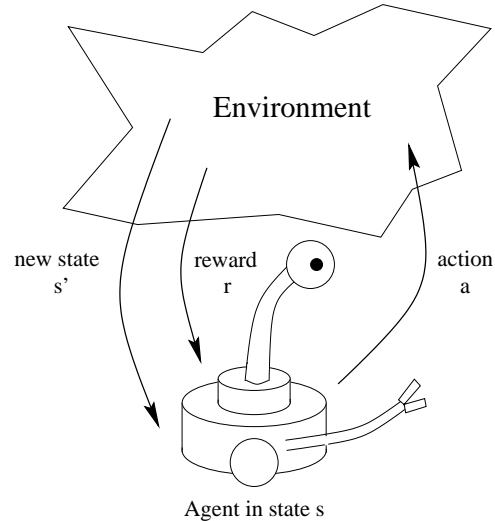


Figure 4.1: A reinforcement learning agent acting in an environment.

4.1.1 The Reinforcement Learning Model

In the standard reinforcement learning model, there is an agent connected to an environment through perception and action. On each iteration (or step) the agent perceives its state s in the environment, performs an action a and receives a reward r , which is a scalar reinforcement signal. In figure 4.1, there is drawing illustrating the basic reinforcement learning model. From the agents' point of view the reward is thought of as a part of the environment.

The problem faced by the agent is to choose the actions that tend to increase the long-run sum of the reinforcement signals. In other words, the agent has to find a policy π , mapping states S to actions A , that maximises some measure of the long-run reinforcement. In general the policy can be stochastic. The model of optimal behaviour that is most widely used is the *infinite horizon discounted model*, where the measure that has to be maximised is the *expected discounted reward*:

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \quad (4.1)$$

Here t is the time-step, r_t is the reward received in time-step t and $\gamma \in [0; 1[$ is the *discount factor*. This discount factor bounds the infinite sum and its size

describes the balance between the value of immediate and future rewards. The higher the γ -value is, the more importance is put on the future rewards of the agent.

Other models of optimal behaviour exist, including the *finite horizon*, *receding horizon* and *average reward* methods. One reason that the *infinite horizon* model is so widely used is that it is more mathematically tractable than some of the other methods. Some authors feel that the *average reward* criterion is closer to the true problem they wish to solve, so some work has been done in comparing the different methods [Mah94].

If it has to make sense to talk about the *expected discounted reward* of performing a given policy, it has to be assumed that the environment is stationary, i.e. state transition probabilities do not change over time. Later on, the implications of this assumptions in connection with reinforcement learning on robots, will be returned to.

4.1.2 Exploitation versus Exploration

The way that the actions of the agent are chosen can have a very great impact on the overall success of the learning-algorithms. At first glance it seems to be the best idea to always choose the action that is currently assumed to give the highest payoff. This strategy is called a *greedy* strategy for obvious reasons. The problem with this strategy is that generally, the model of the world is not known in advance, but has to be derived from the interactions with the environment. In addition always taking the action with the highest estimated payoff can indeed be a bad idea when this estimate is based on insufficient statistics on rewards and state values in different parts of the state-space. The result can be that the *truly* best action in a state is starved out because another supposedly best action always is chosen when visiting that state. On the other hand taking a non-optimal action reduces the immediate payoff, but does ensure exploration of unvisited parts of the state-action space. This trade-off is known as the *exploitation versus exploration* dilemma.

The Epsilon-greedy Action Selection Method

There exist some ad-hoc heuristics for dealing with this dilemma, but none of them have shown to be the best in all cases. The most simple one is the ϵ – *greedy* action selecting method [SB98]. In this method the current *best* action is chosen with probability $1 - \epsilon$ (a greedy step) and a random action is chosen with probability ϵ (an explorative step). This method is widely used because of its simplicity and often ensures a sufficient exploitation / exploration balance. The main drawback of this method is that obviously hopeless actions are chosen

again and again with the same probability as the more promising alternatives.

Softmax Action Selection

A method that tries to fix the above-mentioned problem is the *softmax* action selection method [SB98]. Softmax is sometimes referred to as *Boltzmann exploration* because it usually uses a Boltzmann-distribution for selecting actions. The action a in state s is selected with probability:

$$Pr(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \quad (4.2)$$

The term $Q(s, a)$ is a value function that assigns values to the different actions of a state (see section 4.3 for a definition), and τ is the Boltzmann-temperature. The temperature is usually decreased over time to decrease exploration when better estimates of the values are found. To select an appropriate interval for the temperature-parameter can be tricky, and it is not intuitively clear to see the effect of a specific temperature setting because it depends on the distribution of the Q-values.

Optimism in the Face of Uncertainty

A third useful heuristic for selecting actions is the one known as *optimism in the face of uncertainty* [KLM96]. Here a greedy selection method is used, but with high initial estimates of the payoffs. This ensures initial exploration in an elegant way. An action is potentially selected until there is strong negative evidence of its uselessness. In this case some other promising action is selected until one is found that does not disappoint the initial expectations.

Choosing the right exploration strategy is up to experience and the intuition of the control system designer. No method has yet been proven to be the best in all cases, and the right choice may strongly vary with the task.

4.1.3 Markov Decision Processes

In general, the agent has to be able to learn even though the reward is delayed. An example of *delayed rewards* is an environment where the agent receives zero reward except when it reaches one of the goal states. In this case, the agent has to learn the sequence of actions leading to future rewards in a goal state.

The problem of calculating an optimal policy in an accessible², stochastic environment with delayed rewards can be modelled as *Markov decision processes* (MDPs). A Markov decision process consists of:

²An environment is called *accessible* if the agent has complete access to its state.

- a finite set of states S
- a finite set of actions A
- a scalar reinforcement function R
- a state transition function T

A necessary condition for a reinforcement learning problem to be modelled as a MDP is that the state transitions are independent of any previous environment states and actions. In other words, the state transitions have to be independent of the path of the agent until this point and only depend on the current state. In this case the problem is said to satisfy the *Markov property* and the model is called *Markov*³.

4.2 Finding the Policy

Assume that a given problem can be modelled as a MDP and that a deterministic or stochastic model is known in advance. Examples of this situation are board games like checkers, chess or backgammon. In these games, all states, actions, and transition probabilities are known. Back in 1957, Bellman showed that in this case there exists an *optimal deterministic policy* [KLM96]. Bellman called methods of finding this optimal policy for *dynamic programming*.

In order to formalise things a bit the *value* of a state is now defined by the *optimal value function*:

$$V^*(s) = \max_{\pi} E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \quad (4.3)$$

In words, the optimal value of a state $V^*(s)$ is the expected discounted sum of rewards the agent will receive in the future when following the optimal policy π^* . Whereas the reward function indicates what is good in an immediate sense a value function specifies what is good for the agent in the long run [SB98]. Note that this definition of the value function agrees with the *infinite horizon discounted model* of optimal behaviour defined in equation (4.1).

Value Iteration

There exist several *dynamic programming* techniques for finding the optimal policy given the model. One of them is *value iteration* [KLM96] where the optimal value function is approximated by an iterative process updating an estimate V . Eventually V will converge to V^* . The *optimal policy* can be found from V^* .

³After the Russian statistician Andrei A. Markov [RN95].

Policy Iteration

Another method is *policy iteration* [KLM96] where the policy rather than the value function is manipulated. In a repeated loop the policy is, slightly improved in each iteration and the value of the new policy is calculated. This method requires fewer iterations than *value iteration*, but each step is more computationally expensive.

Full Backups versus Sample Backups

The two methods described above use updates called *full backups*⁴ which means that they consider all possible successor states when updating the value function. These methods are computationally very expensive. On the contrary in *sample backup* methods, a single sample of the environment is used for each backup. This makes them much more attractive in real-time applications where the model is not known in advance, but can be derived from the interactions. In the next section the *sample-backup* method Q-learning will be described in some detail.

4.3 Q-learning

One branch of methods to learn the optimal policy is called *model-free* methods because they learn a policy without building a model of the environment. Q-learning is easy to implement and a widely used model-free reinforcement learning algorithm originally proposed by Watkins [Wat89].

In Q-learning, the *value function* is extended to be an *action-value function*, storing the value of every action in each state. The action-value function $Q^*(s, a)$ expresses the value of taking action a in state s and then behaving optimally. The relationship between the value function and the action-value function is:

$$V^*(s) = \max_a Q^*(s, a) \quad (4.4)$$

Q-learning is very simple because the action-value function is the only thing that has to be stored and updated. The policy follows directly from Q^* by selecting the optimal (or greedy) action in each state:

$$\pi^*(s) = \arg \max_a Q^*(s, a) , \quad (4.5)$$

where $\arg \max_a$ denotes the value of a which maximises the expression.

But how is this this action-value function found? This is done by iterative approximations to the “true” action values. A first observation is that $Q^*(s, a)$ can be written recursively as:

⁴The term *backup* is commonly used to denote an update of a value function.

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a') \quad (4.6)$$

The first term $R(s, a)$ of this equation is the immediate reward of taking action a in state s and the second term is the future discounted rewards. The transition function $T(s, a, s')$ denotes the probability of making the transition from s to s' by action a . For a summary of the notation used, see Appendix A.

At first each entry of the table storing $Q(s, a)$ is initialised arbitrarily. Q^* is now approximated based on successive sample experiences of the agent in the environment. Each experience of an agent can be described by an experience tuple $\langle s, a, r, s' \rangle$; the experience of taking action a in state s receiving reward r and transitioning to state s' . For each experience a backup of the Q-table is made using the update rule:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a')) \quad (4.7)$$

The new factor $\alpha \in [0; 1[$ is the *learning-rate*. High values of α will result in drastic updates of Q-values on each step whereas low values mean more conservative changes. It has been shown [WD92] that if each action is executed in each state an infinite number of times and α is decreased appropriately, the Q-values will converge to Q^* with probability 1, and the optimal now follows from equation (4.5). Furthermore, Q-learning is *exploration insensitive* which means that convergence to optimal guarantee holds, independent of exploration method used during the learning-phase. A summary of the Q-learning algorithm is given in figure 4.2

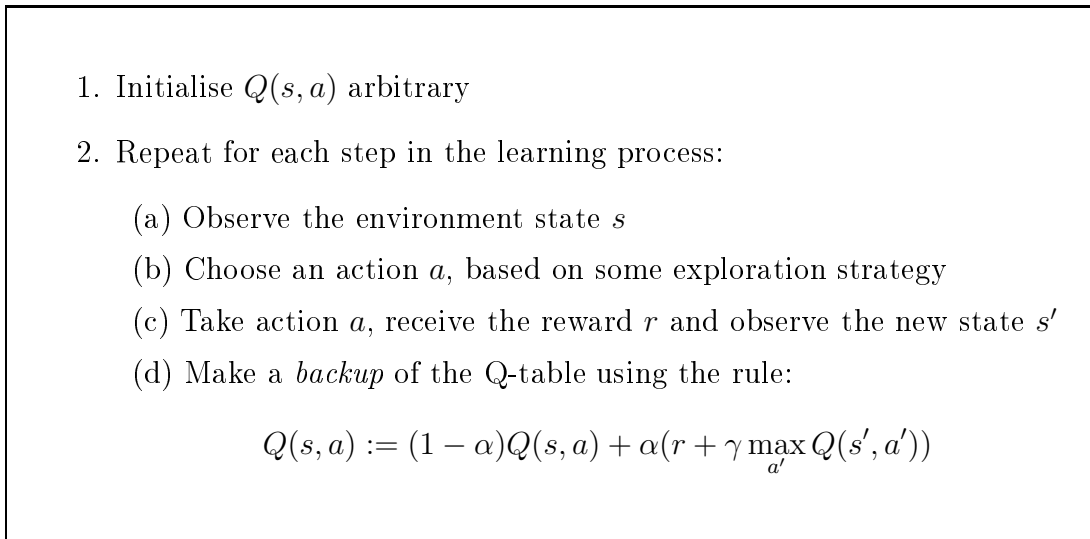


Figure 4.2: *The Q-learning algorithm.*

4.4 The Dyna Algorithm

Q-learning was a big improvement over previous methods because of its simplicity and its convergence guarantees. A drawback of the method, however, is that even though it is guaranteed to converge it may take a large number of learning steps before so. The reason for this is that very little use is made of each environment interaction of the agent. If the agent comes from a part of the state space where no reward is to be found only the last step into a reward state is made use of. The previous steps, or the path of the agent are just discarded. In situations where real world interactions are considered expensive and computation between the interactions is considered cheap, other methods might prove advantageous. Some of these methods are the *model-based* methods where direct reinforcement learning (as in Q-learning) is combined with *planning*. Planning means that a world model helps the decision making of the agent. An overview of the model-based learning/planning architecture can be seen in figure 4.3.

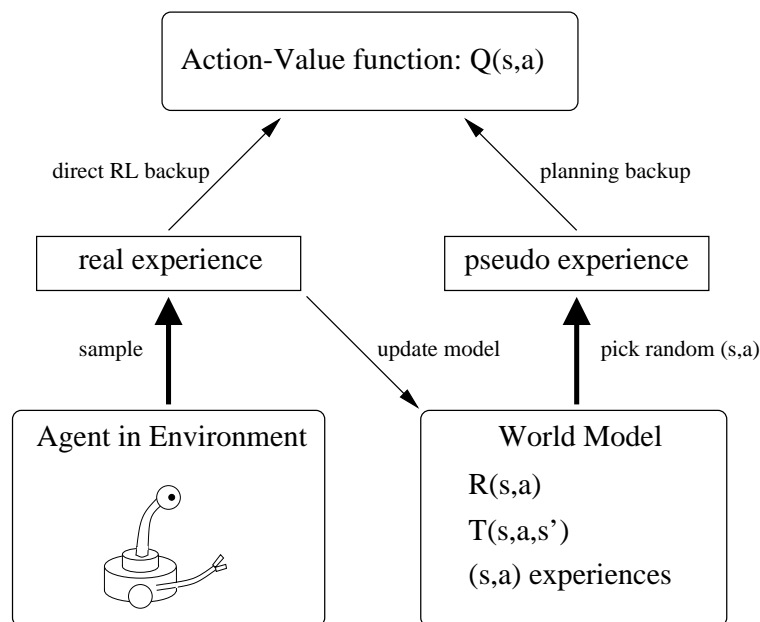


Figure 4.3: *The model-based learning / planning architecture used in Dyna.*

One such method is Sutton's Dyna architecture [Sut90]. In his paper, Sutton calls the algorithm described next, for *Dyna-Q*, because it is based on Q-learning, but here it will just be referred to as *Dyna*. The Dyna algorithm simultaneously uses the agent experiences to build a model and to adjust the policy. The model being build consists of:

$$Model(s, a) = \begin{cases} \hat{T}(s, a, s') & \text{an estimate of the state transition function} \\ \hat{R}(s, a) & \text{an estimate of the reward function} \\ SA & \text{a set of experienced state-action pairs} \end{cases}$$

Based on each experience tuple, the algorithm updates the model and makes a direct backup of the action-value function. In addition k state-action pairs are selected from SA and these “pseudo-experiences” are used for k additional updates.

The models of R and T are estimated by the simple rules:

$$\hat{R}(s, a) = \frac{\text{Sum of rewards received by taking action } a \text{ in state } s}{\text{Number of } (s, a) \text{ experiences}}$$

and

$$\hat{T}(s, a, s') = \frac{\text{Number of } (s, a, s') \text{ experiences}}{\text{Number of } (s, a) \text{ experiences}}$$

These estimates can be maintained by incrementing the statistics after each real world experience. A summary of the Dyna algorithm is given in figure 4.4. As it can be seen the Dyna backup rule is similar to the recursive definition of the action-value function (see equation(4.6)), except that the model being used is the learned model and not the unknown *ideal* model. The intuitive idea behind the k additional updates is to share the new information throughout the state-action space, and thereby speed up the convergence to Q^* . The additional updates are referred to as simulated experiences or “pseudo-experiences” because they access the model build via real experiences, but do not access the environment directly. Actually in the algorithm in figure 4.4, the “direct” update of the Q-function also refers to the model, but based on the actual experienced (s,a)-pair. This is done to get off the α -parameter of Q-learning.

4.5 Prioritized Sweeping

The Dyna algorithm has been criticised that the k additional updates of the Q-table are not chosen in any specific way, but at random. If the updates are done in an uninteresting part of the state-action space, the effective result is just a waste of computation time. Prioritized Sweeping is an extension of Dyna that tries to fix this problem by focusing the computations on the more interesting parts of the state-space. The algorithm referred to here as Prioritized Sweeping (PS) is

1. Initialise $Q(s, a)$ arbitrary
2. Initialise $Model(s, a)$
3. Repeat for each step in the learning process:
 - (a) Observe the environment state s
 - (b) Choose an action a , based on some exploration strategy
 - (c) Take action a , receive the reward r and observe the new state s'
 - (d) Use the experience tuple $\langle s, a, r, s' \rangle$ to update $Model(s, a)$
 - (e) Make a *backup* of the Q-table using the newly updated model and the rule:

$$Q(s, a) := \hat{R}(s, a) + \gamma \sum_{s' \in S} \hat{T}(s, a, s') \max_{a'} Q(s', a')$$

- (f) Choose k additional state-action pairs from SA , and make k backups using the same rule as before:

$$Q(s, a) := \hat{R}(s, a) + \gamma \sum_{s' \in S} \hat{T}(s, a, s') \max_{a'} Q(s', a')$$

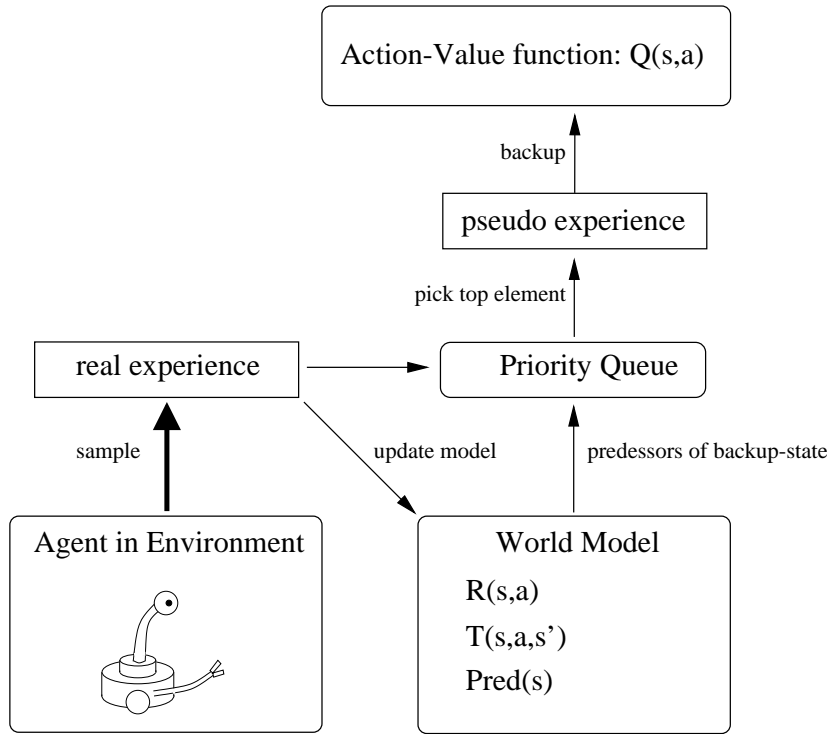
Figure 4.4: *The Dyna algorithm.*

a mixture of two independently developed, but very similar methods: Prioritized Sweeping proposed by Moore & Atkinson [MA93] and Queue-Dyna by Peng & Williams [PW93].

The idea is to assign a priority, p , to each state-action pair. In each learning step the k top-priority pairs are used for the Q-table updates. The priorities are assigned based on prediction difference magnitudes of the Q-values. On each step priorities are assigned to the update candidates, which are:

- The just experienced state-action pair
- The *predecessors* of any state where a backup is carried out

The predecessors of a state are the state-action pairs (\bar{s}, \bar{a}) with a non-zero transition probability to s , i.e. $T(\bar{s}, \bar{a}, s) \neq 0$. These pairs are kept in a predecessor function $Pred(s)$ which is added to the model. This means that the model now includes:

Figure 4.5: *The Prioritized Sweeping architecture.*

$$Model(s, a) = \begin{cases} \hat{T}(s, a, s') & \text{the transition function estimate} \\ \hat{R}(s, a) & \text{the reward function estimate} \\ Pred(s) & \text{the predecessor function} \end{cases}$$

In addition a priority queue data-structure *PQueue*, initially empty, is kept. The elements to be inserted in *PQueue* are state-action pairs (s, a) with priorities p . Before an element can be inserted the priority has to exceed a small threshold δ . The modified architecture can be seen in figure 4.5. The priority queue has been inserted and the backup pairs are now selected from this. As can be seen the real-world experience is no longer automatically backed up, but is inserted in the queue and may be selected from this. See figure 4.6 for a summary of the algorithm.

The global result of this algorithm is that when a surprising transition is experienced, for example when a goal state is reached for the first time, this information is propagated back to the predecessor states making an efficient use of the “interesting” real-world experiences. On the other hand when the result of a transition is the same as expected, no more updates are done in this area, and if the queue is

1. Initialise $Q(s, a)$ arbitrary
2. Initialise $Model(s, a)$ and $PQueue$
3. Repeat for each step in the learning process:
 - (a) Observe the environment state s
 - (b) Choose an action a , based on some exploration strategy
 - (c) Take action a , receive the reward r and observe the new state s'
 - (d) Use the experience tuple $\langle s, a, r, s' \rangle$ to update $Model(s, a)$
 - (e) $p = \hat{R}(s, a) + \gamma \sum_{s' \in S} \hat{T}(s, a, s') \max_{a'} Q(s', a') - Q(s, a)$
 - (f) If $p > \delta$ then insert (s, a) in $PQueue$ with priority p
 - (g) Repeat k times (or until $PQueue$ is empty)
 - Remove top element from $PQueue$. Call it (s, a)
 - $Q_{old} = Q(s, a)$
 - $Q(s, a) := \hat{R}(s, a) + \gamma \sum_{s' \in S} \hat{T}(s, a, s') \max_{a'} Q(s', a')$
 - Compute value change: $\Delta = |Q_{old} - Q(s, a)|$
 - for each $(\bar{s}, \bar{a}) \in Pred(s)$:
 - $p = \Delta \cdot T(\bar{s}, \bar{a}, s)$
 - if $p > \delta$ then insert (\bar{s}, \bar{a}) in $PQueue$ with priority p

Figure 4.6: *The Prioritized Sweeping algorithm.*

not empty interesting pairs from the last step will be used for the remaining updates. On deterministic simulated maze-problems, this method has been shown to outperform the random Dyna-updates [MA93][PW93].

4.6 Summary of the Algorithms

Watkins Q-learning was indeed a big improvement over previous methods because its incremental nature which makes it well suited for real-time applications in an unknown environment. When moving on to the model-based methods, it has been argued that this is a step backwards to the classical AI way of thinking [RN95]. Recall that in the classical AI framework a robot would also build an internal model of the world in order to plan its safest path in the future. In

	Q-learning	Dyna	Pri. Sweep.
Parameters	α and γ	γ and k	γ , k and δ
Updates per step	1	$k + 1$	$\leq k$
Model-based	no	yes	yes
Model consists of		\hat{T} , \hat{R} and SA	\hat{T} , \hat{R} , $Pred$
Value function	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
Other data-structures			$PQueue$

Figure 4.7: *Summary of the three algorithms.*

model-based RL however, even though a model is being build, the model is only used in order to compile some past knowledge into a reactive policy. No planning intervenes between perceiving a situation and responding to it [Sut91], so in this sense it is not fair to compare model-based RL to the classical AI robots where the *sense-think-act* cycle could result in with very long response times.

In figure 4.7 a summary is shown of the parameters to be set and the data-structures to be kept in the three algorithms.

4.7 Putting RL onto Robots

Until this point reinforcement learning has only been discussed on the abstract level that it is often presented in the RL literature. When using reinforcement learning on a concrete robot learning task, several additional issues have to be addressed.

4.7.1 The Markov Property

A serious restriction to the usability of reinforcement learning in situated robot domains is the fact that the Markov property (see section 4.1.3) is seldom or never fulfilled in real-world applications. When the Markov property does not hold, the theoretical guarantees of convergence to optimal behaviour, described earlier in this chapter, do not apply anymore. There are two main reason for the collapse of the Markov property. First of all a mobile robot only has access to its own local sensors when determining its state in the world. From the outputs of local sensors like proximity-sensors, sonars, or even a camera, it is impossible to determine the robot's exact position in the world in absolute (x,y)-coordinates. Somehow these local sensor readings have to be mapped to a description of the state anyhow. When this is done, the previous path of the robot will have an effect on the next state the robot finds itself in not just the current state and action,

and the Markov property is not fulfilled anymore. This problem is known as the problem of *incomplete perception*, *hidden state* or *perceptual aliasing* [KLM96]. Secondly, there is the problem that sensors are noisy. Even if the physical robot does not move, two consecutive readings of analog sensors will never give the exact same result, making the environment non-deterministic.

4.7.2 State-space and Action-space Construction

If the raw sensor-values are mapped directly into a table, the size of it would explode. Some state-space pruning has to be done to make the methods applicable. The way that the state-space is pruned has an effect on the success of the learning algorithm. A clever state-space construction may speed up learning while a inefficient one may prohibit the robot from learning anything within a reasonable time limit. A similar problem occurs when a mapping of a finite action-set into a continuous motor-speed interval has to be found. These problem will be investigated further when setting up the experiments in chapter 5.

4.7.3 The Reward Function

In the classical model of reinforcement learning described in this chapter, the reward function was thought of as a part of the environment. The RL problem was described as an optimisation problem given an agent in an environment. When setting up a RL system, the reward function has to be constructed as well. This reward function should preferably have the property of assigning high rewards to an agent performing well, thus making it possible for the agent to learn a policy that solves the task in question. The problem of reward function construction will be taken up in chapter 5 as well.

4.7.4 Evaluation

The last issue to be described here is the problem of evaluating the performance of the learning robot. In deterministic or stochastic MDPs the optimal policy can be calculated. When evaluating the performance of a learning agent in this case, the current policy can be weighted against the optimal policy and the number of updates before convergence to optimal can be used as a suitable performance measure. In non-Markov problems there is no longer any proof of existence of an optimal policy so some other kind of measure has to be found when evaluating the performance of the learning robot.

4.8 Additional Issues in Reinforcement Learning

In this section some additional important issues in reinforcement learning will be mentioned. It is beyond the scope of this text to go deeper into these subjects.

4.8.1 Partially Observable Markov Decision Processes

The simplest way of dealing with a non-Markovian reinforcement learning problem is just to ignore the fact, and hope that the methods still works even though there are no theoretical guarantees anymore. This is the most common way of dealing with non-Markovian environments. Another way of dealing with noisy incomplete state information is to extend the basic MDP framework into *partially observable Markov decision processes* (POMDPs) [KLM96]. A memory mechanism called a *state estimator* is added to the model. The state estimator computes a *belief state*, the state the agent is believed to be in, as a probability distribution over states of the environment. With this extension, the problem can again be formulated as a MDP and the usual techniques can be used. However, the size of this new MDP can be very larger.

4.8.2 Eligibility Traces

The Q-learning method, described in section 4.3 is sometimes referred as one-step tabular Q-learning. The mentioned way of improving the performance of Q-learning was to introduce the *model-based* methods. An existing model-free extension of Q-learning to make a more efficient use of the learning experiences, is that of eligibility traces. In these methods, long traces of agent experiences are investigated rather than only the last step. When eligibility traces are combined with Q-learning, the resulting algorithm is called $Q(\lambda)$ [SB98]. Watkins has proposed a version of $Q(\lambda)$, looking at sequences of greedy actions. Unfortunately, the sequences of greedy actions are usually not that long when an exploration strategy is being used in the learning process. Peng's $Q(\lambda)$ is a corrected version of $Q(\lambda)$, looking at longer sequences of agent actions not just the greedy ones. This method has been seen to improve the performance of simple one-step Q-learning [PW96], however one problem with the approach is that the method is no longer *exploration insensitive* as was the case with pure Q-learning.

4.8.3 Generalisation

If the state-space of a RL problem is mapped into a huge table with many entries the learning may have some convergence problems. In this case some kind of generalisation over similar states is needed. Instead of storing the Q-function in a table, an artificial neural network can be used. In one of the most impressive

applications of reinforcement learning to date, Tesauro used a three-layered neural network to store the value function in his backgammon player *TD-Gammon* [SB98]. TD-gammon did by self-play reach the performance level of any human backgammon player. In fact human players have been inspired by some of the moves of the TD-gammon program.

Chapter 5

Obstacle Avoidance Learning

In this chapter learning of the well-known task obstacle avoidance on a Khepera robot will be investigated. The task in itself is not particular interesting, but it is well-suited for comparing different learning techniques because it is simple, but not trivial. The hope is not to find new interesting solutions to the problem, but to compare different parameter settings and techniques on a task that is well understood. The interesting part is not the learned policy, but the way that the policy is learned.

5.1 Experimental Setup

5.1.1 The Agent

In this experiment the miniature Khepera Robot is used. The robot has a diameter of 55mm, a height of 30mm, and a weight of about 70 g. The robot is equipped with 8 infrared (IR) sensors, 6 in the front and 2 in the back. In figure 5.1 a photo can be seen of the robot and the placement of the sensors and wheels is illustrated. Each sensor returns two 10bit values; one reflected light value and one ambient light value. In this experiment only the reflected light values will be used. This value gives a measure of distance to obstacles in the proximity of the sensor. An obstacle can be detected in the range of about 2-5 cm depending on the colour and the surface of the obstacle and additional external conditions¹. To be able to move, the robot has two wheels independently controlled by DC motors with incremental encoders. The accuracy of the odometry depends on the motor speeds, the surface of the experiment, and any interactions with walls or other objects. The robot has a Motorola 68331 CPU and rechargeable batteries that allows the robot to run fully autonomous for about 30-40 minutes. In the following experiments, the robot will run in serial communication mode.

¹The reflection angle, external light sources, the room temperature, and infrared noise in the room all affects the measurement.

A lightweight aerial cable with rotating contacts serves both as a serial RS232-connection between the robot and the host computer and as a power supply for the robot. The rotating contacts prevent the cable from being twisted, thus giving the robot freedom of movement. A program running on the host computer sends motor commands to the robot and receives sensor values through the serial connection. The reason that the computation is not done on the robot itself is that this setup will allow a camera turret to be mounted on top of the robot. The current hardware does not allow the image-processing on the robot itself. This restriction that some of the computation is done on a host computer and not on the robot itself is not serious however. Only the robot's own sensors are used so on a robot platform with more computational power it would be able to run fully autonomous. Another advantage of running the control program on a host computer is that it makes it easier to log experimental data. The major disadvantage is of course that battery consumption problems are overlooked.

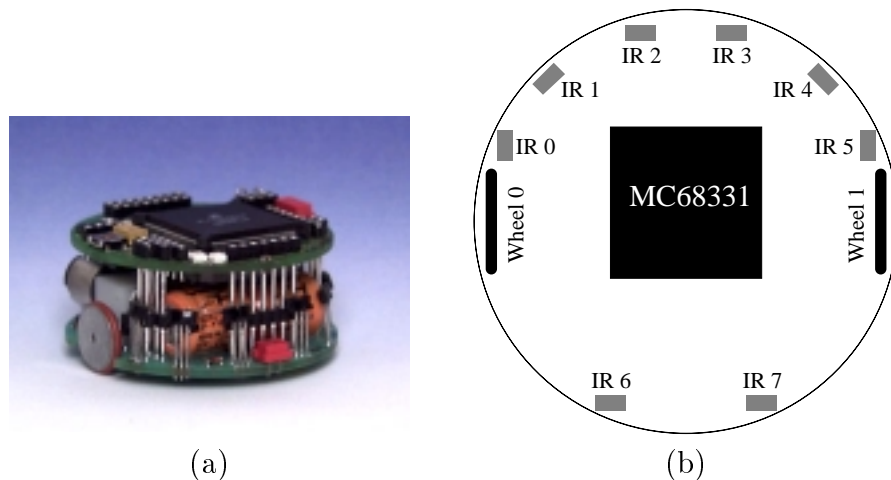


Figure 5.1: *The Khepera Robot.*

5.1.2 The Environment

The environment that the experiments in this chapter are run in is a field made out of LEGO Duplo bricks (see figure 5.2). The field is designed to have a lot of different corners instead of just straight walls, making the task the robot has to solve more difficult.

5.1.3 The Task

The task that the robot has to learn is that of obstacle avoidance, that is maintaining the goal of staying clear of walls and other obstacles in the environment.

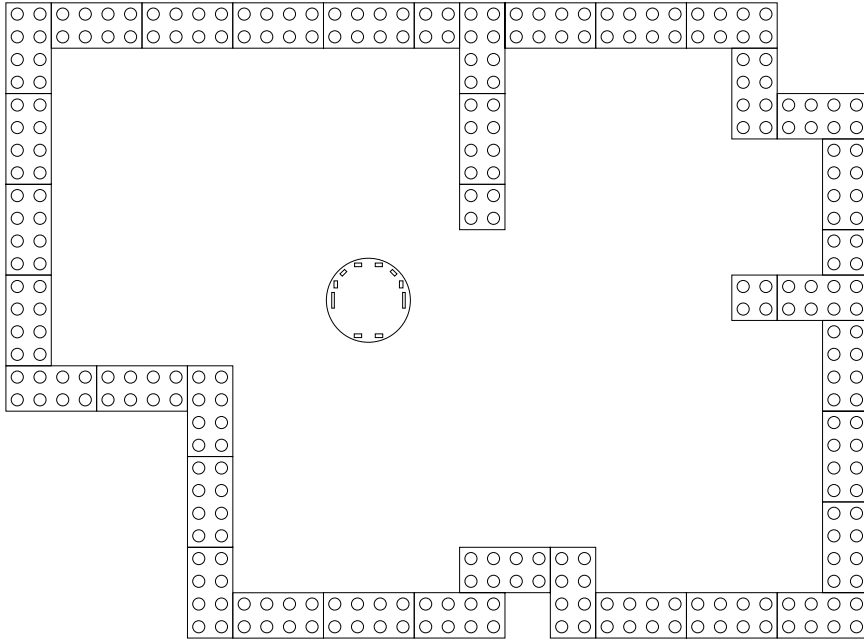


Figure 5.2: *The robot in the field made of LEGO Duplo bricks.*

5.1.4 Sensor-space Division

How to map the direct sensor-input into a set of states is a very important design issue. The ideal situation would be that the human designer did not have to make this choice, but in reality, when using a table for storing the Q -function and no generalisation is used, this problem has to be addressed. The trade-off is that if the sensor-space division is too fine, the size of the Q -table will explode and the learning will never converge, or converge unnecessarily slow because of the many entries in the table that has to be stabilised. On the other hand, if the division is too coarse, an agent action will very seldom result in a state transition and this lack of diversity may make the task impossible for the agent to learn. If the 10bit values of each of eight IR-sensors were mapped directly into a table this would result in a total of 1024^8 states, and this number would have to be multiplied by the number of motor actions in each state to get the table size. The result would be an astronomical number of entries. Instead the choice was made to code each sensor-value with one bit and only use the six front sensors. A sensor is *on* if the measurement is above some threshold and *off* otherwise. A similar state-space division is used in [Tou97]. This division results in $2^6 = 64$ possible agent states. The effective number of states is even less because some of the sensory states will never be experienced. For example a state where every other sensor is *on* will never occur in the field used in these experiments, although is it possible to imagine a physical setup where it would occur. If a state-space

construction is chosen where the theoretical possible number of states is much greater than the ones visited in reality, an implementation, where space for a table entry not allocated beforehand but only when met, could drastically reduce the space consumption.

This choice of the sensor-space construction was mostly based on human intuition and experience, but a strong focus was put on trying to minimise the number of states. First of all fewer states will, all other things being equal, result in faster convergence to the desired behaviour². Secondly, it does not make much sense to discriminate the action based on the least significant bit of the sensor value, when it is a well known fact that the sensors are noisy [MC96]. In this sense the coarse representation is a way of dealing with noise. A third argument is that with this search-space, a hand-coded solution that could successfully solve the task was implemented. This means that there is at least one solution to the problem in the search space of the learning algorithm. This argument is of course not valid when the aim is to learn behaviours that could not be thought of beforehand, but as pointed out earlier that was not the case in this experiment.

In the current implementation the threshold is set to 300 (the range of the sensor is between 0 and 1023). If sensor reading is above 300, the sensor is *on* and otherwise it is *off*. A robot turning on the spot right next to a straight wall will visit 13 different states with this state-space division. If the robot is placed in a corner, or further away from the wall, different states will occur. In section 5.3 some statistics on visited states will be presented.

The Markov Property Revisited

In chapter 4, the claim was that in real robot applications the learning problem is seldom or never Markovian. This is certainly the case here where a robot equipped only with short-range sensors is placed in an open field. It suffers from incomplete access to the true state or *perceptual aliasing* as described earlier. If the robot has no input in the sensors the robot can be anywhere in an open part of the field. The actual position in (x,y)-coordinates cannot be known from just reading the sensors. Another thing that makes the task even less Markovian is that the state transitions are not deterministic and not even probabilistic. The transition probabilities are slightly different in different parts of the field and at different distances to the walls. However, when the wheels do not slip and the robot is not in contact with the walls, the approximation to a stochastic environment is not that unrealistic, and as the experiments will show the task can be solved under these circumstances. This might not always be the case with a more complex task and/or environment.

²In real robot experiments, time *is* an important factor.

5.1.5 Action-set Construction

The next design-question that has to be addressed is the way of choosing the action-set available to the robot. The problem is that if all possible motor speed combinations of the two wheels are chosen, the Q-table size will explode, resulting in a huge space consumption and poor convergence properties. In the case of the Khepera Robot there are 256 possible motor actions for each wheel. Importance was attached to having as few actions as possible, but still making the desired behaviour possible. The choice was four different actions in each state, summarised in the table of figure 5.3.

Action	left motor	right motor
Forward	6	6
Turn right	6	-6
Turn left	-6	6
Backward	-6	-6

Figure 5.3: *The action set. A motor value of 6 on a Khepera robot corresponds to a wheel speed of 48 millimetres per second.*

5.1.6 The Reward Function

In reinforcement learning the reward function plays the same role as the fitness function in evolutionary algorithms (EA), and the problems encountered when designing reward function are similar to those connected with fitness functions in EA. A lot of care has to be taken when designing the function and there is not much help in the literature. Most literature on reinforcement learning do not concern this matter at all, although it is perhaps the most difficult aspect of setting up a RL system [Mat94]. The classical reinforcement learning methodology only concerns the problem of maximising the reward given an agent in an environment and given a reward function.

In robotics the problem often is that even though the robot controller succeeds in maximising the reward, the observed behaviour does not correspond to the wanted one. The typical example illustrating this problem is an obstacle avoiding robot that backs up until it is clear of an obstacle and then moves forward bumping into the same obstacle once more. Despite of these difficulties, a reward function is a more natural way of specifying what we want the agent to do than the teaching patterns used in supervised learning. It is often easier to say whenever the agent is behaving *good* or *bad* than to specify what action it should have taken in any given situation.

The reward function should have the property that the closer the behaviour is to the wanted one, the more reward is given to the agent. If the function has this property finding a good solution is not just a matter of random search, but rather a guided search. In this experiment, the initial idea was a delayed reward function where the agent was given a reward for being in `state 0` (the state where all sensor-inputs are below a given threshold). In addition solutions are not wanted where the agent just turns on the spot wherever it is free of all obstacles, so the reward is only given when the action is `forward` in `state 0`. These considerations suggest the following reward function:

```
if (maxIR < IR-threshold && action == FORWARD)
    reward = 100;
else
    reward = 0;
```

Here `maxIR` denotes the sensor-value of the IR-sensor with the highest activation. The problem with this function is that the robot finds a “solution” where it keeps moving forward, until `maxIR` is above the threshold and the backward until it falls below again. To avoid this suboptimal behaviour, a penalty can be given to the robot whenever it chooses the `backward` action. In fact, the robot is not wanted ever to go backwards, but the `backward` action has to be in the action-set in order to ensure a good search of the state-space during learning. The presence of the *backward* action makes the robot back up and try again once in a while. With this in mind the new reward function has the following form:

```
if (action == BACKWARD)
    reward = -200;
else if (maxIR < IR-threshold && action == FORWARD)
    reward = 100;
else
    reward = 0;
```

The punishment of taking the `backward` action has to be greater than the reward for taking the `forward` action in `state 0` because otherwise the robot will learn to accept the immediate punishment of backing up until it finds itself in the reward-giving `state 0`.

Before implementing the reinforcement learning system on the real robot, some initial experiments were done in the Khepera Simulator made by Olivier Michel [Mic96]. The reason for doing these first experiments was to get a feel of the RL problem in a user-friendly environment and to use the insight gained here when moving on to the real robot. In this simulator the reward function above

was adequate and the simulated robot learned an obstacle avoidance behaviour. However, when moving the system out on the real robot, a new problem arose. The problem was that early in the learning process the robot would often get stuck on a wall after a sequence of `forward` actions. The surface of the field was not smooth enough to allow the wheels to slip all the time, and the motors were not powerful enough to make the robot turn due to friction between the robot and the walls. The result was that no matter which action was chosen, the robot did not move. The only way to get out of this deadlock was to physically lift up the robot, resulting in a release of the tension build up inside the motors.

In order to avoid this problem of the robot getting stuck, the motor speeds in the four actions were doubled. Now the robot was able to move most of the time, but this solution induced some new problems. First of all the new motor speeds made the robot move around too fast in the state-space, jumping over neighbouring states, which made convergence difficult. On top of this, the robot would sometimes slide sideways along a wall and make a random turn when it got free of the wall again due to power tension accumulated in the motors. The result was that the state transition probabilities were disturbed at random, making the resulting learning problem very non-Markovian. The result was that the learning did not converge to the desired behaviour.

The lesson learned was that increasing motor speeds was not a good idea under these circumstances, so they were reset to 6. Another solution had to be found. The reason for the robot getting stuck was that the `forward` action was tried too often, despite of the robot being close to an obstacle. It took several experiences for the robot to learn to turn to the right direction instead. To speed up this learning, another factor was added to the reward function: punishing the robot for moving straight forward when being in front of an obstacle. This was the final modification to the reward function which now had the following form:

```
if (action == BACKWARD)
    reward = -200;
else if (maxIR < IR-threshold && action == FORWARD)
    reward = 100;
else if (maxIR > IR-threshold && action == FORWARD)
    reward = -50;
else
    reward = 0;
```

With this reward function, the robot was now able to learn how to solve the task.

5.1.7 Measures of Performance

When initial setup is done and the learning process seems to be up and running, the next thing that has to be decided upon is how to measure the performance of the robot. Obstacle avoidance is an example of a task where the agent has to *maintain* the goal of staying clear of obstacles in the environment. Other typical tasks where a robotic agent has to maintain a goal are wall-following, line-following and box-pushing. These types of tasks can be seen in contrast to tasks where a goal has to be achieved or *reached*. Examples of these are a robotic footballer that has to score a goal, or a garbage collecting agent that has to clean an arena. In the latter examples there are obvious ways of measuring the performance of the agent. In the robot footballer example the scoring percentage or the time to score a goal can be used. In the garbage collecting example the time to clean the arena or cleaning percentage in a limited time period are obvious choices. In the *goal-maintaining* tasks, it can be more problematic to find a measure of performance, especially if the task is so vaguely defined as in the case of obstacle avoidance. Is it an acceptable behaviour for the agent to find the centre of the arena and start turning on the spot? An intuitive answer to this is no, but there is no way to deduce this from the description of the task given in section 5.1.3. What is wanted for the agent to do is to wander around in the field and whenever it gets close to an obstacle stay clear of it as fast and smoothly as possible. A possible solution is to use a *qualitative* analysis of the behaviour [WHH98] as for example how good the agent is to stay clear of obstacles or how often it bumps into them. This kind of analysis would require an external human observer to sit and judge the agent all the time and it can be very imprecise or subjective. If an automatic judging of the agent is wanted, generally a *quantitative* measure of the performance has to be used. This measure can be based on data logged automatically when the experiments are run. The first measure of this kind that comes to mind is to use the reward gained by the agent during a run. This is a natural choice because the long-term reward was the quantity that the agent was supposed to maximise in the first place. Care has to be taken however, because it is not self evident that an agent that succeeds in gaining a lot of reward is the agent that solves the task at the best. After designing the reward function we have to convince ourselves that there is in fact a correspondence between a high reward and the wanted behaviour. This complex of problems is, as stated earlier, the same as in evolutionary robotics where the fitness function plays the role of the reward function in reinforcement learning.

The reward is a usable measure when we want to compare the effect of different parameters in the technique (α , γ or timestep length), or compare some different learning techniques. It can be used to tell if the learning converged to a high reward-giving behaviour and show the speed of convergence. If we want to compare the behaviours under different reward functions, different state- and

action-spaces or different sensor configurations, however, we will need some other measures. In these cases accumulated motor speeds of the robot can be used. If the proximity sensor-configuration is fixed, a third possibility is to use the maximal or average proximity sensor-value during a run as the measure.

When evaluating the performance of a learning robot, we would like to evaluate the learned controller at each step, but that is impossible in practice. Instead we have to evaluate traces of runs. The different runs can show very different results even with the same technique and the same parameter settings. The robot can be “unlucky” and get stuck in a difficult part of the field. So to say something general about the results we have to repeat each experiment as many times as possible. Time is an important factor because real robot experiments are indeed very time consuming. In this text, the decision was to repeat each experiment 10 times.

5.1.8 Interfacing the Robot

In order to allow easy supervision and adjustment of the learning process, the friendly graphical user interface *KhepReal* was constructed. The interface can be seen in figure 5.4. At the left side of the main window, the sensor-values of the eight IR proximity-sensors are shown. In addition the actual motor values of the robot are shown. These are not the values set by the controller but readings of the actual speeds from the PID controllers³. In the middle of the window the robot can be remote-controlled and various parameters can be set. The actual values of each entry in the *Q*-table, *R*-table and *T*-table can be investigated at the right side of the window. The learning can be single-stepped, allowing the size of the updates under different parameter settings to be investigated.

5.2 Experiment 1: Setting the Learning Rate

In every experiment the robot has to learn a policy based on some exploration strategy. This exploration and learning period will be called the *learning phase*. As noted in [WHH98] in order to measure the worth of the policy learned, it is necessary to have a distinct period during in which exploration is switched off. This period will be called the *acting phase*.

³Both motors are controlled by a PID controller. Every term of this controller (proportional, integral, derivate) is associated to a constant in the acceleration-curve of the robot [K-T95]. The job of the controller is to keep the wheel speed at input level. In the physical world reaching a wheel speed is not an instantaneous action.

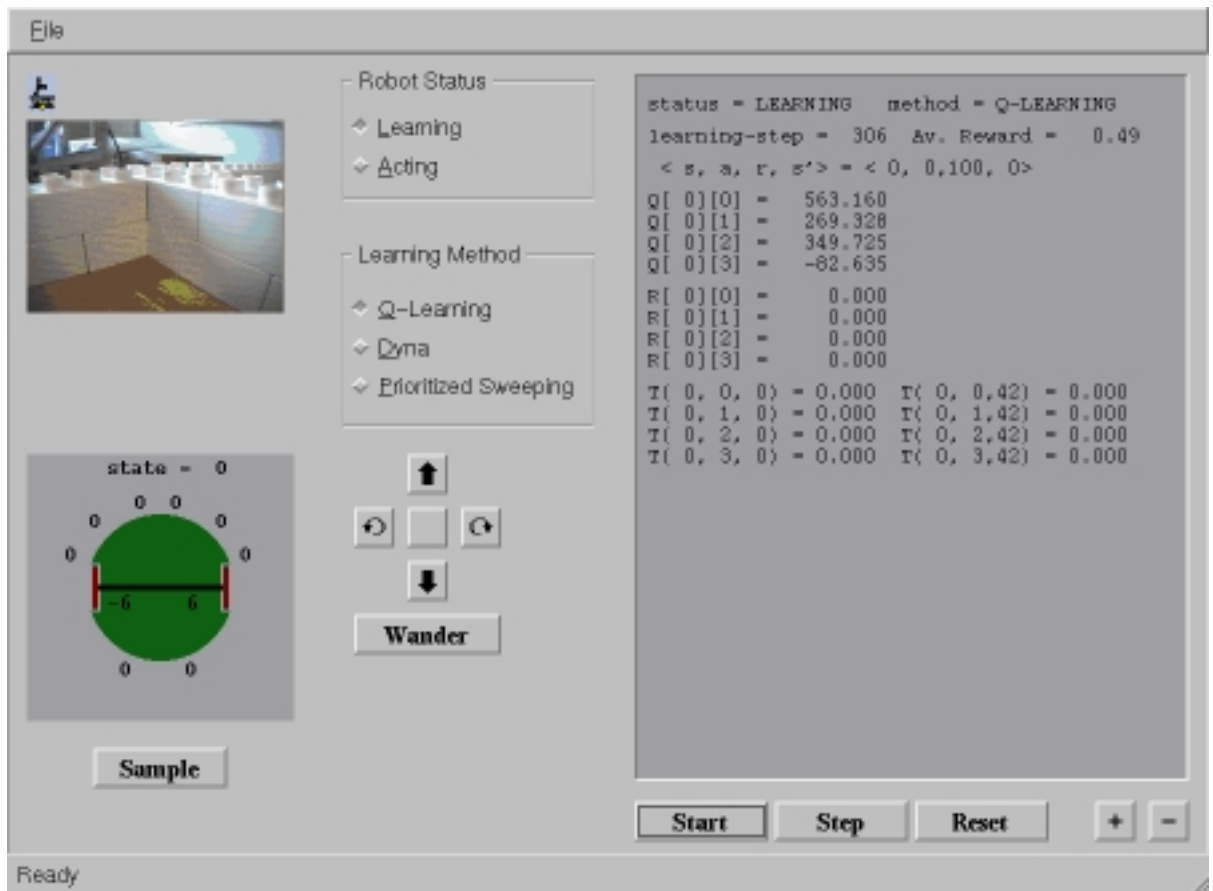


Figure 5.4: *KhepReal*. The graphical user interface.

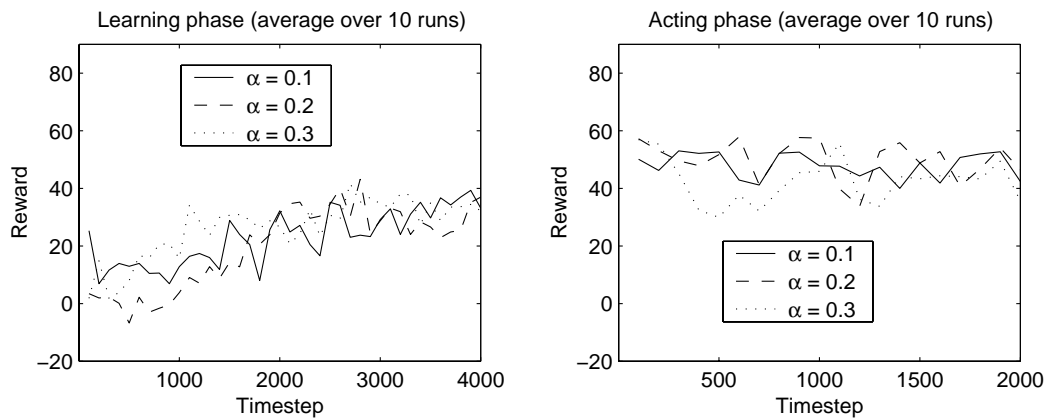


Figure 5.5: *Q-learning*: Different learning rates.

5.2.1 Q-learning

In this first experiment, the setting of the learning rate in pure Q-learning will be investigated. The algorithm from figure 4.2 was straight-forwardly implemented. It was decided to initialise the entries in the table to 0. After some initial tests, it was decided to use a learning-phase of 4000 timesteps and a acting-phase of 2000 timestep. Within the 4000 steps of learning, most controllers did settle down on a stable behaviour solving the task. The following 2000 steps of evaluation were enough for the robot to visit most parts of the environment and encounter the potentially difficult edges and corners. If the learned controller cannot navigate the robot safely round the environment, it will be revealed within this acting phase, resulting in low accumulated rewards. With a timestep length of 150 ms, each run takes 15 minutes. As noted earlier each test was repeated 10 times, meaning that the total test time for one experiment was 2.5 hours. To make the experiments fully automatic, a random repositioning of the robot was run for 15 seconds between the experimental runs.

Parameters for the Q-learning algorithm in experiment 1:

- ϵ -greedy action selection ($\epsilon = 0.75$)
- timestep length: 150 ms
- discount factor: $\gamma = 0.9$
- comparing three different learning rate settings: $\alpha \in \{0.1, 0.2, 0.3\}$

The observed behaviour in the experimental runs was that in most cases the robot did solve the task, but only a few controllers managed to control the robot smoothly. Often the robot would negotiate a wall or a corner turning a couple of times before it succeeded in turning out into the open part of the field again. Sometimes the robot got stuck in an alternating sequence of left and right turns visiting two or three different sensor-states. This did only happen in the acting-phase, however, because in the learning-phase random explorative actions would sooner or later turn the robot away from the walls. The results of the comparison of the different learning-rates can be seen in the graphs in figure 5.5. At left is the learning-phase graph and at the right the acting-phase graph. Each curve shows an average of the 10 runs. The rewards depicted are average value for 100 consecutive timesteps. A general observation from these graphs is that gained reward increases throughout the 4000 learning-steps indicating improvement in the behaviour of the robot. In the acting-phase the learned policy is fixed and it can be seen that the curves are almost horizontal. A small decay in reward gained in the early parts of the acting-phase is to be expected though. The reason for this is that some controllers did manage to navigate the robot for some time in the beginning of the acting phase, but eventually got stuck somewhere in the

field. As can be seen from the curves, no clear conclusion for the best value of the learning-rate can be drawn. It seems that learning rate 0.3 performs a little bit worse in the acting-phase, but learning rates 0.1 and 0.2 perform equally well. It was decided not to any apply statistics on the experimental data because of the limited number of runs performed with each setting of the parameters. This is not an ideal situation, but as mentioned earlier real robot experiments are very time consuming. The experiments in this text can only be used to outline some general results that would have to be verified by repeated experiments. However, the experiments can be used to get an idea of the effect of different learning parameters, exploration strategies, and learning methods.

5.2.2 Conclusion

It is difficult to find the “best” learning rate. Settings that resulted in the fastest progress in the learning-phase, did not necessarily behave at the best in the acting-phase. From this experiment there is no evidence that 0.2 is better than the other settings, but a choice had to be made so in the following Q-learning experiments, a learning rate of 0.2 will be used.

5.3 Investigating the Effective State-space

In section 5.1.4 the claim was that the effective number of state in this experiment is below the possible 64 states because some of the sensor configurations will never be experienced by the robot. Statistics on the state visits during Experiment 1 will be presented here. Sensor activation in some of the states is depicted in figure 5.6. The top row shows the 6 most frequent states and the bottom row shows the 6 least frequent. The data is collected from the 10 runs with $\alpha = 0.2$ in experiment 1.

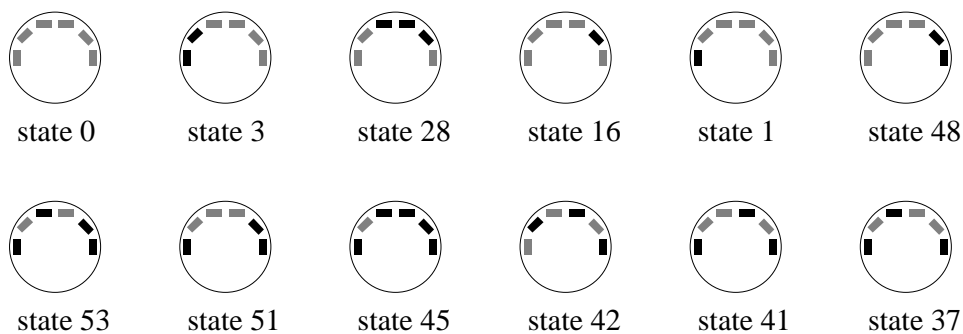


Figure 5.6: *Sensor activations the different states. A black sensor is on and a gray sensor if off. The top row shows the most frequent states. The bottom row shows the least frequent states.*

The 6 bottom states of figure 5.6 were not met in any of the runs, and another 5 states were only met once in the total of 60,000 timesteps. Actually the data shows that the robot spent 94 percent of the time in the 16 most frequent states. A table showing some of the frequencies can be seen in figure 5.7, and a histogram showing the distribution of the state frequencies can be seen in figure 5.8.

State	#visits	State	#visits
0	28521	39	1
3	3370	38	1
28	2887	35	1
16	2887	21	1
1	2620	53	0
48	2573	51	0
24	2250	45	0
12	1849	42	0
6	1591	41	0
2	1515	37	0

Figure 5.7: *State visit frequencies. At the left the most frequent states. At the right the least frequent states.*

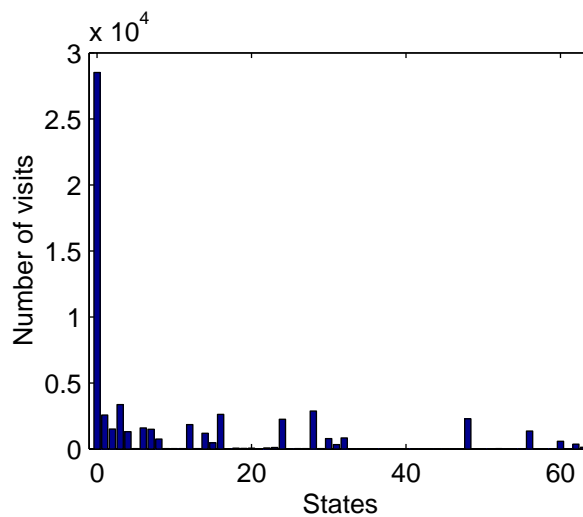


Figure 5.8: *Histogram of state visit frequencies.*

5.3.1 Conclusion

What is the effect of this distribution? The good part is that it is easier to find a good policy for 16 different states than for 64 states which is what the robot does 94 percent of the time. The result is faster convergence to a good policy. The bad part is that robot in the acting may be “unlucky” to find itself in a state that it did not visit during learning, or only visited a few times. In this state the “best” action may be almost random, because the policy in this state is based on very few real-world experiences. This problem resulted in poor performance of some of the learned controllers in the acting phase even though the learning phase did not indicate any problems.

5.4 Experiment 2: Different Timestep Lengths

In this experiment the effect of different timestep lengths will be investigated. The setting of the timestep length can be vital for the overall learning performance. When the raw sensor values are mapped into a coarse discrete state-space, the timestep length has a direct effect on the transition probabilities between the states. If the timesteps are very short the sensors are often sampled. The result is that the agent has to perform a large number actions before a state transition occurs. This makes it difficult to see differences in the effect of different actions which is not desirable. If the timesteps are very long on the other hand, a single action may bring the agent into a completely different state, skipping a lot of neighbouring states. This is not desirable either, because it may result in convergence problems for the learning algorithm. Furthermore, too long timesteps will result in slow reactions for the robot.

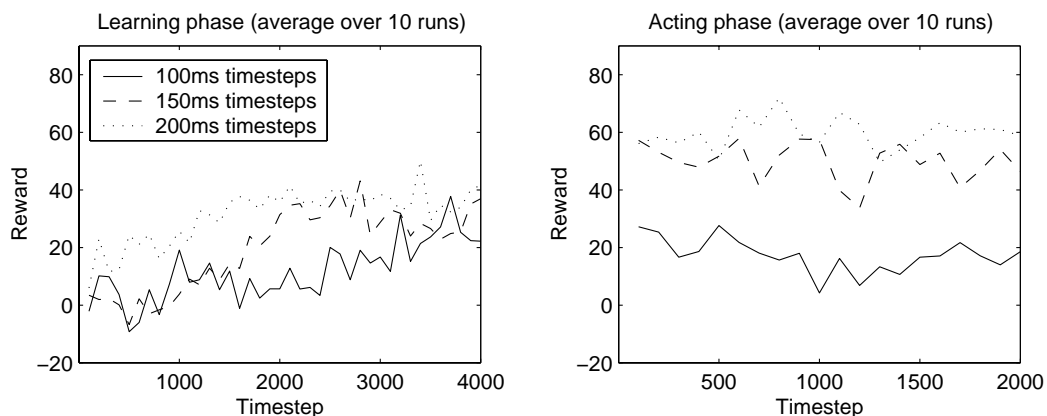


Figure 5.9: *Effect of the timestep length.*

Timestep lengths of *100 ms*, *150 ms* and *200 ms* were tried in this experiment.

Longer timestep lengths would result in too slow reactions of the robot so they were discarded beforehand. The experimental data can be seen in figure 5.9. In this case it is easier to draw conclusions from the graphs. In the case of *100 ms* it can be seen that the controllers only make slow progress during the learning-phase and in the acting phase the performance is very poor. The observation of the behaviours was that in four of the 10 runs, the robot got stuck early in the acting-phase and did not receive any positive reinforcement from that point on.

With a timestep setting at *150 ms* and *200 ms*, the progress during the learning phase is a bit faster. At *200 ms* the initial progress is very high, but from around timestep 2000, the two graphs stay close together. Throughout almost all of the acting phase, the graph of *200 ms* timesteps lies above the *150 ms* graph.

5.4.1 Conclusion

The conclusion on this experiment is that with this sensor-space division a timestep length of *100 ms* is too short, *150 ms* is better but, *200 ms* is the best. It has to be noted that it is the value of each update of the Q-table that is compared in this experiment. In real-time each run with *100 ms* steps took 10 minutes, whereas the runs with *200ms* steps took 20 minutes each of course. Even with this in mind the *100ms* steps proved to be inefficient. If we compare the *100ms* case at step 4000 in the learning-phase with the *200ms* case at step 2000 there is no significant value of doubling the number of updates per second.

In the following experiments a timestep length of 200ms will be used. To shorten down the experimental time, the learning-phase is reduced to 3000 steps and the acting phase to 1500 steps. The total time is 15 minutes per run with these settings.

5.5 Experiment 3: Effect of the Discount Factor

The effect of different settings of the discount factor γ will be investigated in this experiment. Recall the Q-learning update rule from chapter 4:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a')), \gamma \in [0; 1[\quad (5.1)$$

Intuitively, the discount factor describes value future rewards. More formally a reward received k steps into the future is worth only γ^{k-1} of what it would be worth if it was received immediately. With the gamma factor set to zero only the immediate reward r is taken into account. As the gamma factor approaches 1, the more importance is attached to future rewards. When the reward is delayed, as in case of the reward function described in section 5.1.6, we have to be sure that the gamma factor is high enough to allow the immediate reward from the

goal state (`state 0`) to propagate back to any accessible state before vanishing.

In Experiment 1 and 2 γ was set to 0.9 the same value used in [Stø99] on a similar setup. Only one additional setting ($\gamma = 0.5$) was tried in this experiment. The experimental data is shown in figure 5.10. As can be seen from the graphs, the controllers performed equally well in the two cases so not much can be said except that more runs with both these and different settings are needed. What can be said is that with γ set to 0.5, the numerical values in the Q-table were of course smaller than in the $\gamma = 0.9$ case, but they did propagate out to every accessed state, and no significant decrease in performance compared with the $\gamma = 0.9$ case was observed. This is perhaps a bit surprising at first, and seemingly in conflict with the results reported in [Stø99]. When thinking about it the explanation can be that when using a purely delayed reward function with a single goal state with positive reward, the discount factor only influences the overall sizes of the Q-values, but not the relative size between values of different actions. With a more complex reward function with rewards in several states this would not be the case. In [Stø99] the Q-table was stored in a neural network. In this case the relative value of different states has an impact on the policy stored. It was reported that the controllers had stabilisation problems when a discount factor of 0.5 was used. In that case every little update of the network would change the effective policy.

5.5.1 Conclusion

The setting of the discount factor in this experimental setup is not that crucial. Both settings tried worked fine. In the following experiments the discount factor will be kept at 0.9.

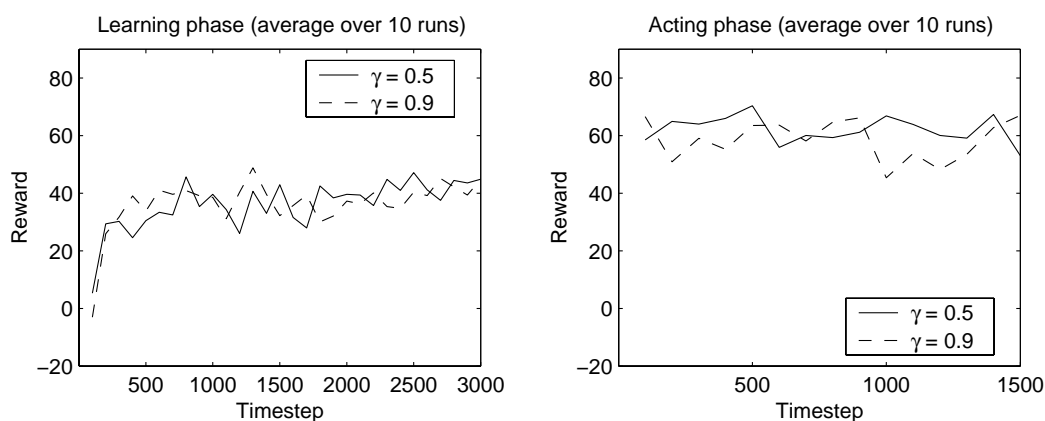


Figure 5.10: *Effect of the discount factor.*

5.6 Experiment 4: Action Selection Methods

So far the action selection method used has been the simple ϵ -greedy method. In this experiment the two methods *softmax* and *optimism in the face of uncertainty* described in chapter 4 are used.

5.6.1 Softmax Action-selection

In the softmax action-selection method with a Boltzmann-distribution, the action a in state s is selected with probability:

$$Pr(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \quad (5.2)$$

The main difficulty when using *softmax* is the setting of the temperature parameter τ . This difficulty arises from the fact that the effect of a specific value of τ depends on the relative distribution of the Q -values in the state in question. This distribution changes dynamically during an experimental run. Initially all Q -values are usually set to zero, but during the run, the values slide up and down as more knowledge about the environment is discovered. The setting of τ effects which actions are picked and then how the Q -values change, so the problem has a dynamic nature. We would like an exploration strategy where initially, when nothing is known about the environment, every action is picked with equal probability, but when more knowledge is gained the promising actions are tried more often. This can be accomplished by starting with a “high” temperature and slowly decreasing it. There is a danger however, that some actions may be starved out. If they are not picked in the beginning of the experiment and the other actions of that state get high Q -values, their probability of ever being picked rapidly decreases with the decrease of τ .

To get an idea of the effect of the temperature parameter and to gain knowledge of how to set it, the distribution of Q -values during one of the runs in experiment 3 was investigated. A snapshot of the action-values of two states during a run can be seen in figure 5.11. As can be seen the values of **state 0** (the robot is clear of all obstacles) are higher than the ones in **state 7** (inputs on sensors 0, 1 and 2). In **state 0** the best action is to go **forward** (action 0) and in **state 7** the best action is to **turn right** (action 1) in this case.

$Q(s,a)$	$a = 0$	$a = 1$	$a = 2$	$a = 3$
$s = 0$	855	720	690	512
		...		
$s = 7$	-10	144	2	-70
		...		

Figure 5.11: A part of a Q -table during an experiment.

Graphs of the corresponding selection probabilities under *softmax* with varying temperatures can be seen in figure 5.12. It is clear from these graphs that when the temperature approaches zero, softmax becomes a greedy action selection. On the basis on these graphs and the discussion of the requirement to the action selection mentioned above, it was decided to start the new experimental runs with a temperature of 100 and decrease it linearly to 50 during the 3000 timesteps. With a different reward function, or a different action-set the Q -values would of course be distributed differently and other settings of the temperature would have to be considered.

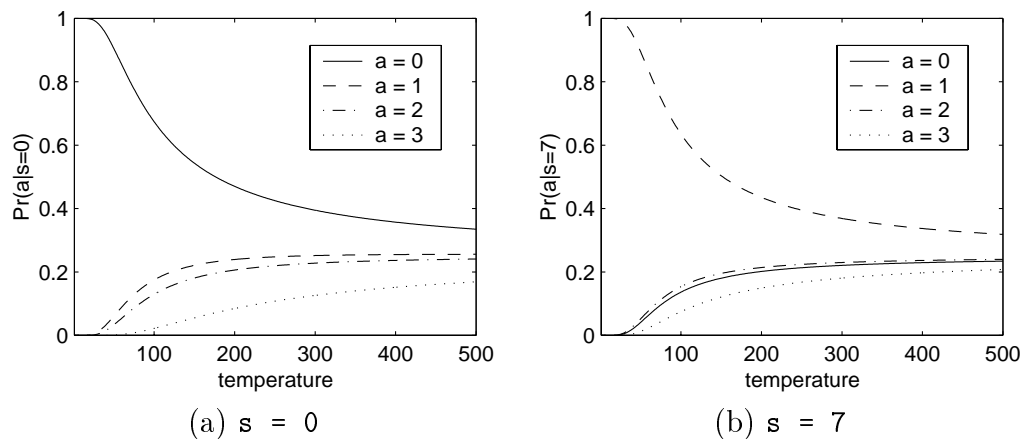


Figure 5.12: *Softmax* action selection probabilities.

5.6.2 Optimism in the Face of Uncertainty

When implementing the *optimism in the face of uncertainty* heuristic described in chapter 4, the only thing that has to be done is to initialise all entries in the Q -table with some high *optimistic* value, and always select the action with the highest value. The initial value 500 was chosen because it is higher than the values to be expected for most of the states, and a bit lower than the value to be

expected in the goal state (`state 0`).

The drawback of this approach is of course that it assures some initial exploration, but when the Q -values settle down, no more exploration is done. Therefore it is not well suited for non-stationary problems because its drive for exploration is inherently temporary [SB98].

5.6.3 Results

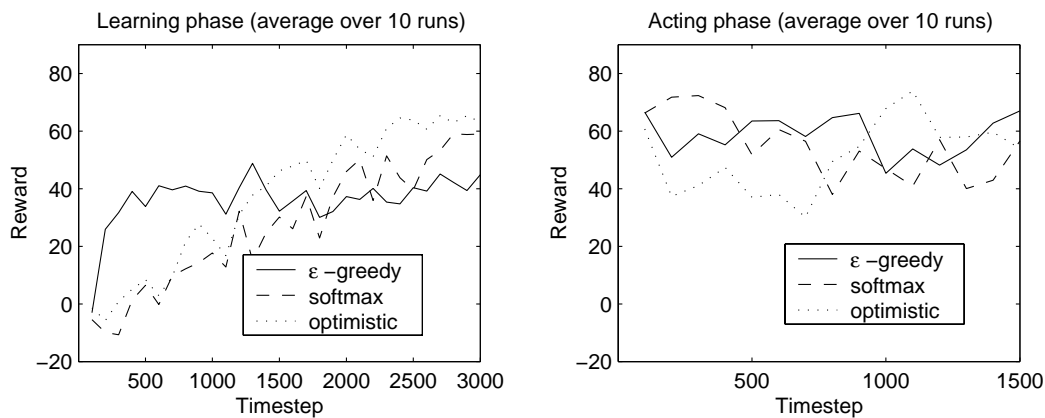


Figure 5.13: *Performance of the three action selection methods.*

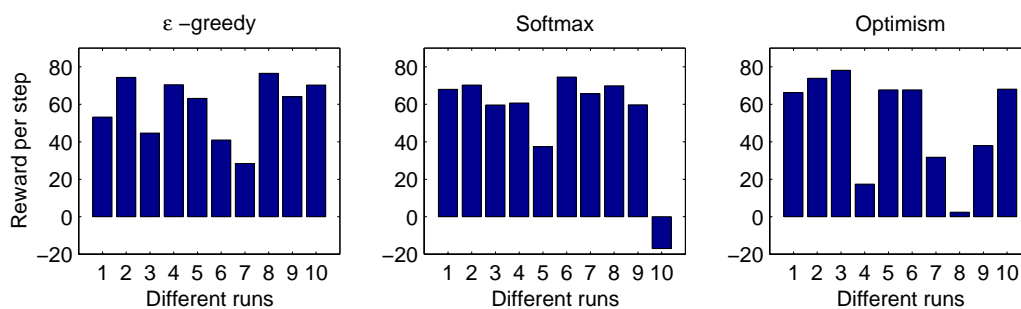


Figure 5.14: *Average reward per step in the acting-phase of three action selection methods.*

The experimental results of comparison of the three different action selection strategies can be seen in figure 5.13. In the learning phase it can be seen that the ϵ -greedy graph very fastly reaches a high level of performance and then levels out. The *softmax* graph continues to improve throughout the learning-phase. This difference simple comes from that fact *softmax* takes a lot of explorative

steps in the beginning when the temperature is high. When the temperature is decreased, more greedy steps are taken, and the received reward increases. The graph of the *optimistic* case follows the *softmax* graph despite the fact that the methods are very different. It improves slower than the ϵ -greedy graph even though all steps taken are greedy, but it succeeds in reaching that same high level as *softmax* by the end of the learning phase. It is difficult to judge the three methods only on the basis of the learning-phase because of the differences in the amount of explorative steps. Now the graphs of the acting phase comes to their right. From these graphs it looks like the performance is almost equal with a small advantage to ϵ -greedy. The average performance in the 10 individual runs is shown as histograms in 5.14. Taken overall ϵ -greedy has the highest average performance, but softmax also reaches a high performance in most of the runs. The “unlucky” controller number 10 in softmax blurs the picture though.

5.6.4 Conclusion

All the three methods worked to some extent but with the *optimism in the face of uncertainty* heuristic fewer controllers reached a high level of performance. It was decided to continue with the *softmax* action selection because of its interesting properties connected with the temperature parameter and its stable improvement seen in the learning-phase of this experiment.

5.7 Experiment 5: Model-based versus Model-free

In this experiment the model-based reinforcement learning methods *Dyna* and *Prioritized Sweeping* are compared with Q-learning used so far.

5.7.1 Dyna

Recall, that in the *backup* rule of Dyna described in chapter 4 the learning-rate is gone. Only the statistical model of state transitions and rewards (\hat{T} and \hat{R}) is used:

$$Q(s, a) = \hat{R}(s, a) + \gamma \sum_{s' \in S} \hat{T}(s, a, s') \max_{a'} Q(s', a') \quad (5.3)$$

When using Dyna in this form we only have to worry about setting the parameter k which is the number of additional updates per step. The value of k was arbitrarily set to 8 which turned out to give satisfactory results.

5.7.2 Prioritized Sweeping

The Prioritized Sweeping algorithm was implemented using the heap based priority queue data-structure described in [Swa94]. Prioritized Sweeping (PS) also uses equation (5.3) as the *backup* rule. The same value of k as in Dyna was used. In PS k denotes the maximal number of updates of the Q -table per step. How often the all of the k update are necessary depends on the threshold value δ (see figure 4.6). Only if the priority of a state exceeds this threshold the state is inserted in the priority queue. Some experimenting with the significance of the value of δ took place. It was found that a value of 5 was suitable. With this value the priority queue did empty when the robot moved around in the open space with no inputs on the sensors, and thus no new information. With a lower value of δ the priority queue did never empty and small updates of the Q -table were constantly made. With higher values of δ something “very interesting” had to happen before the algorithm bothered to do any updates.

5.7.3 Results

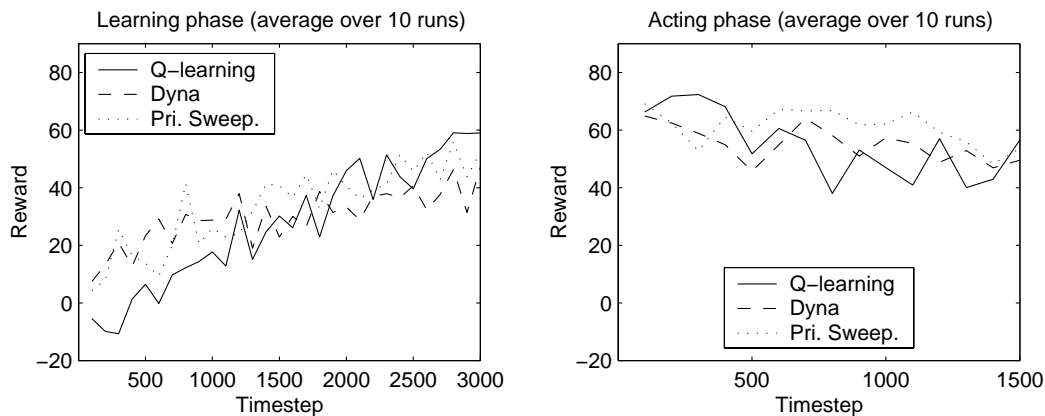


Figure 5.15: *Comparing the different learning methods.*

The reward graphs of the runs in this experiment can be seen in figure 5.15. It can be seen from the graphs of the acting-phase that all three methods succeeded in reaching a high level of performance. No big differences in the reached performance are seen, but the fact of the matter is that every controller converged to an almost optimal behaviour within the learning phase so major difference in acting performance were not to be expected. Actually Prioritized Sweeping but also Dyna did perform a bit better than pure Q-learning in the acting phase. When observing the robot in the field, the Q-learning robots when meeting a wall would start turning in the right direction, but make maybe one or two jerks in the other direction before getting clear of the wall. These jerks indicate that the

state values have not been entirely stabilised over the states visited by the robot. In the Dyna and especially in Prioritized Sweeping a more smooth behaviour was generally observed. When looking at the speed of convergence in the early part of the learning-phase, it can be seen from the graphs on the left hand side of figure 5.15 that both Dyna and Prioritized Sweeping converged faster to a high-reward giving behaviour. Later on in the learning-phase Q -learning manages to keep up at around step 2000 and by the end of the learning-phase Q -learning seems to get the most reward.

It has to be noted that in contrast to experiment 4 for example, it is fair to compare the speed of convergence in the learning-phase graphs in this experiment. The reason for this is that the same exploration strategy is used in all three methods, so no-one can profit from a larger number of greedy steps.

	Learning-phase (first 1000 steps)	Learning-phase (all 3000 steps)	Acting-phase
Q-learning	3.58	27.39	54.88
Dyna	21.58	29.81	55.06
Prioritized Sweeping	18.57	33.60	61.11

Figure 5.16: *Average reward per step in the three learning methods.*

The table in figure 5.16 summarises the results of this experiment. When looking at the average reward gained in the first 1000 steps of the learning-phase, it can be seen that the model-based methods indeed improve much faster than Q -learning in the beginning. It can also be seen that Prioritized Sweeping performs a bit better in the acting phase than the other methods.

5.7.4 Conclusion

In this experiment Dyna and Prioritized Sweeping were seen to converge faster than Q -learning to an near-optimal behaviour. As expected, however, the superiority of Dyna and Prioritized Sweeping was no near as big as in the deterministic and stochastic two-dimensional grid-worlds reported in [MA93] and [PW93]. In these cases Q -learning was outperformed by orders of magnitudes. The performance measures reported in these simulation experiments were the number of steps before converging to the optimal policy. It is intuitively clear, that in Markovian domains the model-based enhancement can improve one-step Q -learning. The transitions are either deterministic or have fixed probabilities so when every transition has been experienced a sufficient number of times the remaining problem is “just” a numerical optimisation problem. Between the actual steps taken in

the environment, the model-based methods do additional updates, thus speeding up the generalisation over states. In situated non-Markovian problems there is no replacement for real-world experiences. The model-based methods only resulted in relatively small improvements over pure one-step Q-learning. An explanation can be that in non-Markovian environments the state transition probabilities are dynamic. If at one point in time optimisation is carried out with respect to the current model, future real-world experiences may alter the probabilities and thus reduce the worth of the optimisations done. It was seen however, that solutions found using Prioritized Sweeping resulted in a more smooth behaviour of the robots. In other words, the resulting policies were more self-consistent with respect to the size of the Q-values of neighbour states than the policies found using Q-learning.

5.8 Experiment 6: Extending the Action-set

In the first 5 experiments the robot had only had 4 different actions to choose from in each step. In this experiment, the number of actions for the robot to choose from will be doubled. It will be investigated what effect this change has on the convergence properties of the learning process. Furthermore, it will be investigated if this change has an effect on the relative strengths of the three different learning methods.

Action	left motor	right motor
Forward	6	6
Turn right	6	-6
Turn left	-6	6
Backward	-6	-6
Forward right	10	-2
Forward left	-2	10
Backward right	-10	2
Backward left	2	-10

Figure 5.17: *The extended Action-set.*

The action-set is extended with four new actions (see the table of figure 5.17). The four new actions are {**forward right**, **forward left**, **backward right**, **backward left**}. Similar actions were used in [WHH98]. At first the four actions were implemented by just blocking one wheel and setting a forward or backward velocity on the other wheel. This turned out to be not such a good idea. The effect of the action **forward right** = (8,0) for example was that the robot often got stuck on the corners of the field in a position with no input on the sensors.

In addition, In the acting-phase the **forward right** or **forward left** did not change the state of a robot driving against a wall, so it would be stuck forever. After a few iterations the motor values in the table above were found to be satisfactory. With these motor speeds the actions are more likely to change the state of the robot when stuck against a wall, making it easier for the robot to get free again. The same settings of the parameters were used as in Experiment 5 and the reward function remained unchanged as well.

5.8.1 Results

When observing the robot with the new action-set available, the behaviours were quite different than before. The robot often used the new actions. The resulting behaviour was that the robot when meeting a wall while driving forward most often did choose a **backward right** or **backward left** action until it was out in the open again. If the robot was not free of the wall when driving forward again it would repeat this action sequence a couple of times. When there was a smaller angle between the robot and the wall, the robot would start turning on the spot as before, but at some point in the turning it would take the **forward right** or **forward left** action speeding up the escape from the wall.

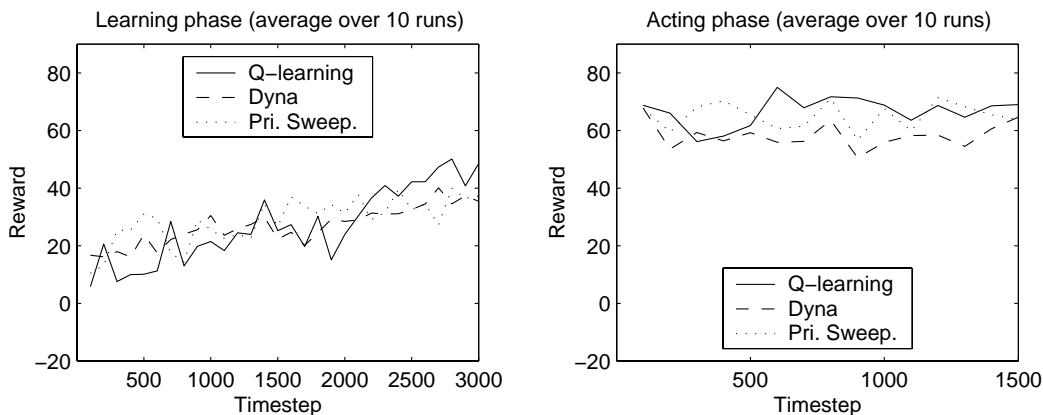


Figure 5.18: *Performance with the extended action-set.*

The experimental data of this experiment can be seen in figure 5.18. The first thing worth noting is that the Q-learning robot performs relatively better in the learning-phase compared to figure 5.15 of Experiment 5. Even though the model-based methods still improve faster in the beginning Q-learning is not so far behind anymore. The reason for this could be that in Experiment 5, where the robot was only allowed to turn on the spot, a robot meeting a wall in an angle close to 90° would have to pass through several states before reaching the goal state

again. In Q-learning this means that the Q-values had to generalise over many neighbouring states. This generalising over neighbouring states was seen to be speeded up by the model-based methods. With the extended action-set, a robot meeting a wall and backing up while turning will only pass through one or two states before `state 0` is met again. This fact may explain why Q-learning is improving faster with the extended action-set. Also Dyna and Prioritized Sweeping improved faster than in Experiment 5, but the difference was not as big as in the case of Q-learning.

In the acting-phase the performance in terms of rewards gained was slightly better than with the simple action-set. The summarised results can be seen in figure 5.19. As in experiment 5 the summarized performance of Prioritized Sweeping is better than that of Dyna. In contrast to experiment 5, Q-learning performs better than Dyna in this experiment and a bit above the level of Prioritized Sweeping. This is remarkable because the extra random updates of Dyna seem to do damage to the resulting learned policy with this action-set, and the focused updates of Prioritized Sweeping are not superior anymore.

	Learning-phase (first 1000 steps)	Learning-phase (all 3000 steps)	Acting-phase
Q-learning	14.83	26.97	66.60
Dyna	21.09	26.81	58.34
Prioritized Sweeping	22.16	28.87	65.19

Figure 5.19: *Average reward per step with the extended action-set.*

5.8.2 Conclusion

In this experiment it was seen that extending the action-set and in fact doubling the size of the Q-table did *improve* the convergence properties of the learning sequence. One might have expected a slow-down. It was also seen that the model-based method Dyna reduced the performance compared with Q-learning. The resulting behaviors were different than the “turn on the spot” policy derived in the previous experiments. The robot was seen to back up and turn a couple of times in order to get free of the walls instead of just turning on the spot as before.

5.9 Experiment 7: On-line Learning in a Dynamic Environment

In a sense it is against the spirit of reinforcement learning to have separate learning and acting-phases. The problem is that when the policy is frozen in the acting-phase the robot does no longer have the ability to adapt to changing environments. In the first place this ability was one of the arguments given in favour of reinforcement learning robots over the evolutionary robotics approaches. Recall that the reason for the controller to be fixed was that at some point it is necessary to evaluate the learned policy in order to be able to compare different methods. In this last experiment of the chapter an example of an on-line learning will be given. On-line learning is closer to fulfilling the *complete agent principle* of chapter 2 and as noted by Wyatt: “*On-line learning is similar in many respects to our common-sense notion of learning*” [Wya95].

5.9.1 Setup

In this experiment only a single individual will be used and a single run made. The learning will be done in a on-line way using a simple internal performance evaluator. This performance evaluator is simply the sum of rewards for the last 100 timesteps. When this value is above some, threshold the robot is performing well and there is no need to keep on exploring the environment. When the performance evaluator drops below the threshold, indicating that the robot is having problems in performing its task, some exploration initiated. A simple way of controlling the amount of exploration is to change the temperature parameter in the softmax action-selection method. As seen in figure 5.12 when the temperature approaches zero the method collapses into a greedy strategy. During the run of this experiment the robot will keep on learning all the time only the amount of exploration is varied.

In order to show a reinforcement learning agent’s ability to adapt to changing environments on a short time-scale, this experiment was initiated in the simple field shown in figure 5.20, but after 4500 timesteps, or 15 minutes, the dashed line bricks are removed and the robot is set to run in the more complex field used previously. The threshold value for the performance evaluator was set to 30. The softmax temperature was initially set to 100 and just as before slowly decreased to 50. When the performance evaluator went above the threshold, the temperature was rapidly, but not instantly decreased to zero, and when above the threshold again the temperature was increased with the same speed.

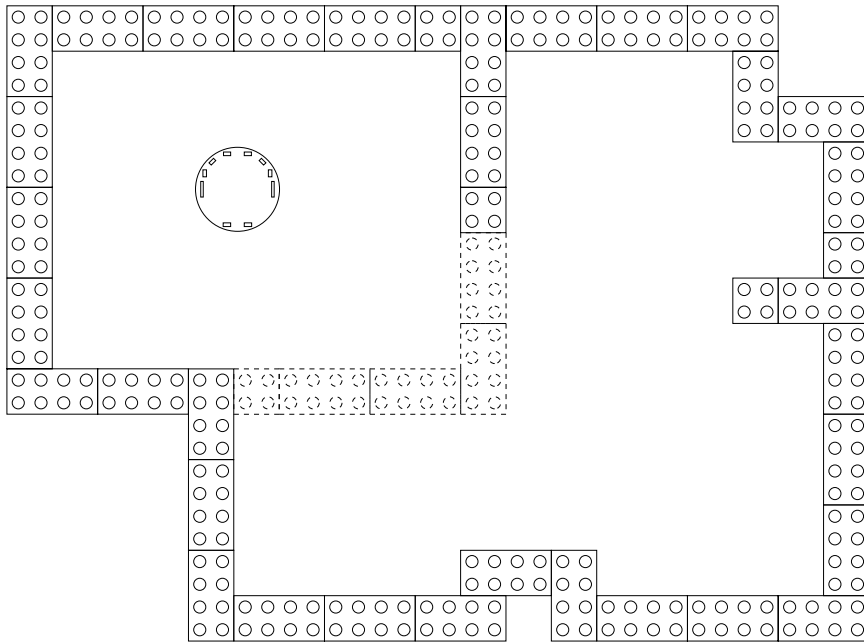


Figure 5.20: *The simplified field. The dashed-line Duplo bricks are removed halfway through the experiment.*

5.9.2 Results

The experimental data of this experiment can be seen in figure 5.21. The experiment was run for 9000 timesteps (30 minutes). The learning method used was Prioritized Sweeping with softmax action selection. As can be seen from the left graph, the robot learns the task within the first 500 timesteps, and after timestep 1300 no more drops below the threshold were seen within the first half of the experiment. After 15 minutes, at timestep 4500, the dashed walls were removed and the robot was set to drive around the original field. Although this field has different corners and edges no drop in performance was seen. Because of this some additional items were put in the field at timestep 6000, and for the last 3000 timesteps new items were added and removed from the field constantly. These new objects were cups, golf balls, Khepera robots, and LEGO Duplo bricks of various sizes (see figure 5.22). The robot did only have to initiate a new exploration phase a few times. Towards the end the robot had very little free space to move on because of the new objects being placed in the field. This explains the drop of the graph towards the end, but the robot did still navigate smoothly around the field.

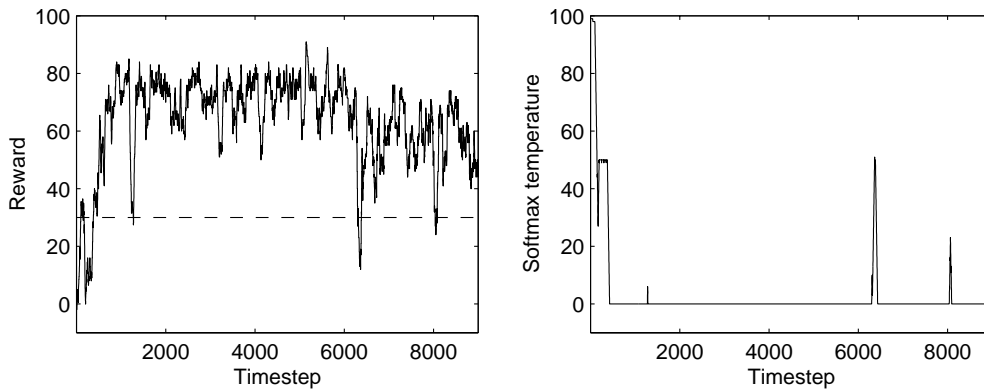


Figure 5.21: *On-line learning. The results of the single 30 minutes run. The left graph shows average rewards. At each step the reward averaged over the 100 previous steps is shown. The right graph shows the softmax temperature. When the left graph drops below the dashed line, the temperature is increased resulting in some temporary exploration until performance improves again.*



Figure 5.22: *On-line learning. Objects were place and removed.*

5.9.3 Conclusion

In this experiment an example of *on-line* reinforcement learning was shown. An internal performance estimator governed the exploration strategy of the robot. The robot was seen to perform the task within the first 500 timesteps. When the robot was put in a more complex environment, no drop in performance was observed. Initially the robot was placed in a simple environment, and later on when the basic behaviour had been learned the robot was placed in a more complex environment. Even though new edges and corners were experienced in the complex field, the basic behaviour necessary was the same and the robot quickly adapted without an observable drop in performance.

Chapter 6

Visually Guided Obstacle Avoidance

In chapter 5 it was seen that a robot was able to learn the task of obstacle avoidance using IR proximity-sensors as inputs to the learning algorithm. In chapter 3 it was argued that if the autonomous agents are to move towards more complex navigation strategies some long-range sensors like a video-camera has to be added to the robots. To take a step in this direction, a camera is added on top of the Khepera robot in this chapter and the learning task will be repeated. With the use of a visual input-sensor the robot is able to solve the task in a slightly different way, maintaining greater distances to the walls of the field.

6.1 Experimental Setup

Several changes to the experimental setup had to be done when adding the camera. They will be mentioned here, but the level of details will be lower than in chapter 5.

6.1.1 Adding a Camera

A photo of the Khepera robot with the K2D video extension module can be seen in figure 6.1. The robot hardware does not allow on-board processing of the camera-output, so the video-image is transmitted through an extended aerial cable and plugged into a frame-grabber on the host computer. Here some image-processing is done (see section 6.1.2) and the output of the image-processing algorithm is mapped into a new sensor-space as input to the learning algorithm (see section 6.1.5).

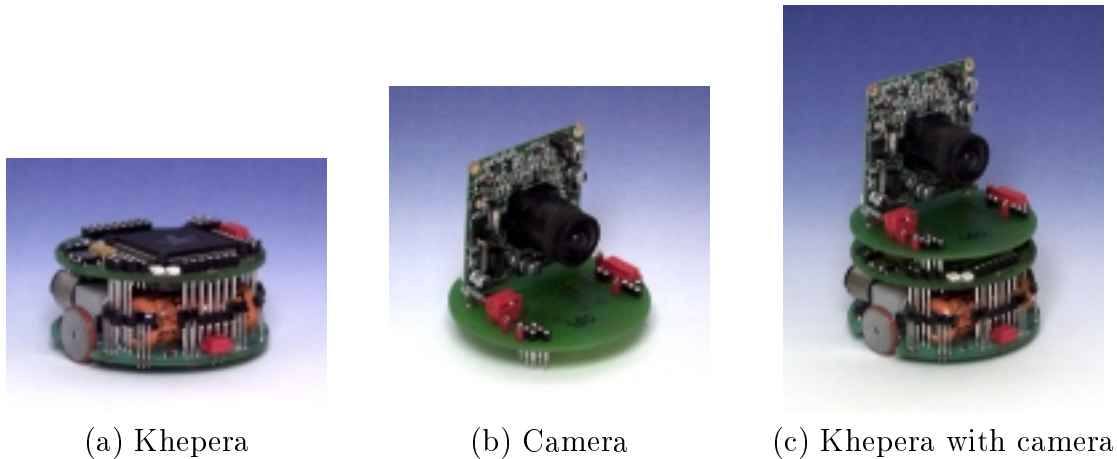


Figure 6.1: *The Khepera robot with the K2D video extension module.*

6.1.2 Image Processing

The focus of this text is *not* robot vision, so only a few methods will be mentioned and a general discussion will not be given. The main source of inspiration for the visually guided obstacle avoidance task was a project reported on the subject in which the Pebbles III Robot was able to avoid obstacles in unstructured environments [LBG97]. In this work fused edge detection modules were used to detect obstacle boundaries. If a robot is moving around on flat grounds there is a direct correspondence between the height of the lower bound of an obstacle on the image plane, and the distance to the object; the so-called *ground plane constraint* [Hor93] is applied. In the control program implemented on the Pebbles robot, the mapping from obstacle boundaries to motor commands was hand-coded.

Edge Detection

In the field of computer vision and image processing the state-of-the-art methods for edge detection are the *Canny Edge Detector* and the *Shen-Castan Edge Detector* (see [Par97] for a description). However, both these methods are computationally very heavy and not well suited for real-time robot vision. A faster, and thus, more suited method is to apply template-based convolution filters on the image-data. This approach was taken, and the filter used was the *Kirsch template* described in [Par97]. This edge-detection worked quite well, but the physical hardware setup induced another problem. The problem was that when the robot got too close to an obstacle, the lower edge of the obstacle boundary disappeared from the video-image. See figure 6.1 (c) an imagine the robot positioned in front of an obstacle. With this configuration the camera cannot see the lower edge of an obstacle immediately in front of the robot. At the distance to an obstacle where the lower obstacle boundary disappears from the video-image the obstacle cannot yet be detected by the frontal IR sensors. If the IR sensors

could have sensed the obstacle, a supervising behaviour could have told the robot to turn away from it. This was not possible, with this hardware setup, and the result was that it was impossible to construct a mapping from detected obstacle boundaries to visual states that would make learning an obstacle avoidance behaviour possible. A solution had to be found. The first idea was to tilt the camera in order to reduce the size of the blind spot, but when the camera was tilted more than a few degrees the robot-base itself became visible in the image shadowing for the edge to be detected. Other morphological changes, e.g. moving the camera-house or adding a mirror, were not possible so a software solution had to be found.

Colour Image Segmentation

The idea of colour image segmentation is to segment the image into regions containing the same colour or texture. Advanced methods of doing this exist. Recently, the JSEG algorithm has successfully been used to divide images into regions containing roughly the same texture [DMS99]. A much more simple method turned out to be sufficient in the problem at hand, however. The environment setup was altered a bit at first. The surface colour was altered from black to green and a halogen lamp lighting the field from above was added. Now simple thresholding calculations on the RGB-values¹ were carried out. Two criteria were used:

- The green pixel-intensity has to exceed the red pixel-intensity by a given threshold.
- The absolute difference in intensity of the red and the blue values must be less than another threshold.

When these two criteria are fulfilled, the pixel is classified as belonging to the green surface of the field. Now the image is divided into six columns and each column is scanned from below. At each column-line if more than half the pixels have been classified as green, the whole column-line is classified as green. The result of this processing is a vector of six values each denoting the vertical position of the first non-green line of that column. These values correspond to the amount of free space in front of the robot in six image columns. An example of the image processing can be seen in figure 6.2.

In contrast to the application of the edge-detection this colour segmentation does not fail when the robot approaches a wall. The processing will just give the result that there is no free space in front of the robot, and the robot can be classified as being in an *unsafe* state.

¹Each image pixel consists of red, green and blue (RGB) values of light intensity.



Figure 6.2: *The image processing. At the left the original image. In the middle the colour segmented image. At the right the 13 regions of the visual state-space division.*

6.1.3 The Task

When using a camera-sensor the task was redefined as to maintain a distance to the wall where the green surface is always visible in the video-image. In other words, the robot has to stay in a situation with *free space* in front of it, and thus staying clear of the *unsafe* state described above.

6.1.4 The Environment

The field of the experiment in chapter 5 was altered a bit for this experiment. With the task for the robot being the one described above, more free space in the field was needed so the internal walls were removed. The altered field for the robot to move around in can be seen in figure 6.3.



Figure 6.3: *The field for visually guided obstacle avoidance.*

6.1.5 Sensor-space Division

The resulting vector of the vision-processing algorithm was mapped into 14 different visual states. The first and the last image column were discarded. Then three vertical lines were added. Finally the regions above the top vertical line were collapsed into one, dividing the image into 13 regions (see figure 6.2). The state of the robot is now the identity number of the small region containing the first non-green line when scanning from below. If all of the small regions are green, the robot is classified as being in the *very safe* state (the top region), and if one of four lower regions does not contain any green lines the state of the robot is classified as being in the *unsafe* (the 14th state).

6.1.6 Action-set Construction

As in Experiment 6 and 7, an action-set of size 8 was decided upon. The different actions can be seen in the table of figure 6.4.

Action	left motor	right motor
Forward	6	6
Turn right	6	-6
Turn left	-6	6
Backward	-6	-6
Forward turn right	10	2
Forward turn left	2	10
Forward right	8	4
Forward left	4	8

Figure 6.4: *The action-set for the visually guided task.*

The argument for using these actions is that they were seen to work on a quick hand-coded solution to the navigation problem.

6.1.7 The Reward Function

The reward function used for the visually guided task had the following form:

```

if (s' == UNSAFE)
    reward = -200;
else if (action == BACKWARD)
    reward = -200;
else if (s == VERY_SAFE && action == FORWARD)
    reward = 100;

```

```
else
    reward = 0;
```

Here s denotes the old state and s' denotes the new state of the current experience tuple. Again it can be seen that a delayed reward method is used. We punish the robot for getting into the *unsafe* state and for moving backward and we reward it for moving forward in the state with plenty of free space.

6.1.8 A Rescue Behaviour

When the robot moves into the *unsafe* state, the visual input can no longer be used to control the robot behaviour. The solution to this problem was to let a *rescue behaviour* take over the control of the robot. The rescue-behaviour uses a simple Braitenberg-inspired obstacle avoidance guided by the IR-sensors. When the robot has been navigated out of the *unsafe* state, control is returned to the learning algorithm. Another natural solution could have been to use one of the learned behaviours from the previous experiments or to run the two learning algorithms simultaneously. To keep focus on the visually guided learning, the simple rescue behaviour was used.

6.2 Experiment 8

With the experimental setup done, the experimental runs were ready to be made. Again each run was repeated 10 times for each of the three learning methods Q-learning, Dyna and Prioritised Sweeping.

Observed Behaviours

The observed results were that the robot quickly learned a behaviour of avoiding the walls by turning when approaching a wall. The behaviour was never learned to perfection though. Even with the controllers that gained the most reward, the robot would sometimes end up in the *unsafe* state for the rescue behaviour to take over, but for long periods of time the best controllers did avoid this state. The shaping of the environment by cutting off the corners (see figure 6.3) helped a lot with this problem.

The behaviours learned by the controllers were sometimes quite different. Some of the controllers would make the robot turn on the spot whenever a wall got into the field of vision. The effective behaviour of this strategy looked a lot like the obstacle avoidance behaviours learned, using the simple action-set in experiments 1-5. The difference was that in this case the turning was initiated at longer distances from the walls. Some other controllers preferred to use the **forward**

turn actions when a wall was detected. The effect of this strategy was a wall-following-like behaviour, where the robot did circle the field for several rounds before approaching a wall in an angle that would make it turn sharply and then follow the wall the other way around the field. The Q-learning controllers seemed to prefer the first strategy whereas the Prioritised Sweeping controllers seemed to prefer the latter. These are just some overall observations though, with a lot of exceptions from the general rule.

6.2.1 Results

The experimental data of this experiment can be seen in the graphs of figure 6.5. It can be seen that the robots in all of the three methods learned a reward giving behaviour within the first 500 timesteps of the learning-phase. After this initial progress the reward graphs level out and only little progress is made through the rest of the learning-phase. No significant differences in the learning speed of the three methods can be observed from these graphs. In the acting phase the graphs of the three methods are also quite similar. The Prioritized Sweeping method performs a bit better with Q-learning taking the second spot. The summarised rewards of the acting phase underlining these observations can be seen in figure 6.6.

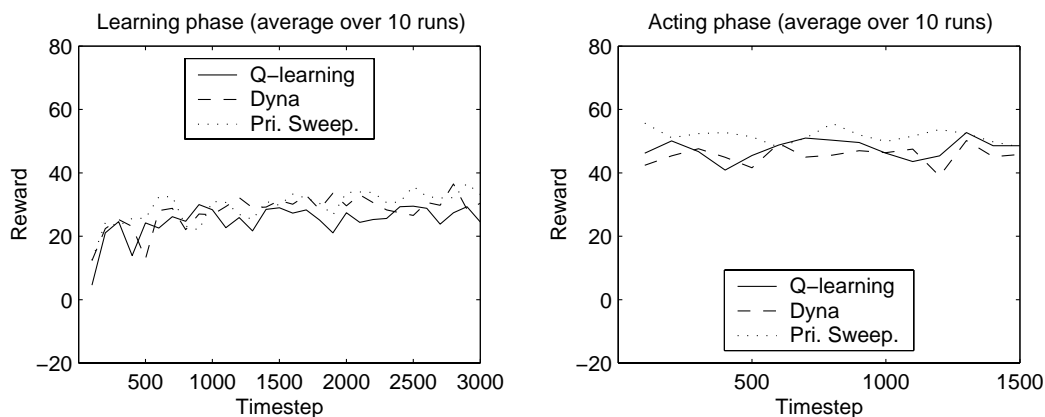


Figure 6.5: *Performance in the visually guided obstacle avoidance task.*

In section 5.1.7 it was argued that a *qualitative* measure of performance would be preferable, but in most cases, if an automatic logging of the data is wanted a *quantitative* measure of performance is what needs to be settled for, like the reward measure used so far. In this experiment luckily there is another useful performance measure, that is the number of times the robot bumped into the walls, or rather the number of times the robot got in the *unsafe* state and the rescue behaviour had to take over. If this performance measure is used, the

	Learning-phase	Acting-phase
Q-learning	24.84	47.61
Dyna	27.88	45.52
Prioritized Sweeping	29.47	51.71

Figure 6.6: *Average reward per step in the visually guided task.*

picture of the relative strengths of the three methods is quite different from the results seen from the reward graphs. The total number of bumps of the three methods in the acting-phase can be seen in figure 6.7. With this performance measure, the behaviour of Prioritized Sweeping is significantly better than that of Q-learning and Dyna in the acting-phase.

	Learning-phases (30,000 steps)	Acting-phases (15,000 steps)
Q-learning	598 bumps	257 bumps
Dyna	596 bumps	307 bumps
Prioritized Sweeping	429 bumps	107 bumps

Figure 6.7: *Total number of bumps in each method.*

6.2.2 Conclusion

In this experiment a visually preprocessed image was divided into 14 different sensor states and given as input to the learning mechanism. This input was sufficient for the robot to learn the task of visually guided obstacle avoidance. In addition the experimental data showed that there was a difference in the results using two different performance measures. In both cases Prioritized Sweeping performed the best, but when looking at the number of bumps the difference in favour of Prioritized Sweeping was more pronounced. To explain this the experimental setup and the effect of the *rescue behaviour* have to be looked at more carefully. The case is that even the controllers that bump into the walls all the time will be rescued and turned around until they face the open field again. With just the minimal knowledge that it is best to go straight ahead when being in the *very safe* state, a controller will still gain a positive net reward. A solution to this problem could have been to raise the penalty for getting in the *unsafe* state, but it is doubtful that it would have changed the overall behaviour of the controllers. A more promising solution could have been to back up the robot and giving it one more shot for avoiding the *unsafe* state from approximately the

same position instead of making a total rescue. The robots learned to avoid the walls, but the behaviour was never learned to perfection. Possible improvements may be found by trying out different action-sets and state-spaces. Altering the physical structure of the robot by moving the camera and giving the robot a broader view might also help to improve performance.

Chapter 7

Final Discussion and Conclusion

In this last chapter, the work done in this text will be related to work done by other researchers in the field of reinforcement learning. A discussion of the obtained experimental results will be given and possible future extensions will be outlined. Finally, the conclusions of this text will be summarised.

7.1 Related Work

The field of reinforcement learning robots is a relatively new one. Here are some examples of work related to the experiments carried out in this text.

Learning Technique Comparison in Non-Robotic Domains

The starting point for the experiments made in this Master's Thesis was the work by [Sut90][MA93][PW93] on comparing pure Q-learning to model-based extensions on grid-world problems. As mentioned in the discussion of Experiment 5, in these domains Prioritized Sweeping were seen to speed up the number of steps before convergence to the optimal policy by an order of magnitude.

In [PW96] the *model-free* method $Q(\lambda)$, mentioned in chapter 4, was seen to outperform Prioritized Sweeping both on Markovian and non-Markovian problems. These results were just briefly mentioned, and not many details were given about the experiments, but the authors expressed surprise about their results. However, one weakness of the $Q(\lambda)$ algorithm is that it is *exploration sensitive*. Recall that in exploration sensitive methods, the policy being executed during learning has an impact on the convergence properties of the method. In [WHH96] some work has been done in modifying the algorithm and removing the exploration sensitivity. Here the domain was a small Markovian grid-world. It is uncertain how serious the exploration sensitivity of $Q(\lambda)$ is when applying it to non-Markovian tasks because in this case the theoretical guarantees of convergence to optimal do not apply anyway.

Learning Robot Experiments

The task of box pushing was investigated in [WHH98]. By behavioural decomposition the task was divided into the three subtasks of *box finding*, *box pushing* and *becoming unstuck*. A LEGO robot was to separately learn each of the three subtasks. In this experiment Q-learning and the *corrected* version of $Q(\lambda)$ from [WHH96] were compared. The result of the experiment was that $Q(\lambda)$ was an improvement over Q-learning in two of the three subtasks.

Matarić has applied reinforcement learning in several multi-robot experiments. In [Mat97] a group of four robots had to learn a collective foraging task. The focus of this work was the use of heterogeneous reward functions and internal progress estimators to speed up the learning process. The robots were to learn how to switch between some predefined primitive behaviours so the abstraction level in this experiment was a bit different from the experiments where the robots are to learn a direct mapping from sensor-input to motor-output.

The problem of dividing a continuous sensor-space into a discrete set of states, that enables learning of the task, is an important issue when applying reinforcement learning to robots. With a poorly divided state-space, the learning problem may become unsolvable. In [MK99] a method of adaptive state-space construction in Q-learning has been proposed. During the run of the learning algorithm, new states are added and old states are collapsed on the fly. This method was applied in two simulated robot experiments.

Neural Reinforcement Learning

Another way of dealing with the problem of state-space construction is to store the Q-function in a neural network rather than in a table. In this method the sensor values are given as inputs to the neural network in order to generalise the sensor-states. This approach has been studied by several authors. In [Tou97] several neural network architectures were compared with tabular Q-learning and enhancements to tabular Q-learning. It was found that the right neural implementation could speed up learning. A similar conclusion is reported in [Stø99]. In this work, the comparison of tabular to neural Q-learning was done in simulation only. The neural architecture was then modified to fit the real robot, and some further experiments were made. An interesting experiment was the one demonstrating the neural implementation's robustness to sensor malfunctioning. This is a step towards fulfilling *the complete agent principle*. In both works described above, the task was that of obstacle avoidance on a Khepera Robot.

Visually Guided Reinforcement Learning

Visually guided reinforcement learning has recently become more common, especially within the RoboCup community. There are many examples of Q-learning applications to visually guided football playing tasks. In [CDCD99] a Khepera robot equipped with a 2D camera was seen to learn a football playing behaviour. Task decomposition was applied and the different tasks of *wall avoiding*, *ball approaching* and *shooting* were learned separately.

Learning from Easy Missions

An alternative to task decomposition to speed up learning progress, has been proposed by Asada et. al. [ANTH96]. They call their method *Learning from Easy Missions* (LEM). The idea of LEM is to position the robot in states close to the goal state in order to ensure early learning progress. When the robot solves the task from the easy sensor-states, it is gradually positioned further away from the reward-giving goal state. This is learning by shaping, but not teaching. The robot is positioned in an easy state, but not told which action to take. This method was applied to both a simulated and a real visually guided robot, and has later on been used on a robot participating in the middle-size league of RoboCup [STU⁺98].

Relations to this Thesis

Here is a short outline of the relations between the above-mentioned work and the experiments carried out in this thesis. As mentioned, the first source of inspiration was the comparisons of model-based to model-free methods done in [Sut90][MA93][PW93] on Markovian grid-world problems. The comparisons were repeated in this thesis, but on a non-Markovian mobile robot problem. Similar comparisons on mobile robot problems, but between the model-free methods of Q-learning and $Q(\lambda)$ were made in [WHH98]. Then visually guided reinforcement learning was applied and whereas [ANTH96][STU⁺98][CDCD99] only used Q-learning, the experiments in this thesis again compared Q-learning to two model-based methods.

7.2 Discussion

In chapters 5 and 6, a range of experiments were made on optimising the learning abilities of a Khepera robot on a simple obstacle avoidance task. It is now time to take a step backwards and discuss how well the methods worked compared with the other design methods of chapter 3 and how well they will scale up to more complex task solving.

The experiments showed that it *is* possible to make a robot learn by the use of reinforcement learning. Several important issues restrict the usability however. First of all there is the problem that the Markov property is never fulfilled in situated mobile robot applications. Sensors give continuous values in real time *not* in discrete timesteps. In addition, when a robot perceives the world through local sensors, the problems of *perceptual aliasing* and *hidden state* arise. In this thesis the approach taken was to accept that fact that the real world is not a discrete mathematical Markov decision process (MDP). Under the right circumstances, the learning problem can be approximated to a MDP and in this case the methods will still work to some extent. It is appropriate to make this approximation when each state is a good basis for predicting future rewards and for selecting actions. This was the case in the experiments described in this text.

In chapter 3, it was argued that when the goal is to make the autonomous agents perform complex tasks, the design of the control systems has to be automated in some way. This was the main argument given for turning away from the purely behaviour-based robotics approach and start paying more attention to evolutionary and learning robotics. To which extent has this automation been accomplished by the use of reinforcement learning on robots? Problematic issues of the initial experimental setup were described in chapter 5. It was mentioned that state-spaces and action-sets have to be decided upon and a reward function has to be constructed which makes the robot solve the task. There are generalisation methods of dealing with state-space and action-set construction (see section 7.1), but still in every application some initial choices have to be made. The problems of dealing with initial setup in reinforcement learning are not very different from the ones encountered in evolutionary robotics. Here the genotype to be manipulated by the evolutionary algorithm has to be decided upon and various parameters have to be fine-tuned. The same similarity can be said to exist in the construction of the *fitness function* in evolutionary robotics and the *reward function* in reinforcement learning respectively. In the future, work has to be done on making the methods more adaptive and thus less dependent on initial biases. When comparing with the pure hand-coding used in behaviour-based robotics it *is* a relief that the designers only have to specify the setup and let the evolution or learning mechanisms come up with usable solutions to the control problem.

In addition to automating the design process, an important issue is how well the methods scale to more complex problems solving. If the automatic methods cannot come up with more complex and robust control strategies than could have been hand-coded by humans, the arguments for not using the behaviour-based methods are no longer valid. It was mentioned in chapter 3 that promising steps in this direction have been taken by applying evolutionary methods, but most experiments have been carried out on miniature robots on laboratory desks. In the future what needs to be done is to put the robots out in unaltered and unstruc-

tured environments, and make them work under those circumstances as well. In reinforcement learning robotics, there is still a lot of work to be done to make the methods scale up to complex tasks. On the basis of the work done in this thesis not much can be said about how well reinforcement learning will scale up because only the relatively simple task of obstacle avoidance was investigated.

Moving on to discuss specific issues of reinforcement learning at first the setup of a reinforcement learning system on a robot navigation task was investigated. Several experiments on the setting of parameters to the learning algorithm was carried out, but the main focus was the comparison of model-free to model-based methods. The starting assumption was that the speedup of learning, that model-based methods have been seen to result in when applying them to Markovian problems, can to some extent be transferred to the situated robot-learning domain. The method Prioritized Sweeping was seen behave above the level of Q-learning both with respect to learning speed and value of the policy learned. Especially in Experiment 8 on the visually guided task, the difference in favour of Prioritized Sweeping was pronounced, and when Q-learning behaved at its best did it only reach about the same level of performance as Prioritized Sweeping. The behaviour of the Dyna algorithm was more disappointing though. The version of Dyna described in section 4.4 was the one where the direct *backup* of the Q-function was replaced by a backup that refers to the model being built. Recall that this was done in order to avoid the learning-rate of Q-learning. In Experiment 1, it was found that the setting of the learning rate was not that critical to the performance of Q-learning algorithm on the setup used. Perhaps an implementation of Dyna, where the original Q-learning update-rule is used for the direct backup, could improve the performance of the Dyna version implemented in this text. On the other hand, there is not much hope that Dyna will be able to improve performance over Prioritized Sweeping.

In experiment 7, an on-line reinforcement learning method was used and the robot successfully adapted to changing environments. The environment changed in the eyes of the observer, but it was not, however, investigated how much the environment changed from the robot's point of view. An important issue, that was not investigated is that if the environment keeps changing parts of the stored world-model will become obsolete as time passes. If the robot is to perform its task over extended periods of time and continuously adapt to changing environments, some sort of knowledge decay has to be applied to the model stored. This is not a problem in the model-free methods like Q-learning.

Finally in Experiment 8 a camera was added to the robot and it was seen to perform the learning task with visual-state inputs. Also this learning problem was solved by the robot, but the problem of state-space construction seems to be harder when visual sensors are added to the robots. This issue was not in-

vestigated in detail. The problem of visual state-space construction is that of discriminating relevant image features from irrelevant ones. An enormous reduction of the continuous data-flow has to be done. In general it is a hard problem to figure out which image features are relevant to the solving of a specific task, so some kind of adaptive extraction of features seem to be necessary. However when adding vision to the robots, reinforcement learning methods seem more promising than evolutionary methods. This was argued in chapter 3, and an indication can be seen at the middle-size league at RoboCup held every year. Here autonomous robots have to perform the complex task of multi-agent football playing partly by visual inputs. In this field applications of reinforcement learning is by far dominant over evolutionary robotics. The main problem with evolution and vision is that it is harder and computationally more heavy to simulate visual sensors than to simulate short-range sensors like proximity sensors and sonars, and the speedup by using a simulator for fast evolution is therefore lost. If this simulation of visual inputs has to be avoided, the evolutionary process has to be carried out on real hardware which is known to be very time-consuming as well.

7.3 Future Work

What are the possible future extensions to the work done in this text? First of all, it seems very natural to incorporate the $Q(\lambda)$ algorithm in the comparative investigations done. $Q(\lambda)$ has been reported to outperform Q-learning on real robot tasks [WHH98] and Prioritized Sweeping in simulated non-Markovian domains [PW96]. The *corrected* version of $Q(\lambda)$ where the exploration sensitivity has been removed is a potential candidate for behaving equally or above the level of Prioritized Sweeping in the experiments in this thesis.

The next step is to investigate how well the methods scale when applying them to more complex tasks. In Experiment 8, Prioritized Sweeping was seen to perform very well on the visually guided learning task. Will the individual strengths of the methods be displaced with the increase in the task complexity? More investigations of adaptive state-space construction from visual input-sensors will be needed as well to enhance the chances of scaling to succeed.

Finally if the model-based methods still performs well it could be interesting to investigate possible combinations of model-based and neural reinforcement learning methods. One neural network could learn the world model and generate pseudo-experience for another neural network learning the mapping from states to actions.

7.4 Conclusion

This Master's Thesis has shown that the task of obstacle avoidance on a real robot can be learned within a couple of minutes by the use of reinforcement learning. The setup was investigated in detail. In Experiment 1, it was seen that the setting of the learning rate of Q-learning did not have great impact on the learning progress. In Experiment 2 on the other hand, it was found that the setting of the timestep length was crucial. When the timesteps were too short, the robot had problems converging to a task-solving behaviour. In Experiment 3, the effect of the discount factor was investigated. It was found that the setting of the discount factor, when using table-based methods, is not as crucial as has been reported on a neural implementation in a similar experiment. In Experiment 4, different action selection methods were compared. The result was that both ϵ -greedy action selection and Softmax action selection worked well. Softmax was seen to enable learning of very smooth behaviours when the Boltzmann-temperature was slowly decreased during the learning-phase. In experiment 5, the first comparison of Q-learning, Dyna and Prioritized Sweeping was carried out. Both Dyna and Prioritized Sweeping outperformed Q-learning with respect to learning progress, but only the policies learned by Prioritized Sweeping was an improvement over the ones learned by Q-learning. When the action-set size was doubled in Experiment 6, a slowdown in learning progress might have been expected. This was not the case, however. On the contrary the extended action-set did allow a faster learning progress. In the Q-learning methods the difference was pronounced. In addition, some new behavioural strategies were seen when the new action-set was applied, but this result was to be expected. In Experiment 7, an on-line learning method with Softmax action-selection was proposed. The methods gradually changed the Boltzmann temperature on the basis of an internal performance evaluator. Finally in Experiment 8, the experimental setup was transferred to a visually guided obstacle avoidance task. Here the Prioritized Sweeping method was seen to behave significantly better than both Q-learning and Dyna. Another conclusion of this experiment was the different performance measures can result in different conclusions.

To summarise, a real robot with and without a camera was seen to perform the task of obstacle avoidance by the use of reinforcement learning. In both cases Prioritized Sweeping performed better than Q-learning. In the future, it remains to be investigated how well the methods used will scale to more complex applications.

Appendix A

Notation

S	a finite discrete set of environment states
A	a finite discrete set of agent actions
\mathfrak{R}	a set of scalar reinforcement signals
$R : S \times A \rightarrow \mathfrak{R}$	a reward function
$R(s, a)$	the reward of performing action a in state s
$\hat{R}(s, a)$	an estimate of $R(s, a)$
$T : S \times A \rightarrow \Pi(S)$	a state transition function
$T(s, a, s')$	the probability of making the transition from s to s' by action a
$\hat{T}(s, a, s')$	an estimate of $T(s, a, s')$
$V : S \rightarrow \mathfrak{R}$	a value function (mapping states to expected future rewards)
$\pi : S \rightarrow A$	a policy
$\pi^* : S \rightarrow A$	the optimal policy
$V^\pi(s)$	the value of state s under policy π
$V^*(s)$	the value of state s under the optimal policy
$Q^* : S \times A \rightarrow \mathfrak{R}$	the optimal action-value function
$Q^*(s, a)$	the value of taking action a in state s and then performing π^*
$\hat{Q}(s, a)$	an estimate of $Q^*(s, a)$
$\langle s, a, r, s' \rangle$	an experience tuple. The experience of taking action a in state s receiving reward r and transitioning to state s'
γ	the discount factor
α	the learning rate used in Q-learning
SA	the set of experienced state-action pairs $(s, a) = \langle s, a, *, * \rangle$
$Pred(s)$	the predecessor function, mapping a state s to the state-action pairs (\bar{s}, \bar{a}) with a non-zero transition probability to it: $T(\bar{s}, \bar{a}, s) \neq 0$
$PQueue$	the priority queue containing state-action pairs used in Prioritized Sweeping
δ	the small threshold a priority has to exceed for the element in question to be inserted into $PQueue$

Appendix B

On-line Documentation

Video-documentation of some of the experiments described in this text can be found on the internet-address:

`http://www.daimi.au.dk/~blynel/thesis/index.html`

The videos will be available from June 8 2000.

Bibliography

- [ANTH96] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23:279–303, 1996.
- [Bra84] Valentino Braitenberg. *Vehicles. Experiments In Synthetic Psychology*. MIT Press, 1984.
- [Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986.
- [Bro90] Rodney A. Brooks. Elephants don't play chess. In Patti Maes, editor, *Designing Autonomous Agents*, pages 3–18. MIT Press, 1990.
- [Bro91a] Rodney A. Brooks. Artificial life and real robots. In *Towards a Practice of Autonomous Systems: European Conference on Artificial Life, Paris, France*, pages 3–10. MIT Press, December 1991.
- [Bro91b] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, January 1991.
- [CDCD99] G. Cicirelli, T. D'Orazio, L. Capozzo, and A. Distante. Learning elementary behaviors with khepera robot. In *Proceedings of the 1st International Khepera Workshop*, pages 109–118, 1999.
- [DMS99] Y. Deng, B. S. Manjunath, and H. Shin. Color image segmentation. In *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 1999.
- [FM96a] D. Floreano and F. Mondada. Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics–Part B: Cybernetics*, 26(3):396–407, 1996.
- [FM96b] D. Floreano and F. Mondada. Evolution of plastic neurocontrollers for situated agents. In *From Animals to Animats IV*. MIT Press, 1996.

- [GG98] T. Gomi and A. Griffith. Developing intelligent wheelchairs for the handicapped. *Lecture Notes in Computer Science*, 1458, 1998.
- [Gom98] Takashi Gomi. Non-cartesian robotics - the first 10 years. In Takashi Gomi, editor, *Evolutionary Robotics, Volume II (ER'98)*, pages 245–306. AAI Books, 1998.
- [Hal] John Hallam. Intelligent sensing and control: Notes on control theory. Lecture Notes, Department of AI, University of Edinburgh.
- [HHC92] I. Harvey, P. Husbands, and D. Cliff. Issues in evolutionary robotics. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1992.
- [HHC94] Inman Harvey, Phil Husbands, and Dave Cliff. Seeing the light: artificial evolution, real vision. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1994.
- [HHC+97] I. Harvey, P. Husbands, D. Cliff, A. Thompson, and N. Jakobi. Evolutionary robotics: the sussex approach. In *Robotics and Autonomous Systems, v. 20*, pages 205–224, 1997.
- [HKP91] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computing*. Addison-Wesley Publishing Company, 1991.
- [HMR91] David Hogg, Fred Martin, and Mitchel Resnick. Braitenberg creatures. Epistemology and learning memo #13, MIT Media Laboratory, 1991.
- [Hor93] Ian Horswill. Polly: A vision-based artificial agent. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 824–829. AAAI Press, July 1993.
- [Jak98] Nick Jakobi. The minimal simulation approach to evolutionary robotics. In Takashi Gomi, editor, *Evolutionary Robotics, Volume II (ER'98)*, pages 133–190. AAI Books, 1998.
- [JHH95] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. *Lecture Notes in Computer Science*, 929, 1995.
- [K-T95] K-Team. *Khepera User Manual version 4.06*, 1995.
- [KAK+98] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. Robocup: A challenge problem for ai and robotics. *Lecture Notes in Computer Science*, 1395, 1998.

- [KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Artificial Intelligence Research*, 4, 1996.
- [KR92] J. R. Koza and J. P. Rice. Automatic programming of robots using genetic programming. In *In Proceedings of AAAI-92*, pages 194–201, 1992.
- [LBG97] Liana M. Lorigo, Rodney A. Brooks, and W.E.L. Grimson. Visually-guided obstacle avoidance in unstructured environments. In *Proceedings of IROS '97, Grenoble, France*, pages 373–379, September 1997.
- [LH96] Henrik Hautop Lund and John Hallam. Sufficient neurocontrollers can be surprisingly simple. Research paper no. 824, Department of Artificial Intelligence, University of Edinburgh, 1996.
- [MA93] Andrew W. Moore and Christopher G. Atkinson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, 1993.
- [Mah94] Sridhar Mahadevan. To discount or not to discount in reinforcement learning: A case study comparing R-learning and Q-learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 164–172, July 1994.
- [Mar94] Fred Martin. Ideal and real systems: A study of notions of control in undergraduates who design robots. In Y. Kafai and M. Resnick, editors, *Constructionism in Practice: Rethinking the Roles of Technology in Learning*, 1994.
- [Mat94] Maja J Matarić. Reward functions for accelerated learning. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 181–189, 1994.
- [Mat97] Maja J. Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, Mar 1997.
- [MC96] Maja J. Matarić and Dave Cliff. Challenges in evolving controllers for physical robots. *Journal of Robotics and Autonomous Systems*, 19(1):67–83, October 1996.
- [MDTP98] O. Miglino, D. Denaro, G. Tascini, and D. Parisi. Detour behavior in evolving robots: Are internal representations necessary? In *Proceedings of First European Workshop on Evolutionary Robotics*. Springer-Verlag, 1998.

- [Mic96] Olivier Michel. *Khepera Simulator version 2.0 - User Manual*. University of Nice - Sophia Antipolis, 1996.
- [MK99] Hajime Murao and Shinzo Kitamura. Q-learning with adaptive state space construction. *Lecture Notes in Computer Science*, 1545, 1999.
- [MLN95] Orazio Miglino, Henrik Hautrup Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995.
- [MSH89] C. A. Malcolm, Tim Smithers, and J. Hallam. An emerging paradigm in robot architecture. DAI research paper 447, Dept AI, Edinburgh U, November 1989.
- [NF98] Stefano Nolfi and Dario Floreano. Co-evolving predator and prey robots: Do 'arm races' arise in artificial evolution? *Artificial Life*, 4(4), 1998.
- [Nol96] Stefano Nolfi. Adaptation as a more powerful tool than decomposition and integration. In T.Fogarty and G.Venturini, editors, *Proceedings of the Workshop on Evolutionary Computing and Machine Learning, 13th International Conference on Machine Learning*, 1996.
- [Nol97] Stefano Nolfi. Evolving non-trivial behaviors on real robots: a garbage collecting robot. *Robotics and Autonomous System*, 22:187–198, 1997.
- [Par97] J. R. Parker. *Algorithms for Image Processing and Computer Vision*. Wiley Computer Publishing, 1997.
- [Pfe96] Rolf Pfeifer. Building fungus eaters: Design principles of autonomous agents. In *From Animals To Animats 4: Fourth International Conference on Simulation of Adaptive Behavior*, pages 3–12. MIT Press, 1996.
- [PS98] Rolf Pfeifer and Christian Scheier. Embodied cognitive science: A novel approach to the study of intelligence in natural and artificial systems. In Takashi Gomi, editor, *Evolutionary Robotics, Volume II (ER'98)*, pages 1–35. AAI Books, 1998.
- [PW93] Jing Peng and Ronald J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.
- [PW96] Jing Peng and Ronald J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22:283–290, 1996.

- [RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, International Editions, 1995.
- [SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. The MIT Press, Cambridge, MA, 1998.
- [Smi90] Tim Smithers. Control and control strategies. Lecture notes on Intelligent Sensing and Control, 1990.
- [Smi98] Tom Smith. Blurred vision: Simulation-reality transfer of a visually guided robot. *Lecture Notes in Computer Science*, 1468, 1998.
- [Stø99] Kasper Støy. Adaptive control systems for autonomous robots. Master's thesis, University of Aarhus, 1999.
- [STU⁺98] S. Suzuki, Y. Takahashi, E. Uchibe, M. Nakamura, C. Mishima, H. Ishizuka, T. Kato, and M. Asada. Vision-based robot learning towards robocup. *Lecture Notes in Computer Science*, 1395, 1998.
- [Sut90] Richard S. Sutton. Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 1990.
- [Sut91] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 4:160–163, 1991.
- [Swa94] Michael I. Swartzbach. Datastrukturer. Lecture note daimi fn - 38, Department of Computer Science, University of Aarhus, Feb. 1994.
- [Tou97] Claude F. Touzet. Neural reinforcement learning for behaviour synthesis. *Robotics and Autonomous Systems*, 22:251–282, 1997.
- [Wat89] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.
- [WD92] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.
- [WHH96] Jeremy Wyatt, Gillian Hayes, and John Hallam. Investigating the behaviour of $Q(\lambda)$. *Colloquium on Self-Learning Robots, IEE, London*, Feb 1996.
- [WHH98] Jeremy Wyatt, John Hoar, and Gillian Hayes. Design, analysis and comparison of robot learners. *Robotics and Autonomous Systems*, 24(1-2):17–32, 1998.

- [Wya95] Jeremy Wyatt. Issues in putting reinforcement learning onto robots. *Presented at: Mobile Robotics Workshop, 10th Biennial Conference of the AISB, 1995.*