Learning Behaviours for Robot Soccer

A thesis submitted for the degree of

Doctor of Philosophy

James P. Brusey, B.App.Sc.,

School of Computer Science and Information Technology,

Faculty of Applied Science,

RMIT University,

Melbourne, Victoria, Australia.

7th October 2002

Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged. The work has been carried out since the beginning of my candidature on July 1, 1997.

James Brusey
School of Computer Science and Information Technology
RMIT University
7th October 2002

Acknowledgments

I would like to begin by thanking my two supervisors, Associate Professor Lin Padgham and Associate Professor Vic Ciesielski, for the enormous amount of support and guidance that they have given me. I would particularly like to thank Lin for encouraging me to attempt a research degree in the first place, and having faith in me throughout the course of my candidature. I am grateful to Vic for helping me to focus on the research questions and for providing insightful feedback.

During my research, I was extremely fortunate to meet and work with Mark Makies and Chris Keen, who built a team of RoboCup robots, one of which is used in this thesis. Thanks also to Mark Makies for providing the picture used for figure 5.1 on page 71. The robot combined with the vision system that was created by, among others, K Do Duy and Si Dinh Quang, provided a robust platform for the experimental work herein. Although the hand-coded behaviours are my own work, Bradley Woodvine fine tuned the parameters for the *rmit-turnball* behaviour. I would also like to thank the School of Electrical and Computer Engineering at RMIT University for lending me the robot for the purpose of performing the experimental work.

Developing software for the RMIT-United autonomous robot soccer team and interacting with other teams at the 1998, 1999 and 2000 RoboCup competitions was a great source of inspiration for me and I would like to thank the RMIT-United team members and also the RoboCup community at large.

My research was greatly supported by the School of Computer Science and Information Technology at RMIT University, who, among other things, provided space for the robot experiments, and an extended loan of a laptop computer for use with the robot.

I would like to thank the many people who have assisted me by providing valuable feedback on my writing at various stages, including Li Mei Brusey, Dr. Pam Green, Andy Song, Pablo Rossi, Tom Loveard, Andrew Innes, and Fred Brkich.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

A central problem in autonomous robotics is how to design programs that determine what the robot should do next. Behaviour-based control is a popular paradigm, but current approaches to behaviour design typically involve hand-coded behaviours. The aim of this work is to explore the use of reinforcement learning to develop autonomous robot behaviours automatically, and specifically to look at the performance of the resulting behaviours.

This thesis examines the question of whether behaviours for a real behaviour-based, autonomous robot can be learnt under simulation using the Monte Carlo Exploring Starts, ϵ-soft On Policy Monte Carlo or linear, gradient-descent Sarsa(λ) algorithms. A further question is whether the increased performance of learnt behaviours carries through to increased performance on the real robot. In addition, this work looks at whether continuing to learn on the real robot causes further improvement in the performance of the behaviour.

A novel method is developed, termed Policy Initialisation, that makes use of the domain knowledge in an existing, hand-coded behaviour by converting the behaviour into either a reinforcement learning policy or an action-value function. This is then used to bootstrap the learning process.

The Markov Decision Process model is central to reinforcement learning algorithms. This work examines whether it is possible to use an internal world model in the real robot to suit the requirements of the Markov Decision Process model.

The methodology used to answer these questions is to take three realistic, non-trivial robotic tasks, and attempt to learn behaviours for each. The learnt behaviours are then compared with hand-coded behaviours that have either been published or used in international competition. The tasks are based on real task requirements for robots used in a RoboCup Formula 2000 robot soccer team. The first is a generic movement behaviour that moves the robot to a target point. The second requires the robot to dribble the ball in an arc so that the robot maintains possession and so that the final position is lined up with the goal. The third addresses the problem of kicking the ball away from the wall.

The results show that for these three different types of behavioural problem, reinforcement learning on a simulator produced significantly better performance than hand-coded equivalents, not only under simulation but also on the real robot. In contrast to this, continuing the learning process on the real robot did not significantly improve performance.

The Policy Initialisation technique is found to accelerate learning for tabular Monte Carlo methods, but makes minimal improvement and is, in fact, costly to use in conjunction with linear, gradient-descent Sarsa(λ). This approach, unlike some other techniques for accelerating learning, does not appear to bias the solution.

Finally, the evidence from this thesis is that internal world models that maintain the requirements of Markov Decision Processes can be constructed, and this appears to be a sound approach to avoiding problems connected with partial observability that have previously occurred in the use of reinforcement learning in robotic environments.

Chapter 1

Introduction

Autonomous robots have long held an elusive promise of revolutionising modern life. Despite this promise, it has turned out to be difficult to write software to make robots perform even basic tasks autonomously. An idea that has been key in simplifying the problem of programming robots is to divide a robotic task into a set of behaviours, and program each one separately. Even so, programming behaviours can still be problematic.

Another approach to devising programs for robots is to teach them, by rewarding their successes and penalising failures. This approach is called reinforcement learning, and it has gained popularity recently, largely due to work that established a foundation for it in the theory of Markov Decision Processes [118, 127]. Reinforcement learning however does not easily scale to large problems. This thesis combines these two approaches to learn individual behaviours through reinforcement learning. It develops an approach that might be used to overcome scaling problems in reinforcement learning, and also produces better performing behaviours than those programmed by hand.

Experience with developing a robot soccer team for the RoboCup Formula 2000 competition provided the inspiration for this thesis. Although there are many aspects to developing a robot soccer team, hand tuning of individual behaviours yielded significant performance improvements. In addition, it was not clear how far from optimum the tuned behaviours were. As this thesis will show, significant further improvement was available through the techniques described here. This thesis focuses on robot soccer, since it is both a rich source of complex, real world problems, as well as being an engaging and readily understood domain. However, the approach suggested by this thesis is generally applicable to real world robotic problems of many types.

It is hoped that this work will be useful to robot developers that are currently coding behaviours by hand, and in particular, to those developing behaviours for complex problems, where the optimal solution cannot easily be derived through analysis.

RoboCup is an international soccer competition played between teams of autonomous robots. The Formula 2000 league have robots that are about 40 by 50 centimetres in size when seen from above and up to 80 centimetres high. Two teams of 4 robots each play on a field that is 9 by 5 metres. The goals, clearly identified by their yellow or blue colour, are situated at each end and are 2 metres wide. All of the robots sensory information must come from devices on the robot, such as cameras, sonar, and laser-range finders.

The RMIT-United team used a behaviour-based approach to control their robots. One of the behaviours designed for the 1999 competition involved moving behind the ball to shoot at the goal. The behaviour was split into a sequence of intermediary points that ensured that the robot went around the ball rather than straight through it. During competition, the robots took much too long to move to each point, despite having powerful motors that gave the robots one of the fastest top speeds in the competition. The problem was that the robot needed to stop before turning, making the movement very slow. In comparison, many other teams' robots at the competition tended to move in smooth curves, so that momentum was not lost, and were generally faster to get to the ball. In a sense, this is a small problem; one that might have been resolved by a better understanding the physics of the robots or a better knowledge of past robotic control systems. For example, Crowley and Reigner [35] had previously looked at the problem of designing a robotic control system to allow turning and moving forward to occur simultaneously. However, this small problem pointed out a larger problem with designing robot control systems in general: basic behaviours that achieve a task are relatively easy to create, but these behaviours are rarely optimal and are often grossly inefficient.

Debugging problems with robotic behaviours can be difficult. First, problems with one behaviour may be difficult to isolate from problems with other behaviours, particularly if the robot is rarely in a situation to activate the behaviour. Second, problems with a behaviour may be difficult to isolate from problems with the hardware, such as the sensors or actuators. For example, a sensor's or actuator's response might vary. Often robot hardware is unstable while the behaviours are being developed and this can cause an especially difficult problem for the behaviour designer. Third, even when problems with other hardware and other behaviours do not interfere, it is often difficult to test a full range of possible situations and to clearly identify intended and unintended responses.

The use of simulation can provide some relief for these problems. In simulation, it is possible to just simulate the robot's actions in situations where the behaviour is active. In simulation, hardware issues, such as faulty sensors, or motor variability, can either be simulated, or ignored. In simulation, it is possible to test the full range of situations where the behaviour is being used. Simulation, by itself, does not resolve the issue of identifying correct responses, but it does make this task somewhat easier by removing other factors.

Two factors have inspired this work. The first is the desire to improve the performance of individual behaviours. Experience suggests that the performance of individual behaviours is critical to the success of a robot soccer team. It seems likely that high performance will also be desirable in other fields. In the competitive environment of robot soccer, though, it is not enough that a new version of a behaviour outperforms the old version; it must outperform the competition as well. To do this, the behaviour must approach the best possible performance.

The second factor is that the complexity of robot behaviours is becoming greater. For example, when robots moved slowly, the momentum of the robot did not need to be considered to navigate successfully. At higher speeds, though, taking account of momentum is important, if good performance is desired. However, designing a behaviour that adjusts its actions depending on the momentum is complex. In addition, the navigaion behaviour is just one among many that need to be adjusted to take account of momentum. Furthermore, there may be other dimensions to the problem, such as the robot's angular momentum, that become critical. A way to deal with this extra complexity is to avoid directly hand-coding behaviours, and instead use some automatic approach to developing them.

1.1 Markov Decision Processes

A Markov Decision Process is a mathematical model of a process that involves a series of sequential decisions about what action to take. This model is discussed in more detail in section 2.4.1 on page 19, and it has a strong resemblance to the problem faced by an autonomous robot attempting to decide what to do. Interestingly, it can be shown that for any Markov Decision Process, there exists an optimal policy, or in other words, a mapping of situation to action that maximises the long term return. This result suggests that if the problem facing an autonomous robot can be converted to the Markov Decision Process model, then it may be possible to find the optimal behaviour for the robot. The basic approach used here is to create a simulator, to define the behavioural task as a Markov Decision Process, and to apply algorithms from reinforcement learning to attempt to find the optimum, or near optimum, controller.

If the optimisation process is only examined in terms of simulation, there is the danger that the resulting behaviour will come to rely on features of the simulation, and will not transfer onto a real world robot. Therefore, this thesis focuses on the performance of these behaviours on real robots.

1.2 Three Robotic Tasks

This thesis looks specifically at three robotic tasks, all of which are connected to the larger problem of robot soccer, but may have more general applicability. The first is to move the robot to a position relative to the current position. In robot soccer, it is important to move quickly, or rather, to get to the target point in the least amount of time. This is referred to in this thesis as the efficient movement task. Interestingly, taking the shortest path does not correspond to taking the least amount of time. Although two wheel robots can spin on the spot, they cannot move sideways. This makes the problem more complicated than for omni-directional robots that can move in any direction with roughly equal cost. In addition, the robot does not necessarily start from rest but may already be moving. Existing momentum has an effect on how quickly the robot can move in any particular direction.

The second task is to turn while maintaining possession of the ball. This is a typical soccer task, although not one attempted by all robot soccer teams. It is particularly difficult in robot soccer, as the robots are not allowed to hold the ball. Furthermore, the shape of the robot must not contain a concavity that allows for more than a third of the ball to fit inside. This task is attempted with a two wheel robot that has a fixed shape.

The final task involves kicking the ball when it is near the wall, so that it comes away. Prior to RoboCup 2002, the playing field for the middle-size robots was surrounded by walls that prevented the ball from being played “out”. The walls presented a particularly treacherous obstacle to robots, not merely due to the potential for collision, but more because the robot might become jammed, and not be able to free itself. Finding a way of playing the ball when it is near the wall without becoming stuck is a difficult problem. It is one that has similar characteristics to many other robotic tasks where the potential cost of approaching something dangerous must be weighed against the benefits of achieving goals.

1.3 Research Questions

In general terms, this thesis explores the advantages and disadvantages of creating behaviours for behaviour-based robots through reinforcement learning under simulation.

The research questions examined by this thesis can be summarised as follows:

1. Can reinforcement learning algorithms be used to develop better performing physical robot behaviours than those developed by hand?

2. If improved behaviours are learnt on a simulator, will that improvement transfer to the physical robots?
3. Can training be refined on physical robots after training on a simulator?
4. Is it possible to represent the world internally to suit the requirements of a Markov Decision Process?
5. Is it possible to bootstrap the learning process by using an existing, hand-coded behaviour as a starting point? In addition, does this bias the solution found?

These questions are now elaborated on in more detail.

1.3.1 Measuring Behaviour Performance

With regard to the first research question, whether one behaviour is better performing than another, may be open to debate. To reduce ambiguity when comparing two behaviours, it is useful to quantify the behaviour's performance in some way. How the performance is quantified is a central design decision. For example, a behaviour that moves the robot from one point to another might have its performance measured in terms of the amount of time taken, or in the expenditure of energy, or it may be some combination. When looking for appropriate performance measures there are usually several possible choices. However, before two behaviour's performances can be compared, the performance measure must first be defined.

In terms of the three behavioural tasks introduced previously, the performance will be quantified as follows. For the efficient movement problem, the basic measure of performance is the amount of time taken. To avoid having to learn perfectly accurate movement, the target point is defined as a target region, and so, although the time taken is measured in terms of time to reach the target region, a small additional penalty is used to encourage behaviours that get closer to the centre of the target region.

For the "turn ball" task, the basic measure of performance is how far the ball is turned towards the target. Also, how quickly the movement is performed is also a factor. A final factor, one that avoids too much of the soccer pitch from being used, is to include a small penalty for the amount of distance travelled.

The most critical factor in "off wall" task is avoiding collisions with the wall, and in particular, avoiding stall conditions. Other factors are whether the ball was kicked in the correct direction, whether it was kicked away from the wall, and how much time was taken.

One difficulty with comparing an approach with what can be done by hand, is that it begs the question "who's hand?" Perhaps the hand-crafted behaviours used as examples are mediocre and a more gifted programmer could do much better. Also, was the programmer deliberately holding back during development, so as to make a preferred solution look better? The answer applied here is to only make use of behaviours that have been developed for an international robot soccer competition, by an expert behaviour designer with several years experience in developing robot soccer behaviours. Where possible, an additional requirement is to make use of behaviours that have been published as part of a conference paper presented at an international conference [23].

1.3.2 Transferring Improvement to a Physical Robot

The second question examines the issue of whether a behaviour learnt using reinforcement learning can successfully "transfer" its performance from the simulator to the real world. Since simulated environments tend not to be exact models of the environment, it seems inevitable that there will be some difference between the performance of a behaviour on the simulator compared with that on a physical robot. Even so, under most circumstances, one might expect that if there are two behaviours that are intended to perform some task, one of which is better performing on the simulator, it seems reasonable to expect that the better behaviour will continue to be better performing when both are transferred to the robot.

This question underlies the use of simulation to develop or learn new behaviours, and has been previously examined in terms of robot controllers learnt using other machine learning techniques [64, 96]. If behaviours that show significant improvement under simulation are not also improved significantly when tested on a physical robot, this would negate any advantage of using simulation to develop or learn behaviours.

This question will be examined empirically by looking at instances of simulator improved behaviours and examining how they perform on a physical robot.

1.3.3 Refining Training on a Physical Robot

The third question pertains to whether or not it is useful to continue learning on the physical robot, based on the results of the simulator learning. The training method proposed in this thesis is based on performing most of the learning on a simulator. The main reason for this is the large number of trials required to converge on a solution. Although the training method suggests following the learnt behaviour without changing it once it has been transferred to the robot, a possible adjustment to this approach is to continue the learning process on the physical robot, in order to further refine the behaviour. One reason for this is that the simulator will not exactly mimic actual physical effects and continued learning with a reasonable behaviour as a starting point may be one way to achieve good behaviours.

In order to test this idea, continued learning is attempted with the “turn ball” task. This task is an example where continued learning might be expected to have a positive effect, since the simulator was not able to simulate the more subtle effects of the interaction between the ball and the robot’s fingers. Continued learning will be attempted by transferring the action-values learnt under simulation and use them on the real robot as a starting point for further learning.

1.3.4 World Representation

The fourth question asks whether it is possible to have an internal representation of the world to suit the requirements of a Markov Decision Process. The basic requirement of Markov Decision Processes is that the world state representation be Markovian, or in other words, all of the necessary information is in the current state and no history of previous states is needed to decide the next action. For example, say the robot sees the position of the ball, and records just the position in its internal representation of the world. Over time, it sees a series of ball positions. Given the current state, or in other words, given knowledge of just the ball position, the robot can only make a poor estimate of where the ball will be in the next state. Including past positions in the state representation would allow it to improve its prediction, and this shows that the state representation is not Markovian.

In general, the aim is to express the combination of past and current ball positions in a compact way, such as by its current position and velocity. Compact representations are important because algorithms that solve or find approximate solutions for Markov Decision Processes increase in complexity with the number of different states.

The problem of creating a compact Markovian state representation is examined for the three behavioural tasks mentioned previously.

1.3.5 Bootstrapping Learning

Reinforcement learning can be a slow process. The early stages of the learning process tend to investigate many illogical actions due to poor estimates of the expected return associated with different actions. Sometimes a reasonable mapping of situation to action, in the form of a hand-coded behaviour, is already available. It seems unnecessary to investigate poorer behaviours than the hand-coded one. In addition, starting from the hand-coded one may provide a guarantee of getting a better behaviour than the hand-coded one for algorithms that are proven to always improve on a non-optimal behaviour, such as policy iteration.

The advantages and disadvantages to bootstrapping reinforcement learning algorithms are explored in terms of both discrete and continuous state space algorithms. This exploration is performed in the context of the efficient movement task.

1.4 Contribution

This thesis makes several contributions to the field of behaviour-based robotics. First is a *methodology for developing real robot behaviours using reinforcement learning with a simulator.* This methodology is shown, for all three tasks explored, to produce behaviours that significantly exceed the performance of hand-coded behaviours that were either published or developed for international competition. This is important since previous work in this area has generally produced behaviours that were similar in performance but not significantly better than their hand-coded equivalents.

Second is the *development of three high performance behaviours.* In particular, this development includes the design of the reinforcement learning parameters that would allow equivalent behaviours to be produced for other robots given a corresponding simulator. This design information is quite general and could be applied to a broad range of robot types.

Third is the *introduction and evaluation of a novel approach, termed Policy Initialisation, that makes use of existing behaviours to bootstrap reinforcement learning.* It is evaluated for the Monte Carlo Exploring Starts, Monte Carlo ϵ-soft On Policy, and linear, gradient-descent Sarsa(λ) reinforcement learning algorithms. This thesis demonstrates that there are significant advantages to using Policy Initialisation with tabular Monte Carlo methods, while there is no advantage and in fact, learning is slowed by using Policy Initialisation with linear Sarsa(λ).

Last is the development of an *approach to designing the internal world model of a real robot so that it is suitable for behaviours developed using reinforcement learning.* The Markov Decision Process model that forms a basis for reinforcement learning algorithms, contains several assumptions about the state signal. This work demonstrates an approach to designing the world model to suit these assumptions, and this appears to form a sound basis for learning behaviours using reinforcement learning.

1.5 Thesis Structure

This thesis is structured around three robotic tasks that have been carefully chosen to present a variety of difficult challenges. Prior to explaining those tasks, it is first necessary to provide background in the form of a literature survey, and this is found in chapter 2. This chapter surveys recent work in the areas of behaviour-based robotics and reinforcement learning, and in particular focuses on previous work that has combined the two approaches. Following this, chapter 3 describes the robot architecture used for the case studies, and in particular, shows the modifications made to traditional approaches that make this architecture well suited to developing behaviours through learning. The subsequent chapter, chapter 4, focuses on the theoretical contribution of this thesis by explaining the methodology used to specify and learn behaviours using the architecture mentioned previously. In addition, chapter 4 describes the detail of the new algorithms used to bootstrap learning based on an existing behaviour. This leads into chapter 5, which introduces the experimental tasks and provides an overview for how they will be presented. The tasks themselves are presented in chapters 6, 7, and 8. Chapter 6, the efficient movement task, looks at the problem of efficient movement for a two wheel robot. Chapter 7, the "turn ball" task, looks at using the same robot to perform a turn while maintaining possession of the ball. Chapter 8, the "off wall" task, presents solutions to the problem of manipulating the ball when it is close to a wall. Finally, conclusions and potential areas for further work are presented in chapter 9.

Chapter 2

Literature Survey

This chapter reviews literature relevant to this thesis, and outlines the theoretical background that forms the basis for the experimental work described in later chapters. It begins with an introduction to the game of robot soccer. Robot soccer provides the motivation for some of the experimental work here, however the approach can be considered more general than this. Central to the game of robot soccer is the requirement that the robots be autonomous. Autonomous robots are often controlled using some form of behaviour-based technique. The second section introduces this general approach and discusses some of the variants tried. Following on from this, the third section looks at how behaviours and possibly other techniques can be used to move the robot from one spot to another.

In addition to hand-coded approaches to developing movement and other types of behaviour, a number of researchers have used various types of machine learning. The fourth section introduces reinforcement learning, while the fifth reviews existing work on applying reinforcement learning to robotic problems. Reinforcement learning is not the only option however, and the sixth section looks at other approaches that have been tried. Finally, work involving the use of simulation to learn behaviour for real robots is examined.

2.1 Robot Soccer

RoboCup (The Robot World Cup Initiative) [8, 70] is an international competition that seeks to progress the field of Artificial Intelligence. There are a number of leagues in this competition, including both simulated and real robot leagues, with the real robot leagues divided into legged [48], small (Formula 180) and medium size (Formula 2000). The competition is based around the game of soccer. Since current robots are not nearly good enough to compete against human competitors, competition is restricted to other robots. In addition, the robots must be autonomous, although they are allowed to pool information amongst themselves. The legged and small league robots play on a pitch that is about the size of a table-tennis table, while the middle-size pitch is 9 metres by 5 metres. The competition rules are based on FIFA soccer rules, but are modified to make them appropriate for a robot soccer competition. For example, in the Formula 2000 league, the robots are allowed to dribble the ball with paddles or “fingers” but are not allowed to completely capture the ball. This is intended to be similar to the hand-ball rule in soccer. The RoboCup rules have been continuously evolving, partly to try and make the game more like real soccer, and partly to ensure that a variety of approaches can be tried and compared fairly.

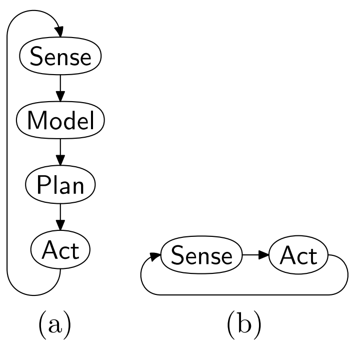

Figure 2.1: Comparison of the deliberative (a) and reactive (b) processing cycles. The deliberative cycle involves potentially time consuming modelling and planning phases, whereas reactive approaches have a tight coupling between sensory perception and action.

Robot soccer is a compelling challenge for Artificial Intelligence. Historically, researchers focused on two player turn based games such as Chess or Go. Unfortunately, for an intelligent robot attempting to move and act in a real world, the lessons learned from computer chess are not very useful. Unlike a chess game, there is little time to deliberate about actions. The position of objects are often unknown or known with little certainty, and it is often quite difficult to identify objects that can be sensed.

Since the beginning of the competition, learning behaviours for real robots has been a central research issue [70], however many real robot teams, including winning ones such as CS-Freiburg [52, 51] and Arvand [65], have used hand-coded behaviours. In the following section, the ideas behind behaviour-based robots are introduced, along with many of the techniques used to coordinate different behaviours within a single robot.

2.2 Behaviour-Based Control

Prior to the introduction of behaviour-based control, the dominant paradigm for controlling autonomous robots was based on logical symbolic inferencing. A robotic system would combine sensory information with *a priori* knowledge to form a set of beliefs about the environment. It would then apply logical inference to determine a plan of action that would achieve the desired goal. This paradigm is variously referred to as either *deliberative* or *hierarchical* [6, 99, 100]. The approach is deliberative from the point of view that prior to taking an action, the robot deliberates over the consequences of that action. It can be considered hierarchical since control is broken down from high-level goals into sub-goals and finally intermediate actions. The approach is generally characterised by a processing cycle that includes *sense*, *model*, *plan*, and *act* phases, as shown in figure 2.1(a), with each phase making use of the results of the previous one.

The concept of behaviour-based control is generally considered to have originated from Brooks [25], although Arkin [6] points out that there were a number of precursors, such as Walter's 1953 *Machina Speculatrix* [126]—a tortoise-like robot that could avoid obstacles, seek weak light, avoid strong light, and dock with a recharging station. Brooks' original approach was called the *subsumption architecture*. In comparison to the sense-model-plan-act processing cycle of deliberative approaches, the subsumption architecture has a much simpler sense-act cycle (figure 2.1(b)), and makes use of a number of relatively independent but simpler units. Rather than having a hierarchy of goals, the subsumption architecture has a series of layers, each of which subsume and build on the functionality of those beneath them. For example, the lowest layer might deal with collision avoidance, while higher layers deal with docking with recharging stations.

Each layer of the subsumption architecture contains a number of *behaviours*. Behaviours are modules that form part of the overall interaction that the robot has with its environment, and encapsulate an identifiable function or skill. In the subsumption architecture, behaviours are traditionally augmented finite state machines or AFSMs [26]. By definition they have a small amount of internal state, while also requiring minimal computation to determine the output, or actuator control signal, from the input, or sensor value.

In terms of reactive architectures, Matarić [92] draws a distinction between *purely reactive*, and *behaviour-based*. Purely reactive systems, also called reflexive systems, can be represented by a set of mappings from sensors to actuators. They have no internal memory and instead use the state of the environment as a form of memory. Behaviour-based strategies are similar to purely reactive and may have reactive elements, but they can also have internal state and an internal representation of the world.

2.2.1 Pengi

Agre and Chapman [2] used the environment of a popular computer game called PENGO to develop a reactive system that played this game. The computer game involves a grid maze with blocks that can be pushed by a penguin-like figure that the user controls. The penguin is chased by bees, but the bees can be squashed by pinning them with blocks. The reactive system that plays this game, called PENGI, does not have time to perform sophisticated reasoning about which action to take next. Instead it must react to the current situation. Agre and Chapman realised that it was easier to write a reactive system in terms of particular features of the state of the environment. For example, rather than refer to `bee#1` or `diamond#2`, they defined features such as `nearest_bee_that_is_chasing_me`. This approach dramatically simplifies the rules used to determine the next action.

2.2.2 Spreading Activation Networks

Maes [83, 84, 86] suggested an approach to solve the lack of goal orientedness of reactive systems. This approach, termed *spreading activation networks*, uses a STRIPS-like language to define how activation is spread both in a forward mode, from active behaviours to those behaviours that have the active ones as prerequisites, and in a backward mode, from goals to prerequisites of those goals. Activation levels work on a winner-take-all basis.

2.2.3 Colony Architecture

An extension of the subsumption architecture, called the *colony architecture* is provided by Connell [34]. Connell developed a robot to pick up drink cans and dispose of them. The robot, called Herbert, used a distributed system of microprocessors, each corresponding to a single behaviour. A laser range finder was used to scan for cans, while the robot wandered about. When a can was found, the robot arm would position the fingers about the can. When infrared sensors detected the can, the fingers closed on it. Detecting the can in the fingers triggered alternate behaviours so that the arm would retract, and the robot would wander in search of a rubbish bin. An example of an emergent behaviour produced by this design occurred if the can was dropped on the way to the bin. The robot would immediately start looking for cans again,

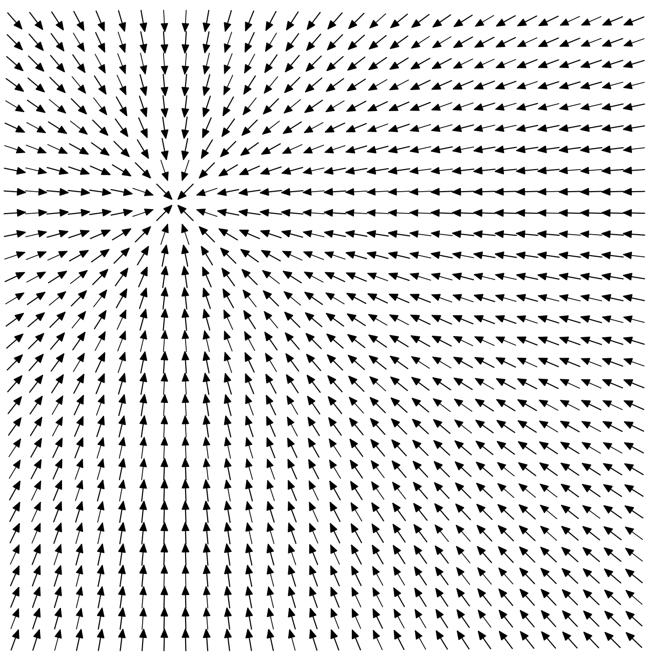

Figure 2.2: Potential field for attracting the robot towards a target region. The "force" is independent of the distance from the object.

and potentially pick up the can that it just dropped. This type of recovery from failure can be difficult to achieve when using deliberative approaches. Here, it is a natural side effect of using the "can-in-fingers" sensors to switch between behaviours.

2.2.4 Motor Schemas

One alternative to the subsumption architecture, called motor schemas, was introduced by Arkin [5, 6]. Motor schemas are based on the idea of "potential fields", some attracting the robot to goals, others repelling from obstacles or areas of danger. Figure 2.2 shows an example of a potential field that is intended to attract the robot to a target location. At the same time, other potential fields may be active, such as the obstacle avoiding potential field in figure 2.3 on the next page. To determine the combined potential field, the vectors from each active behaviour are summed together, as shown in figure 2.4 on the facing page. Note that a key insight in this work is to realise that it is not necessary to calculate vectors for the whole field, simply the ones for the current location.

Motor schemas are a simple and elegant solution to the problem of navigating in a complex or cluttered environment. There are, however, some problems with this approach. The first problem is that the robot can be slow to move around an obstacle as the vector sums tend to be small in the area where the two conflicting goals, target seeking and obstacle avoidance, are weighted equally. Another problem is the possibility of generating zero length vectors where the vectors produced by various behaviours cancel out. Arkin suggests some alterations to the basic technique to resolve these problems, however the results will still be sub-optimal in some cases.

Motor schemas are based on the position of the robot alone, and other factors, such as the speed, angular velocity and current heading, are not taken into account. Current heading is particularly important in two wheel robots as they may need to turn before they can follow the

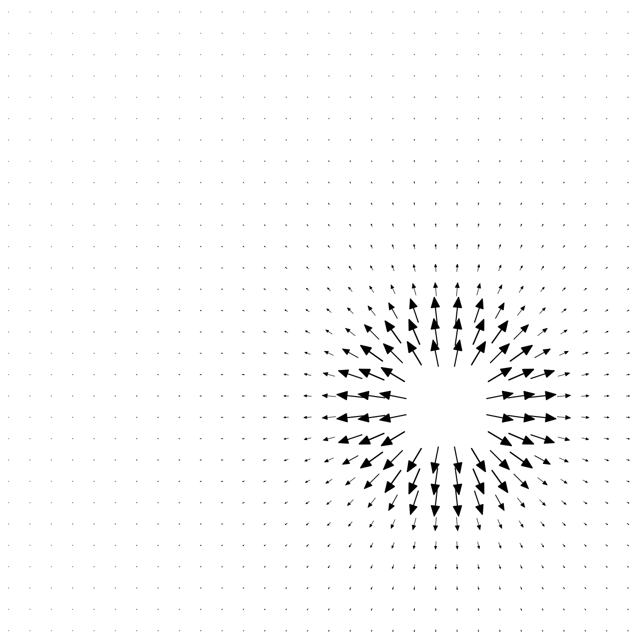

Figure 2.3: Potential field for repelling the robot away from an obstacle. The force is proportional to $\frac{1}{d^2}$, where d is the distance to the obstacle.

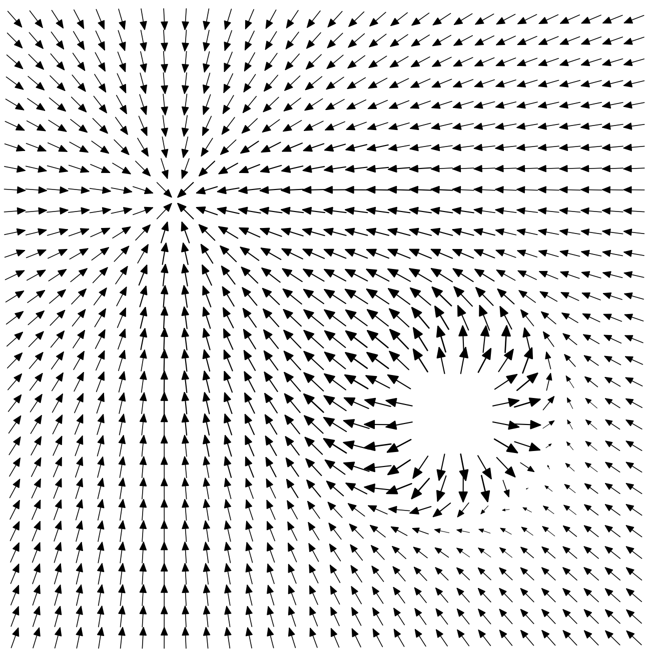

Figure 2.4: Combined potential field that corresponds to the sum of the vectors from figures 2.2 and 2.3.

vector that has been decided on.

2.2.5 Saphira

The Saphira architecture produced by Konolige *et al.* [72] is a sophisticated behaviour-based system that includes basic, reactive behaviours at the low level, a local perceptual space (LPS) that tracks nearby objects, and a planning system called PRS-lite (Procedural Reasoning System). A key feature of the behaviours in Saphira, and their coordination, is the use of fuzzy logic to allow both inputs and outputs to be described in a general way. The use of fuzzy logic with inputs means that the rules that make up a behaviour can work in terms of whether an object is near or far, rather than looking at exactly how far away the object is. When multiple behaviours of the same priority are active, their outputs are combined by looking at the weighted sum, or centroid, of the actions. Saphira also uses priority to coordinate different behaviours, allowing one behaviour to override the output of another.

Another interesting characteristic of Saphira is the combination of the local perceptual space with a technique called *anchoring*. Anchoring is the process of matching up *a priori* information about the map of an office, say, with recent sonar information, thus allowing the robot to infer its global position. The results of such inferences are available to the low-level behaviours, which means that behaviours are not completely dependent on raw sensor data when reacting to the environment.

2.2.6 CMUnited Approach

Rather than arbitrate between behaviours to solve path planning, a simpler approach, used successfully by the CMUnited Small Robot League team [22, 125], involves dealing only with the nearest obstacle. Bowling [22] developed different behaviours to move the robot to a target; to a target but around an obstacle; or to intercept a moving ball. Integrating obstacle avoidance into the motion behaviour allows this motion to be quite efficient.

2.2.7 Hybrid Architectures

Hybrid architectures, that combine low-level reactive systems with high-level planners have been investigated by a number of authors [5, 41, 49]. Hybrid architectures attempt to solve the problem that it can be difficult to "scale-up" behaviour-based systems to tasks that are much more complicated than following walls and avoiding obstacles.

Most hybrid behaviour-based systems correspond to three-layer architectures, described by Gat [49]. Such architectures commonly consist of a skill layer, a sequencing layer, and a deliberative layer. The skills layer contains behaviour modules that react to the environment in a feedback loop with minimal internal state. The sequencing layer controls which modules in the skills layer are active, usually based on the state of the world, but also based on instructions from the deliberative layer. The term "sequencing layer" may seem to indicate that the behaviours are activated in a strict sequence. However, it is necessary for the sequencing layer to take into account the current state of the world, and so this is still largely reactive. Gat allows this layer, however, the ability to take into account past information, such as where the ball was last seen. In this thesis, the term "arbitration" is preferred over "sequencing", on the basis that it avoids the non-reactive connotation, and also that it must do its most difficult work when there are two or more behaviour modules contending for control of the robot.

```
Input: Angle to target $\theta$
Output: Target left and right wheel velocities $v_l, v_r$

1. $t \leftarrow |\cos\theta| \cos\theta$
2. $r \leftarrow |\sin\theta| \sin\theta$
3. $v_l \leftarrow k(t - r)$
4. $v_r \leftarrow k(t + r)$

Note that $k$ is a constant used to control the maximum velocity, while $t$ and $r$ are temporary variables.
```

Algorithm 1: *cmu-move-to:* Bowling's move-to-target behaviour

In comparison to the first two layers, the deliberative layer is not required to react instantly to changes in the environment. Instead it performs planning or search algorithms, supplying plans or search results to the sequencing layer. As opposed to the sequencing layer considering the past, the deliberative layer tends to focus on possible future states.

2.3 Basic Movement Behaviour

The problem of how to cause a robot to move efficiently is basic and essential to autonomous mobile robots and has been extensively studied. There are a number of different types of movement behaviours that have been used with robots, such as wall-following, corridor following, and moving to a target. In this section, the focus is on behaviours for moving to a target. Wheeled robots come in a variety of different configurations, however a common form is to have two drive wheels whose accelerations can be controlled independently.

A basic approach to this movement problem is for the robot to move forward in a straight line if it is lined up with the target, or to stop and rotate if it is not. This behaviour produces slow and careful movement toward the target. If the target moves or if the robot is bumped off-course, the robot must stop and adjust its trajectory.

The next step up from this behaviour is to allow the robot to turn and move forward at the same time. Since the two wheel speeds can be controlled independently, the speeds can be held in ratio to each other, causing the robot to move in an arc. Crowley and Reignier [35] developed a simple approach to controlling the arc by mapping wheel speed controls to forward velocity and angular velocity controls. With this approach, it is straightforward to develop behaviours that turn evenly while moving forward. However, Crowley and Reignier do not provide a specific target seeking behaviour.

More recently, Bowling [22] demonstrated a set of motion behaviours based on directly setting target wheel speeds. The basic move-to-target behaviour algorithm is reproduced as algorithm 1. This behaviour, referred to here as *cmu-move-to* after the CMUnited RoboCup team that it was used by, produces quite efficient motion in comparison to behaviours that do not turn and move simultaneously. Another, more intuitive representation of algorithm 1 is to express the result in terms of average forward velocity v and angular velocity (rate of change of heading) $\dot{\phi}$. Crowley and Reignier [35] provide conversion equations, restated here as follows,

$$\dot{\phi} = \frac{1}{2S}(v_r - v_l),$$

$$v = \frac{1}{2}(v_r + v_l),$$

where S is the distance between the wheels in metres. This allows algorithm 1 to be converted into:

$$\begin{aligned}\dot{\phi} &= \frac{k}{S}(|\sin\theta|\sin\theta),\\ v &= k\,|\cos\theta|\cos\theta.\end{aligned}$$

These equations show that the turn rate is zero when the target is straight ahead or directly behind, and increases up to a maximum of $\frac{k}{S}$ radians per second when the target is perpendicular to the current direction. The turn rate is positive or negative depending on whether the target is to the left or right, respectively. The average velocity is either forward or backward depending on whether or not the target is in front or behind.

Fox *et al.* [45] developed a sophisticated behaviour-based approach to navigation and collision avoidance using a *dynamic window*. The central insight of this approach is to picture the robot's state in terms of its velocity space. In this space, defined by forward velocity along one dimension and angular velocity along the other, a real robot is limited in terms of the maximum acceleration or deceleration that it can achieve. This limitation can be expressed as a window in the velocity space, termed a dynamic window, and can be considered a velocity set. In addition, the relative position of obstacles around the robot can be mapped into the velocity space. This can be used to find the set of velocities that do not cause collisions given the current position. The intersection of the two sets yields a set of target velocities that are both physically possible given limited acceleration, and avoid collisions. Fox *et al.* then select a single forward velocity and angular velocity pair from this set by evaluating an objective function that combines considerations of how well that velocity pair achieves the target heading, clearance and velocity. This was then tested in an office environment with the RHINO robot.

An example of solving the movement problem through analysis is given by Zheng *et al.* [131]. They focus on the problem of time-optimal motion control for a mobile robot with two, independently driven wheels. They assume that the robot is initially stationary and derive a general approach for driving the robot to a target position, with or without a specific speed and angular velocity. They also assume that the space in which the robot moves is unobstructed. Their control mechanism involves three or four switching times, when the control to the wheels needs to be changed. They also show that the controls can be "bang-bang", which means that only maximum control values are needed.

Elnagar and Hussein [40] have developed an algorithm for moving a two wheel robot between two poses, that is, between one position and orientation and another position and orientation, by the application of numerical methods to solve a system of differential equations. Their approach optimises with respect to energy rather than time, and specifically attempts to minimise the integral of the squared acceleration to ensure that the movement is smooth. In addition, the movement is generalised to 3-dimensions. A key insight of this work is to realise that the smoothness of movement is different from the smoothness of the curve of the path. Jerky movement is caused by high magnitude acceleration. Avoiding this jerkiness is critical in some applications. The numerical approach to movement is tested in simulation.

Ball intercept and kicking control for robot soccer has also been analysed by Indiveri [61]. Indiveri made use of a unicycle model of the robot to derive a control formula in terms of the angular velocity. The control tends to chatter, or behave chaotically, particularly when the robot is close to the ball. A possible compensation suggested by Indiveri is to clip the angular velocity to zero when the ball is close. Indiveri also notes that the control mechanism has been successfully used by the GMD robot soccer team [61].

The approaches to movement behaviours described so far do not make use of machine learning. One particular form of machine learning, specifically, reinforcement learning, seems particularly appropriate for developing robot behaviour, and this is discussed in the next section.

2.4 Reinforcement Learning

Machine learning is an area of Artificial Intelligence that focuses on the problem of systems that improve themselves. Machine learning has been used to solve a wide variety of problems and is particularly useful with problems that cannot be solved by analysis. Machine learning approaches either involve supervised or unsupervised learning. In supervised learning, correct classification of a training set of examples is provided, whereas unsupervised learning involves finding groups or clusters without any prior identification of what the clusters are. Reinforcement learning represents an intermediate approach based on learning from experience. That is, the learning agent takes some action, senses a change in the environment and also receives a positive or negative reward. Reward may be delayed with respect to the action that caused it to be received. That is, the agent needs to be able to learn to act appropriately when the rewards are received arbitrarily far in the future [67].

Reinforcement learning seems particularly appropriate for learning robotic behaviour. Robotic problems tend to be of the sort where it is clear when the task is being done correctly but there is usually no way to determine analytically what action should be taken at any particular time. For this reason, some form of machine learning seems to be required. In addition, a sequence of actions may be required to successfully perform a task, meaning that no single action can be attributed with success, but instead, the whole sequence may have played a part. Reinforcement learning is largely focused on solving the problem of correctly assigning credit to actions that may have occurred some time prior to reward being received.

Sutton and Barto [118] provide an overview of the field of reinforcement learning. In particular, they emphasise that Markov Decision Processes form a sound theoretical basis for reinforcement learning. In this section, the theory of Markov Decision Processes and their associated algorithms are summarised. In general, the notational conventions are based on those of Sutton and Barto, with two exceptions. The one-step reward is written as $\mathcal{R}(s, a, s')$ rather than $\mathcal{R}_a^{ss'}$ and the transition probability matrix is written $\mathcal{P}(s, a, s')$ instead of $\mathcal{P}_a^{ss'}$.

2.4.1 Markov Decision Processes

Markov Decision Processes or MDPs are decision processes where the state information used to make a decision about what action to take is fully observable and has the Markov property. The term *state* refers to the current configuration of the environment in which the decision is made. For finite MDPs, it is a requirement that there are a finite number of unique states, and a finite number of actions that can be taken from each possible state.

The Markov Property requires that the state signal allow the probability distribution over possible next states be estimated given the action taken. In essence, the Markov Property is a requirement that the state signal contain enough information. The precise definition of how much information is “enough” is determined by considering whether taking into account past state signals helps in estimating what the next future state will be. Formally, a state signal has the Markov Property if the probability of an action causing a transition to a state, given the current state signal, is the same as the probability given all previous state signals including the current one, or,

$$\Pr(s_{t+1} = s'|a_t, s_t) = \Pr(s_{t+1} = s'|a_t, s_t, a_{t-1}, s_{t-1}, \ldots, a_0, s_0).$$

That is, the probability of the state signal at time $t + 1$ being s', given the state signal and action at time t, is equal to the probability of the state signal at time $t + 1$ being s', given the whole history of previous state signals and actions. For deterministic systems, this is the same as saying that the current state signal s_t and action a_t uniquely determine the next state signal s_{t+1}.

Another way to look at this issue is to consider a mobile robot moving in an area. Let the state signal correspond to its position in that area. Two example episodes are examined. At a certain time, time t, in both episodes the robot is at exactly the same position s_t. However, the path of the robot in the two episodes leading up to t were quite different. Even if the action a_t in both episodes is the same, the probability distribution of possible next states may be different. For example, the robots in the two different episodes may have been travelling in opposite directions. They may not even be facing the same way at time t. On this basis, it can be said that the state signal is not Markovian. For it to be Markovian it must at least contain the robot's direction and speed.

A state is fully observable if the state signal contains enough information to uniquely identify which state is being referred to. This is closely related to the Markov Property because, in cases where the state is not fully observable, the state can often be inferred by looking at the full history of state signals and actions. A state signal may only partially observe the state when objects are obscured from view. This situation is referred to as *partial observability* and is examined further in section 2.4.8 on page 31.

Note that it is desirable for the state signal to be compact. Algorithms based on Markov Decision Processes increase in complexity with the number of states represented. If it is possible to reduce the number of states without affecting the Markov property, then the algorithm will produce a solution more quickly and require less memory to do so.

Formally, a Markov Decision Process consists of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, the policy being followed π, a reward function $\mathcal{R}$, and a transition model $\mathcal{P}$. All possible states of the process are represented by the set $\mathcal{S}$, while actions are represented by the set $\mathcal{A}$. The decision making agent in this process follows a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. That is to say, $\pi(s, a)$ for $s \in \mathcal{S}, a \in \mathcal{A}$ represents the probability of taking action a in state s. For MDPs with deterministic policies, it is common to write $\pi : \mathcal{S} \to \mathcal{A}$ or $\pi(s) = a$ where a is the action that will be taken in state s. Immediate reward is represented by the mapping $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Re$. This can be expressed as a function $\mathcal{R}(s, a, s')$, for $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, that yields a real number representing the immediate reward for taking action a to transition from state s to state s'. Another way to think about the reward is that it is the negative of the one-step cost of taking action a and ending up in state s'. The environment in which actions are taken is represented by a transition model $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. This is a function $\mathcal{P}(s, a, s')$ that gives the probability of a transition from s to s' given that action a was taken. That is, for all $s, s' \in \mathcal{S}, a \in \mathcal{A}$,

$$\mathcal{P}(s, a, s') = \Pr(s_{t+1} = s'|s_t = s, a_t = a).$$

This means that given that the state at time t is s and the action at this time is a, then the probability that the state at time $t + 1$ will be s' is $\mathcal{P}(s, a, s')$.

For any Markov Decision Process there exists at least one optimal policy that maximises the *expected discounted total reward* for all states. The reward is *expected* since it may vary for stochastic processes. Future reward is often *discounted* by a factor γ, where $0 < \gamma \leq 1$. Discount factors < 1 allow infinitely long episodes to be studied, and also correspond intuitively with the greater desirability of an immediate reward compared with a distant future reward. A discount factor of 1 indicates that future rewards are equivalent to immediate ones, and is typically used where the task is episodic. The optimal policy maximises *total* reward, which is the accumulated one-step reward from each step in an episode.

In an episode, the agent traverses a series of states by performing an action at each step and receiving a reward, and therefore sees a sequence of states, actions, and rewards as follows:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

where s_0 is referred to as the start state. The action at step i is a_i, which is generated from the policy π and current state s_i in such a way that,

$$\Pr(a_i = a | s_i = s) = \pi(s, a).$$

The reward r_i is the one-step reward after step i, and corresponds to,

$$r_i = \mathcal{R}(s_i, a_i, s_{i+1}).$$

In the above example, the state s_n terminates the episode and therefore belongs to a subset of all states known as terminal states.

MDPs are formulated in a number of different ways. For some MDP problems, it is useful to set a finite horizon that limits the maximum number of steps possible. Indefinite horizon problems lack a specific maximum, but episodes eventually reach terminal states. Infinite horizon problems, on the other hand, may have episodes of infinite length. Sutton and Barto [118, page 60] have noted that it is possible to unify indefinite and infinite forms under one notation. Indefinite horizon problems can be made equivalent to infinite horizon problems by using *absorbing* states in place of terminal ones. An absorbing state always transitions to itself with zero reward. Furthermore, by using a discount factor less than 1, finite horizon problems with a maximum number of steps N will have total rewards that approach infinite horizon problems, in the limit as N approaches ∞.

2.4.2 Value Functions

Central to theories about Markov Decision Processes is the idea of the value of a state, or the value of a state-action pair. Roughly speaking, "value" corresponds to the state's utility, that is, how useful it is to the agent to be in that state. The value of a state-action pair corresponds to how useful it is to be in a state and then take a particular action. More specifically, the value of a state corresponds to the total expected, discounted reward of being in that state if a particular policy is applied subsequently. In addition, the value of a state-action pair corresponds to the total expected, discounted reward of first taking a particular action and then following a particular policy.

These two types of value can be represented by functions $V^\pi(s)$ and $Q^\pi(s, a)$ for the values of states and state-action pairs, respectively. Note that it is necessary to identify which policy π will be applied since this can affect the value.

Given an episode where policy π was followed, involving some infinite sequence of states $s_0, s_1, s_2, \ldots$, actions $a_0, a_1, \ldots$ and rewards $r_1, r_2, \ldots$, the value for a deterministic system can be calculated from the sum of the discounted rewards received,

$$r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots.$$

However, for a stochastic system, all paths must be considered, and the corresponding rewards weighted according to their probability to give the expected value. The value function is therefore,

$$\begin{aligned} V^\pi(s) &= E_\pi\left\{r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots \middle| s_0 = s\right\} && (2.1)\\ &= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| s_0 = s\right\}, && (2.2)\end{aligned}$$

where $E_\pi\{x\}$ corresponds to the expected value of x if the policy π is followed. From the above equation, a recursive expression can be derived for $V^\pi(s)$ in terms of the known components of the MDP,

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s')\left[\mathcal{R}(s,a,s') + \gamma V^\pi(s')\right]. \tag{2.3}$$

This is known as the Bellman equation for V^π and it forms the basis for dynamic programming algorithms that find the optimal policy for Markov Decision Processes.

The value of state-action pairs can be defined on the same basis,

$$Q^\pi(s,a) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| s_0 = s, a_0 = a\right\}. \tag{2.4}$$

That is, the expected discounted reward when starting from state s with first action a. This is also referred to as the *action-value function*. A recursive definition, equivalent to (2.3) can be derived,

$$\begin{aligned} Q^\pi(s,a) &= \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s')\left[\mathcal{R}(s,a,s') + \gamma V^\pi(s')\right] && (2.5)\\ &= \mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s') \sum_{a'\in\mathcal{A}} \pi(s',a') Q^\pi(s',a'), && (2.6)\end{aligned}$$

where $\mathcal{R}(s,a)$ is the expected one-step reward of taking action a in state s, and corresponds to $\sum_{s'} \mathcal{P}(s,a,s')\mathcal{R}(s,a,s')$.

2.4.3 Optimal Value Functions

The optimal action-value function is denoted with a $*$ and is defined as,

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) && (2.7)\\ &= \mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s') \max_{a'\in\mathcal{A}} Q^*(s',a'). && (2.8)\end{aligned}$$

for all $s \in \mathcal{S}, a \in \mathcal{A}$. Given the optimal action-value function, the action that maximises the expected total reward from state s is,

$$a^* = \arg\max_{a\in\mathcal{A}} Q^*(s,a).$$

It can be shown that a policy π that selects action a^* with probability 1, for all $s \in \mathcal{S}$, is an optimal policy.

Algorithms that find the optimal policy can be broadly classified into model-based, where a transition model is known, and model-free, where the transition model can be assumed to exist, but is unknown. In this work, the focus is on model-free approaches, and specifically on Monte Carlo methods and Sarsa, which are discussed in the following sections.

Model-based algorithms such as Policy Iteration and Value Iteration have been proven to always converge on the optimal solution [118, page 97]. Policy Iteration works by starting with an arbitrary policy π and then finding the associated values V^π. This phase is referred to as *policy evaluation* and is done iteratively by applying the update rule,

$$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') \left[\mathcal{R}(s,a,s') + \gamma V_k(s')\right],$$

for all $s \in \mathcal{S}$, until V_{k+1} is sufficiently close to V_k. This update rule is essentially an algorithmic form of the Bellman equation for V^π (2.3). Policy evaluation is followed by *policy improvement* which is based on the observation that a policy π' that is greedy with respect to V^π will always be better than or equal to π. A policy is *greedy* with respect to V^π when it selects the action that maximises the expected reward according to V^π. Also, when π' is equal to π, then π is optimal. The algorithm repeats the two phases until the policy stabilises. Note that the choice of initial policy used for π cannot prevent the optimal solution from being found, however a better choice of initial policy may reduce the number of iterations required.

2.4.4 Monte Carlo Methods

Monte Carlo approaches seek to make an estimate of the value of a state by taking a series of random samples. There are several variations on the Monte Carlo theme, but the basic algorithm is to pick a random start state, then to execute a fixed policy until the episode terminates. When the episode terminates, it is then possible to work back through the sequence of states and update each value estimate based on the total reward received and how much reward had been received up to that point. Singh and Sutton [107] show that it is important to only update on the basis of the first visit to a particular state. If this is not done, the value estimate will be biased.

This work makes use of two particular Monte Carlo variants. The first, referred to by Sutton and Barto [118] as *Monte Carlo ES*, for Monte Carlo with Exploring Starts, only explores during the first step of an episode. It does this by randomly generating the first action taken. It subsequently follows a greedy policy. In this case, a greedy policy is one that maximises the expected reward with respect to the estimate of the action-value $Q(s,a)$. The full algorithm is summarised as algorithm 2 on the next page.

This algorithm is based on generating an episode according to the current policy π. The initial state is selected at random, and the first action is also selected at random. During the episode, it is necessary to keep track of the states visited, actions tried and also the reward experienced. The value for any particular state-action pair corresponds to the discounted reward received for the rest of the trial. This is the same as the total reward received over the whole episode Q_0 less the discounted reward received so far r. Note that it is necessary to ignore repeat visits to the same state-action pair. For stochastic environments, a number of samples will be required before the correct state-action pair value (or action-value) is arrived at. These samples are averaged, and in the form of the algorithm shown here, a running average is kept. Once the action-value estimates $Q(s,a)$ have been updated, the policy is revised where necessary so that for any state that was visited, the action corresponding to the largest action-value is selected.

A potential problem with the Monte Carlo ES variant is that it only explores new actions at the start of the episode. This can be a problem if the initial state is not randomly chosen from the full set of possible states. In addition, Monte Carlo ES assumes that episodes are *proper*[118,

```
1. (Initialise policy arbitrarily.) $\pi(s) \leftarrow$ arbitrary, $Q(s,a) \leftarrow$ arbitrary, $N(s,a) = 0 \ \forall s \in \mathcal{S}, a \in \mathcal{A}$
2. Repeat forever:
    (a) select $s, a$ randomly
    (b) $r \leftarrow 0$, $R \leftarrow$empty list
    (c) (Generate an episode.) While $s$ is non-terminal
        i. Append $(s, a, r)$ to $R$ if first visit to $s, a$ this episode
        ii. (Observe next state.) $s \xrightarrow{a} s'$
        iii. (Track total reward so far.) $r \leftarrow \gamma r + \mathcal{R}(s, a, s')$
        iv. (Assign action based on current policy.) $s \leftarrow s'$, $a \leftarrow \pi(s)$
    (d) (Store the value of whole episode.) $Q_0 \leftarrow r$
        $Q_0$ is the value of the episode starting from the first, randomly selected state-action pair.
    (e) For each $(s, a, r)$ in $R$:
        i. (Update the average value recorded for the state-action pair.)
            $$Q(s,a) \leftarrow \frac{Q(s,a)N(s,a) + (Q_0 - r)}{N(s,a) + 1}$$
        ii. (Increment the number of visits to this state-action pair.)
            $$N(s,a) \leftarrow N(s,a) + 1$$
    (f) For each $s$ in $R$:
        i. (Update policy so that it is greedy with respect to the action-value.)
            $$\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a).$$
```

Algorithm 2: *mces*: Monte Carlo ES

page 111], that is, all episodes eventually reach a terminal state regardless of the initial state and actions taken.

A variant that allows occasional exploration is the Monte Carlo ϵ-soft On Policy algorithm, shown here as algorithm 3 on the following page. Instead of using a greedy policy, it "softens" the policy by making the probability of selecting the greedy action slightly less than 1. In this algorithm, ϵ refers to the probability with which a random action will be chosen rather than the greedy action. This algorithm is referred to as an "on-policy" variant since it uses the policy that it evaluates.

A strength of Monte Carlo methods is the complete evaluation of an episode prior to updating the estimated action-values. It is also a weakness since it means that the policy does not change until the episode terminates.

2.4.5 Representing Continuous State Spaces

A difficulty when attempting to apply MDP-based algorithms to real world robotic control problems is that Markov Decision Processes have a finite number of states, whereas the state of the real world is more naturally represented using continuous values. There are two approaches to resolving this dilemma. The first is simply to ignore it, and to define a finite number of states that each correspond to regions within the state space. This was used by Mahadevan and Connell [88] who encoded the state as a sequence of bits. For example, two bits were reserved for the distance associated with each sonar reading depending on whether the reading indicated a near or far object. Once the state space has been converted into a finite set, tabular reinforcement learning algorithms can be applied. Here the term *tabular* is used to refer to algorithms that store the action-value for a state in some form of table or array.

An alternative is to not represent individual states directly, but rather define a smooth function that returns the utility value given a state or state-action pair. This approach is termed *function approximation*. In general, a number of machine learning approaches, such as neural networks or genetic algorithms, can be used for function approximation. This thesis focuses on the use of a linear function for function approximation of the general form,

$$Q_t(s,a) = \vec{\theta}^T.\vec{\phi}_{sa} = \sum_{i=1}^{n} \theta_t(i)\phi_{sa}(i). \tag{2.9}$$

Here, $Q_t(s,a)$ is the estimate at time t of the value of the state-action pair s, a. The vector $\vec{\phi}_{sa}$ corresponds to the combined vector of features seen in s, a. This vector can usually be divided into a set of state features and a set of action features. The vector $\vec{\theta}_t$ contains the set of parameters at time t, which are adjusted to attempt to improve the approximation. The aim is to minimise the mean squared error (MSE) between the approximator and the function being approximated, with respect to a probability distribution over all states. The probability distribution weights the mean so that those states that are more likely to be seen in the real environment are weighted more heavily than those that occur less frequently. In reinforcement learning, this weighting is performed by sampling from complete episodes, and by ensuring that the start state is sampled realistically.

Gradient descent is used to minimise the MSE. The basic approach is to update the parameter vector based on the gradient of the action value function, or,

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[q_t - Q_t(s_t,a_t)\right]\nabla_{\vec{\theta}_t} Q_t(s_t,a_t), \tag{2.10}$$

```
1. (Initialise policy arbitrarily.) $\pi(s) \leftarrow$ arbitrary, $Q(s,a) \leftarrow$ arbitrary, $N(s,a) = 0\ \forall s \in \mathcal{S}, a \in \mathcal{A}$
2. Repeat forever:
   (a) select $s$ randomly
   (b) $r \leftarrow 0$, $R \leftarrow$empty list
   (c) (Generate an episode.) While $s$ is non-terminal
       i. (Select action.) $a \leftarrow \begin{cases} \pi(s) & \Pr(1-\epsilon) \\ \text{random} & \Pr(\epsilon) \end{cases}$
       ii. Append $(s,a,r)$ to $R$ if first visit to $s,a$ this episode
       iii. (Observe next state.) $s \xrightarrow{a} s'$
       iv. (Track total reward so far.) $r \leftarrow \gamma r + \mathcal{R}(s,a,s')$
       v. $s \leftarrow s'$
   (d) (Store the value of whole episode.) $Q_0 \leftarrow r$
       $Q_0$ is the value of the episode starting from the first, randomly selected state-action pair.
   (e) For each $(s,a,r)$ in $R$:
       i. (Update the average value recorded for the state-action pair.)
          $$Q(s,a) \leftarrow \frac{Q(s,a)N(s,a) + (Q_0 - r)}{N(s,a)+1}$$
       ii. (Increment the number of visits to this state-action pair.)
          $$N(s,a) \leftarrow N(s,a) + 1$$
   (f) For each $s$ in $R$:
       i. (Update policy so that it is greedy with respect to the action-value.)
          $$\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a).$$
```

Algorithm 3: *mcsoft*: Monte Carlo ϵ-soft On Policy

where α is the learning rate, and q_t is the experienced return. Given (2.9), the gradient of the action-value function is simply,

$$\nabla_{\vec{\theta}_t} Q_t(s,a) = \vec{\phi}_{sa}, \tag{2.11}$$

and this leads to a straightforward update.

Santamaría *et al.* [104] demonstrated that certain “skewing” transformations of the sensor data prior to using a function approximator could improve learning rates, and also benefited the final performance. The advantage of performing such transformations is that more resolution can be applied to areas of the state space that are of greater relevance to the problem, or where the gradient of the action-value function is larger.

2.4.5.1 CMAC or Tile Coding

A function approximation method that is commonly used in conjunction with reinforcement learning is the Cerebellar Model Articulation Controller or CMAC. In the original description of this technique by Albus [4, 3], the CMAC was intended to be a model for the mammalian cerebellum, and the overall system used to control a manipulator. A CMAC is an adaptive approach that allows for control functions with many degrees of freedom. Sutton and Barto [118] use the term *tile coding* to refer to the form of CMAC typically used in reinforcement learning. In being applied to reinforcement learning, a tile coding maps a state vector to an action-value rather than an action, and so the mapping needs to be queried for each possible action to determine the action that produces the greatest action value.

In its simplest form, tile coding is a grid dividing up some space, so that each point in that space corresponds to a “tile” or grid reference. Each tile has a corresponding weight. This weight corresponds to the value that the point maps to. To increase the resolution of the mapping, either the resolution of the grid can be increased, or additional grids can be added, offset from each other and from the initial grid. When multiple grids are used, the sum of the weights of each relevant tile are used to produce the value. An example tile coding with three grids is shown in figure 2.5 on the next page. The state being queried is shown as a small dot. The relevant tiles are the tiles from each grid that overlap the state. This diagram shows the situation for a two dimensional state vector, however the approach generalises to any number of dimensions. Sutton and Barto [118, page 206] point out that the offsets of each grid should be randomly chosen.

Tile coding is a form of *binary feature* encoding of the state-action pair. That is, it maps the state-action pair to a vector of binary numbers. Given that each tile weight corresponds to the parameter $\theta_t(i)$, if a state-action touches that tile, then $\phi_{sa}(i) = 1$. Conversely, $\phi_{sa}(i) = 0$ if the state-action does not touch. This can also be expressed as the feature set,

$$\mathcal{F}_{sa} = \{i : \phi_{sa}(i) = 1\}. \tag{2.12}$$

Since the set $\mathcal{F}_{sa}$ always has exactly one element per tiling grid, it can be represented as an array of indexes to tiles that touch s, a.

2.4.6 Sarsa

Sarsa(λ) [103, 118] takes its name from the state, action, reward, state, action sequence. λ refers to the use of an eligibility trace, and is actually the decay factor used. Sarsa is a variant of Temporal Difference Learning or TD(λ) [115, 116] that is appropriate for control problems. This thesis makes use of a combination of Sarsa(λ) and tile coding function approximation. The

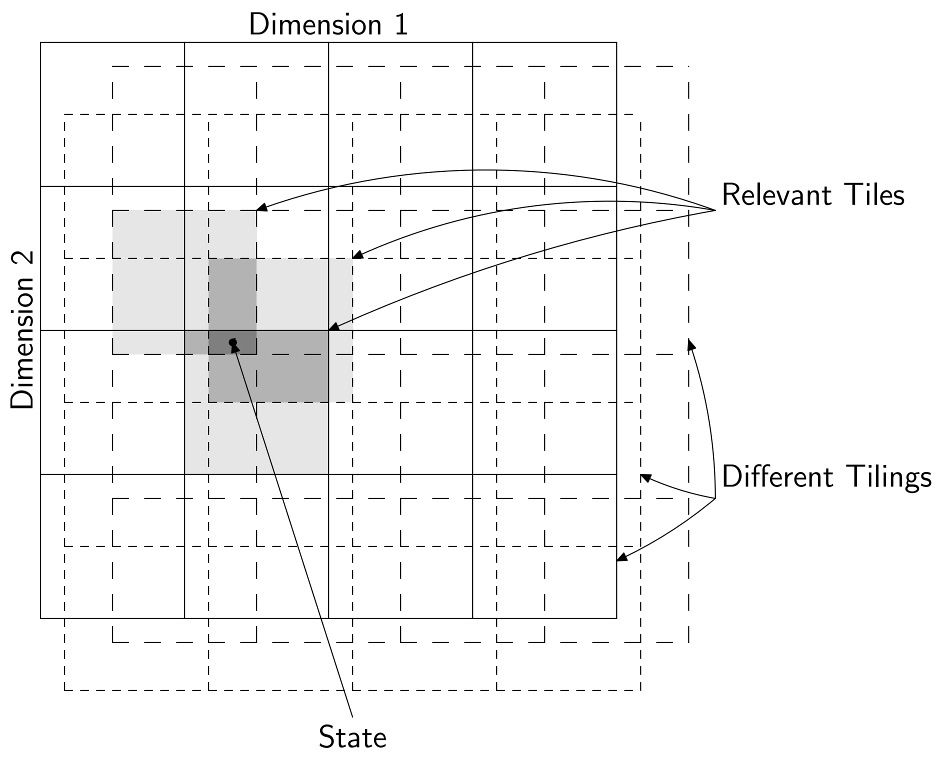

Figure 2.5: An example of tile coding. In this case, three grids or “tilings” are used, shown by the solid line, long dashed line and short dashed line. The shaded areas are three overlapping tiles.

algorithm is formally referred to by Sutton and Barto [118] as linear, gradient-descent Sarsa(λ) with binary features and an ϵ-greedy policy. It is reproduced here as algorithm 4 on the following page. The variant shown here has replacing traces, which means that when a state-action pair is visited, the corresponding eligibility trace is set to 1. The conventional approach has been to use accumulating traces, which increment the eligibility by 1 each visit, however Singh and Sutton [107] have shown that this produces bias in TD(λ) and a greater mean-squared error in the long term.

Like Monte Carlo ES and MC ϵ-soft on-policy, Sarsa(λ) is an on-policy technique. That is, the policy being followed is the one being evaluated. In comparison, off-policy algorithms evaluate one policy while following another. In the form of Sarsa(λ) shown as algorithm 4, an optional loop is included, 2(f)ii, that zeros the eligibility for any feature that corresponds to an action not taken from this state. Also, the recalculation of $Q_s(a)$ that is performed at the end of the loop at line 2(f)xii was not included in the experimental work performed in this thesis. This line is taken from the errata provided by Sutton and Barto for their book [118], but they note that leaving it out has only minor effect.

The eligibility trace, denoted $\vec{e}$, is a vector that tracks the eligibility of a weight to receive credit. This mechanism allows adjustments to the estimate of an action-value to apply not simply to the current state-action pair, but also, usually to a lesser extent, to other action-value pairs visited recently. A key factor in the use of an eligibility trace is the decay of eligibility λ. When $\lambda = 0$, no credit is assigned to past state-action pairs, whereas when $\lambda = 1$ credit is assigned equally to all previously visited states.

Sutton and Barto [118, page 207] note several modifications to the basic algorithm that can be made when implementing Sarsa(λ) with tile coding. One is to use hashing to reduce the dimensionality of the state space. Another improvement is to use sparse coding for the eligibility trace. This is particularly useful when episodes are short and the number of weights large [118, page 189]. This latter point is also mentioned by Stone and Sutton [110]. In addition, Stone and Sutton also mention the “trick” of ignoring some dimensions in some of the tilings as a way of reducing the total number of tiles.

2.4.7 Convergence Results for Reinforcement Learning

A number of reinforcement learning algorithms are guaranteed to converge to an optimal solution, in the same way that Value Iteration does, when the states and actions are finite, and the action-value function can be represented by a look-up table. Recently, Singh *et al.* [106] have proven convergence for tabular Sarsa(0). (Sarsa(0) is simply Sarsa(λ) with $\lambda = 0$.) A notable exception is Monte Carlo ES, for which there is no convergence proof as yet [117]. This is surprising as this is one of the simplest model-free approaches and seems one of the most likely to converge as it avoids bootstrapping from estimates of action-values and only uses experienced action-values.

Tsitsiklis and Van Roy [120] have shown that TD(λ) with a function approximator will converge to the optimal solution provided that a linear approximator is used and that an “on-policy” approach [118] is used. They point out that their proof does not extend to non-linear function approximators and give an example where the algorithm will produce increasingly divergent values.

This is in contrast to Baird’s counterexample [13], which demonstrated that, even where linear function approximation was used, an off-policy reinforcement learning algorithm may not converge to the optimal, and may even diverge. Baird proposed a set of variants to the standard

```
1. Initialise $\vec{\theta}$ arbitrarily
2. Repeat (for each episode):
   (a) Initialise $\vec{e} = \vec{0}$
   (b) $s \leftarrow$ initial state of episode
   (c) For all $a \in \mathcal{A}$
       i. $\mathcal{F}_{sa} \leftarrow$ set of features present in $s, a$
       ii. $Q_s(a) \leftarrow \sum_{i \in \mathcal{F}_{sa}} \theta(i)$
   (d) $a \leftarrow \arg\max_a Q_s(a)$
   (e) With probability $\epsilon$: $a \leftarrow$ a random action $\in \mathcal{A}$
   (f) Repeat (for each step of episode) until $s'$ is terminal:
       i. $\vec{e} \leftarrow \gamma\lambda\vec{e}$
       ii. For all $\bar{a} \neq a$ :
           A. For all $i \in \mathcal{F}_{s\bar{a}}$: $e(i) \leftarrow 0$
       iii. For all $i \in \mathcal{F}_{sa}$:
           A. $e(i) \leftarrow 1$ (replacing traces)
       iv. Take action $a$, observe reward $r$ and next state $s'$
       v. $\delta \leftarrow r - Q_s(a)$
       vi. For all $a \in \mathcal{A}$:
           A. $\mathcal{F}_{s'a} \leftarrow$ set of features present in $s', a$
           B. $Q_s(a) \leftarrow \sum_{i \in \mathcal{F}_{s'a}} \theta(i)$
       vii. $a' \leftarrow \arg\max_a Q_s(a)$
       viii. With probability $\epsilon$: $a' \leftarrow$ a random action $\in \mathcal{A}$
       ix. $\delta \leftarrow \delta + \gamma Q_s(a')$
       x. $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
       xi. $a \leftarrow a'$, $s \leftarrow s'$
       xii. $Q_s(a) \leftarrow \sum_{i \in \mathcal{F}_{sa}} \theta(i)$
```

Algorithm 4: *sarsa*: Linear, gradient descent Sarsa(λ) with binary features and replacing traces.

algorithms, called residual algorithms, that avoid this problem.

Sutton [117] notes that although linear Sarsa(λ) seems to perform well in many cases, there is a tendency for it to "chatter" or oscillate rather than converge. Gordon [50] found that Sarsa(0) converges to a region. That is, although Sarsa(0) may be subject to an oscillation, this is the worst case, and it does not diverge from this. Gordon points out that this bound may not be of immediate practical use since the bound may not be small enough to ensure that the state-action values are correct or even nearly correct.

2.4.8 Dealing with Partially Observable Environments

An important assumption for Markov Decision Processes is that the environment is fully observable. That is, the state of the world is completely accessible and does not contain hidden elements. Complete accessibility is rarely the case in real world environments, however. Robots are often subject to *perceptual aliasing*, where current sensory information is insufficient to clearly identify the current state. An interesting characteristic of environments that contain hidden elements is that the optimal policy often involves taking action to obtain information. For example, in robot soccer, a robot might need to move in order to see if the ball is hidden behind another robot. An extension of the MDP model to such environments, that allows the information gathering value of actions to be taken into account, is the POMDP or Partially Observable MDP.

In a POMDP, the state may not be known unambiguously, but past history may allow belief distributions over all possible states to be calculated. According to Blondel and Tsitsiklis [21], a finite-horizon POMDP can be converted into a fully observed MDP, but the complexity of the problem is PSPACE-complete. The situation is even worse for infinite horizon problems, which on conversion to MDPs have infinite states and are, in fact, undecidable [21].

When solving finite-horizon POMDP problems, the POMDP is first converted into a fully observed MDP by mapping each possible belief distribution over the set of states to a fully observed state in the MDP. There may, however, be a large number of possible belief distributions. Cassandra *et al.* [31] realised that some of the possible distributions were more important than others, and that many belief distributions could be considered to be the same as they would lead to the same action being taken. They devised the Witness algorithm to cluster similar belief distributions and applied it to several simple problems. The simplification of grouping together distributions allows POMDPs to be solved in polynomial time but as a consequence of the simplification, the solution may be slightly less than optimal in some cases. Also, even if this approach can be considered tractable, it is not typically feasible to apply it to problems where there are more than about 15 states [66].

Littman *et al.* [78] managed to increase the number of states slightly to around 100, however this is still much smaller than most realistic problems. Kaelbling *et al.* [66] realised that it would be possible to develop controllers for POMDP problems off-line using a simulation. Koenig and Simmons [71] made use of this approach to develop a POMDP navigation mechanism for a robot in an office environment. They allowed a small set of actions: turn left 90 degrees, turn right 90 degrees or move forward one metre. The POMDP allowed the robot to track the belief distributions over possible poses in an environment where many situations look the same to the robot.

POMDP navigation is somewhat similar to Markov Localisation [46], except that the POMDP is used for control as well as tracking the most probable pose. This is important as the POMDP will tend to favour actions that help to identify the state. On the other hand, experimental work with Markov Localisation by Fox *et al.* [47, 46] shows that state disambiguation tends to occur quite quickly as the robot moves around. Recent work has extended this approach by using a Monte Carlo approach to reduce the overhead of computing the updated belief distribution. This new approach is termed Monte Carlo Localisation or MCL [119, 44].

Quite a different approach to the issue of control in partially observable environments was taken by Lanzi [74]. Lanzi's approach is to include a set of internal memory registers that can be altered by the agent. These registers also form part of the state and are intended to help the agent in disambiguating states that otherwise appear to be the same. Part of the task being learnt then includes decisions about when to write to registers and what to write there. A difficulty with this approach is that the Markov Property does not necessarily hold during the learning process.

2.4.9 State Estimation

If it is not feasible to develop a control method to deal with a partially observable environment, a compromise is to make improvements in the estimate of the state of the world. Behaviour-based systems have usually taken the approach that the "world is its own best model" [28], and only made use of current sensor information when deciding what to do. However robotic sensors are notoriously unreliable and if taken at face value can cause robot behaviours to behave poorly. When the whole history of a sensor's value is examined, it is often possible to deduce more information than that available from the current value alone. Note that it is not always necessary to keep the entire history but sometimes just a summary of it. In addition, information from different sensors can be combined or "fused" to form a more reliable estimate than the sensors provide individually.

Given that individual sensors are sometimes unreliable, and that sensor values may need to be combined with other sensor values or historical information, it is often useful to provide behaviours with a processed form of the sensor data. This form is often referred to as a *world model.*

A number of different approaches have been taken to improve the quality of data received into the world model [1, 52, 54, 60, 124, 125]. For example, various forms of Kalman Filters have been used for improving the quality of vision-based data [36, 54]. The CS-Freiburg team made use of laser range-finders to develop an accurate form of localisation called scan-matching [53]. This tries to match up a scan with what is known about the shape of the room. Monte Carlo Localisation has also been used in RoboCup to improve vision-based position information [1]. Finally, teams of robots have shared information amongst team members to sense the environment cooperatively [36, 60].

2.4.10 Hierarchical Approaches

It has been shown that the optimal solution to an infinite-horizon, discounted MDP problem can be found using linear programming, and therefore the problem can be considered to have polynomial complexity, and in fact is P-complete, in terms of the number of states and actions [21, 79]. Unfortunately, many real-world and even some simulated problems have extremely large state spaces, and it is often impractical to solve them directly because of this. For example, Riedmiller *et al.* [102] note that the state space for the RoboCup simulation league is at least $(108 \times 50)^{23}$ in magnitude if position is considered alone. The large number of action variants possible mean that, in each decision cycle, the team's action space has a magnitude of 300^{11}.

A strategy that can be used to deal with such large state spaces is to decompose the larger problem into a hierarchy of learning problems [37, 55, 56, 68, 101, 105, 129]. A general form of this hierarchy is a simplified behaviour-based architecture termed *gated behaviours* [67] where only a single behaviour is active as determined by a *gating function*. The basic approach is to first learn the behaviours and then learn the gating function. There have also been a number of variations on this approach, either fixing the gating function and learning the behaviours or predefining the behaviours and learning the gating function.

Hengst [56] gives the example of a robot moving through a set of interjoining offices. In this example, the problem can be divided into the high level task of working out which office to move to next in order to get to the correct room, and the low level task of working out how to move within the room to reach the correct exit. Hengst presents the CQ algorithm that uses the ordering of the state vector to work out how to break the problem into layers. In the above example, the first element of the state vector is the room number, while subsequent parts of the vector represent the position in each room. Since the room number comes first, this is used for the upper layer of the hierarchy, while the position in the room corresponds to the lower layer. This is a straightforward approach but it depends heavily on the dimensions chosen for the state space to achieve a good breakdown of the task.

Wiering and Schmidhuber [129] developed HQ-Learning, which attempts to solve POMDPs using a hierarchical approach. Their insight was to realise that within a task that must be considered partially observable as a whole, there tend to exist sub-tasks that can be performed entirely reactively. Their approach breaks down the larger task into these sub-tasks by learning to identify sub-goal states. HQ-Learning is well suited to problems where the agent needs to solve a series of sub-tasks such as moving through a series of rooms. It also extends existing hierarchical approaches to partially observable problems.

Sun and Peterson [114] developed an algorithm that automatically partitions the state space for a reinforcement learning problem by using multiple agents. Each agent is assigned its own region, initially starting with one agent that is assigned the whole of the problem space. Partitioning is performed along one dimension of the state space at a time, based on which various heuristics to do with the Bellman residuals received for that region during reinforcement learning so far. In comparison to HQ-Learning, their algorithm does not require sub-goals to be reached for the task to be handed over. Although they state that there is no dependency on *a priori* information, the effect of dividing regions along one dimension at a time will depend on what dimensions are chosen, just as it does for the CQ algorithm.

In summary, there are a number of promising approaches for automatically, or semi-automatically dividing up large reinforcement learning problems into a number of smaller problems. These approaches tend to focus on the solvability of the sub-task, rather than whether the overall solution is optimal.

2.5 Robotic Applications of Reinforcement Learning

Robotic problems are a natural application of reinforcement learning algorithms, and this is exemplified by the frequent use of robot problems to illustrate theories about reinforcement learning.

Belker *et al.* [17] have developed navigation plan execution policies using an MDP-based approach. The main aim of their approach is to provide a low-level reactive policy that works well with a high-level path planner. To do this, the state representation of the MDP includes the robot's position and remaining intermediate destinations provided by the path planner.

In addition, it contains features for such things as minimal clearance between the robot and obstacles. The reward function is “shaped” by rewarding actions that get closer to the destination rather than only providing a reward when the robot reaches it.

Kalmár *et al.* [68] have examined a number of different reinforcement learning algorithms in the context of a module-based approach. The task is first divided up into sub-tasks, then controllers for each of these sub-tasks are learnt, and finally a coordination controller or sequencing layer is learnt. The approach is evaluated using a realistic task on a Khepera robot [98]. In the example task, the state is only partially observable. For example, one of the sub-tasks involves finding the ball. In a fully observable domain, such a task would not be required; the ball’s position would already be known. Two types of reinforcement algorithm are tried: model-based and model-free. The model-based algorithms gradually improve estimates of the transition probabilities and the value of each state, rather than estimating the action-value for each state-action pair. The reward function is also estimated rather than being supplied *a priori*. The model-free algorithm used was Q-learning [127]. Their work focuses on the problem of learning the coordination controller. For this, the model-based algorithms were found to be significantly better than the model-free ones, and were on a par with hand-crafted algorithms. The state is determined by seven binary features, yielding a state space of 128. Some behaviours are only relevant during some states, thus reducing the problem further. The authors do not state how the behaviours were developed and they may have been hand-crafted.

Asada *et al.* [11, 9, 10, 12] have examined the use of Q-learning to develop a shooting behaviour for a mobile soccer robot. The main characteristic of the work is the development of a state-space that is defined in terms of qualitative attributes, such as the ball being near or far, or the goal being to the left or right. This approach reduces the state space to about 300 different states. However, a problem with this approach is that there is considerable perceptual aliasing, since many different situations map to the same state signal. A POMDP approach for this problem would be impractical as the best results so far have worked with only a dozen or so observations [66]. Since the state does not always change quickly, one approach tried [10] was to continue to take an action until the state changed. A decay factor $\gamma < 1$ is used to encourage learning how to perform the task more rapidly.

Uchibe *et al.* [121] developed a modular reinforcement learning approach that combines state-action values learnt for different sub-tasks, such as shooting or avoiding an obstacle. The modular approach is designed to assist in quickly learning a large scale problem and is based on previous work by the same authors [12]. Hidden states, or in other words, situations where the signal does not unambiguously identify the state, are detected using statistical analysis. The approach is tested using computer simulation and also real robot experiments.

2.5.1 Reinforcement Learning Applied to Behaviour-Based Systems

The application of reinforcement learning to behaviour-based robots has been investigated by a number of researchers [58, 88, 93, 94, 97]. Some have focused on learning individual behaviours, while others provided the behaviours, and learnt their coordination.

Mahadevan and Connell [88] provided one of the first attempts at applying reinforcement learning to behaviour-based robots. The task was for the robot, called OBELIX, to push a box across the room. This involved three behaviours: *finder*, *pusher*, and *unwedger*. The selection of which behaviour to execute was determined by a hard-coded state machine. OBELIX used sonar to sense the state of its environment, however it could not sense it completely. Mahadevan and Connell reduced the information provided by the sonar array to 18 bits. This allowed 2^{18} or

262 144 possible states. They tried two approaches to reduce the complexity of the problem. The first was to reduce the state representation to 9 bits, thus reducing the number of states to 512. The other was to keep all 18 bits, but to cluster some states statistically. They found that statistical clustering produced behaviours with better performance. The best learnt behaviours were comparable in terms of performance with hand-crafted behaviours. They point out that 18 bits is a dramatic reduction on the actual state information available to the robot, and that the number of possible actions is also substantially reduced.

Hoar *et al.* [58, 130] reexamined Mahadevan and Connell's experiment with a smaller robot, appropriately[1] called ASTERIX. In their work, they found that perceptual aliasing and latency, in conjunction with the design of the reward function, produced some unexpected behaviours. Rather than developing a box pusher and wall avoider, the behaviours tended toward box avoider and wall pusher. One possible explanation for avoiding boxes is that the reward for pushing a box is outweighed by the possibility that the detected box is really a wall, and by the cost of becoming wedged against a wall. In addition, sensor latency causes the robot to be rewarded slightly while pushing a wall as the sensors indicate that the robot is still moving for a short period after the robot has stopped.

As a result, Hoar *et al.* recommend using both internal and external evaluation of the developed behaviours where the sensors are unreliable. They note that appropriate reward functions are hard to design and that some additional metric must be used to determine whether or not the correct behaviour has been produced by the learning process. They suggest using either an external quantitative metric, where the sensors may be unreliable, and a qualitative assessment in order to identify differences between the desired behaviour and that which was rewarded by the reward function. They also note that it is important for the Markov assumption to hold, if the reinforcement learning algorithms are to be applied.

Matarić [94] applied reinforcement learning to a multi-robot foraging task. The task involved a set of robots searching for, then picking up pucks, returning to a home area and depositing the pucks. The robots were given a number of fixed behaviours, such as *safe-wandering*, and *homing*, and attempted to learn how to switch between these behaviours based on a set of predicates, such as *have-puck?*, and *at-home?* Each robot learnt the problem separately using a modified form of Q-learning. For each robot, the learning problem consisted of 16 states and 4 actions. Matarić found that the style of reinforcement strategy had a significant effect on what could be learnt by the robots within a reasonable amount of time. To learn the policy correctly it was necessary not only to reinforce correct foraging behaviour, but also to provide reinforcement for increasing the distance from other robots, for moving towards the home base when a puck is held, and for moving away from the home base when nothing is held. Matarić terms these extra hints *shaped reinforcement*. Although, shaped reinforcement has the potential for introducing programmer bias, it can simplify the learning problem sufficiently that it can be learnt in a reasonable time.

Millán [97] developed a system called TESEO that used an Actor-Critic system to learn a simple navigation task. To ensure that the robot operates safely, TESEO uses a fixed set of basic reflexes in a subsumption style architecture, as a starting point for learning the task. These are used when there are no reaction rules learnt yet for the observed state or when the learnt rules are performing badly. TESEO attempts to learn the whole task, rather than individual behaviour modules, and, as Millán states, this makes it difficult to generalise to other situations or other tasks. It is similar in the way that behaviours are used as a starting point for the learning

[1]Asterix and Obelix are characters from a series of comic books by Goscinny and Uderzo. Asterix is a short, small character, while Obelix is large and fat.

process, both as way of reducing the total time to learn the task, and to improve the robustness of the behaviour during the early stages of learning.

Baltes and Lin [16] used reinforcement learning under simulation to learn a controller for a toy car on a race track. The car itself has no feedback mechanism, but its motion is tracked by an overhead camera. The aim is for the car to stay on the track and to complete a certain number of laps in the minimum time. It is modelled as a Dubins car, the optimal paths for which, according to Balluchi *et al.* [15], were originally analysed by Dubins [39]. The main restrictions for a Dubins car are that it has a fixed minimum turning radius and can only move forward. It should be noted that the optimisation provided by Dubins was in terms of minimising the path length. The resulting paths always consist of a combination of minimum radius circle arcs and line segments.

The controller developed by Baltes and Lin used an input feature vector $(\tilde{y}, \tilde{\theta}, R)^T$ where $\tilde{y}$ is the perpendicular distance from the desired path to the robot, $\tilde{\theta}$ is the angular error in the heading and R is the radius of curvature of the path. The forward speed of the robot is fixed, and control is in terms of steering only. Reward is defined in terms of the negative of a weighted sum of the position and orientation errors plus an additional cost for adjusting the steering. Including a steering cost is intended to encourage smoother control. Rather than use a discretised state space, a case-based function approximator maps the state-action space to the action-value function. The controller was developed under simulation and then used on a real robot. Although learning was continued, Baltes and Lin reported no noticeable improvement in performance due to less exploration and due to the much larger number of training cycles performed under simulation. Their results were that the developed controller was comparable in performance to the best hand-coded controller developed so far.

Balch [14] developed an environment for integrating motor schemas and reinforcement learning called CLAY. Balch's approach is a many layered hierarchical approach with the primitives corresponding to a set of predefined but parameterised motor schemas. These were used to solve team problems, such as simulated robot soccer and foraging tasks. Balch found that in most cases the learnt systems were comparable with the hand-coded ones, with one exception that was better. Balch also examined a number of other aspects of team problems, and focused particularly on the issue of diversity.

2.6 Other Approaches to Learning Behaviours

Robot behaviours have been learnt using a number of different approaches other than reinforcement learning.

Maes and Brooks [87] used a learning approach similar to reinforcement learning to teach a 6 legged robot, called GENGHIS, to walk. Learning was based on a combination of negative feedback, when the robot rests on its body, and positive feedback, when forward motion is detected by a wheel trailing behind the robot. GENGHIS successfully learnt a tripod gait that kept the body up and moved the robot forward. Interestingly, the algorithm used did not attempt to resolve the temporal credit assignment problem. That is, if a reward related to an action that occurred in the past, the algorithm would not attribute any portion of the reward to the original action. In this case, the rewards and punishments are immediate and so the problem of correctly assigning credit to past actions did not seem to occur.

Dorigo and Colombetti [38] developed a learning classifier system called ALECSYS. ALECSYS is similar to a behaviour-based system, in that it involves a set of production rules that map state to action, and a conflict resolution mechanism. ALECSYS also differentiates between internal and external messages and has an auction system for determining which messages are transmitted internally. New production rules are discovered using genetic algorithms, while rule strength is governed by an "apportionment of credit" system. A key finding was that learning was improved by using a reinforcement producing program, or RP, to "shape" the learning. The authors also provided a methodology for analysing and creating behaviours.

Genetic algorithms and other evolutionary approaches have been used widely to learn robot tasks [20, 42]. Floreano and Mondada [42] used genetic algorithms to learn a neural network that controlled a Khepera robot [98] navigating and avoiding obstacles. They also used genetic algorithms to learn a neural network for homing to a recharging area. To avoid the problem of having to wait for batteries to recharge, the robot was tethered to a power supply in a way that allowed it to move freely, thus allowing the genetic learning process to run continuously over 10 days. A "helmet" was used to allow a laser tracking system to be used to provide an external reference for the movement of the robot.

A similar approach was used by Hornby *et al.* [59] to develop a fast gait for the Sony AIBO robot. As with Floreano and Mondada's work [42], the gait was learnt on a real robot. In this case, a limited set of gait parameters were learnt, rather than attempting to learn a neural network controller. The robot was tethered and motion was evolved over 500 generations, which took 25 hours in total. Hornby *et al.* discovered that a gait learnt on a single robot was better at generalising to other robots if the gait was learnt on a ridged carpet. Ridging forces the gait to lift the feet higher to be successful and thus made the gait more robust when the characteristics of the robot changed slightly.

Blair and Sklar [19, 20] used evolutionary learning to develop a controller to play a simulated ice hockey game called *Shock*. The evolved controller is in the form of a neural network and the process of evolution learns the weights for the network. The game involves two players that each attempt to knock a puck into the opponent's goal, however learning was done with a single player attempting to shoot at an open goal. This has the advantage that it reduces the state space considerably—the controller does not need to change its actions depending on the position of the opponent. Several forms of neural networks were tried, all with 12 inputs and 4 outputs. The inputs were in terms of the global position and estimated velocity of the puck and player, as though an overhead camera existed [19]. A later enhancement [20] used local Braitenburg-style [6] sensors that gave information about distance and rough direction of the various object types relative to the position of the player. Several hundred thousand simulated trials were used to develop a controller. There was no aim to transfer the controller to a real robot as the target environment was a simulated one.

Maire and Taylor [90] used a mathematical model of the robot and its environment to develop a behaviour to cause the robot to intercept and kick a moving ball. They provide as input the position and velocity vectors of the ball and the robot, and also include the number of time steps before desired interception. The output is an acceleration vector for the next time step. The system is formulated as a quadratic programming problem; that is, the problem of finding the minima of a quadratic function that conforms to a system of linear inequalities. Two neural networks are then used; one to learn the solution to the quadratic programming problem, and the other to learn to predict the time to interception.

Maaref and Barret [81] dealt with the problem of navigating in a partially known environment. They point out that rules based on human expertise may be sub-optimal and therefore it may be better to generate the rules automatically. They use a back-propagation-like mechanism to learn the rules of a fuzzy inference system. The aim of the learning process is to minimise a cost function. Since the environment is partially known, an A^* planner is used to globally plan the route, while low-level fuzzy logic rules deal with navigating between the intermediary points of the planned route, and avoiding unexpected obstacles.

Stone and Veloso [112, 113] have done much work in the context of learning behaviours for the simulation RoboCup league. One approach made use of decision trees to distinguish between situations that were better for passing or shooting in the context of the RoboCup simulation league. An approach called Layered Learning [108] was used to develop the behaviours. First a neural network was trained to perform ball-interception. Next, a decision tree was used to learn the likelihood that, once the ball has been intercepted, a pass might succeed. Any particular player only tries to intercept the ball when it perceives that it is the nearest. Once it reaches the ball, the decision tree is used to decide what to do with it. It can either shoot at the goal, dribble, or pass to another player.

Stone *et al.* [111] have applied reinforcement learning to a sub-problem in simulated robot soccer, called "3 vs. 2 keepaway". The simulator used is the standard simulator for the RoboCup simulator league competition. The sub-problem involves 3 forward players with possession of the ball, and 2 defenders. The forwards need to keep the ball from the defenders by running to the ball, then holding the ball, or passing it. The defenders on the other hand, must tackle the robot with the ball. However, it is not useful for both defenders to try to do this, one or the other could be blocking a pass. In this case, the low-level behaviours are already provided. What is learnt is the coordination of when these behaviours should be active. TD learning is applied and is able to generate a policy that is significantly better than could be easily hand-crafted.

2.7 Simulation

The trend for behaviour-based roboticists is to avoid simulation. Brooks [29] argues, in the context of genetic programming, that there are inherent problems with transferring a program evolved under a simulated environment to a real robot. One such problem is that without regular validation on a real robot, there is the danger that much effort will be spent solving problems that only occur in simulation. Conversely, Brooks argues that programs that work well under simulation are likely to fail in the real world because simulating the actual dynamics of the real world is hard to do. Another issue is that of calibration. Robots that are supposedly the same, may be quite different. Behaviours may need to be tuned to work with each individual robot, and to do this, some mechanism is required to allow calibration. Finally, Brooks concludes that there are methodological problems with using simulations to test programs that are eventually to be used on real robots, and a need for careful validation procedures that counter these concerns. More recently, Brooks [24] has argued for more work involving physical robots with sensors and actuators operating in a physical world, while pointing out that most research is conducted using simulation, and that simulation rarely accounts for realistic physical effects.

However, Jakobi [63] has shown that simulators can be used successfully to generate neural network controllers using evolutionary algorithms. Such algorithms typically require thousands or even millions of epochs to find a near-optimal controller, so simulation is the only feasible approach. As Jakobi points out, even if sufficient resources and time existed to run millions of epochs on a real robot, the robot itself would wear out or at least its physical characteristics would change before much could be learnt.

In addition, Jakobi analyses what is required for a simulation to allow the controller learnt using it to bridge the "reality gap" [64], or in other words, for it to continue to produce the same behaviour after being transferred from the simulator to a real robot. The first issue is that the simulator cannot model *everything.* Instead, the simulator designer should focus on modelling that part of the world that is relevant to a correctly behaving controller. It is not necessary to model in detail what happens when a robot crashes into a wall since this is not a desired behaviour. As long as the simulator models such an event as something equivalently negative, then the controller will learn to avoid it. The second issue is that the simulator cannot model *any* aspect of the real world completely accurately. However, often a naïve simulation is sufficient to produce a controller with the desired behaviour.

Jakobi defines controllers to be *base set exclusive* when they rely exclusively on features of the simulator that correspond to the real world. If some part of the controller relies on a characteristic of the simulated world that does not have any correspondence with the real one, it is unlikely to transfer well from the simulator to the real robot. A controller is *base set robust* when its performance does not change significantly when moved from the simulator to a real robot. Even if the controller is base set exclusive, the features in the base set will not be modelled perfectly accurately. This inaccuracy may lead to controllers that do not maintain their performance when transferred to the real world. These issues, identified in the context of evolutionary learning, also seem to apply to learning a policy with reinforcement learning on a simulator and transferring it to a real robot.

A slightly different issue is the question of whether hand-coded behaviour developed on a simulator will transfer to a real robot successfully. A review of this field, with a focus on genetic programming, has been provided by Matarić and Cliff [91]. They point out that there are some specific challenges for work involving simulation. On the one hand, it is difficult to develop a simulator that accurately portrays the behaviour of the robot, including the modelling of noise in the sensors. On the other hand, even if such a simulator can be developed, it is then highly specialised to the robot, and perhaps even application that the robot has been used for.

Vaughan *et al.* [122] used a minimalist simulation to simulate the flocking behaviour of animals such as ducks or sheep, and designed a behaviour that herded the animals to a target point in an arena. The behaviour was then transferred to a real robot that successfully herded ducks to a target point in a circular pen. This work argues that it is possible to simulate not only physical reactions but also simple behaviour of live animals.

Miglino *et al.* [96] evolved neural controllers for a Khepera robot in computer simulations. As with Jakobi's experiments, they found that adding some conservative form of noise during simulation reduced the "performance gap" between the simulator and the real robot.

2.8 Summary

This chapter began with an introduction to robot soccer and specifically the RoboCup competition. One of the aims of this competition is to foster research into the application of machine learning to robotic control. However, most robotic behaviours are developed by hand, and a number of different architectures are used to structure this task. To understand some of the issues in developing behaviours, it is useful to look at a specific example. An interesting and general problem for robotic control, particularly for two-wheeled robots, is how to move efficiently from one point to another and there has been much research in this area.

There are many possible ways that such a task could be learnt. However, reinforcement learning approaches seem appropriate to this type of problem. One aspect that makes reinforcement learning particularly suitable is that, just as with behaviour-based control, the result is a reactive mapping between situation and action. Furthermore, reinforcement learning provides a theoretical framework for defining when this mapping is optimal.

There are a number of different reinforcement learning algorithms available, each with different characteristics. This thesis focuses on Monte Carlo ES, ϵ-soft On Policy Monte Carlo control, and linear gradient-descent Sarsa(λ). The first two make use of a tabular representation of the value of a state-action pair. The latter uses a function approximator to represent the action-value function.

These algorithms are based on the Markov Decision Process model that assumes that the state is fully observable. When the state is only partially observable, the POMDP or Partially Observable MDP model seems more appropriate. However, even though many robot problems involve a partially observable state, algorithms designed to solve POMDPs do not scale well and only small problems have been solved in this way.

A number of researchers have investigated the use of reinforcement learning, both with robots generally, and specifically in the context of a behaviour-based control architecture. In addition, other machine learning methods have been tried, such as neural networks and evolutionary algorithms. In some cases, the learnt controllers were compared to best available hand-coded controllers and found to be roughly equivalent.

Learning behaviour on a real robot can be a slow process. The use of a simulator offers the possibility of speeding up this process as well as avoiding some of the costs of running a real robot. However, the use of a simulator may be misleading and performance gains under simulation may disappear when the behaviour is transferred to the real robot. To avoid this type of problem, the simulation must be designed carefully, modelling accurately those aspects important to the problem, while providing an approximation for or ignoring entirely those aspects that are not.

This chapter has introduced the fields of behaviour-based robotics and reinforcement learning, and reviewed recent literature that is relevant to this thesis. The chapters that follow develop the contribution of this thesis, beginning with a summary of the architecture for the robot used.

Chapter 3

Robot Architecture

The previous chapter examined the current literature relevant to this thesis. This chapter describes the software and hardware architecture of the robot in detail. This lays the foundation for the subsequent chapter, which looks at how behaviours are learnt for this robot.

The two main components of the software architecture that are described here are the internal model used to represent the state of the world, and the system for coordinating behaviours.

The first section that follows describes the overall architecture, which is then broken down into hardware and software components. Software components include the world model and the behaviour-based control mechanism.

3.1 Overall Architecture

The robot architecture used in this thesis is summarised in figure 3.1 on the next page. A camera provides visual sensory input in the form of image frames, and these are processed by a vision system external to the main processing unit. The specialised vision system is described in more detail in section 3.2.1 on the following page. A summary of what is seen is then incorporated into the world model, which is described in section 3.3.1 on page 43. The world model also incorporates position and heading information (x, y, ϕ) estimated by the motor control interface. Communication between the world model and the motor control interface is two-way since the world model also sends position and heading corrections to the motor control interface based on landmark sightings.

From the world model, a state signal s is created that is specific to the active behaviour. The active behaviour acts by sending an action a to the motor control interface. The motor control interface transmits the action to the motor controller and this adjusts the power being sent to the motors. The motor controller is described in more detail in section 3.2.2 on the following page.

3.2 Hardware Architecture

In this section, the characteristics of the physical robot architecture important to this thesis are outlined. Apart from a laptop computer that sits on top of the robot, two other processors are used. One, a digital signal processor, or DSP, is used to convert digital images into a summarised form. The other controls power to the wheel motors and activates the kicking device.

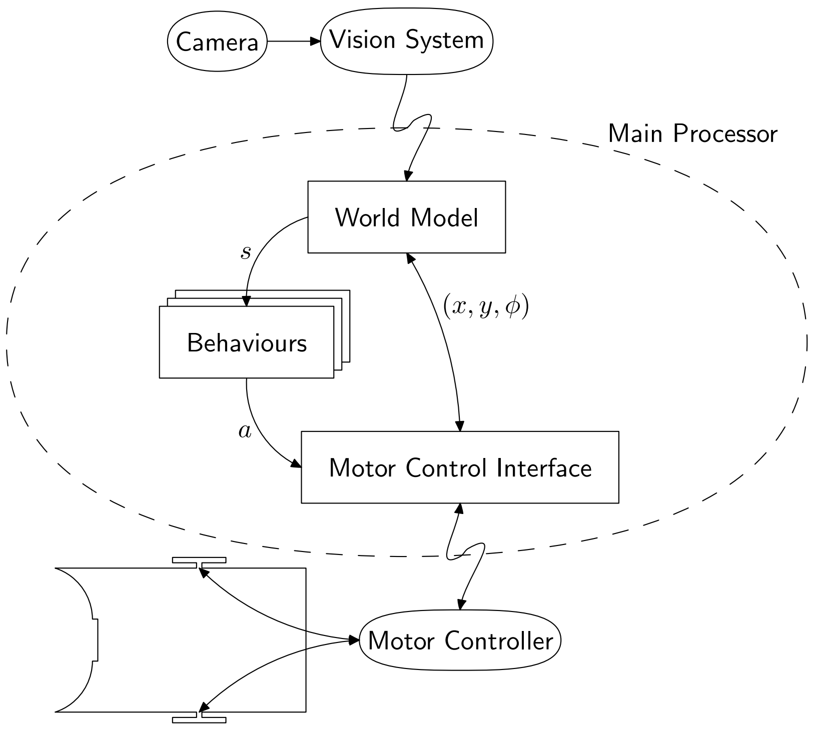

Figure 3.1: Schematic diagram of the robot architecture.

The robot hardware and vision system were created by the staff and students of the School of Electrical and Computer Engineering, RMIT University.

3.2.1 Vision System

The robot carries a CCD (charge-coupled-device) camera, that is used to capture images every 30th of a second. Visual information is complex and time consuming to process, therefore a specialised digital signal processor (DSP) was developed to segment the image based on colour, and to pass the bounding rectangle of each segment to the main processor. A Hough transform [33] was used to detect the line of the base of the wall, and this information was transformed into an estimate of the distance and relative angle of any walls seen. Information about the relative position of the wall was useful in keeping track of the robot's position. The approach is similar to that used by Iocchi and Nardi [62].

The vision processor, along with some additional computation on the main processor was able to track the position of the ball, other robots, the relative angle and distance to any walls seen, and the position of any goal posts.

3.2.2 Motor Controller

The motor controller controls the wheel motors and kicking device. The kicker is generally not relevant to this thesis and is not described further. The two wheel motors are controlled independently by dedicated motors, both of which have a shaft encoder attached to the gearbox. The encoders allow the control system to determine how much each wheel has rotated and thus estimate wheel velocities. The control system uses this information to adjust the amount of power to the wheel, to try to maintain a specific velocity for each wheel. To do this, a proportional-integral-derivative, or PID, controller is used. This controller updates the amount of power sent to the motors 100 times per second.

The high-level controller communicates with the low-level controller via a 9600 baud serial link. It receives information about how far each wheel has moved, and in what direction, ten times per second. At this time, it can optionally send a command to adjust the target wheel velocities. The advantage of using the low-level controller to adjust power, rather than the high-level controller, is that the low-level one operates on a much faster cycle, and therefore the power can be adjusted with greater accuracy.

The shaft encoder information provided by the low-level can be integrated by the high-level to estimate the position of the robot. This estimate assumes that the wheels do not slip. Although this estimate is usually sufficiently accurate for short term estimate of position, in practice, it cannot be used for a longer term with any accuracy.

3.3 Software Architecture

The two main components of the software architecture are the "world model" and the behaviour system. The world model is a combination of the internal representation of the state of the robot's external environment and the mechanism used to update that internal representation. The behaviour system coordinates a set of behaviours, controlling which ones cause the robot to act.

3.3.1 World Model

The state of the world is not completely or directly available to a robot. At best, robot sensors can provide evidence about some part of the state. When considering this evidence, it is necessary to reason about what is being measured, and to allow for the possibility of sensor noise or faulty sensors. For example, sonar devices are often used to detect obstacles around a mobile robot. However, what is being measured is the amount of time for an ultrasonic ping to be returned. As well as being subject to noise, it is subject to interference from other equipment, and variations in how well obstacles reflect sound. A sonar device can provide evidence about how far away an object is, but not conclusive evidence.

To assist with this distinction between observations and what they imply, the term *belief* is used for an observation or set of observations that have been interpreted. Typically, beliefs correspond to some aspect of the true state of the world. However it needs to be remembered that beliefs cannot be guaranteed to be reliable and errors are possible, either through sensor error or due to errors in interpreting what the sensor data imply.

In this architecture, beliefs are simply stored as one or more real-valued variables. For example, the position of the ball is stored as a 2-dimensional Cartesian coordinate position based on a fixed field reference point. Ball position information is originally obtained using vision information by extracting the red coloured segment and using its position in the field of view to estimate the ball's relative position. The relative location of the ball is then translated to an absolute position based on the robot's information about its own position and orientation.

3.3.1.1 Hysteresis

When using beliefs to determine action, it is common to determine some form of threshold point where, once the value associated with the belief reaches the threshold, the behaviour changes its action, or possibly the behaviour becomes active or inactive. The problem with using thresholds is that when near to the threshold, noise in the sensory inputs can cause the behaviour to toggle on and off in an undesirable way. One possible resolution of this problem is to use two thresholds rather than one, and to require the more difficult or higher threshold to be reached before the behaviour is changed, and then once the behaviour is changed, the other threshold is used to trigger when to change the behaviour back. This is similar to the way a thermostat works. When the temperature drops below a certain threshold, the heater turns on, and it stays on until a higher threshold is reached. This avoids turning the heater on and off too frequently. The difference between the two thresholds is known as the *hysteresis.*

Hysteresis is particularly useful in specifying the context condition for a behaviour's activation. Context conditions are described in section 3.3.2 on the next page. However, built-in hysteresis runs counter to the approach that will be used in learning behaviours, since it predefines, to some extent, the way that the behaviour acts, and thus limits what can be learnt. Instead, the approach described in this thesis is to provide the behaviour with a representation that roughly corresponds to the true state. For example, rather than use a flag to indicate if the ball is in the defensive half, and then apply hysteresis to avoid that flag from switching too frequently, the preferred approach is to provide the position of the ball. This allows the learning process to decide where the thresholds should lie. In addition, consideration of the last action taken, or the effects of the last action, such as the robot's velocity, allows the learning process to properly assess the advantages of continuing with an action versus changing action and thus avoid vacillation.

The use of hysteresis to avoid chattering decision making for RoboCup robots has previously been suggested by Castelpietra [32] in the context of coordinating actions amongst robotic team members.

3.3.1.2 Allocentric Representation

It is common in behaviour-based robotics to prefer internal models that are relative to the robot [2, 10, 27, 34, 72]. That is, it is usual to make use of an *egocentric* representation of the positions of objects. An immediate difficulty with an egocentric point of view is that it is difficult to relate what is currently perceived to what has been perceived previously, if the robot has moved. For example, if the robot sees the ball at an angle of 15 degrees to the right, and 3 metres away, and then moves forward 1 metre but can no longer see the ball, where should it expect the ball to be? Many behaviour-based systems avoid this problem by not keeping an internal representation of where the ball is. Non-representational systems often justify their approach by pointing to the unreliability of past information [27]. Perhaps the ball was never there, and faulty sensors led the robot to this conclusion erroneously. However, as sensor technology improves, the need to incorporate past sensor information into the representation of the world is important enough to require internal representation and reasoning about historical data.

The alternative, which is used in this work, is to have an absolute coordinate system for representing the position of objects external to the robot. This is called an *allocentric* or non-egocentric representation [57]. An allocentric representation allows the robot to incorporate past beliefs more readily. In addition, converting such beliefs into a relative position is a straightforward calculation, should this be required by a behaviour.

3.3.2 Behaviour System

The behaviour system used in this thesis uses two mechanisms: context and priority. Context determines whether a behaviour is applicable to the situation, whereas priority allows one behaviour to override another.

Formally, the system for a single robot consists of a tuple $\langle \mathcal{B}, \pi, C, P, \mathcal{S}, \mathcal{A} \rangle$. The set of behaviours $\mathcal{B}$ are each associated with a context condition $C : \mathcal{B} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$, a priority ordering $P : \mathcal{B} \rightarrow \{0, 1, 2, \ldots\}$, and a policy $\pi : \mathcal{B} \times \mathcal{S} \rightarrow \mathcal{A}$ mapping between possible states $\mathcal{S}$ and corresponding actions to be taken $\mathcal{A}$. For a particular robot, the priority ordering is unambiguous, so that $P(a) = P(b) \Rightarrow a = b$ for all $a, b \in \mathcal{B}$. This means that different behaviours cannot share the same priority. The context condition $C(b, s)$ is true for a behaviour $b \in \mathcal{B}$ that is in context, or active, based on the current state $s \in \mathcal{S}$. However, it only has control of the robot if it has the highest priority $P(b)$. Specifically, behaviour b controls the robot when the state is s if $C(b, s) \wedge (\forall b' \neg C(b', s) \vee P(b') < P(b))$. That is, if b is in context and no other behaviour is both in context and has a higher priority. In this case, the robot's action is determined by $\pi(b, s)$. In this thesis, only the policy mapping π is learnt. The context condition and priority are predetermined and fixed for the task. Note that it is unnecessary to learn $\pi(b, s)$ for states where the behaviour is out of context.

This design avoids the more complex arbitration of the subsumption architecture [25, 26], motor schemas [6], or even Saphira [72], by only allowing one active behaviour to control the robot. However, a problem with only allowing one behaviour to control the robot is that design complexity is moved from arbitration into the behaviour.

A simple example helps to illustrate this point. Consider a robot attempting to navigate its way across a room containing obstacles. Most arbitration schemes allow the problems of avoiding obstacles and moving toward a target to be expressed as two or more separate behaviour modules. However, if only one behaviour is controlling the robot, then this behaviour will need to incorporate both competing requirements. Bowling and Veloso [23] give an example of how this can be done by using single behaviours that incorporate both movement to a target point and avoiding an obstacle.

The single controlling behaviour approach can produce more efficient motion, simply because the single behaviour is able to take into account all factors. This also means that behaviour modules designed using this approach may be more difficult to program, thus making learning of behaviours more attractive.

3.3.3 Decision Cycle

Different sensors provide readings at different intervals. For example, certain types of electronic compass provide readings 2 or 3 times a second, whereas vision systems usually operate at about 30 times a second. Actuators generally operate on a certain cycle also, and for the physical robot used here, that cycle allowed updates 10 times per second. Of these cycle times, the key cycle is that of the actuator. If the robot processes sensor readings, decides on an action and transmits actuator signals faster than the actuator can receive them, then some signals will be lost. If the robot is restricted to processing data less frequently than the actuator can process, and if at least some sensors provide new information at least as regularly as the actuator can receive commands, some opportunities to react to this new data may be lost. Therefore, as the vision system returns data more frequently than the actuator cycle, the decision cycle is tied to the actuator cycle.

3.4 Related Work

The behaviour-based architecture used here is quite different from the original subsumption architecture proposed by Brooks [25, 26]. In the subsumption architecture, behaviours override each other based on levels, a wiring diagram and explicit suppression or inhibition signals. In this architecture however, behaviours have a strict priority ordering and outputs are overridden. Also, in this architecture, all sensor information is processed into a world model before being used. However, it is desirable to ensure that sufficient real sensor evidence is used to form the world model, so that behaviours are not based on spurious data.

The architecture used here is different from the motor schemas architecture proposed by Arkin [6], in that behaviour output is not formed into a vector sum, and it is not possible to use multiple behaviours to affect a single actuator. In the preliminary experiments performed with the soccer robots used here, motor schemas were tried and found to produce less efficient movement compared with using a single movement behaviour that combined obstacle avoidance and movement.

Like Saphira [72], a priority system is used to resolve the problem of having multiple active behaviours, however, unlike Saphira, a strict ordering is required. In Saphira, behaviours can have the same priority, and if they do, their outputs are summed using fuzzy logic. The resulting action is determined from the centroid, or weighted sum, of the fuzzy logic sum. As with Arkin's motor schemas, this arbitration mechanism can sometimes cause the robot to perform an action that was not requested by any of the active behaviours. For example, if one behaviour wants the robot to turn left, and the other wants the robot to turn right, the centroid approach may cause the robot to not turn at all. By requiring a strict ordering of behaviours by priority, this architecture avoids the situation of two behaviours being active simultaneously. This is similar to the gated behaviour approach used by some hierarchical approaches [67].

3.5 Summary

The robot architecture is a combination of hardware and software. The robot hardware consists of a vision system and motor controller, while the software is a combination of a world model, and a behaviour system. The world model is a summary of current and historical sensor data, and is of an interpreted form that allows multiple data sources to be fused. The behaviour system is structured to allow only a single active behaviour at any one time. This combined configuration of a summary world model and single controlling behaviour were initially chosen to develop efficient hand-coded behaviours, but is also suitable for learning behaviours with reinforcement learning.

Chapter 4

Developing Behaviours through Learning

In the previous chapter, a behaviour-based architecture that provides a good basis for learning behaviours through reinforcement learning was described. This chapter focuses on how this work extends existing approaches to robot learning. It details a methodology for using reinforcement learning algorithms in a simulator framework to create real robot behaviours. It explains several approaches to designing the behaviour-based system to reduce complexity when learning individual behaviours. Finally it presents a novel approach to bootstrap reinforcement learning from a hand-coded behaviour in a way that does not bias the solution.

Traditionally, experiments with robots have focused on whether or not the outcome was successful, rather than the comparative speed or efficiency. This is mainly because successful outcomes were infrequent. Real world environments are highly complex, and sensory information is limited and unreliable. The problem domain of robot soccer requires a slight change of emphasis. Although the robot soccer environment is still complex, the complexity has been restricted. A map of the area is known precisely beforehand, and does not need to be charted. Objects can be clearly identified by their colour. In addition, recent sensor technology improvements have allowed researchers to greatly increase the robots ability to accurately estimate the position of the ball, and other robots. Given these changes, soccer playing robots are now competing on the basis of such things as: how fast they can get to the ball, how quickly they can pass or shoot the ball when they get to it, their overall strategy, and how well they work as a team. The first two correspond to the performance of individual behaviours. The latter two are important, but good strategy and teamwork will fail if the behaviours used are inefficient or ineffectual.

Given that the performance of individual behaviours is an important factor in the performance of the whole robot, it seems sensible to try to improve their performance. On the other hand, only improving individual behaviours and not the overall strategy means the resulting overall performance may not be optimal, simply because the strategy may not be optimal.

This raises the question of whether learning should be applied to the task as a whole. The problem with attempting to learn the whole task is that the complexity is often too great to deal with in this way. An alternative, favoured by hierarchical approaches to reinforcement learning, is to factor the task is some way, and then to learn the hierarchy (see section 2.4.10 on page 32). As Kaelbling et al. [67] point out, hierarchical methods correspond to the combination of a gating function and a set of gated behaviours. Here the focus is on learning gated behaviours given an existing gating function.

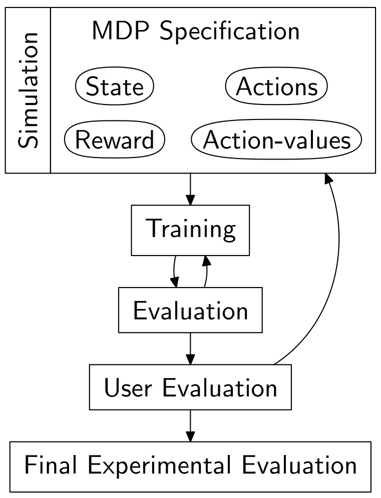

Figure 4.1: Process for learning a robot behaviour module.

4.1 Learning Methodology

The learning process used in this thesis is summarised in figure 4.1. The first step in the process is to formulate the problem as a Markov Decision Process and to design a simulated environment. The MDP formulation is broken into four related parts: defining the state, the set of actions, a reward function, and how the action-values will be represented. The next stage is to perform training and evaluation. This is an iterative process involving some number of training episodes followed by a set of evaluation episodes. When this has finished, the user may need to review the behaviour produced and see if it matches expectations. This step is required since it is often the case that some part of the specification is faulty. For example, there may be insufficient penalty for causing the robot to crash into an obstacle, leading to a behaviour that causes the robot to crash too frequently. Another potential problem is that the behaviour may "exploit" the reward function or simulator in unexpected ways. For example, during preliminary experiments, a learnt behaviour for moving to a target reversed to the target in nearly all cases. It was later discovered that the simulator did not properly limit acceleration in reverse. Similarly, several design iterations may be required to determine an appropriate reward function.

The final step in the process is to evaluate the performance of the behaviour on the physical robot, to obtain a final measure of the behaviour's performance.

Unlike other machine learning approaches, reinforcement learning generally proceeds by generating scenarios or start states at random, rather than selecting them from a training set. Furthermore, performance of the learnt policy is usually estimated by looking at recent returns from the learning process, instead of testing with an independent set of scenarios. This difference is due to the idea that reinforcement learning mechanisms continue to learn while they are being usefully applied. In other words, reinforcement learning generally focuses on the overall performance of the agent including the performance during learning, rather than focusing on how well the agent has learnt the task, or the performance after learning has completed.

In this thesis, most learning occurs on the simulator, and where this is the case, it is possible to evaluate how well the agent has learnt the task by using a set of test cases. The test set is generated independently from the training set but some scenarios may exist in both. By using a predefined test set to test the performance of the policy, it is possible to compare the result of this test at one stage of the learning process with the test result at a different stage. It also becomes more straightforward to compare one learning algorithm with another. If the learning process improves the overall policy, it will tend to improve the test set result.

The main impetus for using a test set, rather than simply looking at average returns during learning, is based on the observation that the average reward returned from a set of episodes sometimes decreased even when the policy had been improved. This was particularly noticeable when the reward function contained a discontinuity; say, a large negative return corresponding to a rarely occurring situation, such as the robot crashing into a wall. Typically some starting states are more prone to causing the robot to crash. For the problems examined, the selection of start state was a factor in whether or not an extreme value was obtained for the reward. Therefore, using a standard test set ensured that average test rewards could be fairly compared.

Another factor that might affect test results is the amount of exploration being pursued. Exploration occurs where the learning algorithm occasionally deviates from the policy. If the policy is nearly optimal, but the learning algorithm includes a significant amount of exploration, then the exploration may make the policy appear to be less optimal than if no exploration were performed. To ensure that results from different algorithms can be compared, the evaluation does not include exploration.

This thesis makes use of algorithms that alter the policy during an episode, such as Sarsa(λ) (see algorithm 4 on page 30), and those that do not, such as Monte Carlo ES (see algorithm 2 on page 24). To allow the policy produced by both of these to be fairly compared, no alteration of the policy is permitted during the evaluation. This has the advantage that, since the same test set is used periodically for evaluation, the possibility of this test set dominating the learning process is avoided, and learning episodes are based on a uniform random selection from the complete set of possible starting states.

4.2 Are Behaviours MDPs?

The argument against applying the Markov Decision Process model to behaviour-based robots has been made by Matarić [93]. Matarić argued that the agent-environment interaction does not follow MDP assumptions. Assumptions made by the MDP model, that do not appear to hold for real world robots are:

1. That the agent interacts with the environment in lock-step, and at discrete time intervals,
2. That states and actions are finite, and
3. That the state of the environment is fully observable.

Violation of these assumptions does not necessarily mean that it will not be possible to produce a valid behaviour. However, it may prevent MDP-based algorithms from finding the optimal behaviour or even from converging on a solution.

The assumption that the agent interacts in lock-step with the environment would require that decisions about what action to take were made instantaneously, and that the transmission of sensor and actuator data were also instantaneous. Real robots are not capable of doing this and are subject to a latency or delay between when they sense the environment and when they act. Although it is common for a robot to interact at discrete time intervals, the latency prevents this interaction from being in lock-step with changes in the environment.

The assumption that states and actions are finite does not hold in most real-world environments. This work uses function approximation, introduced in section 2.4.5 on page 25, allowing MDP algorithms to be applied to continuous state spaces. Although robots typically allow only a finite set of actions to be taken, this set can be extremely large. The approach used in this work is to use a limited subset of the possible set of actions.

The assumption that the state of the environment is fully observable is a critical one, and was discussed in section 2.4.8 on page 31. Some researchers [11, 58, 89] apply reinforcement learning to environments that clearly violate this assumption without attempting to resolve the issues of partial observability. Others [71, 78] use a POMDP model. However there are severe limitations to the number of states that can be feasibly dealt with in this way (see section 2.4.8 on page 31). Wyatt *et al.* [130] suggested that violations of Markov assumptions should be addressed by adding sensors or changing the way that they are processed. They point out that this process can consume a significant part of the development effort.

The approach used here is to attempt to resolve partial observability in two ways: First, the target behaviour is derived using a simulator with a fully observable environment. Second, the real robot makes use of a state estimator or world model that attempts to predict the full state by integrating current sensor data with past history of sensor data and actions taken.

Simulators are often used for learning robotic tasks when it would be too time consuming to learn using a real robot (see section 2.7 on page 38). Generally, the trend is for simulators to be as realistic as possible. For example, Jakobi *et al.* [64] recommend simulating sensor noise. In the approach described in this thesis, the simulator provides a fully observable environment. For example, the position of the ball is available to the behaviour even when it is behind the robot and would normally be hidden from view. The aim of this is to avoid violating the assumption of full observability during learning.

This approach necessarily shifts the burden of interpreting sensory input to the state estimator that operates on the real robot. Since the state signal provided by the real robot's world model is only an estimate, it will sometimes be incorrect. Unlike the POMDP approach—where if the state estimate is known to be poor, then this is allowed for when deciding on an action—this approach ignores the uncertainty in the estimate of the state. The experience in this work is that it was possible to derive a sufficiently accurate state estimate for there to be no significant side-effects to ignoring the uncertainty in the state estimate.

Although there are some remaining reservations about the use of the MDP model, this work will demonstrate that it is possible to use this model to do effective learning.

4.3 Task Decomposition

The problem of task decomposition, or sub-dividing a task into smaller, more manageable chunks, has been investigated extensively. In particular there has been much recent work in the area of simplifying reinforcement learning tasks by finding a suitable decomposition, either through geometric characteristics of the problem, or through teleo-reactive planning. Recent literature in this area has been covered in section 2.4.10 on page 32. Although the focus of this thesis is mainly on the problem of learning behaviours within an existing task decomposition—specifically, the decomposition designed for a robot soccer team—the use of reinforcement learning introduces some constraints on the way that the task is divided up.

To illustrate the issues involved in decomposing a larger task into behaviours that can be converted to MDPs, it is useful to look at the issues that led to the design of one of the behavioural tasks.

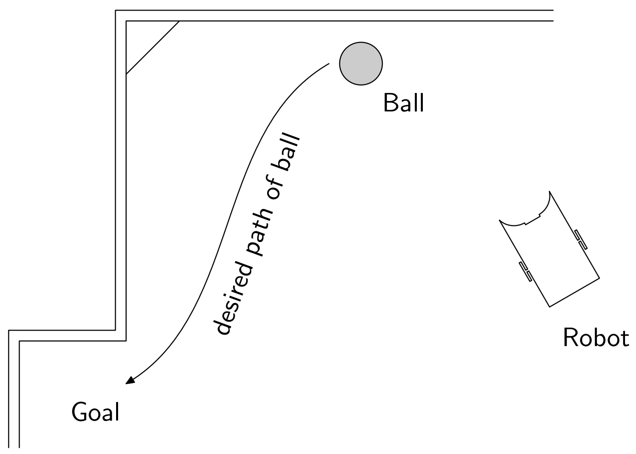

Figure 4.2: Illustration of the `NearWallIntercept` behaviour.

The tasks examined in this thesis were derived from the set of behaviours used by the RMIT-United Formula 2000 RoboCup team [30]. For example, the "off wall" task that will be examined in chapter 8 on page 113 was derived from the `NearWallIntercept` behaviour. The aim of the `NearWallIntercept` behaviour is to get the ball away from the wall and / or kick it toward the opponent's goal as illustrated in figure 4.2. This behaviour is in context if the ball position is known and within 300 millimetres from a wall, if the robot is meant to be chasing the ball rather than supporting or defending, and if the ball is not near a goal. Hysteresis is used so that once the behaviour is active, the ball only needs to be within 350 millimetres of the wall for the behaviour to continue to be active. This behaviour actually comprises three sub-behaviours that are executed according to where the robot is with respect to the ball. When the robot is far from the ball, it moves to a point beside the ball. In this case, "beside" means on the other side of the ball from the wall, wherever the wall is, and 500 millimetres from the wall. If the ball is in a corner, "beside" means 500 millimetres from each of the walls involved.

Once the robot is roughly beside the ball, it then moves toward the ball until the ball is within reach. If at any stage, the robot is no longer beside the ball, it drops back to the first sub-behaviour of moving beside the ball. When the ball is within reach it spins on the spot. A number of rules were used to determine which way to spin. If the ball is in the offensive half and the ball is on the right relative to the opponent's goal, the robot spins counter-clockwise. If the ball is in the defensive half and it is on the left with respect to the own goal, the robot also spins counter-clockwise. Otherwise, the robot spins in a clockwise direction.

As described above, this behaviour would require a large number of states when framed as a Markov Decision Process. In this work, the first step to reducing the number of states was to divide up the task into moving beside the ball and then kicking the ball away from the wall. Second, kicking the ball when it is near a straight segment of wall is different from kicking it when it is near a corner and so these tasks were considered separately. Third, the direction of the kick depends on which wall the ball is near and which goal is being played towards. Rather than try to make a behaviour that handles all situations, this sub-task was dealt with as a series of separate behaviours.

This series of decompositions reduced the size of the state space for each individual behaviour, but created a large number of behaviours where previously there was only one. In this thesis, it is argued that this is not necessarily a limitation to the approach as a small set of learnt behaviours can be reused for symmetrical situations. For example, a behaviour that kicks one way will be the mirror image of a behaviour that kicks the other way. Similarly, a behaviour that kicks the ball west on a north wall will be the same as a behaviour that kicks the ball east on a south wall.

If the behaviour has learnt how to kick toward a goal at the east end, it will be necessary to provide the behaviour with a mirror image of the state information to get it to kick toward the west end. Note that if a behaviour is to be reused in a mirror image situation then the actions will also need to be reflected. In this case, this can be done by swapping the left wheel command with the command for the right wheel.

Reuse through symmetry has been suggested previously by Lin [77] in a slightly different fashion to that developed in this work. In Lin's case, the agent could move in one of four directions, north, south, east or west, in a grid. Sensory information would be rotated so that each action appeared as "go north" and a utility established for each.

Lin's approach assumes that the actions are few and that they match up with a rotation of the state space. The approach used here extends this idea in that while the sensory information is still transformed, the result of the behaviour is an action rather than a utility, and this action may need to be transformed. This allows mirror image situations to be handled by the same behaviour rather than just ones that are similar under rotation. Another reason why Lin's approach could not be applied here is that it relies on the robot being able to move laterally as easily as moving forward or backward. This is not the case for a two wheel robot, which must rotate to go in a direction other than forwards or backwards.

4.4 MDP Specification

Behaviours are often developed with loose, descriptive specifications, such as "kick the ball away from the wall", or "shoot the ball into the goal". To develop a behaviour as a Markov Decision Problem, it is necessary to specify the task more precisely. For example, specifying the reward function defines a performance measure for the task so that it becomes possible to determine whether or not one behaviour is better than another.

Referring back to figure 4.1 on page 48, specification of the MDP can be broken down into defining the state, set of actions, reward function, and action-value representation. These components are examined in turn below.

4.4.1 Defining the State Vector

The state vector is a subset of the world model, or total set of beliefs, that is supplied to the behaviour during each decision cycle. In reinforcement learning terms, this corresponds to the state signal, which was described in section 2.4.1 on page 19. It is convenient to represent the state signal as a vector in n-space $\vec{x} = (x(1), x(2), \ldots, x(n))^T$. Each component of this n-dimensional vector corresponds to a real valued aspect of the state of the world. For example, one component might be the distance to the ball, while another, the relative angle to it.

The state vector should be as compact a representation of the world as possible, while maintaining the Markov property. Note that the Markov property must be examined from the point of view of the task that the behaviour is being developed to perform. This also means that the state vector will be different for different behaviours.

To illustrate the issues involved in defining the state vector, consider the task of moving to a target location for a two wheeled robot. This task will be examined in detail in chapter 6

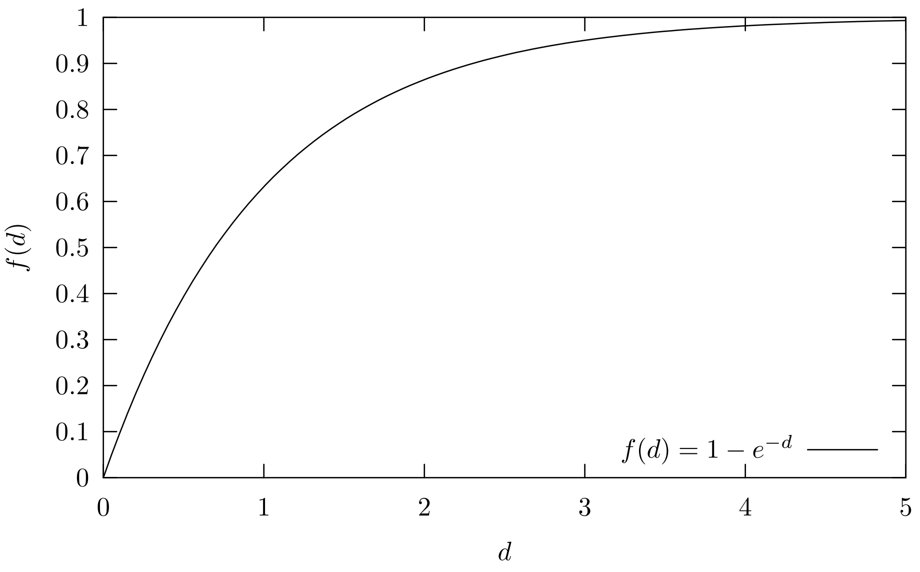

Figure 4.3: Transform used with distance d to distribute more resolution of the tabular or tile coding to states closer to the target.

on page 75. Relevant aspects of the state would seem to include where the robot is currently, and where it is moving to. In this task, the heading of the robot is also important, since that affects how the actions change the robot's position. Also, it is assumed that there are no obstacles. Given that the task is symmetric under translation, the state can be simplified somewhat. Specifically, the four coordinates can be shifted so that the target position is always at the origin $(0, 0)$. The state vector can therefore be encoded by as $(x, y, \phi)^T$ where x and y give the Cartesian coordinates of the robot relative to the target and ϕ gives its heading. This seems to be all that is required for a robot attempting to intercept a fixed point. However, if the dynamics of the robot are taken into account, the momentum and angular momentum are important. Assuming the mass and shape of the robot is constant, momentum and angular momentum can be coded as left and right wheel velocity (v_l and v_r), yielding an expanded state vector $(x, y, \phi, v_l, v_r)^T$. Since the choice of the orientation in the Cartesian axes is irrelevant (in this case, at least), the position and heading information can be simplified to distance plus relative angle to target θ. At this stage, the state vector is of the form $(d, \theta, v_l, v_r)^T$. A potential problem, however is that it is necessary to provide an upper limit on the distance. At the same time, given the nature of the task, it seems likely that more resolution will be required when the distance is small. Also it is desirable to generate a behaviour that will handle any distance. Therefore, the distance is mapped using a function $f(d)$, that is defined as,

$$f(d) = 1 - e^{-kd}, \tag{4.1}$$

which has a domain of $[0, \infty)$ and a range of $[0, 1]$. The constant k controls the shape of the curve and was set at 1 for the experiments here. A graph of this transform is shown in figure 4.3. This technique is based on work by Santamaría *et al.* [104]. The final vector used for the efficient movement task is then $(f(d), \theta, v_l, v_r)^T$ where $f(d)$ is defined by equation (4.1).

The state vectors for each of the three robotic tasks are given in section 6.3 on page 77 for the efficient movement task, section 7.3 on page 96 for the "turn ball" task, and section 8.3 on page 113 for the "off wall" task.

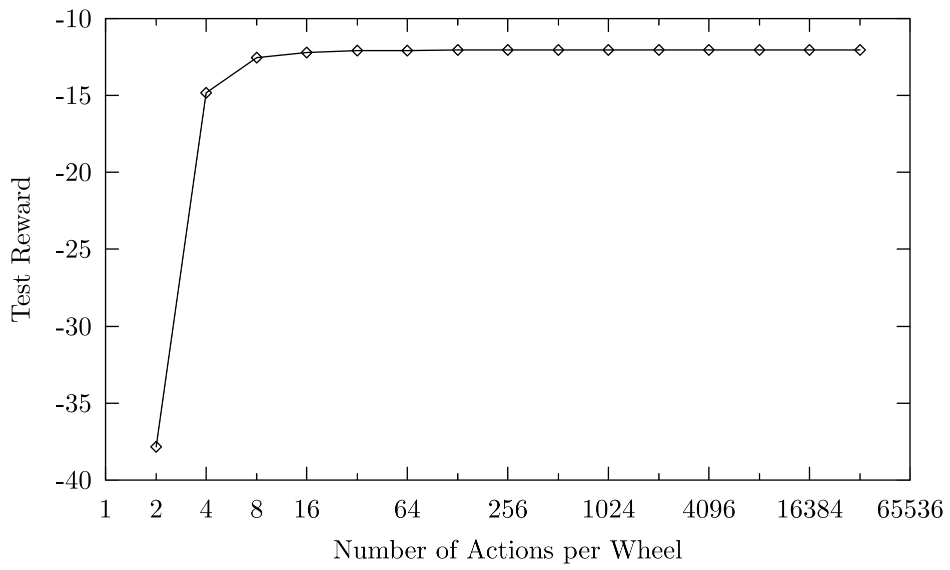

Figure 4.4: Effect of adjusting number of actions on the test performance of *cmu-move-to* under simulation. Note that the X axis is the number of actions per wheel, so the total number of possible actions is the square of this number. The test reward shown here is further explained in section 6.9 on page 81, but roughly speaking it corresponds to the negative of the average number of steps to reach the target, so a higher value for the test reward is better. Also note that the X axis uses a logarithmic scale.

4.4.2 Defining the Action Set

Part of the formulation of a Markov Decision Process is to define the set of available actions $\mathcal{A}$. In this thesis, all of the tasks examined involve actions that are setting target wheel speeds. As with the definition of the set of states, the set of actions should be as small as possible. However, the experience here is that reducing the set of actions too much may have negative consequences. Figure 4.4 shows the effect of adjusting the number of available target wheel speeds per wheel on the *cmu-move-to* movement behaviour. The value of k used for the *cmu-move-to* was 1 (see algorithm 1 on page 17).

Normally, *cmu-move-to* outputs continuous valued wheel velocities. Here, the values have been discretised by finding the nearest discrete wheel velocity. The discrete wheel velocities are evenly spread over the range $[-1, 1]$ metres per second. As the graph in figure 4.4 shows, performance with 2 or 4 actions per wheel is poor but increases quickly as the number of actions increases. After 8 actions per wheel increasing the number of actions does not significantly improve test performance. These results demonstrate that the number of actions should not be too small. However, it also illustrates that there may not be much advantage in allowing many actions. For the efficient movement task, 5 actions per wheel were used as this seemed to be a reasonable compromise between having a small number of actions while still maintaining reasonable performance for behaviours such as *cmu-move-to*.

Actions for the three tasks are discretised differently depending on the task. The set of actions for each of the three robotic tasks is described in section 6.4 on page 77 for the efficient movement task, section 7.4 on page 97 for the "turn ball" task, and section 8.4 on page 115 for the "off wall" task.

4.4.3 Representing Action-Values

How the action-values, or the values of each state-action pair, are represented internally is a key factor in the performance of MDP control algorithms. The most basic approach to representing action-values is to use a table with one element per state-action pair. However, if the state is continuous, it is not possible to enumerate all state-action pairs in this way. There are two possible approaches to resolve this problem. The first is to use a finite set of representative states, and to map the actual state onto this finite set in some way. This is then combined with reinforcement algorithms that are based on a tabular coding of the data. This first approach is referred to here as *discretisation* and is a generalisation of the bit-vector approach used elsewhere [11, 58, 88]. The second is to use a generalised function approximator that maps the continuous state-action pair directly to its corresponding value.

The bit-vector approach assigns a certain number of bits per sensor reading. For example, each sonar device might be assigned two bits, one each for near and far objects. Sonar devices do not simultaneously detect near and far objects and so one of the four possible combinations is never used. Given that an array element is required to store the action-value for each possible state-action pair, this coding can lead to a substantial number of unused array elements. Therefore, in this work, a slightly different approach is taken to determining the index into the action-value array, to avoid wasting array elements. The main reason for reducing the size of the array is to reduce the space required by the learning algorithm. The amount of time required to update the eligibility trace may also be reduced however this will not be the case if a sparse coding of the eligibility trace is used.

The aim of discretisation is to map a real-valued state-vector onto the set of integers in the range $[0, m-1]$ where m is the number of states. The integer then corresponds to an encoded form of the state signal s. The approach used in this work is to begin by defining vectors for the minimums $\vec{a} = (a(1), a(2), \ldots, a(n))^T$, the maximums $\vec{b} = (b(1), b(2), \ldots, b(n))^T$, and the number of intervals $\vec{c} = (c(1), c(2), \ldots, c(n))^T$, where $a(i)$ is the minimum value encoded for $x(i)$, $b(i)$ is the maximum, and $c(i)$ is the number of intervals (or the interval $\underline{\text{c}}$ount) in the discretisation. Note that the number of states encoded is the product of the number of intervals in each dimension, or,

$$m = \prod_{i=1}^{n} c(i). \tag{4.2}$$

To encode the vector $\vec{x}$, the first step is to find the nearest interval for each element $x(i)$. This intermediate result is denoted $p(i)$, with the complete vector of intermediate results being $\vec{p} = (p(1), p(2), \ldots, p(n))^T$. The vector $\vec{p}$ is then mapped uniquely to the state s,

$$p(i) = \left\lfloor \frac{x(i) - a(i)}{b(i) - a(i)} (c(i) - 1) + \frac{1}{2} \right\rfloor, \tag{4.3}$$

$$s = p(1) + \sum_{j=2}^{n} \left[p(j) \prod_{i=0}^{j-1} c(i) \right], \tag{4.4}$$

assuming the number of dimensions $n \geq 2$. The mapping from $\vec{p}$ to s is one-to-one and is straightforward to reverse using remainder arithmetic, however only an approximate reversal can be performed to go from $\vec{p}$ to the original state vector $\vec{x}$. A simple example of this technique is illustrated in figure 4.5 on the next page. In this simple example, the state-vector consists of two dimensions. Regions inside the boxes are mapped to s according to the box number. It is an assumption of this process that the coding covers all possible values of the state-vector.

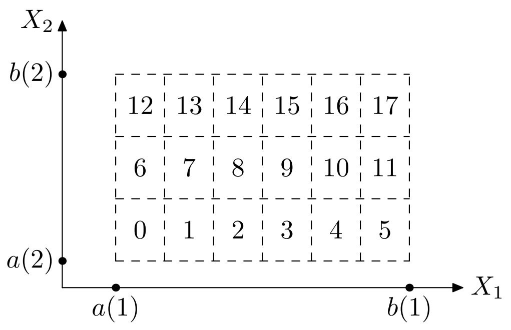

Figure 4.5: Example discretisation of a two-dimensional state space. The minima and maxima of the ith dimension are $a(i)$ and $b(i)$. The numbered boxes are the regions that correspond to each possible "state", or in other words, each possible value of s. The number of intervals are $\vec{c} = (6,3)^T$.

Of the experiments performed here, only the efficient movement task made use of discretisation. As noted in the previous section, the state vector corresponds to $\vec{x} = (f(d), \theta, v_l, v_r)^T$. The minimum, maximum and count vectors were $\vec{a} = (0, -\pi, -2, -2)^T$, $\vec{b} = (1, \pi, 2, 2)^T$, and $\vec{c} = (21, 40, 21, 21)^T$. In this case, the number of encoded states is $m = 370\,440$. To give an example, if the robot's current state is 1 metre distant from the target, if the target is at an angle of 90 degrees, and the left and right wheels are moving at 0.5 and -0.5 metres per second, respectively, then the vector $\vec{x}$ is $(f(1), \frac{\pi}{2}, 0.5, -0.5)^T$. Referring back to equation (4.1), the transformed distance is $f(1) \approx 0.632$. Applying equation (4.3) for each element gives $\vec{p} = (13, 29, 13, 8)^T$, and then applying equation (4.4) gives $s = 80\,994$. This is then used in conjunction with the action index to look up a table of action-values. Note that the two dimensional look-up table is stored using 4-byte floats and is thus $370\,440 \times 25 \times 4$ or about 37 megabytes in size.

The tile coding algorithm uses a different mechanism for converting the state vector into an action-value, and this was described in detail in section 2.4.5 on page 25. The parameters chosen to set up the tile coding are described for each of the three robotic tasks in section 6.6 on page 79 for the efficient movement task, section 7.6 on page 98 for the "turn ball" task, and section 8.6 on page 117 for the "off wall" task.

4.4.4 Reward Function

A reward function is a statement of the desired goal of the behaviour. It specifies what the intermediate reward will be for using a particular action in a particular state. For many problems the reward will be some constant, negative value for all states except the goal state. In order to maximise the expected reward for this sort of problem, an optimal policy needs to get to the goal state as quickly as possible. Goal states are terminal states that give positive rewards, or at least, are not excessively costly.

One aspect that was important to consider was what is referred to in this thesis as *illegal states*. As Jakobi pointed out [63], such states are both undesirable and difficult to model precisely. For example, when a robot crashes into a wall it has entered a state that is undesirable. What happens as a result of the crash is difficult to model. There are several ways to deal with this aspect of real world problems. One is to restrict actions to those that stay within the set of desired states. This is not always possible if the cause and effect model of actions is not available. Another is to model the effect of such actions in a simplified way. In the example above, the robot collision might be modelled as if the robot immediately stopped just before touching the wall. A third approach, and the one recommended by Jakobi, is to include illegal states, and to assign a large penalty when they are reached. It is important that the overall reward for reaching the goal state is not less than the penalty for reaching an illegal state.

This approach, of penalising illegal states but also making them terminal, is used for the "turn ball" (section 7.5 on page 97) and "off wall" tasks (section 8.5 on page 115). For the turn ball task, illegal states are where the ball has drifted out of the robot's grasp too soon. This is an artificial restriction on the nature of the solution, and it would also have been reasonably straightforward to simulate what happens when the ball is further away. However, in the case of the "off wall" task, when the robot crashes into the wall, and in particular, when the ball is squashed between the robot and the wall, the simulator, being somewhat of a simplification of the physics involved, sometimes simulated the ball being pushed through the wall. Fortunately, it was not necessary to make this part of the simulation more realistic, just to detect it and consider it an illegal state. To make it illegal, the robot was "stalled", thus terminating the episode.

It is a requirement for Markov Decision Processes to have reward functions that are some function of the state transitioned from s, the action taken a, and the resulting state s' (see section 2.4.1 on page 19). This makes sense, since it is assumed by algorithms based on MDP assumptions that it is possible to converge upon the action-value function of each state-action pair. If the reward function is dependent on factors other than the state or action, then it may not be possible to converge on a single value for the action-value for a state-action pair. For example, when designing the reward function for the turn-ball task, a desirable attribute of the turn is that it is sufficiently tight, so that it executes within the space allowed by the soccer pitch. A factor that measures this is the Euclidean distance to the starting point. Although this information is available to the simulation, it is not coded into the state-vector. Therefore, if it was included into the reward function, this would violate an assumption of the MDP model, and could affect how accurately the action-values can be estimated. Although this factor might be added into the state-vector, another, simpler option was tried, which was to include the velocity of the robot in the one step reward function. Note that over a complete episode, the sum of the velocity at each step is proportional to the distance travelled. The reward function for the turn-ball task is described in more detail in section 7.5 on page 97.

4.5 Simulation Environment

An important characteristic of the simulator used in this work is that it allowed both an interactive mode, with a graphic display of the robot and its environment, and also a batch mode that still performed the same simulation but without displaying it. This met the requirement previously noted in figure 4.1 on page 48 to allow both a fast environment for batch learning and also a mode where the user can evaluate the qualitative aspects of the learnt behaviour.

The simulation environment was altered slightly for each robotic task. For example, there were no simulated obstacles for the first two tasks, and no ball for the efficient motion task. The latter two tasks also required collision detection and response simulation. Common aspects include the simulation of the kinematics of the robot. This aspect of the simulation is explained in more detail in appendix A on page 133. All of the characteristics were simulated in two dimensions, using a plan view.

Jakobi [63] points out that, in the context of evolutionary algorithms, simulators should be designed so that they are minimal representations of the environment. This approach also applies to MDP based algorithms. In addition, Jakobi points out that aspects that indicate failure, or in other words, situations that are undesirable, do not need to be simulated properly. One example of this issue occurred in connection with the simulation of collision response.

Calculating the correct collision response can be complicated and it is usual to make some simplifications when simulating it. A common approach is to estimate the response based on how far the two colliding objects intersect. A more accurate method, and the one used here, is to rewind the simulation to the instant of the collision, and to calculate the response as a transfer in momentum. Unfortunately, this approach treats the objects as though they were perfectly inelastic and does not give accurate estimates when several objects are resting against one another—for example, when the robot is squashing the ball against the wall. As mentioned previously, this situation is difficult to simulate properly and also is not a desirable situation. It was therefore treated as though the robot had stalled, and was thus an illegal terminal state.

Some other characteristics of the simulation environment relate to how partial observability, sensor error and latency are dealt with, and these are taken up in the following three sections.

4.5.1 Partial Observability

The problem of sensors providing incomplete information on the state of the world was reviewed in section 2.4.8. In the robot tasks examined in this thesis, partial observability becomes an important issue with the "off wall" task. In the efficient movement task, it is assumed that there are no obstacles and the odometry is assumed to be sufficiently accurate, or if not, is corrected by some other means. In the "turn ball" task, the task ends when the ball is out of the fingers and so the ball must always be visible. However, in the "off wall" task, the ball can sometimes be behind the robot where it cannot be seen.

This thesis does not make use of the POMDP model for resolving issues of partial observability as this model does not seem to scale to large problems. Instead, simulated training occurs with "perfect" sensors that continue to see the ball when it is behind the robot. When the learnt behaviour is implemented, the world model substitutes for the perfect sensors of the simulator by providing an estimate of the position of the ball when it cannot be seen. This allows both a policy to be learnt on the simulator for situations where the ball would normally be out of sight, and for that policy to be used on the real robot. A potential problem with this approach is that, if the estimate of the state provided by the world model is incorrect, then the behaviour may produce an incorrect response.

4.5.2 Sensor Error

As reviewed in section 2.7 on page 38, many authors advise that proper simulation of sensor error is critical to learning behaviours. It may be surprising then, that sensor error was not simulated in the simulator used here.

The main reason for choosing not to simulate sensor error was that the architecture used incorporates a world model that already filters sensory information. When writing hand-coded behaviours, it makes sense to attempt to improve the quality of the data in the world model as this tends to lead to better performing behaviours. Section 2.4.9 on page 32 reviews some of the approaches that have been used to do this. Similarly, when deriving behaviours through learning that are intended to make use of the same architecture, it seems sensible to rely on the world model and the process that builds it. Therefore, the simulator does not simulate the sensor error, and instead the world model used in the physical robot tries to reduce the error in the data supplied to the behaviour.

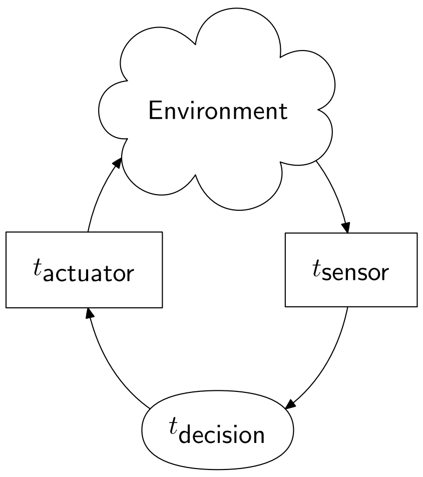

Figure 4.6: Latency in the interaction between the robot and the environment. The combined latency is the sum of the latencies, or $t_{\text{sensor}} + t_{\text{decision}} + t_{\text{actuator}}$.

In the experiments here, a predictive filter was used to improve estimates of the state of the ball and the position relative to the wall. Since only a single robot was used, it was not possible to make use of cooperative techniques. Also, during the first experiment, the vision-based localisation mechanism was not used.

4.5.3 Latency

Another significant factor in the experimental configuration is the control latency in the physical robot. Some control latency is inevitable, but experiments on the robot used here demonstrated that the latency was a significant factor in its performance. The latency in this system, or the time delay between an action being taken and the effect of that action showing up, is summarised by figure 4.6. Part of the delay is due to the time that it takes to read sensors. Although sensors usually can be read rapidly, the low-level controller only polls them every so often, and takes additional time to report status back to the main computer. The amount of time taken to make a decision depends on many factors. For example, in a preliminary experiment, a learnt behaviour's tile coding data consumed most of the available memory. This caused more paging to occur, and thus caused a significant increase in the decision latency, thus causing the robot's performance to be worse than expected. Finally, actuator latency corresponds to the time taken to transmit a command to the low level system and for that system to adjust the power to the motors.

Given the importance of latency on the behaviour of the robot, a possible approach would be to learn under a simulated environment that included latency. However, a problem with including latency in the simulator is that it breaks the assumption that the state has the Markov Property. That is, the state transition that is the result of an action, actually corresponds to the action before or even an action performed some time ago. If it were possible to say that the latency corresponded to exactly one decision cycle, then the assumption that the state has the Markov Property could be restored by including the previous action in the state signal.

An alternative approach, and the one used in this work, is to forward predict the state of the robot to attempt to counteract the effect of latency. The reason for taking this approach is that this was the approach originally used to counteract latency in the RMIT-United hand-coded behaviours.

Specific aspects of the simulation environment pertinent to each task are given in section 6.7 on page 80 for the efficient movement task, section 7.7 on page 99 for the "turn ball" task, and section 8.7 on page 118 for the "off wall" task.

4.6 Initialisation using a Behaviour

In this section, a new technique, termed *Policy Initialisation*, is proposed. This technique makes use of an existing behaviour to bootstrap the learning process. To implement this technique, this thesis makes several modifications to the standard reinforcement learning algorithms. To initialise Monte Carlo based algorithms, an initial policy is provided and a modification is made to the greedy policy update to protect this initial policy slightly. To initialise linear Sarsa(λ), four variant approaches are used, three of which perform their work prior to Sarsa(λ) running allowing the standard Sarsa(λ) algorithm to be used. The last variant used to initialise Sarsa(λ) is integrated into the tile coding algorithm used by Sarsa(λ).

The idea of Policy Initialisation originates from two characteristics of reinforcement learning algorithms: First, they typically start with an arbitrary policy, and second, they tend to improve on the initial policy. Sutton and Barto [118, page 56] note that imparting prior knowledge through the reward function can cause side effects. Their example is that a chess program should not be rewarded for taking pieces but for winning the game. They also suggest that prior knowledge might be better imparted through the initial policy or value function. In this thesis, Policy Initialisation is proposed to make use of a hand-coded behaviour to provide prior knowledge. As Sutton and Barto have suggested for this type of approach, variants are tried that either provide an initial policy or an initial value function.

The advantage of Policy Initialisation is simplest to explain in terms of the Policy Iteration algorithm. Policy Iteration works in two phases, the first phase makes the policy greedy with respect to a known transition probability distribution and current estimates of the value of each state. The second phase updates the state values by finding the action that gives the best expected return by considering the possible next state values and the associated one step rewards.

Policy Iteration will always find the optimal policy regardless of the initial values assigned to states, or the initial policy followed (see section 2.4.3 on page 22). Therefore, using a particular policy as a starting point will not make the final policy found less optimal.

The question is whether using an initial policy will speed up the algorithm or cause less policy improvement cycles to be required. Consider a robot moving toward a target position. For a robot that can move equally well in any direction, a simple policy is to move according to the direction of the target, and if there are no other obstacles, this is the optimal policy. If an obstacle is introduced, the policy is no longer optimal for all cases. However, it is still a very good starting point. The simple policy provides reliable values up to the point of the barrier, so only the values from then on need to be adjusted.

The approach suggested here is based on seeding the learning process with an initial policy. This work proposes that, if this initial policy is better performing than the arbitrary policy typically used by the learning algorithm, then this will lead to earlier convergence to the optimal policy.

4.6.1 Initialisation for Tabular Algorithms

In this section, Policy Initialisation modified forms of tabular Monte Carlo ES and tabular Monte Carlo ϵ-soft On Policy algorithms are presented. When modifying these algorithms several factors were found to be important. First, if the only change to the algorithm is to set the initial policy, then there is little advantage to this approach since most of the initial policy disappears within the first few iterations. Second, for the task examined, specifically the efficient movement task (see section 6 on page 75), it was necessary to set an upper limit on the length of the episode to avoid a situation where the algorithm never terminates. These issues are now discussed in more detail.

Setting the initial policy alone is not sufficient to gain immediate benefit from the prior knowledge entailed by the hand-coded behaviour provided as an initial policy. This was the approach used in preliminary experiments for the efficient movement task. The action-values were initialised to zero, but this was optimistic, since episodes in this task always yield a negative return. During the first few episodes, the action-values corresponding to the initial policy were revised down, causing the current policy to be changed. All other actions were tried before the one suggested by the initial policy was considered again.

This work proposes a simple solution to this problem: do not trust action-values for state-action pairs that have not been visited. That is, if the state-action pair has not been visited, the action-value stored for it is never considered better than one that has been visited. This solution is implemented by the last line of the Policy Initialisation modified form of Monte Carlo ES shown in figure 5 on the following page. Note that the modified algorithm still explores by randomly choosing the first action.

The second factor that was found to be critical with respect to using Monte Carlo methods for control, was that episodes must terminate. This aspect of Monte Carlo methods is mentioned by Sutton and Barto [118, page 111]. With the efficient movement task, episodes following arbitrary policies may not terminate. A reason for this is that it is possible for the robot to move in a loop, or not to move at all, and thus never reach the target.

To terminate such episodes, a limit on the maximum number of steps per episode is used. This seems reasonable for the efficient movement task, since it is possible to guess the maximum number of steps taken by the optimal policy given any starting point, and thus ensure that the limit is much greater than this. However, it is conceivable that for some problems this would not be the case.

The Policy Initialisation modified form of Monte Carlo ES is shown as algorithm 5 on the next page. Algorithm 5 is based on Monte Carlo ES (algorithm 2 on page 24) and has been modified to (a) use a behaviour for the initial policy and (b) ignore unvisited state-action pairs when finding the greedy action. A similar modification based on the Monte Carlo ϵ-soft On Policy algorithm (algorithm 3 on page 26) was also evaluated. The ϵ-soft variant differs from Monte Carlo ES in that exploration occurs throughout the episode, rather than just at the start.

The approach of providing an initial policy should trim the search space for Monte Carlo ES in the same way that it would for Policy Iteration. Also, just as changing the initial policy for Policy Iteration does not affect the ability of the algorithm to find the optimal solution, similarly, Monte Carlo methods should not be affected by changing the initial policy. However, it has to be remembered that there is no proof of convergence for Monte Carlo ES or Monte Carlo ϵ-soft. Only selecting policies that include visited actions should not prevent the algorithms from converging on a solution. The reason for this is that both algorithms will eventually explore every state-action pair regardless of this modification, because they randomly select some actions. Also

```
1. (Initialise policy from behaviour.) $\pi(s) \leftarrow \pi_B(s)$ for all $s \in \mathcal{S}$
   Behaviour $B$ is converted to a policy by querying it with each possible state $s$, and storing
   the resulting action in the mapping $\pi$.
2. (Initialise state-action visit count to zero.) $N(s, a) \leftarrow 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
   The state-action visit count keeps track of how many times an action has been tried from
   a particular state. This allows the value of the state to be estimated by an average.
3. Repeat forever:
   (a) select $s, a$ randomly
   (b) $t \leftarrow 0$, $r \leftarrow 0$, $R \leftarrow$empty list
   (c) while $t \leq t_{\text{max}}$ and $s$ is non-terminal
       i. Append $(s, a, r)$ to $R$ if first visit to $s, a$ this episode
       ii. (Generate next state.) $s' \leftarrow \text{simulator}(s, a)$
       iii. (Track total reward so far.) $r \leftarrow r + \mathcal{R}(s, a, s')$
       iv. (Move to next state.) $s \leftarrow s'$
       v. (Assign action based on current policy.) $a \leftarrow \pi(s)$
       vi. (Increment number of steps.) $t \leftarrow t + 1$
   (d) (Store the value of whole episode.) $Q_0 \leftarrow r$
       $Q_0$ is the value of the episode starting from the first, randomly selected state-action
       pair.
   (e) for each $(s, a, r)$ in $R$:
       i. (Update the average value recorded for the state-action pair.)
              $Q(s, a) \leftarrow \frac{Q(s, a)N(s, a) + (Q_0 - r)}{N(s, a) + 1}$
       ii. (Increment the number of visits to this state-action pair.)
              $N(s, a) \leftarrow N(s, a) + 1$
   (f) for each $s$ in $R$:
       i. (Update policy so that it is greedy with respect to the action-value.) Ignore
          $Q(s, a)$ elements for state-action pairs that have not been visited.
              $\pi(s) \leftarrow \arg\max_{a \in W_s} Q(s, a)$
          where $W_s = \{a : N(s, a) > 0\}$, or in other words, the set of actions from state $s$
          that have been tried at least once.
```

Algorithm 5: *mces-policy-init*: Monte Carlo ES using an initial policy π_B

note that, assuming that the algorithms are capable of converging on the optimal (something that seems likely but remains unproven), then this solution will not be biased since the reward function does not need to be altered by this approach.

There remains a possibility that changing the initial policy will affect how close the solution will be to the optimal. Without a proof that the base algorithms converge, it is impossible to prove that this is not the case. However, given that the convergence to the optimal of Monte Carlo ES and ϵ-soft algorithms "seem likely" [117], and given that these algorithms do not place restrictions on the initial policy used, it seems likely that they will also always converge to the optimal solution.

The above extensions to Monte Carlo ES and Monte Carlo ϵ-soft On Policy algorithms are evaluated experimentally in section 6.9.1 on page 82. These experimental results will show that Policy Initialisation modified Monte Carlo algorithms produce a better policy more quickly than if unmodified Monte Carlo algorithms are used.

4.6.2 Initialisation for Function Approximators

In this section, four ways to perform Policy Initialisation for reinforcement learning algorithms that make use of a function approximator are presented (see section 2.4.5 on page 25). Initialisation from a behaviour is more complex when using a function approximator to represent the action-value than when a tabular representation is used. In the previous section, it was possible to encourage the algorithm to use the initial policy until action-values were learnt by not considering unvisited state-action pairs when updating the policy. When a function approximator is used to represent the action-value function, it is no longer clear whether a state-action pair has been visited. The solution, used for the first three variants presented below, is to provide an initial value function instead. These first three do all their work prior to running the standard linear Sarsa(λ) algorithm (see section 2.4.6 on page 27). The only change to the Sarsa(λ) algorithm is that the initial function approximator weights $\vec{\theta}$ are set according to the result of the Policy Initialising algorithm, rather than being arbitrary. The fourth variant involves a change to the tile coding algorithm to generate sample values for tile weights based on the initialising behaviour when the weights are first used.

The question remains how to set the initial parameter vector $\vec{\theta}$ based on an initial policy. The solution presented here is to learn the parameter vector. Learning with tile coding is performed by updating relevant weights to bring their sum closer to a training sample. Relevant weights are ones that contribute to the estimate (see section 2.4.5.1 on page 27). The approach is not specific to tile coding, however, and whatever mechanism is used to learn the parameter vector can also be used during initialisation.

This changes the problem from how to select appropriate values for $\vec{\theta}$, to one of how to select state-action pairs to sample. Several approaches were tried. The first was to use a Monte Carlo ES style algorithm. A second approach modified this by also initialising other actions for visited states to be pessimistic with respect to the initialising policy. The third approach is similar to the first but uses linear Sarsa(λ) as the basis for learning the values associated with a policy. The last approach examined uses lazy initialisation of a tile weight. That is, estimating the value of a state-action pair based on the initial policy is delayed until the value is first needed.

The aim of all approaches is to maximise the benefit gained from the *a priori* knowledge given by the hand-coded behaviour. An experimental evaluation of each of these approaches is given in section 6.9.2 on page 84. Unfortunately, this evaluation will show that, in the context of the efficient movement task, none of the Policy Initialisation approaches for function approximated

```
1. (Zero parameter vector.) $\theta_i \leftarrow 0$ for all $i$
2. Repeat for some number of episodes
   (a) (Randomly select start state-action.) $s \leftarrow$ random, $a \leftarrow$ random
   (b) (Generate episode.) Generate an episode using policy $\pi_B$ storing reward so far for
       each visited state-action.
   (c) For each visited state-action $(s, a)$:
       i. $r \leftarrow$estimated reward based on episode
       ii. (Compute error in function approximation.)
               $\delta \leftarrow r - Q(s, a)$
       iii. (Update $\vec{\theta}$ based on $\delta$ and learning rate $\alpha$.)
```

Algorithm 6: *mces-value-init*: Monte Carlo ES Value Initialisation using an initial policy π_B.

action value functions provide significant performance gains, when the cost of initialisation is included.

4.6.2.1 MCES Value Initialisation

The first approach is referred to as Monte Carlo ES Value Initialisation or *mces-value-init* and is shown as algorithm 6. As with Monte Carlo ES, a random state and action is used as a start point, then the hand-coded behaviour is used subsequently. In the discussion of these algorithms, the estimate produced by the function approximator is referred to as $Q(s, a)$.

Exactly how $\vec{\theta}$ is updated depends on the function approximator. The algorithm was explored using tile coding, and in this case, the tiles that were relevant to $Q(s, a)$ are adjusted by $\alpha\delta/n$ where n is the number of tile layers.

A particular behaviour will only ever take one action from any particular state. Therefore, all other actions for that state will not be assigned a value until both the state and action are selected as the first state-action pair for the episode. With a large number of states and actions, this approach may take a long time to produce values for actions other than the action selected by the initial behaviour.

4.6.2.2 Pessimistic MC Initialisation

For the efficient movement task and the "turn ball" task, episode returns are always negative. Therefore, initialising all values to zero can be seen as optimistic. That is, the estimate of how much reward will be received when starting from that state-action combination is greater than the amount that could possibly be received. Optimistic initial action-values encourage exploration. That is, when a state is first visited, all action-values will be zero, and so an action will be chosen at random. Experience will cause the selected action to have its value reduced. Subsequently, when the state is visited again, the action will be chosen from all states except the one just visited. This cycle continues until all actions have been tried at least once. This can be seen as a sort of breadth-first search.

```
1. (Zero parameter vector.) $\theta_i \leftarrow 0$ for all $i$
2. Repeat for some number of episodes
   (a) (Randomly select start state.) $s \leftarrow$ random
   (b) (Generate episode.) Generate an episode using policy $\pi_B$ storing reward so far for
       each visited state-action.
   (c) For each visited state $s$:
       i. $r \leftarrow$estimated reward based on episode
       ii. For each possible action $a$:
           A. (Compute error in function approximation.)
                  $\delta \leftarrow r - Q(s, a)$
           B. if $a \neq \pi_B(s)$: $\delta \leftarrow \delta - 1$
           C. (Update $\vec{\theta}$ based on $\delta$ and learning rate $\alpha$.)
```

Algorithm 7: *minus-one-init*: Pessimistic MC Initialisation using an initial policy π_B.

Normally this form of natural exploration is beneficial. However, to make the best use of the information provided by a working behaviour, actions suggested by the behaviour should not be discarded from the policy until a demonstrably better action is found. Given this, the MCES Value Initialisation algorithm was modified to initialise actions not suggested by the initial behaviour to a value more negative than the action suggested by the behaviour. This algorithm is denoted *minus-one-init* and is shown as algorithm 7.

Algorithm 7 does not select the initial action at random, but instead updates all actions from a particular state. The update depends on whether the action matches that produced by the behaviour for that state. If it does, the update is straightforward, and based on the amount of reward received by following the policy. Otherwise, the update is one less. The basic idea is to make the greedy policy initially match the policy of the behaviour, and also to establish reasonable estimates for the value of the state-action pair. Note that, unlike *mces-value-init*, this algorithm does not explore in any sense. This is acceptable since it is only intended to be used to find the action-values associated with a particular policy, not to find the optimal policy.

4.6.2.3 Sarsa Value Initialisation

The previous two approaches attempt to initialise the tile coding weights for use by linear Sarsa(λ) using a Monte Carlo view of what those weights should be. One immediate difficulty with this approach is that it ignores the effect of λ. The λ constant controls the decay of eligibility in the eligibility trace. Monte Carlo methods do not include this decay and are equivalent to $\lambda = 1$. Therefore, to correctly estimate the action-values associated with a policy in an environment with $\lambda < 1$, one approach is to use a Sarsa(λ)-like algorithm to perform initialisation, modified to apply a fixed policy. This approach is denoted as *sarsa-init* and is presented here as algorithm 8 on the next page.

The *sarsa-init* algorithm fixes the policy by choosing it according to the initialising behaviour rather than greedily selecting it on the basis of the current estimate of action-values. Note

```
1. Initialise $\vec{\theta}$ arbitrarily
2. Repeat for some number of trials:
    (a) Initialise $\vec{e} \leftarrow \vec{0}$
    (b) $s \leftarrow$ select randomly $\in \mathcal{S}$
    (c) $a \leftarrow$ select randomly $\in \mathcal{A}$
    (d) While $s$ is not terminal:
        i. $\vec{e} \leftarrow \gamma\lambda\vec{e}$
        ii. For all $i \in \mathcal{F}_{sa}$:
            A. $e(i) \leftarrow 1$ (replacing traces)
        iii. Take action $a$, observe reward $r$ and next state $s'$
        iv. $\delta \leftarrow r - Q(s, a)$
        v. Get next action from behaviour B: $a' \leftarrow \pi_B(s')$
        vi. With probability $\epsilon$: $a' \leftarrow$ a random action $\in \mathcal{A}$
        vii. If $s'$ is not terminal:
            A. $\delta \leftarrow \delta + \gamma Q_{s'}(a')$
        viii. $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
        ix. $a \leftarrow a'$, $s \leftarrow s'$
```

Algorithm 8: *sarsa-init*: Linear, gradient descent Sarsa(λ) with binary features and replacing traces used for value initialisation.

that this policy is softened based on the exploration parameter ϵ. That is, the policy has a non-zero probability that any particular action will be taken in any state. As opposed to the standard linear Sarsa(λ) algorithm, the first action is chosen at random. Given that the policy is fixed, most updates will occur for actions corresponding to the actions chosen by the initialising behaviour. Therefore, selecting the first action at random means that more updates will occur for state-action pairs that do not correspond to the initialising behaviour.

In this scheme and also the previous two, the algorithms execute prior to running the normal learning algorithm. The following approach, however, is a direct modification to the standard linear Sarsa(λ) algorithm that performs initialisation on demand.

4.6.2.4 Lazy Initialisation

A difficulty faced by the initialisation schemes suggested so far is that if there are a large number of weights, it will take some time for all weights to become initialised. In addition, there is no clear point at which to stop initialising, since the initial states and actions are selected at random, and therefore an infinite number of episodes would be required to guarantee that all weights are initialised. Another issue is that some weights may not need to be initialised to successfully learn the task. For example, the weights may correspond to states that are not possible.

An alternative to attempting to initialise all weights prior to using Sarsa(λ) is to flag all weights as uninitialised in some way, and to estimate their value lazily; that is, estimate their value when they are first accessed. This thesis uses the term Lazy Initialisation for this approach.

Lazy initialisation is implemented by first initialising all of the tile weights to a sentinel value. When an action-value is queried, if any of the associated tile weights contain the sentinel value, a value for the state-action pair is estimated using the initialising behaviour. The estimation is similar to that of the Monte Carlo ES algorithm—generate an episode beginning with the state-action pair, follow the initialising behaviour from then on and find the discounted total reward for the episode. This estimate is then used to update the uninitialised weights. Updating the uninitialised weights occurs in a slightly unusual fashion. Given an observed error δ, and k uninitialised tile weights in the set of relevant tiles $\mathcal{F}_{sa}$, each uninitialised tile weight is set to δ/k. This update only occurs for uninitialised weights and therefore only occurs a single time for any particular weight. This update changes the weights to set the estimated action value to be exactly that of the observed action-value. It seems reasonable to perform this type of update as it only occurs once, and is essentially a form of initialisation.

4.6.3 Related Work

The Policy Initialisation approach developed here is in contrast to Matarić's shaped reinforcement approach [94, 93], and is more similar to Millán's TESEO [97]. Matarić argues for shaping the reinforcement by using heterogeneous reward functions that include progress estimators. This accelerates learning but at the potential cost of biasing the learnt solution. Millán, on the other hand, attempts to bolster the reinforcement learning approach with basic "reflexes", to make them safer during the initial learning phase. The reasoning is that reinforcement learning algorithms act randomly when they encounter states that they have no information for. Millán's approach is most similar to the "lazy initialisation" method developed in this work.

Learning the initial values is similar to the supervised teaching approach suggested by Lin [77]. Lin's approach differs in that the supervision is provided in the form of a single trial or a small set of example trials, whereas here a behaviour is used to perform the supervision and "teaching" trials can be generated automatically. Lin points out that this type of mechanism has a more significant effect on the learning as the complexity of the task increases. In particular, Lin [76] demonstrated an example where a recharge station docking manoeuvre could not be learnt without the assistance of a supervised example trial. If a number of teaching trials were provided, the task could be learnt quite rapidly.

Another approach in this vein is given by Maclin and Shavlik [82], who developed a simple language to allow advice to be passed to the learning agent. An advantage of their scheme is that advice can be passed to the agent during learning. The reinforcement learning algorithm used is connectionist Q-learning [103], and so the action-values function approximator is a neural network. The advice given is integrated into the neural network, essentially increasing the value associated with state-action combinations suggested by the advice. In comparison with the initialisation approaches in this chapter, Maclin and Shavlik's approach does not attempt to find realistic action-values for the advice provided and is quite specific to neural networks as the function approximator. Finding realistic action-values for state-action pairs is important because (a) overly positive estimates will need to be corrected by the reinforcement learning and (b) overly positive estimates for particular actions may unnecessarily delay the investigation of other actions.

4.7 Summary

The learning approach described in this chapter formulates behaviour-based tasks as a set of Markov Decision Process problems. It assumes that a behaviour-based breakdown of tasks has already been performed, but it also influences the way that breakdown should occur. The potential for a large number of highly specialised modules being developed can be avoided by making use of symmetrical characteristics of the behavioural problem.

The learning process is an iterative one, that necessarily allows for tuning of the specification of what is to be learnt. Learning takes place in a simulated environment but final validation occurs in a real one.

The second part of this chapter examined two types of initialisation for the learning algorithm. One for tabular Monte Carlo algorithms that can take a policy as a starting point, and four variants for initialising the action-value function for use with linear Sarsa(λ) and a function approximator. The intuitive basis for this approach is that if a hand-coded behaviour already exists, this should provide useful *a priori* information and hopefully avoid any extensive examination of policies that do not perform at least as well as the hand-coded one. Another potential advantage is that performance improvement over the hand-coded behaviour should occur immediately, rather than having to wait an indefinite number of runs before finding a policy that is better.

Chapter 5

Experimental Overview

The previous chapter developed a process for learning new behaviours based on the Markov Decision Process model and proposed an extension to make use of the *a priori* information that is typically available; that is, a hand-coded behaviour. This chapter gives an overview of the physical experiments used to support the main hypothesis of this work. That is, that behaviours learnt in the manner described in the previous chapter yield better performance not only in simulation, but also with the physical robot.

The experimental evaluation consists of three case studies: the efficient movement task, the task of turning the ball, and the task of kicking the ball off the wall. These tasks were chosen because all three were critical to the performance of the RMIT United RoboCup robot soccer team and because they are tasks for which it is difficult to achieve optimal performance through hand-coded behaviours. The tasks are all self-contained, and their performance can be measured without reference to other behaviours. Also the three tasks have varying levels of complexity, with the efficient movement task being the simplest, the "turn ball" task requiring more sophisticated control, and the "off wall" task having the most complex state space.

5.1 Measuring Performance

Performance is measured by looking at the amount of reward accumulated over an episode. Reward is defined according to a reward function that is specified as part of formulating the problem as a Markov Decision Process. One problem with this measure is that if the reward function is "shaped" to improve the learning rate, then the reward function may not exactly correspond to a true measure of the performance of the behaviour. In this case, it is necessary to look at fundamental properties such as the amount of time taken per episode, or the success rate. To avoid this problem, most reward functions used in this thesis are not shaped. An exception to this is the reward function used for the turn-ball task in chapter 7 on page 93. It is "shaped" by rewarding partial success. In this case, other factors, such as the overall success rate, and the time taken for successful turns need to be examined.

In reinforcement learning, the accumulated reward for an episode starting at a particular state s, while following policy π, corresponds to the value of that state, $V^\pi(s)$. A policy π_1 is at least as good as another, π_2, if

$$V^{\pi_1}(s) \geq V^{\pi_2}(s), \tag{5.1}$$

for all $s \in \mathcal{S}$. When comparing two policies experimentally, it is not usually possible to examine the value obtained from all possible states. Instead, it is necessary to select a set of test states

T where $T \subset \mathcal{S}$. The elements of T are unbiased random samples of $\mathcal{S}$. If $V^{\pi_1}(s) \geq V^{\pi_2}(s)$ for all $s \in T$, then this provides evidence to support the hypothesis that the policy π_1 is at least as good as policy π_2, the amount of support being dependent on the cardinality of T.

Note that this assumes each element of $\mathcal{S}$ is just as likely as any other. This may not necessarily be the case. For example, if a component of the state were the robot's wheel speeds, some wheel speeds would tend to be less likely in real situations. For example, during a robot soccer game, the robot would only rarely reach top speed, but might often be stopped. This factor is ignored here.

A weaker form of comparison between two policies involves looking at the sum of values over a test set. That is, π_1 is better on average than π_2 if,

$$\sum_{s \in T} V^{\pi_1}(s) > \sum_{s \in T} V^{\pi_2}(s). \tag{5.2}$$

This corresponds to saying that policy π_1 may not be better in every respect, but that the average performance can be expected to be better, under the assumption that the set of states seen during an episode starting from a state in T is representative of initial states seen in normal operation.

In a real environment, it is not always possible to produce a particular start state exactly. Therefore, a compromise is to randomly select the start state. In this case, the above equation changes to,

$$\sum_{s \in T_1} V^{\pi_1}(s) > \sum_{s \in T_2} V^{\pi_2}(s), \tag{5.3}$$

where T_1 and T_2 are independent, randomly selected subsets of $\mathcal{S}$. Statistical methods such as a Student t-Test can be used to test the hypothesis that the populations for V^{π_1} and V^{π_2} have different means, given equation (5.3). This does not guarantee that policy π_1 is at least as good as π_2 for all possible states. It does, however, support the idea that the average performance of π_1 will be better than π_2, assuming that actual selection of initial states is uniform over $\mathcal{S}$.

In the physical robot experiments that follow, only the efficient movement task experiments make use of same test set comparisons (5.2), while the other two tasks are based on a comparison of two independent test sets (5.3). For the efficient movement task, the starting conditions include the speed of the wheels and the relative position of the target. These conditions were reproduced by causing the robot to accelerate to the desired velocities for one second and then resetting the coordinate system before starting the behaviour.

For the turn-ball task, the starting conditions include the position and velocity of the ball and the angle with respect to the target. Rather than try to reproduce exact ball starting positions, the ball was rolled slowly towards the robot, and the behaviour started when the ball was within range. That is, close enough for the behaviour to be in context and for it to take control. The robot was always stopped at the start of each episode.

For the off-wall task, the starting position includes the position of the robot with respect to the ball, the distance from the wall to the ball and the speeds of the robot and the ball. Starting positions were generated by positioning the ball near to the wall, and then using a movement behaviour to move the robot to a point next to the ball and about 0.5 metres distant from the wall. After each kick, if the ball was too far away from the wall, the robot waited until the ball was manually moved close to the wall, and then restarted the cycle by moving next to the ball. This approach of moving next to the ball has the advantage that it is selecting test cases from the intended population. Thus the performance estimate will be more realistic than if the test cases were selected arbitrarily.

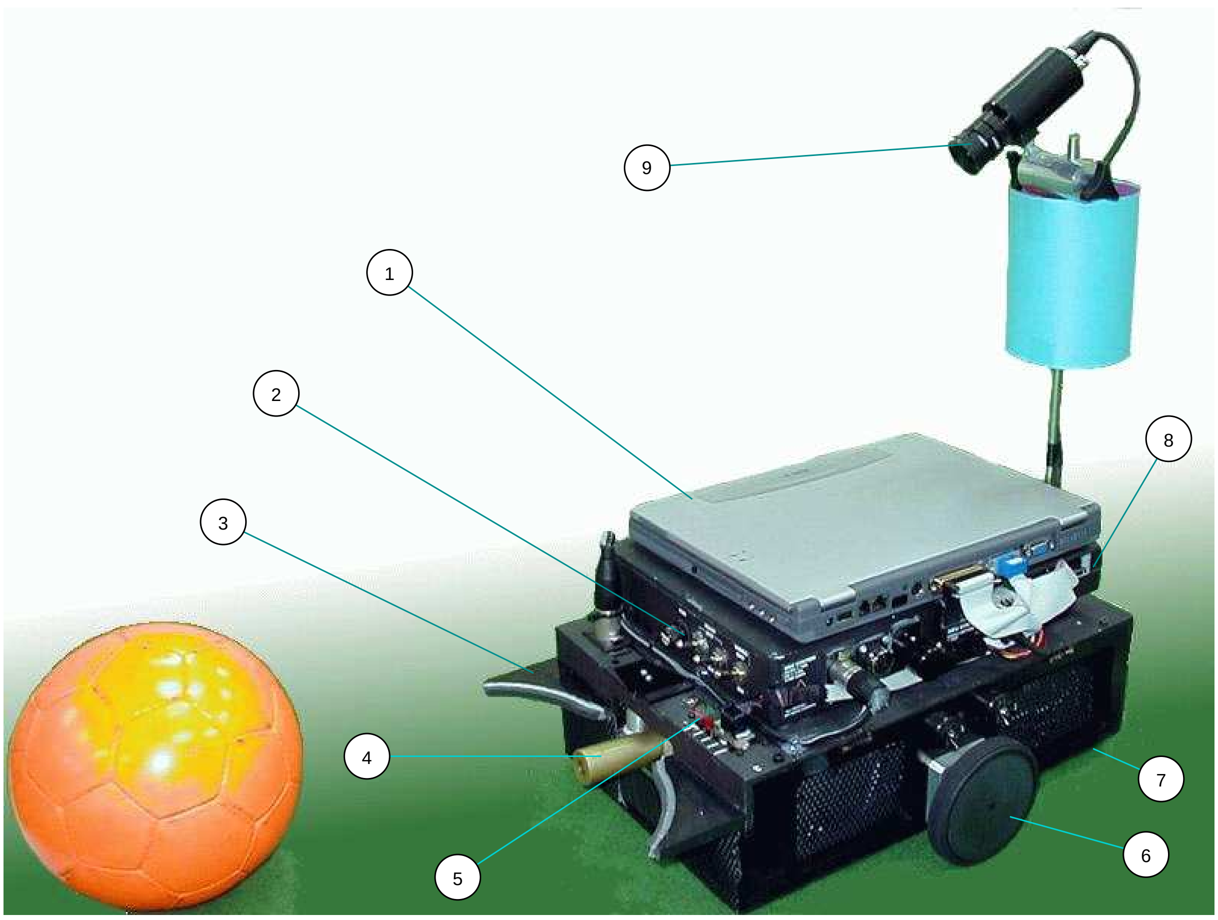

Figure 5.1: Robot used for experiments. Components include: (1) a Pentium II laptop, (2) a battery unit, (3) fingers for manipulating the ball, (4) a kicking device, which is normally retracted, (5) LED status lights, micro-switches, and an RS232 connector, (6) 98 millimetre diameter custom wheels, (7) Teflon skid pads (underneath), (8) vision digital signal processor, and (9) a Pulnix CCD Camera

5.2 Physical Test Environment

The physical robot tests were performed using a robot that had previously been used by RMIT United during the RoboCup 2000 competition. This robot is shown in figure 5.1 and was custom built for RoboCup by staff at RMIT University. The communications interface to the robot is roughly based on the Pioneer 1 robot produced by ActivMedia. The position and velocity of each wheel is detected using shaft encoders attached to the gearbox driving each wheel. This information is integrated to find the relative location of the robot. Localisation based on shaft encoders can be unreliable, because wheels tend to slip, even on high friction surfaces.

For the first two tasks, the encoder information was accurate enough to obtain relative position data during the short manoeuvres attempted here. To test this, the robot was commanded to return to its start position and heading after each manoeuvre. During the tests, the robot returned to its starting point with a small positioning error (usually no more than 10 centimetres) and a medium heading error (typically about 15 degrees).

The flooring of the test environment is carpeted as this is most similar to the robot soccer target environment.

5.3 Experiment Plan

The experiment plan is summarised in figure 5.2 on the facing page. The three tasks, efficient movement, "turn ball", and "off wall" are dealt with in chapters 6 on page 75, 7 on page 93, and 8 on page 113, respectively. For each task, there are some experiments performed in a simulated environment and some in a real environment with a physical robot. For the efficient movement algorithm, two hand-coded (*rmit* and *cmu*) and three learnt (*mces*, *mcsoft* and *sarsa*) behaviours are compared. In addition, for the learnt behaviours, policy initialisation is also examined for the Monte Carlo based learning algorithms and value initialisation for the Sarsa(λ) algorithm. Initialisation mechanisms are only examined in the context of the efficient movement task. In the real environment, the behaviours learnt with initialised Monte Carlo-based algorithms are tested against both a hand-coded behaviour and a behaviour learnt using Sarsa(λ), without initialisation.

The subsequent two tasks only examine Sarsa(λ) as the larger state spaces required by these problems made the use of a discrete coding difficult. For the "turn ball" task, two additional factors are considered. The first is the effect of forward prediction of the state. That is, rather than showing the behaviour the state signal as at the last sensor update, a forward estimate of the state signal is performed and passed to the behaviour. Forward estimate times of 0.1 second (predict 0.1s) and 0.2 seconds (predict 0.2s) are tested in the real environment, alongside no forward estimation (no prediction). Another issue that is examined in the context of the "turn ball" task is the effect of continued learning versus a frozen or fixed behaviour. In other experiments shown here, the behaviour has been frozen and left unchanged during test runs on the real robot. A set of additional experiments are examined in the context of "turn ball" that continue to use the Sarsa(λ) algorithm during runs on the real robot.

The final task, "off wall", is examined in terms of the relative performance of the hand-coded (*rmit*) and learnt (*sarsa*) behaviours.

5.4 Related Work

Wyatt *et al.* [130] provide some guidelines for the design of robotic experiments when testing reinforcement learning. Specifically, they recommend using both external and internal evaluation. The main difficulty that they try to address by this is that a poorly designed reward function may lead to poor behaviours that, nevertheless, give apparently good reward results. To avoid this problem in this work, qualitative aspects are included in the final evaluation of the behaviours.

5.5 Summary

This chapter introduced the common characteristics of the experimental environment, and the basis for distinguishing behavioural performance. The main measure of behavioural performance is quantitative and based on the reward function. However, it is also necessary to consider qualitative aspects, such as whether or not the behaviour was the one desired. This additional

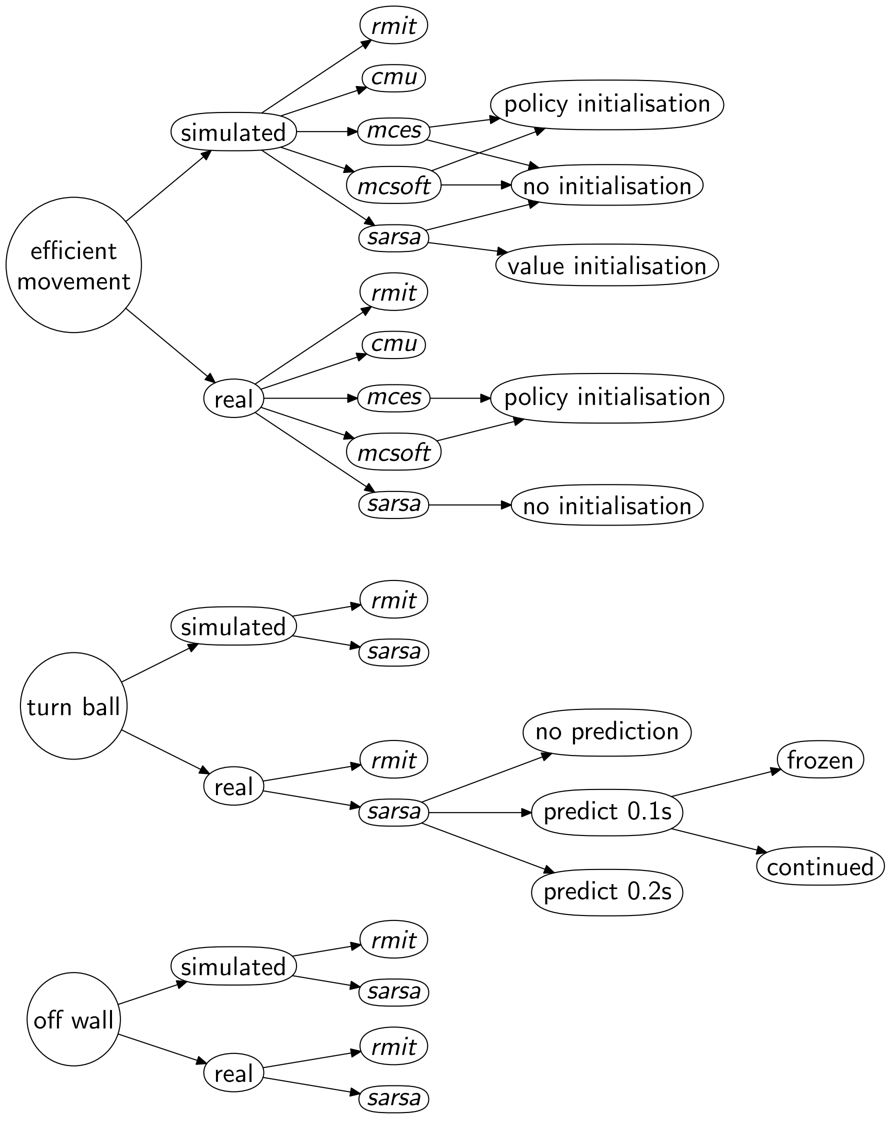

Figure 5.2: Experiment plan. Overall, the experiments are divided into three different behavioural tasks. Each task involves some simulated and some real experiments. These are then broken down into behaviour instances and then variations on the testing of these behaviours. Refer to the text for a description of the abbreviations used in this figure.

consideration avoids the possibility that the generated behaviour is merely one that manages to get a high reward, rather than one that is well suited to the task.

Chapter 6

Efficient Movement Task

This chapter is the first of three that look at experimental evidence in support of the hypothesis that it is possible to produce better performing behaviours using reinforcement learning than with hand-coded behaviours. In this chapter, the problem of efficient autonomous robot movement is posed, and solutions produced using various algorithms. Posing of the problem is broken down into "what the problem is" and "why it might be worth solving". In order to apply machine learning techniques, it is then necessary to define what the robot can perceive, and what actions it can take. The solution is tested using a set of experiments in both real and simulated environments.

The efficient movement problem is a question of how to cause a two-wheeled robot to reach a fixed target within the minimum amount of time. The target can be either an actual object, or it might be an intermediate destination. To keep this problem simple, it is assumed that there are no obstacles that need to be negotiated. However, it is not assumed that the robot is initially stationary, and it may be moving in an arbitrary direction at the start of the episode.

In some robotic problems of this sort, it is necessary for the robot to have a particular velocity and / or heading at the end of the movement. The simplest version is where the final heading and velocity are arbitrary, and this is the version that is used here.

6.1 Motivation

Previous work on efficient movement behaviours for mobile robots was surveyed in section 2.3 on page 17. Of the different techniques examined, the basic movement behaviour from Bowling [22] is both simple and efficient, and will be examined further. This behaviour is referred to here as *cmu-move-to* after the CMUnited RoboCup team that it was used by.

Bowling's approach and other similar smooth motion approaches are more efficient than approaches that involve stopping before turning, especially when handling a moving target, but it is not clear how close to optimal they are and a much more efficient movement behaviour may exist. An optimal behaviour would use the minimum time to reach the target. Optimal performance could be defined in many ways. For example, it might be expressed in terms of minimising energy consumption, or maximising wheel traction. For the purpose of this case study, however, optimal performance is defined in terms of the time taken.

When producing behaviours for the RoboCup 2000 event, the RMIT United team found that the performance of Bowling's movement behaviours were unsatisfactory on their robots. The team tested several other approaches, eventually settling on a simple function relating angle

The *rmit-moveto* behaviour takes as input the current robot position (x, y) and heading ϕ, and calculates the next set of wheel velocity commands to get to the target at (x_p, y_p).

1. (Forward estimate.) From the current position (x, y) and heading ϕ, forward predict robot's position (x', y') and heading ϕ' at time $t + 0.2$ seconds, based on most recent commands (u_l, u_r).
2. (Find relative angle to target.) $\theta \leftarrow \arctan \frac{y_p - y'}{x_p - x'} - \phi'$
3. (Normalise angle.) Add or subtract 2π to/from θ so that it is in the range $[-\pi, \pi]$
4. (Find Euclidean distance to target.) $d \leftarrow \sqrt{(x_p - x')^2 + (y_p - y')^2}$
5. (Determine speed and angular velocity.) Stop and turn left if the angle is large and positive. Stop and turn right if angle is a large negative. Move forward and turn towards target otherwise.
   (a) (Turn left.) If $\theta > \frac{\pi}{4}$, $(v, \omega) \leftarrow (0, \frac{\pi}{2})$
   (b) (Turn right.) If $\theta < -\frac{\pi}{4}$, $(v, \omega) \leftarrow (0, -\frac{\pi}{2})$
   (c) Else $(v, \omega) \leftarrow \left(1, \text{bound}\left[\frac{K\theta}{d}, -\frac{\pi}{2}, \frac{\pi}{2}\right]\right)$

   K is a steering constant, which can be adjusted to control the turn rate, and was set to 2 for all trials.
6. (Convert to wheel velocities.) If S is the distance between the two wheels, the target wheel velocities are:
   (a) $u_l \leftarrow v - \frac{S\omega}{2}$
   (b) $u_r \leftarrow v + \frac{S\omega}{2}$

Algorithm 9: The *rmit-move-to* behaviour.

to the target to the angular velocity. A key addition to this approach was to forward predict the robot's motion and to calculate the relative angle to the target based on this. The forward prediction compensates for the time lag, or latency, between the reception of sensor data and the action determined on the basis of that data taking effect.

6.2 Hand-coded Algorithms

A simplified version of the behaviour used by RMIT-United during RoboCup 2000 is referred to here as *rmit-move-to* and is shown in algorithm 9. The full behaviour also included collision avoidance mechanisms, which have been removed as they are not relevant to this task as specified. Also note that in step 2, the arctan function is implemented using the Java `atan2` function, which correctly calculates the angle for all quadrants. The algorithm for the CMUnited RoboCup small-league team, or *cmu-move-to* was given as algorithm 1 on page 17.

The two behaviours discussed above, *cmu-move-to* and *rmit-move-to*, form a benchmark for what can be done with hand-coded movement behaviours. To proceed with learning a new behaviour to perform this task it is first necessary to define the state, the set of available actions, the simulation approach and the reward function.

6.3 State Representation

Due to the simplicity of the task, it is possible to reduce the robot's state to four variables, as follows,

$$\langle d, \theta, v_l, v_r \rangle.$$

These are: the distance to the target, d, the angle to the target from the forward heading of the robot, θ, and the velocities of the left and right wheels, v_l and v_r respectively. It is not necessary to know the absolute position of the robot, because the position can be sufficiently inferred from the distance and angle to the target. If the task required the robot to end the motion facing a certain direction, then it might be necessary to include the robot's heading relative to this.

For the purpose of this task, a state is a terminal state if and only if the distance to the target is less than 0.3 metres. Allowing for a large margin for error in this way means that the robot is less likely to "fuss" around unnecessarily at the end of the motion, and more likely to shoot through the target area at high speed. This is important if the target is merely a step on the way to some other destination.

As described in section 4.4.1 on page 52, prior to encoding the distance either in tabular or tile coding, it is first transformed so that any distance maps to the range $[0, 1)$.

6.4 Action Representation

Having determined how the state should be represented, it is now necessary to define the set of actions that can be taken.

The robot control system allows the target speed of each wheel to be controlled independently. Independent control of each wheel speed means that the robot can be made to turn on the spot, to go in a straight line, or to turn in an arc. It should be kept in mind that it is not necessarily possible for the control system to produce a desired velocity immediately as the robot may need to accelerate or decelerate. Due to variations in conditions, it is also not possible to keep to a desired velocity exactly even after it has been achieved. Therefore, velocity commands to the robots are in terms of "set points" or target values that are used by the control system to work out how to adjust the power to the wheels. The left and right wheel velocity set points are denoted as u_l and u_r respectively.

Most algorithms for solving Markov Decision Processes (as discussed in chapter 2), including the ones used here, require that it is possible to enumerate all actions available in any particular state. It is therefore necessary to decide on a set of possible velocity set points (or actions) for each wheel. For the purpose of the following experiments, 5 wheel velocity set points were used, evenly distributed over the range $[-2, 2]$ metres per second. This range is based on the limitations of the robot used in the experiments. This gave a total of 25 possible actions over both wheels. Note that the hand-coded behaviours return target wheel velocities directly and so they are not limited to particular values but can use continuous values.

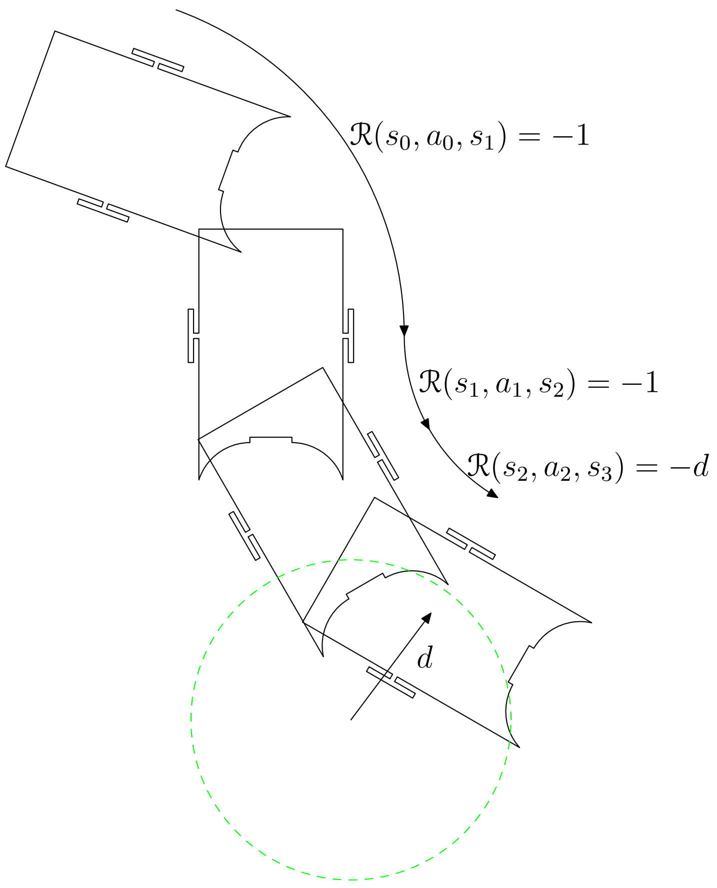
$\mathcal{R}(s_0, a_0, s_1) = -1$
$\mathcal{R}(s_1, a_1, s_2) = -1$
$\mathcal{R}(s_2, a_2, s_3) = -d$
d

Figure 6.1: How the reward function applies to an example episode. The target area is shown as a green dashed circle. The episode consists of three transitions, the final transition receiving a reward $-d$, or the distance in metres to the centre of the target. In this case, the distance on the last step is 0.25 metres, so the last reward is -0.25, and therefore the total reward for the episode is -2.25.

6.5 Reward Function

The reward function (6.1) simply returns -1 for all steps, except for the last step where the reward is the negative of the distance left to go.

$$\mathcal{R}(s, a, s') = \begin{cases} -d & \text{if } s' \text{ is a terminal state} \\ -1 & \text{otherwise} \end{cases} \tag{6.1}$$

where d is the distance to the target in metres. Since the state is terminal, this can be no greater than 0.3. An example of how this reward function works is shown in figure 6.1.

Since the learning algorithm tries to maximise the reward per episode, this reward function encourages behaviours that get to the target as quickly as possible. By including a small penalty based on the final distance to the target, behaviours that more accurately intercept the target are encouraged. It is not necessary for the robot to stay at the target point, so training is episodic, with the episode ending as soon as the target area is reached. The rate at which reward decays, γ, is set to 1, meaning no decay. This has the effect that reward received at the end is just as

Table 6.1: Parameters used to discretise the state and action variables for Monte Carlo methods.

Variable	Minimum	Maximum	Intervals	Range	Resolution
$f(d)$	0	1	21	1.0	0.05
θ	$-\pi$	π	40	2π	$\frac{\pi}{20}$
v_l	-2	2	21	4	0.2
v_r	-2	2	21	4	0.2
u_l	-2	2	5	4	1.0
u_r	-2	2	5	4	1.0

important as reward received at the start. Since it is possible to generate a behaviour that never reaches the target, it is necessary to set an upper limit on the number of steps. For this task 1000 steps are allowed. This limit is chosen because it is well in excess of the number of steps required, even for behaviours that are very inefficient.

Although this reward function is quite easy to understand, and the total reward has a simple correspondence with the time taken to achieve the task, one difficulty with it is that it is not possible to decide, from the accumulated reward, between two policies that do not cause the robot to reach the target.

6.6 Learning a Policy

Three different algorithms were used to learn a policy to solve the efficient movement problem: Monte Carlo ES (see algorithm 2 on page 24), Monte Carlo ϵ-soft On-Policy (see algorithm 3 on page 26), and Sarsa(λ) (see algorithm 4 on page 30). The two Monte Carlo algorithms are different in the way that they explore. The Exploring Starts variant of the Monte Carlo algorithm randomly chooses the first step, whereas the ϵ-soft On-Policy variant chooses an action randomly every so often. The frequency of randomly chosen actions is based on ϵ. For the following experiments, ϵ was set at 0.1. Sarsa(λ) is a variant of the Temporal Difference algorithm, TD(λ), that is suited to control problems. For the following experiments, a value of 0.7 was used for λ. More information about this algorithm can be found in section 2.4.6 on page 27.

An important design decision when implementing these algorithms is how to represent the action-value function estimate. One approach is to assume that the state is discrete, and to represent the action-value using a table, with one entry per state-action combination (see section 4.4.3 on page 55). This is the approach used with the Monte Carlo algorithms. The parameters used to discretise each variable of the state and action are given in table 6.1.

Another approach to representing action-values is to use a tile-coding function approximator (see section 2.4.5 on page 25) with Sarsa(λ) (see algorithm 4 on page 30). The parameters used to define the tile coding are shown in table 6.2 on the next page. Note that in this case, the range for $f(d)$ is limited to $[0.42, 0.76]$, which corresponds to a range for d of $[0.3, 1.0]$. The range was cropped to reduce memory requirements of the tile coding data structure. This cropping was also based on an intuition that the most interesting characteristics of the behaviour occur when the target is within about a metre. Also, the cropping does not prevent the behaviour from being used for larger distances, and it seems likely that performance will not be degraded significantly by this simplification.

These algorithms typically begin by assuming that the action-value function or table has been initialised to arbitrary values. In this work, an alternative is tried. For the Monte Carlo

Table 6.2: Parameters used to define the Sarsa(λ) tile coding for the state and action variables. This tiling was repeated 40 times at randomly assigned offsets.

Variable	Minimum	Maximum	Intervals	Range	Resolution
$f(d)$	0.42	0.76	6	0.34	0.067
θ	$-\pi$	π	6	2π	$\frac{\pi}{3}$
v_l	-2	2	5	4	1.0
v_r	-2	2	5	4	1.0
u_l	-2	2	5	4	1.0
u_r	-2	2	5	4	1.0

algorithms, the initial policy is set to the *cmu-move-to* motion behaviour. This technique is described in section 4.6.1 on page 61. For Sarsa(λ), *cmu-move-to* is also used, but this time to initialise the action-values. Four different variants are tried and these are described in section 4.6.2 on page 63.

6.7 Simulated Environment

The simulated environment for this task is simple. It consists of an infinite space and a single simulated robot. The kinematics of the robot are simplified to allow quick calculation. For simplicity, the two wheels are treated as two independently driven wheels, each sharing half the full weight of the robot. The change in wheel position is calculated as though it travels along a straight line, and then the overall robot movement is worked out by the distance travelled by each wheel. Friction is incorporated into the wheel movement calculation by using a constant friction factor that acts against the direction of movement. This approach is only a rough approximation of the true kinematics of the robot. A detailed description of this part of the simulation is given in appendix A on page 133.

In the simulated environment, a standard set of 200 randomly generated test *scenarios* are used. The term scenario denotes the initial state of the robot and its environment, essentially describing the current wheel velocities and the relative location of the target. The Monte Carlo algorithms were run for 100 million learning episodes, and tested using the test scenarios every million. Sarsa(λ) was run for 10 million learning episodes and tested every 1000 episodes. Note that the testing is independent of the learning agent and the policy is not updated during the testing trials.

6.8 Physical Test Environment

The physical tests were divided into 20 scenarios. As noted previously, each scenario defines an initial velocity for each wheel and a relative target location. At the start of each test, the robot was commanded to achieve the start velocities for the scenario, and given one second to do so. After this, it then repeatedly reads the current robot state, and executes the behaviour until the target point is within 0.3 metres. Each scenario was tried 20 times with each behaviour. The behaviours tested were the hand coded behaviours, *cmu-move-to*, and *rmit-move-to*, and the learnt behaviours *mces*, *mcsoft*, and *sarsa*.

To try to eliminate any effect from battery charge levels, the tests were randomly ordered based on a coin toss. Battery charge did not seem to have any effect, except when the battery

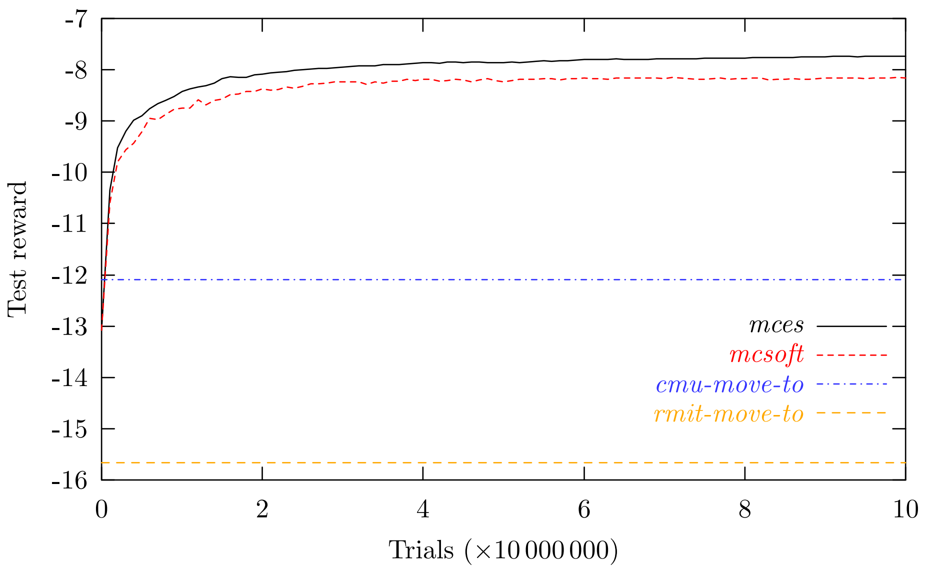

Figure 6.2: Policy performance during learning for Monte Carlo ES (*mces*) and Monte Carlo ϵ-soft On Policy (*mcsoft)* algorithms. Both algorithms were initialised using *cmu-move-to*.

was extremely low. Therefore, the batteries were always recharged prior to them reaching this level.

6.9 Simulator Results

The relative performance during learning of the two Monte Carlo algorithms is shown in figure 6.2. Note that the test reward corresponds to the average reward over a standard set of 200 randomly generated test cases. The initial performance for both algorithms corresponds to the performance for the hand-coded behaviour *cmu-move-to* since that behaviour was used to generate the initial policy. The actual initial performance is slightly less than *cmu-move-to*'s because the learnt behaviours output a discretised set of actions. Improvement stops after about 20 million episodes, but the Monte Carlo ES (*mces*) variant produced a slightly better result. Learning for both algorithms completed in 3 to 4 hours on an Athlon 1GHz PC.

As shown in figure 6.3 on the following page, Sarsa(λ) learns a policy with fewer episodes. However, the overhead per episode is much larger, and the total learning time for 10 million episodes took around 40 hours or about 10 times as long as it took to process 100 million episodes using either Monte Carlo algorithm.

The reason for the downward trend, evident in the graph after about 2 million trials, is not clear, but possibly indicates some divergent effect in the underlying function approximator. For the function approximator to converge, updates to the weights (components of $\vec{\theta}$) should get smaller and smaller. However Baird [13] has shown that unless the updates are constrained in a particular way, the function approximator cannot be guaranteed to converge and updates may get larger rather than smaller. More recently, Gordon[50] has shown that linear Sarsa(0) converges to a region implying that although it will not necessarily converge, it also will be limited in how far it diverges. Current convergence results for Sarsa(λ) were covered in more detail in section 2.4.7 on page 29.

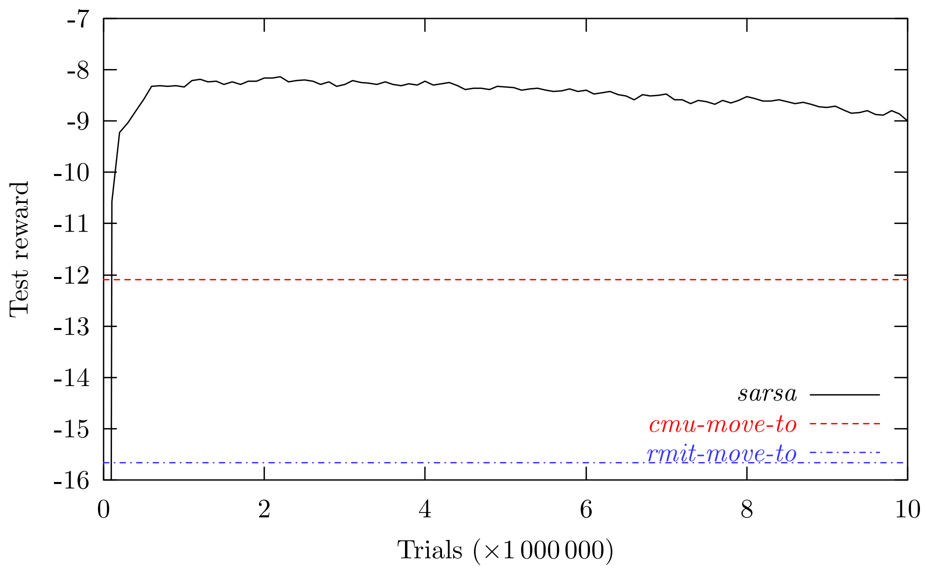

Figure 6.3: Policy performance during learning for Sarsa(λ). The initial test reward, not shown on this graph, was less than -500.

6.9.1 Monte Carlo Policy Initialisation

The results presented for Monte Carlo algorithms so far have made use of the policy initialisation approach presented in section 4.6.1 on page 61. The *cmu-move-to* behaviour was used to produce the initial values in all cases. The question remains about whether or not there is an advantage to using this approach compared with starting with a random policy. An obvious down side to using policy initialisation is that it is necessary to have an initial behaviour to start from. An additional concern might be that providing an initial behaviour would bias the learning algorithm towards behaviours that were similar. For example, if the original behaviour tended towards forward motion rather than backward motion, as *cmu-move-to* does, then it might be expected that this would carry through to the learnt behaviours if they were initialised from it, causing them to get stuck in a local minima. Figure 6.4 on the next page shows the performance of Monte Carlo ES over the first 5 million learning trials with and without policy initialisation. This graph shows how the use of policy initialisation is roughly equivalent to skipping over the first 2 to 4 million trials.

Figure 6.5 on the facing page shows the performance, with and without policy initialisation, for the Monte Carlo ϵ-soft On Policy algorithm over the first 20 million learning trials. During the first 10 million trials, the performance of the version without policy initialisation jumps about, and as with Monte Carlo ES without policy initialisation, performance actually decreases during some stages of the learning. A likely explanation is that in the case where no policy initialisation is used, the action-values used to decide which action to take may be arbitrary.

The results for the two Monte Carlo algorithms imply that there is no long term disadvantage to using policy initialisation. In fact, the learnt behaviours use reverse in many cases where *cmu-move-to* would use forward movement.

The amount of elapsed time required for a certain number of trials was decreased by the use of policy initialisation for the Monte Carlo algorithms. Monte Carlo ES took 3 hours with policy initialisation to perform 100 million trials, whereas it took 5.5 hours to perform the same number

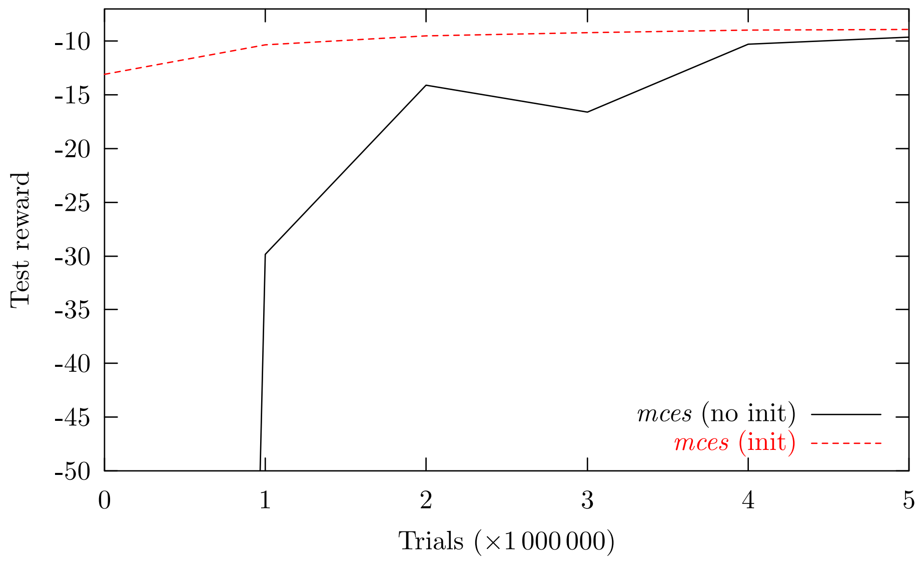

Figure 6.4: Policy performance during learning Monte Carlo ES (*mces*) with (init) and without (no init) policy initialisation. Note that the first test reward *mces* (no init) was less than -600 and thus off the scale used here. Also note that sampling is only performed every million episodes.

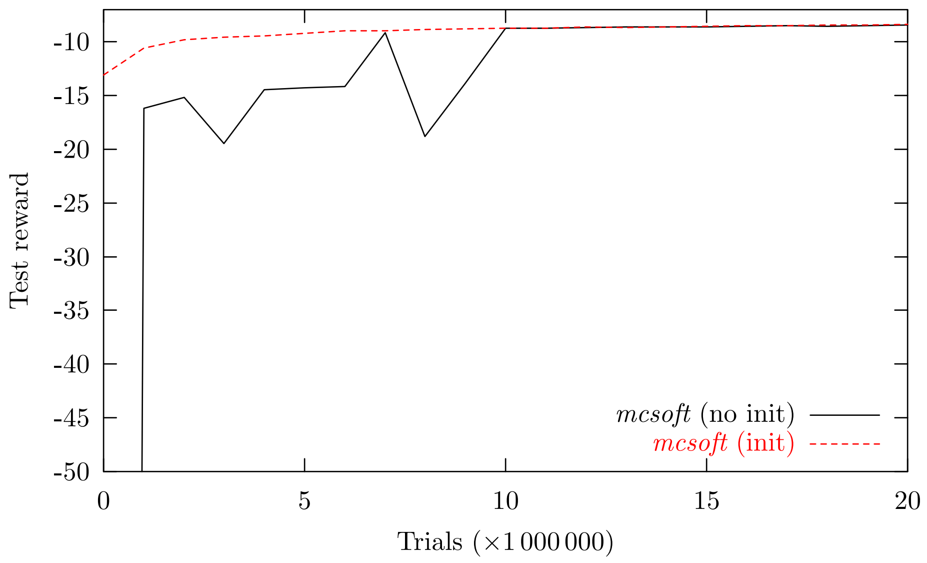

Figure 6.5: Policy performance during learning Monte Carlo ϵ-soft On Policy (*mcsoft*) with (init) and without (no init) policy initialisation. Note that the first test reward *mcsoft* (no init) was less than -600 and thus off the scale used here. Also note that sampling is only performed every million episodes.

Table 6.3: Elapsed times with and without policy initialisation for Monte Carlo ES (*mces*) and Monte Carlo ϵ-soft On Policy (*mcsoft*).

Algorithm	Trials	Elapsed time (no init)	(minutes) (init)
mces	First 10^6	27	3
mces	First 10^8	335	179
mcsoft	First 10^6	5	3
mcsoft	First 10^8	216	205

of trials without policy initialisation. Most of the additional elapsed time when not using policy initialisation occurred early on. For example, with policy initialisation, the first million trials took 3 minutes, whereas without it, the first million trials took 27 minutes. This difference is due to the large number of steps required per trial when the policy is poor. Interestingly, this problem does not occur with the Monte Carlo ϵ-soft On Policy. Policy initialisation only makes a small difference to the first million trials and the total training time for 100 million trials are relatively unaffected. It seems likely that this difference, for Monte Carlo ϵ-soft compared with Monte Carlo ES, is due to the addition of exploration during each episode, rather than just at the start. Certain policies will cause Monte Carlo ES to loop through the same sequence of states indefinitely, whereas Monte Carlo ϵ-soft will occasionally choose an action randomly and escape the loop.

Elapsed times with and without policy initialisation are summarised in table 6.3. All simulation trials were performed on an Athlon 1GHz PC.

6.9.2 Sarsa Policy Initialisation

Four different approaches to Policy Initialisation for linear Sarsa(λ) were tested using the simulated environment and compared with performing no initialisation. Section 4.6.2 on page 63 describes each of these approaches. The base case of performing no initialisation trials was denoted *no-init*. The most basic form of initialisation tried was to use the Monte Carlo ES algorithm to initialise the tile coding weights. This was denoted *mces-init*. With this, and all other initialisation techniques, *cmu-move-to* was used as the initialisation behaviour. A variant of this is to initialise actions that are not prescribed by the behaviour to a value one less than the value estimated for the prescribed action. This variant is referred to as *minus-one-init*. Monte Carlo approaches estimate the value differently to Sarsa, and so another possibility is to use Sarsa(λ) directly, but using a fixed, but soft, policy corresponding to the behaviour. Using Sarsa for initialisation is denoted *sarsa-init*. A final variant, denoted *lazy-init*, requires no initialisation trials, but instead initialises weights when they are first referred to. The discount factor γ was 1 as this corresponds to the task being learnt. The learning rate α was 0.01 for *mces-init* and *sarsa-init* and 0.2 for *minus-one-init*. Also, *sarsa-init* used an eligibility decay λ of 0.7 and an exploration factor ϵ of 0.1.

To speed up the algorithms for the purpose of this evaluation, a smaller tile coding was used than that defined in table 6.2 on page 80. The tile coding consists of 3 tiles per state variable with 10 tilings. The basis for differentiating the various algorithms is to run 100 000 initialisation episodes, followed by 30 000 Sarsa(λ) trials. After this, the resulting policy is tested using the standard 200 test trials. This process was repeated 10 times for each initialisation approach. Figure 6.6 on the facing page shows an example of the test reward performances

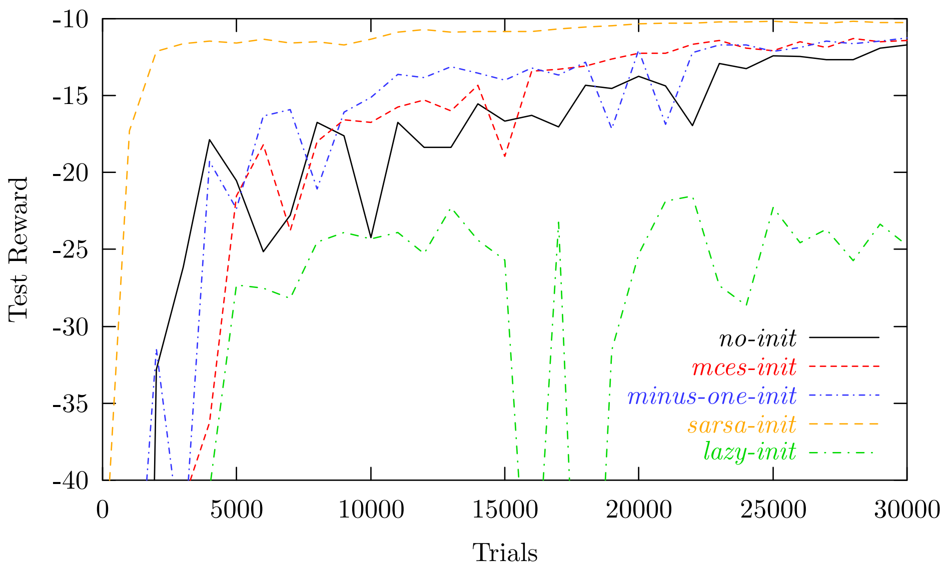

Figure 6.6: Simulation test reward during first 30 000 Sarsa(λ) trials using various initialisation approaches. The graph shows the results for a randomly selected run for each initialisation method.

Table 6.4: Mean simulation test reward after 100 000 initialisation trials followed by 30 000 Sarsa(λ) trials. In the case of *no-init* and *lazy-init*, no initialisation trials were performed, and the test reward is after just 30 000 trials. Ten runs of each variant were used to produce the mean value.

Algorithm	Mean reward	Standard deviation
no-init	-12.6	0.51
mces-init	-11.9	0.45
minus-one-init	-11.5	0.50
sarsa-init	-11.4	2.2
lazy-init	-24.4	7.9

for each approach during the first 30 000 trials. The results of all 10 runs of each approach are summarised in table 6.4, with the statistical significance of the difference in these means examined in table 6.5 on the following page.

The most distinct result is that the lazy initialisation approach clearly is not as good as any other, including using no initialisation at all. Although the standard deviation of this approach is large, the best result is only -15.5, which is worse than the worst result produced by no initialisation. Lazy initialisation is also the most complex approach, requiring that the tile coding algorithm specifically cater for detecting uninitialised values.

Ignoring for the moment the issue that initialisation algorithms have the advantage of an additional 100 000 trials, the best approach appears to be either *minus-one-init* or *mces-init*. Although *sarsa-init* produces the best mean, it has a large standard deviation in the test reward produced after 30 000 trials. From the results obtained, the mean produced through *sarsa-init* cannot be said to be significantly better than no initialisation, whereas the mean test rewards for behaviours initialised with *mces-init* and *minus-one-init* are significantly better than that produced without initialisation. Since *mces-init* produces this improvement in a simpler fashion

Table 6.5: Analysis of t-Test statistic for effect of initialisation after 30 000 trials, assuming unequal variances, and showing the p-value for a one-tail test. P-values less than 0.05 are shown in bold. These correspond to a 95% confidence level that the null hypothesis, that the two populations have the same mean, can be rejected.

	no-init	mces-init	minus-one-init	sarsa-init
mces-init	**0.0018**			
minus-one-init	**0.00012**	0.074		
sarsa-init	0.069	0.28	0.45	
lazy-init	**0.00052**	**0.00035**	**0.00029**	**0.00024**

than *minus-one-init*, it would seem that *mces-init* was the initialisation scheme of choice.

However, a more realistic assessment must include the additional cost of the extra 100 000 trials plus the requirement that a hand-coded behaviour needs to be available to perform this sort of initialisation at all. To assess this, Sarsa without initialisation was continued for an additional 100 000 trials. The mean test reward produced after a total of 130 000 trials, repeated 10 times, was -9.88 with a standard deviation of 0.56. A t-test established that this result was significantly greater than that produced by *mces-init* with a p-value of 4.7×10^{-8}. On the other hand, for the simulation environment used here, the time to process 130 000 trials is around 13 minutes, compared with *mces-init*, which takes around 6 minutes. After 6 minutes, Sarsa without initialisation had performed around 50 000 trials, and had a test reward of about -11.0. This indicates that even when using time as the basis for comparison, *mces-init* performs roughly as well as Sarsa(λ) without initialisation. This indicates that there are no significant benefits to initialising values prior to running Sarsa using a hand-coded behaviour. In addition, there are significant costs, which include the cost of providing a hand-coded behaviour to initialise with.

6.10 Physical System Results

Each different version of the movement behaviour was tested with 20 different, randomly generated scenarios or start conditions. The scenario defines the relative position of the target and the initial speed and angular velocity of the robot. Two of these test cases (scenario numbers 0 and 18) were completely trivial as the robot was already close enough to the target location prior to taking any action. As with the simulated environment, the episode ends when the robot is within 0.3 metres of the target location.

Figure 6.7 on the next page shows the performance of each behaviour during the physical tests. The hand-coded behaviours correspond to the behaviour developed by Bowling [22] for the CMUnited RoboCup team (referred to here as *cmu-move-to*) and the movement behaviour developed by the RMIT United RoboCup team (*rmit-move-to*). Also shown are the results for behaviours learnt using the Monte Carlo ES (*mces*), Monte Carlo ϵ-soft On Policy (*mcsoft*), and SARSA(λ) (*sarsa*) algorithms. The graph shows that the *cmu-move-to* behaviour had much more variability in how it performed even when considering the same start conditions. In comparison, *rmit-move-to* produced better results than *cmu-move-to* with little variability.

Table 6.6 on the facing page shows the average combined time to perform one run of each of the 20 scenarios. The two Monte Carlo approaches gave the best results, with SARSA(λ) close behind. The *rmit-move-to* behaviour performed reasonably well, but was not as good as the learnt behaviours. The *cmu-move-to* behaviour, on the other hand, took around twice as

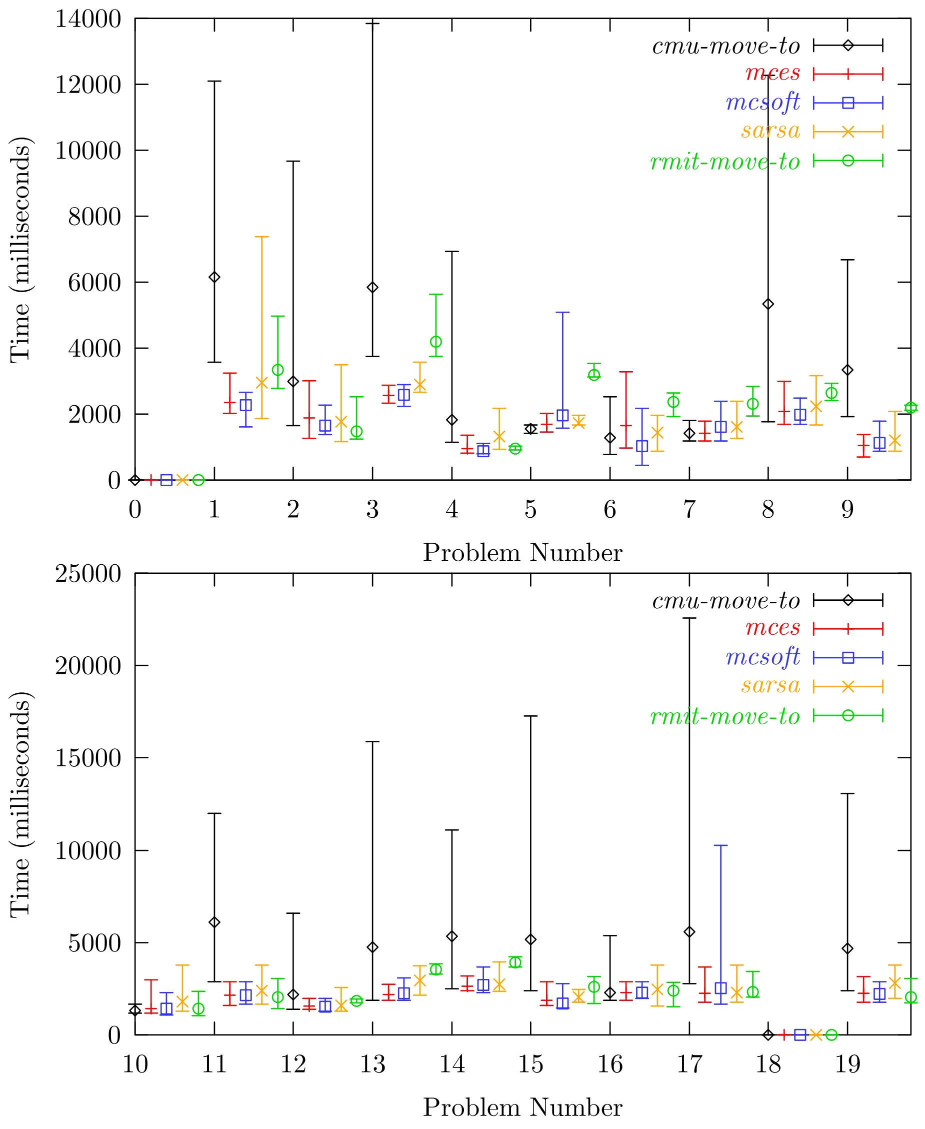

Figure 6.7: Time to complete first 20 test scenarios for five different sorts of behaviours, on the physical robot. The error bars show the minimum, maximum and average time taken by a behaviour.

Table 6.6: Average total time to perform 20 test scenarios on the physical robot.

Behaviour	Total time (seconds)
cmu-move-to	67.15
rmit-move-to	44.80
mces	34.23
mcsoft	33.90
sarsa	38.23

Table 6.7: Analysis of t-Test for behaviour time performance assuming unequal variances, showing the p-value for a one-tail test. P-values less than 0.05 are shown in bold. These correspond to a 95% confidence level that the null hypothesis, that the two populations have the same mean, can be rejected.

	cmu-moveto	rmit-moveto	mces	mcsoft
rmit-moveto	$\mathbf{4.43 \times 10^{-9}}$			
mces	$\mathbf{1.02 \times 10^{-11}}$	$\mathbf{7.95 \times 10^{-25}}$		
mcsoft	$\mathbf{1.44 \times 10^{-12}}$	$\mathbf{8.04 \times 10^{-16}}$	0.62	
sarsa	$\mathbf{3.86 \times 10^{-11}}$	$\mathbf{3.99 \times 10^{-11}}$	$\mathbf{8.48 \times 10^{-7}}$	$\mathbf{7.07 \times 10^{-6}}$

long as the best learnt behaviours to perform the manoeuvres. Interestingly, the ranking of *rmit-move-to* and *cmu-move-to* performance on the robot was the reverse of that predicted by the simulator results (see section 6.11).

Note that the times shown in table 6.6 on the preceding page cannot be exactly compared to the simulator test rewards as the test sets are not the same. Roughly speaking, multiplying these values by $-\frac{1}{2}$ will give an estimate of the average test reward. By this estimate, *mcsoft* produced an average test reward of about -17 on the real robot. This indicates that the simulator gives an underestimate of the amount of time that the real robot takes to move.

To examine the results statistically, a null hypothesis is posed that the performance of two behaviours have the same means. The variances of the different behaviours are generally not equal, and so a t-Test statistic assuming unequal variances is used, rather than a standard t-Test. The statistical analysis results for the performance of each behaviour compared with each other behaviour are shown in table 6.7. These results show a clear difference in the means for all behaviours, with the exception of the two Monte Carlo methods. In other words, the difference in times shown in table 6.6 on the preceding page is statistically significant, with the exception of the difference between *mces* and *mcsoft*.

6.11 Discussion

Analysis of the motion caused by the learnt behaviours compared with the hand-coded ones shows two main reasons for differences in performance: (a) the learnt behaviours change their behaviour according to the current velocity of the robot, and (b) they move backwards when it is more efficient to do so. In addition, the *rmit-move-to* behaviour performed well because it specifically addressed latency that occurred in the physical robot. Since latency was not simulated, the performance of *rmit-move-to* was artificially low on the simulator, just as the performance of *cmu-move-to* was artificially high, leading to their ranking reversing when the behaviours were transferred to the physical robot.

Figure 6.8 on the next page shows how *cmu-move-to* and *mces* handle scenario 1 in simulation. In this scenario, the robot has a high initial forward velocity, but is heading away from the target. The *cmu-move-to* behaviour ignores this and behaves as if the robot were still. It tries to move backwards, turning in a clockwise direction to face the target. The *mces* behaviour turns in a smooth arc towards the target.

The path chosen by *mces* for this scenario is very close to that found using A^* search (shown in figure 6.9 on page 90). The A^* search solution was found using the same simulator but with a 0.2 second decision cycle. This is in comparison to the rest of the experiments, which used

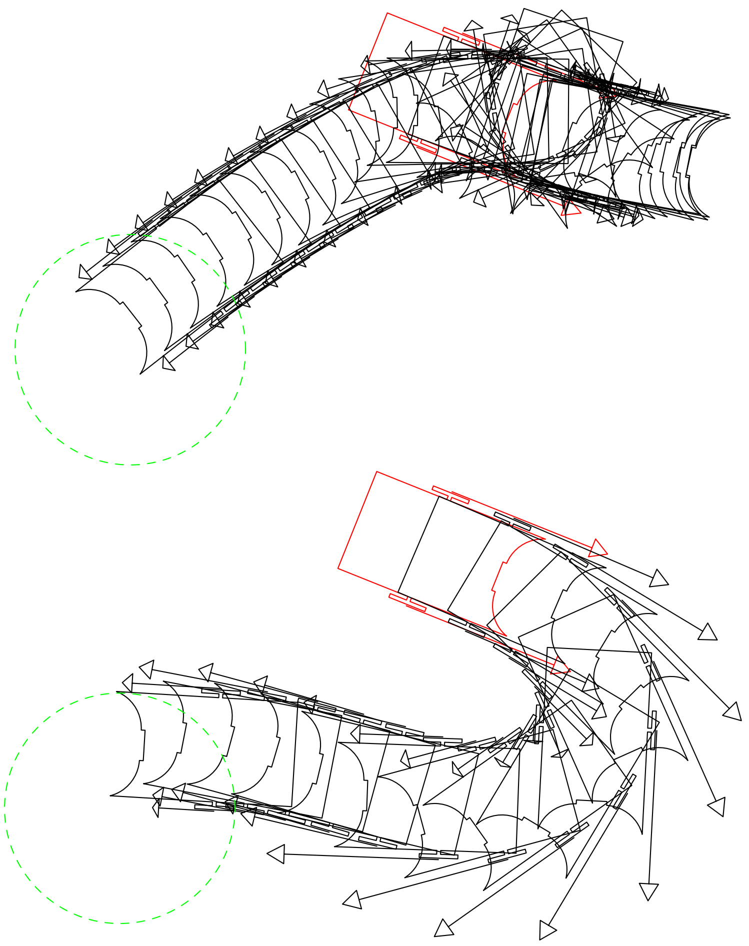

Figure 6.8: Simulator example of scenario 1 using the *cmu-move-to* hand-coded behaviour above and the Monte Carlo ES (*mces*) behaviour below. The starting point is shown in red and the target area is shown as a green circle. The position of the robot and velocity of each wheel is shown for each decision cycle. The wheel velocities are shown as arrows, with a larger arrow indicating a greater velocity.

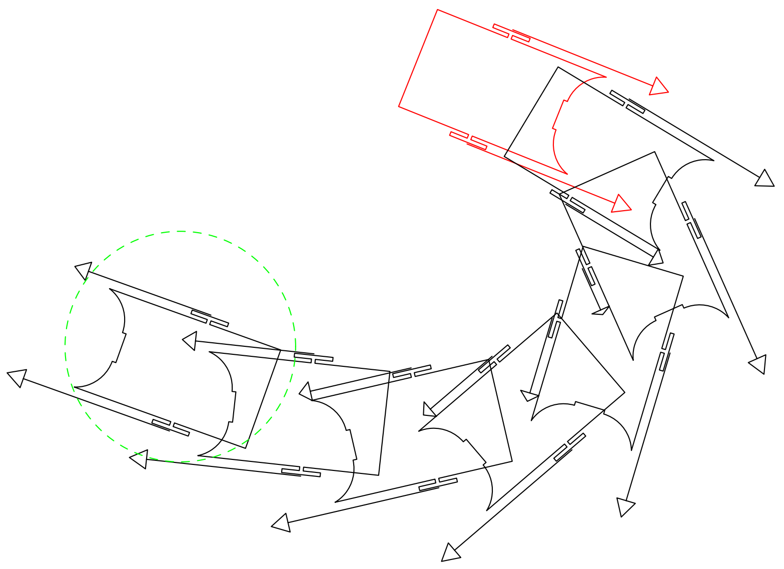

Figure 6.9: Scenario 1 solution found using A^* search. This solution is based on a 0.2 second decision cycle. Other experiments used a 0.1 second cycle.

a 0.1 second cycle. It was necessary to use a longer cycle time to reduce the search space, so that a solution could be found with the memory available. The heuristic used for the search was based on the number of steps taken if the robot was able to instantly face toward the target and head towards it at maximum velocity. Since this will always be equal to or less than the number of steps actually required, the search algorithm is *admissible* [80]. Therefore, the result of the search is optimal for a 0.2 second cycle time. It seems likely that the optimal solution for a cycle time of 0.1 seconds will be similar to that for a 0.2 second one, however it was not possible to confirm this for all examples using A^* due to the increase in computational complexity that this entails. Note that the search result can only be considered optimal for the simulated robot, and is not necessarily optimal for the real one.

When the behaviours are tested on the physical robot, the results are quite different for the hand-coded behaviour. Figure 6.10 on the next page shows a trial of *cmu-move-to* on the robot. In this case, the robot thrashes about on the spot for some time before moving towards the target. One possible reason for this degradation in performance, in comparison with the simulator, is that, on the physical robot, there is a time lag or latency between the robot sensing its position and it effecting an action based on that information.

In comparison, figure 6.11 on the facing page shows that *mces* roughly maintains its performance and follows a similar path when moved from the simulator to the physical robot. The other learnt behaviours, *mcsoft* and *sarsa*, cause the robot to follow roughly the same curved path as *mces*. The learnt behaviour seems to cope with the latency on the physical robot even though this has not been simulated during the learning process. The learnt behaviour benefits from altering its response when the angular velocity is high, and therefore is less susceptible to over-correcting the heading, even when the sensor information is delayed.

The *rmit-move-to* behaviour (figure 6.12 on the next page) has a similar problem to *cmu-move-to* in that it stops and turns. However, it has the advantage that it does not over correct as much, and the path is much more direct.

The *mces* and *mcsoft* behaviours are primed with the *cmu-move-to* behaviour, and it is

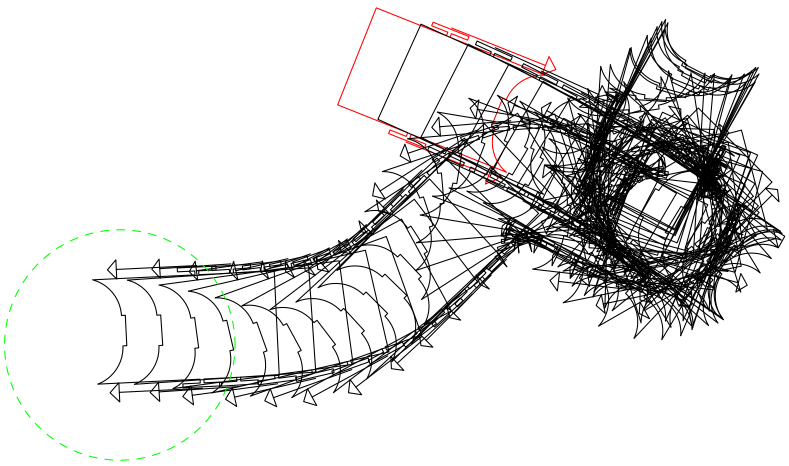
Figure 6.10: Physical robot trial for scenario 1 using *cmu-move-to* behaviour

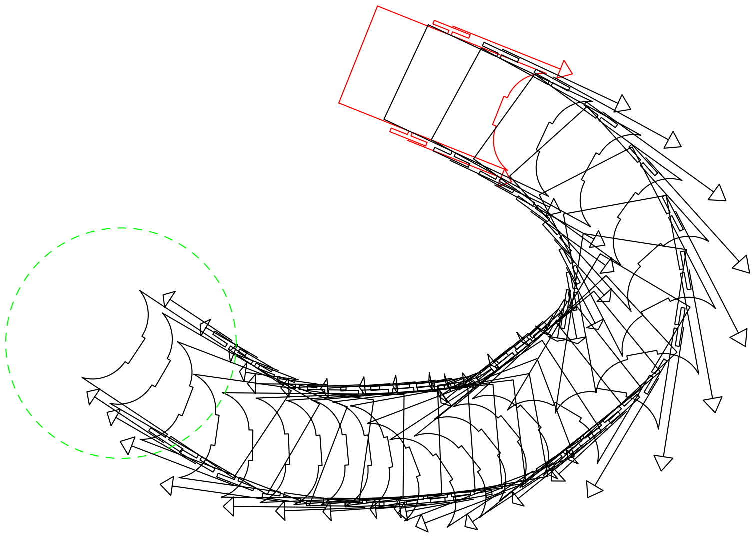
Figure 6.11: Physical robot trial for scenario 1 using *mces* behaviour

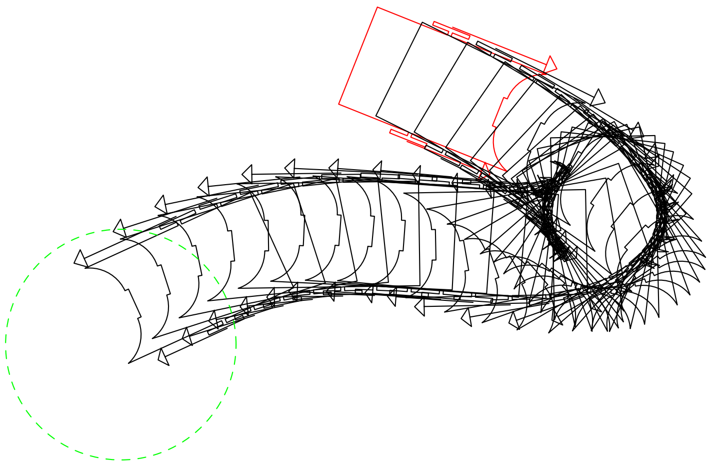
Figure 6.12: Physical robot trial for scenario 1 using *rmit-move-to* behaviour

therefore surprising to discover that although *cmu-move-to* tends to treat reversing situations differently to forward movement situations that this distinction is lost in the learning process. In other words, although the Monte Carlo algorithms were initialised based on *cmu-move-to*, this did not cause them to get stuck in the local minima of moving forwards, when reversing might be more efficient.

6.12 Summary

Latency, or the time delay between sensing and acting, is a significant problem for this robot architecture. As the desired speed of the robot increases, the control latency becomes more and more significant. Other robots of a similar class, such as the Pioneer 1, have similar latencies but travel at slower speeds, and so the latency is not as critical a factor in the overall performance of the robot. A chief cause of latency in this architecture is the narrow bandwidth connection between the robot and the controlling laptop computer.

The poor performance of *cmu-move-to* may be due, in part, to the large control latency, combined with the high movement speeds attempted. In comparison, *rmit-move-to* had better performance, largely through the use of virtual sensors that predicted the future position and speed of the robot, thus counteracting some of the effect of latency.

The learnt behaviours were all better again, despite not using virtual sensors, and without latency being included in the simulator model used to learn the behaviour. Most of the improvement comes from allowing for the current velocity of the robot. Inclusion of this factor is a natural outcome of ensuring that the Markov Property holds for the state. Further improvement of the learnt behaviour may be possible by either including latency in the simulation, or by using forward estimation in the world model.

Of the learnt behaviours, those developed using a Monte Carlo and discrete coding of the state space had slightly better performance than those developed using Sarsa(λ) and tile coding of a continuous state space. The Monte Carlo ones took less elapsed time to learn, but needed more iterations. Since Sarsa(λ) requires fewer simulator iterations, as simulation complexity increases, the advantage of using simpler algorithms, such as Monte Carlo, may decrease.

Various sorts of initialisation techniques were examined in the context of the simulated environment. When using Monte Carlo methods with a tabular representation of the state-action values, initialising the policy from a hand-coded behaviour produced a clear improvement in policy performance until about 5 to 10 million episodes. After this point, performance was comparable, indicating that there was no bias caused by initialising in this way.

When using Sarsa(λ) with a tile coding to represent the action value function, there appeared to be some benefit to using initialisation if the cost of the initialisation process was ignored. The cost, either in terms of trials or in terms of processing time, was significant and, if taken into account, meant that there was actually a cost in using Policy Initialisation with Sarsa(λ).

Chapter 7

Turn Ball Task

In this chapter, a second experimental case study, based on the problem of turning with the ball, is explored. This task is more complicated than the previous one, which looked at efficient movement, because it involves both movement and interaction with another object. As with the previous task, this is not a toy problem, but is based on real requirements for the RMIT United RoboCup team. Also, as was the case in the previous task, the final evaluation of behaviours are performed on a real robot.

A basic aim for a soccer playing robot is to move behind the ball, relative to the direction of play, in order to kick it towards the goal. This can be achieved either by moving around the ball, or by moving the ball. An excellent example of the former approach is provided by Jamzad et. al. [65], who devised a special rotatable wheel configuration that allowed the robot to move up to the ball, then move crab-like, pivoting around the ball. One difficulty with moving around the ball, particularly for robots that cannot perform the manoeuvre while facing the ball, is that the ball may move. Unless the robot has a moving camera or 360 degree vision, it will usually lose sight of the ball while lining up on it. Losing sight of the ball means that movement to a position relative to the ball may be unreliable. Even if the ball does not move while it is out of sight, the success of the manoeuvre becomes dependent on the accuracy of the last sighting, and specifically, the accuracy of the distance estimate, that is more prone to error than the estimate of the relative angle.

One solution is for the robot to turn with the ball. This means that the robot uses its "fingers" to push the ball in an arc. Figure 7.1 on the following page shows an example of how this behaviour is intended to steer the ball so that it is lined up with the goal. According to RoboCup Federation rules, robots cannot be designed to capture the ball completely, and fingers, or other protrusions, cannot wrap around the ball. Any concavity in the shape must not hold more than one third of the ball's diameter (see figure 7.2 on the next page). Therefore, the robot must nudge or guide the ball, while keeping it in its grippers. While it is possible to move the ball in an arc without keeping it between the fingers, this task requires that the ball be close to or between them. Dribbling the ball with a series of sideways kicks would be an option if keeping close possession of the ball were not a requirement. The advantage of close possession of the ball is that it protects it from attack by opponent players. This task is specific to RoboCup but has features that are common to many robot manipulation problems that involve controlling the position of an object without being able to control all of its degrees of freedom.

This chapter follows a similar format to the previous one. It starts with an examination of the motivating factors. The problem is broken up into how states and actions are represented, how the situation was simulated and the correspondence between states and reward. This is

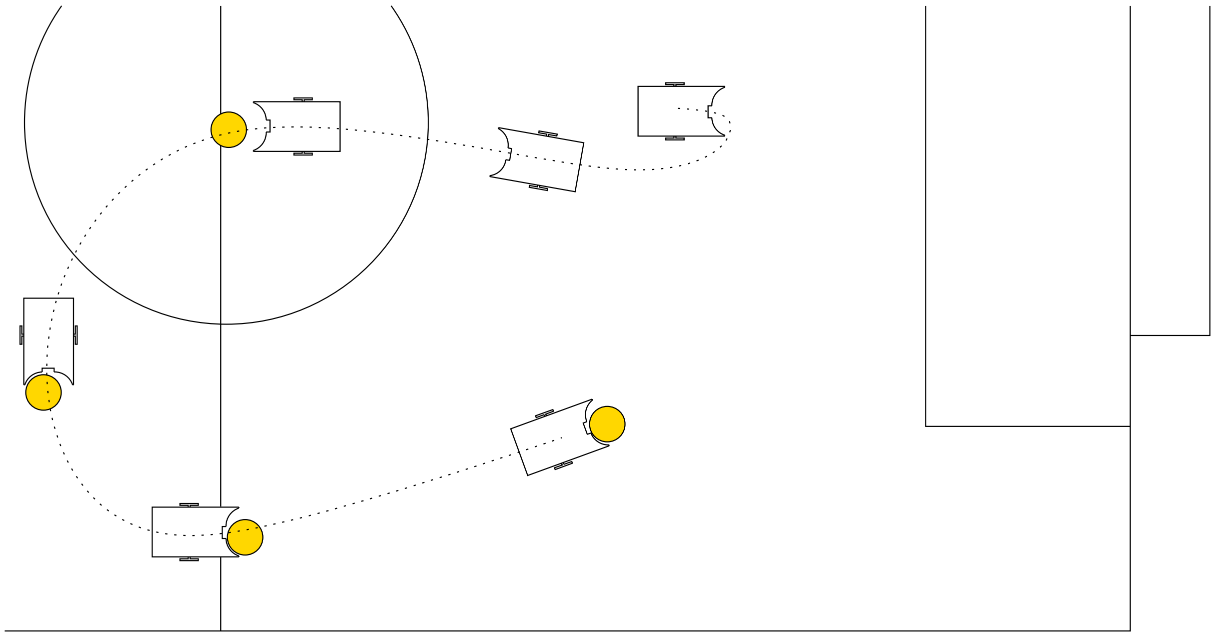

Figure 7.1: The desired behaviour for a robot intercepting and "turning" the ball. Note that this diagram includes interception of the ball, which is a separate behaviour.

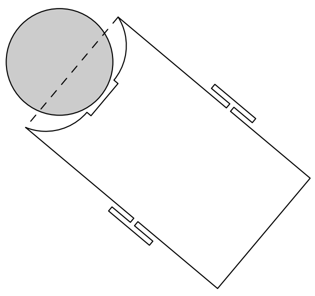

Figure 7.2: The maximum allowable concavity, according to RoboCup 2000 regulations, must not exceed one third of the ball's diameter.

followed by details about how a behaviour was learnt, and the physical test environment used to evaluate the learnt and hand-coded behaviours. Finally, an analysis of the results is presented.

7.1 Motivation

An inspiration for attempting to develop the behaviour with machine learning rather than code it by hand, comes from the experience of attempting to hand-code the behaviour for the RMIT United team for the RoboCup 2000 robot soccer competition. The main problem with the hand-coded behaviour was that it required extensive tuning before it worked reliably. The hand-coded behaviour operated by turning either left or right in a fixed arc while the ball was close and almost straight ahead. When the ball was close but more to the side, the behaviour moved the robot straight ahead, thus attempting to use the fingers to guide the ball more towards the centre of the grasp. When the ball was out of the fingers, another behaviour moved up to the ball, slowing as it got closer to avoid knocking it further away.

In preparation for RoboCup 2000, the RMIT United team spent much time testing and re-tuning this behaviour. This effort was only partially successful. The movement tended to be unresponsive and rigid. This was particularly noticeable when the robot was nearly losing the ball from its grippers.

One possible reason for the lack of success with the hand-coded behaviour is that much more sophisticated logic is required. It is not enough to simply tune the one turning speed that is used under almost all conditions. Instead more factors in the state of the environment need to be taken into account and a greater variety of actions used in response. However, there seems no obvious way for a programmer to guess which states to differentiate and what actions should be tried. In addition, when testing such a sophisticated behaviour, the programmer cannot easily differentiate between those parts that failed and those that were successful. The need to develop sophisticated behaviour, and the difficulty involved in developing it by hand, provide a central motivation for attempting to learn robot behaviours.

Given that it is desirable to learn the behaviour, the next step is to define what will be learnt. The first aspect presented here is the definition of how the state of the environment and robot will be represented.

7.2 Hand-coded Algorithm

The algorithm for the hand-coded behaviour used by RMIT-United during RoboCup 2000 is shown as algorithm 10 on the following page. The *rmit-turnball* behaviour attempts to turn the robot in a small arc while keeping the ball in its fingers. The resulting behaviour can be used to line up on a target. The difference between the angle to the ball and the angle to the target is represented by α. The angle to the ball is θ_b and is derived from the relative position of the ball (x_b, y_b). This behaviour sets the target wheel velocities v_l and v_r, which are calculated from the desired velocity v and angular velocity ω. It also sets acceleration factors for the left and right wheel a_l and a_r. A larger factor is used on the left wheel when turning right, and vice versa. This was found to help smooth the turn.

```
1. If $|\theta_b| > \frac{\pi}{12}$(If the ball is too left or right.)
    (a) (Drive straight ahead slowly.) $v \leftarrow 0.4$, $\omega \leftarrow 0$, $a_l \leftarrow \frac{1}{13}, a_r \leftarrow \frac{1}{13}$
2. (Else if target is on the right.) If $\alpha < 0$
    (a) (Turn right slowly.) $v \leftarrow 0.4, \omega \leftarrow -\frac{\pi}{2}, a_l \leftarrow \frac{1}{9}, a_r \leftarrow \frac{1}{13}$
3. Else
    (a) (Turn left slowly.) $v \leftarrow 0.4, \omega \leftarrow \frac{\pi}{2}, a_l \leftarrow \frac{1}{13}, a_r \leftarrow \frac{1}{9}$
4. Convert speed and angular velocity to wheel velocities.
```

Algorithm 10: The *rmit-turnball* behaviour

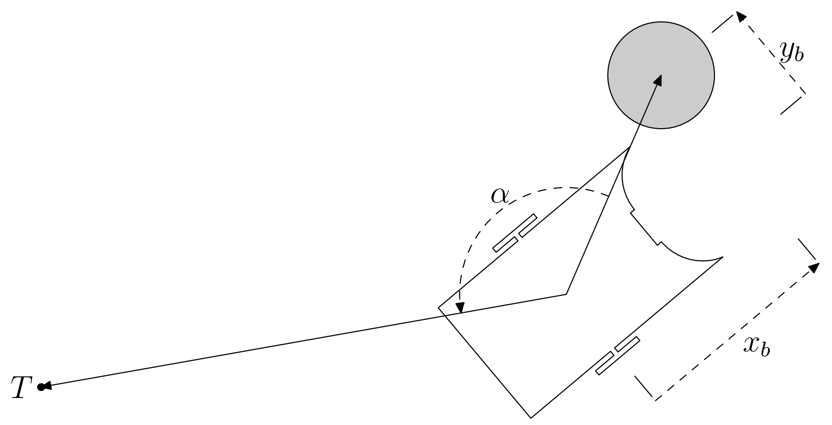

Figure 7.3: Static components of the state representation for the turn ball task. T represents the target. The line from the robot to the ball is at an angle α to the line from the robot to the target T. The relative position of the ball is $(x_b, y_b)^T$.

7.3 State Representation

The state for this task can be represented by the vector

$$\langle x_b, y_b, \dot{x}_b, \dot{y}_b, \alpha, \omega_R, v_R \rangle,$$

where $(x_b, y_b)^T$ represents the position of the ball with respect to the robot, $(\dot{x}_b, \dot{y}_b)^T$ denotes the ball's velocity, α represents the angle to the target, ω_R is the robot's angular velocity and v_R is its speed. These components are shown in figures 7.3 and 7.4 on the facing page.

The coordinate system for the ball's position $(x_b, y_b)^T$ and velocity $(\dot{x}_b, \dot{y}_b)^T$ has the robot at the origin and the X axis pointing in the forward direction of the robot. For this task, the robot's motion was represented by the forward velocity v_R and the angular velocity ω_R. There is a straightforward mapping between these parameters and the left and right wheel velocities [35]. As stated previously, the difference between the angle to the ball and the angle to the target is represented by α.

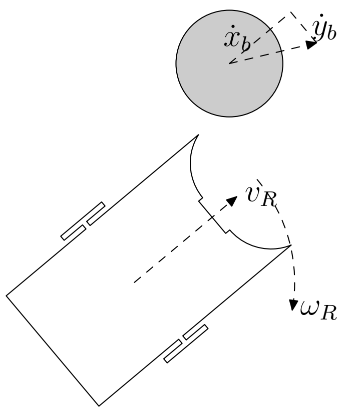

Figure 7.4: Dynamic components of the turn ball task. The ball's motion is represented by the vector $(\dot{x}_b, \dot{y}_b)^T$ with respect to the current heading, whereas the robot's motion is a combination of scalar velocity v_R along the line of the robot's heading and angular velocity ω_R. It is assumed that the ball's spin is negligible.

The dimensionality of the state is much greater than in the previous task, however the amount of computation and memory required to learn the task can be reduced by restricting the range of possible values for some of the dimensions. The behaviour's activation condition provides these limits. This behaviour is only active when the ball is between the grippers, and this allows the range of x_b and y_b to be limited to values where this is the case. Specifically, they can be restricted to $0 \leq x_b \leq 0.4$ metres and $-0.235 \leq y_b \leq 0.235$ metres.

It is not necessary to consider the absolute position of the robot (or its heading) nor the absolute ball position, because it is assumed that no other obstacles or reference points exist. To avoid the location of the target T negating this statement, it is considered to be infinitely distant from the robot. If the target T was close by, the angle to it would be dependent on how far it was away. By making it infinitely far away, it acts like a compass heading and the angle is then independent of where the robot is in absolute terms.

7.4 Action Representation

The set of possible actions available are similar to the efficient movement task; that is, target velocity settings for each wheel. It is possible to reduce the set of actions by eliminating reversing velocities, as it is not possible to keep the ball within the grippers while going backwards. Also, high speeds are unlikely to be useful. The resulting range is $[0, 1]$ metres per second for each wheel. The smaller range allows a finer resolution of 0.1 metres per second without too great an increase in the number of possible actions. In the previous case study, 5 actions were possible for each wheel, giving 25 possible actions overall. For this task, 11 actions are possible for each wheel, or 121 actions overall.

7.5 Reward Function

The reward function (7.2) for the turn-ball task builds on top of the reward function for efficient movement, penalising terminal states that are illegal. For illegal states, the penalty is reduced in proportion to how far around the ball has been turned. A state is considered legal if some part of the ball is between the fingers, or more precisely, if $0 \leq x_b \leq 0.4$ metres and $-0.235 \leq y_b \leq 0.235$ metres. Otherwise the state is considered illegal, and thus terminal. A state is also considered terminal if the absolute target angle is sufficiently small. Specifically, a state is terminal if $|\alpha| < \frac{\pi}{18}$. The reward function can then be stated as

$$\begin{aligned}
\mathcal{R}(s,a,s') &= \mathcal{R}_T(s') + \mathcal{R}_V(s') + \mathcal{R}_P(s') && (7.1)\\
\mathcal{R}_T(s') &= \begin{cases} -1000\,|\alpha| & \text{if } s' \text{ is illegal} \\ 0 & \text{otherwise} \end{cases}, && (7.2)\\
\mathcal{R}_V(s') &= \begin{cases} -v_R & \text{if } s' \text{ is legal and non} - \text{terminal} \\ 0 & \text{otherwise} \end{cases}, && (7.3)\\
\mathcal{R}_P(s') &= \begin{cases} -\frac{|\alpha|}{10\pi} & \text{if } s' \text{ is legal and nonterminal} \\ 0 & \text{otherwise} \end{cases}. && (7.4)
\end{aligned}$$

The first term $\mathcal{R}_T$, or termination reward, penalises policies that allow the ball to roll out of the fingers before reducing the absolute target angle $|\alpha|$ to $\frac{\pi}{18}$ or less. If the angle has been reduced sufficiently, the reward is simply zero. The second term $\mathcal{R}_V$ penalises movement, to encourage policies that make a tight turn. Note that the sum of this reward component over an episode corresponds to the distance travelled that episode. This term was added after finding that without it, the robot would perform fast, but wide turns. The final term $\mathcal{R}_P$ provides a penalty based on how far the robot has turned so far. This term is intended to "shape" the learning process by providing it a hint about which policies are better even when it does not complete the turn. In addition, this term encourages policies that complete the turn quickly.

In forming the final reward function (7.1), a variety of approaches were tried, to encourage learning a policy that turned the ball in minimal space. One approach, that produced reasonable results was to include a penalty for the current speed of the robot in the reward function. By discouraging high speeds, smaller turns were encouraged. Small turns are preferable because the playing area is limited.

The penalty for an illegal state needs to be sufficiently large that it is not considered an attractive shortcut. Illegal states are readily available; the robot merely needs to bounce the ball out of the fingers. Legal terminal states, on the other hand, are more difficult to achieve. Therefore, the penalty for an illegal state is multiplied by 1000.

If the robot partially achieves the goal, learning can be accelerated by rewarding it for doing so. The measure of how well the goal has been achieved is the difference between the angle to the ball and the angle to the target, α. If the absolute difference, $|\alpha|$, is small, this can be considered better than a policy that leads to it being large. Therefore, the absolute difference is multiplied into the reward function for illegal states.

7.6 Learning a Policy

A policy for the turnball problem was learnt using Sarsa(λ) (algorithm 4 on page 30). The resultant policy is referred to as *sarsa-turnball*. Due to the size of the state space, and the greater number of dimensions, it was not possible to use a discrete coding of the state space without greatly restricting the resolution allowed per dimension. Therefore, the Monte Carlo variants were not tried.

Sarsa(λ) was applied with an eligibility decay λ of 0.7, a learning rate of 0.0005, and an exploration factor ϵ of 0.1. Since the task is episodic, an appropriate choice for γ is 1. The tile

Table 7.1: Tile coding parameters used to learn the *sarsa-turnball* policy.

Variable	Minimum	Maximum	Intervals	Range	Resolution
x_b	0.295	0.4	6	0.105	0.021
y_b	-0.235	0.235	6	0.470	0.094
v_R	-2.0	2.0	6	4.0	0.8
α	$-\pi$	π	10	2π	$\frac{\pi}{5}$
u_l	0.0	1.0	11	1.0	0.1
u_r	0.0	1.0	11	1.0	0.1

coding used to represent the action-values included 15 tilings, 5 incorporating $\langle x_b, y_b, \alpha, u_l, u_r\rangle$, 5 with $\langle v_R, u_l, u_r\rangle$, and 5 with $\langle \alpha, u_l, u_r\rangle$. The coding of each variable is defined in more detail in table 7.1. Note that some variables that have been defined as part of the state of the robot were left out in the final definition of the tile coding, since they seemed to have little effect on the outcome but greatly increased the size of the tile coding data structure.

7.7 Simulated Environment

The simulated environment for this problem is more complex than for the previous task. The simulation still uses just two dimensions using a plan view. Since the simulation is in two dimensions only, the ball is simulated as a sliding puck, rather than a rolling ball. That is, the ball is simulated as though it slides along the ground rather than rolling.

With this task, the simulator is more complicated since the simulated objects collide. It is necessary for the simulation to accurately portray both where collisions occur (collision detection) and how the objects react to each collision (collision response). The objects are simulated as rigid bodies, and collisions are assumed to occur instantaneously.

Collision response is determined from the momentum of the two objects, their angular momentum, their moment of inertia and the position relative to the object's centres that the collision occurred. The robot's response is dealt with somewhat differently, as it is assumed that it is not possible for an impact with the ball to cause it to skid sideways. Therefore, all momentum transferred is adjusted to be in-line with the wheels.

Realism is desirable in the simulator, but it also needs to be fast. Some approximations were allowed in the collision handling to reduce the time required to compute each cycle. For example, the front area and fingers of the robot were approximated by a set of convex polygons, rather than two concave arcs.

7.8 Physical Test Environment

The real robot tests involved the robot's vision system to detect the position of the ball. The vision system used specialised hardware, built by the RMIT RoboCup team, to segment a video image 30 times per second, based on colour. For this problem, the main area of interest was the orange coloured segment that corresponds to the ball. Image segment data is transmitted to the laptop, which then performs a correction for the fish-eye distortion that occurs with the 90 degree field-of-view lenses used.

The ball's distance and angle from the camera were estimated and this was then converted to a position with respect to the centre of the robot. Relative ball velocity $(\dot{x}_b, \dot{y}_b)$ was then estimated from previous values of x_b and y_b. The ball speed in the X direction was estimated using the following equation:

$$\hat{\dot{x}}_t \leftarrow \hat{\dot{x}}_{t-1} + k(x_{b_t} - x_{b_{t-1}} - \hat{\dot{x}}_{t-1}) \tag{7.5}$$

and similarly for $\hat{\dot{y}}_t$. Note that the average velocity in the X direction over the last time period Δt is given by the change in position over time, or $\frac{x_{b_t} - x_{b_{t-\Delta t}}}{\Delta t}$. Here it is assumed that Δt is 1, thus allowing the simpler form shown in (7.5) to be used. The constant k was set at 0.2 for all experiments. This value was found to be sufficiently small to smooth sensor noise, while being sufficiently large that the robot could react to sudden changes in the ball's velocity.

An important factor in the performance of the robot is the calibration of the vision system. The behaviour performance is sensitive to visual accuracy because the position of the ball is a key factor in deciding what action to take. In addition, deciding which states are illegal is also based on vision data and this can have an effect on the behaviour's performance. The ball distance estimation is calculated based on the horizontal and vertical angle to the top or bottom of the ball from the camera. Therefore, the orientation of the camera is critical and must be adjusted carefully.

The vision system was slightly adjusted from that used during RoboCup 2000 due to two factors. The first was that the centre of the camera was discovered to be slightly offset from the centre of the robot — slightly to the right. Since the image was corrected for fish-eye distortion, this had the effect that, if the camera was directed towards the middle of the front of the robot, and thus was angled slightly left, across the centre line of the robot, the ball would be seen as further away when to the left of the centre line in comparison to when it was to the right of the centre line. This was corrected by straightening the camera so that its line was parallel to the centre line of the robot, and then adjusting for this in the calculation of the ball position. The second adjustment was to correct the estimate of the angle of the ball as this was out by a small factor when the ball was close to the robot.

Calibration of the angle and position of the camera was based on making sure that when the ball was between the fingers and close to the robot, that the position estimate corresponded to the actual position, and also that the position estimate corresponded to the actual position when the ball was positioned near the tips of the left and right fingers.

Each episode was performed by first setting the ball in front of the robot. When the robot sensed that the ball was between the fingers, it started executing the behaviour until the ball was out of its grasp or until the turn was completed. During the episode, the current time, state vector, actions taken and reward were written to a log. After the completion of an episode, a movement behaviour was used to return the robot to the centre of the room facing roughly 180 degrees away from the target. The ball was manually brought back in front of the robot and this caused the next episode to start. Due to accumulated errors in the robot's positioning mechanism, it was occasionally necessary to pick up the robot to move it back to centre of the room.

In order to avoid vision calibration or battery charge skewing results, the learnt behaviour was tested for several episodes, then the hand-coded behaviour was tried for a number of episodes, and then the learnt behaviour was tested again for some further episodes. It was necessary to discard some episodes where battery charge was low.

During the physical evaluation, fewer episodes were executed for the hand-coded behaviour (46) than for the learnt one (161) on the basis that less variability in performance was expected from the hand-coded one. Fortunately, sufficient trials were executed to demonstrate significant differences in performance between hand-coded and learnt behaviours.

7.8.1 Using Prediction

For the task examined in chapter 6, the *rmit-move-to* behaviour made use of a forward estimation of the state of the robot. Since this was quite successful, forward estimation was also used here when the behaviours were transferred to the physical robot. Another approach would have been to simulate the latency of the physical robot. By default, the latency was assumed to be 100 milliseconds. From the episode logs of actual runs on the the previous task, the latency appeared to be at least this, and possibly up to 200 milliseconds.

To counter the latency, most experiments with both hand-coded and learnt behaviours made use of a forward estimate of the state of the robot by 100 milliseconds. However, some additional experiments were run using the learnt behaviour with no forward estimate and with a 200 millisecond forward estimate.

7.8.2 Using Continued Learning

As with the previous task, most experiments were run with a fixed policy. That is, the policy is learnt under simulation, and then, when the policy is transferred to the robot, learning is not continued. Some experiments were performed with a limited amount of continued learning. Unfortunately, it was only possible to perform a small number of learning trials compared with the number of learning trials performed on the simulator.

The approach of continuing to learn with Sarsa(λ) on the physical robot is particularly relevant to this task, where fine control of the robot is required, and where the exact parameters of the interaction between the ball and the fingertips of the robot can only be loosely modelled by the simulator.

To ensure a fair comparison between learning and not learning on the robot, the policy must still be frozen prior to the final test. For example, Sarsa(λ) involves an element of random exploration, which may degrade the average performance. Therefore, starting with the policy developed on the simulator, some number of learning trials (154) using Sarsa(λ) are run on the physical robot, followed by a series of test trials (41) using the policy learnt as a result.

7.9 Simulator Results

Simulation performance during learning is shown in figure 7.5 on the next page. From the graph, a gradual increase can be seen for the first 300 000 trials, while after that, the performance seems to reach a plateau but also continues to vary somewhat from test to test. This tendency of Sarsa(λ) to oscillate was discussed in section 2.4.7. The learning process took about 34 hours on an Athlon 1GHz PC to perform 760 000 trials.

7.10 Physical System Results

The reward function provides us with a basic fitness measure. The mean reward totals are given in table 7.2 on the following page. An arithmetically greater reward is more desirable, and so on this basis *sarsa-turnball* should be considered the better behaviour. However, it is not clear from looking at the average whether or not these results would be reproduced in practice. For instance, there may not have been sufficient trials to allow the mean to be established sufficiently accurately. Usually, a Student t Test would be used to test the hypothesis that the rewards had different means, however in this case, the reward does not follow a normal distribution. This

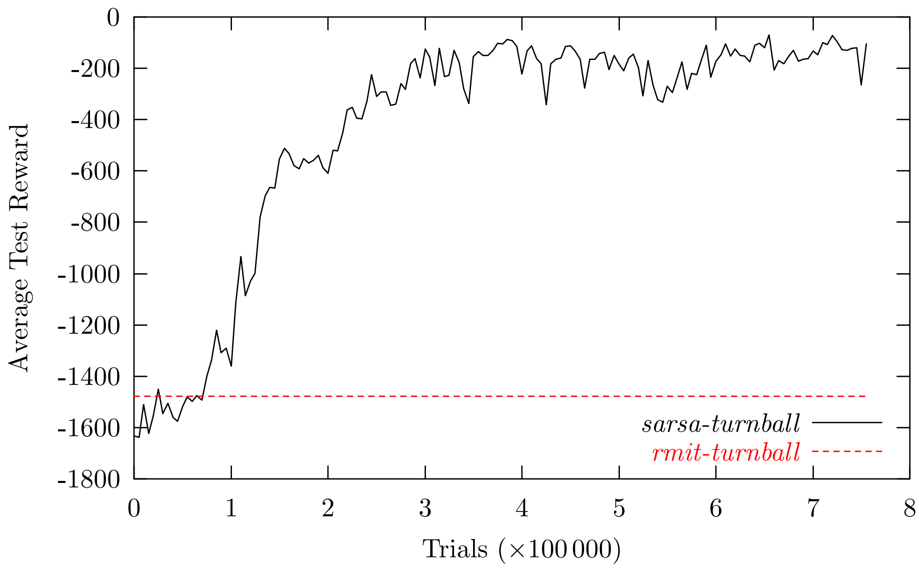

Figure 7.5: Policy performance during learning for the Sarsa(λ) algorithm.

Table 7.2: Mean reward obtained for turn-ball behaviours on the physical robot.

Behaviour	Mean reward
sarsa-turnball	-1032.9
rmit-turnball	-1550.4

can be seen in a histogram of the rewards, shown in figure 7.6 on the next page. Therefore, to compare the two populations, one approach is to apply a normalising transformation and then use some form of t test. Alternatively, a non-parametric procedure, that does not rely on the populations having a normal distribution, can be used. For this problem, the latter approach was taken, and the non-parametric procedure used was a Wilcoxon Rank Sum Test [75].

Let H_0 be the null hypothesis, that the median reward for *rmit-turnball* is equal to or greater than the median reward for *sarsa-turnball*. The alternative hypothesis is that *sarsa-turnball* has a median reward that is greater. The medians obtained for the sample data are shown in table 7.3. The Wilcoxon Rank Sum Test gives a Z test statistic of -2.10, which is less than the critical value -1.64 for a lower-tail test with a 95% confidence level, indicating that the null hypothesis H_0 can be rejected. The p-value, or probability of incorrectly rejecting the null hypothesis from this data, is 0.018.

The success or failure rate for a behaviour is another important measure of performance. Success occurs when the robot is able to completely turn the ball to the desired direction without the ball slipping from between its fingers. Success is independent of the amount of time taken and so is only a partial measure of the performance of the behaviour. Table 7.4

Table 7.3: Median rewards for *sarsa-turnball* and *rmit-turnball*.

	Median reward	Trials
sarsa-turnball	-503.9	161
rmit-turnball	-1934.2	46

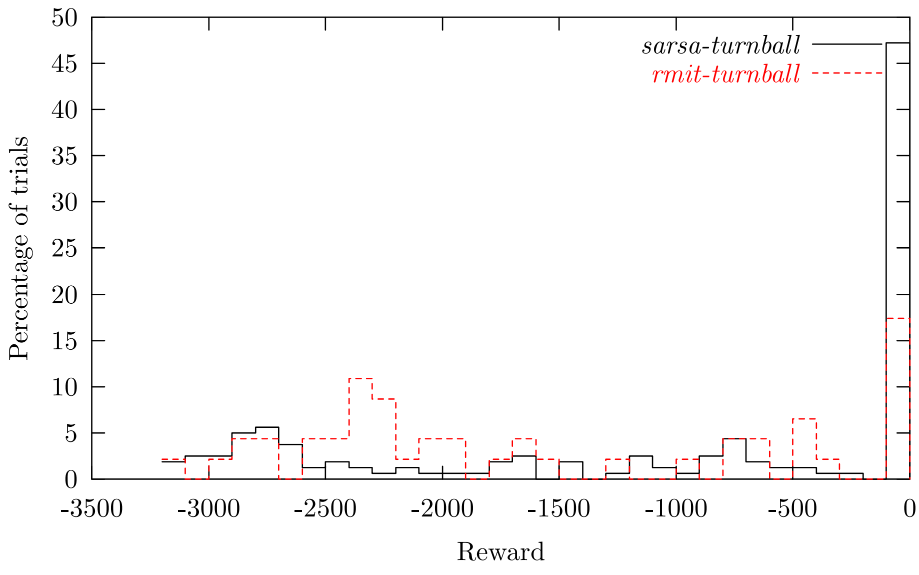

Figure 7.6: Histogram of rewards for turn-ball behaviours

Table 7.4: Success rate for turn-ball behaviours

	sarsa-turnball	*rmit-turnball*
successful episodes	76	8
success rate	47%	17%
failed episodes	85	39
failure rate	53%	83%

shows the numbers of successes and failures for the learnt and hand-coded behaviours. A χ^2 test can be used to test whether these proportions are different. The χ^2 test statistic is 13.2, which is greater than the critical value of 3.84 for a 95% confidence level, indicating that the success rates for the two behaviours are significantly different. The p-value, or probability of this data being produced by two behaviours that had the same actual success rate, is 0.00028. Since the observed success rate for *sarsa-turnball* is higher, it can be concluded that *sarsa-turnball* is significantly more successful than *rmit-turnball*.

Apart from whether or not the manoeuvre was successful, another important factor in the performance of the ball turning behaviour is the amount of time taken. In particular, it is interesting to look specifically at the amount of time taken during successful episodes. The mean episode times for successful episodes are shown in table 7.5. The hypothesis that *sarsa-turnball* takes longer during successful episodes than *rmit-turnball* can be tested using a Student t Test. The t statistic is -3.18 and this is less than the critical value of -1.66 for a 95% confidence level for a lower-tail test. Therefore, the null hypothesis, that *rmit-turnball* takes

Table 7.5: Mean episode times for successful episodes

	Mean time (seconds)	Standard deviation
sarsa-turnball	7.5	1.97
rmit-turnball	5.2	1.18

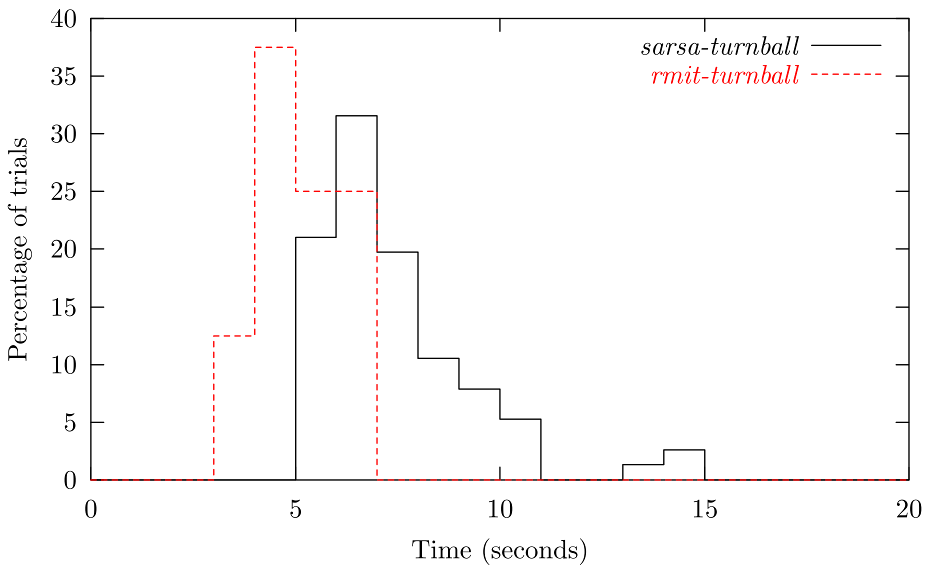

Figure 7.7: Histogram of episode times (in seconds) for successful episodes

Table 7.6: Final absolute angle to target (radians)

	Trials	Median angle (radians)
sarsa-turnball	161	0.47
rmit-turnball	46	1.91

the same time or longer, can be rejected. The p-value, or probability of incorrectly rejecting a correct null hypothesis with this data, is 0.0010. Note that this assumes that the times are normally distributed. The distribution of times can be seen in the histogram in figure 7.7. This is the only performance factor for which *rmit-turnball* produced a better result.

The angle to the target at the end of an episode is the final performance measure examined here. To simplify analysis, the absolute target angle, $|\alpha|$, is used. If the target angle is close to zero, this may indicate partial success that will not necessarily be reflected in the success rates for the behaviour. The results for the target angle are not normally distributed, and this can be seen in the histogram in figure 7.8 on the next page. As shown in table 7.6, the median is smaller for *sarsa-turnball* than for *rmit-turnball*, indicating that *sarsa-turnball* had better performance in this respect. As with the analysis of the reward results, a non-parametric test, specifically, the Wilcoxon Rank Sum Test, is used to establish the significance of the results. The null hypothesis, H_0, is that the median absolute target angle for *rmit-turnball* is less than or equal to that for *sarsa-turnball*. The alternative hypothesis is that the median absolute target angle for *sarsa-turnball* is less than *rmit-turnball*. The Z test statistic is 2.53, which is greater than the upper-tail test critical value of 1.64 for a 95% confidence level, indicating that the null hypothesis can be rejected. The p-value, or probability of rejecting a null hypothesis that is correct, is 0.0057. This can be summarised as indicating that *sarsa-turnball* produced a median final target angle significantly closer to zero than *rmit-turnball*.

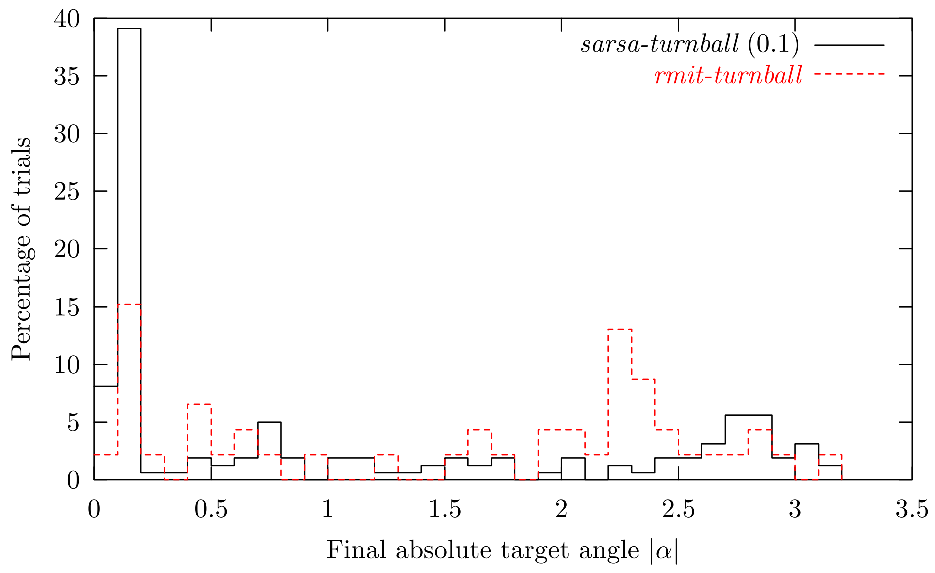

Figure 7.8: Histogram of final target angle (in radians)

Table 7.7: Success rate for turn-ball behaviours with various forward estimate times.

	sarsa-turnball 0	*sarsa-turnball* 0.1	*sarsa-turnball* 0.2
successful episodes	10	76	9
success rate	42%	47%	69%
failed episodes	14	85	4
failure rate	58%	53%	31%

7.10.1 Forward Estimation Time

In all previously mentioned physical test results for this task, forward estimation of the position of the robot of 0.1 seconds was used in the world model. This was intended to counteract the latency between receipt of sensor data and actions taking effect. Figure 7.9 on the next page shows the effect on the final target angle of varying the time used to forward estimate the position of the robot. A forward estimate time of 0 indicates that no forward estimation was performed. The relative success rates for different forward estimation amounts are shown in table 7.7. Note that the total number of episodes varies for the different types for several reasons. Some episodes had to be discarded as the battery voltage was too low—in particular a large group of *sarsa-turnball* 0.2 trials were discarded for this reason. Also, rather than specifically count the number of episodes performed with a particular set up, the experiment was run until either the battery needed recharging or until after the experiment had been running for about 20 minutes.

The χ^2 statistic between forward estimate times of 0 and 0.1 is 0.26, which is less than the critical value of 3.84 and so the null hypothesis, that the two populations have the same success rate, cannot be rejected. Between 0 and 0.2, the χ^2 statistic is 2.56, and between 0.1 and 0.2, the χ^2 statistic is 0.13. Therefore, there is insufficient evidence to suggest that the success rates are affected by altering the forward estimation time.

A Wilcoxon Rank Sum Test can be used to analyse differences in the final angle to the target for the different forward estimate times. Between times of 0 and 0.1, the Z test statistic for

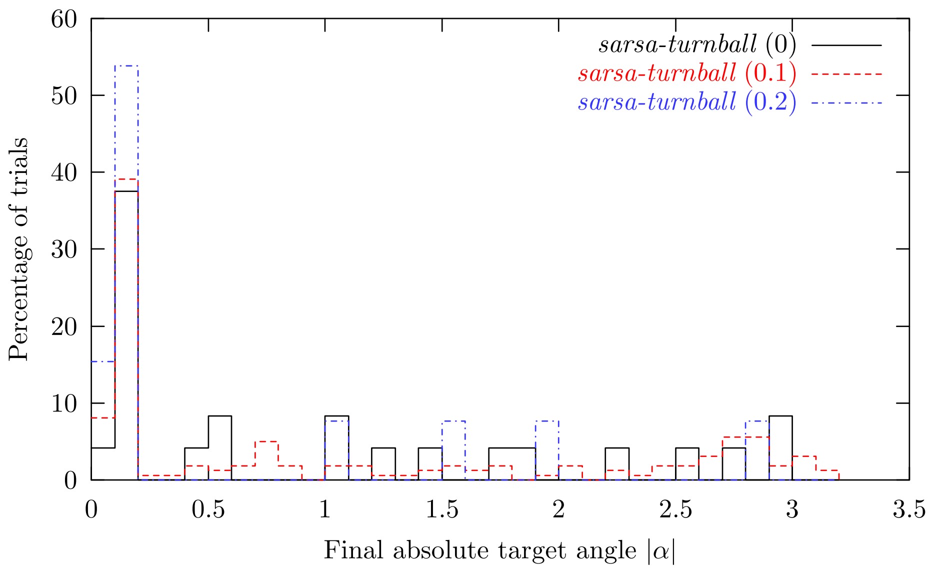

Figure 7.9: Histogram of final absolute target angle during physical trials of *sarsa-turnball* with forward estimate times of 0, 0.1 and 0.2 seconds.

Table 7.8: Final absolute angle to target (radians) with various forward estimate times.

Forward Estimate Time (seconds)	Trials	Median angle (radians)
sarsa-turnball 0	24	0.51
sarsa-turnball 0.1	161	0.47
sarsa-turnball 0.2	13	0.14

the final target angle is 0.30, which indicates that the null hypothesis, that the two populations have the same median final target angle, cannot be rejected. Between times of 0 and 0.2, the Z test statistic is -2.13, which is less than the critical value of -1.96 for a 95% confidence level, indicating that the null hypothesis can be rejected and that there are significant differences in the median final target angle for these two forward estimate times. A similar result occurs between times of 0.1 and 0.2, where the Z test statistic is -2.09. Therefore, although there is not sufficient evidence to suggest that the success rate is improved by increasing the forward estimate to 0.2, there does appear to be sufficient evidence to demonstrate a significant improvement in the final target angle. The median final target angles are shown in table 7.8.

If successful trials only are extracted, the time taken for each episode for the different forward estimate times can be analysed with a t-test. The mean successful episode times were 8.5, 7.5, and 7.4 seconds for 0, 0.1, and 0.2 second forward estimate times, respectively. There was insufficient evidence to conclude that the differences in successful episode times were significant.

7.10.2 Continued Learning

The possible advantages of continuing to learn on the physical robot were explored by running 190 trials where the robot continued to learn using the Sarsa(λ) algorithm, and then a further 41 trials where the resulting policy was used without learning. As table 7.9 on the facing page shows, the success rate was clearly degraded during the learning process, but then recovered slightly after learning was discontinued. The χ^2 test statistic for the success and failure rates

Table 7.9: Observed success rates for Sarsa(λ) turn-ball behaviours during and after continued learning compared with a base case of no learning on the robot.

	before continued learning	during continued learning	after continued learning
successful episodes	76	51	16
success rate	47%	27%	39%
failed episodes	85	139	25
failure rate	53%	73%	61%

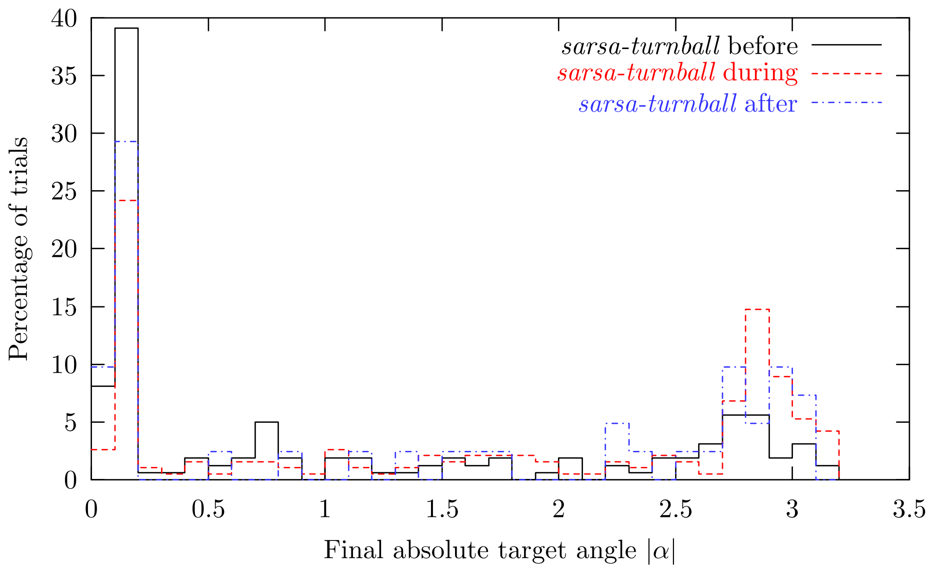

Figure 7.10: Histogram of final absolute target angle $|\alpha|$ from trials of *sarsa-turnball* before, during and after continued learning on the physical robot.

prior to and during learning is 15.6, which is well in excess of the critical value of 3.84 for a 95% confidence level. This indicates that the success rate during learning is significantly worse than prior to continued learning. Although the success rate after continued learning does not appear to recover to the levels prior to continued learning, there is insufficient evidence to suggest that the success rate is significantly different.

As mentioned previously, it is useful to examine the final absolute target angle as this may show partially successful turns occurring. Figure 7.10 shows that trials during continued learning and to some extent those after continued learning, terminated while the target angle was still quite large. A Wilcoxon Rank Sum Test can be used to test the hypothesis that the median absolute target angle is affected by continued learning. Let the null hypothesis be that the policy before and the policy after continued learning have the same median absolute target angle. The Z test statistic is 1.34, which is less than the critical value of 1.96 for a 95% confidence level. Therefore the null hypothesis cannot be rejected. In other words, there is insufficient evidence to suggest that continued learning of 190 trials has an effect on the final target angle.

Another possible approach is to continue learning indefinitely. Again, the Wilcoxon Rank Sum Test can be used to determine the effect on the target angle *during* continued learning. Let the null hypothesis be that the policy before and the policy produced during learning have the

Table 7.10: Observed final absolute angle to target (radians) for Sarsa(λ) before, during and after continued learning.

	Trials	Median angle (radians)
before continued learning	161	0.47
during continued learning	154	1.81
after continued learning	41	1.52

same median absolute target angle. The Z test statistic is -4.56, which is much less than the lower critical value of -1.96 for a 95% confidence level. The p-value is 5×10^{-6}. Therefore the null hypothesis can be rejected, and the target angle is significantly worse during continued learning than prior to it. This confirms the suspicion that some aspects of the learning algorithm might reduce the performance. The median absolute target angles before, during and after continued learning are shown in table 7.10.

It is interesting to note that although the behaviour was less successful during learning on the physical robot, the average amount of time taken for successful episodes (7.5 seconds) decreased (to 6.8 seconds). It then increased again (to 7.8 seconds) after learning was halted . A t-test can be used to examine the differences in mean successful episode times. Let the null hypothesis be that the policy before learning had the same mean time for successful episodes as the policy produced during learning on the robot. In this case, an F-test indicates that the variances are different, and so unequal variances are assumed. The t-test statistic is 2.36, which is larger than the critical value of 1.66 for a one tail test with a 95% confidence level. The p-value is 0.010. This indicates that the null hypothesis can be rejected and that the mean time for successful episodes was smaller during learning (6.8 seconds) than before learning (7.5 seconds).

Times after continued learning were significantly higher than those during learning (p-value of 0.05), but not significantly higher or lower than those prior to learning (p-value of 0.24). A possible explanation is that the learning tended to disrupt the behaviour as the agent attempted to explore different actions every so often. The task is a delicate one, and even a small amount of disruption causes the ball to be lost from between the fingers. Since Sarsa(λ) distributes exploration roughly evenly over time, it disrupted longer episodes with a greater probability that it did short episodes. This is also consistent with the increased failure rate during continued learning.

7.11 Discussion

Apart from the quantitative aspects of the behaviours, it is also instructive to look at qualitative aspects. Specifically, it is useful to look at the shape of the path of the robot, and the behaviours response to certain sorts of problems that occurred.

Figures 7.11 on the next page and 7.12 on the facing page show successful turns by *rmit-turnball* and *sarsa-turnball*, respectively. The initial conditions are similar, but slightly different and this is reflected in the southward turn by *rmit-turnball* compared with the northward turn by *sarsa-turnball*. The arc is clearly much smaller for *rmit-turnball*. Note that, in these figures, the position of the ball has been estimated by the vision system, and may not correspond to exactly where the ball actually was. The positions are logged at 0.1 second intervals, so for the successful turn for *rmit-turnball*, in figure 7.11, the speed of the robot is relatively fast towards the end of the episode. It is able to keep control of the ball while turning because the ball has

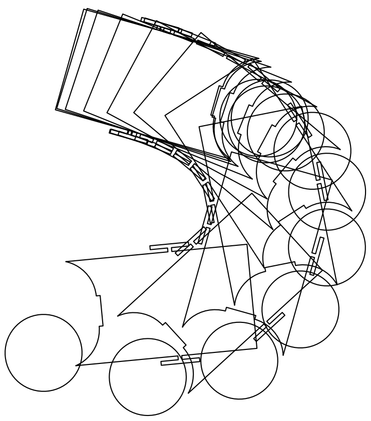

Figure 7.11: A successful turn executed by *rmit-turnball* on the physical robot.

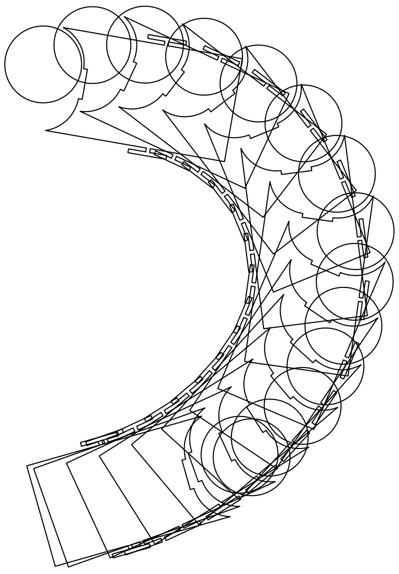

Figure 7.12: A successful turn executed by *sarsa-turnball* on the physical robot.

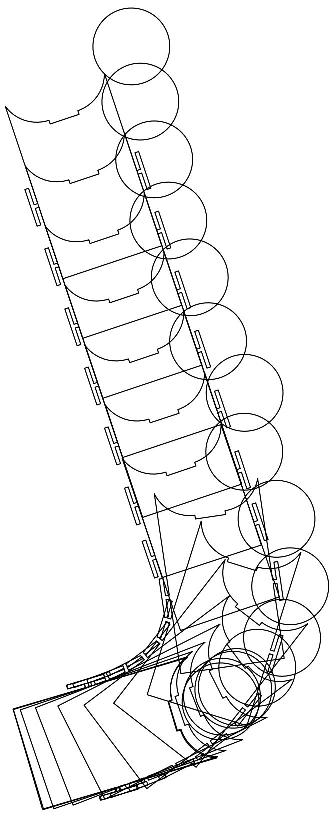

Figure 7.13: An unsuccessful turn executed by *rmit-turnball* on the physical robot. The behaviour does not respond to the ball drifting to its right but just moves in a straight line.

stopped rolling and is skidding along the floor. The difficulty with this is that it requires the ball to be in a specific position for this to occur. This somewhat accounts for both the relatively low success rate of this behaviour, and also its greater speed when successful. Figure 7.13 shows an example of the rigid nature of *rmit-turnball*. In this case, the ball is too far out of its grasp to be brought back by moving straight ahead, but it fails to adjust. Figure 7.14 on the next page, on the other hand, shows how *sarsa-turnball* turns back towards the ball a little after it starts to leave its grip. Unfortunately, in this case, it over-reacts and this knocks the ball with the left finger. Note that the starting conditions for both successful and unsuccessful turns were quite similar. That is, the ball was within the fingers, slightly less than a 180 degree turn was required, and the ball was either stationary or moving slowly.

7.12 Summary

The results establish significant differences in the performances of *rmit-turnball* and *sarsa-turnball*. In general, *sarsa-turnball* gave better performance than *rmit-turnball* in all factors except for how quickly it turned the ball, when it did so successfully. The *sarsa-turnball* behaviour did better in terms of the reward function. However, since this includes shaped components, that is components that are designed to assist learning, comparing the accumulated reward may not be entirely fair. If the shaped components are removed, the overall measure is the absolute target angle at the end of the trial. This has been shown to be significantly closer

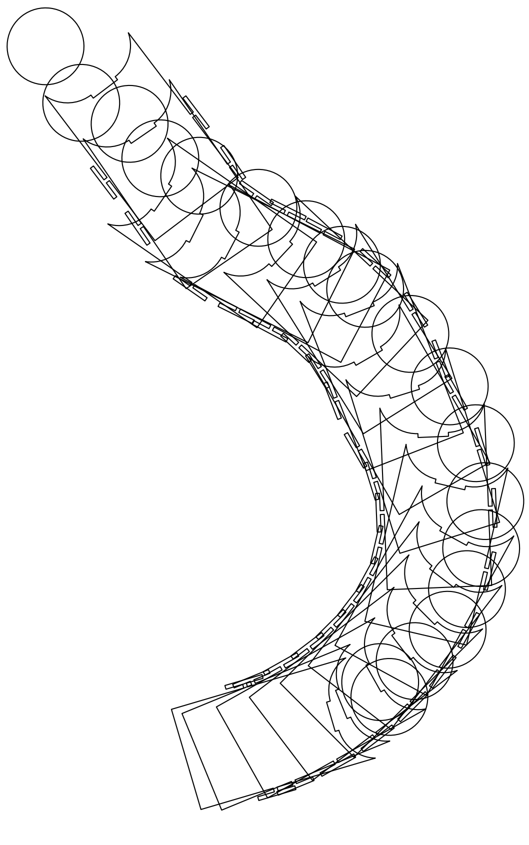

Figure 7.14: An unsuccessful turn by *sarsa-turnball* on the physical robot. The behaviour brings the ball back into the robot's grasp but ultimately loses control of it.

to zero, and thus corresponding to better performance, for *sarsa-turnball* than for *rmit-turnball*.

In solving this problem, the learnt behaviour had some small handicaps in comparison to the hand-coded one. Despite the handicaps, it was able to produce a better performance overall.

The first handicap was to do with the simulation model of the interaction between the ball and the robot's fingers. The hand-coded behaviour *rmit-turnball* was designed to make use of the fact that the ball tends to stick to the fingers. When the ball sticks, it skids along the floor rather than rolling. This increases the friction between the ball and the floor, and, if the robot is moving forwards, the ball pushes back more strongly. Therefore, if the ball is in the right position, a tighter turn is possible. The change in friction is not simulated and so the learnt behaviour is unlikely to be optimal in this regard.

Another aspect that was different between the hand-coded behaviour and the learnt one was that the hand-coded varied an acceleration factor as well as the target velocity. For most tasks, robot acceleration is usually required to be as high as possible, because it is desirable for the robot to be able to change speed and direction as quickly as possible. In control system terms, the velocity control is under-damped, meaning that the velocity increases or decreases rapidly, but then tends to oscillate around the target velocity. Reducing the acceleration factor increases the damping on the velocity control. For this problem, lower acceleration and smoother, less jerky velocity changes are desirable as they are less likely to cause the ball to bounce out from between the fingers. The hand-coded behaviour used small acceleration factors, varied the acceleration factor depending on the situation, and used differing acceleration factors for each wheel. The learnt behaviour was trained using a proportional control system and restricted to a single acceleration factor.

The learning problem was constrained by limiting the possible set of legal states and the allowed actions. By restricting the ball to being between the fingers, the solution was restricted to some form of dribbling to turn the ball. Had this restriction been removed, it might have been found to be more efficient to use a series of sideways flicks to turn the ball. The action set did not include kicking or reversing either wheel. Although it does not seem likely that kicking will help this task, it is possible that reversing might be useful in some cases, such as reversing slightly to help capture a ball moving towards the robot, or reversing one wheel slowly while the other moves forward quickly, to execute a tight turn.

In comparison to the previous case study, two additional issues were examined. The first issue examined was the effect of prediction or forward estimation on the success of the learnt behaviour. Predicting the position of the robot is intended to give the behaviour a picture of the world as it will exist when the action happens. The second issue examined was the effect of continued learning, or, in other words, continuing to use reinforcement learning after transferring the policy to the robot. It was expected that continuing learning on the robot would cause the policy to improve, however this did not appear to happen. However this is probably not surprising given that only a few hundred learning trials were run, whereas the simulator trials indicate that a few hundred thousand would be needed to learn the task from scratch.

The key finding of this chapter is that it is possible to learn a sophisticated behaviour using simulator based learning, and for that learnt behaviour to out-perform a hand-coded equivalent. The next chapter will present a third and final case study involving a different robotic task, again involving manipulation of the ball, but with the additional factor that a wall is nearby.

Chapter 8

Off Wall Task

This task involves getting the ball clear of the wall or running the ball down the wall towards the goal. When the ball is close to the wall, it can only be accessed from one side. Also, it is important not to get too close to the wall or risk stalling the robot.

A stall may occur for a number of reasons, but the general problem is that the motors cannot turn one of the wheels in the desired direction. The low level controller flags a stall condition when the motor current exceeds a certain threshold for a period of time. Stalls generally occur when the robot bumps into an immovable obstacle. Sometimes it is possible to recover from a stall by reversing slightly. However, if the robot becomes jammed near the wall, the robot, due to its rectangular shape, may not be able to recover. It is a good idea to avoid stalling in RoboCup as it gives the other team an advantage while the robot recovers.

For the purpose of this study, it is assumed that the wall lies along the Y axis ($x = 0$) with the playing field on the negative X side. The target goal is in the positive Y direction.

8.1 Motivation

As with other case studies presented here, the task is realistic. From the point of view of RoboCup, it is desirable to be able to keep playing the ball when it is near the wall, rather than simply having to avoid it entirely. Also, it is desirable to treat it as a separate case so that the robot does not ignore the wall and become jammed against it. Finally, it is an increase in complexity from the previous tasks in that it involves both manipulation of the ball and close interaction with a wall.

8.2 Hand-coded Algorithm

The hand-coded algorithm is based on the algorithm used for the RMIT United Team and is called *rmit-offwall*. The output of the hand-coded algorithm consists of a speed and an angular velocity. It is shown as algorithm 11 on the following page.

8.3 State Representation

In choosing the state representation for the off-wall task, the main consideration was to ensure that sufficient information was available so that the state could be considered Markovian, and also that just enough information was provided, and no more. An initial formulation of the

```
1. (If ball in fingers.) if $d_b < 0.37$
   where $d_b$ is the distance from the centre of the robot to the centre of the ball in metres.
   The ball has a radius of 0.1 metres, while the distance from the centre of the robot to its
   front is 0.2 metres. The fingers extend an addition 0.07 metres in front of this.
     (a) (Stop forward movement.) $v \leftarrow 0$
         where $v$ is the target velocity for the centre of the robot, or the average velocity for
         both wheels.
     (b) (Spin counter-clockwise.) $\omega \leftarrow \pi$
         where $\omega$ is the target angular velocity.
2. (If ball not in fingers.) else
     (a) (Move closer to the ball.) $(v, \omega) \leftarrow$ rmit-moveto
         use the hand-coded behaviour for moving to a target point to move closer to the ball.
3. (Convert velocities.) Angular and average velocity is converted into left and right wheel
   velocities.
```

Algorithm 11: The *rmit-offwall* behaviour

state included the absolute position of the robot and the ball. This includes the Y coordinate for both the robot and the ball. However, the robot could be considered to be at $y = 0$ without changing the problem, and therefore one of the Y coordinates could be discarded. That is, it is not necessary for the robot to know where about it is on the field, but merely its distance from the wall. It is not necessary to know the absolute position of the ball, but merely its relative position to the robot.

The above considerations led to a state comprised of the following components,

$$\langle d_{rw}, d_{bw}, y_{rb}, \phi_r, v_r, \omega_r, \dot{x}_b, \dot{y}_b \rangle.$$

The static and dynamic components of the state representation are shown in figures 8.1 on the next page and 8.2 on page 116, respectively.

The distance from the robot to the wall is denoted by d_{rw}, the robot's heading by ϕ, the distance from the ball to the wall by d_{bw}. Previously, the robot's speed was described by the two wheel velocities but for this problem it is represented by the average velocity, v_r, and angular velocity, ω_r. The reason for using this form of representation is that in this task, it is expected that the robot will tend to turn on the spot, meaning that the forward velocity will usually be small or zero, while the angular velocity will vary greatly. However, when this is represented using wheel velocities, both will vary considerably. Therefore, forward and angular velocity appear to be a simpler representation for this task and may assist the function approximation process. The ball speed $(\dot{x}_b, \dot{y}_b)^T$ is based on filtered estimates of the absolute (x, y) position for the ball, rather than relative coordinates. The filter is based on equation (7.5).

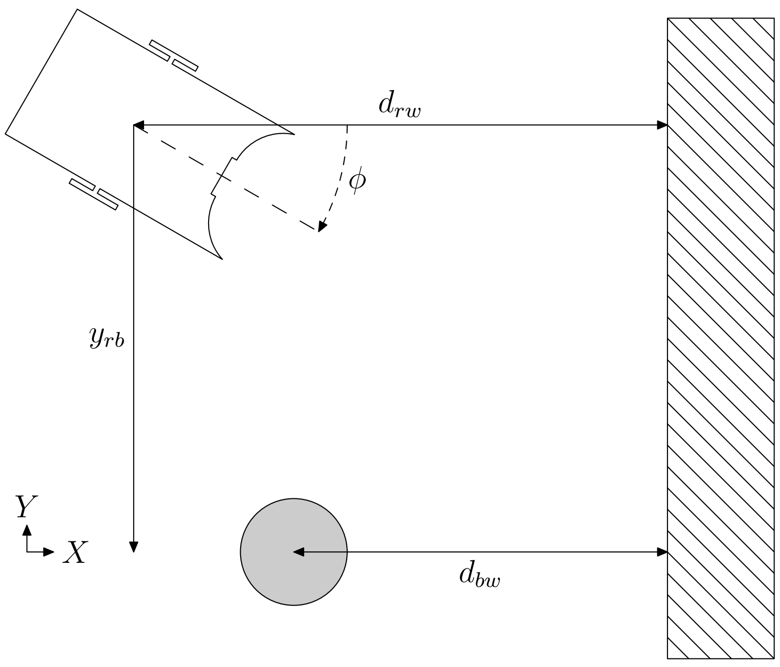

Figure 8.1: Static components of the state representation for the off-wall task. In the above example, both the distance from the robot to the ball y_{rb} and the heading of the robot ϕ are negative. The two distances from the wall d_{rw} and d_{bw} are measured perpendicular to the wall.

8.4 Action Representation

As with the previous tasks, the robot's action is represented by left and right target wheel velocities, u_l and u_r. These two target velocities, in metres per second, are selected from the range $[-1, 1]$ with a resolution of 0.2. This yields 11 different target velocities for each wheel, or a total of 121 different possible actions. Some preliminary tests with the simulator showed that this was a sufficiently limited number of actions for a policy to be learnt in a reasonable amount of time. Increasing the number of actions per wheel to 21, thus increasing the total actions to 441 made the learning process much slower and it did not converge to a solution running for about a week.

8.5 Reward Function

It took some trial and error to define a useful reward function for this task. The main difficulty was that the full effect of a set of actions may be uncovered sometime after the ball is out of reach of the robot. However it is also desirable to end the episode when the ball is out of reach, so that the scope of the behaviour is reasonably limited. Therefore, it is hard to assess the true reward due to the episode, even after the last step. The compromise used here was to assume that the ball's velocity would be a good indicator of where the ball would end up. This assumption will not be valid in cases where the ball is bounced against the wall, and seemed, at the terminal state, to have a velocity directed towards the wall. Further simulation runs would probably show the ball bouncing off.

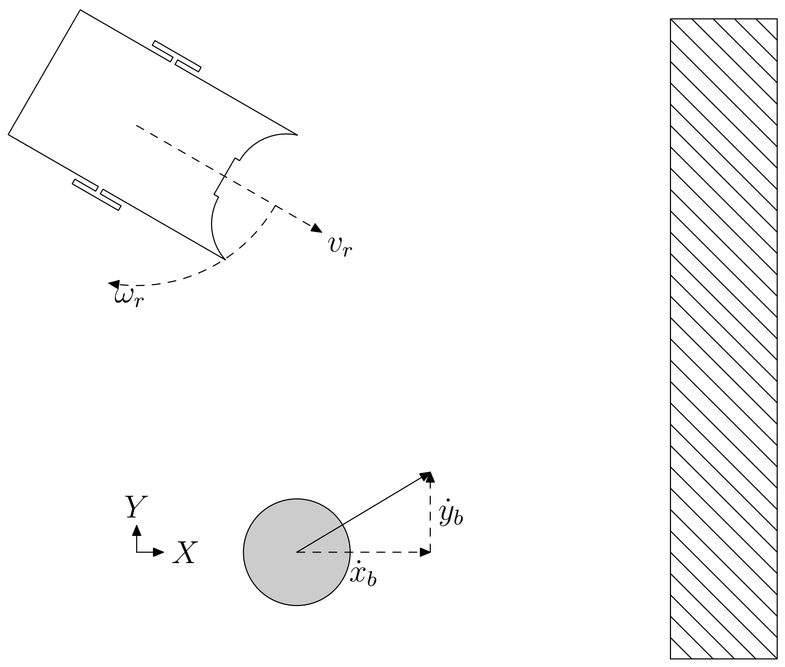

Figure 8.2: Dynamic components of the state representation for the off-wall task. The velocity of the robot, v_r is calculated from the average of the speeds of the two wheels, and is negative when the robot is moving backwards. The angular velocity of the robot, ω_r, is measured in radians and is negative in the above diagram. The speed of the ball is represented by the vector $(\dot{x}_b, \dot{y}_b)^T$ and is based on the absolute motion, rather than relative to the motion of the robot.

The reward function used had the following form,

$$\mathcal{R}(s,a,s') = \mathcal{R}_S(s') + \mathcal{R}_Y(s') + \mathcal{R}_X(s') + \mathcal{R}_T(s'), \tag{8.1}$$

$$\mathcal{R}_S(s') = \begin{cases} -1000 & \text{if the robot is stalled} \\ 0 & \text{otherwise,} \end{cases} \tag{8.2}$$

$$\mathcal{R}_Y(s') = \begin{cases} 20\dot{y}_b & \text{if } s' \text{ is terminal} \\ 0 & \text{otherwise,} \end{cases} \tag{8.3}$$

$$\mathcal{R}_X(s') = \begin{cases} -100\dot{x}_b & \text{if } s' \text{ is terminal and } \dot{y}_b > 0 \text{ and } \dot{x}_b < 0 \\ 0 & \text{otherwise,} \end{cases} \tag{8.4}$$

$$\mathcal{R}_T(s') = \begin{cases} 0 & \text{if } s' \text{ is terminal} \\ -0.1 & \text{otherwise.} \end{cases} \tag{8.5}$$

The main reward function (8.1) is divided into four parts, all of which are dependent on the state being transitioned to s' and are independent of the originating state s or the action a. The four parts are: a penalty for stalling $\mathcal{R}_S$ (8.2), a reward for pushing the ball in a positive Y direction $\mathcal{R}_Y$ (8.3), a reward for pushing the ball in a negative X direction $\mathcal{R}_X$ (8.4), and a small, time-based penalty for taking a step $\mathcal{R}_T$ (8.5). The ball velocity components, $\dot{x}_b$ and $\dot{y}_b$ are measured in metres per second.

The state s' is terminal if and only if, for the state s', $|y_{rb}| > 0.4$ metres, or $d_{bw} > 0.6$ metres, or $d_{rw} > 0.6$ metres, or the robot is stalled. In other words, the episode terminates when the distance along the wall between the ball and the robot is too large, or when either the ball or the robot drifts too far away from the wall. A stall almost always occurs when the robot comes into contact with the wall, but can also occur if the ball gets jammed between the robot and wall.

There are several things to note about the design of this reward function. The $\mathcal{R}_Y$ component rewards an episode that finishes with the ball moving in a positive Y direction and penalises episodes that terminate with the ball moving in a negative Y direction. This means that the behaviour is specific to the case where the goal is in the positive Y direction. However, as discussed in section 4.3 on page 50, it is possible to reuse behaviours for symmetrical tasks. The $\mathcal{R}_X$ component rewards an episode that ends with the ball moving in a negative X direction, or in other words, moving away from the wall. However it only provides this reward when the ball is also moving in a positive Y direction. This avoids rewarding episodes that get the ball to move away from the wall but in the wrong Y direction. Note that if the ball is moving in a positive X direction this is not penalised. Movement away from the wall is more highly rewarded per unit velocity than movement in a positive Y direction. In practise, getting the ball to move away from the wall is harder for the robot to achieve than kicking it along the wall. The final part of the reward, $\mathcal{R}_T$, attempts to ensure that the robot acts in a timely manner by penalising the robot based on the time spent in an episode.

8.6 Learning a Policy

The Sarsa(λ) algorithm was used to learn a policy for this task. As with the ball turning task, the dimensionality of this task is high and it is therefore difficult to discretise the state since the high dimensionality means that a large number of array cells are required to fully map action-value function for each part of the state space. For this reason, the approach of using Monte Carlo algorithms with discretised state was not tried.

Table 8.1: Tile coding parameters used to learn the *sarsa-offwall* policy. Distances for d_{rw}, d_{bw}, and y_{rb} are shown in metres, speeds $\dot{x}_b$, $\dot{y}_b$, v_r, u_l, and u_r in metres per second, angles ϕ in radians, and angular velocities ω_r in radians per second.

Variable	Minimum	Maximum	Intervals	Range	Resolution
d_{rw}	0.1	0.6	4	0.5	0.166
d_{bw}	0.1	0.6	2	0.5	0.5
$\dot{x}_b$	-1	1	2	2.0	2.0
$\dot{y}_b$	-1	1	2	2.0	2.0
v_r	-1	1	2	2.0	2.0
ω_r	$-\pi$	π	2	2π	2π
y_{rb}	-0.4	0.4	4	0.8	0.266
ϕ	$-\pi$	π	8	2π	$\frac{\pi}{4}$
u_l	-1	1	2	2.0	2.0
u_r	-1	1	2	2.0	2.0

Several different approaches were tried to find a configuration of the tile coding that produced a good result quickly. The main difficulty is that the state space has a high dimensionality and that if a straightforward coding of the data is used, the function approximator requires more memory than is available. Even if sufficient memory is available for the function approximator, training time may be excessive if there are many weights to train. Initially, the approach of tiling across only a few dimensions at a time was tried. For example, one tiling could involve just (y_{rb}, v_r, u_l, u_r). This type of approach is useful when the function being approximated contains some variables that are more relevant than others, and therefore need more detailed representation. This approach is mentioned by Stone and Sutton [111] along with the trick of hashing the tile coding. The hashing approach was not tried.

The final approach used here was to reduce the number of intervals for most variables to 2. This reduction allowed all variables to be included in all tilings. This approach has the advantage that it does not make assumptions about the relationships between the variables involved.

The experiments shown here made use of 20 randomly positioned tilings over the full set of variables, with the ranges and intervals described in table 8.1. The learning rate was set to 0.0005 or $\frac{0.01}{N}$where N is the number of tilings. The final sparse tile coding was about 2.6 megabytes in size. The decay rate for eligibility λ was set to 0.7, and the exploration factor ϵ was set to 0.1.

8.7 Simulated Environment

The simulated environment makes use of the same collision detection and response mechanism as the previous case study. During simulated training episodes, a wall is positioned along the line $x = 0$ and the robot is initially positioned at a random distance from the wall, between 0.1 and 0.6 metres away. It has a random heading and random wheel velocities, each in the range $[-1, 1]$ metres per second. The ball's position is selected at random so that its centre is between the robot's centre and the wall and that its Y position relative to the robot is in the range $[-0.4, 0.4]$ metres. The ball's velocity $(\dot{x}_b, \dot{y}_b)$ is also selected at random by selecting $\dot{x}_b$ from the range $[-1, 1]$ and doing similarly for $\dot{y}_b$.

A difficulty with a completely random selection of starting conditions, is that the robot, ball or wall may already be colliding, or overlapping with one of the other objects. Therefore, it is necessary to throw away any randomly selected situations that involve overlapping objects. Also discarded are any situations where the ball is more than 0.3 metres away, since if the ball is too far away, the robot may not have an opportunity to influence the ball's direction prior to the episode ending.

8.8 Physical Test Environment

An important problem when transferring behaviours from the simulator to the physical robot, is how to accurately estimate the state. This task introduces two problems in estimating the state, that did not come up in the tasks previously examined in this thesis. The first problem is that the state must identify the relative position and angle of the wall, even when the robot cannot see the wall directly. The second problem is that the state must identify the relative position of the ball, even when it cannot be seen. These two problems are both instances of problems with the assumption of full observability. They are resolved here by maintaining a set of beliefs in the world model that track the relative position of these objects even when they cannot be seen.

To resolve the problem of maintaining a belief of where the robot is relative to the wall, wheel encoder data are integrated into the robot's position. However such data are prone to errors that accumulate over time. Therefore, it is useful to take landmark sighting data from the vision system and to fuse this with the wheel encoder data to form a combined estimate of the position. The landmark in this case is the wall, and the vision system, when it sees it, provides an estimate of the angle to the wall $\hat{\phi}$, and the distance to the wall $\hat{d}_{rw}$. The vision estimates are most accurate when the robot is facing perpendicular to the wall. The position information provided by the wheel encoders are combined with the visual estimates using a filter.

To resolve the problem of maintaining a belief about where the ball is when it cannot be seen, the world model contains a current estimate of the ball's position and speed. This is updated by new sightings using a filter.

When using a simulated environment, it was possible to randomly select the starting conditions from a uniform distribution. However, in the physical environment, it was more natural to use those starting conditions obtained by using a behaviour to move the robot within range of the ball. The behaviour used to move the robot into range actually attempted to move to a target point with the same Y coordinate as the ball, but with an X coordinate of -0.5, or in other words, 0.5 metres away from the wall. Another advantage to this approach was that it was similar to the behaviour used during the RoboCup competition to bring the robot alongside the ball.

8.9 Simulator Results

The Sarsa(λ) algorithm was run for about 20 hours on an Athlon 1GHz PC, and in that time performed 830 000 training episodes. The performance of the policy learnt using Sarsa(λ) is shown in figure 8.3 on the next page. As with the previous chapters, the performance is based on testing the policy using a fixed test set of 200 randomly generated scenarios after every 1000 learning trials. Overall, this graph shows that the performance of *sarsa-offwall* quickly achieves a level well above that produced by the hand-coded behaviour *rmit-offwall*. For the test set, the hand-coded behaviour obtains an average reward of -179 per trial whereas *sarsa-offwall* receives an average reward of -17 after 20 000 trials and by 830 000 trials produces an average

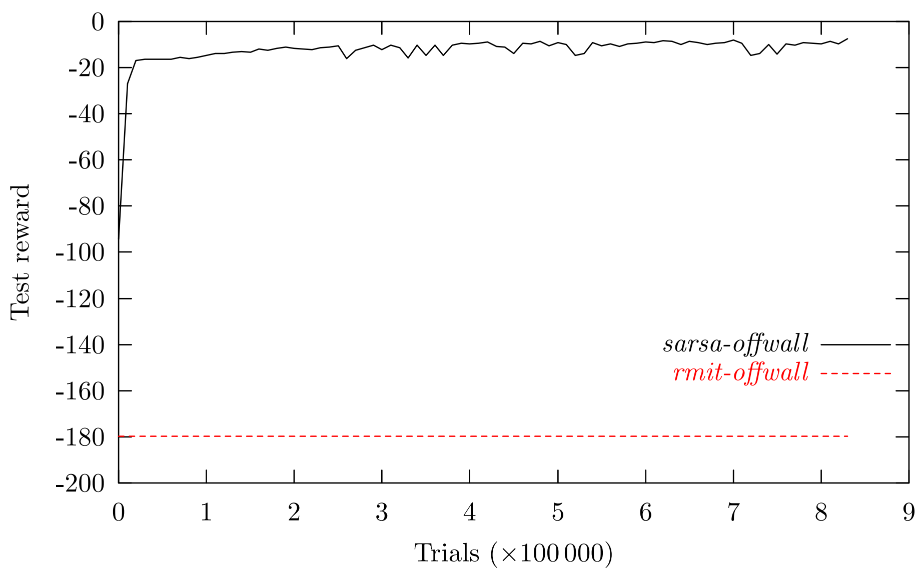

Figure 8.3: Policy performance during simulator learning for Sarsa(λ) (*sarsa-offwall*) compared with the performance of the hand-coded behaviour (*rmit-offwall*).

Table 8.2: Comparison of the average reward obtained using the learnt behaviour (*sarsa-offwall*) and the hand-coded one (*rmit-offwall*). Twenty of the *rmit-offwall* trials ended in a stall condition. The last row shows the effect of removing these trials from consideration.

Behaviour	Average reward	Std. Dev.	Trials
sarsa-offwall	25.7	32.3	248
rmit-offwall	-104.9	325.6	175
rmit-offwall (excluding stalls)	11.8	27.2	155

reward of −7.6.

Figure 8.4 on the facing page is a magnification of the previous figure that shows how the performance varies considerably but the overall trend is a slight increase in performance.

8.10 Physical System Results

Table 8.2 shows the average reward obtained for the hand-coded versus the learnt behaviour. Some of the trials ended in a stall condition, skewing the distribution of rewards, as a stall condition was punished severely by the reward function. Only *rmit-offwall* produced collisions with the wall that caused the robot to stall its motors. If the trials involving a stall condition are removed from the sample, the average reward improves but is still not as good as that for *sarsa-offwall*.

To understand these results more clearly, consider the histograms in figures 8.5 and 8.6 on page 122. These show the distribution of reward results for the two behaviours. From these graphs, we can see that *sarsa-offwall* produces negative results less often, and when producing positive results, is more likely to produce a reward above 40.

The total reward obtained during an episode is useful as a combined measure of overall fitness, however it is not very meaningful on its own. Other factors to look at are: how long

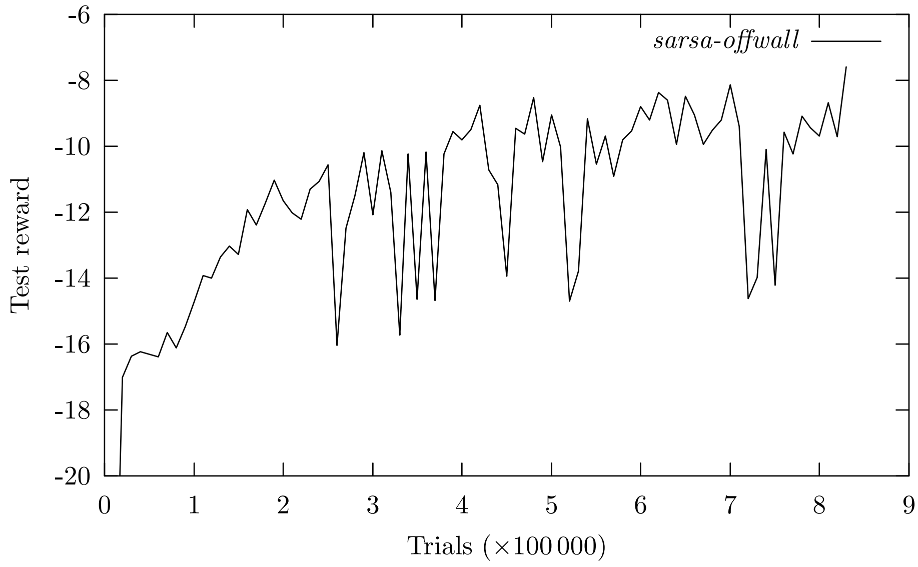

Figure 8.4: Policy performance during simulator learning for Sarsa(λ) (*sarsa-offwall*) showing in more detail the performance towards the end of the learning process.

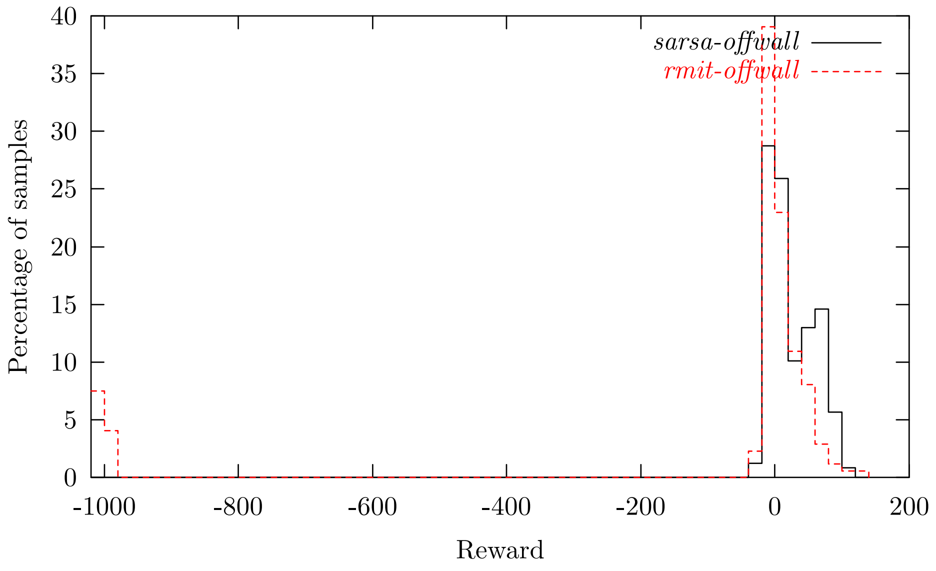

Figure 8.5: Histogram of rewards obtained including stall conditions. The learnt behaviour (*sarsa*) did not cause a stall and always produced results near or above zero.

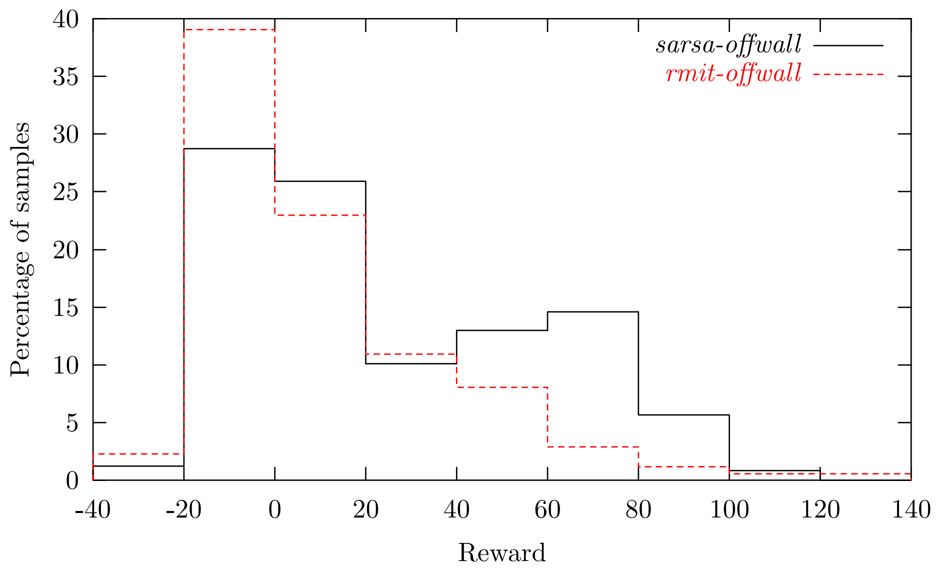

Figure 8.6: Histogram of rewards obtained showing the detail from the previous graph for rewards above −40. This graph clearly shows that *sarsa-offwall* produces better results more frequently than *rmit-offwall* even if the problem of stalling were ignored.

Table 8.3: Average trial time

	Time (seconds)	
	Average	Std. Dev.
sarsa-offwall	2.81	2.42
rmit-offwall	3.59	3.05

each trial took, how frequently stalls occurred, the average speed of the ball in the positive Y direction, and the average speed of the ball in the negative X direction (away from the wall). With the last factor, if the ball speed is positive in the X direction, this can also be beneficial as the ball will tend to bounce off the wall.

The average trial time is smaller for *sarsa-offwall* as shown in table 8.3. According to a standard t-test for the difference in two means, and assuming that the trial times are normally distributed, the mean trial time for *sarsa-offwall* is significantly smaller with a p-value of 0.0037. A short trial time may be due to the behaviour reaching a terminal state early, by moving out of range, for example. Therefore, a short trial time on its own does not necessarily indicate good behaviour. On the other hand, since behaviours do not act in isolation, but are part of a collection of behaviours that form the whole system, a short trial time can be positive as it allows other behaviours an opportunity to act. In this case, the longer trial time for *rmit-offwall* is due, at least in part, to the tendency of this behaviour to stall the robot.

Frequency of stalls is clearly differentiated between the two behaviours, as summarised in table 8.4 on the facing page. The *sarsa-offwall* behaviour did not cause any stalls, while they were frequent for *rmit-offwall*. When observing the action of the *rmit-offwall* behaviour, it was quite noticeable that in a number of trials where a stall was not detected by the robot, the robot was briefly jammed, holding the ball against the wall. Also, the wall became scratched and marked from where the robot tried to rotate when too close to the wall. On examination of the episode logs, it appears that the main problem is that the distance estimate to the ball

Table 8.4: Frequency of stalls

	Number of stalls	Number of trials	Percentage of trials with stalls
sarsa-offwall	0	248	0%
rmit-offwall	20	175	11.4%

Table 8.5: Average Y component of ball velocity at end of trial

	Average	Std. Dev.
sarsa-offwall	0.22	0.42
rmit-offwall	0.07	0.37

makes it seem that the ball is more than 0.37 metres away, when in fact the ball is actually 0.3 metres away and as close to the robot as it is going to get. This was not merely a calibration error, as it did not occur when the ball was not near the wall. It seems that it is either due to an orange reflection from the white wall being misinterpreted by the vision system, or that when the ball is squeezed between the robot and the wall, the top of the ball is higher and thus appears to be further away. On the other hand, *sarsa-offwall* had better stall behaviour because it always allowed for how far the wall was away, regardless of the distance to the ball. Therefore, when the ball appeared to be behind the wall, it did not cause the robot to charge into the wall regardless.

The average speed of the ball in the Y direction is significantly larger for *sarsa-offwall* than for the hand coded behaviour, as shown in table 8.5. The t-test p-value is 0.00017. A positive speed in the Y direction is desirable because this is the direction of the target goal. If the ball is moved off the wall but is pushed in the direction of the home goal, this could be worse than not getting the ball off the wall.

The average speed of the ball in the X direction was more negative for the *sarsa-offwall* behaviour, as shown in table 8.6. The t-test p-value is 0.0375. It is desirable for the ball speed to be negative in the X direction so that the ball moves away from the wall. The overall distribution of results for the *sarsa-offwall* behaviour shows two peaks, as shown in figure 8.7 on the next page. One of these peaks is where the X component of the ball's velocity is zero, indicating the ball being run along the wall. The other is around -0.5, indicating the ball being pushed away from the ball successfully.

8.11 Discussion

An advantage of always moving to a point 0.5 metres away from the wall before starting the behaviour is that it allowed an observer to monitor the accuracy of the estimate of the robot's position. In all tests, the estimate was accurate to within a few centimetres.

The action representation used here constrains the solution found by SARSA(λ). In compe-

Table 8.6: Average X component of ball velocity at end of trial

	Average	Std. Dev.
sarsa-offwall	-0.22	0.51
rmit-offwall	-0.13	0.30

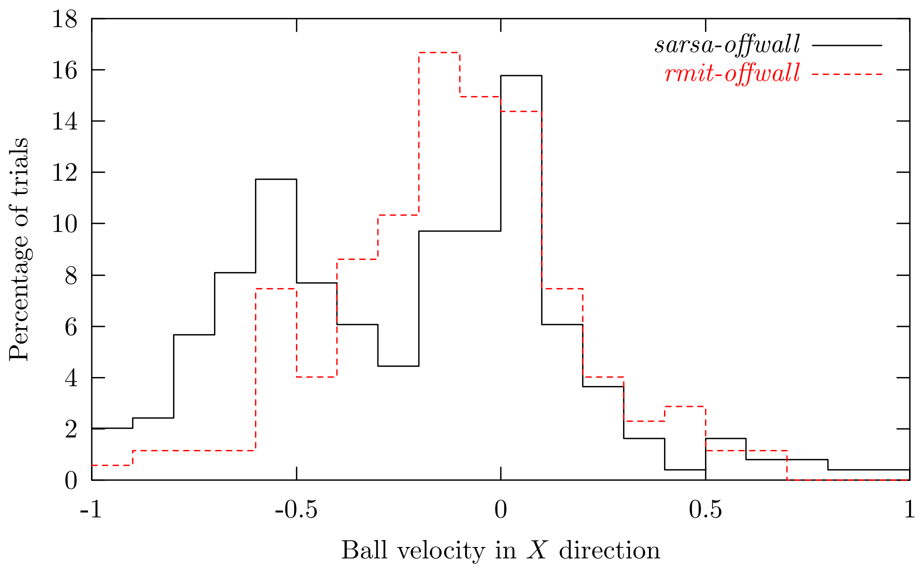

Figure 8.7: Histogram of X component of terminal ball velocity

tition at RoboCup, the CS-Freiburg team [128] also used another approach to deal with the ball when it was near the wall, and that was to kick the ball against the wall at a slight angle. It would be possible to extend the action representation to include the option to kick and extend the simulation to simulate an appropriate response from the ball.

The most obvious difference between the learnt and hand-coded behaviours for this task were to do with proximity to the wall, and avoiding, or not avoiding, stall conditions. The hand-coded behaviour has no explicit mechanism for avoiding collision with the wall, but there were some implicit mechanisms. Firstly, if the ball was between the robot and the wall, and the robot was not close enough to hit the ball, then it also would not be close enough to collide with the wall. Secondly, if the ball was close enough, the only action would be to spin around, an action that should not bring the robot closer to the wall.

The results show that stalls were commonplace for the hand-coded behaviour. One reason is that the estimate for the distance to the ball is occasionally wrong when near the wall. This is due to the fact that the colour of the ball, a bright orange, is sometimes reflected in the white gloss of the wall, making the top of the ball appear to be higher. The ball may be squeezed when pressed against the wall, making the top of the ball appear even higher. The distance to the ball is estimated based on the vertical visual angle from the camera to the top of the ball. Thus, if the top of the ball appears to be higher than it normally would be at that distance, the estimate of the distance to the ball will be larger than it should be. Another reason for so many stall conditions is that even when the distance estimates are accurate, it is simply not possible to spin with the ball between the fingers if the wall is too close. Even though the spin does not bring the robot closer to the wall, the combined ball and robot easily become wedged.

The experiments show that the hand coded behaviour was poor at avoiding stall conditions. On the other hand, the learnt behaviours did not encounter them.

From the point of view of an observer watching the movement of the robot during the trials, the hand-coded behaviour clearly caused the robot to move too close to the ball when trying to kick it away from the wall and the walls became visibly scuffed by the robot. The hand-coded behaviour also sent the ball in the wrong direction on occasion.

The learnt behaviour sometimes caused the robot to spin around when it was too far away from the ball. On the other hand, it occasionally moved slightly into the ball as it spun causing a surprisingly effective kick. In one particular incident it performed this type of kick successfully when the ball was behind it—relying on the ball position information in the world model. It was generally more reliable than the hand-coded behaviour at sending the ball in the correct direction.

8.12 Summary

In this chapter, a new behaviour to kick the ball away from the wall was developed using Sarsa(λ) under simulation. This is a challenging task as the robot needs to weigh up the benefits of moving towards the ball while avoiding getting stuck and thus stalling while near the wall. The reward function was strongly weighted towards behaviours that avoided stall conditions and this is reflected in the results for the learnt behaviour. In comparison, the hand-coded behaviour was significantly more likely to stall. In addition, the learnt behaviour also had significantly better performance in other factors such as how fast the ball was moving in the target direction at the end of the trial.

Chapter 9

Conclusions and Future Work

In this dissertation, an approach to learning real robot behaviours through reinforcement learning under simulation has been presented. This approach has been evaluated using three realistic robot behaviour tasks, and shown to produce behaviours that significantly outperformed behaviours that had been developed by hand. The three robotic tasks were not randomly chosen but were problems that were difficult to code by hand, and represent a variety of levels of difficulty. They are all from the domain of robot soccer, however the approach is not restricted to this domain and might be applied to other robot tasks.

A novel approach, called Policy Initialisation, that bootstraps the learning process was presented. This approach makes use of prior knowledge, in the form of an existing hand-coded behaviour, as a starting point for the learning process.

9.1 Research Questions

The research questions posed by this thesis are as follows:

1. Can reinforcement learning algorithms be used to develop better performing physical robot behaviours than those developed by hand?
2. If improved behaviours are learnt on a simulator, will that improvement transfer to the physical robots?
3. Can training be refined on physical robots after training on simulator?
4. Is it possible to represent the world internally to suit the requirements of a Markov Decision Process?
5. Is it possible to bootstrap the learning process by using an existing, hand-coded behaviour as a starting point? In addition, does this bias the solution found?

9.1.1 Can reinforcement learning algorithms be used to develop better performing physical robot behaviours than those developed by hand?

To answer this question, three example robotic tasks were examined: the efficient movement task, the ball turning task, and the kick the ball away from the wall task. In all three cases, significantly better performances were obtained from behaviours learnt using reinforcement learning algorithms and a simulator, than those produced using a hand-coded behaviour. The hand-coded behaviours were either ones that had been published in a refereed conference proceedings, or had been used in the RMIT-United RoboCup team, and been designed by an expert behaviour designer with several years of experience in this field. Although it is not possible to say from this that better behaviours could not be produced by hand, it does seem reasonable to suggest that these behaviours are representative of those typically produced.

In the context of the efficient movement task, it was possible to compare the learnt behaviour performance with A^* search, which is guaranteed to find the optimal behaviour. In this way, it was shown that in the simulation the learnt behaviours were close to optimal for the set of test cases.

An analysis of the differences between the learnt and hand-coded behaviours showed up several flaws in the hand-coded approaches. In the context of the efficient movement task, the hand-coded behaviours did not adjust depending on the velocity of the robot or its angular velocity. This appears to be quite critical, especially when the robot is already moving at high speed. Hand-coded behaviours tend to ignore such factors because the problem is already complicated enough. Another flaw in one of the hand-coded behaviours was not allowing for the control latency. The learnt behaviours did well in an environment with a large amount of latency even though this was not simulated. Further improvement of the learnt behaviours might be possible through simulating the latency. An approach used to combat latency was to forward predict the movement of the robot and to base the behaviour's decision on this.

In the context of the turn ball task, a noticeable feature of the hand-coded behaviour was that it was inflexible in the way that it attempted to recover the ball. That is, when the ball drifted towards the tips of the fingers, the hand-coded behaviour drove the robot straight ahead. In comparison, the learnt behaviour turned towards the ball slightly and seemed to do better at recapturing the ball when it had drifted from its grasp.

The most noticeable feature of the hand-coded behaviour for the off-wall task was a surprising tendency to hit the wall and stall. This is surprising because the behaviour seems as though it has been designed not to do this. However, the learnt behaviour was much more successful at avoiding both the wall, and a stall condition.

Several different learning algorithms were tried. The first task was small enough to allow a tabular coding of the state space. This, combined with the two different types of Monte Carlo algorithm produced significantly better results than using Sarsa(λ) with the state-action value function represented by tile coding. In addition, it did so in much less time. The two types were the Monte Carlo Exploring Starts and Monte Carlo ϵ-soft On Policy algorithms. Although this result suggested that the Monte Carlo algorithms should also be tried with other tasks, insufficient memory was available to code the larger number of dimensions required by these other tasks.

9.1.2 If improved behaviours are learnt on a simulator, will that improvement transfer to the physical robots?

In all three tasks, the learnt behaviours successfully outperformed their hand-coded equivalents under simulation and subsequently also outperformed them on the physical robot.

On the other hand, there is evidence that even if a hand-coded behaviour outperforms another hand-coded behaviour on the simulator, the latter may outperform the former on the real robot. This was the case with the *rmit-move-to* and *cmu-move-to* behaviours. This appears to be due to an aspect of the physical environment, specifically, control latency, that was not simulated.

9.1.3 Can training be refined on physical robots after training on simulator?

This question was examined in the context of the “turn ball” task. This task was seen as particularly applicable to further learning for several reasons. Firstly, the success rates, although higher for the learnt behaviour, seemed to leave room for improvement. Secondly, there were several aspects of the environment that were relevant but were not included in the simulation. One specific aspect was the way that the ball could be turned much more sharply and at greater speed if enough of its surface area was touching the fingers of the robot. This occurred because the ball would then skid on the carpet, rather than rolling.

The overall result was that using the physical robot to refine behaviours learnt on the simulator did not significantly improve performance. In fact, there was a significant degradation in performance during the learning process, probably due to the exploration of different actions that occurs during learning.

The lack of improvement is slightly surprising, as intuition would suggest that any learning on the real robot would improve the behaviour’s ability to handle the actual situation. There are two difficulties with this, however. The first is that the number of trials required to converge on a solution under simulation—of the order of hundreds of thousands of trials for the turn ball task—suggests that many more real robot trials would be needed than were attempted here. The second problem is that the episode lengths tended to be longer on the real robot than on the simulator. This has a corresponding effect on the reward and means that all estimates of the action-value will need to be adjusted.

9.1.4 Is it possible to represent the world internally to suit the requirements of a Markov Decision Process?

In all three cases studied, it was possible to gather enough information from the sensors to form a state signal with the Markov property that fully observed the environment.

When defining a problem as a Markov Decision Process, a necessary step is to ensure that the state signal is Markovian, or in other words, that knowledge of past states makes no difference in the agent’s ability to predict what will happen next. In addition, the state should be fully observable, or in other words, the agent should be able to unambiguously identify the state of the world from the state signal. These two issues are implicit in much of the current work in the field of reinforcement learning, although they are not always considered to be required factors. However, they have not been considered at all when looking at behaviour-based robots. Instead, hand-coded behaviours tend to overlook aspects of the state that would be considered relevant when the problem is phrased as an MDP. The experience here has been that these are precisely the aspects that were needed to produce better behaviours. This suggests that a significant part of the improvements over hand-coded behaviours seen in this thesis are to do with including all the relevant aspects of the state to ensure that the state has the Markov property and corresponds to a fully observable environment.

9.1.5 Is it possible to bootstrap the learning process by using an existing, hand-coded behaviour as a starting point? In addition, does this bias the solution found?

The possibility of bootstrapping the learning algorithm with an existing, hand-coded behaviour was examined in the context of the efficient movement task. It was demonstrated that the Policy Initialisation modification to the standard Monte Carlo learning algorithm produces a better test reward more quickly than if Policy Initialisation is not used. The result obtained for the efficient movement task was that using Policy Initialisation was roughly equivalent to skipping the first 2 to 4 million learning trials.

A similar approach was tried with Sarsa(λ) and the tile coding function approximation. Four different algorithms were tried. The most successful approach was to use Monte Carlo ES to initialise the values based on the initial policy. However, even this did not compare well with just using Sarsa(λ) by itself.

When initialising in this way, the question arises whether the solution will be limited or biased by the form of the initial policy. In the example examined, the answer appears to be that it will not. The main evidence for this is that the long term test performance is the same regardless of whether policy initialisation is used. In addition, when looking at some parts of the policy, it was noticeable that radical improvements could be made. Specifically, in many cases where the hand-coded algorithm for efficient movement preferred to go forward, reverse motion was more optimal, and these cases were changed to reverse motion in the learnt behaviour.

9.2 Future Work

This thesis has developed several high performance behaviours for a RoboCup robot and this work could be further verified by applying it to the other robot soccer behaviours. For example, the movement of the goalie robot is critical and also quite subtle. At the most basic level, the goalie simply places itself between the goal and the ball. However, more sophisticated behaviour is possible when the position of other robots is taken into account. In particular, there is a difficult trade-off to make between staying close to the goal and rushing out to meet the attacker. Sophisticated tasks such as that of the goalie can be difficult to express as Markov Decision Processes because of the large number of variables that need to be considered.

The efficient movement task addressed movement without obstacles. Future work will need to verify that the approach used in this thesis can be extended toward situations where the robot must avoid obstacles when moving to a target location. A possible approach to restricting the size of the state space is to use an approach developed by Bowling [23] that only considers the nearest obstacle. A more sophisticated approach would be to use a high-level path planner, such as that used by CS-Freiburg [51], to plan paths around obstacles and to provide the low-level movement behaviour with the next intermediate target point.

Some tasks in robot soccer are usually specified in a way that implies that the state must be partially observable. For example, the RMIT-United team used a *find-ball* behaviour that was employed when the position of the ball was unknown. In future work, this work will be extended to such problems. One possible approach that could be used is to convert the partially observable problem into a fully observable one. For example, rather than consider the ball position to be unknown, CS-Freiburg [51] used a probability grid for the position of the ball over the whole field. The *find-ball* behaviour can then be converted to a behaviour that moves to the ball's most probable position based on previous sightings.

In this work, the tabular Monte Carlo method algorithms were only attempted on the smallest problem, since memory was limited and this limits the maximum number of states. However, the tabular Monte Carlo algorithms produced the best performing behaviour most quickly when they were used. Tabular algorithms can still be applied to large state spaces but a more coarse discretisation may be required. Future work will examine whether tabular Monte Carlo methods can be applied to problems with larger state spaces without sacrificing performance.

All of the work here was performed using one robot. In future work, robots of the same type and other types of robots will be used to verify that this work is not specific to any particular robot or type of robot. It seems likely that it may be necessary to learn the behaviours again on a different simulator if the class of robot changes significantly.

An additional avenue of future research is to examine the effect of further refinement of the simulation. The simulator used in this work is only an approximate one. For example, the simulation did not accurately predict the number of steps that would be taken by the real robot when moving to a target location. It seems likely that improvements in the realism of the simulator will make corresponding improvements in the behaviours developed. Therefore, an interesting question to be explored in future work is whether this correspondence holds and whether there are large improvements in the performance of the behaviour to be made.

Another issue is the decomposition of large tasks into ones that are small enough to learn as individual behaviours. Although several automatic hierarchical task decomposition approaches exist (see section 2.4.10 on page 32), none currently make use of the symmetries that can occur in real, two wheel robot tasks. This thesis has presented an approach for manually decomposing tasks based on symmetrical aspects of the task. Future work will look at exploring ways of automating this process.

Furthermore, automatic task decomposition typically requires transition states, or groups of states, where the robot transitions from using one behaviour to using another. Wiering and Schmidhuber [129] refer to these as sub-goal states. In this work, hysteresis was used to avoid small changes in sensory information causing oscillation in the selection of the current behaviour. Hysteresis thresholds are currently selected manually. As part of an automatic task decomposition, future work will investigate automatic selection of the hysteresis thresholds.

In this work, a method was developed that produced better performing behaviours than produced by hand. However, in comparison to the hand-coded behaviours, the learnt ones tend to be memory intensive. This is because they make use of a large, unsummarised map of states to actions, or state-action pairs to values. In future work, one aim will be to substantially reduce the memory requirement of these maps, preferably without reducing performance. A possible approach is to use some further machine learning technique such as genetic programming to summarise the policy once it has been found. An advantage of genetic programming is that the result of learning is a program and may thus be more directly compared with hand-coded behaviours. This would also need to be compared with using different types of function approximation techniques during the reinforcement learning process.

Another aim in future work will be to examine the problem of continued learning on the robot. Continued learning offers the potential benefit of overcoming limitations in the simulator, and allowing the behaviour to tailor itself to its environment. Although some attempt was made to do this in this thesis, no substantial improvement was seen. This seems to reflect the idea that the policy transfers more readily than the action-value function. If this is the case, some variant of the approaches used to initialise learning from a policy may provide a method to combine the benefits of both simulated learning and continuing that learning on the robot.

If the above approach is successful, it may provide some solution to the problem of learning in cases where the robot has dynamic instability. For example, many current legged robots attempt to solve the problem of walking by avoiding any instability. The difficulty with attempting to learn how to walk using a real robot is that it is a complex task and it can be expected to require a large number of trials to train. In addition, as with the ball turning tasks, many trials will tend to be short, and provide little useful feedback. A possible solution is to make use of a simulator. However it also seems likely that simulator-based learning will not produce a policy sufficiently tailored to the robots characteristics. If an approach can be developed to allow continued learning on the physical robot, using the simulator-based policy as a starting point, it may be possible to develop good policies for difficult problems such as walking with dynamic instability.

New application areas for autonomous robotics, such as home entertainment and intelligent manufacturing, are emerging. These areas demand robustness and reliability, but performance is also a key issue. The techniques developed in this thesis are quite general and should allow behaviour-based performance to be improved in a range of autonomous robotic application areas.

9.3 Summary

In conclusion, this work has demonstrated the design of an internal world model to meet the requirements of a Markov Decision Process. It has evaluated a new technique to seed reinforcement learning algorithms from an existing behaviour. Furthermore, this work has demonstrated that significant performance improvements over hand-coded behaviours are possible through the application of reinforcement learning in a simulated environment. Most significantly, it has shown that the improved performance on the simulator carries through to improved performance on the real robot. Finally, the strength of the results indicate that this approach will make real improvements in the way autonomous robots are developed.

Appendix A

Robot Motion Simulation

A simplistic model of the motion of the robot is used in this thesis. The main simplification is to assume that no skidding occurs during motion, other than that to perform normal turns. This allows the position of the robot to be integrated from changes in the wheel positions. Based on work by The equations giving the rate of change of position over time are,

$$\begin{aligned}\dot{x} &= v\cos\theta,\\ \dot{y} &= v\sin\theta,\end{aligned}$$

where v is the velocity of the centre of the robot in the forward direction, θ is the heading, (x, y) is the position of the robot, $\dot{x}$ is the rate of change of position in the x direction, and $\dot{y}$ is the rate of change of position in the y direction. The velocity can be calculated from the wheel speeds, and the heading can be integrated from the rate of change of heading, which can also be derived from the wheel speeds, as follows:

$$\begin{aligned}\dot{\theta} &= \frac{r}{2S}\left(\omega_R - \omega_L\right),\\ v &= \frac{r}{2}\left(\omega_R + \omega_L\right),\end{aligned}$$

where ω_L and ω_R are the angular velocities of the left and right wheels, respectively, r is the radius of the wheel, and S is the distance between the two wheels. Position integration is approximated numerically by

$$\begin{aligned}x_{t+1} &= x_t + \dot{x}_{t+1},\\ y_{t+1} &= y_t + \dot{y}_{t+1},\\ \theta_{t+1} &= \theta_t + \dot{\theta}_{t+1}.\end{aligned}$$

At this stage, it is possible to calculate the updated position of the robot given an initial position vector $(x, y, \theta)^T$ and angular velocity of the wheels ω_L, ω_R over time.

This leaves the question of simulating the movement of the wheels. To keep things simple, and for the purpose of estimating their velocity, the wheels are simulated as though they were independent, and moving along a linear track. In the physical robot, the power sent to each wheel is controlled by a PID (proportional-integral-derivative) controller, based on a target velocity. This is simulated here as a simple proportional controller, and the power considered to correspond to the resulting acceleration. That is,

$$\dot{\omega}_L = \text{bound}\left[\frac{u_L}{r} - \omega_L, \dot{\omega}_{\min}, \dot{\omega}_{\max}\right],$$

where u_L is the desired velocity for the left side of the robot. This is converted to a desired wheel angular velocity by dividing by the radius of the wheel. The difference between this and the current wheel velocity is then bounded. An equivalent calculation is performed to find the right wheel acceleration.

The acceleration is bounded to simulate the response of an electric motor. Specifically, electric motors produce maximum torque when stopped, which reduces to zero at the peak velocity of the motor, at which point, adding more power does not increase the speed of the motor. This is simulated by bounding the wheel acceleration $\dot{\omega}$ according to a maximum $\dot{\omega}_{\max}$ and minimum $\dot{\omega}_{\min}$, that are defined by,

$$\begin{aligned} \dot{\omega}_{\max} &= \begin{cases} k_1\left(1 - \frac{\omega}{k_2}\right) & \text{where } \omega \geq 0, \\ k_1 & \text{otherwise,} \end{cases} \\ \dot{\omega}_{\min} &= \begin{cases} -k_1 & \text{where } \omega \geq 0, \\ -k_1\left(1 + \frac{\omega}{k_2}\right) & \text{otherwise,} \end{cases} \end{aligned}$$

where k_1 is the nominal maximum wheel acceleration, and k_2 is the nominal maximum wheel velocity. The equations are split into two parts, since forward acceleration can be at a maximum when the wheel is reversing, regardless of the speed, and similarly for reverse acceleration.

There are several forms of friction that affect the movement of the robot. For example, there is rolling resistance for each wheel, and also friction acting on each skid pad. These varied forms of friction are simulated as a single constant friction force that acts on each wheel against the direction of motion of that wheel. Friction can be treated as a constant force with the strict exception of when the wheel is stopped. To handle this exception, the time when the wheel will stop is estimated, and if it occurs during the current simulation cycle, that cycle is broken up according to when the wheel stops. This approach also allows the simulation to correctly handle the situation when a wheel transitions from forward to reverse movement or visa versa.

Fox [43] gives the motion equations for a synchro-drive robot. Fox assumes that the acceleration $\dot{v}$ and rate of change of heading $\dot{\theta}$ remain constant between commands. However, the robot used here controls the wheel speeds using "set points" or target wheel velocities. This means that the low-level controller will adjust the amount of power to each wheel several times per cycle to try to achieve the desired velocities. For this reason, the overall acceleration can change during a cycle, and the rate of change of heading can also change. For the purposes of simulation, however, the wheel accelerations are assumed not to change over a cycle. Note that a limitation of this assumption is that it may allow the wheel speed to exceed the maximum possible.

Bibliography

[1] Giovanni Adorni, Stefano Cagnoni, Stefan Enderle, Gerhard K. Kraetzschmar, Monica Mordonini, Michael Plagge, Marcus Ritter, Stefan Sablatnög, and Andreas Zell. Vision-based localization for mobile robots. *Robotics and Autonomous Systems*, 36:103–119, 2001.

[2] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, Los Altos, California, 1987. Morgan Kaufmann.

[3] J. S. Albus. Data storage in the cerebellar model articulation controller (CMAC). *Transactions of the ASME: Journal of Dynamic Systems, Measurement and Control*, 97(3):228–233, September 1975.

[4] J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Transactions of the ASME: Journal of Dynamic Systems, Measurement and Control*, 97(3):220–227, September 1975.

[5] Ronald C. Arkin. Towards the unification of navigational planning and reactive control. In *Working Notes of the AAAI Spring Symposium on Robot Navigation*. Stanford University, March 1989.

[6] Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[7] Minoru Asada and Hiroaki Kitano, editors. *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of *Lecture Notes in Computer Science*. Springer, 1999.

[8] Minoru Asada, Hiroaki Kitano, Itsuki Noda, and Manuela Veloso, editors. *RoboCup: Today and tomorrow—What we have learned*, volume 110, 1999.

[9] Minoru Asada, Shoichi Noda, and Koh Hosoda. Non-physical intervention in robot learning based on LfE method. In *Proceedings of Machine Learning Conference Workshop on Learning from Examples vs. Programming by Demonstration*, pages 25–31, 1995.

[10] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Vision-based behavior acquisition for a shooting robot by using a reinforcement learning. In *Proceedings of IAPR / IEEE Workshop on Visual Behaviors*, pages 112–118, 1994.

[11] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23(2-3):279–303, 1996.

[12] Minoru Asada, Eiji Uchibe, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Coordination of multiple behaviors acquired by a vision-based reinforcement learning. In *Proceedings of IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, Munich, Germany, September 1994.

[13] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis and S. Russell, editors, *Proceedings of the 12th International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[14] Tucker Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, Georgia Institute of Technology, 1998.

[15] A. Balluchi, A. Bicchi, A. Balestrino, and G. Casalino. Path tracking control for Dubin's cars. In *Proceedings of the 1996 IEEE International Converence on Robotics and Automation (ICRA-96)*, pages 3123–3128, Minneapolis, MN, 1996.

[16] Jacky Baltes and Yuming Lin. Path-tracking control of non-holonomic car-like robot with reinforcement learning. In Veloso et al. [123], pages 162–173.

[17] Thorsten Belker, Michael Beetz, and Armin B. Cremers. Learning action models for the improved execution of navigation plans. *Robotics and Autonomous Systems*, 38:137–148, 2002.

[18] Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors. *The RoboCup 2001 International Symposium*, Berlin, August 2001. Springer-Verlag.

[19] Alan D. Blair and Elizabeth Sklar. The evolution of subtle manoeuvres in simulated hockey. In *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behaviour (SAB '98)*, Zurich, Switzerland, August 1998. Bradford.

[20] Alan D. Blair and Elizabeth Sklar. Exploring evolutionary learning in a simulated hockey environment. In *Congress on Evolutionary Computation*, pages 197–203, 1999.

[21] Vincent D. Blondel and John N. Tsitsiklis. A survey of computational complexity results in systems and control. *Automatica*, 36(9):1249–1274, 2000.

[22] Michael Bowling. Motion control in CMUnited-98. In Manuela M. Veloso, editor, *Third International Workshop on RoboCup*, pages 52–56, Stockholm, Sweden, August 1999.

[23] Michael Bowling and Manuela Veloso. Motion control in dynamic multi-robot environments. In Veloso et al. [123].

[24] R. A. Brooks. From earwigs to humans. *Robotics and Autonomous Systems*, 20(2–4):291–304, June 1997.

[25] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Roboics and Automation*, 2(1):14–23, 1986.

[26] Rodney A. Brooks. The behavior language; user's guide. Technical Report 1227, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1990.

[27] Rodney A. Brooks. Elephants don't play chess. In Maes [85], pages 3–15.

[28] Rodney A. Brooks. Intelligence without reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.

[29] Rodney A. Brooks. Artificial life and real robots. In Francisco J. Varela and Paul Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 3–10, Cambridge, MA, 1992. MIT Press.

[30] James Brusey, Mark Makies, Lin Padgham, Brad Woodvine, and Karl Fantone. RMIT United. In Stone et al. [109], pages 563–566.

[31] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1023–1028, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[32] Claudio Castelpietra, Luca Iocchi, Daniele Nardi, Maurizio Piaggio, Alessandro Scalzo, and Antonio Sgorbissa. Communication and coordination among heterogeneous mid-size players: ART99. In Stone et al. [109], pages 86–95.

[33] C. H. Chen, L. F. Pau, and P. S. P. Wang, editors. *Handbook of Pattern Recognition & Computer Vision.* World Scientific, 2nd edition, 1999.

[34] Jonathan H. Connell. *Minimalist Mobile Robotics: a colony-style architecture for an artificial creature.* Academic Press, 1990. Revision of author's thesis (Ph.D.—Massachusetts Institute of Technology, 1989).

[35] James L. Crowley and Patrick Reignier. Asynchronous control of rotation and translation for a robot vehicle. *Robotics and Autonomous Systems*, 10:243–251, February 1992.

[36] Markus Dietl, Jens-Steffen Gutmann, and Bernhard Nebel. CS Freiburg: Global view by cooperative sensing. In Birk et al. [18].

[37] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, pages 118–126. Morgan Kaufmann, San Francisco, CA, 1998.

[38] Marco Dorigo and Marco Colombetti. *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press, 1998.

[39] L. E. Dubins. On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.

[40] A. Elnagar and A. Hussein. On optimal constrained trajectory planning in 3d environments. *Robotics and Autonomous Systems*, 33:195–206, 2000.

[41] R. James Firby. Modularity issues in reactive planning. In Brian Drabble, editor, *Proceedings of the Third International Conference on AI Planning Systems, AIPS-96*, pages 78–85, Edinburgh Scotland, May 1996. AAAI Press.

[42] Dario Floreano and Francesco Mondada. Evolution of homing navigation in a real mobile robot. *IEEE Trans. on Systems, Man and Cybernetics—Part B: Cybernetics*, 26(3):396–407, June 1996.

[43] Dieter Fox. *Markov Localization: A Probabilistic Framework for Mobile Robot Localization and Navigation*. PhD thesis, Institute of Computer Science, University of Bonn, Germany, 1998.

[44] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI/IAAI*, pages 343–349, 1999.

[45] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation*, 4(1):23–33, 1997.

[46] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[47] Dieter Fox, Wolfram Burgard, Sebastian Thrun, and Armin B. Cremers. Position estimation for mobile robots in dynamic environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 983–988, Madison, WI, July 1998.

[48] M. Fujita, S. Zrehen, and H. Kitano. A quadruped robot for robocup legged robot challenge in paris '98. In Asada and Kitano [7], pages 125–140.

[49] Erann Gat. Three-layer architectures. In Kortenkamp et al. [73].

[50] Geoffrey J. Gordon. Reinforcement learning with function approximation converges to a region. In *NIPS*, pages 1040–1046, 2000.

[51] Jens-Steffen Gutmann, Wolfgang Hatzack, Immanuel Herrmann, Bernhard Nebel, Frank Rittinger, Augustinus Topor, and Thilo Weigel. The CS freiburg team: Playing robotic soccer based on an explicit world model. *AI Magazine*, 21(1):37–46, 2000.

[52] Jens-Steffen Gutmann, Wolfgang Hatzack, Immanuel Herrmann, Bernhard Nebel, Frank Rittinger, Augustinus Topor, Thilo Weigel, and Bruno Welsch. The CS Freiburg robotic soccer team: Reliable self-localization, multirobot sensor integration, and basic soccer skills. In Asada and Kitano [7], pages 93–108.

[53] Jens-Steffen Gutmann, Thilo Weigel, and Bernhard Nebel. Fast, accurate, and robust self-localization in the robocup environment. In Veloso et al. [123], pages 304–317.

[54] Kwun Han and Manuela Veloso. Physical model based multi-objects tracking and prediction in robosoccer. In *Working Note of the AAAI 1997 Fall Symposium*. MIT Press, 1997.

[55] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence*, 1998.

[56] Bernhard Hengst. Generating hierarchical structure in reinforcement learning from state variables. In *Proceedings of The Sixth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000)*, 2000.

[57] Kazuo Hiraki, Akio Sashima, and Steven Phillips. From egocentric to allocentric spatial behavior: A computational model of spatial development. *Adaptive Behaviour*, 6(3/4):371–391, 1998.

[58] John Hoar, Jeremy Wyatt, and Gillian Hayes. Multiple evaluation techniques for robot learning. In *Proceedings 10th International Florida AI research symposium (FLAIRS-97)*, Florida, May 1997.

[59] G. S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo. In *IEEE International Conference on Robotics and Automation*, pages 3040–3045, 2000.

[60] Andrew Howard and Les Kitchen. Cooperative localisation and mapping. In *Proceedings of the 1999 International Conference on Field and Service Robotics*, Pittsburgh, PA, August 1999.

[61] Giovanni Indiveri. On the motion control of nonholonomic soccer playing robot. In Birk et al. [18].

[62] Luca Iocchi and Daniele Nardi. Hough localization for mobile robots in polygonal environments. *Robotics and Autonomous Systems*, 40:43–58, 2002.

[63] Nick Jakobi. The minimal simulation approach to evolutionary robotics. In T. Gomi, editor, *Evolutionary Robotics - From Intelligent Robots to Artificial Life (ER'98)*, Ontario, Canada, 1998. AAI Books.

[64] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life: Proc. 3rd European Conference on Artificial Life*, volume 929, pages 704–720. Springer-Verlag, 1995.

[65] M. Jamzad, A. Foroughnassiraei, E. Chiniforooshan, R. Ghorbani, M. Kazemi, H. Chitsaz, F. Mobasser, and S. Sadjad. Arvand: a soccer player robot. *AI Magazine*, 21(3):47–51, Autumn 2000.

[66] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[67] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[68] Zsolt Kalmár, Csaba Szepesvári, and András Lörincz. Module-based reinforcement learning: Experiments with a real robot. *Machine Learning*, 31(1–3):55–85, April 1997.

[69] H. Kitano, editor. *RoboCup-97: The First Robot World Cup Soccer Games and Converences*, Berlin, 1998. Springer-Verlag.

[70] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 340–347, New York, 5–8, 1997. ACM Press.

[71] S. Koenig and R. G. Simmons. Xavier: A robot navigation architecture based on partially observable markov decision process models. In Kortenkamp et al. [73].

[72] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The Saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 9(1):215–235, 1997.

[73] David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems.* AAAI Press / The MIT Press, 1998.

[74] Pier Luca Lanzi. Adaptive agents with reinforcement learning and internal memory. In Meyer et al. [95], pages 333–342.

[75] David M. Levine, Patricia P. Ramsey, and Robert K. Smidt. *Applied statistics for engineers and scientists : using Microsoft Excel and Minitab.* Prentice-Hall International, 2001.

[76] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of AAAI-91*, pages 781–786, 1991.

[77] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[78] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, San Francisco, CA, 1995. Morgan Kaufmann.

[79] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 394–402, Montreal, Québec, Canada, 1995.

[80] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence and the design of expert systems*, chapter Heuristic Search. Benjamin/Cummings Publishing, 1989.

[81] H. Maaref and C. Barret. Sensor-based navigation of a mobile robot in an indoor environment. *Robotics and Autonomous Systems*, 38:1–18, 2002.

[82] Richard Maclin and Jude W. Shavlik. Incorporating advice into agents that learn from reinforcements. In *National Conference on Artificial Intelligence*, pages 694–699, 1994.

[83] Pattie Maes. The dynamics of action selection. In *AAAI Spring Symposium on AI Limited Rationality, IJCAI*, pages 991–997, Detroit, MI, 1989.

[84] Pattie Maes. How to do the right thing. Technical Report NE 43-836, AI Laboratory, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, USA, 1989.

[85] Pattie Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back.* The MIT Press, Cambridge, MA, 1991.

[86] Pattie Maes. Situated agents can have goals. In *Designing Autonomous Agents* [85], pages 49–70.

[87] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviors. In *Proceedings of AAAI-90: The American Conference on Artificial Intelligence*, pages 796–802, Boston, MA, August 1990. AAAI Press / MIT Press.

[88] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learing. *Artificial Intelligence*, 55(2–3):311–365, June 1992.

[89] Sridhar Mahadevan, Nikfar Khaleeli, and Nicholas Marchalleck. Designing agent controllers using discrete-event markov models. In *AAAI Fall Symposium on Model-Directed Autonomous Systems*, Cambridge, November 1997. MIT.

[90] Frederic Maire and Doug Taylor. A quadratic programming formulation of a moving ball interception and shooting behaviour, and its application to neural network control. In Stone et al. [109], pages 327–332.

[91] Maja Matarić and Dave Cliff. Challenges in evolving controllers for physical robots. *Robotics and Autonomous Systems*, 19(1):67–83, 1996.

[92] Maja J Matarić. Behavior-based control: Main properties and implications. In *Proceedings of the IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pages 46–54, Nice, France, May 1992.

[93] Maja J. Matarić. Reward functions for accelerated learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 181–189, San Francisco, CA, 1994. Morgan Kaufman.

[94] Maja J. Matarić. Reinforcement learing in the multi-robot domain. *Autonomous Robots*, 1(4):73–83, March 1997.

[95] Jean-Arcady Meyer, Alain Berthoz, Dario Floreano, Herbert Roitblat, and Stewart W. Wilson, editors. *Proceedings of the Sixth International Conference on Simulation of Adaptive Behaviour (SAB2000).* MIT Press, 2000.

[96] Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995.

[97] José del R. Millán. Rapid, safe, and incremental learning of navigation strategies. *IEEE Trans. on Systems, Man and Cybernetics—Part B: Cybernetics*, 26(3):408–420, June 1996.

[98] Francesco Mondada, Edoardo Franzi, and Paolo Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Experimental Robotics III, Proceedings of the 3rd International Symposium on Experimental Robotics*, pages 501–513, Kyoto, Japan, October 1994.

[99] Jörg P. Müller. Control architectures for autonomous and interacting agents: A survey. In Lawrence Cavedon, Anand S. Rao, and Wayne Wobcke, editors, *PRICAI Workshop on Intelligent Agent Systems*, volume 1209 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 1996.

[100] Robin R. Murphy. *Introduction to AI Robotics.* MIT Press, Cambridge, MA, 2000.

[101] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[102] M. Riedmiller, A. Merke, D. Meier, A. Hoffmann, A. Sinner, O. Thate, Ch. Kill, and R. Ehrmann. Karlsruhe brainstormers - A reinforcement learning approach to robotic soccer. In Stone et al. [109], pages 367–372.

[103] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report 166, Cambridge University Engineering Department, September 1994.

[104] Juan C. Santamaría, Richard S. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behaviour*, 6(2):163–217, 1998.

[105] Satinder P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, Menlo Park, 1992. AAAI Press.

[106] Satinder P. Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[107] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996.

[108] Peter Stone. Layered learning in multiagent systems. In *AAAI/IAAI*, page 819, 1997.

[109] Peter Stone, Tucker R. Balch, and Gerhard K. Kraetzschmar, editors. *RoboCup 2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Computer Science.* Springer, 2001.

[110] Peter Stone and Richard S. Sutton. Scaling reinforcement learning toward robocup soccer. In *Proceedings of the 18th International Conference on Machine Learning (ICML-2001)*, 2001.

[111] Peter Stone, Richard S. Sutton, and Satinder Singh. Reinforcement learning for 3 vs. 2 keepaway. In Stone et al. [109].

[112] Peter Stone and Manuela Veloso. Using decision tree confidence factors for multi-agent control. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 86–91, New York, 9–13, 1998. ACM Press.

[113] Peter Stone and Manuela Veloso. Using decision tree confidence factors for multiagent control. In Kitano [69], pages 31–36.

[114] Ron Sun and Todd Peterson. Partitioning in multi-agent reinforcement learning. In Meyer et al. [95], pages 325–332.

[115] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[116] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh Int. Conf. on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[117] Richard S. Sutton. Open theoretical questions in reinforcement learning. *Lecture Notes in Computer Science*, 1572:11–17, 1999.

[118] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, Massachusetts, 1998.

[119] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2):99–141, 2001.

[120] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42:674–690, 1997.

[121] Eiji Uchibe, Minoru Asada, and Koh Hosoda. Behavior coordination for a mobile robot using modular reinforcement learning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems 1996 (IROS '96)*, pages 1329–1336, 1996.

[122] Richard Vaughan, Neil Sumpter, Jane Henderson, Andy Frost, and Stephen Cameron. Experiments in automatic flock control. In *Proceedings of the Sixth Symposium on Intelligent Robotic Systems (SIRS '98)*, 1998.

[123] M. Veloso, E. Pagello, and H. Kitano, editors. *RoboCup-99: Robot Soccer World Cup III*, Berlin, 2000. Springer-Verlag.

[124] Manuela Veloso, Michael Bowling, Sorin Achim, Kwun Han, and Peter Stone. The CMUnited-98 champion small-robot team. In Asada and Kitano [7], pages 77–92.

[125] Manuela Veloso, Peter Stone, Kwun Han, and Sorin Achim. The CMUnited-97 small robot team. In Kitano [69], pages 242–256.

[126] W. G. Walter. *The Living Brain.* Norton, New York, 1953 (reprinted 1963).

[127] C. J. C. H. Watkins. *Learning with delayed rewards.* PhD thesis, University of Cambridge, 1989.

[128] Thilo Weigel, Willi Auerbach, Markus Dietl, Burkhard Dümler, Jens-Steffen Gutmann, Kornel Marko, Klaus Müller, Bernhard Nebel, Boris Szerbakowski, and Maximilian Thiel. CS Freiburg: Doing the right thing in a group. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer. World Cup IV*, volume 2019 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2001.

[129] Marco Wiering and Jürgen Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

[130] Jeremy Wyatt, John Hoar, and Gillian Hayes. Design, analysis and comparison of robot learners. *Robotics and Autonomous Systems*, 1–2(24), 1998.

[131] Y. Zheng, Z. Z. Han, P. R. Moore, and Q.-H. Max Meng. The design of time-optimal control for two-wheel driven carts catching a target trajectory. *Mechanism and Machine Theory*, 34:877–888, 1999.