THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Reinforcement Learning for Racecar Control

Ben Cleland

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

March 2005 - March 2006

"The only true wisdom is in knowing you know nothing." Socrates

Abstract

This thesis investigates the use of reinforcement learning to learn to drive a racecar in the simulated environment of the Robot Automobile Racing Simulator. Real-life race driving is known to be difficult for humans, and expert human drivers use complex sequences of actions. There are a large number of variables, some of which change stochastically and all of which may affect the outcome. This makes “driving” a promising domain for testing and developing Machine Learning techniques that have the potential to be robust enough to work in the real world. Therefore the principles of the algorithms from this work may be applicable to a range of problems.

The investigation starts by finding a suitable data structure to represent the information learnt. This is tested using supervised learning. Reinforcement learning is added and roughly tuned, and the supervised learning is then removed. A simple tabular representation is found satisfactory, and this avoids difficulties with more complex methods and allows the investigation to concentrate on the essentials of learning. Various reward sources are tested and a combination of three are found to produce the best performance. Exploration of the problem space is investigated. Results show exploration is essential but controlling how much is done is also important. It turns out the learning episodes need to be very long and because of this the task needs to be treated as continuous by using discounting to limit the size of the variables stored. Eligibility traces are used with success to make the learning more efficient. The tabular representation is made more compact by hashing and more accurate by using smaller buckets. This slows the learning but produces better driving. The improvement given by a rough form of generalisation indicates the replacement of the tabular method by a function approximator is warranted. These results show reinforcement learning can work within the Robot Automobile Racing Simulator, and lay the foundations for building a more efficient and competitive agent.

Acknowledgements

To my main supervisor Dr Tony Smith, for the original idea of using a racecar simulator; for lots of lively dialogue and debate; for your enthusiasm; for believing in me even when I did not; for patience and understanding; and for sticking by me. During the dread write-up process you stated you were the devil's advocate, and now I know what you meant! To my other main supervisor Dr Bernhard Pfahringer for many long discussions and for looking after me when Tony was overseas, and also for providing financial support from your own research account. I discovered another of your talents when I lost against you in an arm wrestle! And to Dr Kurt Driessens, our visiting post-doc and local expert on reinforcement learning, for setting me straight on a number of important theoretical matters. I wish I had pestered you more while you were here, so you were fortunate. Your skill at drawing diagrams may be appalling but you can certainly think clearly!

Special thanks to my long suffering Mum.

An apology to Richard Sutton: The term "Q-value"[1] has been used frequently in this work! However, the more helpful terms "action value", "state-action value", "expected-sum-of-future-rewards" and even "sum of rewards historically-available/expected from a given state-action-pair" are also used from time to time throughout, as an equivalent.

[1] Sutton's compelling disparagement of the term Q-value is at: http://www.cs.ualberta.ca/~sutton/RL-FAQ.html#Q-value

Contents

List of Figures

List of Tables

1 Introduction

This thesis investigates the use of reinforcement learning to learn to drive a racecar in a simulated environment. The aim is to use Q-learning to learn to drive a robot racecar at a competitive pace in the simulated environment provided by the Robot Automobile Racing Simulator (RARS) [Timin, 1995]. Driving a car involves taking a set of sensory inputs and producing some control outputs. Driving at minimum lap time involves an optimisation of this process; and reinforcement learning provides a mechanism to achieve this automatically.

1.1 Context

The ultimate aim of artificial intelligence is to match or exceed the intelligence of humans. Whether or not this is even theoretically possible is a matter of long debate. A widely agreed definition of the term "intelligence" is also elusive. For a discussion of this issue see the introduction and conclusion of any artificial intelligence text (e.g. [Russell and Norvig, 2003]).

However, some aspects of human intellect are easier to define, and would be clearly useful to imitate. One of these is the act of learning. It would be very useful for a machine to be able to learn in a manner similar to human learning. Such a machine could be placed in environments too dangerous for humans; or too difficult for humans; or set on tasks that no human has yet mastered. Various types of learning have been used during the history of artificial intelligence. One method by which a machine can learn is by interacting with its environment, observing the effect of its actions and discovering how to alter its actions to achieve some desired outcome. "Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence." [Sutton and Barto, 1998]. This is the idea behind Reinforcement Learning. A rough historical context of reinforcement learning follows.

During the 1970s and 1980s "expert systems" were popular and are still used today. Expert systems encode the detailed knowledge of a human expert in some narrow domain. They typically include the "fuzzy" or "intuitive" knowledge gained by experts after long experience. These systems are rule-based, painstaking to construct, require a large amount of the expert's time during construction and need updating as the (human) expert's knowledge grows. These systems make the expert's skills much more widely available. But they can not discover new knowledge; although there may be some "clean-up" effect through generalisation (similar to interpolation and extrapolation in statistics).

The bottleneck in developing an expert system has proved to be the knowledge acquisition phase. Machine learning techniques relieve this bottleneck to a large extent. In general, machine learning requires a supply of correctly classified examples—for example, a situation along with the correct action to take, as judged by an expert. During the learning phase the machine learning algorithm automatically finds patterns in the input/output relationship and from this it (usually) forms a model. The model can then be applied to novel inputs (i.e. unclassified examples) to yield the correct outputs.

Machine learning systems are much easier to build than are expert systems, in that: the expert needs only to supply correctly classified positive and negative examples; the expert's intuition is implicitly encoded; and the expert does not need to supply the reasoning behind the classifications (as is usually needed for setting up an expert system). This particular form of machine learning is known as "supervised learning".

In some problem domains there is no expert; or the best "expert" is not very accomplished; or it is not certain that the best solution known is the actual optimum. In these sorts of situations a learning mechanism is needed that is able to discover new facts. One such method is Reinforcement Learning, a sub-branch of machine learning. Reinforcement Learning combines ideas from the much earlier work of Dynamic Programming (which was developed under the study of Control Theory) and Monte Carlo methods (from Statistics) [Sutton and Barto, 1998]. Evolutionary Methods (e.g. Genetic Algorithms) can also be viewed as a type of reinforcement learning [Thrun and Littman, 2000]. Reinforcement learning, within machine learning, is related to the concept of reinforcement learning in psychology.

Reinforcement learning involves learning by trial and error by interacting with the environment in the domain of interest. The environment must provide feedback (called a "reinforcement" or "reward") and this may be delayed from the actions responsible (e.g. a simple "win" or "lose" at the end of a game). Reinforcement learning is goal-directed, where the goal is to maximise the sum (or future-discounted sum) of numerical "rewards" that are received during an episode. This is done by learning how to map situations to the most suitable actions. With reinforcement learning no expert is needed; the method can continuously adapt if the task requirements vary (sometimes called "concept drift"); and new knowledge can be discovered. Prior domain knowledge, as is used during supervised learning or in setting up an expert system, can be incorporated in order to give the learning a head-start or to direct the learning.

The most impressive example of reinforcement learning is possibly Gerry Tesauro's TD-Gammon. This plays backgammon at the level of the best human players in the world, and demonstrated some superior tactics that have now been adopted by the best human players. [Tesauro, 1994, 1995]. Success in other reinforcement learning endeavours has been more modest, however it is still early days in the history of reinforcement learning.

1.2 The Problem Domain

The problem domain addressed by this thesis is that of learning to drive a race car around a circuit in the minimum lap time. In real-life, this is known to be difficult for humans. The optimal actions are not exactly known because there are a very large number of variables, many of which often change (sometimes stochastically) and all of which may affect the outcome—for example: suspension geometries; road surface; temperature; air pressure; and tyre wear. Some of the best solutions found by expert human drivers use sequences of actions that may appear obscure. For example, these sometimes rely on side effects that only occur near the limit of the car's adhesion. These side effects may usually be a disadvantage but when used at the correct moment can counteract some other difficulty—for example: using the instability, that occurs at the point of maximum braking, to turn the car into a corner; or using the rear drift imposed by the sweeping action of rally tyres under power on a loose surface to steer the car, when the (useful) extra traction gained by the sweeping action has actually reduced the amount of steering effect given by the front tyres.

This shows that the optimal solution (at least, that known to human drivers) is sometimes in a remote corner of the search space. This makes it very difficult to learn excellent race driving. However, near-optimal solutions are well known, due to vast human experience in the real world. Nevertheless, humans require procedural learning—that is, learning-by-doing—rather than factual learning, to be able to drive a car, ride a bicycle, touch type, and so forth. That is, being *told* how to ride a bicycle is not enough, it also requires physical experience, and therefore every human must learn from scratch. The difficulty of "driving" makes it a promising domain for testing and developing Machine Learning techniques that have the potential to be robust enough to work in the real world. This assumes a realistic simulator is available.

Some early work in artificial intelligence that was successful in simple microworlds (sometimes called "toy" worlds) proved unscalable to more complex worlds, let alone the real world which is usually highly dimensional and often has all manner of confounding influences at work. For example, Terry Winograd's natural language understanding program works successfully in the "blocks" world, but due to its lack of general knowledge does not scale to the real world [Russell and Norvig, 2003]. The work presented in this thesis uses a simulator and therefore an artificial world; however, it is a reasonably complex world and is nearer to real life than some classic reinforcement learning domains such as the inverted pendulum and mountain car [Sutton and Barto, 1998; etc]. The simulator used in this work has 45 state parameters that describe the immediate environment, plus an additional 20 or more concerning nearby cars, many of which are continuous values (e.g. the speed of the car, the distance from the side of the track, etc). These can be used as inputs to agents within the world, and are all fully described in the Appendix. The simulator requires two simultaneous continuous actions (outputs) from each agent: steering-angle and desired-velocity. Furthermore, a proportion of the state-changes are executed randomly.

The simulator used in this work is the Robot Automobile Racing Simulator (RARS) [Timin 1995]. This provides a dynamic, closed environment of a simulated racetrack. It provides a ready-made graphical output and an interface for "automobile robots". That is, it provides to each robot a set of parameters that describe the current environment, and receives settings for steering and velocity from the robot. The effects of the steering and velocity commands are calculated by RARS and used to update the robot's location. Some of the RARS state parameters, and information that can be derived from them, lend themselves for use as feedback in reinforcement learning, (e.g. lap time, instantaneous speed, average speed, and damage).

RARS has been in use for about ten years, during which time some excellent heuristic robot drivers have been developed. Rémi Coulom [2002] divides these roughly into two categories:

- "Cars that compute an optimal path first. This method allows to use [sic] very costly optimization algorithm. … Drivers based on this kind of methods are usually rather poor at passing,… but often achieve excellent lap time on empty tracks."
- "Cars that do not compute an optimal path first. They can generate control variables by simply observing their current state, without referring to a fixed trajectory. … uses clever heuristics and obtains very good performance… particularly shines in heavy traffic conditions, where it is necessary to drive far away from the optimal path. … good passer … These car [sic] are usually slower than those based on an optimal path when the track is empty."

[Coulom, 2002, p110-111]

Cars that pre-compute the optimal path typically retrieve a copy of "track_desc" from the main RARS code. "track_desc" provides a detailed description of the specific track the car is about to drive on. The same description is used by the GUI to actually draw the track.

Coulom's aim was to use reinforcement learning to build a controller that had both good trajectories and good passing abilities. He fulfilled neither aspiration, but did produce a reinforcement learning robot of modest ability. Reinforcement learning still holds the prospect of producing such a controller; however, the work in this thesis does not succeed in those terms, either.

Availability of excellent heuristic robots is useful because they can be used as a standard against which to compare the robot that is developed in the research presented here. Another useful feature of RARS is its large variety of tracks. These are good for testing the ability of the robot to generalise. Finally, thanks to its graphical interface, RARS can be a lot of fun to watch!

1.3 The General Approach

The overall design of this research is an incremental approach: where only one aspect at a time is changed. First of all, a simple tabular representation for the Q-function is implemented using an array. This is then trained by using supervised learning of data generated by running a simple heuristic robot within RARS. The success of this training shows that the choice of state parameters, their discretisation and the array can work to store the Q-value in sufficient detail. Given this success, reinforcement learning can be implemented using the representational framework developed so far. The array is then hashed. This frees up a lot of memory. The discretisation resolution is increased until the hashed array uses nearly all the available RAM (without thrashing occurring). The effects of generalisation are then tested, without the complications of using a function approximator, by using a simple nearest neighbour method. The generalisation proves useful (as is expected), so a function approximator can be implemented. There are several possibilities, but in the reinforcement learning domain those that allow on-line incremental use are more elegant than those that require batch updates—for example, G-learning [Chapman and Kaelbling, 1991], and variations on G-learning [Uther and Veloso, 1998; Pyeatt and Howe, 1998c] or tile coding (CMAC) [Watkins, 1989]. Future possibilities for testing include incremental representation (e.g. variable resolution)—a type of hierarchal reinforcement learning; and guided exploration, some ideas for which are discussed in [Cleland, 2003]. Generalisation, hierarchal reinforcement learning and guided exploration are particularly active areas of research in the RL community.

This incremental development approach appears staircase-like (to use a software engineering term), but actually turns out to be more spiral like. This is because at each step (i.e. development iteration) there is still a functioning robot: it just, usually, performs better after each step is completed. This allows comparison of performance from one development step to the next to be used to judge the usefulness of the most recent development step.

1.4 Thesis Outline

The aim of this work is to build an agent based only on reinforcement learning that performs optimally in the domain of the Robot Auto Racing Simulator and uses the minimum of prior knowledge. The thesis is organised as follows. Chapter 2 presents background information on reinforcement learning and the Robot Auto Racing Simulator (RARS). Reinforcement learning is defined, and then an overview is given of previous work that uses reinforcement learning within RARS. The main questions remaining in the field of reinforcement learning are given, and the work of this thesis is placed into that context. Issues involved in making the agent run continuously are described. The choice of state description parameters is discussed and their meanings are given. The method used to measure performance is stated.

All the experiments in Chapter 3 involve the learning being primed by initial supervised learning. The tabular representation is discussed, as are the state parameters and their discretisation. The track and teacher used for supervised learning are considered, as are the rewards of damage, lap time and speed. The idea of temporal difference back-ups is introduced. Initial experiments are performed that use damage rewards, lap time rewards and combined lap time and damage rewards. The use of exploration is motivated, and a method of regulating it is given. An unexpected problem caused by inherent exploration is revealed and solutions are explored.

Chapters 4, and onwards, use experiments that do not employ initial supervised learning. The reasons for discontinuing initial supervised learning are explained, then the investigation of pure reinforcement learning in RARS is continued. The effect of the initial Q-value is investigated, and higher values are found preferable, for reasons that are explained. The work of this thesis uses very long episodes; therefore, discounting is needed to constrain the Q-value size. A compromise value is found experimentally.

Chapter 5 describes experiments with various damage reward sizes. The use of speed rewards is then introduced and various schemes are tested. It is shown that the use of speed rewards addresses the lap-time/damage trade-off problem. Eligibility traces are implemented, and this improves learning efficiency considerably. Hashing is used to free up memory that is utilised to increase the discretisation resolution. This results in slower learning but improved driving. Finally, a simple nearest neighbour generalisation method is described that improves the transfer of knowledge between different driving tasks.

Chapter 6 deals with peculiarities of the RARS domain that give rise to implementational issues that need to be addressed to enable reinforcement learning to be used within RARS. These matters include a range of issues concerning pit stops. Chapter 6 then details the variety of methods employed to judge the progress and performance of the learning algorithm. Some screen shots of the RARS circuit are also provided to help visually compare the driving of the reinforcement learning robot with that of an expert robot and a basic robot.

Finally, Chapter 7 gives a summary of the thesis as a whole, and gives a broad view of the main achievements of the work. Possibilities for future work are described, and these show this thesis is but a small part of a much larger picture, and has raised more questions than it has answered.

2 Background

This chapter defines reinforcement learning, both intuitively and formally. An overview is then given of previous work that investigates the use of reinforcement learning within the Robot Auto Racing Simulator (RARS). A broad inspection of the main questions remaining in the study of reinforcement learning is presented, and the work of this thesis is placed into the context of those questions. The objective of this thesis is then stated. Major design decisions peculiar to the RARS domain are then discussed: how to run the agent continuously, by flawlessly dealing with crash recovery, and how the time steps are managed during these periods, are described in detail. The choice of state parameters is discussed and their description is given. Finally, the method used in this work to measure performance is stated.

2.1 Reinforcement Learning Defined

A succinct description of reinforcement learning is given in Sutton and Barto's text [Sutton and Barto, 1998] as follows:

> "Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them." [Sutton and Barto, 1998, Section 1.1]

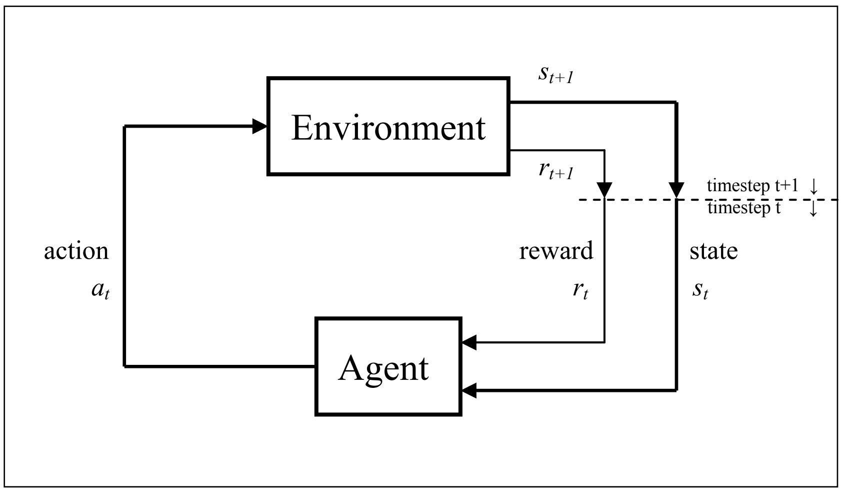


Figure 2-1 The Agent-environment Interaction (Reproduced from [Sutton and Barto, 1998])

A broader description of reinforcement learning from the on-line encyclopaedia Wikipedia is summarised below. Figure 2-1 is useful for understanding the formal description.

> "Reinforcement learning refers to a class of problems in machine learning which postulate an *agent* exploring an *environment* in which the agent perceives its current state and takes *actions*. The environment, in return, provides a *reward* … . Reinforcement learning algorithms attempt to find a *policy* for maximizing cumulative reward for the agent over the course of the problem.
>
> The environment is typically formulated as a finite-state Markov decision process (MDP), and reinforcement learning algorithms for this context are highly related to dynamic programming techniques. State transition probabilities and reward probabilities in the MDP are typically stochastic... .
>
> Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). …
>
> Formally, the basic reinforcement learning model consists of:
>
> 1. a set of environment states S;
> 2. a set of actions A; and
> 3. a set of scalar "rewards" in $\mathbb{R}$.
>
> [*Refer to Figure 2-1*]. At each time t, the agent perceives its state $s_t \in S$ … . It chooses an action $a_t \in A(s_t)$ and receives from the environment the new state s_{t+1} and a reward r_{t+1}. Based on these interactions, the reinforcement learning agent must develop a policy $\pi{:}S \to A$ which maximizes the quantity $R=r_0+r_1+...+r_n$ for MDPs which have a terminal state, or the quantity $R=\Sigma_t \gamma^t r_t$ for MDPs without terminal states (where γ is some "future reward" discounting

> factor between 0.0 and 1.0), [*and where R is some sort of sum of the rewards received on entering a particular state;* r_0 *is the most recent reward, and so forth*].
>
> Thus, reinforcement learning is particularly well suited to problems which include a long-term versus short-term reward trade-off."
>
> [http://en.wikipedia.org/wiki/Reinforcement_learning As at 30 March 2006]

2.2 Previous Work on Learning within RARS

Cleland [2003] investigates the use of supervised learning and evolutionary techniques within RARS. The supervised learning algorithms used are those in the WEKA machine learning workbench[2]. Agents built using supervised learning were able to equal their teacher in driving ability. The use of evolutionary learning along with the WEKA classifiers only produced very small gains in performance, although the implementation was rudimentary as only a small proportion of the research time was spent on this part of the work. The most successful classifier was m5′ which is a decision tree with linear regression models at the leaves. Instance based methods, such as LWR and IBk, were also shown to work, but their drawback is slow classification speed. Generalisation to other tracks was successful to a limited degree with each of these classifiers. These WEKA supervised learning classifiers are inspectable. This is invaluable because the reasoning used by the classifier can be seen; as opposed to function approximators such as neural nets, which are difficult to inspect and are more like a "black box".

As recent work has shown however, the application of function approximation to reinforcement learning is not a straight-forward matter. This is because the errors generated by the function approximator (classifier), while not a problem with supervised learning which has only one learning phase, cause trouble with reinforcement learning. This is because most reinforcement learning methods boot-strap. That is, they base estimates, in part, on previous estimates. With a tabular representation (which has no

[2] Details of the WEKA machine learning workbench are at: http://www.cs.waikato.ac.nz/ml/

generalisation error) the estimates gradually become more accurate as the agent experiences the environment. However, with a function approximator the errors introduced by the approximation can accumulate with each learning update and can render the information useless [Tsitsiklis and VanRoy, 1996]. Recent work has proved that with certain types of reinforcement learning (e.g. SARSA and averaging reinforcement learning), and under certain conditions (e.g. when using linear function approximators), convergence can be guaranteed, [Gordon, 2001; Perkins and Precup, 2002; Reynolds, 2002a; Szepesvari and Smart, 2004; Wiering, 2004]. These findings have yet to be investigated in the context of the RARS domain.

Reinforcement Learning

There are three main groups of people who have previously worked on using reinforcement learning in the RARS domain. These are Rémi Coulom [Coulom, 2002], Larry Pyeatt and Adele Howe [Pyeatt and Howe, 1998a,b,c; Pyeatt, Howe and Anderson, 1996], and Marco Barreno and Darren Liccardo [Barreno and Liccardo, 2003]. An overview of their work is given below.

Coulom

The theme of Coulom's work is the investigation of the use of continuous (both state and action) Temporal Difference(λ) learning with neural networks to build controllers for simulated motor control tasks. RARS is only one of many problem domains used in this work. The aim, in RARS, was to produce an agent that had both good trajectories and good passing abilities, (his car description is quoted in Section 1.2). However, this work did not get to investigate the learning of passing, and only produced modestly good trajectories. (Note that the supervised learning in [Cleland, 2003] did successfully learn passing, which demonstrates that this detail can be held by the m5′ decision tree).

Coulom [Coulom, 2002] uses an empty track, (as does the work presented in this thesis), and appears to treat the task as episodic, where the episode finishes when the car crosses the start-finish line. A feed-forward neural network with 30 neurons is used. Both the state and actions are treated continuously, and both the steering and speed actions are controlled.

The state parameters are transformed (sometimes known as forming "features", or "shaping", although both these terms have other meanings) to help ease learning. This is worth trying in the work of the current thesis. However, Coulom also uses the distance from the start line as a state parameter. As discussed in Sections 2.6 and 3.2, this parameter is specifically avoided because it can be used as a "key" (i.e. its values uniquely identify the position around a circuit, which makes some parameters such as current-radius redundant, and leaves the possibility open of forming a model that is completely non-transferable to another track of different layout). The reward used was the velocity of the car. The result of Coulom's first experiment was clean driving, but with poor driving lines and slow lap times.

Coulom added "features" —that is, redundant relevant inputs—with the intention of making the learning easier. Some features used were the distance to the wall dead ahead and the angle of incidence. This resulted in much improved driving lines and a faster, respectable, lap time. Although the time is still well short of the fastest heuristic robots in RARS.

It is of interest to note that Rémi Coulom also wrote k1999, a RARS robot based on path optimisation (it does not use reinforcement learning). This is one of the best RARS robots, winning the 2000 and 2001 seasons. It finds excellent and near-perfect driving lines.

Pyeatt and Howe

Pyeatt and Howe used RARS for a series of works. Their earlier work [Pyeatt and Howe, 1998b] studied coordinated control: they compared the use of a single function approximator (a neural net) against using two, that is, one function approximator per action space (steering and speed). The two networks were synchronised by providing the steering and speed actions used in the previous time step as inputs to both networks. Learning speed and generalisation were tested. The distributed (two-network) representation was clearly found to be the best performing and fastest learning. This contrasts with [Cleland, 2003] where the synchronisation of two function approximators was found to be critical and problematic in the RARS domain for anything other than very conservative driving styles. This appeared to be due to the interaction between the two actions. However, this work used decision trees, not neural networks. Pyeatt and Howe tested both actor-critic and Q-learning architectures, and Q-learning was found superior.

The overall aim of Pyeatt and Howe's work is to develop a method for automatically generating separate low level reactive behaviours, using reinforcement learning, that can be automatically combined in a modular fashion for control by a higher level coordinating mechanism. The idea being that the agent can add new low level behaviours as needed. For example, a reinforcement learning based overtaking behaviour is added to the steering and speed control behaviours. Essentially, structure in the domain is used to hierarchically decompose the problem (e.g. if there is a car in front then switch to overtaking behaviour).

The high level controller determines which behaviours are active at any time, and also assigns rewards to the learners. Different rewards may be given to each learner. The speed control learner's rewards are: -1 if crashed or travelling at less than 15mph; +1 at end of race. The steering learner's rewards are: -1 if crashed; +1 at end of race (no penalty is given for low speed). The overtaking learner's rewards are: -10 if crashed; +1 if overtaking succeeds within 15 time steps. The task is set up as a continuous learning problem. A crash recovery heuristic is used (the details of the heuristic are not given in their report). Making pit stops is mentioned, but it is not clear if damage effects are disabled, and if not then how pitting is handled to avoid interference with learning. Pyeatt and Howe found that coarse-coding the input gave better performance than using continuous inputs. It was also found that the neural networks needed careful, manual, parameter tuning before they would work in the RARS domain. It is not mentioned what track is used in the experiments. Therefore, performance can not be compared with other studies. It is stated that performance is comparable to the medium speed heuristic robots (this work was performed around 1996 to 1998, when RARS was 2 or 3 years old).

Neural networks have a problem in that their updates are non-local and so when weights are changed to correct one output error this can adversely affect other outputs. This is manifest as unlearning of useful information and occurs later in training. Sometimes this is known as "over training" (not to be confused with over fitting). To avoid this trouble, Pyeatt and Howe [Pyeatt and Howe, 1998c] investigate the use of decision trees as function approximators for reinforcement learning in RARS. The trees are based on the G-tree algorithm of Chapman and Kaelbling [1991]. This starts by representing the world as a single node, and recursively splits as needed according to a t-test of historical data. This gives variable resolution discretisation. The tree is updated continuously (not batch updated). This arrangement clearly learns faster, performs better (crashes less often) and does not over train, compared to the use of a neural network. Various methods of choosing split points were tested, and Student's t-test was found to be clearly best. It is not mentioned which track is used in these experiments, and therefore the results can not be compared to other work. However, (reading between the lines) the performance of the decision tree function approximator might be comparable to, or may crash more often than, the work described in this thesis which uses a hashed look up table. However, the decision tree learns much faster. Pyeatt and Howe moved from using neural networks with the actor-critic architecture to neural networks with Q-learning to decision trees with Q-learning.

Barreno and Liccardo

Barreno and Liccardo [2003] test reinforcement learning in the RARS domain using three different set ups. They use discrete states and discrete actions, and discrete states and continuous actions, and both of these arrangements use tile coding as the function approximator. The third arrangement uses continuous states and continuous actions and a neural net as the function approximator. All three arrangements use a fixed speed, so the agent only has to learn the steering behaviour. The problem is treated as an undiscounted, episodic task. Each episode ends when the agent completes one lap or crashes. The track used most often was oval2.trk, which is a simple oval. The learning parameters on all experiments were set at: $\varepsilon = 0.05$; $\lambda = 0.9$; $\gamma = 1.0$; $\alpha = 0.5$. The state parameters used were: velocity in the normal direction, distance from the centre line and the curvature of the track.

On the experiments with discrete states and discrete actions (and tile coding), the rewards used were a fixed negative value on crashing and a fixed positive value on lap completion. This converged quickly and produced a simple lane-following policy where the agent kept near the outside of the track. However, as the agent has a slow fixed speed, the optimal policy should be to follow the inside of the track. A further reward was added of a fixed negative value on each time step, with the intention of encouraging minimum lap time. Unfortunately, this did not help. The reward scheme was then changed to: $average\_lap\_speed^3$ if the lap is completed; *1/distance_from_start* if crashed. It is guessed that the authors intended to write −*1/distance_from_start*, so that an early crash is worth less—has greater penalty—than a crash further down the track (*1/distance_from_start* encourages the agent to crash as soon as possible). This produced a similar result to the previous experiment, but the agent followed a lane closer to the inside of the track. It is noted that the use of average lap speed as a reward is incorrect, *1/lap_time* should be used instead. This is because, as discussed elsewhere in this thesis, a lap with the fastest average speed does not necessarily have the fastest lap time (e.g. the agent may take a long path).

An experiment run in [Barreno and Liccardo, 2003] with discrete states and continuous actions (and tile coding) uses the same reward schemes as above, and the corresponding results are very similar to those above, however the driving is noticeably smoother. Both of these arrangements succeeded when learning on tracks more complex than oval2.trk, but the driving lines were nowhere near optimal—the agent did little more than avoid crashing. All of these agents learn much faster than those in the work of this thesis, typically converging by about 80 laps. This may indicate the power given by the generalisation provided by tile coding, but the comparison is murky as the agents in this thesis use more inputs, more actions and produce better driving performance. The neural net agent, with continuous states and a continuous action, failed to learn anything useful, even after millions of episodes and many attempts at parameter tuning. Barreno and Liccardo [2003] is also interesting because it includes a perspective from the control theory and nonlinear systems theory point of view.

Each of the three groups of researchers, above, have different motivations for their exploration of reinforcement learning in RARS. They arrive at three different solutions. None of these perform excellently, but all work at least as well as the simple heuristic robots. Coulom's robot probably works best, judging by the driving lines shown. However, the three investigations all use different tracks and the results are impossible to compare objectively.

2.3 Remaining Questions in the Field of Reinforcement Learning

There are many unsolved areas of reinforcement learning, and this certainly includes aspects of the use of reinforcement learning in RARS (there are a number of examples in this work, e.g. how to make use of multiple reward sources). Some well-researched areas of reinforcement learning do not yet have a complete theoretical basis. For example, function approximators that work well for supervised learning do not always work with reinforcement learning, and the reasons for this are not entirely understood. As discussed in Section 2.2, function approximators can sometimes diverge, or converge to an incorrect policy. The causes of this, and solutions to it, are a subject of current research (an overview of such research is given in Section 2.2).

A current direction in reinforcement learning research is to study real-world problems. These are also known as "real life" or "situated" domains [RL3; Mataric, 1994; Isbell, et al, 2001; Shelton, 2001; Natarajan and Tadepalli, 2005]. These domains tend to expose unexpected difficulties with classic reinforcement learning techniques. For example, often there are several different goals, all of which must be achieved. Difficulties occur when the goals have very different time scales; or the order of achievement is important; or the goals have different priorities; or goals need concurrent achievement. This often occurs when a large task has to be disassembled into a set of smaller tasks. A classic thought-example ("gedankenexperiment") is the task of learning to make a cup of tea: there are numerous sub-tasks, and the order of most of these is critical. It is a surprisingly complex task. Meta-techniques can be useful to guide the learning in these situations [Mataric, 1994; Isbell, et al, 2001; Shelton, 2001; Natarajan and Tadepalli, 2005].

These trends in reinforcement learning research may reflect a trend in the wider area of artificial intelligence: specifically, looking at broader problems and treating them more holistically, for example by adopting techniques from control theory (from engineering), and statistics. The early days of artificial intelligence research (up to mid 1970s) did not produce the results expected. Part of the reason was over-optimism in what since proved to be a difficult domain. Focus changed to tackling smaller, more achievable, sub problems. For example: searching, planning, reasoning, knowledge representation, acting under uncertainty, supervised learning, and so on, as seen in any artificial intelligence text [e.g. Russell and Norvig, 2003]. It has now become more evident that some of these solutions can be more powerful when they are combined, rather than used in isolation. It has been argued that this approach is essential for long-term progress in artificial intelligence research [Hendler, 2000]. It has been claimed that the field of reinforcement learning has a wider frontier than, for example, supervised machine learning, in that it tackles the *whole* problem of a goal-seeking agent interacting with an uncertain environment. For example, problems such as the exploration-exploitation dilemma do not even arise in supervised learning. [Sutton and Barto, 1998, 1.1]. In this context, the current movement towards tackling real-world problems is simply a continuation of the pre-existing, pragmatic approach of the work in reinforcement learning. Reinforcement learning is progressively borrowing concepts from other areas of artificial intelligence and engineering because the methods work well together. Perhaps reinforcement learning may prove to be a unifying framework for artificial intelligence; or it may be a sub-part of a larger framework.

A specific area of current reinforcement learning research is function approximation. Function approximation is an important part of most reinforcement learning agents. This is because of the compression it gives to the value function representation, and because of the generalising effect of function approximators which allows the agent to successfully deal with previously unseen situations and thereby also speed up the learning. It is suggested that a specific problem domain can be suited to a specific type of function approximator. This may be due to the representation used in the function approximator being more closely matched to the underlying structure of the problem domain, and/or this may mean the errors introduced by the function approximator are such as to not contribute to the (yet-to-be-completely-understood) divergence phenomena. The amount of generalisation given by the function approximator may have a related effect, and so, being able to control the amount of generalisation may be useful. Another aspect of function approximators is their use as a place for incorporating prior domain knowledge. This can be used to guide learning (i.e. give it some hints) and to give it a (possibly large) head-start. In practice this has proven highly effective. However, the prior knowledge will bias the learning and can make the agent less likely to discover some excellent novel solutions that are outside the concepts known by the human supplying the prior knowledge. The best way to include prior knowledge is an open question as is the best way to control generalisation and the best way to choose a function approximator to suit a specific problem domain.

Another open question in reinforcement learning is the pervasive problem of learning-speed and efficiency. That is, how to make the most of the experience already gained and not discard any useful knowledge previously accumulated. There is clearly much room for improvement here, and this is empirically shown by the ability of humans and animals to learn new tasks without needing to make the vast number of blunders that reinforcement learning agents typically do when learning similar tasks. This also relates to the generalising ability of the function approximator used. A number of approaches have been tried, such as storing the experience history and replaying it later, and several interesting variations on this [Lin, 1992, 1993; Cichosz, 1997, 1999; Reynolds, 2002b]. Other ideas that involve making better use of experience include: learning a model of the environment and using this for planning which is integrated with the learning (e.g. Dyna agents [Sutton, 1990, 1991]); prioritised sweeping; heuristic search; trajectory sampling, [Sutton and Barto, 1998]; and maintaining a selection of different policies [Natarajan and Tadepalli, 2005]. Fast Q(λ) uses lazy learning in that values are only updated when they are needed (i.e. when a state is revisited) [Wiering and Schmidhuber, 1998a&b]. Another efficiency-related problem is that of how to make combined use of several different sources of rewards, each of which provides effective feedback but in different environmental situations and/or at different times, when there is only one goal. That problem also arises in this thesis.

Some other broad areas of current work are on eliminating the requirement that the state representation has the Markov property (therefore allowing the solution of a larger class of problems). There is a history of work on this matter going back at least 20 years, yet the problem is still open [Littman, Cassandra, and Kaelbling, 1995; Parr and Russell, 1995; Chrisman, 1992; McCallum, 1993, 1995, Hochreiter and Schmidhuber, 1997; Mitchell, 2003]. Another area involves the ideas of modularity and hierarchy—that is, being able to learn and plan at a variety of levels of abstraction. For example, being able to treat a task as a single primitive action, and then being able to perform higher-level more complex tasks consisting of those actions. Being able to easily move between these levels of abstraction would give an agent flexibility that appears closer to the way humans solve problems.

Matthew Grounds [Grounds, 2004] gives an enlightening summary of current work in reinforcement learning. Another excellent overview is given in [Barto and Mahadevan, 2003]. For example, Grounds describes hierarchical reinforcement learning as decomposing a problem into subproblems, and the object of this is to constrain the scope of the learning. This means the true optimal policy is unlikely to be found; but a "good" policy is found in "orders of magnitude less time". He divides hierarchical learning into four techniques. The first technique is parallel decomposition, and suits tasks that can divide into independent or almost-independent subproblems which are then run in parallel. The second technique is state aggregation, in which states are grouped into higher level abstract states. Separate policies are learnt within each group of states, and an overall policy is learnt across the abstract (groups of) states. Some methods that fall within this category are Feudal reinforcement learning; the Parti-game algorithm [Moore and Atkeson, 1995] and derivatives such as [Munos and Moore, 2002]. The third technique is temporal abstraction, which uses the idea of macro actions (abstract actions). These are a temporal series of actions that perform some useful subtask. These are then used like primitive actions. Macro actions may be used within a macro action. Related methods include: (the early) H-DYNA; Hierarchies of Abstract Machines(HAMs) [Parr and Russell, 1997]; and the formalism of "options" [Sutton, Precup and Singh, 1999]. The fourth technique involves methods that combine temporal and state abstraction. This includes MAXQ [Dietterich, 2000] and the ALisp framework [Andre, 2003; Andre and Russell, 2001]. The decomposition into subproblems, upon which all hierarchical methods rely, is most often done by using prior domain knowledge (i.e. supplied by a human). Learning how to decompose the problem domain automatically has had little attention. However, methods do exist [McCallum, 1995; Thrun and Schwartz, 1994], also [Şimşek and Barto, 2004] use "relative novelty" to automatically create temporal abstractions.

Predictive Representations [Rafols et al, 2005] represent the world in terms of predictions about possible future experience. This is shown to result in better generalisation and faster learning. Sutton and Tanner [2005] use "T.D. networks", which produce a type of predictive representation. Littman et al [2002] and James et al [2005] use "Predictive State Representations" (PSRs) which are another type of predictive representation. Tanner and Sutton [2005] use T.D. networks extended with recent history. This gives the speed advantage of predictive representation and also allows the state representation to be a partially observable Markov decision process (POMDP), therefore expanding the class of solvable problems.

Sutton and Barto [1998, Section 10.1] outline at least 14 different areas for investigation, and each of these can be viewed as a dimension (a continuum) of the space of possible reinforcement learning methods. In the time since then many additional new areas have been examined. This gives a huge volume of reinforcement learning methods to investigate, the majority of which have yet to be tested. "Using reinforcement learning in practice is still as much art as science" [Sutton and Barto, 1998].

The Current Work, In Context

RARS is not a real life domain, but it is a fairly complex simulation. The current work does not use a function approximator, but uses a hashed tabular method. This avoids the confounding effects of a function approximator and allows the work to concentrate on other matters. In early experiments prior knowledge is supplied by using supervised learning, but this is discontinued in later experiments. Efforts are made to test the effect of varying the amount of generalisation. This is done by using a nearest neighbour method.

Learning speed and efficiency is poor. Millions of laps are often used in experiments. Speed-up methods such as planning and Fast Q(λ) are not used; but eligibility traces are used. However, this does not prevent experiments from being run long enough to test learning-convergence, because the RARS environment can cheaply supply endless amounts of experience (and quickly, as it runs much faster than real-time). Speed-up methods are useful in real life domains because of the lesser amount of experience required by the agent. However, their computational overhead means there may or may not be a saving in processor time. There is no guarantee that speed-up techniques will help the agent to ultimately find a better policy. Variable resolution (the ability to change the coarseness, i.e. the level of discretisation, of state variables according to the accuracy needed) is likely to be useful in the RARS domain, but is not tested. Hashing, as used, goes part way towards producing a similar effect by eliminating unused representation space, but it does not increase resolution where needed. In a similar vein, logarithmic discretisation may make more efficient use of resources, and thereby improve learning, but was not tried. Linear discretisation was used.

The non-Markov and the partial-observability problems do not occur in the RARS domain. The RARS domain does have the Markov property. That is, in each time step all the information needed to choose the optimal action is contained in the state parameters: there is no need to know the state or action history. Certainly, previous states and actions have a major effect on the current situation, but knowing about them does not help the agent. Knowing the direction and speed and what lies ahead at the current time step is all that is needed. What leads to the current situation makes no difference to the current action choice. Indeed, the correct sequence of actions is essential for a fast lap time, and this is what is rewarded by the temporal difference back ups.

Neither predictive representation nor hierarchical learning are attempted in this work with RARS. However, it is worth noting that the precursor to this current work [Cleland, 2003] does give the germ of an idea that might be useful for automatic problem domain decomposition (upon which hierarchical methods rely). Briefly: M5′ [3] is a supervised learning classifier that forms a decision tree with linear models at each leaf. This was successfully used as a function approximator in the RARS domain. It was observed that at run time the RARS agent did not jump all around the tree as it chose actions but used one leaf for tens or hundreds of time steps. If the leaves used are graphed against the track layout then particular leaves are seen to correspond to particular types of situation on the track. The same leaf is used at different locations around the track when they are similar in nature (e.g. approaching a corner). This naturally leads to the observation that this automatic division of the state space might be useful for some form of meta (e.g. hierarchical) learning. The tree also decomposes the space at higher levels of abstraction, that is, at its internal nodes. The split points in m5′ are determined by intra-subset variation. The use of a decision tree as a function approximator in reinforcement learning has not been widely investigated, but was used by [Pyeatt and Howe, 1998c] (which, incidentally, used the RARS domain). This was based on the G-tree [Chapman and Kaelbling, 1991], and Pyeatt and Howe found Student's t-test to be the best statistic for choosing split points. All these observations beg the obvious question: Can a decision tree function approximator be integrated with hierarchical reinforcement learning to achieve two things at once—automatic decomposition of the problem space, and function approximation?

This thesis presents empirical research that thoroughly investigates the essentials of using reinforcement learning in the RARS domain, and produces surprising performance considering the size of the search space. However, the work has a certain fragility: most experiments should have been repeated 5 times, or more, because of the variation in results due to the randomness in the environment. This was not done, due to the impractically large amount of additional time needed (months or years). Nevertheless, the fact that a number of different approaches/experimental-threads all lead to a well-performing agent; and that these different experimental threads did not uncover anomalies in results gives some empirical confidence that the results have solid foundation.

[3] The m5′ implementation from the WEKA machine learning workbench was used. Details can be found at: http://www.cs.waikato.ac.nz/ml/

2.4 Objective

The objective of this work is to build an agent based purely on using reinforcement learning that performs as well as is possible in the domain of the Robot Auto Racing Simulator, using the minimum of prior knowledge. There is no intention to investigate any specific sub-area of the reinforcement learning field, although this does not preclude doing so if it assists in the objective.

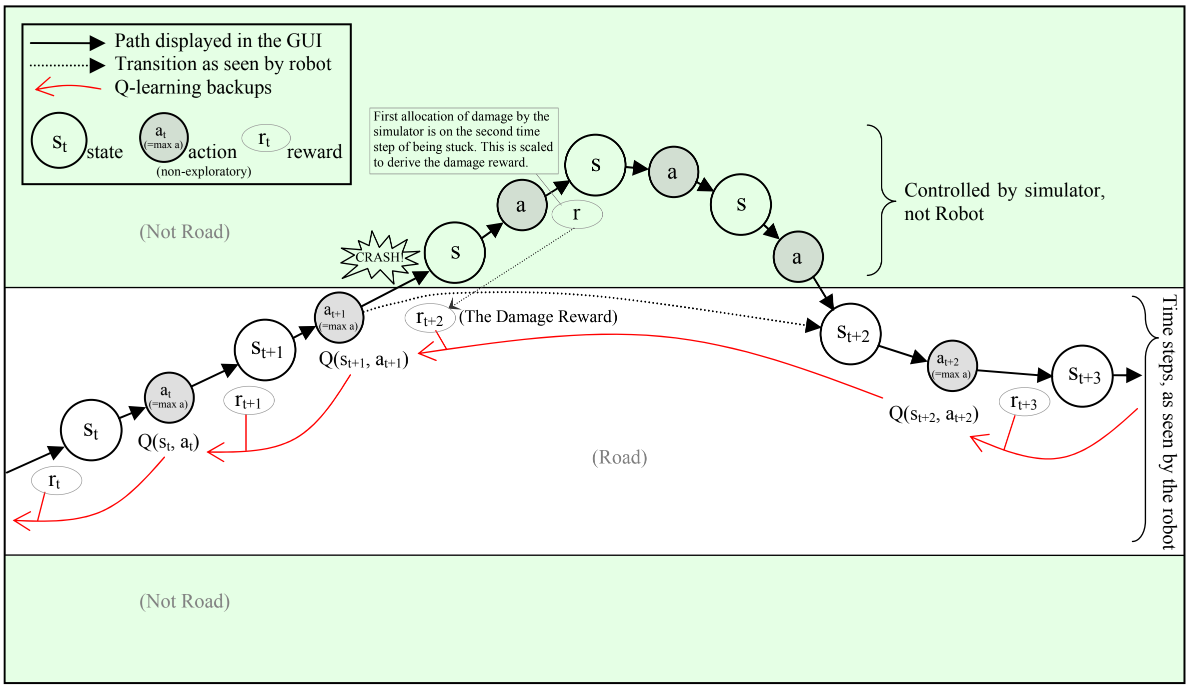
2.5 Crash Recovery

There are (at least) two ways to deal with recovery from crashes in RARS. The first gives episodic running and the second gives continuous running. The first method is to restart the car from the start/finish line after every crash: $$\begin{tikzcd}\text{(re)start} \arrow[r] & \text{run} \arrow[r] & \text{crash} \arrow[ll, bend right]\end{tikzcd}$$. This requires the robot to learn to avoid the first crash situation before it can continue. This means the robot initially gets a very limited range of experience until it perfects avoidance of the first crash situation. For this reason the learner may tend to over-fit.

A second method is to make use of the crash recovery code built into the RARS simulator. Using this, the car is automatically steered back onto the track under control of the simulator, and control is then handed back to the robot code: $$\begin{tikzcd}\text{start} \arrow[r] & \text{run} \arrow[r] & \text{crash} \arrow[r] & \text{return to track at point further along.} \arrow[lll, bend right]\end{tikzcd}$$ This method gives the robot a wider range of crash and non-crash experience earlier on in its life. This means it may take longer to learn to avoid any sort of crash, but it is also less likely to over-fit, thanks to the wider variety of experience.

The second method of crash recovery is chosen for this work. The time steps during which the simulator drives the robot back onto the track are visible to the robot, but because the robot has no control it must not perform learning during this period. For this reason those time steps are hidden from the reinforcement learning code. The method of temporal difference back ups is explained in Section 3.4, and how this is applied across a crash is best explained by Figure 2-2. The back up on these steps is needed so the crash damage feedback is taken into account. The derivation of the damage reward is explained in Section 3.3, and is also illustrated in Figure 2-2. Hiding the time steps from the learner that occur during crash recovery appears to open the opportunity for the agent to learn to use a crash as a means of gaining distance down the track in zero time. But this is not the case for two reasons. Firstly, the lap time is based on the time elapsed in the simulator, not in the agent, and so the time spent off course (which is considerable during a crash) is included. Secondly, the damage reward has a minimum value and this is set high enough to make any crash inadvisable.

An ordinary back-up is shown at the left hand side of Figure 2-2. The value of taking action a_{t+1} in state s_{t+1} (i.e. $Q(s_{t+1}, a_{t+1})$ along with any reward received on entering state s_{t+1} are used to modify the value of the previous state action pair, $Q(s_t, a_t)$, by moving its value slightly closer to that of $Q(s_{t+1}, a_{t+1})$. This is known as a temporal difference back-up, formally described as $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$, and is explained in Section 3.4. This means the value of $Q(s_t, a_t)$ becomes more representative/a-better-estimate of the rewards that lie ahead when action a_t is taken in state s_t. As the agent moves from state to state performing back-ups the value of taking each action in each state (the Q-values) become more accurate. If the Q-values are completely accurate and the best action is chosen in every state visited then the optimum policy is followed.



Path displayed in the GUI
Transition as seen by robot
Q-learning backups
s_t state
a_t (=max a) action (non-exploratory)
r_t reward
(Not Road)
First allocation of damage by the simulator is on the second time step of being stuck. This is scaled to derive the damage reward.
CRASH!
s
a
r
s
a
s
a
Controlled by simulator, not Robot
s_t
a_t (=max a)
$Q(s_t, a_t)$
r_t
s_{t+1}
r_{t+1}
a_{t+1} (=max a)
$Q(s_{t+1}, a_{t+1})$
r_{t+2} (The Damage Reward)
s_{t+2}
a_{t+2} (=max a)
$Q(s_{t+2}, a_{t+2})$
r_{t+3}
s_{t+3}
(Road)
Time steps, as seen by the robot
(Not Road)

Figure 2-2 The Crash Recovery Set-up

When the robot crashes off the track it is automatically steered back on by the simulator. The robot must not learn (by performing back-ups) during this time as it is not responsible for its actions. However, the damage suffered due to the crash needs to be learnt about since the robot is responsible for it, and this means the reward r_{t+2} needs to be taken into account. Additionally, to make the learning task continuous (cf. episodic) the states before and after a crash are, in effect, linked as shown by the dotted arrow in Figure 2-2. The back-up from this state transition is the occasion for the incorporation of the crash damage reward r_{t+2}. As it happens, the first damage allocated by the simulator during a crash does not occur until the second time step off the track. This is used to derive the damage reward r_{t+2}.

2.6 State Description Parameters

A robot in the RARS domain on each time step receives as input a set of parameters that describe the current state of its environment. From this information it determines the actions to take and returns these to the simulator as speed and steering requests. This is illustrated in Figure 2-3.

The choice of state description parameters is critical to successful learning. This is because some parameters may be redundant, irrelevant or carry very little information and thus cause over-fitting[4] or possibly mislead the learning algorithm. The RARS simulator has over 44 parameters, some of which are continuous (a complete list of the RARS parameters is given in the Appendix). The size of the state space grows exponentially with the number of parameters, and therefore this number must be reduced. There must be enough parameters to describe a state in sufficient detail, yet few enough to allow an array of them to fit into memory. The choices made here are based on earlier work when the same matter was investigated [Cleland 2003]. These decisions rely on domain experience, testing and experimentation from that earlier work. Some of the reasoning is given briefly below.

[4] Over-fitting is a well documented phenomenon that plagues machine learning. Briefly, it is caused by machine learning algorithms attaching significance to irrelevant data (or too much significance to data of little relevance).

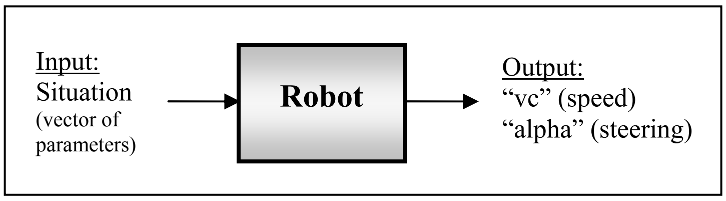


Figure 2-3 A Robot Takes a Situation Input and Produces an Action Output

The parameter name abbreviations used in the following discussion are explained in the Appendix. Many of these parameter choices rely on domain knowledge, are "educated guesses" and are therefore subjective. "Power" (power delivered) and "power_req" (the ratio of power requested by driver to maximum power) are not used, as they are proportional to "vc" (velocity commanded—translates to throttle/brake) which is one of the actions being derived, that is, they are internal to the car rather than the road. Bestlap_speed, lastlap_speed and lap_time are suitable as reinforcement learning feedback, not environmental descriptions, because they describe the robot's performance and not the environment. Parameters that describe the track that lies quite far ahead: after_rad; after_len; aftaft_rad; aftaft_len are omitted initially to reduce the state space size, but could be added back later. These may allow the classifier to detect patterns that correspond to specific long sequences of corners (e.g. left-right-left; straight-left-right, etc). The parameters that describe the track a shorter distance ahead cur_rad; cur_len; nex_len; nex_rad are initially used. Time_count, distance and seg_ID may act as "keys". That is, their values are always unique. This enables the learner to simply index by distance or time. This forms a driving script specific for one track which can never generalise to describe driving on any track. Given that track_width is constant for a particular track and that *track_width = to_rgt + to_lft*, using both to_rgt and to_lft is redundant. Therefore, to_rgt is removed. A more satisfactory solution would be to use a parameter that gives lateral position as a *proportion* of track width. The preliminary solution of using only to_lft was retained. Cen_a (centripetal acceleration) was found to be used in error by the classifiers in [Cleland, 2003] as a cause of turning, whereas in practice it is generated by (is an effect of) turning. Cen_a is a type of feedback parameter, rather than an environmental description, and so is omitted. A robot going backwards appears to be always involved in a crash, in which case default code in the main RARS program takes over to set the robot back on the track facing forward after assigning it damage. Therefore, the backwards parameter is deemed unnecessary and is removed.

Name of Environmental Parameter	Explanation of Environmental Parameter
v	the speed of the car
vn	component of v perpendicular to track direction
to_lft	distance to left wall
cur_rad	radius of inner track wall. 0 ⇒ straight; negative ⇒ right
nex_rad	radius of inner wall of next segment. 0 ⇒ straight; negative ⇒ right
cur_len	length of current track segment (if segment is a curve then angle; if segment is a straight then feet) *Note: the use of cur_len is later discontinued.*
nex_len	length of next track segment (if segment is a curve then angle; if segment is a straight then feet)
to_end	how far to end of current track segment (if segment is a curve then angle remaining; if segment is a straight then feet remaining)

Table 2-1 Explanation of the Environmental Parameters Selected as Inputs

The final choice of parameters are: cur_rad; cur_len; to_lft; to_end; v; vn; nex_len; nex_rad. The meanings of these are given in Table 2-1. In addition, there are two actions: vc ("velocity commanded") and alpha ("steering angle").

2.7 Measurement of Success

The main parameters used to measure learning, in this work, are lap time or total-rewards-received-per-lap. These give a good indication of learning progress over long experiments. However, empirical observations are occasionally given. For example, the driving is described as "wobbly" or "the robot makes no attempt to turn" or "the robot brakes too late into the corner". These empirical observations are inherently subjective and difficult to quantify. However, they are useful over short time periods when the robot is having difficulty. This is because they can give clues to possible causes that can not be obtained by just observing the lap time. This is an advantage of having the graphical interface of RARS, and that the simulation can be slowed down. In this work, the discussion of results is largely empirical (e.g. the comparison of average rewards or lap times, or shapes of graphs): there is minimal theoretical discussion.

Goal

In the experiments of this thesis the goal is to minimise total reward. This is explained towards the end of both Sections 3.3 and 3.4. When the goal is minimisation, the objective is for a learning curve like that shown in Figure 2-4. That is: a curve with a fall as steep as possible, at all points along it; one that falls as low as possible; and with as little variation as possible (i.e. “smooth”), especially not increasing at any place. A compromise between these characteristics may be useful. For example, steepness of fall could be traded against ultimate lowness; earlier variability could be traded against later flatness, etc. It is useful if the trend towards the minimum is not asymptote-like, in that the ultimate minimum is actually reached, rather than never quite achieved. The theoretical optimum learning curve is L-shaped: the minimum time is reached on the first time step and is never departed from. However, in reality learning takes longer than this, and typically looks more like Figure 2-4.

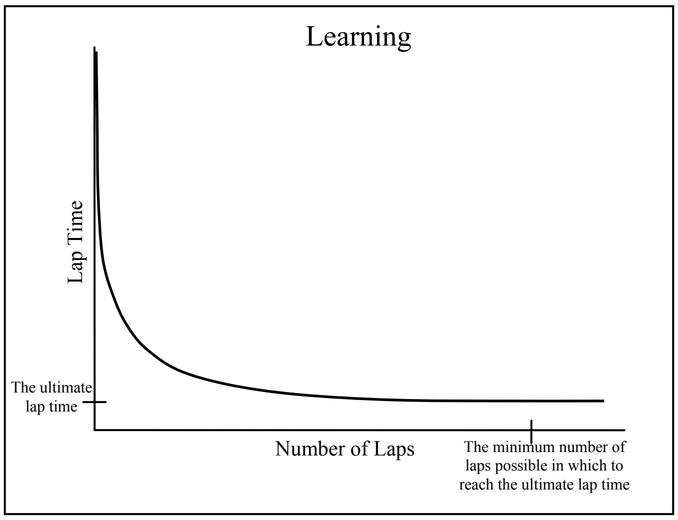


Figure 2-4 A Superb Learning Curve when the goal is Minimisation

2.8 Chapter Summary

Reinforcement learning is a method of learning by trial and error. Previous knowledge of the problem domain is not needed. The method is powerful, yet there are many unsolved questions concerning its implementation. The field is immature. Reinforcement learning has been used previously in the RARS domain, but very little and with only modest success. The objective of this work is simply to build a reinforcement learning agent that performs as well as is possible in RARS, using the minimum of prior knowledge. “As well as possible” means finding the path that gives the fastest lap time, and doing this quickly and consistently. To allow the agent to run continuously in RARS, the time steps and rewards occurring around crash recovery need to be dealt with in a particular fashion. The first design task of all is a careful choice of state description parameters.

3 Primed Learning

This chapter describes the initial implementation of reinforcement learning within RARS, and covers the representation used for the action-value function; the rewards and exploration. All the experiments in this chapter involve the learning being primed by initial supervised learning.

The organisation of this chapter is as follows. Section 3.1 discusses the tabular representation. Section 3.2 discusses the state parameters and their discretisation; the track and the teacher used for supervised learning. Section 3.3 discusses using rewards of damage, lap time and speed. Section 3.4 introduces the idea of temporal difference back-ups. Section 3.5 shows a proof-of-concept experiment. Sections 3.6 and 3.7 show experiments using lap time rewards and combined lap time and damage rewards. Section 3.8 motivates the use of exploration in learning and shows its long term effect and a method of regulating the amount of exploration. Section 3.9 discusses a problem caused by unforeseen inherent exploration and explores various solutions. Finally, Section 3.10 summarises the important points from this chapter.

3.1 Tabular Method

A tabular method refers to the state-action values as being held in a simple look-up table. That is, when the structure is indexed first by state and then by action a unique memory location is accessed that holds the value of taking that action in that state (also known as the Q-value of the state-action pair).

3.1.1 Reasons for using a Tabular Method

Classic reinforcement learning uses a tabular method to represent the value function. This is the simplest method to analyse, and proofs of convergence for various reinforcement learning methods all assume a tabular method. It is also easy to implement as a simple array.

The aim of this paper is to implement reinforcement learning in a domain with a high number of state parameters, two continuous actions and randomness in the simulator. This is a reasonably difficult domain, therefore a simple tabular method is used at the outset as a straightforward way to see if it is possible to use reinforcement learning within RARS.

However, it is widely held that function approximators (e.g. neural networks, decision trees, etc.) should be of great benefit when combined with reinforcement learning. This is because the generalisation they provide is expected to help the agent learn more quickly and with fewer errors when encountering novel situations. This is because some of the knowledge gained in situations previously visited that are *similar* to a novel situation can be applied to the novel situation. Another reason for using function approximation is the compression it provides in a high-dimensional space, especially if the space is sparsely populated. However, this combination has turned out to have both successes and difficulties, such as over-estimation that can lead to divergence [Thrun & Schwartz 1993, Wiering 2004]. The use of function approximators in reinforcement learning is currently an active area of research. Function approximation is a long-term aim of this work. But an incremental approach is taken and it is first shown that reinforcement learning can work successfully in the RARS domain when using a simple tabular method. This occupies large parts of this work, as a number of refinements are made. It is then shown that compression is worthwhile, because it frees up memory allowing finer discretisation which results in higher performance (Section 5.4). Next, it is shown that generalisation improves performance, although only a small gain is made, which is possibly due to the simplicity of the technique used (Section 5.5). At this point, the ground has been made ready for the application of function approximation. That is, reinforcement learning is working successfully in the RARS domain; and it is shown to be advantageous to use a generalising and compressing technique.

3.1.2 Reasons for using Initial Supervised Learning (with the tabular method)

It has been suggested that it may not be possible to use a tabular method with RARS [Pyeatt & Howe 1998c]. The conjecture is that the large number of state parameters plus two actions will result in an impractically large array. That is, a function approximator may be mandatory because of its more compact representation. The amount of RAM memory commonly available during early studies was considerably less than what is usual now, and therefore the use of a tabular method within RARS is worth reinvestigation. First, proof is needed that when all the spare main memory in the available hardware is used up in the description of the state-action space then sufficient detail (accuracy) is provided to enable an agent using that description to successfully drive in the RARS simulator. In other words, if the concept needed for modest driving ability can not be stored in sufficient detail, using an array, on the available hardware, then it is pointless attempting to build a reinforcement learning agent that uses the same storage model. A quick way of doing this is to use supervised learning. That is, an existing heuristic robot is run for some time; its inputs and outputs are discretised at the same resolution as used in the array of the reinforcement learning robot and these are recorded. Prior to the first time step of driving, this data is used to fill in corresponding parts of the array belonging to the reinforcement learning robot. This is done by marking the actions taken in a state (by the heuristic robot) with a "desirable" Q-value, while all other actions are initialised to an "undesirable" Q-value. When the reinforcement learning agent uses this array it simply chooses the same (or one of the same, if there are several) action in each state as the teacher (the heuristic robot) would do. If the supervised-trained agent is able to negotiate the RARS circuit then this indicates it is worth continuing with a reinforcement learning implementation using a tabular representation.

Reinforcement learning is typically a slow process, especially early in learning if the agent starts with no knowledge. Another use of supervised learning is to provide some initial knowledge before using reinforcement learning. This is what is meant by the term "primed learning".

3.2 Preliminary setup

Parameters

The choice of state description variables (parameters) is critical to successful learning. This is because some variables might be redundant, irrelevant or carry very little information and thus cause over-fitting or possibly mislead the learning algorithm. The RARS simulator has over 44 parameters. The size of the state space grows exponentially with the number of parameters, and therefore this number must be reduced. The choices made here are based on earlier work, when the same matter was investigated [Cleland 2003]. This relies on previous domain experience, testing and experimentation, which will not be explained here. There must be enough parameters to describe a state in sufficient detail, yet few enough to allow an array of them to fit into memory. Those chosen are: cur_rad; cur_len; to_lft; to_end; v; vn; nex_len; nex_rad. In addition, there are two actions: vc ("velocity commanded") and alpha ("steering angle"). The meaning of these state parameters and the reasons for choosing them are given in Chapter 2.

In particular, distance from the start/finish line is not used because this can act as a key for a particular track. Distance from the start/finish line defines the position around a specific track and therefore makes cur_rad, cur_len, nex_len and nex_rad redundant. If the learning agent discovers this and attaches little or no significance to cur_rad, cur_len, nex_len and nex_rad, then the learned model can not possibly generalise to another track with a different shape. This is because at the same distance around different tracks the states (situations) are quite different. Not using distance from start/finish line forces the agent to make use of cur_rad, cur_len, nex_len and nex_rad. Learning the general skill of driving in the RARS simulator (on any track) is the aim of this research. This contrasts with the other known works on reinforcement learning in RARS, at least two of which use distance from the start/finish line as a state parameter [Coulom, 2002; Barreno and Liccardo, 2003].

Discretisation

Discretisation step size, as well as the number and choice of parameters, affects memory usage and performance. For example, if five steps of discretisation are used for each of the eight state parameters and also for the two actions then this calls for a ten-dimensional array with each element of size five, requiring 5^{10} = 9,765,625 array locations. Each location holds a Q-value of type float, which uses four bytes. For statistical purposes and use in the learning algorithm, each location is also given a number-of-visits value. This is of type int, which uses four bytes. Type short uses only two bytes but, due to memory allocation practices of the operating system (word alignment), four bytes are set aside for each short. Therefore, the total memory needed is 9,765,625 × 8 = 78,125,000 bytes. This is well within the main memory capacity of a typical desktop machine.

Teacher and Track

Earlier work [Cleland 2003] indicates that supervised learning is more likely to be successful if an unsophisticated heuristic robot is used as the teacher, rather than a complex high performing heuristic robot. That is, a robot that has initial supervised training on data generated by a simple heuristic robot is likely to drive well. This is possibly due to the unsophisticated heuristic robots having a conservative driving style, for example, never driving on the edge of the track. This means small driving inaccuracies caused by the supervised learning are not likely to cause a crash. Robot 01.cpp is an example of this, and was used as the teacher. The simplest track in RARS was used, (v01.trk).

Resolution

Resolution refers to the granularity of the discretisation, that is, the step-size of the discretisation. A test experiment early in this work used a supervised-taught robot which used five steps of discretisation for each of the eight state parameters and the two actions. The actions of the supervised-taught robot were compared with those of the teacher (01.cpp) in the same situations (states). The teacher (01.cpp) had used three different actions in states that came to be discretised identically in the supervised-taught robot. Because the three states appeared identical to the supervised-taught robot it could only take the mean of the three actions. This comparison shows that resolution was a problem.

It was anticipated a larger array would still be workable. The state parameters to_lft, to_end, v and vn were discretised using seven steps (rather than five). This used $5^6 \times 7^4 \times 8$ bytes = 300,125,000 bytes = 286.2 Megabytes (MB) of memory. The size of the run-time memory taken by the complete RARS program (as calculated by the compiler) is then about 325 MB, and in practice RARS was able to run on the test machine when using this much memory. If vc and alpha (the actions) are also discretised into seven steps the complete run-time size exceeds 600 MB, and is too large to run without thrashing. So, the set up using 286 MB of memory was implemented, trained and run. The robot still crashed near the same place as it did before the resolution was increased, although it now took a tighter line around the first corner (which more closely imitates the teacher, 01.cpp).

These results show the robot is able to negotiate the first corner and thereby suggest that tabular representation is likely to be effective and worth further investigation. Therefore the schemes described were retained. Initial supervised learning was also retained to provide some initial knowledge (which improves performance in the early stages of learning) until reinforcement learning is implemented, and tuned, satisfactorily.

3.3 Rewards

Reinforcement learning requires feedback to tell the agent how well it is doing with respect to the goal. This may be delayed by many time steps from the actions responsible, and is commonly termed a “reward”. Three sources of rewards are used from RARS: damage, lap time and speed.

Damage

Figure 2-2 illustrates the time steps, back ups and reward allocation occurring during a crash, and helps illuminate the following discussion. Damage is allocated to the robot by the simulator if the robot crosses over the side of the track. Once the robot crosses the track edge the simulator takes over control. The robot’s code is called but it is passed a “stuck” signal, and a routine provided by the simulator is used to steer it back onto the track. From the second time step of being stuck damage points are given to the robot. These points have a range of values and are accumulated every time step until the robot is back on the track. The damage score is maintained by the simulator, and can be accessed by the robot but not modified by it. The damage score can only be reduced during a pit stop. Excessive damage or running out of fuel will put the robot out of the race. The amount of damage is roughly proportionate to the severity of the crash. This applies both to the first damage score (on the second time step of being stuck) and the total damage from an incident (the amount accumulated by the time the robot is returned to the track). However, there appears to be a bug in RARS, where on rare bad crashes the robot goes into thousands of spins and accumulates enormous damage. For this reason the damage awarded on the second time step of being stuck is used as the reward. That this is a smaller quantity than the accumulated damage does not matter because the three different reward sources are scaled relative to each other. Damage rewards are implemented to be approximately proportional to the size of the crash. This provides information to the robot about the severity of each crash. A simpler implementation is to use a fixed damage reward. This is discussed in Section 5.1.3, along with the testing of different ranges of values for the damage reward.

A crash causes the lap time to increase. This is a second form of reward that results from a crash, however it is more delayed from the crash than is the damage reward. The increase in lap time is not as effective for learning crash avoidance as the damage reward. This is shown by comparing the results of the experiments in Sections 3.5 and 3.6.

Lap time

The second source of reward is the lap time. This is only given once per lap, when crossing the start/finish line. One lap of modest speed on v01.trk takes about 1,000 time steps. However, the lap time reward is the most meaningful reward, because the entire aim of the RARS robot is to find the smallest lap time. This is the problem version used in this work, and applies when the robot is racing alone. If the robot were racing against another car then the aim becomes to cross the finish line first. This is often not the same aim as doing the fastest lap time, especially if the robots are closely matched. There are several interesting reasons for this but they will not be discussed here.

Since the start/finish line is only crossed once every approximately 1,000 time steps it will take at least approximately 1,000 laps for the lap time reward to trickle back by 1-step temporal difference back-ups. Even then, a good lap time reward is not attributed directly to the good actions that contributed to it. For example, if good actions occurred far back from the start/finish line and the randomness of the environment has since caused poor outcomes for some intervening state-actions, then the resulting poor lap time will be credited to both the poor and good actions. To alleviate this effect, eligibility traces could be implemented, as described in Section 5.3 . These could be made to a greater depth for lap time rewards than for crash rewards, because the series of actions responsible for a lap time lie deeper into history than the actions responsible for a crash. It is noted that eligibility traces are commonly used, but variable depth (i.e. variable λ) is not usual.

Another possibility is to have several, say four, time rewards around the circuit. The reward given could be the time elapsed since the previous reward; or the lap time up to that point (from the start/finish line); or the time elapsed since crossing that same point one lap ago; or something else. A danger in breaking a lap into segments is that each segment is not independent. So, to give a reward of the time elapsed on one segment only, assumes the surrounding segments have made no contribution. This could pervert the driving style. A simple example is the case of a reward given for a segment that finishes at the end of a straight, just before a tight corner. The best reward is gained by maximum acceleration down the straight. However, this means the robot has no chance of navigating the following turn (which is part of the next segment). So, in order to make a good time in the following segment, the robot must sacrifice some time in the preceding segment. Shifting the reward line to a more "sensible" place, e.g. the start of a straight, or halfway down a straight, is no guarantee against the problem, as it may become more subtle. For example, having a reward at the halfway point in an s-bend seems reasonable, if the s-bend is symmetrical. However, from domain experience it is known that the best line through a symmetrical s-bend is not itself symmetrical. The first turn is usually faster than the second, with a tiny amount of braking in-between (across the reward line). So, this example has the same problem as the first example.

Given enough experience, the learner will probably (automatically) find a good compromise between adjacent segments. But, because this introduces the additional task of finding a compromise it must slow the learning. Giving a reward of the time for the whole lap completed since last crossing the same reward line still suffers the original problem. That is, the reward for a poor lap may not necessarily trickle back to the early actions that caused the poor time, while actions near the reward line get given "blame" even if they are excellent.

A useful approach may be to combine both eligibility traces and multiple time rewards. Time rewards would be given at several places around the track, and the back-ups at these places given eligibility traces. The traces would go only as deep as the most distant action that could influence the position at each reward line. For example three, or perhaps two, corners back from each reward line. The reward given is the elapsed time since that most distant action. This way, blame or credit is more likely given to the deserving action, and never given to actions in the too distant past. Also, an action is soon given a reward, (within two or three corners), rather than after as long as a whole lap. It could be argued that all the actions in these traces are equally responsible for the segment time, and therefore the eligibility trace discount, λ, should be set to 1, as this gives all the action values equal weight.

However, this does not solve the "suicidal" problems mentioned above, (e.g. caused by rewards being given at the end of straights). *This might be solved by having each of the reward segments overlapping, for example, by one or two corners.* This would require two or more eligibility traces to be maintained simultaneously. However, no established theoretical foundations can be found for this idea.

Another way of giving time rewards is to give a penalty of +1 on every time step, except on crossing the start/finish line when no penalty is given. This is the typical way maze problems are given motivation to find the shortest path. Yet, this may be an equivalent to giving a lap time at the start/finish line. Both methods encourage the robot to reach the start/finish line as soon as possible, by giving a penalty proportional to the number of time steps taken. (The robot tries to minimise this number, hence the lap time "reward" acts, in a sense, as a "penalty").

Speed

Speed is a third source of reward, but was not used until later in the research, and is discussed in Section 5.2. The general idea is that a higher average speed is more desirable. This reward is available every time step, and can be different every time step, unlike damage and lap time rewards which are more sparse. Speed turns out to be a very effective form of feedback. However, maintaining the highest average speed around a circuit will not give the fastest lap time, as every racing driver knows, and this research also shows! (The minimum lap time often does not occur on the lap with the highest average speed). The fastest lap time is not given by taking the shortest path, either. The shortest path on a circuit is always the hard inside line, and involves lower speed due to the corners of tighter radius. This lower speed is not compensated for by the shorter path. It turns out it is possible to maintain a higher average speed, than that maintained during the fastest lap, by taking a longer path. However, the longer path is not compensated for by the higher average speed. This is not the longest path, which is the extreme outside path, because that involves violent steering changes on corner entry and exit, and violent steering changes require a low speed to be executed. The best lap time requires a particular balance between a short path and a high average speed (and a gradual change of direction—a maximum centripetal acceleration, although this is a main requirement for high average speed).

There are two speed measurements provided by the simulator. One is the speed of the robot in the direction it is travelling; the other is the speed of the robot in the direction normal to the track wall. Neither always indicates the robot's speed of progress around the track. Normal velocity is sometimes useful, (e.g. when turning into a corner), sometimes not. If the robot is travelling directly towards the track edge it may have a high velocity, which will all be in the normal direction, but this is probably not a useful thing to be doing. The most useful measurement of progress is the tangential velocity, that is, the velocity in the direction of the track. This is given by the size of the vector difference of the two previously mentioned velocities: $V_t = \sqrt{V^2 - V_n^2}$. The relationship of these three velocities is shown in Figure 3-1.

Figure 3-1 illustrates the derivation of tangential velocity in three different scenarios. Scenarios 2 and 3 both result in the same tangential velocity, yet scenario 3 is a much better state for the robot to be in because its actual velocity (V) is closer to the direction of the track *ahead*, compared to scenario 2 where the actual velocity is towards the track edge *ahead*. Both these situations are given the same tangential velocity reward. This appears counter-intuitive, but it is not a problem because the two situations are different states due to their normal velocities being in opposite directions. Normal velocity is one of the state parameters, and its direction is encoded. This means that as the robot learns, the state action pair representing situation 3 will develop a "good" Q-value; while the state action pair representing situation 2 will develop a "poor" Q-value (because it crashes soon after, or wastes time turning to avoid the wall).

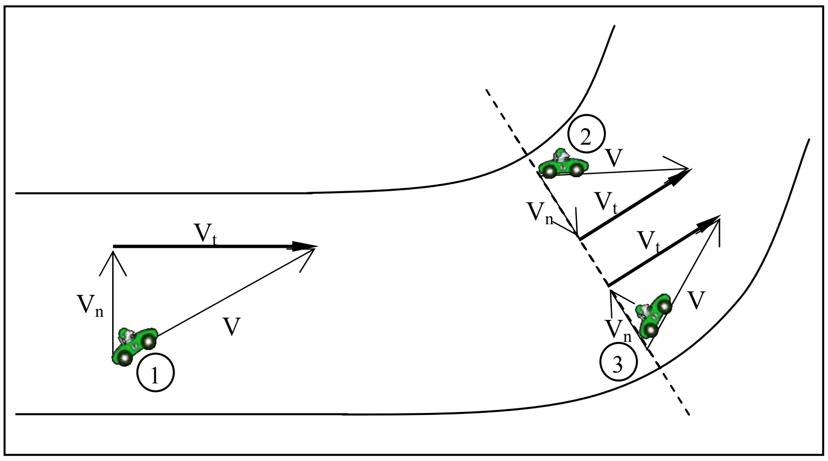


Figure 3-1 The Derivation of Tangential Velocity

Experiments were run to compare the use of the robot's actual velocity versus the tangential velocity as speed rewards. The tangential velocity reward proves more effective for learning.

General Discussion and Terminology

The hard problem of delayed rewards in reinforcement learning may now be more evident. That is, the things responsible for a lap time are *all* of the actions and any good or bad luck (randomness) experienced since the previous crossing of the start/finish line. In fact, a little further back than that because the speed and position of crossing the start/finish line affects the following lap. A similar situation applies to crashes. The things responsible for a crash lie some indeterminate number of time steps before the crash, and each has different and unknown amounts of blame/credit. This can include random responses by the environment.

Rewards in reinforcement learning are traditionally quantities that are maximised. But the goal can instead be minimisation if smaller reward sizes are optimal (e.g. if rewards are interpreted as penalties). In the RARS domain minimisation makes more intuitive sense: the best lap time is the smallest lap time, and minimising damage is obviously advantageous (i.e. zero damage is best). However, the speed reward needs to be inverted so that the largest speed appears the most desirable. Minimisation of reward is used in all the experiments of this work. This is also discussed at the end of Section 3.4.

Feedback is traditionally viewed as a reward in reinforcement learning, but at times it seems to make more sense to talk of the feedback as a penalty. For example, a "crash penalty" sounds more intuitive than a "crash reward". A "speed reward" may sound better than a "speed penalty", although a low speed can be viewed as a "speed penalty". It is not clearly meaningful to describe a lap time as a penalty or a reward. That depends on if it is a "good" lap time or a "bad" one. What is a "good" lap time on a particular track may not be known, and in fact this is what the agent is trying to discover. It would be clearer to talk only about "feedback", without attributing a judgement such as "reward" or "penalty". However, it appears to be a tradition in reinforcement learning to call the quantity a "reward". Therefore in this work all the three feedback quantities will always be called "rewards".

Experiment

In an early experiment damage rewards were implemented. This is seen to affect learning after a number of crashes, but only if the display is in slow motion. The robot can then be seen to turn left or right on the single time step immediately prior to crashing. This does not prevent the crash as action needs to be taken earlier. This requires the rewards to be fed back to earlier time steps, and that is what temporal difference backups do.

3.4 Temporal Difference Backups (1 Step)

Temporal difference back-ups are the heart of reinforcement learning. There are established theories and proofs concerning the convergence of 1-step temporal difference learning [Watkins 1989, Sutton & Barto 1998]. A brief descriptive outline is given here, in the context of Q-learning and this thesis. A backup diagram illustrating Q-learning, which helps clarify the discussion in this section, is given in Appendix A.

The array used in this work stores a representation of the robot. That is, for every possible action in every possible state there is stored an associated "Q-value". Each Q-value indicates the "usefulness" of taking that action in that state. More specifically, it represents (some sort of) sum of all expected future rewards to be gained after taking that action in that state. The task for the robot is usually to simply choose the action with the "best" Q-value whenever it finds itself in a new state. The temporal difference back-up method is used to update these Q-value estimates. The basic idea is that if an action is taken in a state that causes the robot to move to a new state, then the value of that new state (in terms of the Q-value of its *best* action) should be reflected in the Q-value of the previous state-action-pair (the state-action-pair that just lead to the new state). In addition, any reward received during the state transition must also be accounted for. Therefore, the value of the current state-action pair is modified to be slightly closer to the value of its succeeding state-action pair plus any reward received. In this way, as the robot visits a succession of states (taking actions in each) the experience it gains, in terms of rewards, gradually trickles, or "bubbles", back to the previous state-action pairs, thereby giving a measure of how "useful" those earlier state-action pairs are in terms of the rewards they lead to. State-action pairs are revisited on subsequent laps, and also some are revisited on the same lap. This maintains the "trickling back" effect, that is, on each pass of the algorithm estimates of future rewards, (i.e. Q-values), are fed back by one step. From all of this, the robot eventually forms a set of Q-values that represent the optimum path through the state space in terms of receiving maximum reward.

The back-up formula for 1-step Q-learning is:
$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

Where $Q(s_t, a_t)$ is the Q-value of the current state-action pair; $\max_a Q(s_{t+1}, a)$ is the Q-value of the best action in the next state (that is, the highest Q-value), this is the *action that is normally taken* in the next state; r_{t+1} is the *reward* given on transition from s_t to s_{t+1} (if any); α is the *learning rate*, $0 < \alpha \leq 1$, and determines how much the current Q-value is altered, it also has the effect of decreasing the weight of, or "forgetting", rewards received earlier in the history of the current state-action pair, $Q(s_t, a_t)$; γ is the *discount factor*, $0 \leq \gamma \leq 1$, and is used on continuous tasks to ensure the expected reward sum (i.e. Q-value) is finite, it also has the effect of decreasing the weight of more temporally distant expected rewards, that is, rewards that are expected to be received in states that are expected to be visited when starting from the current state action pair are given less weight as their distance into the future increases: $Q_t = r_t + \gamma Q_{t+1}$ $[\Rightarrow Q_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$; $(r_{t+1} + \gamma \max_a Q(s_{t+1}, a))$ together can be regarded as the "target", that is, the direction in which to move the value of $Q(s_t, a_t)$; $[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$ can be regarded as the "error" in the estimate (the estimate being $Q(s_t, a_t)$). r_{t+1} and s_{t+1} are provided by the environment after action a_t is taken in state s_t. The formula above can not be applied until time t+1 because r_{t+1} and s_{t+1} are not known until time t+1 (also, the environment may be stochastic to some extent, therefore r_{t+1} and s_{t+1} can not be pre-calculated). So t+1 is usually the current time step and t is the previous time step, hence the term "back-up".

This equivalent formula may be clearer, and highlights the purpose of α:
$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)]$

The term $Q(s_t, a_t)$ has been factored out. This form shows that $-\alpha$ of the original Q-value is removed: $(1 - \alpha)Q(s_t, a_t)$; and is replaced by α of the target: $\alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)]$. Remember, $0 < \alpha \leq 1$.

Rewards in reinforcement learning are traditionally quantities that are maximised (this is discussed in Section 3.3, under sub heading 'General Discussion and Terminology'). But there is no theoretical reason why the goal can not instead be minimisation if smaller reward sizes are optimal (e.g. if rewards are interpreted as penalties). In the RARS domain minimisation makes more intuitive sense: the best lap time is the smallest lap time, and minimising damage is obviously advantageous (i.e. zero damage is best). However, the speed reward needs to be inverted so that the largest speed appears the most desirable. Minimisation of reward is used in all the experiments of this work. Therefore, "max" is replaced with "min" in the back-up formula:

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

The Q-value of a state action pair still represents the (discounted) sum of all expected future rewards to be had from that state action pair onwards. The optimum policy involves selecting the action with the *minimum* Q-value in each state as it is encountered, not the action with the maximum Q-value as is traditional. The Q-value when following the optimal policy remains as: $Q^*(s_t, a_t) = r_t + \gamma Q^*(s_{t+1}, a_{t+1})$. But this is equal to: $r_t + \gamma \min_a Q(s_{t+1}, a)$, when the minimum is optimal; compared to: $r_t + \gamma \max_a Q(s_{t+1}, a)$, when the maximum is optimal (as is traditional). $Q^*(s_t, a_t)$ is shorthand for $Q^{\pi^*}(s_t, a_t)$ and means the Q-value of taking action a_t in state s_t when following the optimal policy.

3.5 Proof of Concept, using damage reward only

Figure 3-2 plots progress in learning as the robot completes more and more laps of a circuit. The purpose of this initial experiment is as a "proof of concept" of using reinforcement learning in the RARS domain. Rewards are given on crashes, not lap times, as these occur more often and are therefore likely to give faster learning. Backups are performed. The robot is given initial supervised learning before taking the first time step with the simulator. The supervised learning uses the data recorded from the heuristic robot 01.cpp when run for three laps of track v01.trk. The method used is discussed in section 3.1.2.

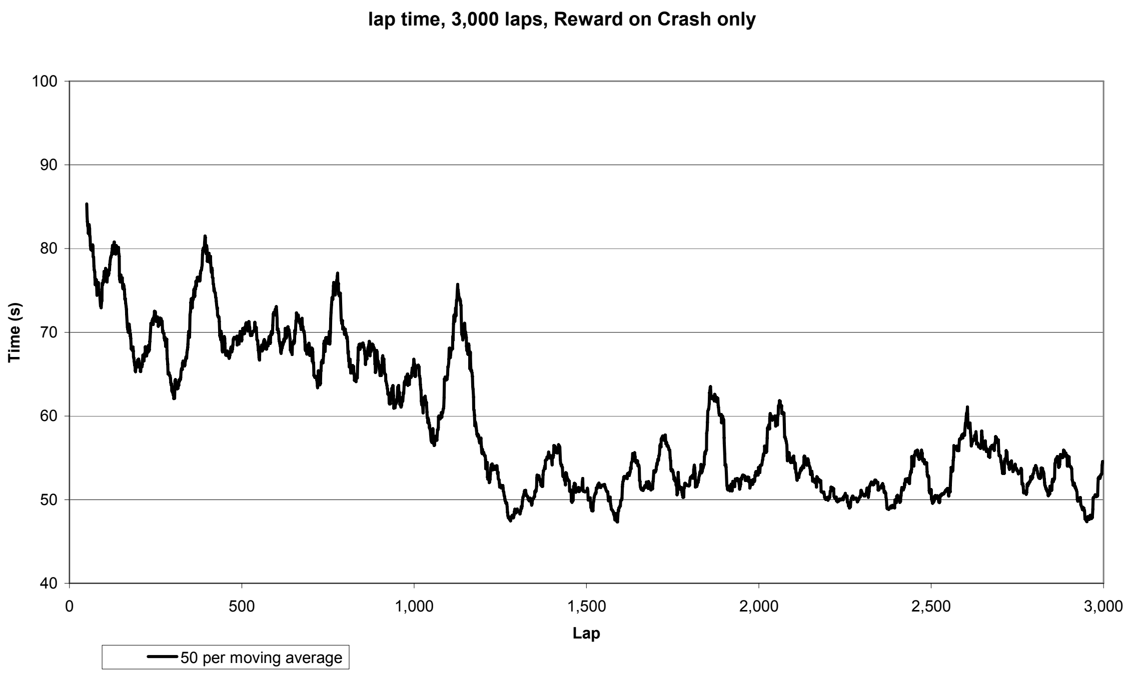


Figure 3-2 Lap Time, 3,000 laps, Reward on Crash only.

Figure 3-2 shows learning occurs over the first 1,300 laps, after which the robot appears to stabilise somewhat at a lap time just over 50 seconds. The raw data has considerable spread and is not shown. To help see the underlying trend in the data a moving average is made. Each datum in Figure 3-2 is equal to the average of the 50 previous raw (original) data. This calculation is made using a sliding window of 50. Despite the averaging, the graph still shows considerable variation between laps, yet the overall trend in the first half of the graph is a decrease in lap time. This indicates the learning algorithm is working. However, a good heuristic robot does a lap time of 36 seconds. The reinforcement learning robot is averaging greater than 50 second laps, has considerable variation in lap times and crashes frequently. There is room for improvement. Nevertheless, Figure 3-2 demonstrates that it is possible for reinforcement learning to work in the RARS domain.

3.6 Lap Time Reward

The effectiveness for learning of using lap time rewards is investigated. Figure 3-3 shows the lap time graph of an experiment in which rewards are given on lap times and not on crashes. The expectation is for more gradual improvement than when rewards are given on crashes. This is because the lap time reward is only given once per lap, and it takes many laps for its effect to trickle back by the method of 1-step temporal difference learning. The original, unaveraged, (raw) data is represented by the grey line. This might look like a bar-chart, but it is actually a line graph with many adjacent points far apart vertically. This shows the very large variation in lap times. The black line again shows the moving average of the 50 previous data.

Figure 3-3 shows learning is slower than in the previous experiment, of Figure 3-2, because in Figure 3-3 it takes about 600 laps (on the x-axis) for the 50-per-moving-average line to reach about 75 seconds (on the y-axis), but in Figure 3-2 it takes only about 150 laps for the 50-per-moving-average line to reach around 75 seconds. Also, in the experiment of Figure 3-3, learning only occurs over the first 600 laps, after which there is no general improvement. This experiment was run for 100,000 laps, and the additional

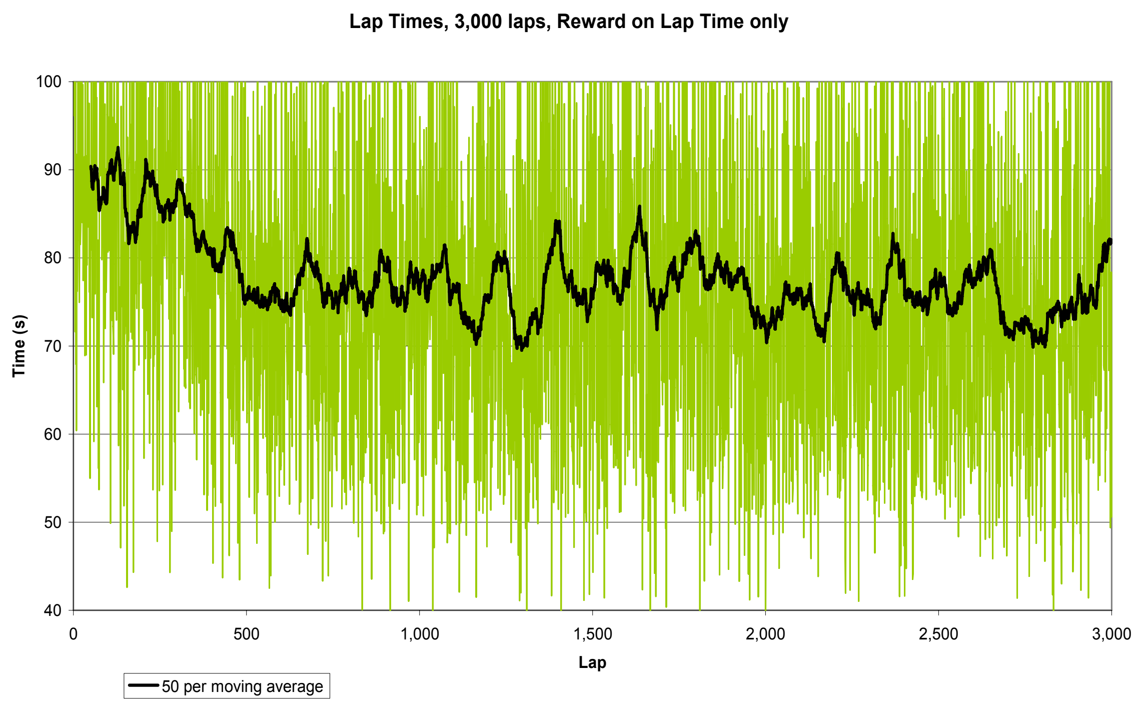


Figure 3-3 Lap Time, 3,000 laps, Reward on Lap Time Only

97,000 lap times look just like the last 2,000 laps of Figure 3-3. This is not as expected. It was later found that errors and omissions in the algorithm were partly responsible for this slow learning and plateau of learning, and also for the wide variation between adjacent lap times which is revealed by the large spread in the raw data, seen as the spread of the grey line in Figure 3-3 and also seen as the up and down jagged nature of the 50-per-moving-average line. The errors, omissions and tuning of the learning algorithm take most of the rest of this work to cover, and some RARS specific matters are covered in Section 6.1. However, even in later work, using only lap time rewards continues to give slower learning that reaches an early plateau.

3.7 Combined Lap Time and Damage Reward

The previous two sections show the use of damage rewards only and lap time rewards only. The effect of using lap time and damage rewards together is investigated by the experiment illustrated in Figure 3-4.

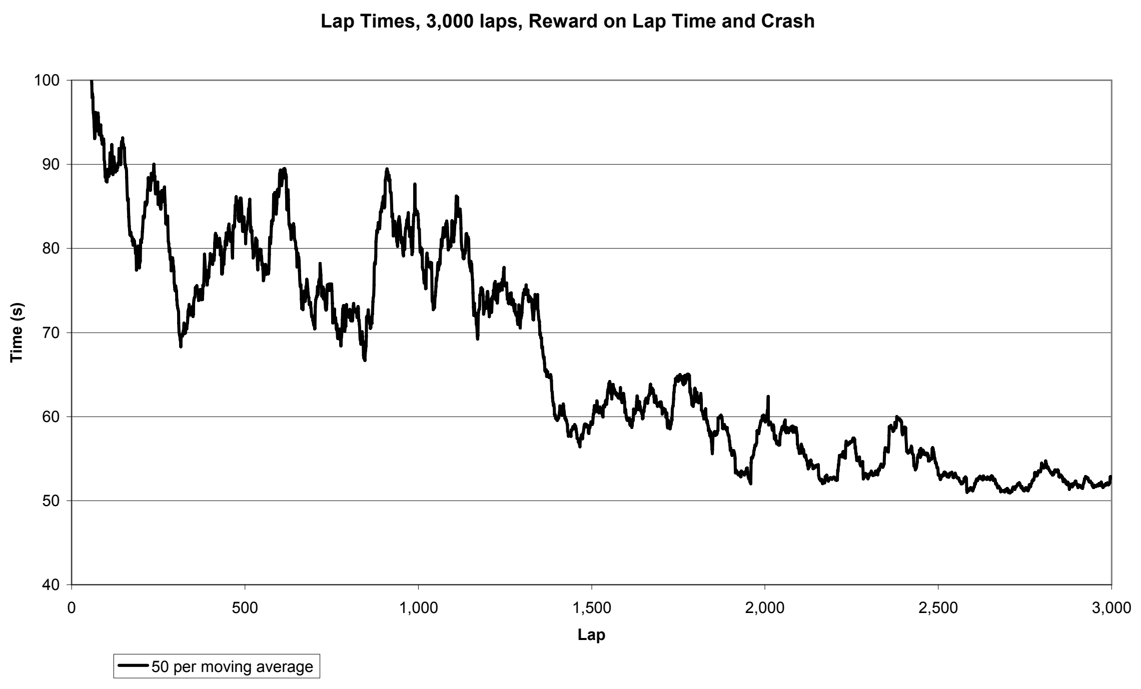


Figure 3-4 Lap Time, 3,000 laps, Reward on Lap Time and Crash

Figure 3-4 shows that using both crash and lap time rewards produces performance somewhere in between the previous two experiments. The graph starts out with as much variation (spikiness) as the crash-reward-only graph, but finishes with considerably less variation (it is smoother) than either of the two previous experiments. It shows slower learning than with crash reward only, and faster than with lap reward only. After 3,000 laps it does not reach as fast a lap time as with crash reward only (52 seconds compared to 48 seconds). However, there is more improvement in the long term, as shown in Figure 3-5. Using a moving average of 25 for each graph does not change this comparison. Over the first 100 laps the learning is slower than in either of the two previous experiments. The reason is not known, although this much variation in results might be due to random variation between runs of the same experiment.

However, an improvement in driving is observed when using both lap time and damage rewards. When only crash rewards are used, as in the experiment of Figure 3-2, then a "wobbly" driving style results. The robot appears to avoid crashes without regard to taking the most direct path. Using both crash and lap time rewards eliminates the "wobbliness". The robot appears to make a compromise between lap time and damage.

3.7.1 Medium Term Trend

Figure 3-5 shows the results when the experiment plotted in Figure 3-4 is allowed to run for 100,000 laps. (This experiment uses both lap time and damage rewards). The moving average line in Figure 3-5 is artificially smoother than that of Figure 3-4, that is, there appears to be less variation in times between adjacent laps. This is because the raw data for Figure 3-5 is averaged over blocks of 10, and the average of each 10 forms one datum on Figure 3-5. The blocks of 10 are adjacent, (and not overlapping, as used when calculating a moving average). This compresses the amount of data from 100,000 to 10,000, and is needed because the graphing tool used can not handle 100,000 data. From statistics it is known that sampled data has less variation than raw data, and this explains the reduced jaggedness ("artificial smoothness") in Figure 3-5 compared to Figure 3-4. Standard deviation of sample distribution = standard deviation of population / √(sample size). So for a sample size of 10 the deviation is decreased by $1/\sqrt{10} \approx 0.316$. On the other hand, in Figure 3-2 to Figure 3-4 the raw data is used directly (note, however, that in these figures

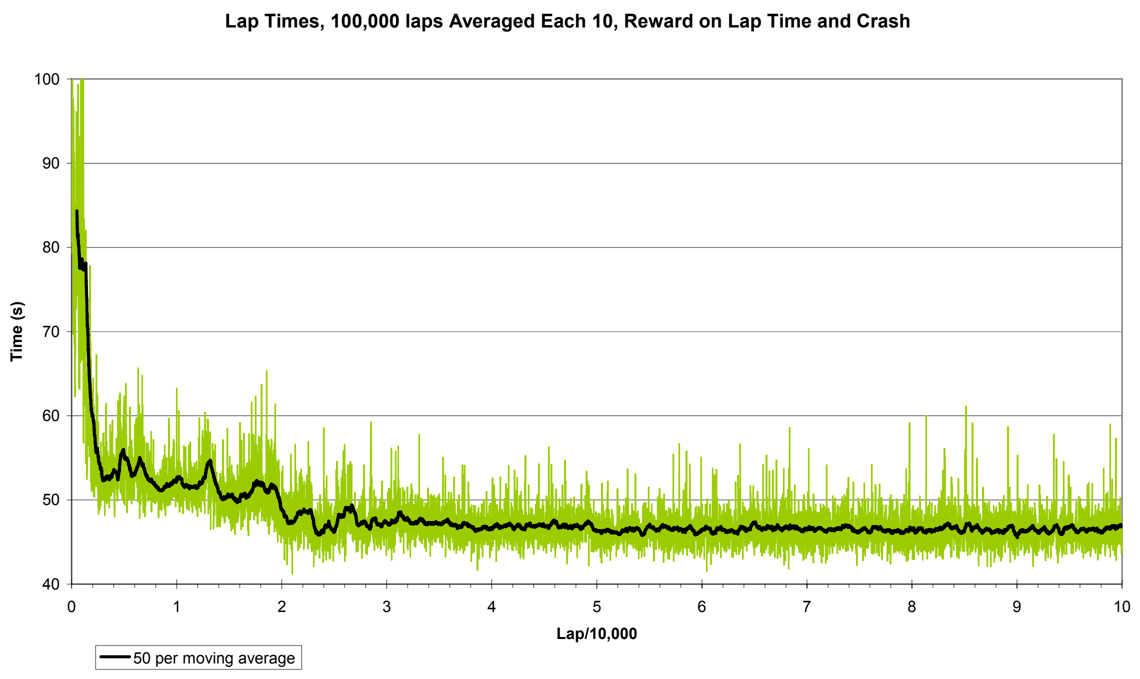


Figure 3-5 Lap Time, 100,000 laps, Reward on Lap Time and Crash

the use of 50-per-moving-average plots has a smoothing effect). The 50-per-moving-average line in Figure 3-5 uses the compressed data, and therefore it is really a 500-per-moving-average plot.

This method of reducing the data is used throughout this thesis because long experiments are run that generate more data than is practical to plot, as the graphing tool can plot a maximum of only 32,000 points. For example, to show 8,000,000 lap times they are first averaged in blocks of 250 to reduce the quantity to 32,000 data. Another way to compress the data is by sampling, say, every tenth lap, and this is discussed in Section 6.2.3. These technical details are only mentioned because they affect the apparent smoothness / variance of the graphs, and therefore need to be known about if graphs are to be meaningfully compared that use different sized blocks for averaging (e.g. the graphs may be of experiments of very different lengths, such as Figure 3-4 and Figure 3-5).

Figure 3-5 also serves as the control (baseline) for Figure 3-7 in Section 3.8.1. Figure 3-5 shows the lap time continuing to fall beyond lap 3,000 (3,000 is the last lap in Figure 3-4) and up until lap 24,000, where it reaches an average of about 47 seconds. This is better than the average lap time reached with crash rewards only, (although the crash-reward-only experiment, graphed in Figure 3-4, is for only 3,000 laps). Both lap time rewards and damage rewards give feedback that encourages learning the fastest lap time. The experiments of Figure 3-4 and Figure 3-5 test the learning effect of using both lap time and damage rewards. One possible outcome is that using both of these two different sources of rewards could be more effective than either alone. This is not absolutely supported by the above three experiments; although the driving style improves but not the lap time. With damage rewards only, the experiment of Figure 3-2 reaches a modest lap time but the robot has a "wobbly" driving style. The robot is avoiding damage by steering away from the sides when it gets close, but it does this without regard for the lap time, hence the unsmooth, "wobbly", driving style. Using lap time rewards only, the experiment of Figure 3-3 shows much less learning than that of Figure 3-2. Using both damage and lap time rewards in the experiment of Figure 3-4, the robot reaches almost as good a time as with damage rewards only, and the "wobbly" driving style does not occur. The robot achieves a compromise between reduction of damage and reduction of lap time, and this results in smoother driving which is closer to the style of a good human driver. The lap time continues to improve up until about lap 30,000.

Some method is needed to balance the relative influence of each source of reward. This matter is investigated in Section 5. However, using more than one reward type turns out to be more complex than simply adjusting the ratio between the scalar rewards.

3.8 Exploration

Why Exploration is Needed

One requirement for convergence in reinforcement learning is that all state-action pairs are visited an infinite number of times! However, in practice, a large number of times is sufficient [Sutton and Barto, 1998], (the Law of Large Numbers). In practice this means the robot's tendency to stay on the best-path-found-so-far needs to be occasionally disrupted. This makes intuitive sense: if the agent keeps doing only the best it has found then how can it ever find a better way that may require some risk, or temporary loss, before it is reached? This is analogous to being stuck on local maxima in hill climbing. The disruption is achieved by using "exploration", that is, occasional random actions. Too much exploration is detrimental because of the mistakes (i.e. suboptimal rewards) it can cause. A balance is needed, and this is commonly known as the "exploration / exploitation dilemma". For this reason, it is useful to decrease exploration as experience and performance increase.

The experiment shown by Figure 3-3, where only lap time rewards are given, shows no improvement after lap 600. One possible explanation for this is that exploration is needed. Exploration introduces occasional random actions, at random time steps (i.e. situations), which are suboptimal according to what the agent has learned so far, but could prove worthwhile as the agent may not have tried the action in the situation before. The experiment of Figure 3-6 makes 0.1% of all actions random and uses lap time rewards only, (the learning rate is 0.1). The action is chosen randomly, and also the time step on which to take it is chosen randomly but in such a way that on average 0.1% of all time steps use random actions. The random choice of time step ensures the exploratory move is unlikely to occur at the same track location on each lap.

Figure 3-6 shows that compared to the experiment without random actions, which is shown in Figure 3-3 (note the different y-axis scales), learning occurs no faster with exploration (random actions), and still ceases improvement after lap 600; the learning

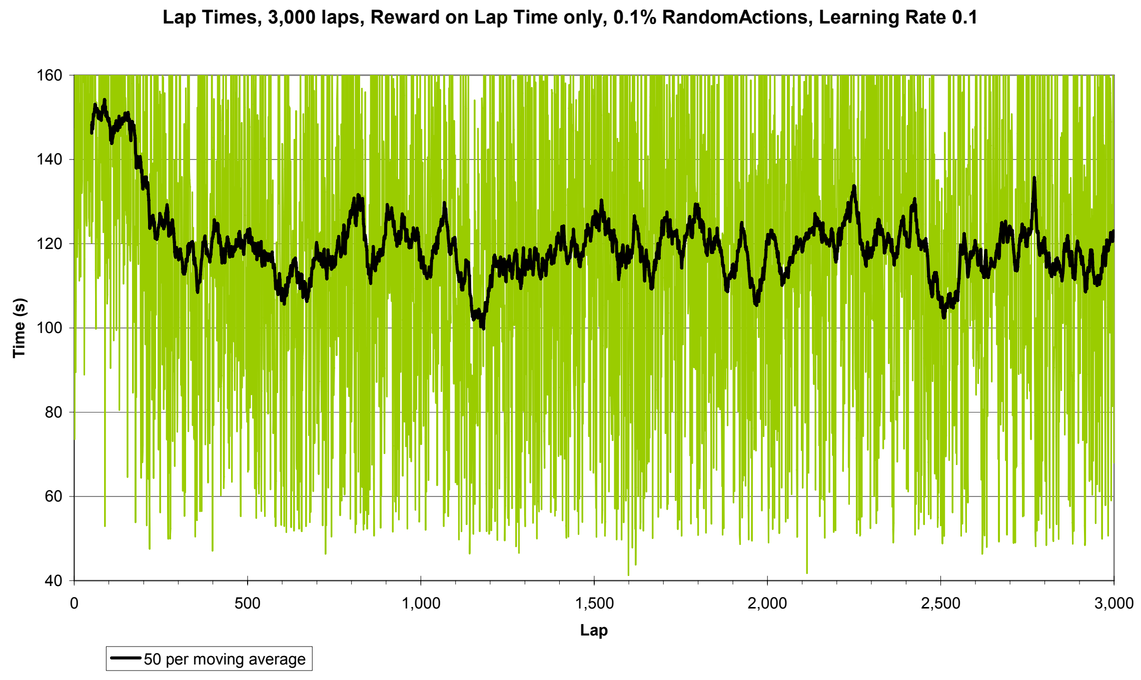


Figure 3-6 Lap Time, 3,000 laps, Lap Time Rewards, 0.1% Random Actions

reaches a poorer (slower) lap time and shows greater variation between laps which is seen by comparing the spreads in Figure 3-6 and Figure 3-3 of both the raw and averaged data. When the experiment of Figure 3-6 is extended to 10,000 laps the last 7,000 laps look like the last 2,000 laps of Figure 3-6.

The hypothesis of this experiment is that the experiment of Figure 3-3 shows no learning after lap 600 because (additional) exploration is needed. The results shown by Figure 3-6 do not support this hypothesis: they show much poorer learning performance, which is opposite to that expected.

Looking Ahead

It was later realised that in effect there is already a lot of exploration occurring without the introduction of random actions. Most of this exploration occurs because early in learning most actions have initial values, and as these are all identical the code makes a random choice between them (tie-breaking). The addition of further exploration, when only lap time rewards are used, makes performance much worse. With only lap time rewards, as shown in Figure 3-6, the learning process struggles to overcome all the randomness in the action choices. When exploration is used with both sources of rewards, as shown in Figure 3-7, learning performance improves dramatically. However, it is not clear that the use of exploration is an advantage even when both sources of rewards are used. Before this can be judged the randomness introduced by the tie-breaking method needs addressing. Improvements in the tie-breaking method are discussed in Section 3.9.2. However, before improvements to the tie-breaking method were tackled, another observation was made which is described in Section 3.8.1.

3.8.1 Exploration, when both Lap time and Damage Rewards are used

Figure 3-7 plots the results from an experiment which uses both lap time and damage rewards, and 0.1% exploration, (the learning rate is 0.1). This is the same as the experiment that produced Figure 3-5, but with the addition of 0.1% exploration. That is, the same as the experiment that produced Figure 3-6, but with both types of rewards, and run for 100,000 laps.

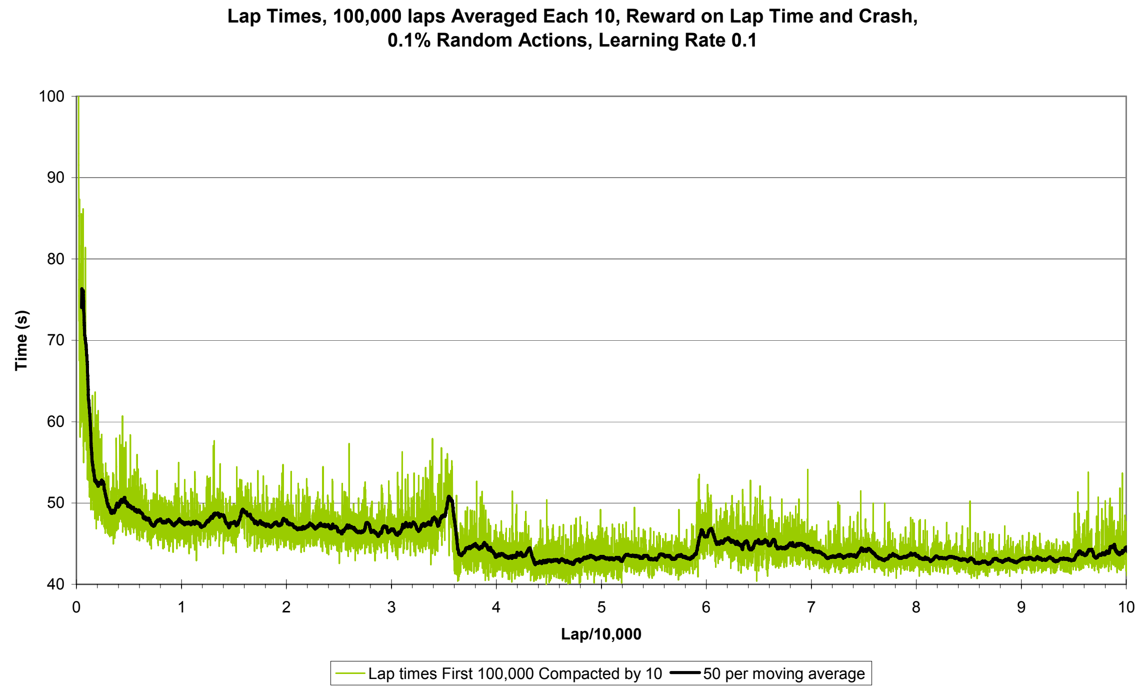


Figure 3-7 Lap Time, 100,000 laps, Lap Time and Damage Rewards, 0.1% Exploration

Compared to Figure 3-5, Figure 3-7 shows faster learning in the first 1,000 laps and to a lesser extent up to lap 10,000. By lap 20,000 both graphs show a similar lap time average of about 47 seconds. The graph without exploration levels out around 47 seconds for the next 80,000 laps. Figure 3-7, which has exploration, continues to drop and reaches an average of around 43 seconds; however it later rises again before falling. This long term rising and falling effect is illustrated over 1,000,000 laps by the bumpiness displayed in Figure 3-8. This rising and dropping in steps appears to be due to the exploration as the previous graph of size 100,000 laps (the experiment of which does not use exploration) does not show this (Figure 3-5). The sudden drop may be due to a particularly opportune exploratory move, which the agent then retains. The sudden rise in time may be due to a particularly poor exploratory move, or a series of temporally close exploratory moves (which is possible but probably rare as the exploratory moves are chosen at random time intervals which average 0.1% of all time steps, as described in Section 3.8). The agent should not retain this move(s), but if the move(s) has thrust the robot into "unknown territory" it then has to cope with a series of situations (states) it may never have encountered before. Over thousands of laps the robot slowly learns better actions in these newer states, and this is shown by the slowly falling graph from lap 60,000 onwards. This is speculation however, and was not observed. To observe this behaviour is difficult as it occurs over tens of thousands of laps and at unpredictable times. Inspection of the decisions made by the agent would involve millions of state-action-pairs and Q-values.

Long-term Trend on Lap Time (1,000,000 laps)

Figure 3-5 looks like it levels out around 47 seconds (it is a 100,000 lap graph), but, as shown by Figure 3-8, when the experiment is run for 1,000,000 laps *and* with 0.1% random actions it "levels" out around 42 seconds, although with considerable variation. The spread of the grey lines appears less in Figure 3-8, partly because the large amount of raw data from the experiment shown in Figure 3-8 must be averaged over groups of 100 data to compress it sufficiently to fit on the graph (and which is then shown by the grey lines), rather than over groups of 10 as is done for Figure 3-5. (The larger group size reduces the variance).

Figure 3-8 shows the same experiment as depicted in Figure 3-7, but for 1,000,000 laps. The long-term phenomenon appearing as "steps" in Figure 3-7 is seen as waviness in Figure 3-8, because the lap times are averaged over each 100 laps to produce the data

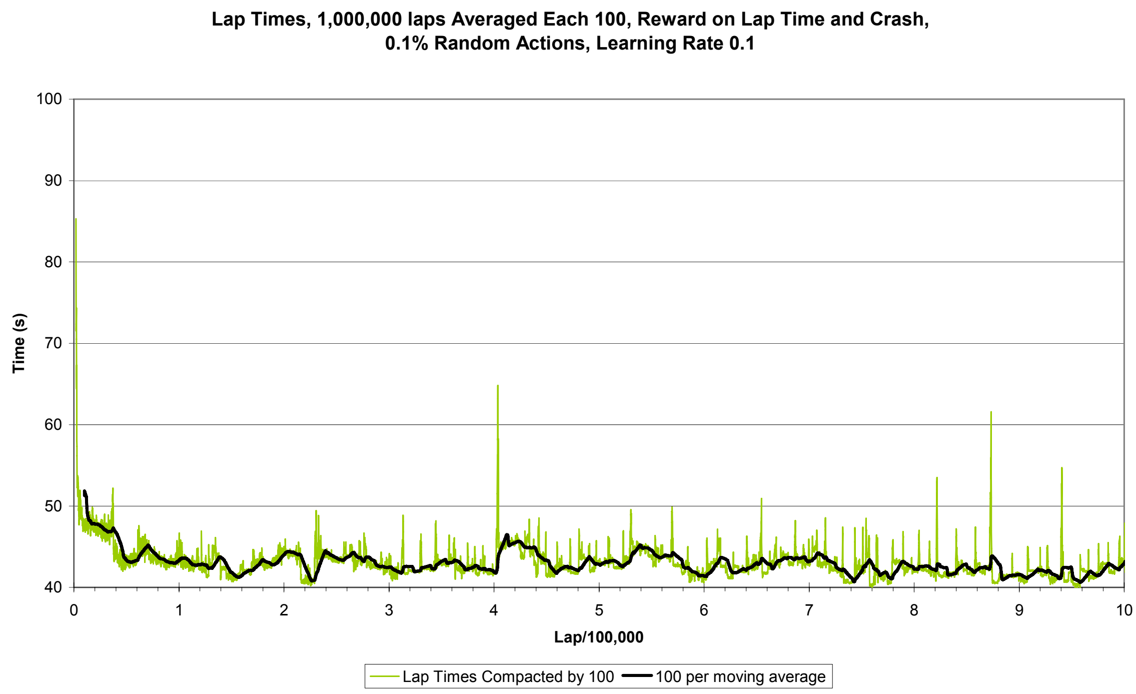


Figure 3-8 Lap Time, 1,000,000 laps, Lap Time and Damage Rewards, 0.1% Exploration

points in Figure 3-8, not 10 as for Figure 3-7; and the moving average is over 100 data, not 50. This means the moving average for Figure 3-8 is over 10,000 laps, while for Figure 3-7 the moving average is over 500 laps. This smoothes out all but the sharpest variations in Figure 3-8. Figure 3-8 appears to improve little after 150,000 laps. However, this must be viewed in conjunction with Figure 3-9, which graphs damage from the same experiment. The waviness and spikiness in Figure 3-8 could indicate excessive exploration. This matter is investigated in Section 3.9.

Long-term Trend on Damage (1,000,000 laps)

Figure 3-9 is the damage graph from the same experiment as Figure 3-8 which shows lap time. The lap times in Figure 3-8 do not improve significantly for the last three quarters of the graph, yet the damage, shown in Figure 3-9, continues to generally decline to the end of the graph.

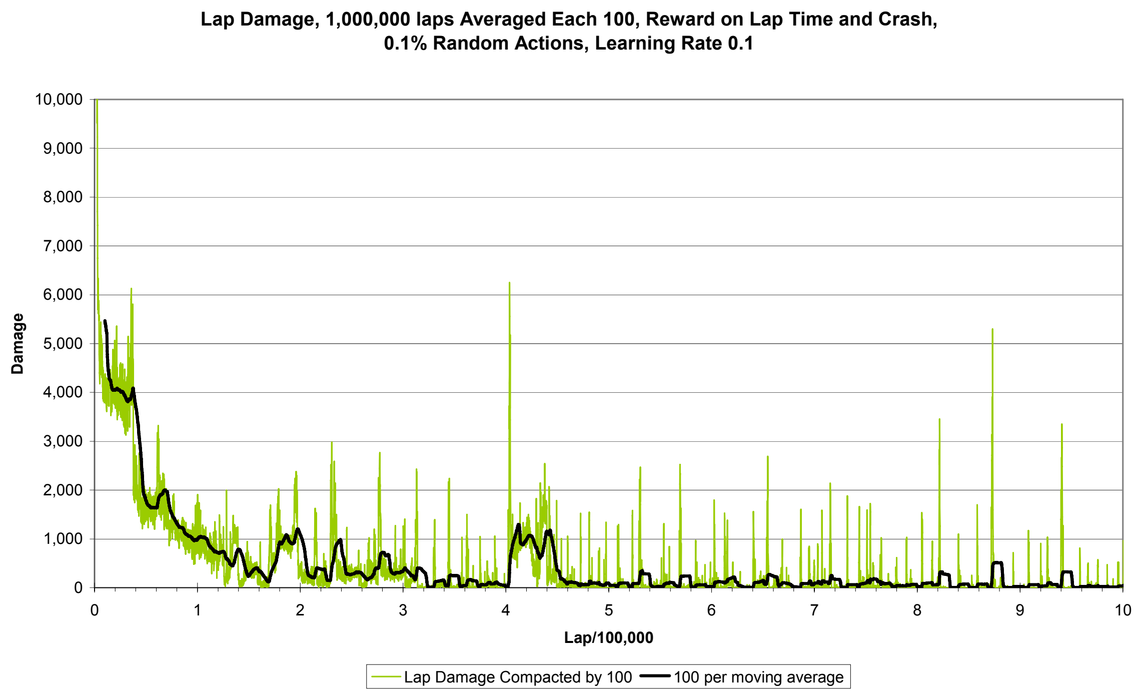


Figure 3-9 Lap Damage, 1,000,000 laps, Lap Time and Damage Rewards, 0.1% Exploration

The Trade-off shown between Lap Time and Damage

Figure 3-8 and Figure 3-9 show that for the last 750,000 laps the robot does not improve its lap time but damage continues to reduce! This is the first time this is observed. All the previous experiments, above, are illustrated with lap time graphs. The corresponding damage graphs are not shown because the shape and trend of those damage graphs closely resembles their matching lap time graph. This indicates that learning to avoid damage was the main method by which lap time was improved. The two graphs above (Figure 3-8 and Figure 3-9) show that reducing damage is no longer improving lap time, yet because the agent is rewarded for reducing damage it continues to pursue this.

3.8.2 Random Proportional Selection

As discussed in the introduction to Section 3.8 the "exploration / exploitation dilemma" makes it useful to decrease exploration as experience and performance increase. One way to do this automatically is to use Random Proportional Selection. This was implemented as follows. A time step is chosen randomly, but with a fixed probability, e.g. 0.1%. This means on average one in a thousand time steps is chosen. This is picked because a medium-speed lap on track v01.trk takes about one thousand time steps, therefore giving about one exploratory move per lap. The randomness ensures the exploratory move is unlikely to occur on the same track position on each lap. The Random Proportional Selection part of the code then picks one of the possible actions, (5 velocity discretisation steps × 5 steering discretisation steps = 25 possible actions), at random but weighted by their desirability. That is, the "better" its Q-value the more likely an action is to be chosen. This means that as one action becomes more desirable than the others it will be chosen more often. When the best action is chosen the move is no longer an exploratory one, so the amount of exploration is automatically reduced as one action comes to dominate. This also means actions that are very poor (e.g. a sudden sharp turn) will be chosen much less often than, say, the second best action, which will be chosen relatively frequently and is more likely to be worth trying. Yet, the poor actions are still tried occasionally.

The Q-values in this work are related to lap time. That is, lap time is used as a reward. Damage rewards are scaled so that a bigger crash gives a bigger reward. This means a small Q-value is a “better” one (as argued in Section 3.3). Therefore in this work the weighting used by random proportional selection to represent “desirability” is proportional to 1 / Q-value.

Accordingly:

$$P(a_i) = \frac{1/q_i}{\sum_{\forall j} 1/q_j},$$

Where, in the current state, $P(a_i)$ is the probability of action i, and q_i is the Q-value of action i. For example, suppose a state has 2 possible actions (although, states in this work on RARS use 121 actions), *x* with a Q-value of 2 and *y* with a Q-value of 3. Then:

$P(a_x) = \frac{1}{2} \div (\frac{1}{2} + \frac{1}{3}) = \frac{1}{2} \div \frac{5}{6} = \frac{1}{2} \times \frac{6}{5} = \frac{6}{10}$

$P(a_y) = \frac{1}{3} \div (\frac{1}{2} + \frac{1}{3}) = \frac{1}{3} \div \frac{5}{6} = \frac{1}{3} \times \frac{6}{5} = \frac{6}{15} = \frac{4}{10}$

This means on exploratory steps action *x* is chosen with probability 0.6 and action *y* with probability 0.4.

Random proportional selection was tried at 0.5%, 0.1%, 0.01% and none. 0.1% gives the best improvement in the fall of the learning curve when balanced against the amount of perturbation caused by exploration. This result is likely to depend on many other factors that are later changed, and so the optimum exploration rate needs to be retested from time to time.

Experiment

The experiment of Figure 3-10 uses both lap time and damage rewards and has 0.1% exploration using random proportional selection, (the learning rate is 0.1). This is otherwise the same as the experiment shown in Figure 3-7 which uses 0.1% simple random actions.

Compared to Figure 3-5, where there is no exploration, Figure 3-10 continues to decline (that is, lap time improves) after lap 30,000. Figure 3-10 looks similar to Figure 3-7 where 0.1% random actions were introduced. This indicates that random proportional selection at 0.1% has a similar effect to simple random actions of 0.1%. The learning up to lap 10,000 is slightly slower in Figure 3-10. This could be because random proportional selection

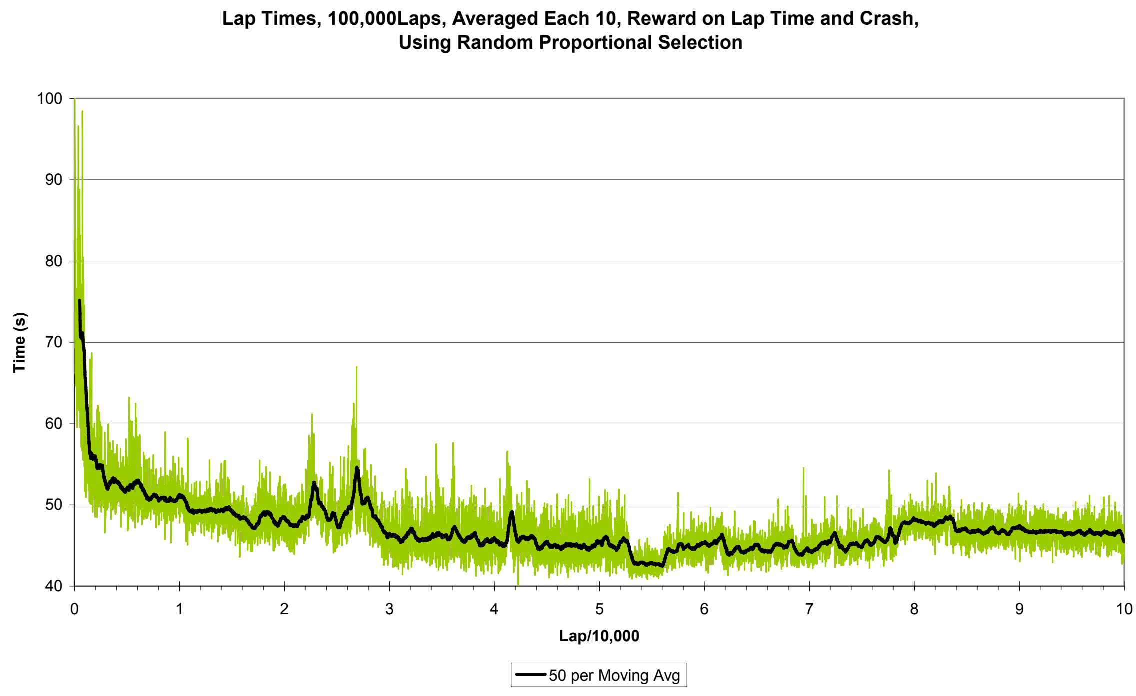


Figure 3-10 Lap Time, 100,000 laps, Lap Time and Damage Rewards, 0.1% Random Proportional Selection

includes the possibility of choosing the optimal action; therefore the actual exploration is lower than 0.1%. It is not clear from comparing Figure 3-7 and Figure 3-10 that random proportional selection has any advantages over simple random selection.

It is an advantage that random proportional selection reduces exploration with experience, because it is known that as performance improves it is beneficial to reduce exploration. Random proportional selection gives more weight to actions with better Q-values. This means more time is spent exploring the second best actions than, say, the poorest actions. For these reasons the use of random proportional selection was retained.

3.9 Non-scheduled Exploration

At one point during this work, 0.01% exploration gave better performance than 0.1% exploration. 0.01% intuitively felt very small for initial exploration. Eventually it was discovered that far more exploration than this was actually occurring. This was due to the method used to choose between equally good actions (tie-breaking), which was a simple random choice. This turned out to be in effect performing a large amount of exploration. When the methods discussed in this section were implemented the overall exploration was reduced such that it was found beneficial to increase the scheduled exploration back to 0.1%. This phenomenon reoccurred for a number of reasons after various coding changes, and thus demonstrates the significance of the effect.

3.9.1 Medium Term Effect of No Exploration, Learning Rate 0.01 (Primed learning)

It was discovered that when the *learning rate* is a small value, for example 0.01, and there is no scheduled exploration, a performance graph like Figure 3-11 occurs. That is, if run for long enough, the learning diverges. The reason Figure 3-11 levels off at about 150 seconds is that the agent can not drive any slower than this. As soon as the robot is restored to the track it turns hard and crashes off, only to be restored again by the simulator and then turn hard off and crash again, perhaps ad infinitum.

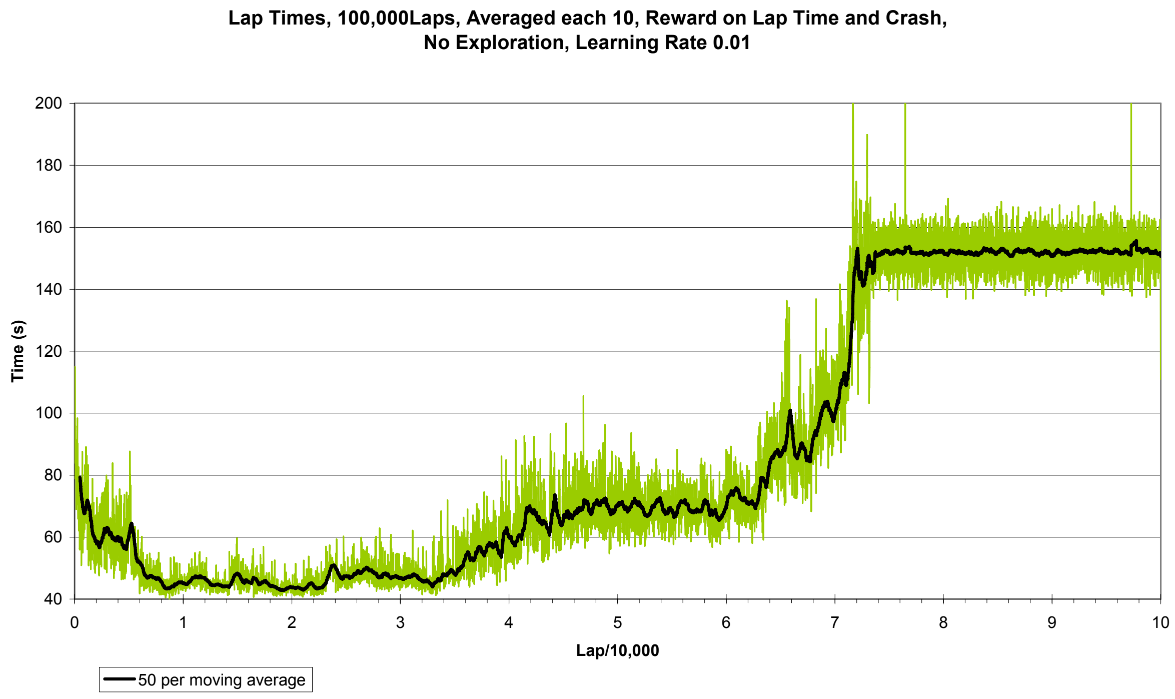


Figure 3-11 Lap Time, 100,000 laps, No Exploration, Learning Rate 0.01

In the experiment of Figure 3-11, learning up to about lap 10,000 is slightly faster than seen previously, although the variation between adjacent laps (shown by the spread of the grey lines) is wider up to about lap 5,000 (remember, the y-axis scale is 40 to 200 in Figure 3-11). This spread could be due to the low learning rate (0.01) not allowing the agent to learn the optimal actions as early as otherwise would be done.

The important point about Figure 3-11, in the context of the current discussion about exploration, is that lap time still declines for the first 10,000 laps, which indicates there must be some sort of exploration occurring, as the 0.1% scheduled exploration is not done in this experiment. The 0.1% scheduled exploration, used in previous experiments, may be excessive because early learning (up to lap 10,000) occurs faster without it in the experiment of Figure 3-11.

The divergence is due to the lack of discounting in the algorithm. This oversight allows the Q-values to grow without bounds, and so on a long episode they can overflow. The overflowed float values do not cause the program to crash, but they become meaningless nonsense and result in the divergence. This important matter is discussed separately in Section 4.3.

3.9.2 Reducing Inherent Exploration

Choosing between Equally Good Actions

An action is chosen in a state by picking the one with the "best" Q-value. Minimisation is used in this work, hence the "best" Q-value in a state is that with the lowest value. It is possible for a state to have two or more actions with identical Q-values that are the minimum in that state. When this situation occurs, some sort of "tie-breaking" method is needed.

Choosing First Best

The first method used was to simply choose the first minimum Q-value encountered when all the actions are searched. This has disadvantages, which were observed as follows. When executed, the robot is first initialised on a supervised-learnt array. On leaving the start line it travels straight for a short distance then turns hard right and crashes off the track. The track used, v01.trk, is run anticlockwise and has no right hand corners, and the teacher robot, 01.cpp, never makes right hand turns. Investigation of the states and actions showed the robot initially worked as expected, but because of the coarse discretisation of states and actions it can not drive as smoothly as its teacher, 01.cpp. Because of this it quickly reaches a slightly different state than any seen by 01.cpp during the training run (the state was different by one step in one parameter). Because this state had never been visited before, all its actions have default, (i.e. initial), Q-values. This means they are all identical. The first-minimum method of tie-breaking results in the first action being chosen. This has value (0,0) which decodes to "very slow; turn hard right". When using reinforcement learning, the robot will eventually learn that this is a poor move. However, the point in this discussion is that the supervised-taught knowledge appears to not be represented satisfactorily by the current method.

Possible improvements

One cause of the above problem is that the array could not hold enough detail to enable the robot to drive as well as the teacher robot. More resolution of detail (by using finer discretisation), or generalisation between similar states is needed. Both of these things together can work best, and are investigated in Chapter 5. The amount of memory available restricts discretisation, and if finer discretisation is implemented it is foreseeable that sooner or later the robot would still arrive in a unique state, so it would still take action "slow; hard right".

Another idea is to discretise the inputs and outputs of 01.cpp and then use it to generate the initial array. This would force 01.cpp to attempt to drive within the constrictions of the array representation. What is currently done is to run 01.cpp with continuous inputs and outputs. The data from these are then discretised and used to generate the initial array, that is, the robot 01.cpp and the simulator do not see the discretised data. However, reinforcement learning had not yet been introduced and the robot was only using initial supervised learning. It was seen that the use of reinforcement learning should allow the robot to learn by itself that "slow; hard right" is not a good action in the situation, and by using reinforcement learning it should adapt its driving style to suit the limitations imposed by the array.

Random Choice

Increasing the resolution and generalisation were tried later, and are discussed in Chapter 5. Discretising the inputs and outputs of the teacher, 01.cpp, was not tried because it was not deemed worthwhile as the use of supervised learning is a stop-gap measure that is intended to be discontinued when reinforcement learning is implemented. What was done next was to change the tie-breaking rule, from choosing the first minimum Q-value to a simple random choice. This worked much better in practice, because a large number of random choices over a group of actions (for example, all possible actions, if the actions all have default Q-values as is the case when a state has no previous visits) averages out around a central value (such as (2, 2) which decodes to "medium speed; straight ahead"). This central value tends to be more conservative than the first-minimum. The robot then makes better progress around the track, although it is very "wobbly" due to all the random action choices. This tie-breaking method is used for many experiments, including when temporal difference backups are introduced. It is shortly after this that the problem of "excessive exploration" was identified, as is discussed at the beginning of this section (Section 3.9).

Using the Mean

The coding of the actions is arbitrary. However, their order does have meaning. That is, vc=0 means very slow, and this graduates up to vc=4 which means full speed; alpha=0 means turn hard right, alpha=2 means straight ahead and alpha=4 means turn hard left. This observation leads to the idea that it could be useful to find the mean of a group of equally-good "best" actions (i.e. actions that tie for the minimum Q-value in a given state). The mean should give an action that is a reasonable compromise of the identical "best" actions. This action is likely to be conservative in that it tends to the centre of the group and so avoids extreme steering or speed commands. Random choices also average out around a central value but the distribution is flat, whereas finding the mean will always find a centre value.

The calculated mean action must be checked for having the minimum Q-value. If it does not then the nearest best action is used. The nearest best action is defined as the one with the least number of vc discretisation steps + alpha discretisation steps distance. This situation occurs, for example, if there are two groups of equally good actions either side of an action with a poor Q-value. The mean of the good actions may be the poor action. In practice this is found to occur a large proportion of the time when there are equally good actions. This is because when a group of equally good actions are first encountered in a particular state the central value is picked. This action is tried and usually turns out to be poor, simply because most actions are poor. Next time the same state is encountered there are two groups of equally good actions the mean of which is the central poor action. By choosing the nearest best action, each time the state is visited different actions are tried, starting near central values then working outwards to more radical actions, until the best action is found.

However, there can sometimes be more than one nearest best action, that is, two or more at equal distance. Again, some sort of tie-breaking is needed. If a random choice is made, the amount of non-scheduled exploration can experimentally be shown to significantly increase. The choice can be deterministic, such as the first closest, or (perhaps better) the median of those equally close. The median is used in the current implementation. The median is guaranteed to be a best action, which is not a guarantee for the mean. The median is much more expensive to find than the mean, because all the choices must first be sorted into order, (that is action order, such as hard-left and medium speed; left and medium speed; ahead and medium speed, not action-value order—all these action-values are equal). This is acceptable here because the situation of equally-close-best-actions does not occur often, and when it does occur there are only a few choices to sort.

After the new tie-breaking methods were implemented the robot's behaviour off the start line was observed. After the initial supervised learning, the robot left the start line and ran straight to the first corner. It started to turn left but then ran straight ahead on a tangent and crashed into the outside of the corner. Investigation of states and actions shows the robot reaches a state not visited by the teacher, 01.cpp, and performs the default behaviour for a previously unvisited state of medium speed, straight ahead (i.e. the mean of all actions).

General Discussion

Another tie-breaking method is to, in each state, maintain the actions in order of conservativeness, and then simply choose the first. This eliminates searching, but requires sorting and also requires prior knowledge to determine the order of conservativeness.

Using the mean makes use of information about conservativeness that is implicit in the natural ordering of the actions.

The methods described of tie-breaking between equally good optimum actions get a lot of use early in learning because at that stage most action values are unknown and therefore have default Q-values (which means they have equal values). However the issue turns out to be important at all stages of learning. Later in learning, states occur which have actions with identical Q-values that are *learnt*, not default.

The robot should cope with any of the equal minimum choice schemes described, by eventually learning the correct value of the actions. By this means, what has been called "inherent randomness" or "non-scheduled exploration" will eventually fade away as the Q-values become better known. However, an argument for the tie-breaking method that reduces non-scheduled exploration (by using the mean) is given in Section 3.9.3.

The tie-breaking method uses an average of two or more actions. It was suggested that when tie-breaking is used, the backups performed during temporal difference learning should be divided in some manner among the contributing actions. This is not correct because only the Q-value of the action actually taken can represent the value of that action. If this splitting of the backup is done a problem can occur. This happens when all contributing actions are given equal weight and all, or many, actions are contributing. For example, if a state not seen before is visited all actions have the default Q-value and all contribute to the mean action calculation. If the action receives a "poor" reward then all the contributing actions are equally "penalised". On the next visit to that state all actions still have equal value (but no longer the default value) so the same mean action is calculated. This can become perpetual, and the agent was experimentally observed becoming stuck repeatedly choosing the same "bad" actions, due to this effect. This illustrates that the temporal difference backups must be performed only to the actions actually taken. However, the idea is not without foundation. In a broad sense it relates to "selective heuristic search" as used in artificial intelligence state-space planning methods. This idea is that the search (which can correspond to back ups in the case of reinforcement learning) is concentrated around the state action pairs whose values are around the optimal. That is, the back ups are focused around the actions that appear most likely to be the best. [Sutton and Barto, 1998, 9.7, paragraph 5]. The general idea may be sound, and applicable to every back up, but the implementation used here is just too naïve.

Testing Tie-breaking-using-the-Average-of-Equally-Good-Actions

Figure 3-12 shows the results of an experiment to test tie-breaking-using-the-average-of-equally-good-actions. To reduce the amount of inherent ("non-scheduled") exploration, when there are best actions with equal Q-values, then instead of making a random choice between them, their average is calculated and the best action closest to that average is chosen. 0.1% random proportional selection is still used for "scheduled" exploration.

In Figure 3-12 the spread of the grey lines is noticeably less than *any* of the previous lap time graphs covering 100,000 laps, including Figure 3-11 which does not use exploration (remember Figure 3-11 has a different y-axis scale). Figure 3-12 shows sub-40 second lap times are reached more often than in previous experiments, for example compare to Figure 3-10. Although, medium-term variation (the spikiness of the 50-per-moving-average line) appears similar to previous graphs. This result clearly shows the "average of equal minimums" tie-breaking technique is useful. Furthermore, it is empirically observed that the driving style of the agent is much less "wobbly". This is likely to be due to the more conservative random action choices.

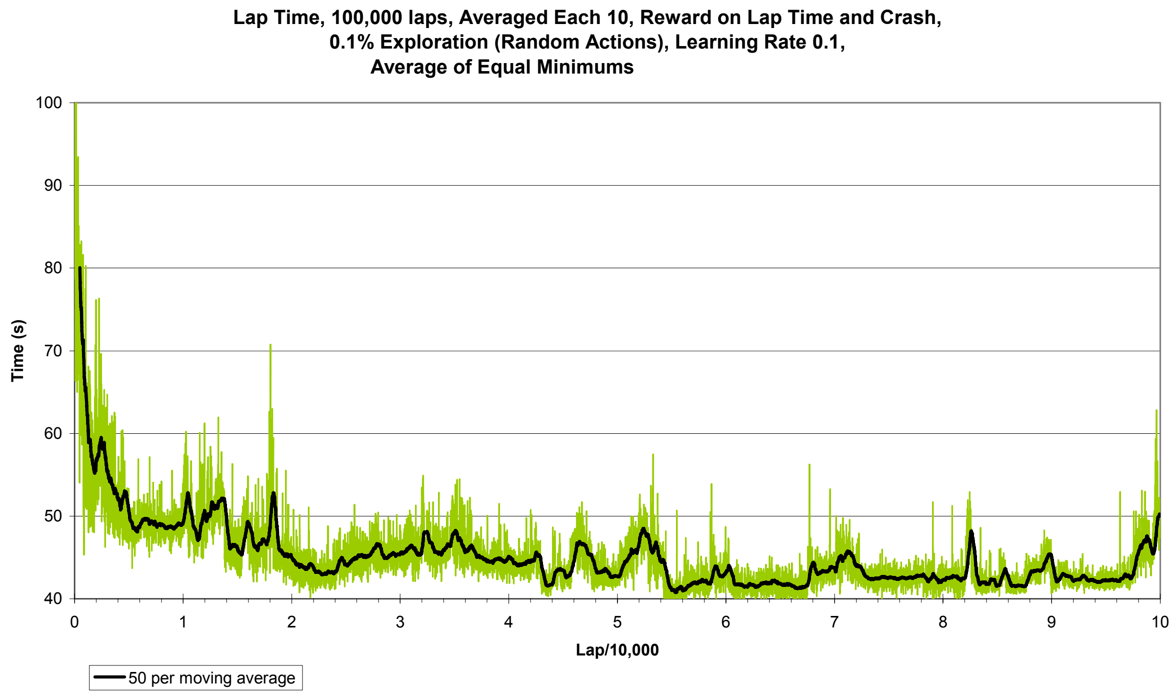


Figure 3-12 Tie-breaking, by taking the Average of Equally Good Actions

3.9.3 Shaping?

The tie-breaking method described in Section 3.9.2 is a heuristic. This is a concern as an intention of this work is to not bias learning with human domain knowledge. Although not using human domain knowledge will slow down learning (or rather, not speed up learning), it leaves open the possibility of finding solutions that are more novel than could be found if the learner is biased with prior knowledge. Instead, domain knowledge can be used for comparisons: If the agent finds similar solutions this may demonstrate its effectiveness. If the agent finds a different solution this might be useful and novel knowledge. However, it could be argued that the equal minimum choice method is not a heuristic of the driver but a missing rule of the simulator: For alpha, it roughly simulates steering self-centring, which is not provided by RARS. In the real-world this is due to suspension geometries such as castor, toe-in, tyre width, etc., and occurs when the driver does not steer. Also, a mid-value, or same-as-previous value for speed simulates coasting (the driver doing nothing with throttle or brake), whereas in RARS a low speed (such as the previous default value which was 0) usually means brake. That is, the tie-breaking method helps the simulator behave more realistically by allowing the driver to in effect give the command "do nothing", in situations where the driver does not know what to do.

3.9.4 The Wider Picture

The discussion on tie-breaking is closely related to points made in [Reynolds, 2002b; Reynolds, 2001; Sutton and Singh, 1994; Thrun and Schwartz, 1993]. Sections in these papers discuss the issue of optimistic initial values. Note that these works use the traditional max operator while this work on RARS uses the min operator. Therefore the descriptions are inverted compared to this work (i.e. over-estimation $\Leftrightarrow$ under-estimation). There are several different issues, and these are given briefly:

- The max operator can cause over-estimation when used with function approximators [Thrun and Schwartz, 1993].
- The choice of step size, α, and eligibility trace factor, λ, affect the speed at which bias in the initial action-values (Q-values) is overcome. [Sutton and Singh, 1994]. Methods are investigated for automatically setting the values of α and λ on each time step. They have considerable success.

- Optimistic initial Q-values (i.e. an optimistic bias) can cause exploration to increase and to continue for longer than is useful, therefore impeding learning progress.
- If the initial Q-values are optimistic, then the agent can not strengthen good actions—it can only weaken poor ones. This means *all* actions in a state must be visited before *any* temporal difference error can be propagated backwards, because until then $\max_a Q(s, a)$ will always be one of the optimistic values. This impedes learning progress. [Reynolds, 2001; Reynolds, 2002b].

All these issues apply to the current work, (except with regard to function approximators, as this work uses none). In particular, the issue of optimistic initial Q-values impeding learning progress applies to the present discussion on "non-scheduled exploration". The contribution of the current discussion is the observation that when an action choice needs to be made between equal optimal actions, such as occurs early in learning (when all Q-values are default) and later in learning (as occurs when the default Q-values are optimistic), then if the choice is made in a principled fashion that favours conservative actions, as opposed to the usual random choice, an increase in the speed and decrease in the variability of learning is realised.

In some domains a yard-stick for the Q-value may not be known. Therefore intentionally avoiding the problems due to optimistic initial values will not be possible, simply because it is unknown what value is optimistic. By using conservative default actions this difficulty is reduced. This, of course, assumes the domain has an action(s) likely to be conservative, and that this is known. Another bonus is that the run-time performance is improved, as observed in the smoother driving in the experiment at the end of Section 3.9.2.

3.9.5 Backups when Exploring

A backup diagram illustrating Q-learning and Sarsa is given in Appendix A, and will help clarify the discussion in this section. Two popular temporal difference learning algorithms are Sarsa and Q-learning. The back-up formula given in Section 3.4 is for Q-learning. The back-up formula for Sarsa is:

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

The difference is in the Q-value part of the target. For Sarsa it is: $Q(s_{t+1}, a_{t+1})$, for Q-learning it is: $\max_a Q(s_{t+1}, a)$. Sarsa always uses as its target the Q-value of the action actually taken in s_{t+1}, whether it is exploratory or not. This means the policy it is following and the policy it is learning about are the same thing. This is termed an "on policy" method. For this reason the amount of exploration must be arranged to eventually reach zero, that is, the policy must converge to the greedy policy, if the Q-value function is to converge to Q* (the value function of the optimal policy). This is a requirement of the convergence proof [Sutton and Barto, 1998].

However, for Q-learning the target is $\max_a Q(s_{t+1}, a)$. This is the Q-value of the optimal action, i.e. the "best" (the greedy) Q-value. For most policies this is what is normally chosen except on exploratory steps. In other words, Q-learning backs up from the optimum Q-value even on exploratory steps and not from the Q-value of the exploratory action. For this reason, eligibility traces (which are mentioned in Section 5.3) must be restarted ("cut") after an exploratory step. Q-learning is termed an "off policy" method because it is learning about the optimal policy while following a different policy, typically an ε-greedy one (i.e. involving exploration on ε of the time steps).

The algorithm used in this work is Q-learning (because it is more powerful than Sarsa in that it can learn about the optimal policy during exploratory moves). Therefore the exploration does not have to reach zero for the optimal policy to be found. It is still a good idea to reduce exploration as it disturbs what would otherwise be good behaviour later in the run (and, because it is fun to watch the driving improve in the RARS domain, the good behaviour later in the run is pleasing to observe!)

Non-scheduled exploration (that is, randomness in the choice of actions as discussed in Section 3.9.2) is not recognised by the learning algorithm, and therefore the backups that occur at these times are from the Q-values of the actions taken, which are not necessarily the optimal ones. In other words, a Sarsa-style update occurs. For the reasons given above, this means the learner can not converge to the optimal Q-value function (Q*) until this non-scheduled exploration has reduced to zero. This will happen automatically as experience is gained because the Q-values of actions become better known and so actions that previously had identical and minimal Q-values become distinguishable. However, it was empirically seen that even after millions of laps the tie-breaking method is still used occasionally, not only on actions having the default Q-value, but also on actions with identical *learnt* Q-values.

To make the choice between those actions (that retain identical, learnt, minimum Q-values over many visits to their state) appear not to be exploratory, the choice needs to be deterministic. However, this is not necessary, and might be unhelpful. This is because, even when random tie-breaking is used in those cases, the Q-learning backups are still performed correctly. This is because the actions in a state that have identical, learnt, minimum Q-values are all equally worthy of choice. That is, although the action choice may be random in these cases, the backup should still occur from the action actually taken because that action is a known optimal action. The randomness allows exploration of the equally-best-valued actions, and they may or may not eventually become distinguishable. In either case, the Q-learning method is being followed correctly.

3.9.6 Exploration needs Further Reduction in the Long-term

The eight million lap experiment of Figure 5-7, Section 5.2.3, was continued for a further eight million laps by starting a new experiment that is initially supervised-learned on the hashed array saved at the conclusion of the experiment of Figure 5-7. This is performed twice: once using 0.1% scheduled exploration, as is used in the experiment of Figure 5-7; and once using no scheduled exploration.

The y-axis of Figure 3-13 is scaled to match the y-axis of Figure 5-7. The experiment with exploration is shown as the grey line in Figure 3-13, and has intermittent spikes similar to the last three million laps of Figure 5-7. The darker line is the experiment without exploration. The first one million laps of this show some curious blips. It is not known what causes these. Perhaps it takes this long for the robot to settle into a consistent path once exploration is stopped. Yet this explanation does not seem correct because Q-learning should not have reinforced the exploratory moves. Perhaps it is connected with randomness within the RARS simulator. This does not seem likely either, because the randomness was never turned off yet the line of the experiment without exploration flattens out.

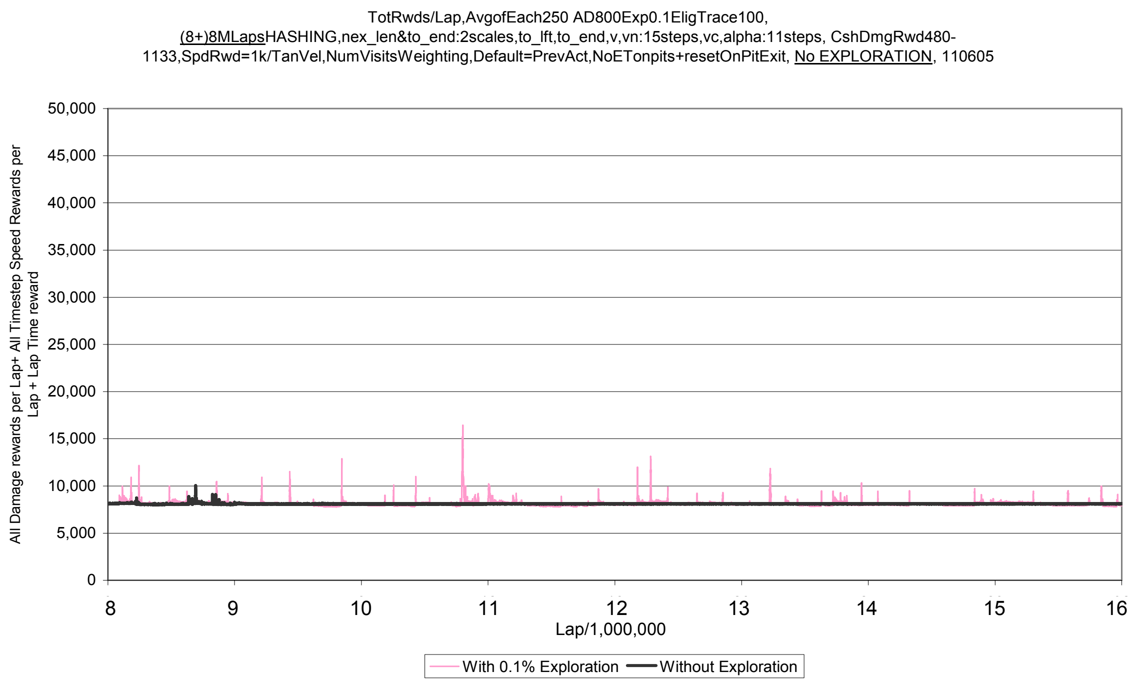


Figure 3-13 Exploration needs Further Reduction in the Long-term

Figure 3-13 shows that with 0.1% exploration and using random proportional selection there is too much exploration still occurring after eight million laps. This is shown because the gains in performance given by exploration are less than the losses in performance given by the disturbance caused by that exploration. Random proportional selection appears to not be reducing exploration sufficiently. This needs investigation and possibly modification. Alternatively, some other method of reducing exploration with experience needs to be used, and this is done successfully as shown by Figure 3-14.

The experiment of Figure 3-14 uses the experimental setup of Figure 5-7, Section 5.2.3, except 0.1% random proportional selection is used at lap one, and this is decreased once per lap by a small fraction so 0.00305% random proportional selection is reached at lap 8,000,000. (The figure of 0.00305% is used for pragmatic coding reasons).

Figure 3-14 clearly shows more consistent performance from laps 4,000,000 to 8,000,000 when compared to Figure 5-7. This shows that explicit, gradual, reduction of exploration benefits performance later in learning.

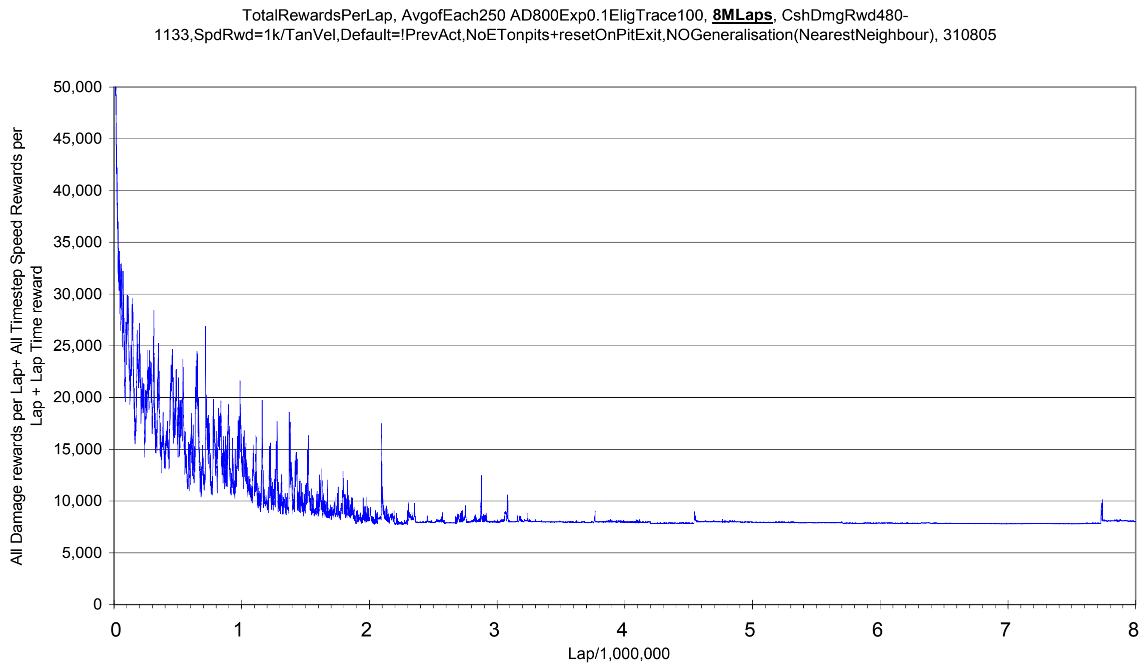


Figure 3-14 Total Rewards per Lap, Exploration Decreasing to 0.003% at Lap 8,000,000

3.10 Chapter Summary

Chapter 3 both describes experimental set-up and discusses results. This chapter begins with a justification for using a simple tabular representation and describes a method of testing it using supervised learning. The choice of state description parameters is based largely on the work of [Cleland, 2003]. The choice of discretisation step sizes is estimated, and these are readjusted after preliminary experiments. The simplest track is chosen (v01.trk), and the simplest heuristic robot is chosen (01.cpp) as the teacher for the supervised learning.

Three sources of rewards are proposed: damage, lap-time and speed. Damage rewards are derived to be approximately proportional to the size of the crash. Lap time is the ultimate metric because the declared aim for the robot is to find the smallest lap time. However, lap time rewards only occur once per lap which makes them both sparse and well delayed from the actions responsible. This makes them difficult to learn from. A tangential velocity reward is shown to speed up learning.

Temporal difference back-ups are at the heart of reinforcement learning. The idea is that if an action is taken in a state that causes the robot to move to a new state, then the value of the new state should be reflected in the value of the previous state action pair. Also, any reward received during the state transition should also be reflected in the value of the previous state action pair.

Initial experiments show learning occurring when damage rewards are used; very little learning when lap time rewards are used; and best results with both damage and lap time rewards. Exploration is needed to encourage coverage of the problem space. Surprisingly, exploration proved useful only later on in the learning process, although it needs gradual reduction as the robot's performance improves. This reduction can be achieved, to some extent, by using Random Proportional Selection. It was discovered there was already too much exploration occurring in the earlier stages of learning due to the random tie-breaking method used to choose between equally good actions. An averaging method of tie-breaking, that uses the mean (which is fast to calculate) and occasionally the median, was shown to improve learning performance in its earlier stages.

4 Unprimed Learning and Continuous Learning

When initial supervised learning is omitted then the early stage of reinforcement learning occurs more slowly but in the long term the learning produces better results. For this reason the experiments from Chapter 4 onwards do not use initial supervised learning. Chapter 4 also investigates the effect of the initial Q-value, and finds higher initial Q-values preferable because they reduce the amount of inherent (”non-scheduled”) exploration. This makes it possible to control the amount of exploration by using scheduled exploration. The work of this thesis uses very long episodes, therefore, discounting is needed to constrain the Q-value size. The choice of size of discount value appears to involve a learning-speed/learning-quality trade off, and a compromise value must be found experimentally.

4.1 Discontinuing Initial Supervised Learning

This section demonstrates that if initial supervised learning is omitted then reinforcement learning happens more slowly but improves beyond the point where it appears to plateau when supervised learning is used. That is, initial supervised learning causes a long-term bias that appears to limit the improvement obtainable by subsequent reinforcement learning, although it greatly speeds the early reinforcement learning.

According to reinforcement learning theory any initial bias should be overcome unless the learning rate, α, is decayed to zero (or in practical terms, decayed too quickly to zero), or the number of learning iterations is less than infinite (in practical terms, less than “sufficiently large”). If this happens it will prevent the algorithm from overcoming any “initial conditions”, of which the bias caused by initial supervised learning is an example. Formally, the requirement is: $\sum_{k=1 \text{ to } \infty} \alpha_k(a) = \infty$. [Sutton and Barto, 1998, page 39]. Decay of α, within certain constraints, is used because it guarantees convergence of learning; although a constant value of α is useful for following a non-stationary target. In the experiments discussed above that indicate bias, the learning rate, α, was kept constant (and not decayed with the number of visits to the state-action-pair, which is a popular method of decay used in later experiments). This fulfils the requirement on the values of α for overcoming initial conditions, except that an infinite number of laps is required. In practice a "large" number of laps is sufficient. However the 100,000 laps of Figure 3-10, Figure 3-12 and Figure 4-1 may not be large enough to show the initial bias being overcome, and experiments over millions of laps may be useful.

The experiment of Figure 4-1 does not use initial supervised learning and shows a slower learning speed until lap 1,000 when compared to Figure 3-10 and Figure 3-12 (where initial supervised learning is used). Figure 4-1 is off the y-axis scale, well above 100 seconds, prior to about lap 800. The rest of the learning curve drops in a similar manner to those seen previously, for example compare to Figure 3-7, although it does not drop as low (it never goes below 40 seconds). The long-term variance is less than that shown in Figure 3-12, which is shown by the decreased "wigglyness" of the 50-per-moving-average line. Additionally, the short-term variation (spread of the grey lines) is less than shown in *any* previous graph (including Figure 3-11, which uses no exploration)! This significant improvement in learning performance upon ceasing initial supervised learning indicates the supervised learning introduced some sort of bias. This bias is probably towards driving like the teacher robot (01.cpp), and is the reason supervised learning is used in the first place. However, the bias appears to have a long term effect, although it may possibly disappear after millions of laps, as postulated above.

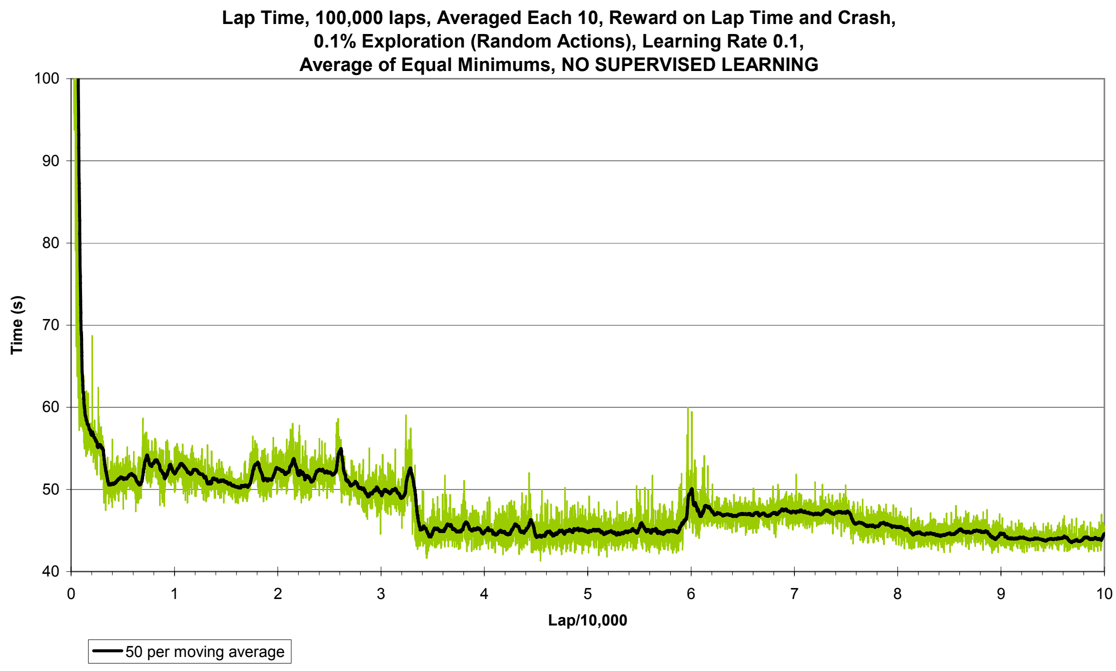


Figure 4-1 Without Initial Supervised Learning

The distinctive steps observed in the graph of Figure 4-1 may have the same explanation as given for those in Figure 3-7. The fact they occur at a similar place in both graphs (laps 32,000 and 60,000) is not explained. Perhaps the random numbers used by RARS and the exploration code within the robot had the same sequence on both experiments.

The experiment shown in Figure 4-1 shows that initial supervised learning gives a learning speed-up only over the first 1,000 or so laps and has no other clear benefits. Because of this; and in the interests of eliminating unnecessary complication from future experiments; and because supervised learning is not reinforcement learning; and because this experiment shows it is unnecessary initial supervised learning ("primed learning") is discontinued. Initial supervised learning has finished its original purpose, which was to assist reinforcement learning to work—despite the early attempts at reinforcement learning having unsuitable parameter values, such as α and γ.

Unprimed Learning, Long-term trend 200,000 laps

The experiment shown by Figure 4-1 was run for 200,000 laps, but only the first 100,000 laps are shown. Figure 4-2 shows all 200,000 laps. An observation made more apparent by the longer timescale of Figure 4-2 is the decreasing short term variance, that is, the decreasing spread of the grey lines, over the length of this graph. The reason for this is not

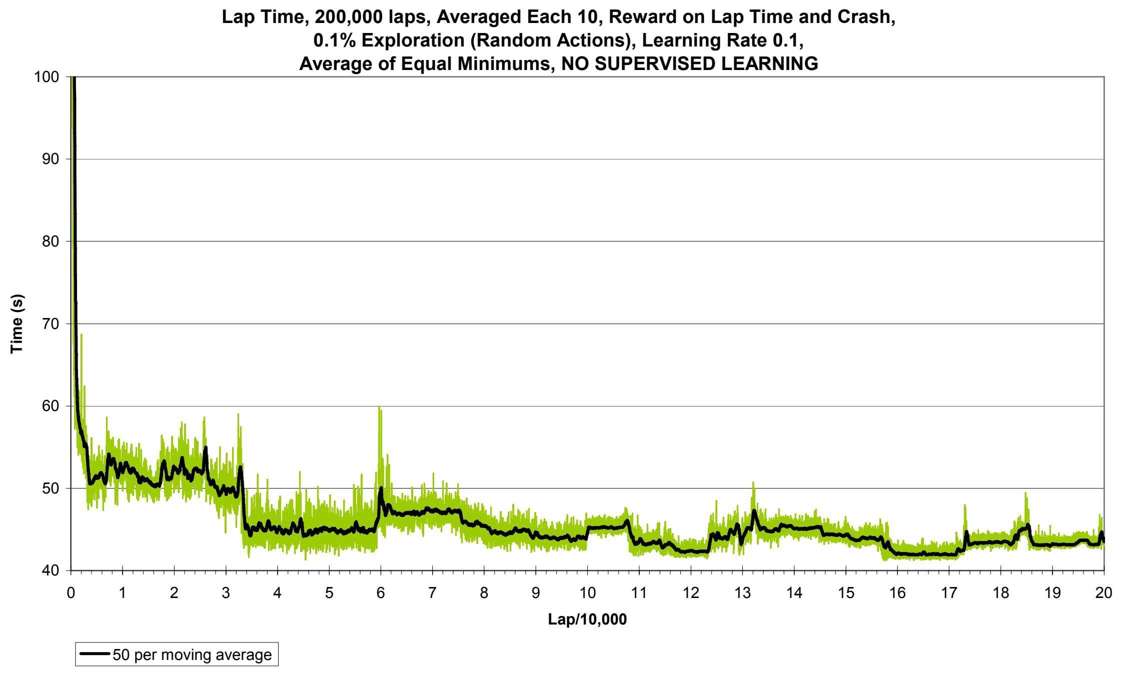


Figure 4-2 Without Initial Supervised Learning

certain. Possibly it is due to the effect of random proportional selection reducing exploration as experience increases. Or, perhaps, as experience increases, the optimality of one (or a few) actions in each state becomes better known and therefore the inherent exploration gradually decreases. That is, the ordinary, non-exploratory steps gradually become less random in their action selection as the Q-values become more accurately known. Another way of viewing this is that as the lap times reduce, so too does the variance between the lap times, and both effects are due to the agent becoming more skilled.

4.2 Effects of Initial/Default Q-value

The purpose of Figure 4-3, Figure 4-4 and Figure 4-5 is to investigate the effect of the initial (default) Q-value. The y-axis scale of 40 to 120 in Figure 4-3, Figure 4-4 and Figure 4-5 is wider than used in previous figures in order to fit the graphs. To help show any effect such as exploration, the experiments are run without exploration and with a low learning rate (0.01). As shown by Figure 3-11 and Figure 4-6, after 30,000 laps the learning diverges when no exploration and a learning rate of 0.01 is used. This problem is solved by using discounting, as discussed in Section 4.3. As it turns out, to observe the effect of the initial Q-value only the first 15,000 laps need observation and these are prior to any divergence.

The effects of initial Q-values of 400, 100 and 30 are discussed in this section. The effect of an initial Q-value of 800 is seen in Section 6.1.1 (Figure 6-2) as a reduction in exploration caused by the increased initial Q-value.

An experiment with the initial (default) Q-value set at 400 is shown in Figure 4-3. The learning rate is 0.01; there is no exploration; rewards are given on lap times and crashes; there is no initial supervised learning; and the tie-breaking method discussed in Section 3.9 is used. In the graph of Figure 4-3 the variance generally decreases as the number of laps increases. This is shown by the decrease in the spread of the grey lines. At lap 1,000 the lap time is averaging about 100 seconds. At 15,000 laps a lap time of around 52 seconds is achieved.

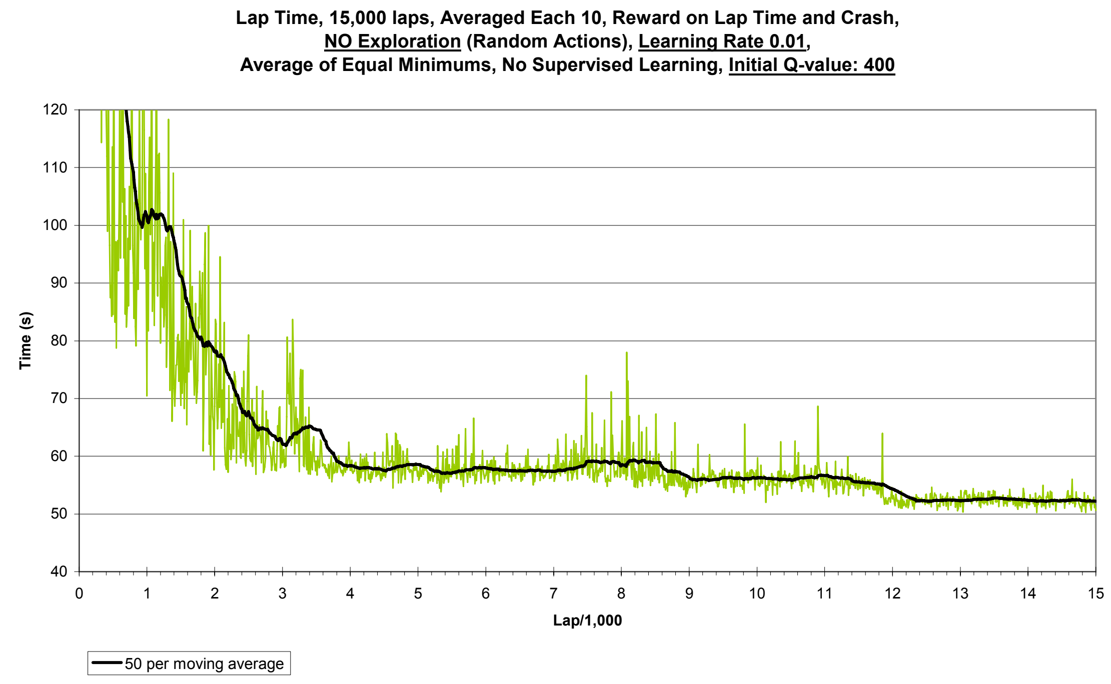


Figure 4-3 Using an Initial Q-value of 400

The experiment shown in Figure 4-4 uses initial (default) Q-values of 100. All other experimental parameters are the same as used in the experiment of Figure 4-3. Compare Figure 4-4 to Figure 4-3 and Figure 4-5. The variance, as shown by the spread of the grey

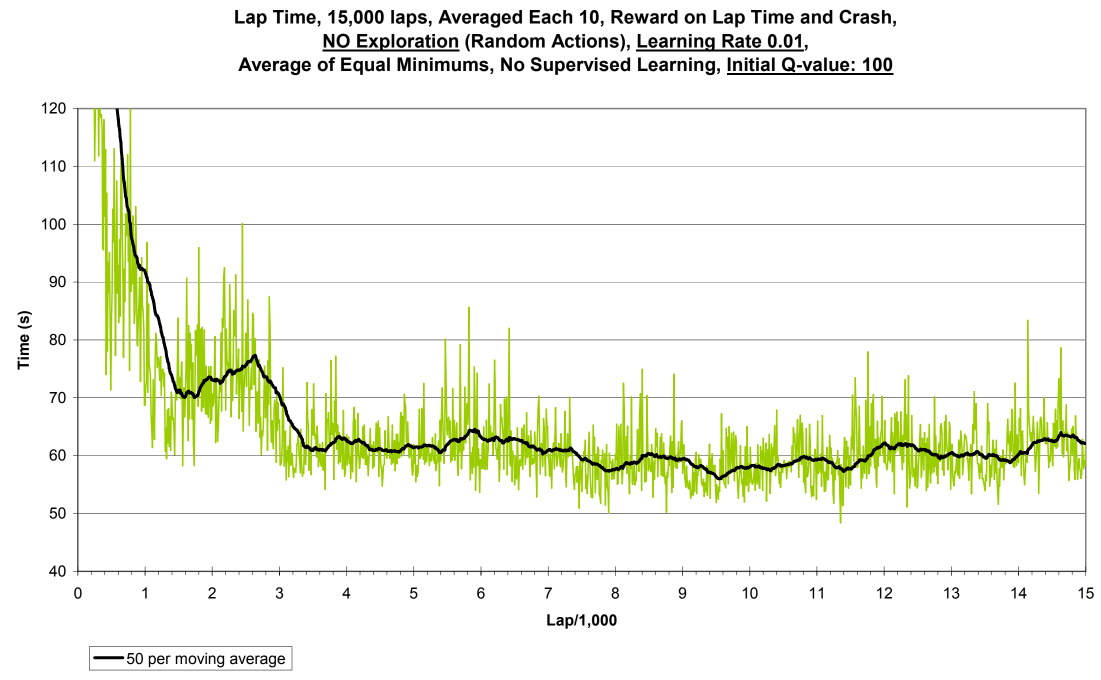


Figure 4-4 Using an Initial Q-value of 100

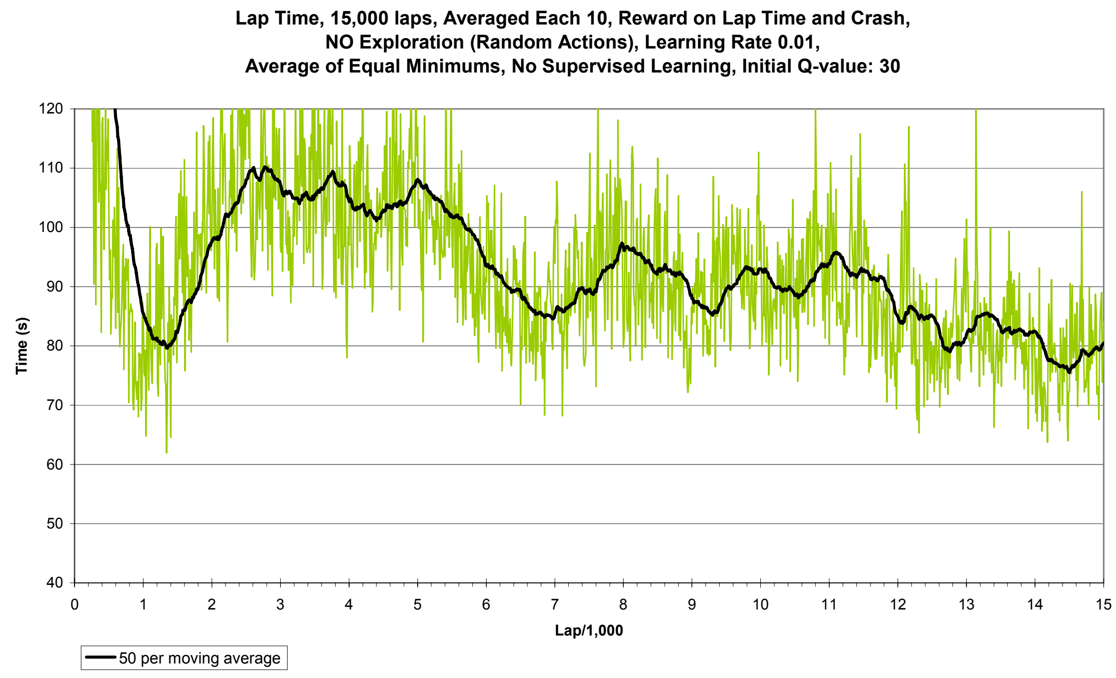


Figure 4-5 Using an Initial Q-value of 30

lines, is roughly consistent across the graph of Figure 4-4. Compared to Figure 4-3 the variance is greater. At 1,000 laps the lap time is roughly 82 seconds. At 15,000 laps the lap time averages around 60 seconds.

The experiment of Figure 4-5 has the initial (default) Q-values set at 30. All other experimental parameters are the same as used in the experiment of Figure 4-3. Compared to Figure 4-3 and Figure 4-4 the variance over the entire graph of Figure 4-5 is many times greater. At lap 1,000 the lap time is around 75 seconds (shown by the grey line—the black line of the moving average typically has a lag). At lap 15,000 the lap time is around 80 seconds.

Explanation

In summary, Figure 4-3, Figure 4-4 and Figure 4-5 show that as the initial Q-value is decreased, lap time variance increases, and the learning speed up to lap 1,000 increases (as shown by the decreased average lap time at lap 1,000); yet the learning speed up to lap 15,000 decreases (as shown by the increased average lap time at lap 15,000). Faster initial learning, but excessive variance later that ultimately slows learning, is consistent with increased exploration. Increased exploration is a likely effect of decreasing the initial Q-value because in the RARS problem domain a better Q-value is a smaller one. Therefore, by using smaller initial Q-values all actions appear as more desirable. With small initial Q-values, when a “poor” action is chosen and its Q-value is increased (made poorer) it is more likely to appear worse than the default Q-value and therefore alternative actions are tried sooner because, while the initial Q-values are lowered, the rewards are left the same.

It is concluded that decreasing the initial (default) Q-value appears to increase (non-scheduled) exploration. As discussed earlier (Sections 3.8 & 3.9), it is an advantage to control the amount of exploration, for example by using scheduled random actions. For this reason the initial Q-value is returned to 400, but is later set to 800 (see Section 6.1.1, Figure 6-2) to further reduce the amount of non-scheduled exploration.

The non-scheduled exploratory effect can be expected to naturally decrease as experience increases, as discussed in Section 4.1 (interpreting Figure 4-2) and Section 3.9.5. This means the interference with later learning, caused by too many actions that are effectively exploratory (due to smaller initial Q-values) should gradually decline. This would become evident if the experiments of Figure 4-4 and Figure 4-5 were run for longer and indeed the effect can be seen in Figure 4-8 of Section 4.3.1.

Some hypotheticals

Perhaps increased variance/randomness/inherent-exploration causes *bias* early in learning, say, up to lap 5,000, that takes many additional later laps to overcome. That is, perhaps the increased exploration from, say, laps 1,000 to 5,000 caused by the lower initial Q-values *not only* slows down the learning speed between those laps because of the increased sub-optimal action choices, *but also* causes slower learning after, say, lap 5,000, due to bias introduced by the increased randomness from laps 1,000 to 5,000 that needs to be *overcome* in the learning after lap 5,000. This same idea from another point of view is that perhaps taking a radical action early on in learning is very likely to lead to a crash. This would mark the state-action-pair responsible with a poor Q-value. If that same radical action was taken in the same state but much later in learning, after the agent had become more skilful in avoiding crashes, it might not be so poor an action, or could even lead to a better lap time. Yet if the state-action-pair had been given a poor Q-value early in learning it may then need to be chosen by a number of (scheduled, random) exploratory moves (which are rare) before the backups can improve its Q-value significantly. Alternatively, if the state-action-pair is first tried much later in learning it just might turn out to have a reasonable Q-value or perhaps even be the optimum in that state. Given "enough" experience, which could be infinite, the same results would be reached in either case. However, it is suggested that limiting exploration in early learning may "significantly" speed up learning in the "medium" term. Real-life analogies can be made, such as trying parallel parking in the first driving lesson, causing car damage and then not wanting to try parallel parking again until a time long after which the driver is probably capable of the manoeuvre! This postulation appears to be similar, but perhaps not quite the same as, ideas suggested by other authors, as discussed in Section 3.9.4.

4.3 Discounting

Discounting is used on continuous tasks to ensure the Q-value (the "reward sum", i.e. sum of rewards historically-available/expected from a given state-action-pair) is finite. It is also useful on very long duration tasks, such as the current work, to ensure the Q-value is limited to a practical size. This is achieved by exponentially decreasing the weight of more temporally distant rewards that are reachable from a given state-action-pair. This means that rewards likely to be achieved soon (e.g. on the next state transition) are worth more than rewards likely to be achieved some number of time steps after the current one. The discount factor is γ in the formula in Section 3.4, which is repeated here:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \qquad 0 \leq \gamma \leq 1$$

This formula is used on each time step, and in doing so it becomes recursive and this results in an exponential series in γ.

In general: $Q_t = r_t + \gamma Q_{t+1} \quad \Rightarrow \quad Q_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$

4.3.1 Motivation for Using Discounting

The experiment in Section 3.9.1, which is shown in Figure 3-11, demonstrates divergence after about 40,000 laps. This was seen in a number of other experiments, when there is no exploration and a learning rate of 0.01. One of these is shown in Figure 4-6 which illustrates 100,000 laps from the same experiment as used for Figure 4-3 which shows 15,000 laps. The experiment uses no exploration, a learning rate of 0.01, unprimed learning, and no discount (γ=1.0). Figure 4-6 shows the divergence phenomenon starting to occur (as discussed in Section 3.9.1 and shown in Figure 3-11). Unlike the experiment of Figure 3-11, the experiment of Figure 4-6 has no initial supervised learning, so this can be excluded as a cause of the divergence.

Discounting is Needed on Long Episodes to Constrain the Q-value Size

It was observed that in the experiment of Figure 4-6 (and in the others where learning diverges) that there are a few Q-values stored that have meaningless values. For example, some values are very large negative numbers, yet the algorithm does not produce negative Q-values. This shows that some Q-values grow and overflow the representational abilities of the storage type in use (a C++ float). This, unfortunately, does not cause the program to crash, but does cause unexpected behaviour. That is, some of the absurd Q-values (that ought to be large, poor values) are interpreted as being small, and hence the actions associated with them are chosen as being optimal. These actions result in further large (poor) rewards, and so on. From this point on there is a self perpetuating collapse of the algorithm, causing the learning to diverge, as seen in Figure 4-6.

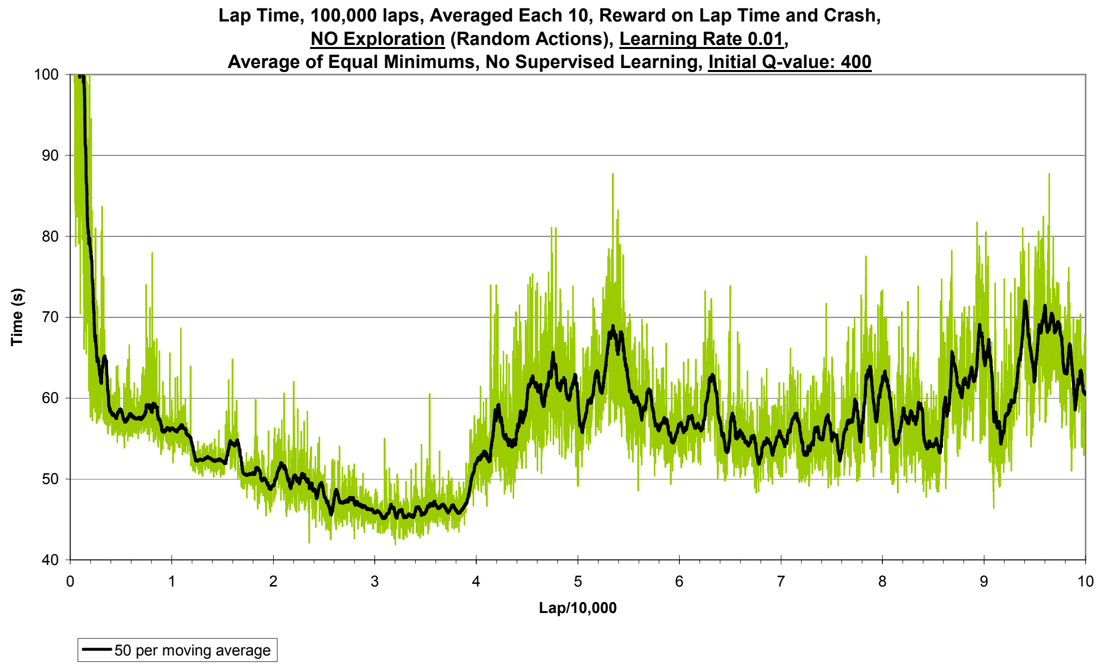


Figure 4-6 Divergence when: No Exploration; Learning Rate 0.01; Initial Q-value 400; γ=1.0

A solution to this problem is to treat the RARS domain as a continuous task, not an episodic task. Each run of the agent is not infinite, but it is often a very long episode, (e.g. millions of laps). On such long episodes there are a few state action pairs, leading to crashes (these give high rewards), which get chosen a number of times, perhaps due to exploration or randomness in the environment. The expected-sum-of-future-rewards (Q-values) of these state action pairs are very large. In the continuous implementation, discounting is used to ensure such sums remain finite (or, in practical terms, below a certain threshold).

(Surprising) Effect of Increasing the Learning Rate

As an alternative to discounting, increasing the learning rate prevents the divergence occurring. This is unexpected and remains unexplained. When the learning rate is increased to 0.1 a graph such as Figure 4-7 results. The experiments of Figure 4-6 and Figure 4-7 are otherwise identical.

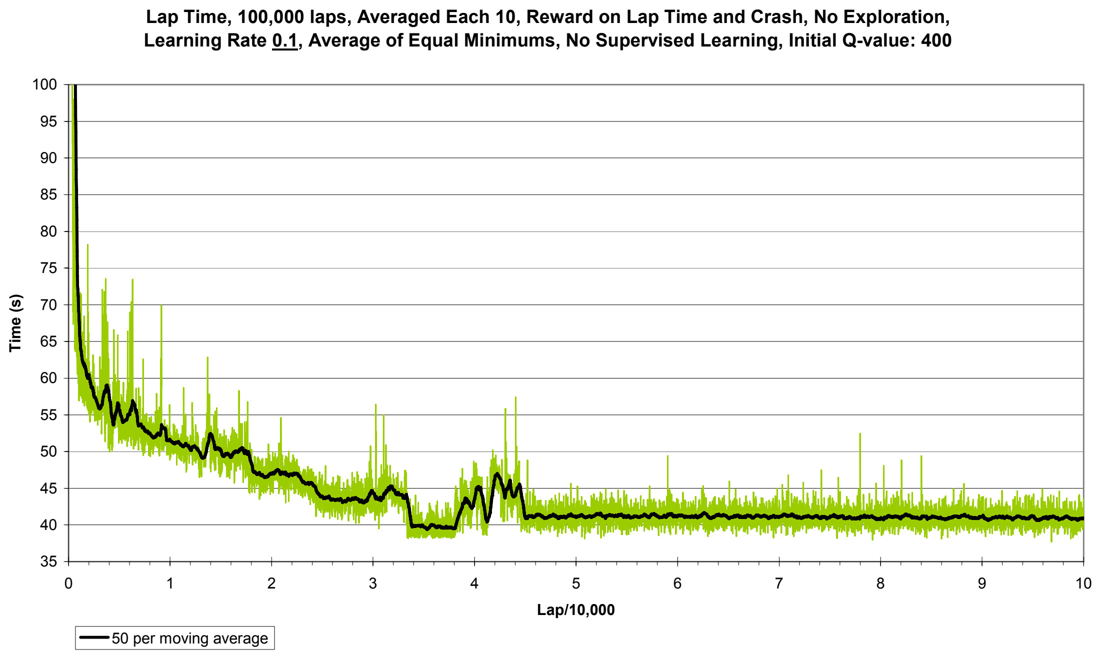


Figure 4-7 No Divergence when: No Exploration; Learning Rate 0.1; Initial Q-value 400; γ=1.0

The first part of Figure 4-7, up to about lap 200, shows lap time decreasing more rapidly compared to Figure 4-6, while up to about lap 3,000 it has a similar gradient. Figure 4-7 reaches a lower minimum than Figure 4-6, which may indicate the higher learning rate has improved learning performance. The remainder of Figure 4-7 avoids the instability shown in Figure 4-6, and this freedom from instability is demonstrated in the long term by Figure 4-8. Figure 4-7 and Figure 4-8 are more or less flat from lap 50,000 onwards, indicating that performance is not improving. The agent has reached a fixed range of actions. At this point the agent is not yet driving to its ultimate ability (e.g. it still has regular crashes). This indicates that exploration is needed to force the agent to take risks and thereby possibly find some better new actions.

The higher learning rate in the experiment of Figure 4-7 appears to prevent the type of decline in performance as seen in the second half of Figure 4-6. This could be due to the higher learning rate improving the learning performance, which is indicated by the steeper negative gradient and lower minimum reached in Figure 4-7, compared to Figure 4-6. The slow learning of crash avoidance in the experiment of Figure 4-6 means the agent will incur more damage rewards. Yet, because of the small learning rate (α=0.01) in the experiment of Figure 4-6, the Q-value of the state action pair prior to a crash is increased by only a very small amount. Therefore it is difficult to see how a small learning rate (which slows learning speed, as expected) can lead to divergence in learning (seen in the second half of Figure 4-6) which appears to be caused by Q-values that become meaningless because they overflow the machine representation. This remains an open question in this work.

An Incidental Observation

The experiment of Figure 4-7 was run for much longer (one million laps) to check that it does not diverge in the long term, and is shown in Figure 4-8. This led to an incidental observation. The data is averaged over blocks of 100 laps, not 10 laps as in the previous figure, and this makes the spread of the graph appear narrower. Figure 4-8 shows how "inherent exploration", as discussed on page 65 of Section 3.9.2, can be seen to cease when given enough time, (and also providing there is no "scheduled" exploration, the learning rate is 0.1 and the initial Q-value is 400). That is, the act of choosing between equally good (default) Q-values in early learning provides enough variety of actions ("inherent exploration") for the agent to find some modestly good actions, with better than the default Q-value.

Without "scheduled" exploration, the agent uses greedy action selection and therefore once one good action (with better than the default Q-value) is found in each state visited it no longer has to choose between equally good (default Q-valued) actions and then becomes stuck following a fixed but suboptimal policy. This happens at about lap 50,000, where the graph levels off. At this point "scheduled" exploration is needed to force the agent to take risks and thereby possibly find some better new actions. This levelling-off does not happen in Figure 4-2 (200,000 laps) where exploration is 0.1%, nor in Figure 3-8 (1,000,000 laps) where exploration is 0.1%, nor when an experiment was run for 200,000 laps with exploration of only 0.01% (graph not shown). The main point made here about the experiment of Figure 4-8 (which does not use scheduled exploration) is that this is the first time the levelling off has been seen since the experiment of Figure 3-5 ("scheduled" exploration was introduced after the experiment of Figure 3-5) and this supports the idea put forth in Section 3.9.2 about the existence of "inherent exploration". The minimum (which occurred at about lap 40,000) is not retained, because lap time is only one of the rewards the agent is trying to minimise.

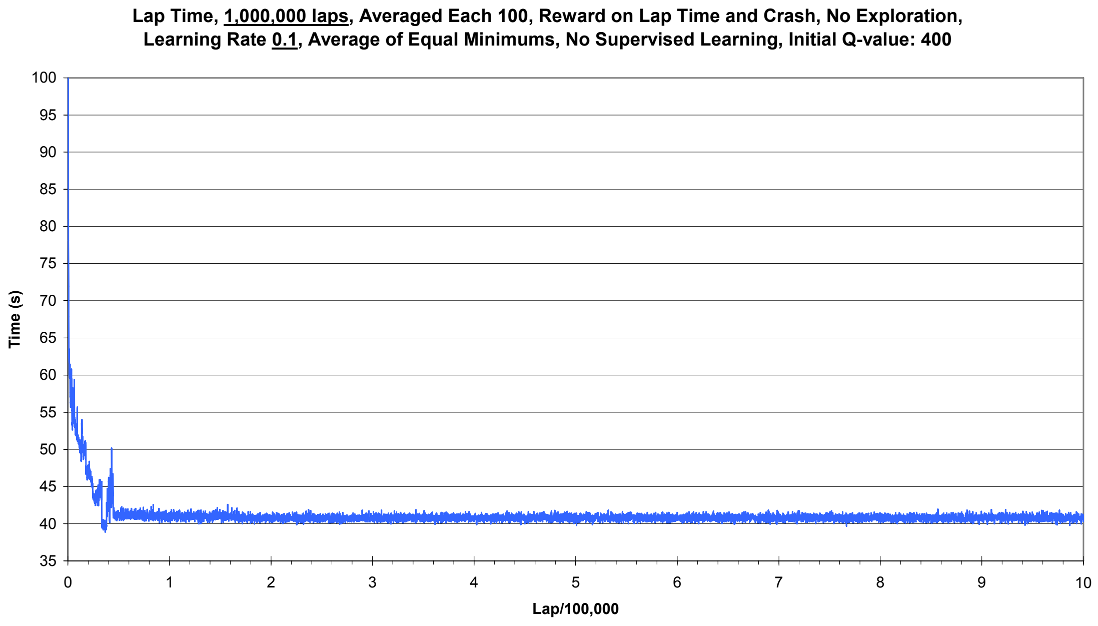


Figure 4-8 "Inherent Exploration" Dissipates

4.3.2 Discount of 0.99

A discount (γ) value of 1.0 gives no discounting. This is because each Q-value is not decayed each time it is used to update the previous Q-value during temporal difference backups, and so the Q-values can theoretically become infinite on infinitely long tasks. Any γ value between zero and 1.0 gives a discounting effect, and the closer it is to zero the greater the discounting effect. Typical practical values are often around 0.99, but this depends on the problem domain. Values nearer zero make the agent more "short sighted", that is, rewards more temporally distant from the current state action pair are given less weight; values nearer 1 make the agent more "far sighted".

The experiment shown in Figure 4-9 uses a discount of 0.99, whereas the experiment shown in Figure 4-6 uses a discount of 1.0. The two experiments are otherwise identical (no exploration, learning rate 0.01, and unprimed learning). Note that Figure 4-9 shows 1,000,000 laps, while Figure 4-6 shows 100,000 laps.

Figure 4-9 shows that a discount value of 0.99 has slowed down the learning: the curve falls more slowly over the first 100,000 laps. Figure 4-9 is clearly not diverging. The lap times become consistent at around 61 seconds, and never get as good as the best lap times

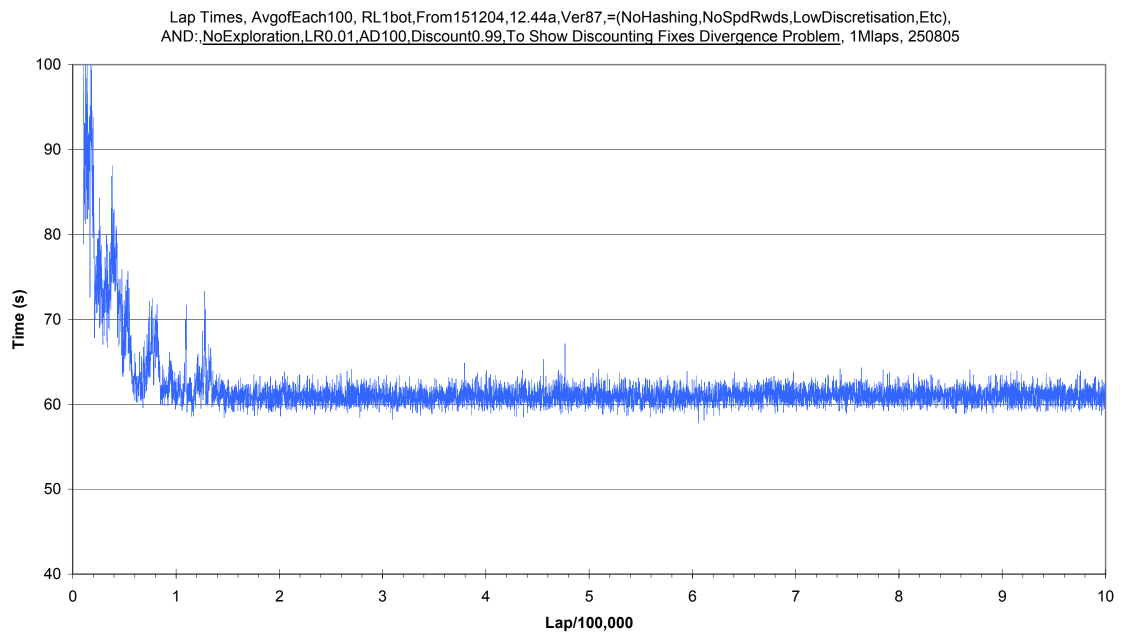


Figure 4-9 No Divergence when: No Exploration; Learning Rate 0.01; Discounting of 0.99

shown in Figure 4-6. This may indicate a larger value of γ is needed; that is, this domain may favour far-sightedness. However, the divergence shown in Figure 4-6 does not occur in Figure 4-9. This is thought to be due to discounting preventing the Q-values from overflowing the limit of the machine representation.

The experiment shown in Figure 4-10 and Figure 4-11 uses a discount of 0.99. Also, exploration is returned to 0.1%; the learning rate is returned to 0.1; and initial supervised learning is not used.

Compare Figure 4-10, the experiment of which uses discounting, to Figure 3-8 Section 3.8.1, the experiment of which does not use discounting, (and note that Figure 3-8 is of an experiment of one million laps). The learning in Figure 4-10 is slower, by about three times, and reaches a lap time of around 43 seconds after 300,000 laps, after which the lap time then degrades, whereas Figure 3-8 shows a lap time of about 41 seconds after about 150,000 laps. The use of discounting appears to have degraded performance. However, the damage graph from the same experiment (Figure 4-11) sheds a different light.

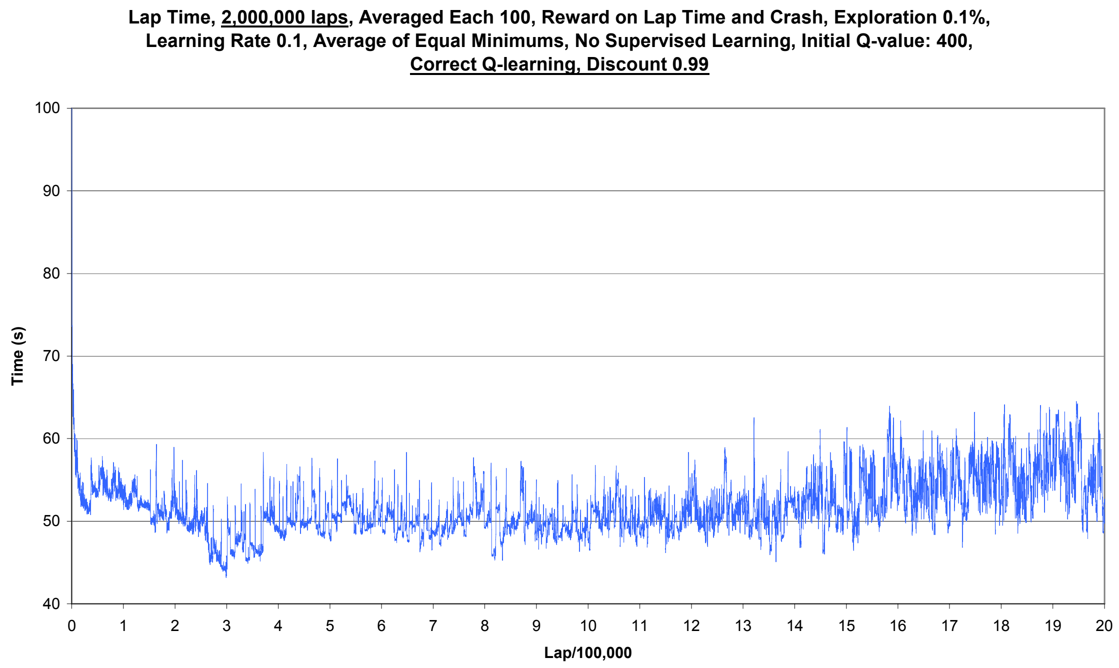


Figure 4-10 Discount of 0.99: Lap Time Effect

Figure 4-11 shows the damage per lap from the same experiment as shown in Figure 4-10. The learning in the experiment shown by Figure 4-11 (which uses discounting) is about three times slower than that shown in Figure 3-9 Section 3.8.1 (which does not use discounting). This is the same as the observation made when comparing the respective time graphs. Figure 4-10 and Figure 4-11 show that damage first reaches zero when the fastest lap time is set. From this time onwards the average damage continues to decline (i.e. often being zero); yet the lap time rises. This demonstrates a similar trade-off between lap time and damage to that shown in Figure 3-8 and Figure 3-9. On the other hand, the set-up where γ=1, as used in the experiment of Figure 3-8 and Figure 3-9, can result in the divergence phenomenon shown in Figure 3-11 and Figure 4-6; but Figure 4-9 shows divergence does not occur when γ=0.99, the same γ value used for the experiment of Figure 4-10 and Figure 4-11. In summary, discounting appears to slow learning but prevents the divergence phenomenon, and so allows the experiment to usefully run for many more laps.

Discounting prevents the Q-values growing without bound, but it also makes the agent more short-sighted, that is, Q-values represent expected return over a shorter time scale. The decrease in learning speed that occurred with discounting may indicate a larger value of γ is needed. This is tested in Section 4.3.5.

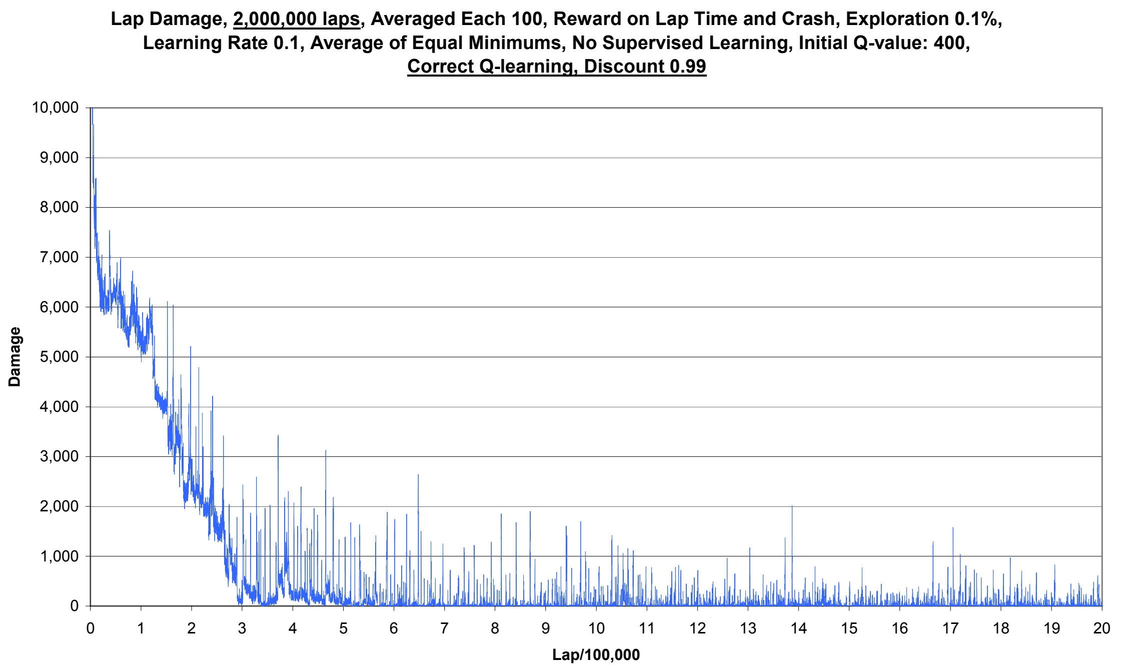


Figure 4-11 Discount of 0.99: Damage Effect

4.3.3 Discount of 0.9

The experiment shown in Figure 4-12 uses all the same parameter values as that of Figure 4-10, except the discount (γ) which is 0.9. Compare Figure 4-12 to Figure 4-10 where the discount used is 0.99, but note the different y-axis scales. Figure 4-12 shows much slower learning but appears more stable. However, the experiment of Figure 4-12 is predicted to diverge after many more laps (to follow the pattern shown in Figure 4-10, but over a longer time scale). This proves to be the case, and this is shown by Figure 4-14.

Figure 4-10 / Figure 4-11 (from an experiment with discount 0.99) and Figure 4-12 / Figure 4-13 (from an experiment with discount 0.9) suggest trying a larger discount of, say, 0.995 to see if convergence of the early parts of the time and damage graphs would occur sooner. However, this must wait until Section 4.3.5 (Figure 4-19).

Figure 4-13 shows the damage per lap from the same experiment as used for Figure 4-12. A discount of 0.9 is used in that experiment. In comparison to Figure 4-11 (discount of 0.99; and note the different y-axis scales used) the damage graph descends more slowly, and is predicted to continue declining because the lap time graph from this experiment

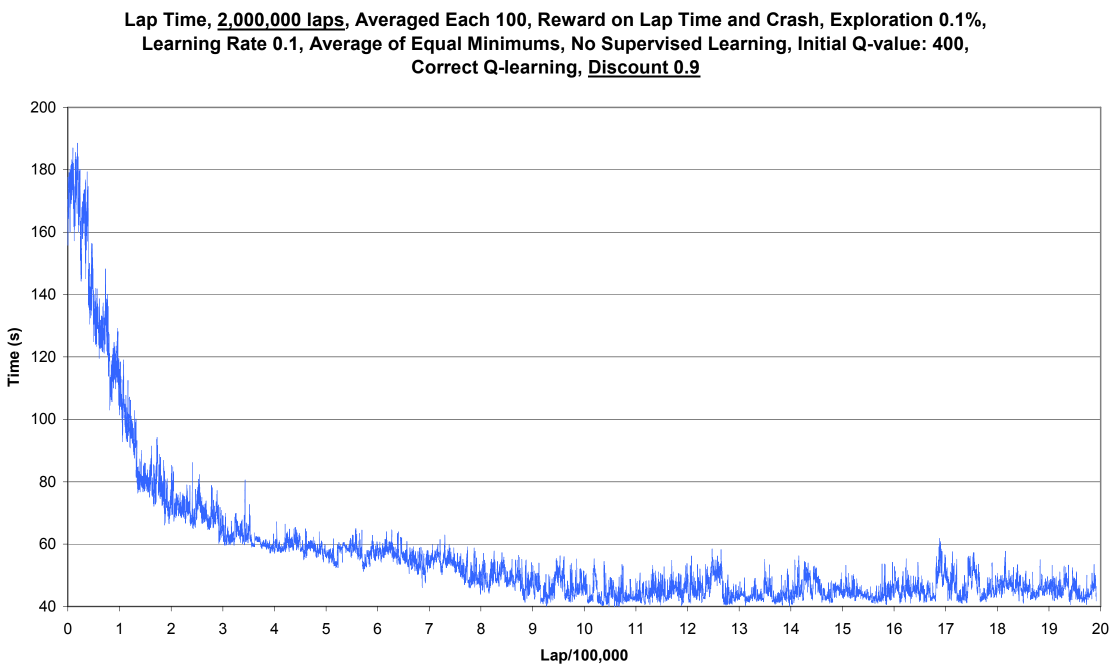


Figure 4-12 Discount of 0.9: Lap Time Effect

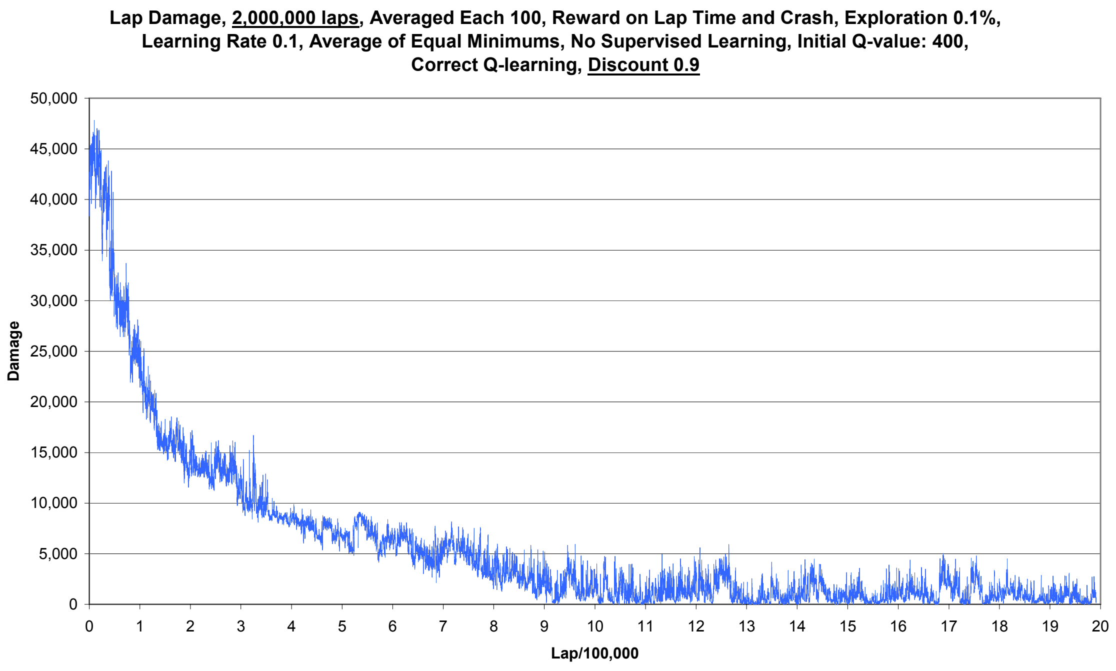


Figure 4-13 Discount of 0.9: Damage Effect

(Figure 4-12) does not start to diverge (in the manner seen in Figure 4-10). That is, the experiment of Figure 4-12 / Figure 4-13 is predicted to follow the patterns seen in Figure 4-10 / Figure 4-11, but over a longer time scale. This proves to be the case, as is shown in Figure 4-15.

From theory it is known that a smaller discount value makes the agent more "short sighted". This means the expected rewards in the more distant future (physical distance down the track) are given less weight when estimating the value of the current state-action-pair. A comparison of Figure 4-10 and Figure 4-11 to Figure 4-12 and Figure 4-13 shows that in the RARS domain this results in slower learning.

Very Long-term Trend, Discount of 0.9

Figure 4-14 shows the time per lap for an experiment like the one used for Figure 4-12 (using a discount of 0.9), but run for 40,000,000 laps. Figure 4-12 looks like it may continue to decline, but as predicted, Figure 4-14 shows that when run for a very large number of laps (that is, 40,000,000) the lap time graph starts to rise and spread, in the same way as shown by the lap time graph Figure 4-10 which uses an experiment with a discount of 0.99.

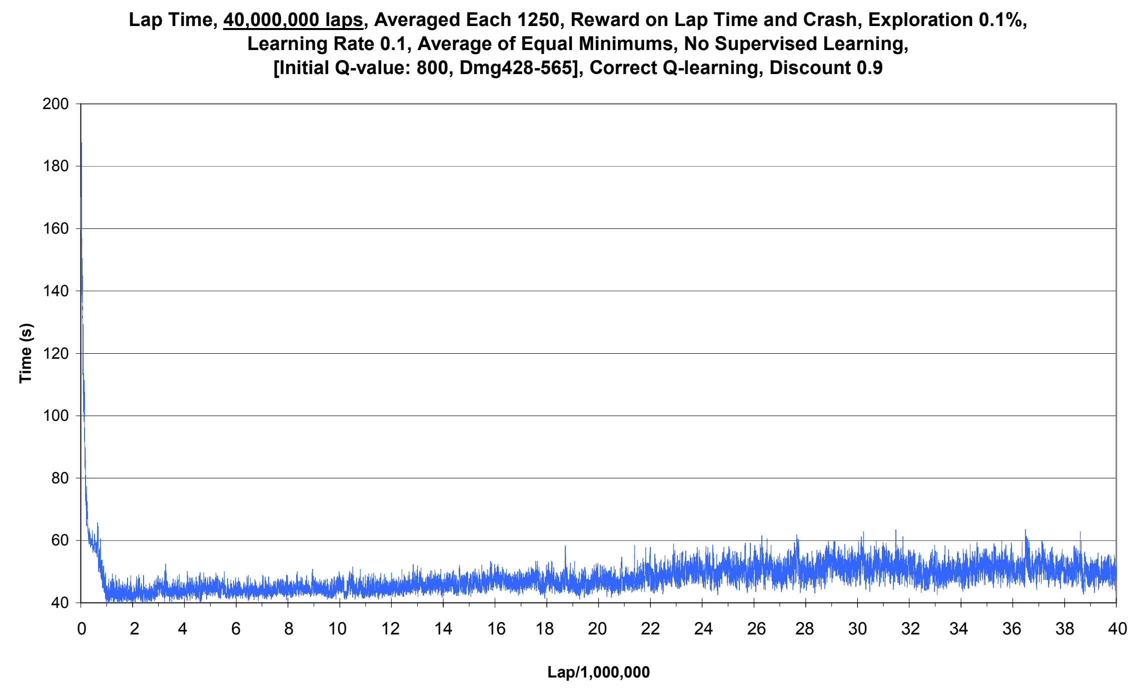


Figure 4-14 Discount of 0.9: Lap Time Effect across 40,000,000 Laps

Figure 4-15 shows the damage per lap from the same experiment as used for Figure 4-14, (using a discount of 0.9, and run for 40,000,000 laps). Figure 4-13 was predicted to continue declining, and when the experiment is run for 40,000,000 laps Figure 4-15 shows this to be the case. Damage becomes very rare towards the end of Figure 4-15. Damage

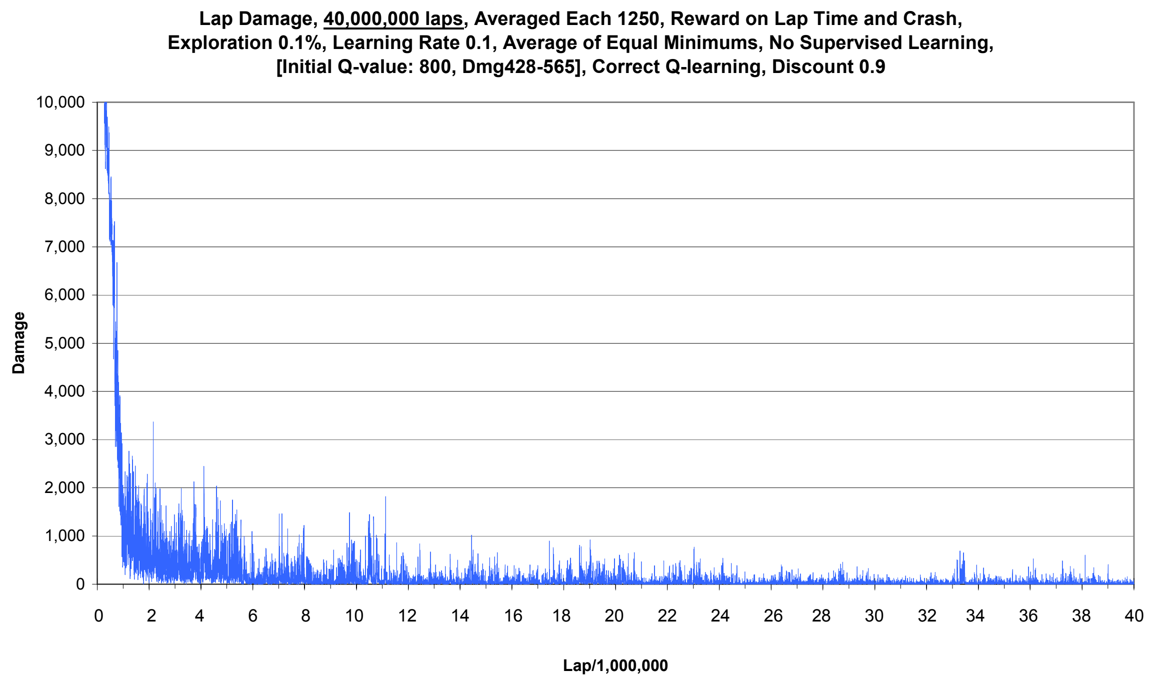


Figure 4-15 Discount of 0.9: Damage Effect across 40,000,000 Laps

appears more frequent than it may actually be, because of the averaging over blocks of 1,250 laps. This means only 1 lap in 1,250 need have damage for it to show on the graph, the other 1,249 laps could be damage free.

Figure 4-14 and Figure 4-15 show that, despite the slow learning when a discount of 0.9 is used, the learning continues to at least lap 40,000,000. An increase in lap time is being traded off against a decrease in damage, over the last 39,000,000 laps. What would be useful for judging the overall learning progress is a graph showing the *total* rewards per lap, that is, damage-per-lap plus time-per-lap. That is what is graphed in the figures following Figure 4-15.

4.3.4 Discount of 0.8

Figure 4-17 is a graph of total-rewards-per-lap, that is, total-damages-per-lap plus time-per-lap, and uses a discount of 0.8. The experiment of Figure 4-17 also involves an increase in the range of the size of the crash damage reward. This is discussed in Section 5. To assist in making Figure 4-17 meaningful, a control (baseline) experiment is run with identical parameters to the experiment used for Figure 4-17, except for having a discount of 0.99, and this is shown in Figure 4-16.

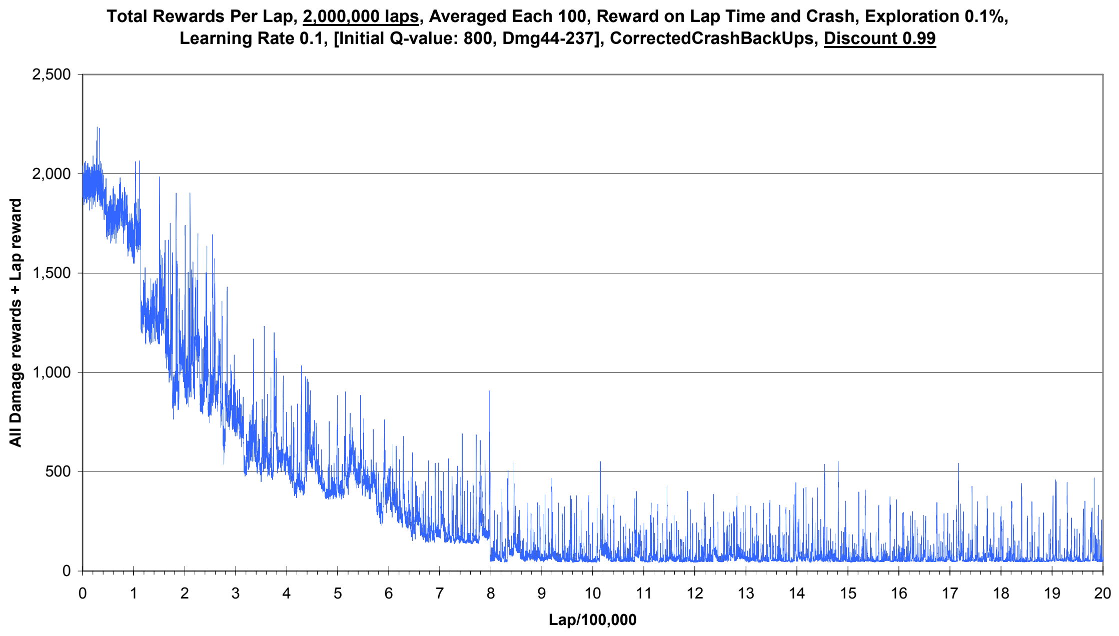


Figure 4-16 Discount of 0.99: Total-rewards-per-lap Effect (Baseline for Figure 4-17)

When Figure 4-16 is compared to Figure 4-10 and Figure 4-11 it can be seen to reach a low point or turning point about three times the number of laps later. This shows slower learning given lower damage rewards than observed in Figure 4-10 and Figure 4-11. This is despite both experiments using a discount of 0.99. Altering damage rewards is discussed in Section 5. However, the current purpose of Figure 4-16 is to serve as the control (i.e. baseline) for Figure 4-17.

One other side-issue needs addressing: Figure 4-10 shows the lap time increases later in the experiment, while Figure 4-11 shows the damage continues to decrease throughout the experiment. It is thought that increased lap time is being traded off against decreased damage, such that the total rewards are still decreasing. This is more-or-less confirmed by the total rewards graph of Figure 4-16, which shows total rewards level-out in the second half of the graph, but do not increase.

Figure 4-17 shows an experiment using a discount value of 0.8, but otherwise identical to the experiment used for Figure 4-16 which uses a discount of 0.99. Comparison of Figure 4-17 with Figure 4-16 shows that a discount value of 0.8 leads to slower learning (note the very different y-axis scales) than a discount value of 0.99. With difficulty and caution, due

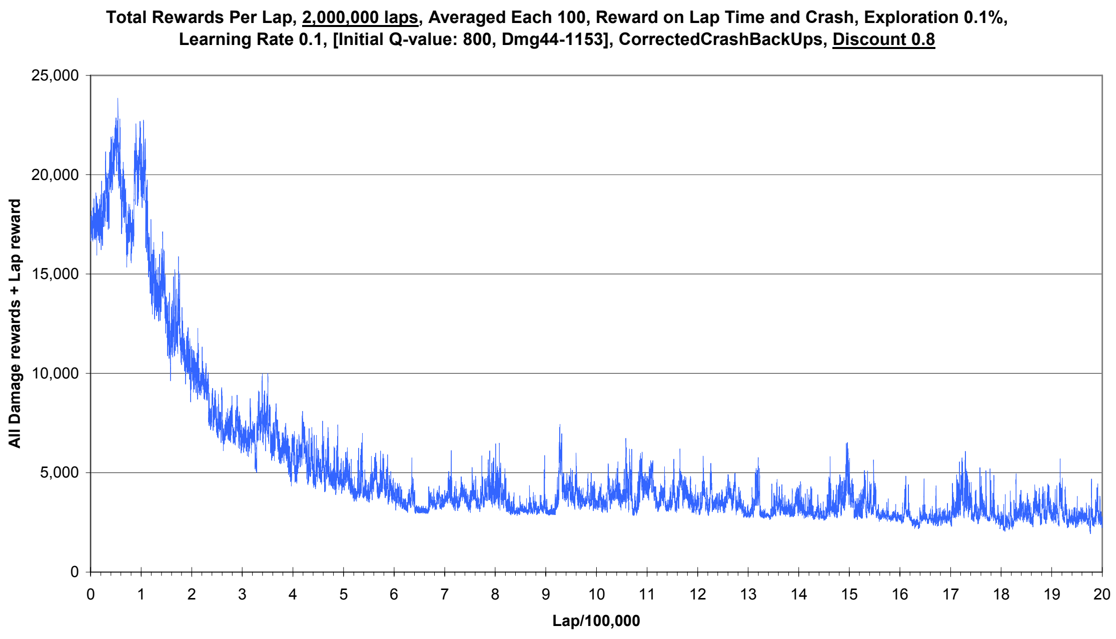


Figure 4-17 Discount of 0.8: Total-rewards-per-lap Effect

to different quantities being measured, Figure 4-17 can be compared to Figure 4-12 and Figure 4-13. This indicates slower learning with a discount of 0.8, compared to with a discount of 0.9. The two larger bumps at the beginning of Figure 4-17 are not explained other than being longer-term versions of the same bumps seen throughout the remainder of the graph.

Long-term Trend, Discount of 0.8

The experiment that generated Figure 4-18 is the same as generated Figure 4-17, but Figure 4-18 shows 20,000,000 laps. Figure 4-18 shows that even with a small discount number (0.8) the reward graph continues to fall in the very long term (20,000,000 laps). Figure 4-18 appears to fall to zero on some laps, but it does not because the minimum total reward possible is the fastest lap time. This is seen in Figure 4-16, but it is not apparent in Figure 4-18 because of the large y-axis scale.

In Figure 4-18, the smallest total reward per lap (which is also the fastest lap time, because damage is zero) is 36.5s on lap approximately 17.6 million. This occurs much later in learning, but is a far better lap time than when a discount of 0.99 is used, as is shown in Figure 4-16, which has a smallest total reward of 41.2 seconds on lap approximately 1.7 million.

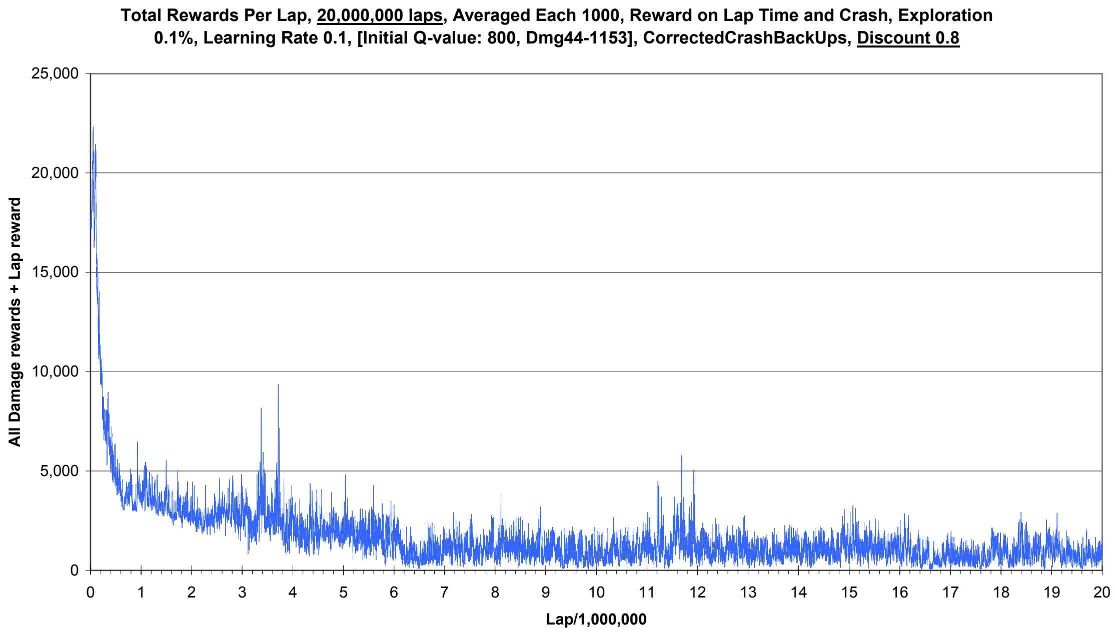


Figure 4-18 Discount of 0.8: Total-rewards-per-lap Effect, 20,000,000 Laps

Figure 4-18 may be compared to the first half of Figure 4-15; and Figure 4-17 may be compared to Figure 4-13, (the experiments of Figure 4-13 and Figure 4-15 use a discount of 0.9). However, note that the x-axis and y-axis scales differ, and the experiments of Figure 4-17 and Figure 4-18 use smaller damage rewards than those of Figure 4-13 and Figure 4-15. Also, Figure 4-17 and Figure 4-18 show total-rewards per lap while Figure 4-13 and Figure 4-15 show damage per lap. Nevertheless, the comparison of these four graphs strongly suggests that a discount of 0.8 results in (much) slower learning than a discount of 0.9.

4.3.5 Discount of 0.995

Experiments using a discount of 0.99 (shown in Figure 4-10/Figure 4-11) show faster learning than those using a discount of 0.9 (shown in Figure 4-12/Figure 4-13). This result suggests trying an experiment with a discount of 0.995, and such an experiment is shown in Figure 4-19. The smallest total reward per lap (which is also the fastest lap time, because damage on that lap is zero) is 38.3s on lap approximately 95,000.

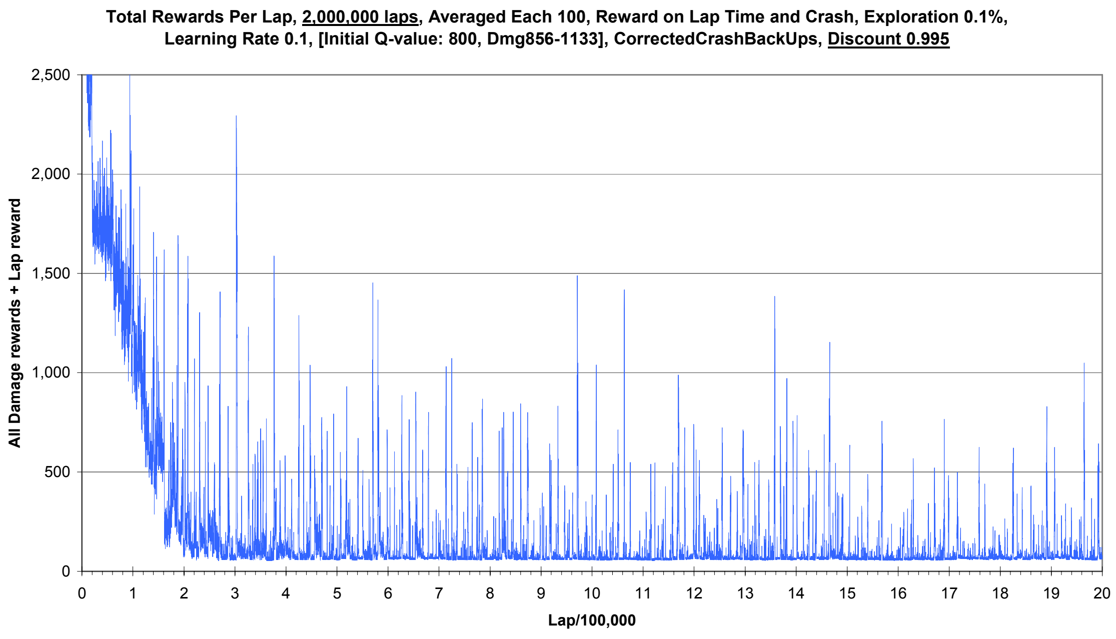


Figure 4-19 Discount of 0.995: Total-rewards-per-lap Effect

Comparing Figure 4-19 (discount 0.995) to Figure 4-16 (discount 0.99) and Figure 4-17 (discount 0.8): the discount 0.995, used in the experiment of Figure 4-19 gives the fastest learning. It does not result in a best lap time as fast as that shown in Figure 4-18, where the discount used is 0.8, although the experiment of Figure 4-18 is run for far more laps, so this comparison is not helpful. Compared to Figure 4-16, Figure 4-19 depicts slower learning up to about lap 20,000, and after that point, learning of about four times faster. Note that the experiment of Figure 4-19 has higher damage rewards than the experiment of Figure 4-16. This may explain the slower learning up to lap 20,000 as this is the same pattern as seen when comparing Figure 4-16 and Figure 6-4, Section 6.1.2, (the experiment of Figure 6-4 uses high damage rewards). Comparing Figure 4-19 to Figure 6-4 (note the different y-axis scales): Figure 4-19 shows faster learning at all stages. The experiment of Figure 4-19 uses higher damage rewards than the experiment of Figure 6-4. It is difficult to compare Figure 4-19 with Figure 4-13, which uses a discount of 0.9, because of several differing parameters and scales.

Summarising Figure 4-10 to Figure 4-19:

Increasing the discount value makes learning much faster: the curve descends faster and the lap with smallest total-reward occurs sooner. However, increasing the discount value can mean the value of the smallest total-reward lap is poorer (i.e. higher). One possible explanation is that because a larger discount value means the algorithm is more "far-sighted" this contributes to the decreased learning time. However, best performance (smallest total-reward per lap) may occur when the algorithm is more "near-sighted" because it may be more worthwhile taking actions likely to avoid the next crash, rather than attempting to also avoid crashes in the more distant future. That is, it may be possible to look too far ahead when making action decisions.

4.4 Chapter Summary

The experiments in Chapter 4, and onwards, do not employ initial supervised learning. When initial supervised learning is omitted reinforcement learning initially occurs more slowly but improves beyond the point where it appears to plateau when initial supervised learning is used. The long-term bias caused by the initial supervised learning may be overcome after many millions of laps, but at this stage of the work it is more profitable to discontinue its use.

As the initial Q-value is decreased, lap time variance increases, and the learning speed up to lap 1,000 increases; yet the overall learning speed up to lap 15,000 decreases. Faster initial learning, but excessive variance later that ultimately slows learning, is consistent with increased exploration. Increased exploration is a likely effect of decreasing the initial Q-value because in the RARS problem domain a better Q-value is a smaller one. Therefore, by using smaller initial Q-values all actions appear as more desirable. With small initial Q-values, when a "poor" action is chosen and its Q-value is increased (made poorer) it is more likely to appear worse than the default Q-value and therefore alternative actions are tried sooner because, while the initial Q-values are lowered, the rewards are left the same. It is concluded that decreasing the initial (default) Q-value appears to increase (non-scheduled) exploration. It is an advantage to control the amount of exploration, for example by using scheduled random actions. For this reason the initial Q-value is set to a large value, such as 400 or 800, to reduce the amount of non-scheduled exploration.

Discounting is used on continuous tasks to ensure the Q-value (the "reward sum") is finite. It is also essential on very long duration tasks, such as the current work, to ensure the Q-value is limited to a practical size. Empirical results show that increasing the discount value makes learning much faster: the curve descends faster and the lap with smallest total-reward occurs sooner. However, increasing the discount value can mean the value of the smallest total-reward lap is poorer, (i.e. higher). There appears to be a learning-speed/learning-quality trade off that is based on the usefulness of "near-sightedness" versus the usefulness of "far-sightedness", and a compromise value for discounting must be found experimentally.

5 Rewards and Efficiency

This chapter is organised as follows. Section 5.1 describes experiments with various damage reward sizes. Section 5.2 introduces the use of speed rewards and tests various schemes. It is also shown that the use of speed rewards addresses the lap-time/damage trade-off problem. Section 5.3 involves the implementation of eligibility traces, and this improves learning efficiency considerably. Section 5.4 shows that hashing is used to free up memory that is then utilised to increase the discretisation resolution. This results in slower learning but improved driving. Section 5.5 describes a simple nearest neighbour generalisation method that improves the transfer of knowledge between different driving tasks, and suggests the use of function approximation. Section 5.6 summarises the chapter.

5.1 Altering The Damage Rewards

In this section, damage rewards are varied to find out what effect this has on the learning performance. The lap time reward scheme is not altered.

5.1.1 Increasing the Damage Reward

Table 5-1 summarises performance of experiments run with various damage reward schemes. The damage reward is the only parameter that varies between the experiments. Table 5-1 shows that when the damage reward is increased the learning speeds up,

Damage reward	Minimum total-of-all-rewards	Occurs on lap number	First lap after which there are 150 or more consecutive damage-free laps
856 to 1133	40.9	19,598	25,800
1712 to 2226	37.89	39,015	3,289
4280 to 5566	37.74	5,296	1,512
8560 to 11133	37.32	25,651	292
17120 to 22266	44.61	14,324	147

Table 5-1 Effect of Increasing the Damage Reward

especially over the first 30,000 laps, (observe column 4). The graphs corresponding to the five experiments of Table 5-1 are not shown due to space requirements, however, these show the learning curve falls more steeply as the damage reward is increased, which agrees with the trend suggested in Table 5-1 column 4. After 1,000,000 laps there is no difference in learning speed. However, the best (i.e. minimum) total-of-all-rewards does not always occur sooner, and is not necessarily better with a higher damage reward, (see columns 2 and 3 of Table 5-1).

5.1.2 Decreasing the Damage Reward

Table 5-2 summarises performance of experiments run with various different damage reward schemes with lower amounts of damage. Row 1 of Table 5-2 shows that when there is no minimum crash reward the learning curve falls much more slowly. After 2,000,000 laps the experiment used in row 1 of Table 5-2 reaches a total-reward of about 1,000. When the same experiment is run but using a minimum reward of 856, then after 2,000,000 laps the total-reward is around 200, (incidentally, the graph of this experiment is Figure 5-12, Section 5.4.2). This pattern confirms the pattern shown in Table 5-1, where larger rewards are used. That is, a lower damage reward decreases the learning speed.

The Damage/Lap-time Trade-off

One reason for changing the damage reward size is to attempt to alter the balance between the agent optimising for smallest lap time and smallest damage. The trade off of higher lap time for less damage to give lower total-reward is first seen in Figure 3-8 / Figure 3-9, and then more clearly in Figure 4-10 / Figure 4-11. This is correct from the reinforcement learning agent's point of view, because the total reward is being lowered. However, the

Damage reward	Minimum total-of-all-rewards	Occurs on lap number	First lap after which there are 150 or more consecutive damage-free laps
0 to 1133	39.89	1,224,939	Far Beyond 2,000,000
44 to 237	41.21	1,713,087	Beyond 2,000,000
428 to 565	39.19	64,880	317,600

Table 5-2 Effect of Decreasing the Damage Reward

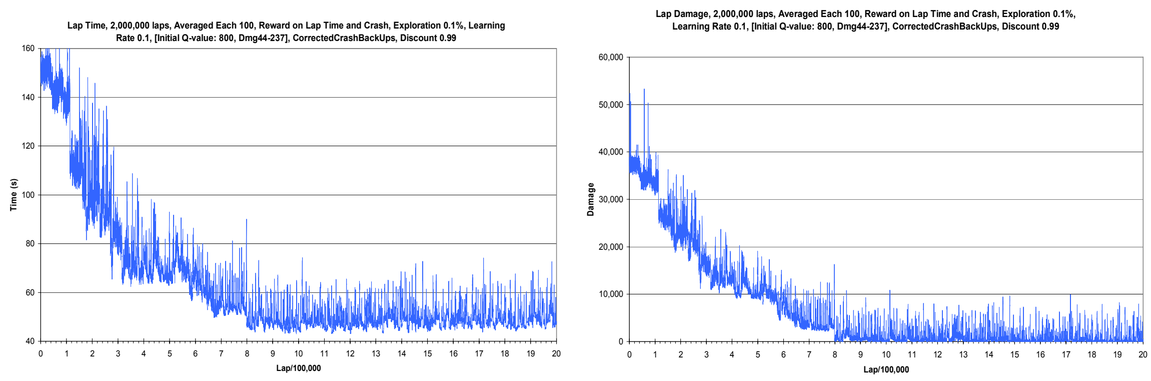


Figure 5-1 Lap Time (Left), Lap Damage (Right), With Damage Reward of 44 to 237

intended goal for the agent is to find the lowest lap time. One way to attempt to address the trade-off just mentioned, and the resulting eventual increase in the lap times, is to lower the damage reward relative to the lap-time reward. This is what is done in the experiments of rows 2 and 3 in Table 5-2. The experiment shown in Figure 4-10 / Figure 4-11 uses damage rewards of about 428 to 565, as does the experiment shown in row 3 of Table 5-2. Row 2 of Table 5-2 shows an experiment using damage rewards of 44 to 237. The total-rewards graph for this experiment is Figure 4-16, Section 4.3.4, which continues falling over its whole length, although only slowly in its second half. The lap time and damage graphs corresponding to the experiment of Figure 4-16 are both shown in Figure 5-1.

Figure 5-1 shows the same pattern as Figure 3-8 / Figure 3-9. That is, the lap time falls until the damage starts to reach zero, after which the damage continues to fall slowly as does the total reward (see Figure 4-16), but from that point on the lap time gradually starts to increase (on average). The trade off of higher lap time for less damage to give lower total-reward is still occurring, despite the reduced ratio of damage rewards to lap time rewards. The fall of the curves happens more slowly and the point at which the lap times start to increase is about three times the number of laps later. The lap time increase happens more slowly, but it still increases. This same pattern is seen using other ranges of small damage rewards. The same pattern is also seen when damage rewards are increased, that is, the curves fall more quickly and the point at which lap times start to increase occurs sooner. In summary, altering the ratio of damage reward size to lap-time reward size does not solve the problem of the long-term trade-off of lower damage for higher lap time.

5.1.3 Distribution of Damage Rewards in the Final Scheme Used

Figure 5-2 shows the distribution of the damage rewards given when using the final choice of damage reward scheme (480 to 1133), measured for the first 18,689 crashes of a run (that, incidentally, uses the final choice of discretisation scheme which is discussed in Section 5.4). Figure 5-2 is made by sorting all the damage rewards received into ascending order and then plotting them. Hence, the graph is bound to be monotonic; a horizontal area indicates a set of rewards of identical value; and the x-axis scale simply indicates the reward numbering after sorting.

Figure 5-2 illustrates the distribution of damage rewards. Some crashes suffer no damage, but are given a reward of 480 by the scheme used in the experiment of Figure 5-2. If this minimum crash reward is not given then the graph drops to zero on the y-axis at 2,138 on the x-axis, that is, the flat part on the left hand end of Figure 5-2 sits at zero, not 480. The resulting change in performance is illustrated by comparing the first row of Table 5-1 with the first row of Table 5-2. This shows there is benefit in giving a minimum damage reward to those crashes given no damage (on the second time step) by the simulator. In the final damage reward scheme, illustrated by Figure 5-2, 480 is used as a minimum rather than 856 (856 is seen in Table 5-1, row 1), as this gives a reward proportional to the crash size for most crashes.

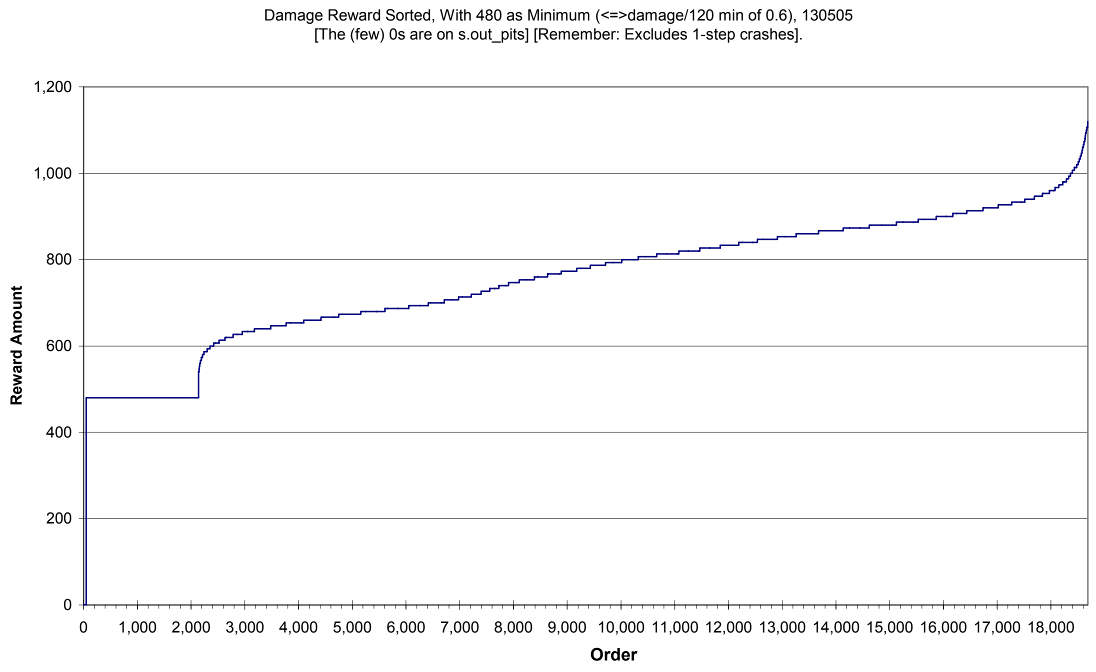


Figure 5-2 Damage Rewards, Ordered by Size

The large increase in learning speed (evidenced by the comparison of Table 5-1, row 1 and Table 5-2, row 1) justifies the use of a minimum damage reward, (shown as the flat part, below 2,138 on the x-axis of Figure 5-2). However, this observation does not justify these rewards being based on the damage in the second time-step (no damage is ever allocated on the first time-step) of the crash, rather than being based on the total damage accrued before return to the track. However, this is justified by other observations: the amount of total accrued damage does not appear as consistent with (proportional to) the severity of the crash as does the damage on the second time step. For example, on some (rare) crashes the car goes into a spin that takes thousands of time steps and accrues enormous damage, (this looks like a bug in RARS). None of these observations justify the use of a reward proportional to the severity of the crash instead of a simple fixed reward.

5.2 Speed Rewards

The idea behind using a speed reward is that in the RARS domain a higher average speed is more desirable, in general. Speed rewards are discussed in Section 3.3, and that discussion is reiterated below. The experiments of this section utilise eligibility traces, hashing and finer discretisation, although these are not described until Sections 5.3 and Section 5.4. This is only mentioned to facilitate the reproducibility of the data—it is irrelevant to the current discussion in that it does not affect the comparison of results.

Speed rewards provide frequent feedback about the robot's performance because a speed measurement is available at every time step, and can be different every time step, unlike damage and lap time rewards which are sparse. Speed turns out to be a very effective feedback. However, maintaining the highest average speed around a circuit will not give the fastest lap time, as every racing driver knows! This research confirms this in that the minimum lap time often does not occur on the same lap as the minimum sum of speed rewards (which gives the lap with the highest average speed). The fastest lap time is not given by taking the shortest path, either. The shortest path on a circuit is always the hard inside line, and involves lower speed due to the corners of tighter radius. This lower speed is not compensated for by the shorter path. It turns out it is possible to maintain a higher average speed, than that maintained during the fastest lap, by taking a longer path. However, the longer path is not compensated for by the higher average speed. This is not the longest path, which is the extreme outside path, because that involves violent steering changes on corner entry and exit, and violent steering changes require a low speed to be executed. The best lap time requires a particular balance between a short path and a high average speed (and a gradual change of direction, although this is a main requirement for high average speed).

There are two speed measurements provided by the simulator. One is the speed of the robot in the direction it is travelling ("v"); the other is the speed of the robot in the direction normal to the track wall ("vn"). Neither always indicates the robot's speed of progress around the track. Normal velocity is sometimes useful, (e.g. when turning into a corner), sometimes not. If the robot is travelling directly towards the track edge it may have a high velocity (v), which will all be in the normal direction (vn), but this is probably not a useful thing to be doing. The most useful measurement of progress is the tangential velocity, that is, the velocity in the direction of the track. This is given by the size of the vector difference of the two previously mentioned velocities: $V_t = \sqrt{V^2 - V_n^2}$. The relationship of these three velocities is shown in Figure 3-1.

5.2.1 Speed Reward Schemes

Minimisation of Q-values is used for this work because this makes more intuitive sense when lap time and crash rewards are used. However, the speed reward needs inversion so that the largest speed appears the most desirable. The speed reward is scaled in relation to the damage and lap time rewards. The scaling factor was determined empirically. The scaling and inversion is achieved by the numerator of 1,000 in the following reward schemes. The two speed descriptions provided by the simulator are "s.v", the velocity of the robot, and "s.vn" the component of s.v that is normal to the track wall. The tangential velocity is derived by: squareroot(s.v×s.v − s.vn×s.vn).

Various speed reward schemes are tested, and are described briefly in the list below:

- 1,000 / velocity & crash damage ranging from 856 to 1133;
- 1,000 / tangential velocity & crash damage ranging from 856 to 1133;
- 1,000 / tangential velocity & crash damage ranging from 480 to 1133;
- 1,000 / tangential velocity & crash damage ranging from 0 to 1133;
- 1,000 / tangential velocity & no crash rewards;
- 1,000 / sqrt(tangential velocity) & crash damage ranging from 480 to 1133;
- 1,000 / (tangential velocity)$^{3/4}$ & crash damage ranging from 480 to 1133;
- 1,000 / (average velocity so far this lap) & no crash rewards.

"1,000 / tangential velocity & *no crash rewards*" worked reasonably well in practice, although it did not perform as well as when crash rewards were also used. "1,000 / (average velocity so far this lap) & no crash rewards" appears to be a promising idea because the use of "average velocity so far this lap" provides information about the current, and all the past, speed rewards on the current lap at every time step. That is, it gives a measure of performance-so-far, rather than speed-at-this-instant. However, in practice it performed poorly.

The best performance is observed experimentally when using the scheme: "1,000 / tangential velocity & crash damage ranging from 480 to 1133". This changes the trade-off between lap-time and damage, resulting in decreasing lap-times, as well as decreasing damage, as shown in Figure 5-4 to Figure 5-7. However, the speed of learning decreases, that is, the total-rewards-per-lap decreases more slowly with speed rewards than without speed rewards, also, the variance in the total rewards curve increases. But, when run for 8 million laps the final performance is much better than after 2 million laps. Therefore, the increased spread at 2 million laps is because the learning has not progressed as far as previously. The sizes of the learning rate (α), the discount (γ), the amount of exploration (ε) and the initial Q-value all need to be re-experimented with in various permutations, since the algorithm has had a major modification by the addition of speed rewards. A combination of parameter values may be found that improve the learning speed, however the experiments were not tried due to the large amount of time needed to run (months).

5.2.2 The Lap-time / Damage Balance is Altered

The use of speed rewards has enabled the lap-time/damage trade off to be successfully addressed for the first time. Figure 5-3 is the lap-time graph from the same experiment as the total-rewards graph shown in Figure 5-12 (Section 5.4.2). This experiment is just prior to the introduction of speed rewards.

The experiment of Figure 5-3 and Figure 5-12 uses eligibility traces and fine discretisation to give much improved convergence (incidentally, Sections 5.3 and 5.4 discuss this), however, these two graphs still show the trade-off of higher (slower) lap times for lower damage, which results overall in lower (better) total rewards. This unsatisfactory situation has clearly occurred ever since discounting was introduced in the experiment of Figure 4-10 and Figure 4-11, and may be seen as far back as the work shown in Figure 3-8 and Figure 3-9. This contrasts with the following four graphs (Figure 5-4 to Figure 5-7) which show both lap-time and total-rewards graphs falling at the same time, now that speed rewards are also used.

The experiment of Figure 5-4 and Figure 5-5 uses the same setup as the experiment of Figure 5-3 and Figure 5-12, except that speed rewards are also used. The reward scheme used is: "1,000/tangental velocity, and with crash rewards ranging from 480 to 1133".

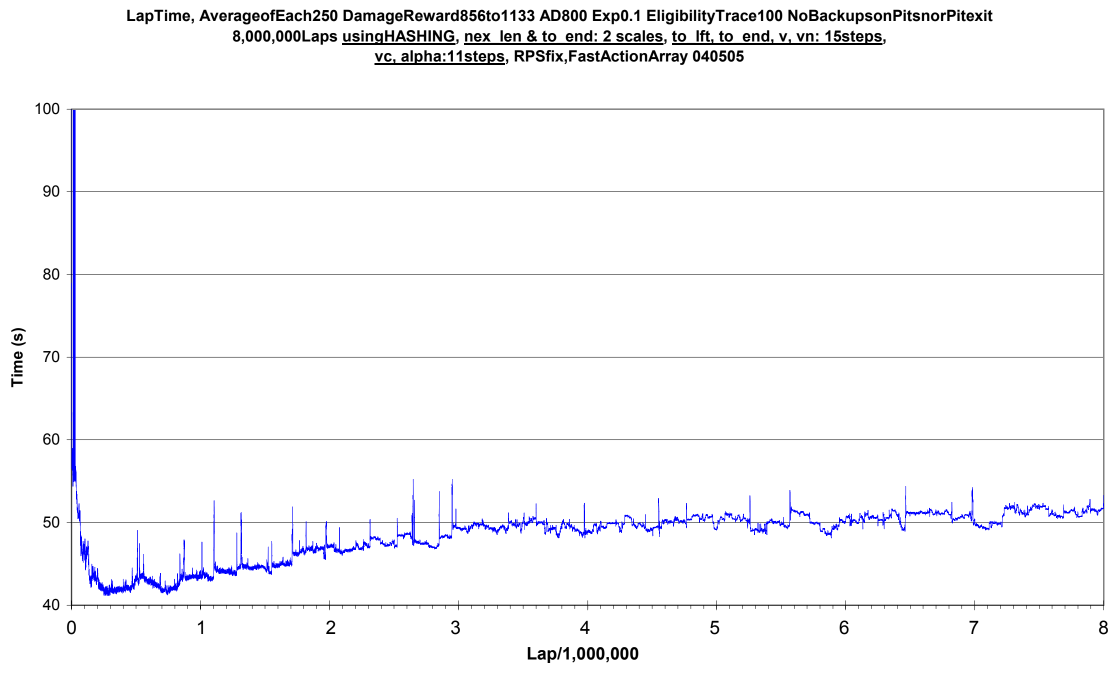


Figure 5-3 Lap times, from the Experiment of Figure 5-12

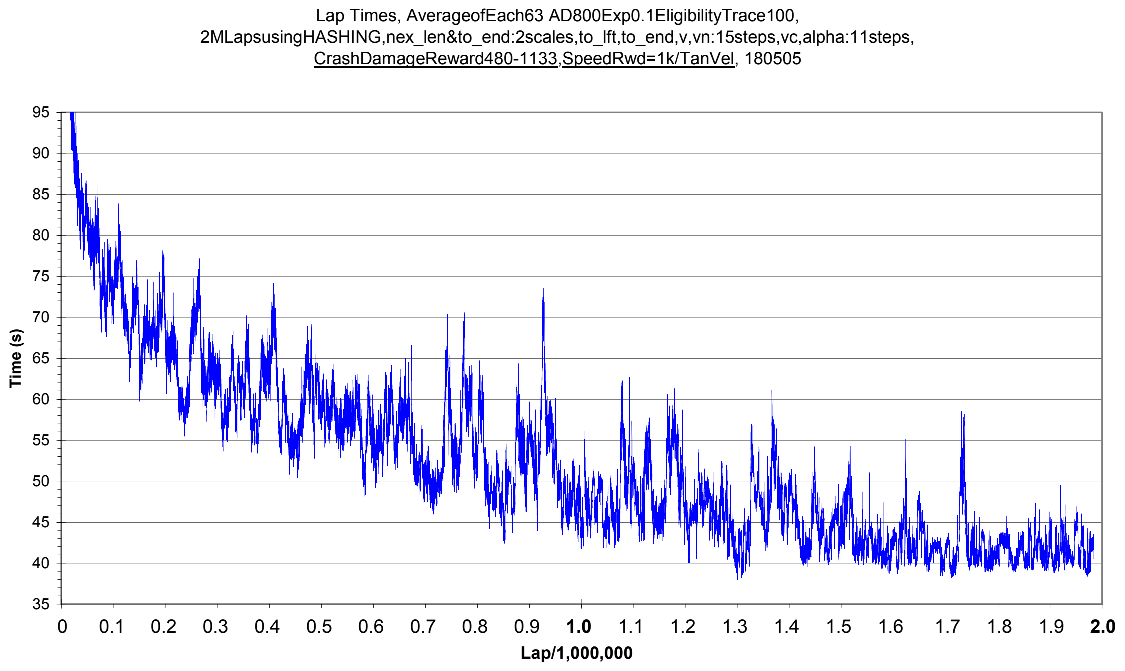


Figure 5-4 Lap times, with Speed Rewards used, 2,000,000 Laps

Figure 5-4 shows lap time generally decreasing over all of the 2,000,000 laps. This is in contrast to the experiment shown in Figure 5-3, which does not use speed rewards, where, over the first 2,000,000 laps, the lap time falls and then rises again. Figure 5-5 shows total rewards per lap generally decreasing over all of the 2,000,000 laps. (Figure 5-5 and Figure

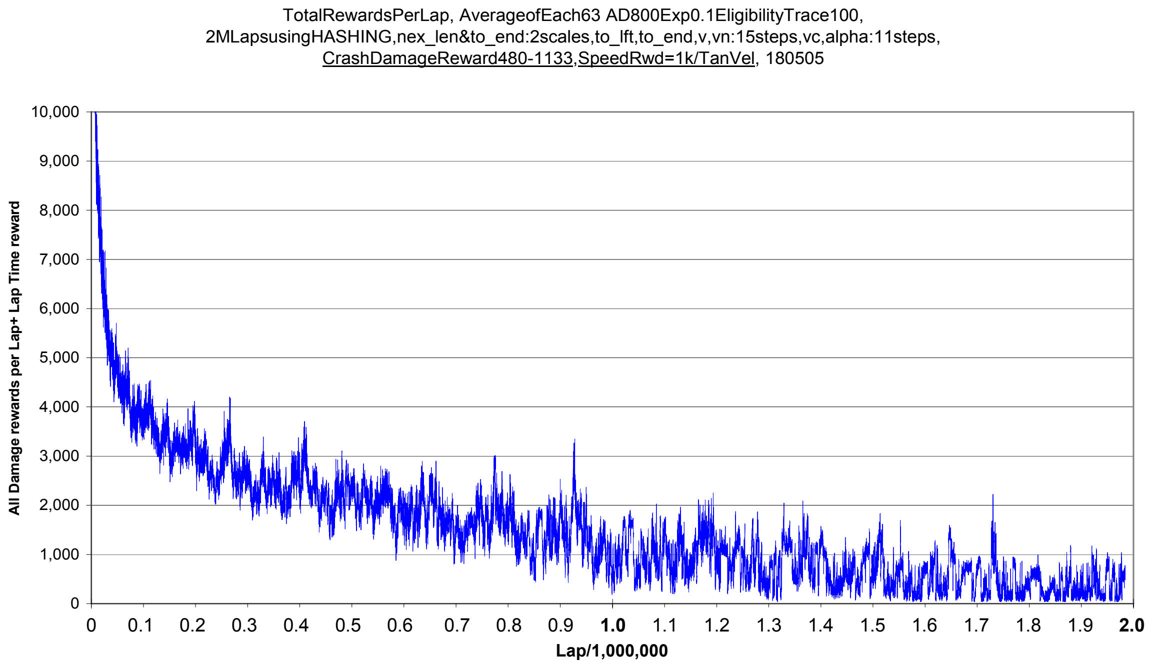


Figure 5-5 Total Rewards, with Speed Rewards used, 2,000,000 Laps

5-4 are from the same experiment). Comparing Figure 5-5 to the first 2,000,000 laps of Figure 5-12, (the experiment of Figure 5-12 is identical to the experiment of Figure 5-5 except for not using speed rewards): the spread (variation) is difficult to compare because the two graphs are averaged over different sample sizes (Figure 5-5 over blocks of 63 laps, and Figure 5-12 over blocks of 250 laps). However, Figure 5-5 shows an average total reward after 2,000,000 laps of around 400 per lap; compared to an average total reward of about 200 per lap after 2,000,000 laps shown in Figure 5-12. This comparison shows the use of speed rewards has slowed down the learning. Yet, performance is seen to improve in the long-term when the experiment of Figure 5-5 is run for 8,000,000 laps. This is shown in Figure 5-6 and Figure 5-7, below.

5.2.3 The Long-term Effect

Figure 5-6 and Figure 5-7 show the experiment of Figure 5-4 and Figure 5-5 re-run for 8,000,000 laps. For the sake of correctness, it is mentioned that this longer experiment includes two other modifications. (1) The number of visits to each state-action-pair is used as an additional weighting when selecting between equal minimum Q-values. (2) If a state has never been visited before then the same actions are taken as were taken on the previous time step. (Before this modification, the default actions of straight ahead and moderate speed were taken in a new state, due to the averaging of equal minimum Q-values). This partly simulates the inertia of the robot, but was later discontinued because it appeared to give no improvement, and also because it uses prior domain knowledge. If supply of prior domain knowledge is avoided then this leaves the agent as free as possible to more quickly come up with a novel solution. For example, if turning too sharply does not work directly (as would be prevented by simulated inertia), then the agent will learn that, but if it has some useful side effect then the agent may discover that also.

Neither of these modifications make any noticeable difference to performance when tested in separate earlier experiments, however they are retained in the following experiment used for Figure 5-6 and Figure 5-7. In all other respects Figure 5-6 and Figure 5-7 use an experiment with the same set up as that used for Figure 5-4 and Figure 5-5, except it is run for 8,000,000 laps.

Comparing Figure 5-6 to Figure 5-3 (the experiment of Figure 5-3 is virtually identical to the experiment of Figure 5-6 except for not using speed rewards) shows the use of speed rewards helps the algorithm focus on reducing lap time as well as reducing damage, as a means of reducing the total reward. As shown by Figure 5-3, without speed rewards lap time increases later in learning. This is due to the robot using more cautious driving as a means of reducing damage. This trade-off between lap time and damage is valid in the reward regime used in the experiment of Figure 5-3 because the total rewards are reduced. However, the aim of this work is to find the fastest lap time. Therefore, the results shown in Figure 5-6 are better suited to this aim.

Figure 5-7 is the total-rewards graph for the same experiment as the lap-time graph Figure 5-6. Figure 5-7 shows that total rewards per lap reduce more consistently than lap time. This is because it is the total rewards per lap that the agent is trying to optimise. A comparison of the 8,000,000 lap experiments of Figure 5-7 and Figure 5-12 shows that learning is clearly faster in Figure 5-7. (As mentioned, the experiment of Figure 5-12/Figure 5-3 is virtually identical to the experiment of Figure 5-7/Figure 5-6 except for not using speed rewards). This shows the use of speed rewards gives a learning speed-up during an 8 million lap experiment; whereas the previous section shows speed rewards give a learning slow-down when the experiment is for only 2 million laps.

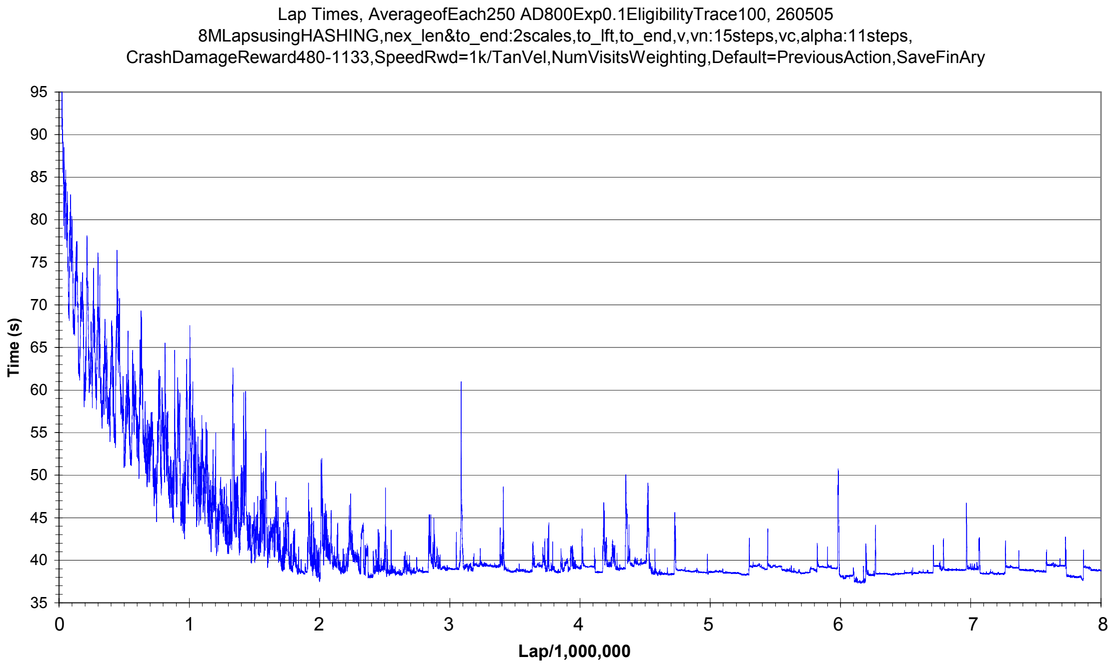


Figure 5-6 Lap times, with Speed Rewards used, 8,000,000 Laps

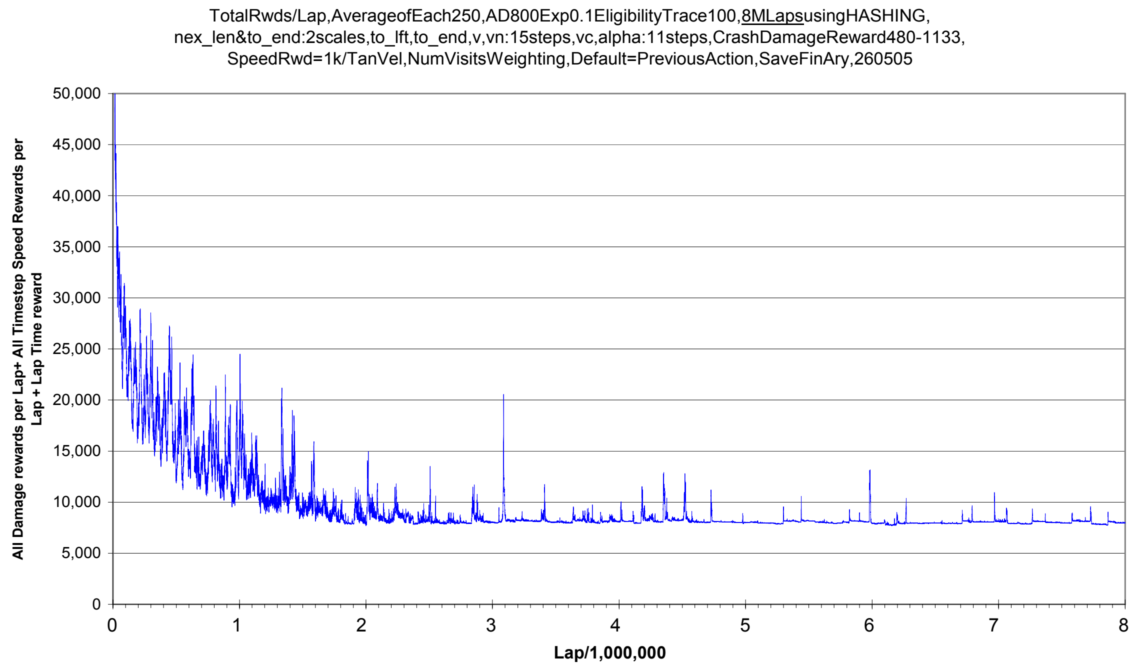


Figure 5-7 Total Rewards, with Speed Rewards used, 8,000,000 Laps

Importantly, note that the total rewards graph Figure 5-7 has a different y-axis scale and a different minimum towards which it is tending than total rewards graphs from all earlier experiments, (e.g. Figure 5-12). This is because the *speed-rewards given each time step are now included in the total-rewards per lap*.

Summary

When a speed reward is given on every time step (while also using damage and lap time rewards), the learning across 2 million laps is slower, but across 8 million laps it is faster, than without speed rewards. Also, lap time now reduces along with total reward. This is a major improvement because reduction of lap time is a goal of this work (and the intended goal of the agent).

5.2.4 Speed Rewards may be Ambiguous

Speed rewards are intended to encourage the robot to maintain the maximum speed at all times, and therefore achieve maximum average speed. (This discourages crashing, and crashing is further discouraged by using damage rewards). The results from Section 5.2 clearly show speed rewards give performance improvements. However, their mechanism may not be working quite as intended.

Speed rewards are the only type of reward given on every time step. The object in this work is to *minimise* the rewards (or rather, the Q-values), so by giving a positive reward on every time step the agent is being encouraged to use the minimum number of time steps. This is clarified by observing that this is the same style of reward typically used to encourage finding the shortest path through a maze (e.g. -1 per step, when the object is maximising the reward). In other words, just the act of giving a reward on every time step may be more important than those rewards being related to speed.

Hypotheticals / Future Work

The size of the contribution the effect mentioned above makes to the benefits shown by the use of speed rewards can be judged by comparison with the learning performance when a *fixed* "speed" reward is used. That is, the maze-style reward effect can be measured by replacing the speed reward with a fixed reward per time step.

The fixed, maze-style, reward per time step may be a better reward regime than using rewards per time step proportional to 1/speed. This is because, as discussed at the start of Section 5.2, the maximum average speed per lap does not necessarily give the minimum lap time; but, the minimum number of time steps per lap *must*, by definition, give the minimum lap time. However, speed rewards may give more localised benefits by providing a more immediate feedback. That is, higher speed is encouraged on the time step just occurring; while the maze-style reward benefit is only seen after a whole lap is completed. Perhaps speed rewards could be used earlier in learning (to give more rapid learning), and replaced by, or faded into, a fixed maze-style reward later in learning (to focus the learning more accurately on lap time reduction).

In a maze-style reward regime the goal is usually given a "good" reward (e.g. 0 or 1 when the object is to maximise the reward). This corresponds to *not* giving a speed/fixed reward on crossing the start/finish line, in the domain of this work, (where the object is to minimise the reward, and only positive rewards and Q-values are allowed). The sum of *all* reward types given on crossing the start/finish line must be less than any one of the speed/fixed rewards on all other time steps, if the crossing of the start/finish line is to appear advantageous in the context of the maze-style reward. A speed reward must not be given on the time step when the start/finish line is crossed. Currently, lap times are given as a reward on start/finish line crossing. These can still be used so long as the worst possible lap time reward (or, perhaps, a fairly poor lap time reward) is better (less) than any speed/fixed reward given. That way, completing a lap still appears as a benefit, and getting a smaller lap time appears as a greater benefit.

The remaining type of reward that can be awarded on crossing the start/finish line is a damage reward. These are larger than most speed rewards (a minimum of 856 in later experiments). This effectively prevents a lap time reward being given for that lap, because the damage reward is added to the lap time reward and this gives a large (poor) reward. This is probably a good set-up, because it is more useful giving the damage reward to encourage learning to avoid the crash, than giving a lap time reward which must be a poor one anyway.

5.3 Eligibility Traces

Eligibility traces are an important and common method used to speed up temporal difference learning by making it more efficient. An eligibility trace uses a temporary record of the state action pairs visited. Back ups are made not just from the current state action pair to the previous state action pair, but from the current state action pair to all the recent previous state action pairs, with a weighting that decreases with distance (λ). This explanation is known as the "backward" view [Sutton and Barto, 1998].

Incidentally, eligibility traces are utilised in the experiments of Section 5.1 (Damage Rewards) and Section 5.2 (Speed Rewards) to speed learning. Their use does not affect the comparison of results within those sections. Also, the experimentation for the work in the current section (5.3) was performed prior to the introduction of speed rewards, (i.e. the only reward types used are damage and lap time). This is irrelevant to the current discussion but is mentioned to assist the reproducibility of the data. Rewards of all types were talked about first because they are fundamental to reinforcement learning; eligibility traces are discussed only now as they are non-essential although they give a large increase in learning efficiency.

5.3.1 Method Used

A naïve implementation updates every state action pair on every time step. This was tested, and because of the size of the state-action space the agent runs very slowly. With the naïve implementation, RARS runs at one frame per second, compared to the equivalent of several thousands of frames per second with the modified eligibility trace method. Although, the naïve method is expected to run faster when also used with hashing because the majority of possible states are never visited and these are not stored when hashing is used. However, in the domain of this work, as in most other reinforcement learning domains, the eligibility values of most (previously visited) state action pairs are virtually zero, except for state action pairs that have been visited recently. The modified method uses a circularly indexed array of pointers to, say, the 100 most recently visited state-action-pairs. This is updated every time step, on a first-in-last-out basis, and also the back-ups are made to all 100 indexed state-action-pairs every time step. That is, the back-up occurs from the current state-action-pair to all 100 immediately previous state-action-pairs, and is discounted by distance by $\gamma\lambda$ (lambda is the eligibility discount); and the current state action pair is added to the head of the list of previous state action pairs; the state action pair at the bottom of the list is pushed off.

Figure 5-8 shows the results from the first use of the modified eligibility trace method, and has a previously unseen "blocky" appearance because it shows raw (not averaged) data of total-rewards. Raw data is used because only 32,000 laps are shown on this graph. The crash component of the rewards has a minimum value of 856 (as discussed earlier, this is to make small crashes significant). Therefore, a lap with one crash has a total reward of 856 + lap-time; a lap with two crashes has a total reward of 1712 + lap-time; this gives the graph the stepped appearance.

The smallest total reward is 37.742 on lap 184, and because this lap has no damage, that corresponds to a fastest lap time of 37.742 seconds. Figure 5-8 shows that most of the learning occurs in the first 300 laps. This is much faster learning than any seen previously.

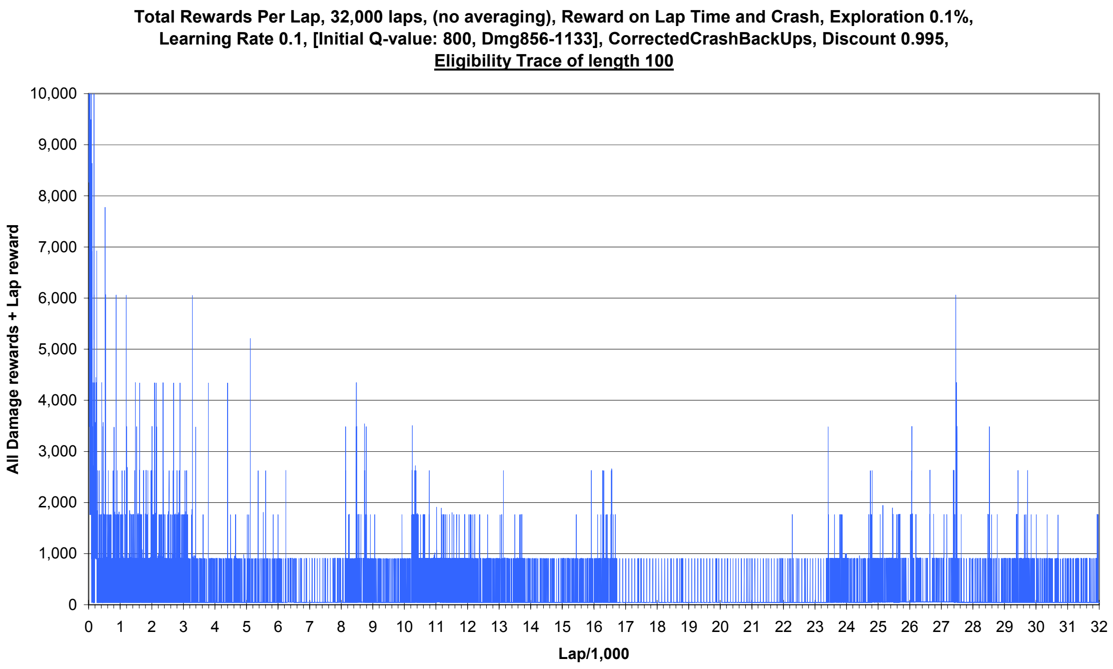


Figure 5-8 The First Use of an Eligibility Trace

5.3.2 A Peculiarity of RARS

Figure 5-8 shows regularity, particularly between laps 17,000 and 23,000. This is much more apparent on a high resolution printout of this graph than if viewed on a computer monitor. A total-reward of about 900 occurs regularly, every 80 to 95 laps. This is the same frequency at which pit stops are taken. It turns out that damage rewards are given on pit entrance, that is, a crash is experienced when the robot crosses the road edge on pit entry. This is simply the way RARS works. This "crash" has to be ignored by the reinforcement learning robot because the design decision was made earlier to, in effect, by-pass pitting, that is, mask pitting from the reinforcement learning agent. If pitting is later introduced as part of the task these rewards must still be ignored because the robot is not in control of its actions during pitting: the simulator is entirely in control. A graph after a work-around is made in the code no longer shows the regular rewards seen in Figure 5-8. This graph is Figure 6-5 in Section 6.1.3.

5.3.3 Medium-term Effects

Figure 5-9 is from the experiment also displayed in Figure 6-5 (Section 6.1.3) but Figure 5-9 shows 100,000 laps. This experiment contains the work-around to ignore the damage rewards given on pit entrance. To fit the data into Figure 5-9 the total-rewards are averaged each 10 laps to give one datum. This has the side effect of making the rewards appear smaller but more frequent.

The Speed-up Gained from using Eligibility Traces

An experiment was run that is identical to that shown in Figure 5-9, except eligibility traces are not used. The total reward graph from this experiment is not shown due to space. However, it is similar in shape to Figure 5-9, except for the x-axis scale. In Figure 5-9 the graph first starts to drop below 250 on the y-axis at about lap 4,000 on the x-axis. The same point, on the total reward graph from the experiment that does not use eligibility traces, occurs at about lap 100,000. When other similar points are chosen on the two graphs the comparison between them is like that above. This indicates eligibility traces have sped up the fall of the curve by around 25 times (100,000/4,000 = 25).

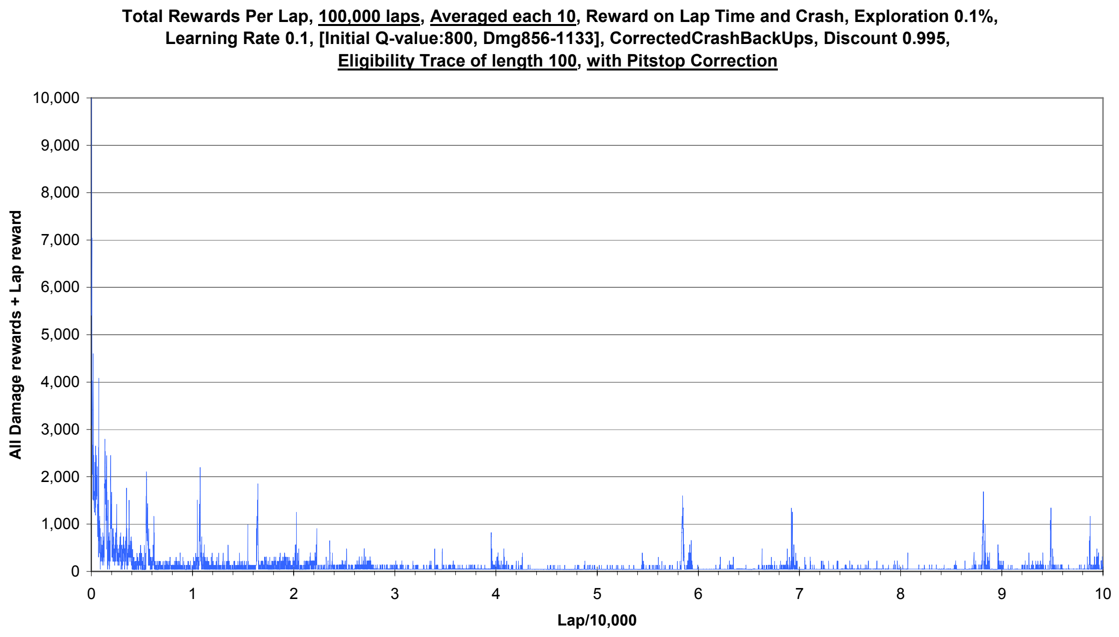


Figure 5-9 Using an Eligibility Trace, with the Pit stop Damage Reward Ignored

Figure 5-9 shows learning continuing beyond 32,000 laps, although not consistently. Although Figure 5-9 is only for 100,000 laps, it shows the learning has already levelled out by about lap 50,000. The fastest lap time is 37.95 seconds on lap 5,127. This lap time is comparable to other results at this point in the research, but occurs sooner in the learning. The inconsistencies in lap time seen in the second half of Figure 5-9 are largely overcome in later work, which allows performance to improve considerably. These problems are overcome by: using finer discretisation; using speed rewards; decreasing exploration later in learning; and fixing various problems during pitting, such as preventing back ups.

5.3.4 Lengths of Eligibility Traces

In this section, various lengths of eligibility traces are experimented with. The depth of the trace also depends on the horizon effect caused by the eligibility trace discount (λ). An eligibility trace length of 1,000 allows a lap-time reward to be credited to (most time steps of) the lap responsible. While a length of 1,000 appears useful for time steps that receive lap time rewards, in practice it turns out to slow down learning. This is possibly because each crash reward is also credited back over the previous 1,000 steps, and crashes are often more frequent than 1,000 time steps. This means the reward ("blame") for a crash can be credited back over steps that lead up to a crash that is previous to the current crash, and those steps are not related to the current crash because the robot is freshly restored to the track after those steps are taken (due to the earlier crash). This is a peculiarity of the RARS domain.

Therefore a suitable length for an eligibility trace appears to be the typical minimum number of steps between crashes. A good heuristic for this turns out to be about 100 steps. However, it may be an advantage to use longer traces on the time steps on which lap time rewards are given. This is because *all* the actions in a lap contribute towards the lap time. In summary, the length of an eligibility trace may be best when it relates to the depth of history that is relevant to the current state/reward. Several lengths were experimented with, and 100 is found to be best overall.

This is another aspect of a problem seen earlier, which is that when two or more different types of rewards are used in the same experiment, they may need treating differently in various ways, although they must all combine, in some way, to contribute to the Q-values if they are to influence the learning. How to do this is not clear, and is currently an open question, (e.g. [Isbell, et al, 2000]). However, this problem is not as difficult as that of an agent with multiple concurrent goals. In the form of the RARS domain used for this work there is only one goal: to find the fastest lap time.

It may be an advantage to relate the size of λ to the length of the eligibility trace. Longer traces could use a larger value for λ as this means the effect of the back ups reduce less rapidly by distance from the current state action pair. For example, lap time rewards could use $\lambda=1$, or very close to 1 (with a trace length of 1,000), because each action in a lap contributes about equally to the lap time. However, this was not experimented with due to time constraints, and remains as future work.

5.4 Discretisation

Discretisation (specification) can be made more fine-grained when hashing is implemented, because hashing allows a more efficient use of memory. Incidentally, all experiments in Section 5.4 use eligibility traces, of length 100.

5.4.1 Hashing

Figure 5-10 shows the performance when hashing is used, without the introduction of any other changes such as finer discretisation. The control, or baseline, for the experiment of Figure 5-10 is the experiment shown in Figure 6-7, page 139. The only difference between the two experiments is that hashing is used in the experiment of Figure 5-10. The two graphs appear virtually the same, and the differences are possibly due to randomness in the simulator. This shows that despite a hashed data structure being used the reinforcement learning algorithm is giving similar results. This is as expected because hashing should not effect the data, only the method by which it is stored.

The memory used with hashing implemented is 24 to 55 megabytes, compared to over 300 megabytes used without hashing. This saving allows the discretisation to be increased until the memory in use is back to about 300 megabytes, which is just before the point when excessive paging (thrashing) occurs on the particular machine used for experiments. This saving shows that over 80% of the possible state-action-pairs are never used in practice. The run time with hashing is about half the speed of without hashing, when run at “full” speed (which is as fast as the processor will go). Run time refers to execution speed (in seconds), not learning speed (in laps) which is unaffected by using hashing.

The experiment of Figure 5-10 uses hashing of both states and actions. In the reinforcement learning implementation used in this thesis, every time a state is visited all of its actions are searched to find the best. This means every time step there are 11 velocity discretisation steps × 11 steering discretisation steps = 121 hash calculations made. The algorithm was modified so the actions are not hashed, only the states. This means some space is wasted. This is because not always are all actions tried in every state that is visited, yet space will now be allocated for all actions, (but only in those states visited). This also means there is now only one hashing calculation per time step. This is a time-space trade off that increases execution speed back to almost as fast as without hashing, and only slightly increases the amount of memory used compared to hashing both states and actions.

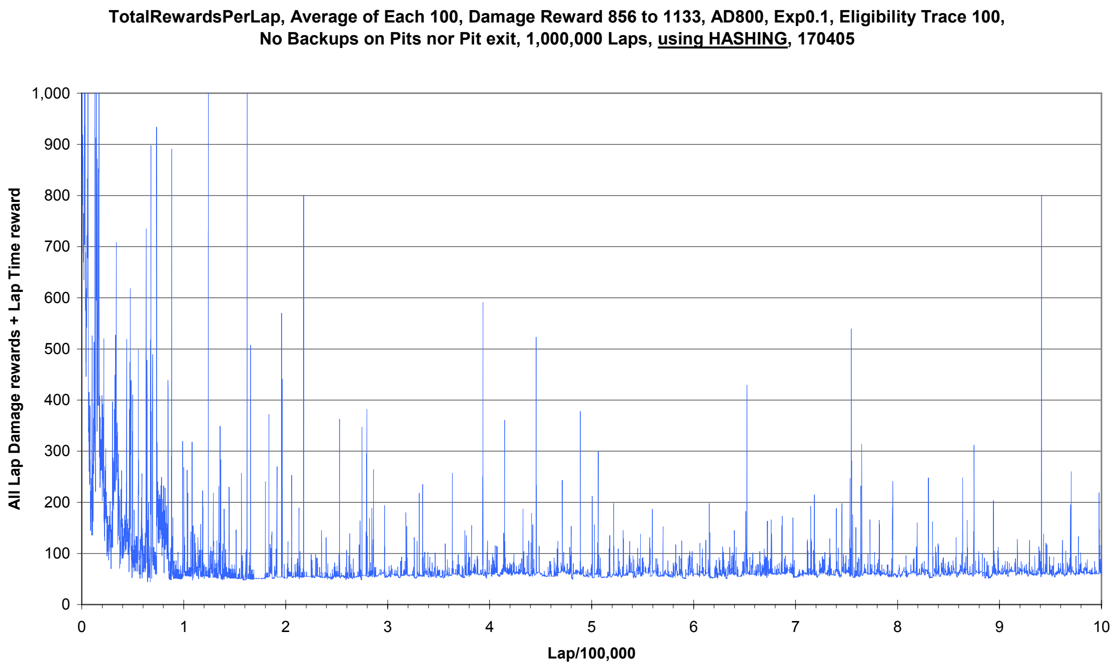


Figure 5-10 Performance when Hashing is used

A simple linear search of the 121 actions is not the most efficient method. However, something more sophisticated may not save much space or time, and is not worth spending time on because the entire data structure is intended to be eventually replaced by some sort of generalising structure, such as a decision tree. The purpose of using hashing is purely to gain enough extra memory space to allow finer discretisation to be tested.

5.4.2 Discretisation Schemes

Before Increasing Discretisation Resolution

Figure 5-11 is from an experiment the same as that shown in the previous graph, Figure 5-10, except the actions are not hashed (which gives a speed-up, as discussed). This does not affect the general shape of the graph, but Figure 5-11 shows eight million laps while Figure 5-10 shows one million laps. The purpose of Figure 5-11 is as the control (baseline) for the following graph, Figure 5-12. Figure 5-11 shows that learning makes little progress beyond one million laps. The minimum total-reward is 38.98 on lap number 105,533.

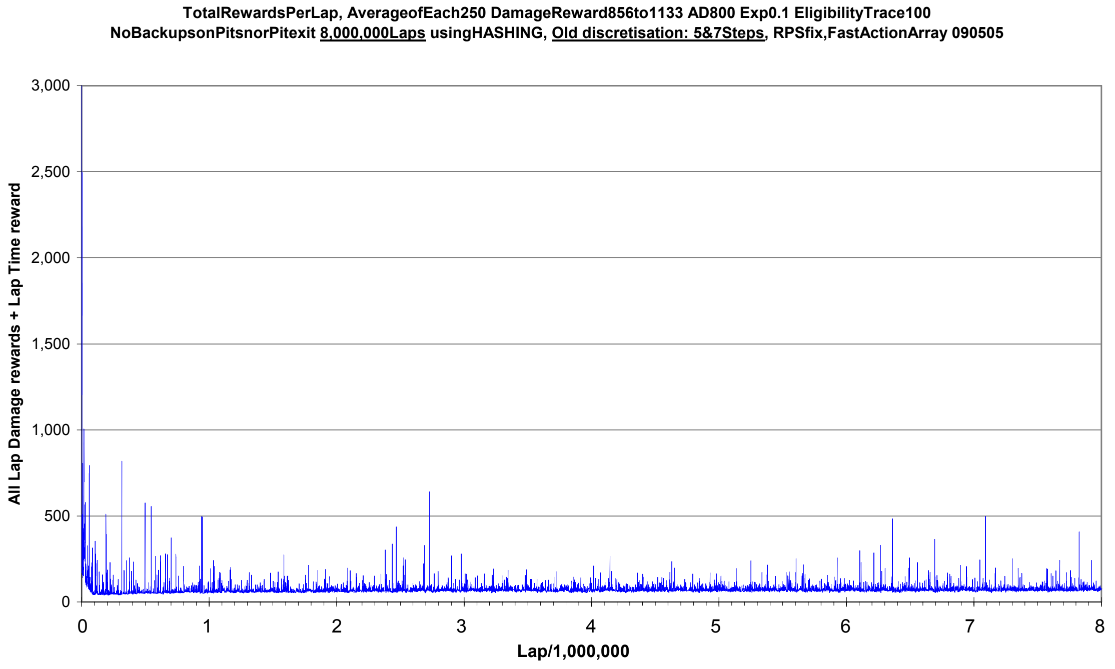


Figure 5-11 Total Rewards, Before Increasing Discretisation Resolution

The experiment of Figure 5-11 uses the same discretisation intervals as used in all previous experiments up to this point. That is, 5 steps for each of the state parameters to_lft, to_end, v and vn; 5 steps for nex_len, (although it only uses 3 steps on track v01.trk); 5 steps for each of nex_rad and cur_rad (although they only use 2 steps each on track v01.trk); and 7 steps for each of the action commands vc and alpha.

After Increasing Discretisation Resolution

Figure 5-12 shows learning performance with the more fine-grained discretisation scheme. That scheme uses 15 steps for each of the state parameters to_lft, to_end, v and vn; 3 steps for nex_len (which is all it needs on track v01.trk); 2 steps for each of nex_rad and cur_rad (which is all they need on track v01.trk); and 11 steps for each of the action commands vc and alpha; (the state parameter cur_len is no longer used). Nex_len and to_end each have 2 scale ranges, which are chosen depending on the context (corner or straight). The context is signalled by cur_rad which is zero if on a straight and not zero if on a corner.

The scheme was checked in practice to see that the whole range of values are in use. This is done by running the robot on track v01.trk, recording all inputs and outputs and then graphing the distribution of each input and output. This shows how many different values

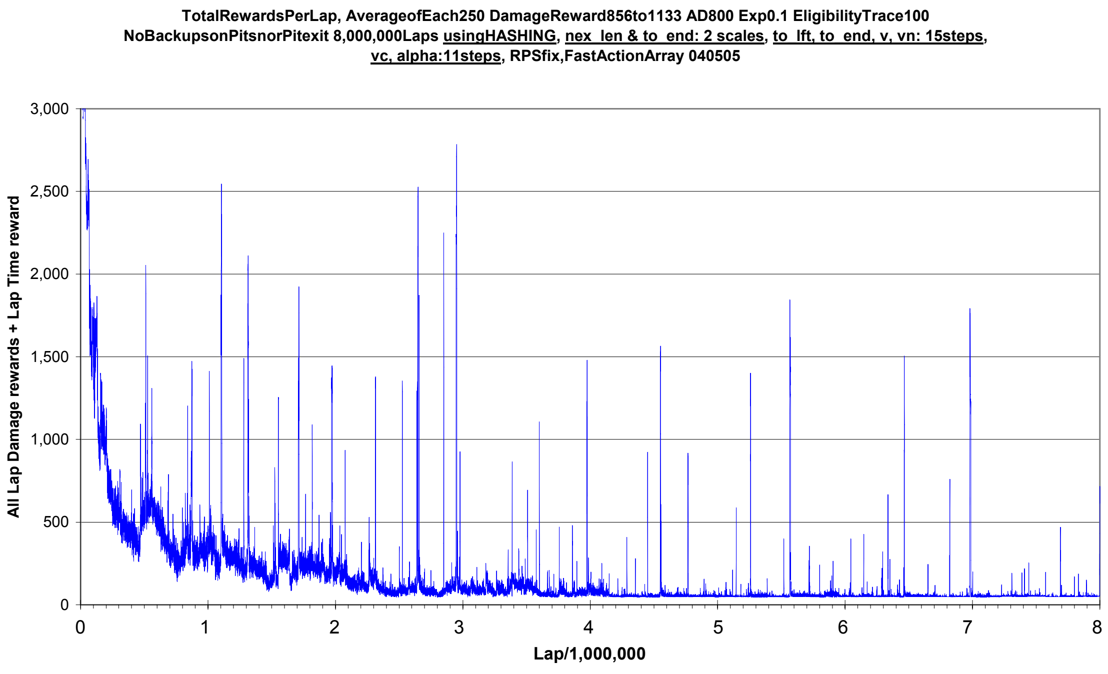


Figure 5-12 Total Rewards, After Increasing Discretisation Resolution

of each parameter are used in practice. The discretisation intervals are set to maximise this range without exceeding the upper or lower ends. It would be more elegant to have some automatic way of doing this. It is wise for the steering output, alpha, to have an odd number of intervals, such as the 11 used, so there is a central position that explicitly represents “straight-ahead”. Over a dozen experiments were run to try out various discretisation schemes. The final scheme chosen uses the greatest number of intervals possible without causing memory problems, such as thrashing.

Learning is slower in the experiment of Figure 5-12 than in the experiment of Figure 5-11 by a factor of about 40 (Figure 5-12 starts to flatten out at about lap 4,000,000 compared to about lap 100,000 in Figure 5-11). This is because there are about 10 times more state action pairs in use, that all need learning, when using the finer discretisation. However, learning ultimately becomes more stable when using the finer discretisation resolution (Figure 5-12). This is seen by comparing laps seven to eight million of Figure 5-11 and Figure 5-12: the average variation is visibly less over those laps in Figure 5-12. Also, Figure 5-12 continues to fall up to lap eight million, whereas Figure 5-11 plateaus from about lap one million onwards. When the driving behaviour is observed in real-time the robot in the experiment of Figure 5-12 drives more smoothly and takes “better” looking lines (judged subjectively using domain knowledge) than the robot with the coarser discretisation (Figure 5-11). This is probably due to the more close-grained control and more close-grained perception (i.e. state representation) afforded by the finer discretisation used in the experiment of Figure 5-12. This may allow greater performance improvements to be made in the long-term than if the coarser discretisation is used.

The minimum total-reward in the experiment of Figure 5-12 is 37.45 on lap number 486,573. This is surprising when the graph is observed as it is relatively early in learning, and must indicate a wide variation of total rewards between laps, keeping in mind that Figure 5-12 is of rewards averaged over each 250 laps (averaging tends to make variation appear less, particularly if averaged over a larger number of laps such as 250). Compared to Figure 5-11 the best lap occurs later and is a better score (Figure 5-11’s minimum total-reward is 38.98 on lap number 105,533). Figure 5-12 shows the agent spends most laps of this experiment reducing the variation in performance between laps, that is, the consistency of the driving, rather than reducing the time for the best lap. The large “spikes” in Figure 5-12 may indicate too much exploration. Or perhaps they are due to some side effect of the finer discretisation.

5.5 Generalisation (Nearest Neighbour)

Background

As previously stated in Section 3.1.1, classic reinforcement learning uses a tabular method to represent the value function. This is the simplest method to analyse, and proofs of convergence for various reinforcement learning methods all assume a tabular method. It is also easy to implement as a simple array.

The aim of this paper is to implement reinforcement learning in a domain with a high number of state parameters, two continuous actions and randomness in the simulator. This is a reasonably difficult domain, therefore a simple tabular method is used at the outset (see Section 3.1.1) as a straightforward way to see if it is possible to use reinforcement learning within RARS.

However, it is widely held that function approximators (e.g. neural networks, decision trees, etc.) should be of great benefit when combined with reinforcement learning. This is because the generalisation they provide is expected to help the agent learn more quickly and with fewer errors when encountering novel situations. This is because some of the knowledge gained in situations previously visited that are *similar* to a novel situation can be applied to the novel situation. Another reason for using function approximation is the compression it provides in a high-dimensional space, especially if the space is sparsely populated. However, this combination has turned out to have both successes and difficulties, such as over-estimation that can lead to divergence [Thrun & Schwartz 1993, Wiering 2004]. The use of function approximators in reinforcement learning is currently an active area of research.

Function approximation is a long-term aim of this work. But an incremental approach is taken and it is first shown that reinforcement learning can work successfully in the RARS domain using a simple tabular method. This occupies much of this work, as a number of refinements are made. It is then shown that compression is worthwhile, because it frees up memory allowing finer discretisation which results in higher performance (Section 5.4). In this section (Section 5.5) the aim is to show that generalisation improves performance. If this is successful, the ground will be ready for the application of function approximation.

A nearest neighbour method is used to perform generalisation. This provides no compression. The aim is to demonstrate whether or not generalisation is useful in this domain, without the added complexity and estimation error introduced by generalising methods such as neural nets or trees.

The Generalisation Scheme, algorithm description

This section describes the nearest neighbour generalisation algorithm used. This provides generalisation on-the-fly, and only when a state action pair is chosen that has had no previous visits.

If the chosen (optimal) state action pair has had zero previous visits (this means the Q-value must be a default value) then either the state has never been visited before, or all previously visited state action pairs in that state have Q-values poorer (higher) than the default value. Because the state action pair has zero previous visits, the action is based on no previous experience, (other than the fact that other actions tried in the state, if any, turned out to be poor ones). Then in this case, and provided the current move is not exploratory, search the adjacent neighbours of the current state by varying the parameters to_lft, to_end, v, vn and nex_len by $\pm$ 1 discretisation steps from their current values. This gives 242 maximum possible neighbours ($3^5 - 1$ (the current state) = 242), although there are often less. Cur_rad, nex_len and nex_rad are not varied because altering these causes a large change of state, as these parameters indicate position in a wider sense (i.e. they are very coarse state descriptors). Also, the number of neighbours visited needs to be limited to give a practical computational run time.

For each neighbouring state, calculate the optimal actions in the usual manner (choose the action with the lowest Q-value in each state); record these actions (vc and alpha), the number of visits to that state action pair and the Q-value. After searching all neighbours, find the mean of their vcs (velocity action commands) and the mean of their alphas (steering angle action commands), weighted by some combination of their number of visits, Q-value and distance of the state from the current state. Early in learning, the number of visits will all, or mostly, be zero. However, these state action pairs are still useful as they may be avoiding other actions in the state, with a number of visits greater than zero, but with Q-values poorer than the default (i.e. the other actions previously tried turned out to be poor, therefore an untried action is being tested). Finally, check these mean values are not suboptimal in the current (actual) state. If they are, they must have been tried previously, (i.e. the number of visits must be greater than zero), and the Q-value must be greater than the default. If this is the case then choose the closest optimal actions; or perhaps use the actions originally chosen (i.e. optimal) in the current state (despite the state action pair having zero previous visits); or perhaps take the mean actions that were just calculated (despite them being suboptimal). These three possibilities are experimented with.

Experiments

Further matters that can be experimented with are:

• The conditions under which to use generalisation: In the work of this section, generalisation is only used on state action pairs with zero previous visits. However, it could be used on state action pairs with less than, say, 5 previous visits, and then the current Q-value could also be given a weighting.

• The neighbourhood area: The work of this section varies to_lft, to_end, v, vn and nex_len by ±1 discretisation steps from their current values. However, fewer or other state parameters could also be varied, and they could be varied by more than ±1 step.

• The weighting formula used to combine the neighbours' actions: Four different weighting schemes are tested, combined with strategies to use when the mean action values happen to be suboptimal. It turns out to be important for the number_of_visits to have a small influence, because the number_of_visits indicates confidence in the Q-value rather than desirability or importance of the state action pair. For example, a state action pair could have been visited on an exploratory move and have a poor Q-value but have 2 visits. The distance of the neighbouring state from the current state is the total number of discretisation steps by which the state parameters to_lft, to_end, v, vn and nex_len vary between the two states. The weighting schemes tested are:

° (number_of_visits × 2)/(Q-value × distance × 800)

And, if the actions derived by generalisation are suboptimal in the current state, then take those actions anyway.

◦ $1 / (\text{Q-value} \times 5^{\text{distance}})$

And, if the actions derived by generalisation are suboptimal in the current state, then discard, and use the optimal actions in the current state that are closest to the actions derived by generalisation.

◦ $(\sqrt[100]{\text{number\_of\_visits}}) / (\text{Q-value} \times \text{distance}^4)$

And, if the actions derived by generalisation are suboptimal in the current state, then discard, and use the optimal actions in the current state that are closest to the actions derived by generalisation.

◦ $(\sqrt[100]{\text{number\_of\_visits}}) / (\text{Q-value} \times \text{distance}^4)$

And, if the actions derived by generalisation are suboptimal in the current state, then discard, and use the optimal action in the current state (i.e. as originally chosen, before generalisation).

Many more schemes could have been tried. But, as often occurs in this thesis, an intuitive and sometimes arbitrary choice has to be made of combinations of things to test out, due to time constraints due to the long run time of the experiments. The last scheme performs best, and is used in the experiment shown in Figure 5-13.

5.5.1 Generalisation, Long-term Effect (8,000,000 laps)

Figure 5-13 shows the lap times and Figure 5-14 shows the total-rewards-per-lap from an eight million lap experiment that uses the last generalisation scheme listed above. The experiment of Figure 5-6 and Figure 5-7 is the control (baseline) for the experiment shown by Figure 5-13 and Figure 5-14. That is, the experiment of Figure 5-6 and Figure 5-7 is identical to that of Figure 5-13 and Figure 5-14 except for not using generalisation. The lighter colour trace in Figure 5-14 is the trace from Figure 5-7.

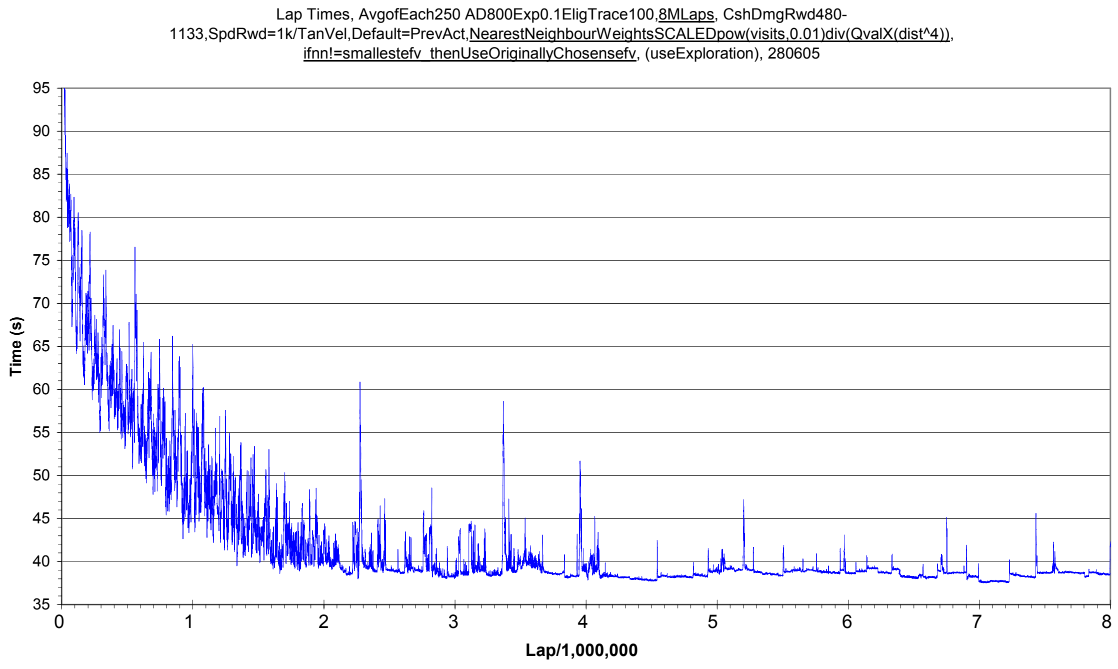


Figure 5-13 Lap Times when using Nearest Neighbour Generalisation

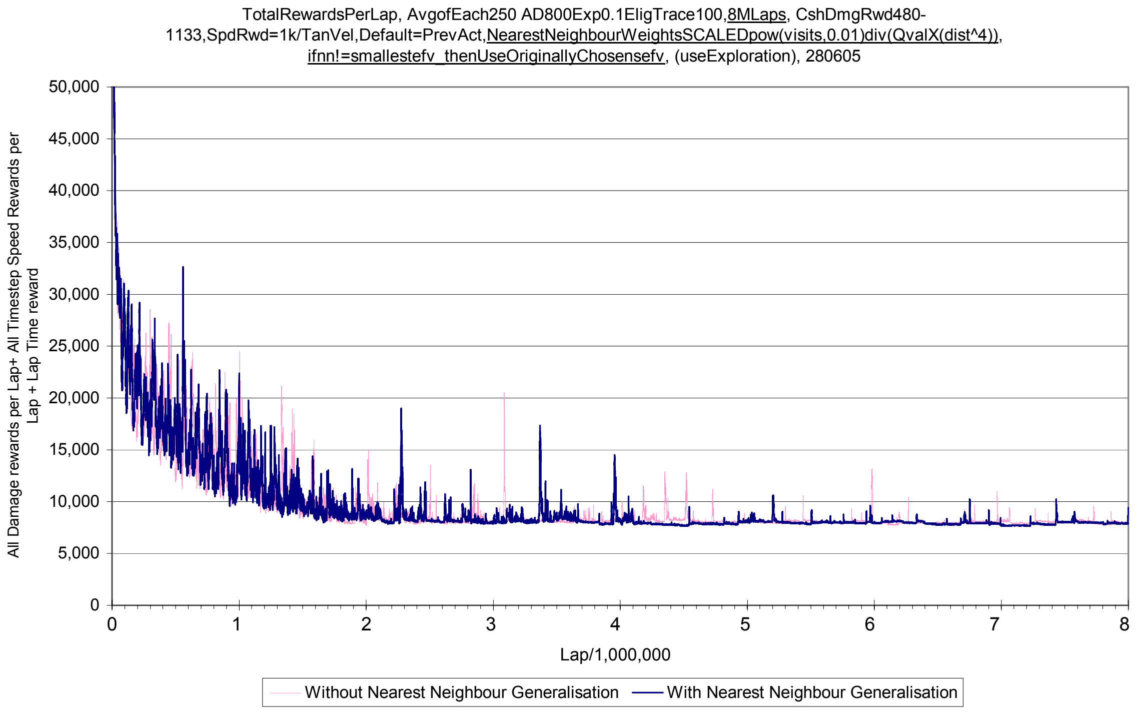


Figure 5-14 Total Rewards per Lap when using Nearest Neighbour Generalisation

The experiments with and without generalisation give very similar results as seen by their graphs. The use of generalisation may result in slightly flatter lap-time and total-reward graphs over the last 4,000,000 laps. The difference is small, and not clearly significant. It is unknown if this much difference could be due to random effects within RARS. Differences in performance may be masked by excessive exploration, especially later in learning. This is as yet untested, but subsequent experiments show exploration needs further reduction in the long term (Section 3.9.6). Generalisation may be most useful early in learning when previously unvisited state action pairs are likely to be encountered more often. The early performance differences are difficult to judge on Figure 5-14 due to its long time scale. Therefore, the first 100,000 laps from Figure 5-14 are shown in Figure 5-15.

5.5.2 Generalisation, Early Learning Effects (100,000 laps)

Figure 5-15 compares learning with and without generalisation, over the first 100,000 laps of the experiments of Figure 5-14. The first 100,000 laps from Figure 5-14 provide the "with generalisation" data; the first 100,000 laps from Figure 5-7 provide the "without generalisation" data. These two curves do not look significantly different. The generalisation method used does not appear to improve learning performance over the first 100,000 laps of the experiment of Figure 5-15.

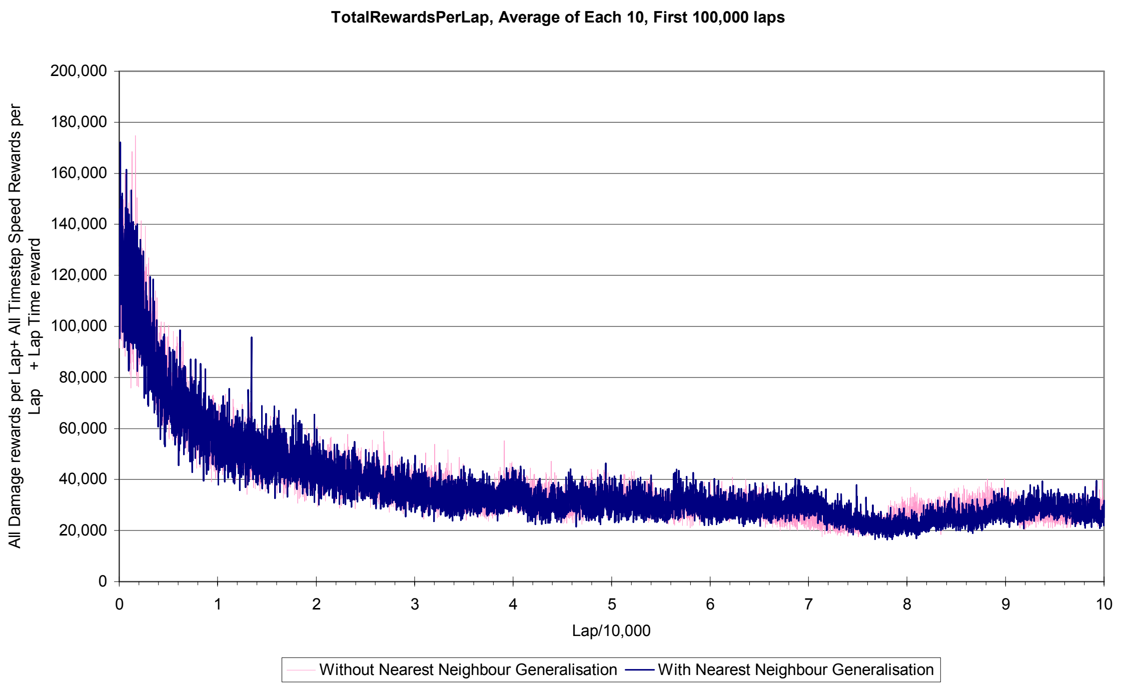


Figure 5-15 Total Rewards per Lap, Nearest Neighbour Generalisation, First 100,000 Laps

5.5.3 Generalisation to Other Tracks

Generalisation allows informed action choices in situations never previously visited. The actions taken in situations previously visited that are similar to a new situation are used as a guide for the choice of action in the new situation. As shown in Figure 5-14 this can (slightly!) improve learning performance. Generalisation can also improve the transfer of knowledge to a new task. The new task needs to be related to the skill domain of the previous task. This means that a robot with driving skills learned on one track may be able to drive successfully on a different track. To test this, a robot is trained using supervised learning by initially reading in the hashed state-action array that was saved at the end of the experiment shown in Figure 5-14. Figure 5-14 shows that reasonable performance is reached in that experiment. The robot is run on track v01.trk, and this confirms it is driving satisfactorily. The agent then has the nearest neighbour generalisation part disabled, and is run on track v03.trk.

Track v03.trk has similarities to track v01.trk. It has seven corners rather than three and these are of both lesser and greater length and radius. Tracks v01.trk and v03.trk are shown in Figure 5-16. The agent does not drive excellently at any point on track v03.trk, however, it drives far better than if started on track v03.trk with no initial supervised learning. It manages one or two complete corners per lap without crashing. This shows there is considerable transfer of knowledge / over-lap of tasks / generalisation between driving on the two tracks *without* the nearest neighbour generalisation code. Some generalisation must be due to the discretisation, that is, some state parameter values that are close but different, become the same after being discretised.

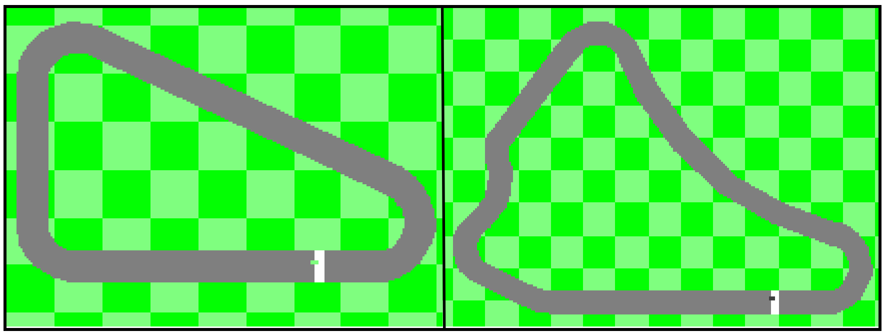


Figure 5-16 Tracks v01.trk (left) and v03.trk (right)

Mean Total Rewards per Lap	Without Nearest Neighbour Generalisation	60,255.01
	With Nearest Neighbour Generalisation	48,615.37
Mean Time per Lap	Without Nearest Neighbour Generalisation	127.15 secs
	With Nearest Neighbour Generalisation	115.55 secs
Mean Damage per Lap	Without Nearest Neighbour Generalisation	18,167.39
	With Nearest Neighbour Generalisation	13,628.52
Minimum Lap Time	Without Nearest Neighbour Generalisation	81.09 secs
	With Nearest Neighbour Generalisation	80.71 secs

Table 5-3 Performance on The First 100 Laps of Track vo3.trk, Using The Model Learnt on Track v01.trk

The experiment on track v03.trk is then restarted, but with nearest neighbour generalisation functioning. The robot's behaviour looks very similar to that shown during the previous experiment, but it appears to be driving more consistently. The performance comparison between these two experiments is made objectively in Table 5-3. This shows the nearest neighbour generalisation method is having a modest but significant beneficial effect on skill transfer. Work in 2003 [Cleland 2003] shows successful skill transfer in the RARS domain using an m5′ decision tree representation.

5.6 Chapter Summary

This chapter first examined the effects of different sizes of damage rewards. It is found that larger damage rewards speed-up the learning over the first 30,000 laps but not when the first 1,000,000 laps are considered. However, the best total-of-all-rewards does not always occur sooner, and is not necessarily better with a higher damage reward. Furthermore, altering the ratio of damage reward size to lap-time reward size does not remedy the problem of the long-term trade-off of lower damage for higher lap time.

Speed rewards provide frequent feedback about the robot's performance because a speed measurement is available every time step, unlike damage and lap time rewards which are sparse. However, maintaining the highest average speed around a circuit will not give the fastest lap time. When a speed reward is given on every time step (while also using damage and lap time rewards), the learning across 2 million laps is slower, but across 8 million laps it is faster, than without speed rewards. Also, lap time reduces along with total reward (i.e. lower damage is no longer traded for higher lap time). This is a major improvement, because reduction of lap time is the intended goal of the agent.

Eligibility traces are implemented with a fixed depth of 100 time steps. This is very successful and gives a learning speed-up of about 25 times.

The implementation of hashing saves about 80% of the memory. This freeing-up of memory allows the discretisation resolution to be increased. The increased resolution slows down the learning by a large factor, due to the increased search space. However, the robot's driving performance becomes more consistent, which is seen as less variation between lap times; as smoother driving and as "better" looking driving lines (judged subjectively using domain knowledge). Also, the best lap time is improved.

Generalisation is implemented using a simple nearest neighbour method. This allows for proof-of-concept without suffering from the estimation errors introduced by more sophisticated methods such as function approximators. However, it does not give any compression, as do function approximators. This generalisation method only slightly improves learning performance; but more clearly improves skill transfer—that is, when an agent learns to drive on one track it is better able to drive on a track with a different layout if the generalisation method is used. These results suggest that the use of function approximation is probably worth investigation in the RARS domain.

6 Domain Effects; Model Analysis; Screen-shots

This chapter first deals with peculiarities of the RARS domain that give rise to implementational issues affecting the reproducibility of most of the experiments in this work. These matters include the use of a minimum crash reward; a modification to the simulator code that gives an execution speed-up; and a sizeable range of issues concerning pit stops. The range of methods used to measure and judge performance are then demonstrated and discussed. Finally, some screen shots of the RARS circuit are given that compare the driving of the reinforcement learning robot with that of an expert robot and a basic robot.

6.1 Domain (RARS) Effects

There are a number of peculiarities associated with the RARS domain that have a major impact on the experiments described in this thesis. These implementational issues are important because they affect the reproducibility of most of the results presented here. This section deals with those issues.

Some of the RARS-specific details have been discussed in earlier chapters when needed: how to deal with crash recovery, and how to connect the time steps before and after a crash (to run the environment continuously cf. episodically); and how to derive crash rewards—these are all detailed in Sections 2.5 and 3.3. However, there are further RARS-specific details which include some rare types of crashes. For example, some crashes last for only one time step, and no damage is suffered, yet these are flagged as crashes. These do not disturb the robot's driving, and so the reinforcement learning robot is coded to ignore these events. This is acceptable because in these situations the robot is driving at its limit (metaphorically, it just touched a barrier but only scuffed the paint!). Other rare crashes occur which have no second-timestep damage, but incur damage on following time steps; and other rare crashes have no second-timestep damage, and no later damage. In these last two cases no damage reward is allocated by a reward scheme used earlier in this thesis. However the robot still in effect receives a penalty in the form of an increased lap time due to the crash. Nevertheless, it was found beneficial to allocate a minimum crash reward in these cases, and this is shown in Section 5.1.3.

The graphical interface of RARS can run in real time, or it can be slowed down which is useful to inspect interesting events, especially if this is done in the replay mode. The interface can also be run at faster than real time, and this is useful for speeding up long experiments. The speed of the "fast" mode is set by a parameter within the RARS code (this is discussed in the RARS list[5]), and this sets the update frequency of the graphical user interface (GUI) relative to the time steps of the simulation. If this update frequency is set low enough then RARS can be made to run at a speed limited only by the power of the main processor, otherwise the simulation is slowed down to make the display visible (the lower speeds can also depend on the processing power of the video card). RARS can alternatively be run in a low graphics mode, but this shows so little detail it is not possible to accurately judge the driving style, only the general line taken by the robot. Running in full graphics but with a very low GUI update rate and the "path trail" option turned on (which displays the robot's path over the previous several hundred time steps) gives the best of both worlds. Clear snapshots are given of the robot and its path taken, once every hundred or so laps; and if closer inspection is needed then the simulation is easily switched into real time, or slow, mode and can then be switched back to fast mode. With this modification and when using the maximum sized hashed representation the execution is sped-up by 16.4 times. Without this saving in real time most of the work of this thesis could not have been completed.

Other RARS-specific work also concerns efforts to decrease the execution time of experiments, and principally concerns pit stops. Unfortunately, pitting turns out to introduce a number of complications.

When reinforcement learning is used with RARS the early learning period involves a lot of crashing. Consequentially, a lot of time is spent in the pits refuelling and repairing damage. This adds a lot to the runtime of experiments, and is not useful because crashing is already penalised by the crash reward. Learning a pitting strategy could be a useful exercise, but it was decided to avoid this additional learning task if possible. Attempts to remove all parts of the pitting code from the RARS source code were not successful after a modest effort was made. The decision was made to retain the pitting but to bypass or effectively mask it

[5] The RARS discussion list is found via http://rars.sourceforge.net/ The discussion referred to is dated 13/04/2004

as much as possible. Doing this also allows for easy re-incorporation of pitting into the environment at a later date, if desired. However in retrospect, it would probably have been simpler to eliminate pitting from the RARS code, if it is not to be used later.

A strategy was devised to reduce the pit time and frequency, and the values used were derived empirically. The maximum allowable damage is increased (exceeding maximum damage puts the car out of the race), and the amount of damage occurring before a pitting request is made is increased. The fact that damage slows the car via "air drag" is not avoided, but effectively adds to the damage reward by increasing the lap time; although, this increased drag has a minor affect. The minimum fuel remaining before a pitting request is made needs to be increased slightly to prevent the car running dry before a pit stop (in which case it is put out of the race). Increasing the fuel load enough to sizeably reduce the pit stop frequency causes the car to slow considerably (due to the extra weight), and so the fuel load is left as standard. The time spent in the pits is set at a constant of one second (and not proportional to damage, when it usually takes minutes).

Another RARS peculiarity is that the crash recovery "stuck()" routine, on very rare occasions, puts the robot into thousands of spins (this looks like a coding bug), and this usually puts the car out of the race. The object is to get the robot to learn from its mistakes, so it must be allowed to continue in these circumstances. The maximum allowable damage can be increased to a large number to allow the car to continue in these situations, but if that is done it then often runs out of fuel and is excluded anyway. Sufficiently increasing the minimum fuel before a pit stop to avoid these very rare situations increases the frequency of all pit stops by about double. A better solution is to calculate the damage accumulated per lap, and if this is above a threshold *and* the fuel is moderately low, then a pit stop is requested. Taking these steps allows experiments to run unfailingly over millions of laps without the robot being put out of the race.

During pitting the robot is under control of the simulator; so a similar approach is taken as used during crash restoration. That is, the time from when the pitting code takes over until the pitting code returns control is treated as non-existent for the reinforcement learning. The two time steps at each end of this period are treated as consecutive. When the robot is restored after a pit stop, it starts from the side of the road at low speed. Therefore the effect of the pit stop omitting the "front straight" does not lead to the robot going from one corner immediately into another (and thereby changing the appearance of the track to the robot), but rather, it appears to the robot as if it is doing another standing start. Also, the effect of the pit stop in saving the robot a lot of travelling distance is not made advantageous because the robot is not given a lap time reward; although, the subsequent lap is timed, albeit from a standing start.

The lap time reward is meaningless on a pit lap because (on track v01.trk) the pit stop bypasses the start/finish line, and the robot is not racing as it passes through the pits. Note that if the robot is involved in a crash at the time of crossing the start/finish line (i.e. it is off the track, but passes the location of the start/finish line) then it does not see the lap flag and does not get a lap time reward, but the lap timer is still reset to zero and restarted as the robot passes the location of the start/finish line. Occasionally missing a lap reward is tolerable for a crash, and not expected to mislead the learning; but this is *not* the case during pitting. Although RARS allocates crash damage at the entrance to the pits (this oddity is discussed in Section 6.1.3), when the robot travels through the pits and passes adjacent to the location of the start/finish line it *does* see the lap flag. Therefore the lap time reward must be specially prevented at these times.

There are several other eccentricities involved in pit stops. For example, RARS gives a damage reward on entering the pits as mentioned above. This is discussed more fully in Section 6.1.3. Also, during the pit entry and exit the robot is steered by the simulator. The robot is called normally but it's steering and speed commands are ignored. Therefore learning must be prevented during these periods. This arrangement appears similar but is not the same as occurs during a crash because during a crash the reinforcement learning part of the robot's code is bypassed by the "stuck()" procedure, (parts of pitting do involve a "crash", but that is a separate matter). This also means the eligibility trace must be "cut" on the pit exit, as discussed in Section 6.1.5.

6.1.1 Effects of back-up routines that occur after crashes

The arrangement for dealing with the time steps and backups surrounding a crash event are discussed in Section 2.5. In preliminary experiments, due to an oversight, the backups occurring after a crash are credited to one time step too far back. This is illustrated by the lighter coloured arrows in Figure 6-1.

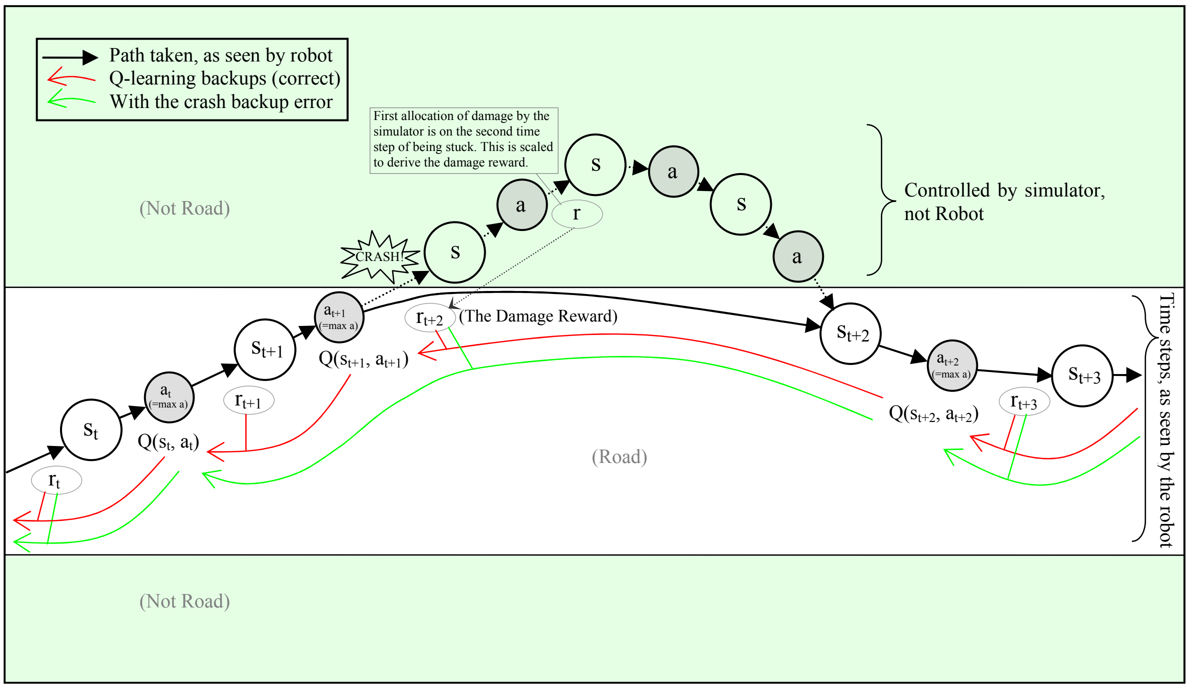

Path taken, as seen by robot
Q-learning backups (correct)
With the crash backup error
First allocation of damage by the simulator is on the second time step of being stuck. This is scaled to derive the damage reward.
(Not Road)
CRASH!
s
a
r
s
a
s
a
Controlled by simulator, not Robot
s_t
a_t (=max a)
r_t
$Q(s_t, a_t)$
s_{t+1}
r_{t+1}
a_{t+1} (=max a)
$Q(s_{t+1}, a_{t+1})$
r_{t+2} (The Damage Reward)
s_{t+2}
a_{t+2} (=max a)
$Q(s_{t+2}, a_{t+2})$
s_{t+3}
r_{t+3}
(Road)
Time steps, as seen by the robot
(Not Road)

Figure 6-1 Backups Across a Crash

Adjacent time steps often use the same state action pair, or very close to the same. This is because the states in adjacent time steps are often very similar, or the same, after discretisation. Therefore a backup that is credited one time step further back would seem likely to have little effect on the algorithm's performance. This shortcoming in the implementation of reinforcement learning was ultimately corrected, and is hardly worth mentioning except that its effect serves to illustrate how a minor detail that appears unimportant can actually turn out to have a major effect on the long term performance in this domain. This can be observed by comparing the graph of Figure 6-2 with Figure 6-3.

Figure 6-3 shows the performance of an experiment *without* the coding oversight, that is, the backups after crashes are performed correctly. Figure 6-2 shows the performance of an experiment *with* the coding oversight. Both experiments are otherwise identical, though some randomness is evident in both graphs.

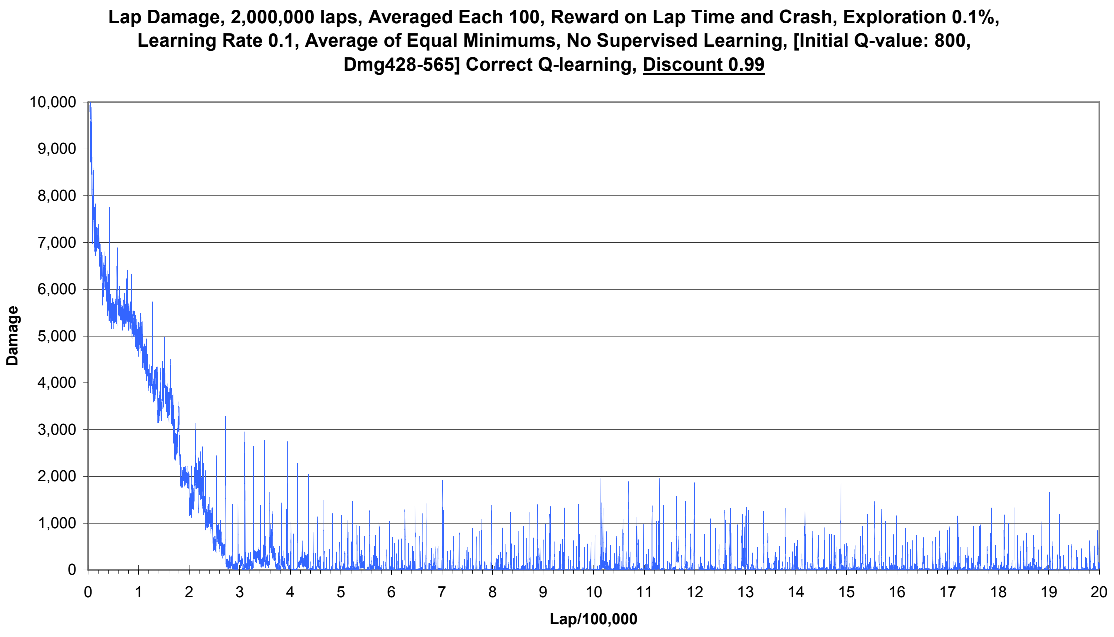


Figure 6-2 Performance With the Crash-backup Coding Oversight

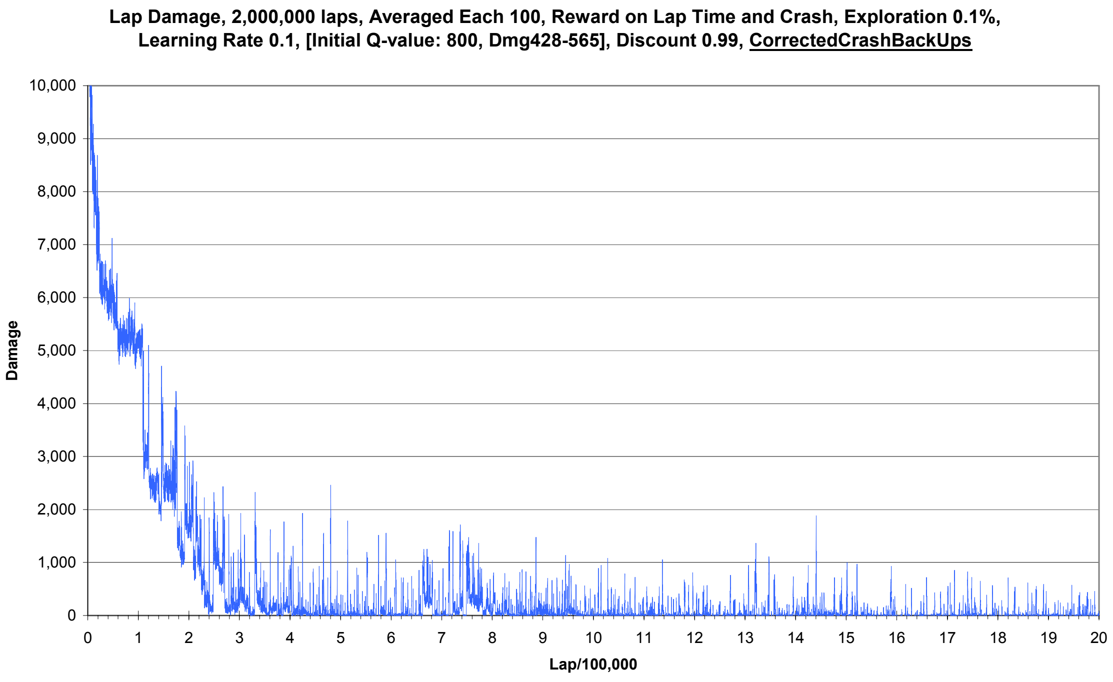


Figure 6-3 Performance With Backups-across-crashes executed correctly

The experiment of Figure 6-2 is the control (baseline) for the experiment of Figure 6-3. Figure 6-2 shows very little reduction in damage from laps 500,000 to 2,000,000. Figure 6-3 shows a gradual, general, reduction in damage from laps 500,000 to 2,000,000. The average damage over the last 500,000 laps of each graph is clearly lower in Figure 6-3. With the code correction, the performance slowly but significantly improves after lap 800,000. In comparison, learning virtually ceases without the correction.

The coding deficiency especially causes learning problems when an exploratory step is immediately followed by a crash. It was measured that a small proportion of the exploratory steps result in a crash on the following time step. In these cases, and with the coding deficiency, the damage reward is not given to the exploratory step, but to the step preceding it. The preceding step is almost certainly an optimal move, and therefore a different state action pair to the following (random) exploratory step. Clearly the 1-step-out deficiency is causing serious misattribution of "blame" in these cases.

6.1.2 Graphing Total-rewards

In some experiments (such as those using speed rewards) it is observed that the lap time and damage graphs level off, which suggests that learning has ceased. But if the total of all rewards received per lap is graphed, the total rewards are observed as still falling. This shows two things: first, learning (according to the reward scheme being used) is still occurring. Second, the reward scheme is no longer helping to improve the lap time in the later stage of the learning process. The aim of this work is to find the path with the fastest lap time, therefore, when the above occurs it shows that the reward scheme is not correct in terms of the aim of this work. These observations show the need to graph total-rewards-received in order to see the true learning progress. Consequentially, for each experiment it is most useful to graph both lap time (to judge learning progress in terms of the thesis goal) and total rewards (to judge learning progress in terms of the reward scheme of the experiment).

The total damage rewards per lap summed with the lap time reward is the quantity the robot is trying to minimise, and in later experiments the speed rewards are included in this sum. Consequently, it is useful to graph this sum, rather than just the lap time or just the damage. This is shown in Figure 6-4, which demonstrates a barely visible decline from

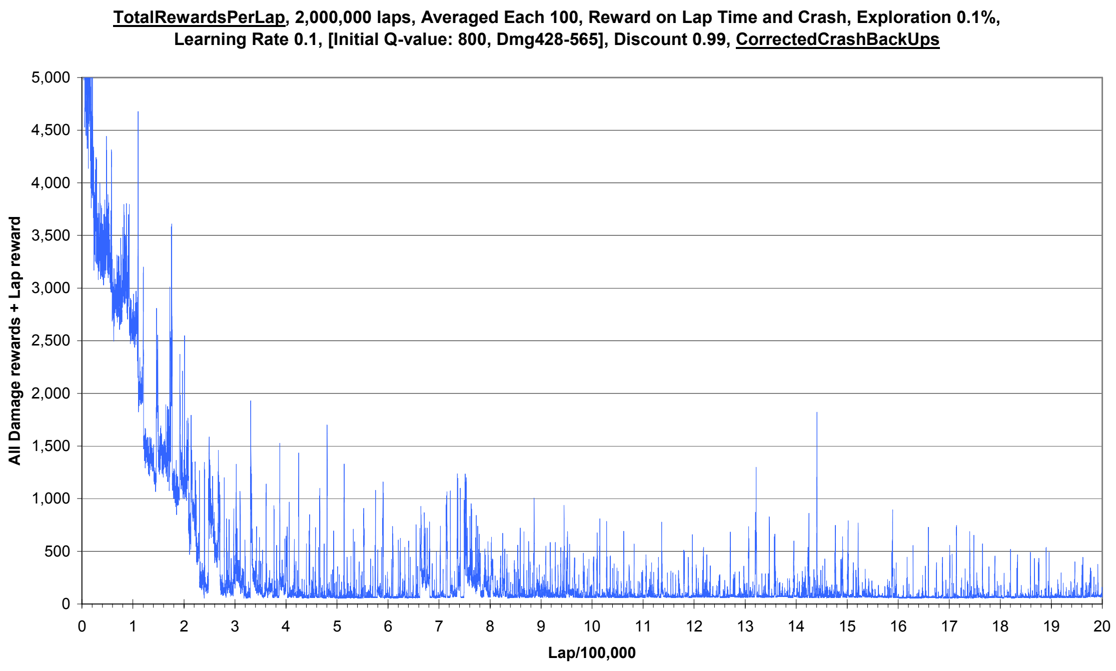


Figure 6-4 Graphing Total-rewards-per-lap Gives the True Measurement of Learning Progress

laps 500,000 to 2,000,000. This decline is made apparent when laps 500,000 to 2,000,000 are extracted, displayed on a separate graph and then fitted with a linear trend line. This graph is not shown (due to space), however the trend line clearly shows a downward slope, which confirms that some degree of learning has occurred over the period. Unlike the damage graph from the same experiment, Figure 6-4 never falls to zero because the minimum total reward possible, per lap, is the lap time (which can not be zero).

6.1.3 Damage Rewards given During Pit Stops

The RARS simulator has a peculiarity in that it allocates a damage reward on pit entrance, that is, a crash is experienced when the robot crosses the road edge on pit entry. This is simply the way RARS works. This “crash” has to be ignored by the reinforcement learning robot because the robot is not in control of its actions during pitting or pit entrance: the simulator is entirely in control. A work-around must be made in the code. Figure 6-5 shows the total rewards per lap from an experiment that is otherwise identical to that shown in Figure 5-8 except that the crash reward given on pit entrance is ignored. The regular rewards that are seen every 80 to 95 laps in Figure 5-8 are not seen in Figure 6-5. Further discussion about this problem is in Section 5.3.2.

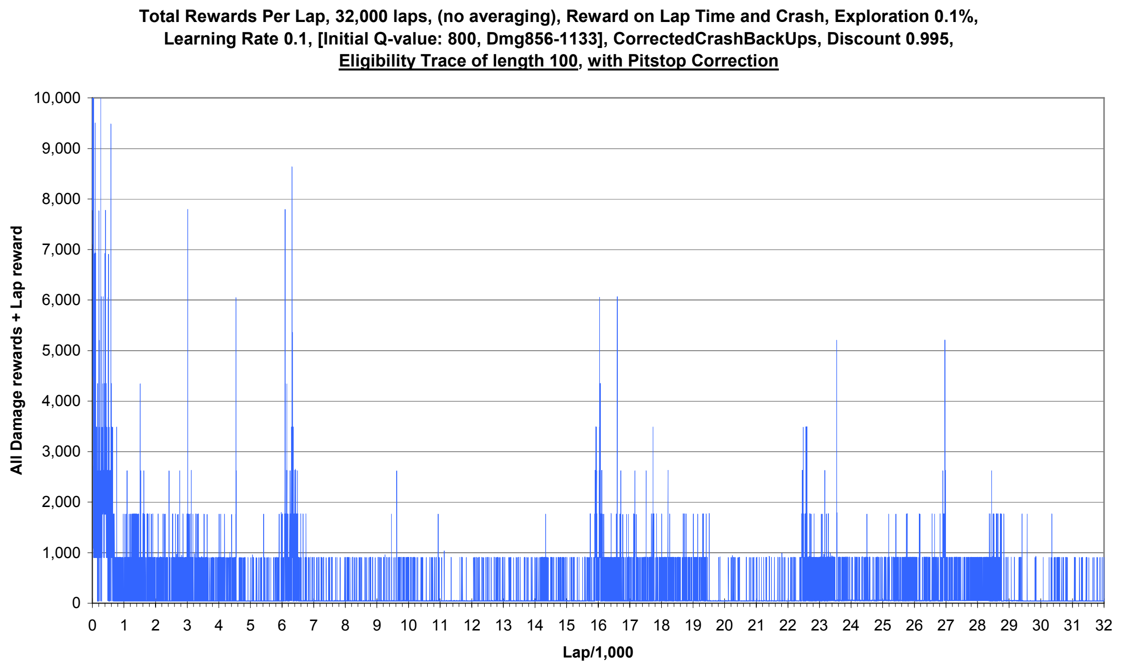


Figure 6-5 Total Rewards per Lap Once the Pit Entry "Crash" is Ignored

As explained for Figure 5-8, Figure 6-5 also has a "blocky" appearance as it shows raw (not averaged) data. Raw data is used as only 32,000 laps are shown and so averaging is not needed. A lap with one crash has a total reward of 856 + lap-time; a lap with two crashes has a total reward of 1712 + lap-time; this gives the graph the stepped appearance.

6.1.4 Back-ups on Pit Entry and Exit

When the robot is sitting in the pits all control is with the simulator. When the robot is in the pits being "refuelled" and "repaired" the code of the robot is not executed, and therefore the robot will not attempt learning at these times. The fuel and damage state is maintained within the simulator.

When the robot is steered into and out of the pits the steering and speed commands are provided by the simulator. Yet during these entry and exit periods the robot's code *is* executed, that is, the robot is passed a situation vector from the simulator and it must return a command vector. Although the robot is consulted during pit entry and exit, its commands are ignored, thus the value of the states reached and the rewards received can not be attributed to the actions of those commands. In other words, during those times the robot

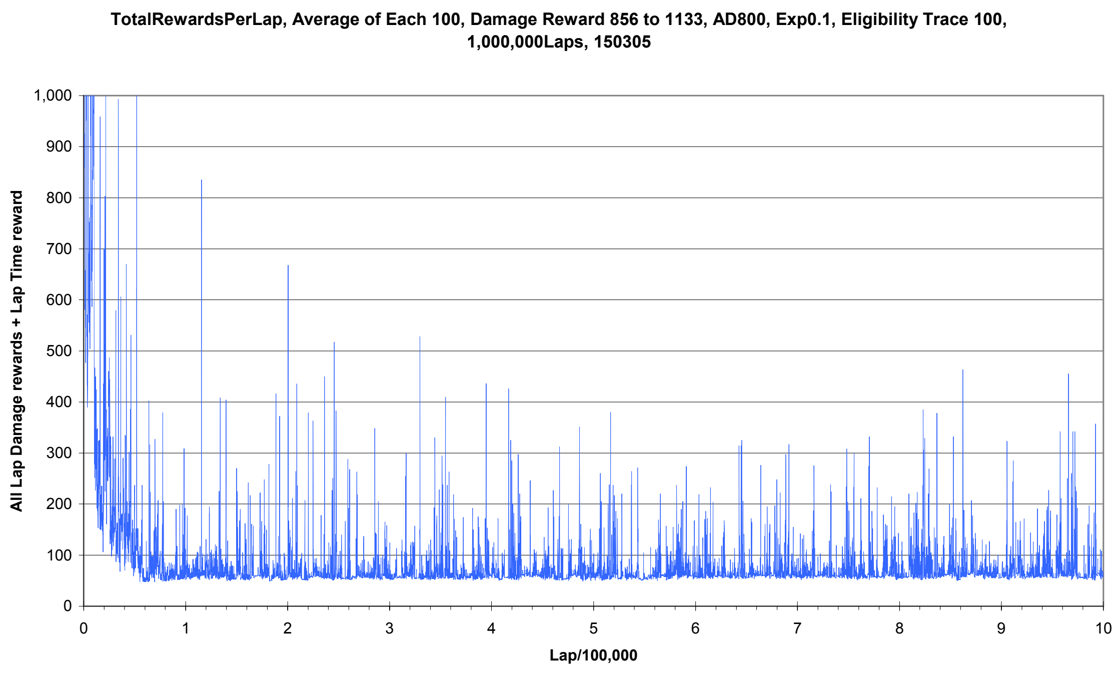


Figure 6-6 Total Rewards per Lap when Learning Is Performed During Pit Entry and Exit

must do no learning, that is, it must not perform back-ups. This is really an oversight in the coding of the simulator (the robot should not be consulted if it is to be ignored), however, this situation must be specially allowed for in the coding of the robot. This gives a considerable improvement in learning performance, which is shown by the difference between Figure 6-6 and Figure 6-7. Figure 6-6 shows the total rewards per lap of an experiment were the robot performs learning during the pit entry and exit periods, and is the control (baseline) graph for Figure 6-7.

Figure 6-7 shows the total rewards per lap when the robot has been modified so it does not perform learning during pit entry and exit periods. Figure 6-7 shows less variation than the control, Figure 6-6. Lowering of variation between laps would be expected if exploration is decreased. The initial learning shown in Figure 6-7 is slightly faster, by about 10,000 laps, than that shown in Figure 6-6. This is visible if the first 100,000 laps are displayed on larger scale graphs, but these are not shown due to space. Therefore, the reduction in variance is probably not due to a reduction in exploration, since decreasing the exploration would be expected to decrease the initial learning speed. Furthermore, exploration has not been knowingly decreased. Therefore the elimination of back-ups during pit stops must have caused the reduction in variance and improvement in learning performance. Misleading Q-values are no longer created by back-ups made during pit entry and exit

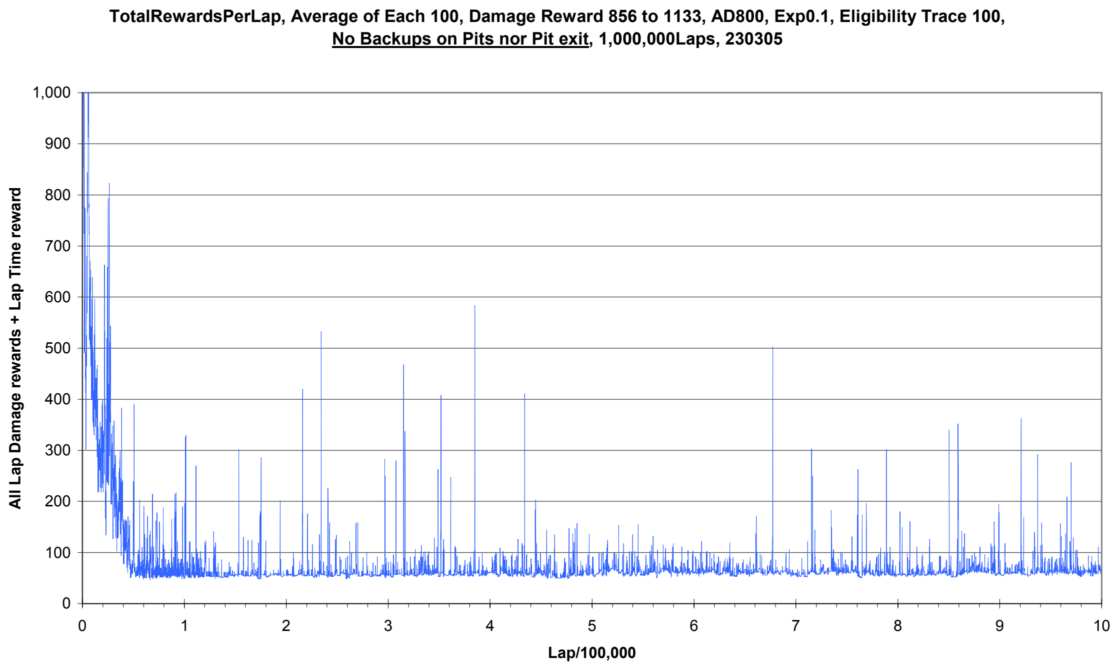


Figure 6-7 Total Rewards per Lap when Learning Is Prevented During Pit Entry and Exit

crediting rewards to vc and alpha action commands that were ignored by the simulator. As these values are incorrect they are, in effect, noise, and this reduction in noise is reflected in the reduction in outliers in Figure 6-7 compared to Figure 6-6.

6.1.5 Eligibility Traces must Not Bridge the Pits

As discussed in Section 6.1.4, learning (i.e. back-ups) needs to be specifically disabled during pit entry and exit. This also means the eligibility trace maintenance (which is the recording of the previous 100 states visited, or whatever the length the trace is set to) can be disabled during pit entry and exit. Furthermore, the trace must not bridge across the pit entry and exit, should the trace ever be set to a length sufficient to allow this. This is because the actions taken prior to pitting are unrelated to the state after pitting, and therefore the back-ups must be "cut" (in the same way as needed after an exploratory move). A simple way to achieve both these things is to reset the eligibility trace on pit exit.

This modification makes no difference to performance that is noticeable in the graphs of 8-million-lap experiments run after the modification is made (graphs not shown, due to space). This is expected, because the eligibility trace of length 100, was too short to bridge the pits. The modification safeguards against future changes of eligibility trace length.

6.2 Analysis of Model, Learning and Performance

Several methods were used to measure and judge performance. Graphs were made of: lap times; lap damages; and total rewards given per lap (total reward is what the robot is trying to minimise). These graphs are judged or compared objectively and subjectively, that is, by using the axis values and by their overall shape. There is a noticeable variation between the graphs of different runs of identical experiments. This is probably mostly due to the randomness included in the RARS simulation. While it would be good practice to do at least five different runs of each experiment, this was almost never done simply because of the time and hardware constraints (each experiment took several days to complete).

Graphs of lap times, lap damages, and total rewards given per lap are used throughout this work, but other measurements were used. The data used for graphing is generated per lap, and over most runs this offers too much detail. To solve this problem, the data is compacted by averaging. For example, after a run of 4,000,000 laps the data is averaged over blocks of 125 laps to give 32,000 data which are then graphed. These are consecutive blocks of laps, not overlapping blocks. Sampling was also tried, e.g. every 125th datum was chosen. Sampling gives graphs with higher peaks and lower frequency. Other statistics are also gathered: the fastest lap time (with damage and lap number); the minimum and maximum damages; the lap with the minimum total-rewards; the median, mean and standard deviation of the total-rewards-per-lap for the last 100,000 laps of the run; and the Q-value range and distribution in the array at the end of the run. Graphs are made of the number of visits to states at different stages of the run to see how much data the decisions were based on. Sometimes an objective description of the driving behaviour is noted in English. This can be very useful as the driving style can not accurately be derived from the statistics. This can also be distracting, misleading and time consuming as it is often more useful to observe the trend in performance over many hundreds of thousands of laps, as shown by the graphs. Clearly it is impractical to watch the robot over this many laps.

6.2.1 Q-value Ranges

Figure 6-8 shows the minimum and maximum Q-values (action-values) that are present in the look-up table when measured once each lap for the first 1,100 laps. The initial Q-values are 800. This experiment has 0.1% exploration, a learning rate of 0.1, a discount of 0.9, damage rewards ranging from 428 to 565 and no initial supervised learning. Measuring the rate at which the minimum and maximum action values spread apart from the initial default value (which is 800 in this experiment) gives an indication of the speed at which learning is progressing. Figure 6-8 shows results from very early in learning: before the time or damage curves start to fall. Both curves use the same y-axis scale. Both curves change very rapidly, then level-off in an asymptote-like manner.

Figure 6-9 shows the minimum and maximum Q-values (action-values) that are present in the look-up table when measured once each 1,000th lap for 2,000,000 laps. Figure 6-9 is over a much larger number of laps than Figure 6-8. To help reveal details, the two curves in Figure 6-9 are each given different y-axis scales, which are also different to the y-axis scale of Figure 6-8. These curves (Figure 6-9) also appear to be falling/rising in an asymptote-like manner. However, while the curves in Figure 6-8 appear to be levelling out

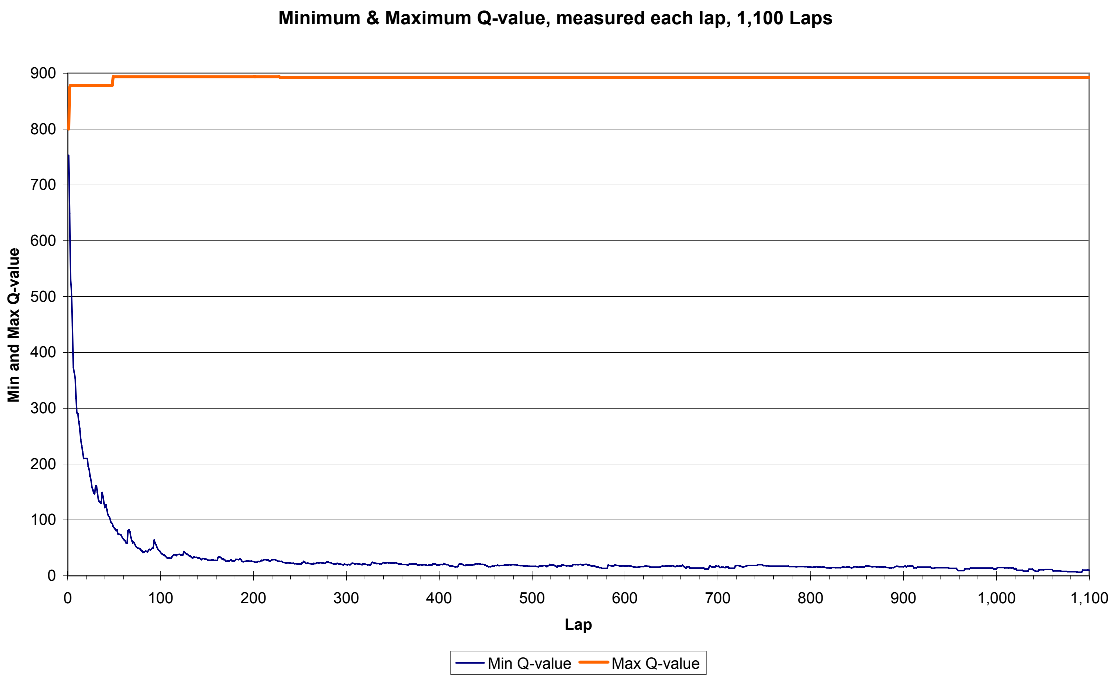


Figure 6-8 The Minimum and Maximum Q-value, Measured Each Lap, for 1,100 Laps

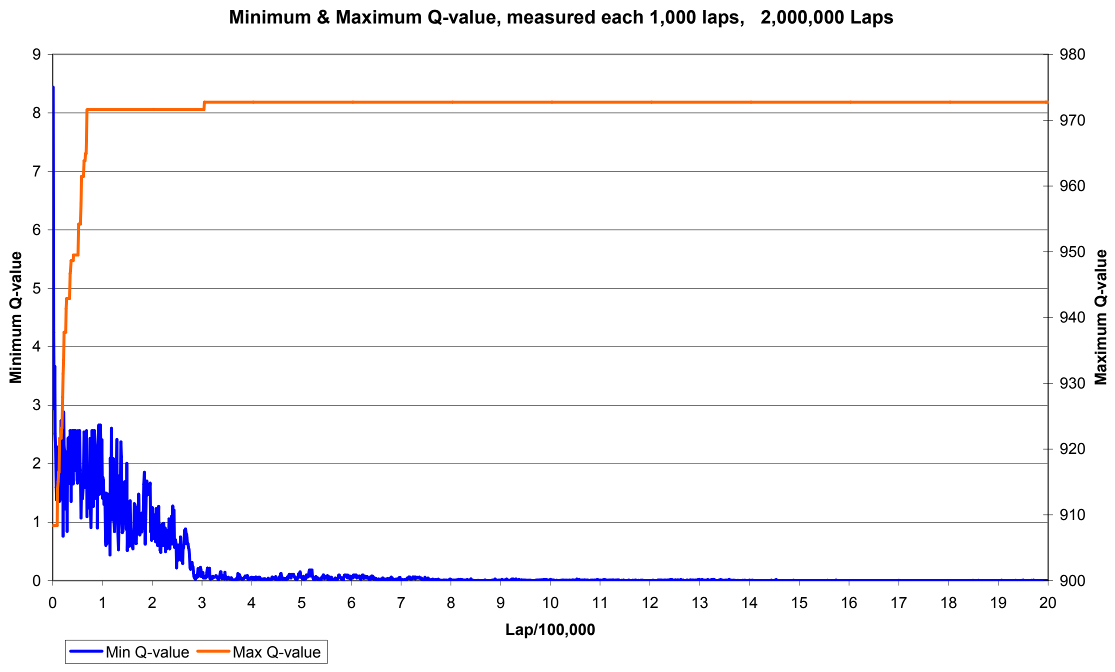


Figure 6-9 The Min. and Max. Q-value, Measured Each 1,000 Laps, for 2,000,000 Laps

at values around 10 and 900, Figure 6-9 (which is from the same experiment) shows that this is not the case: Over a much longer time period the minimum curve continues to fall and the maximum curve continues to rise. Yet, this can not continue forever because that is the purpose of discounting—to ensure there is an horizon to the Q-values. The discount factor used here is $\gamma = 0.9$. Figure 6-8 and Figure 6-9 also show that although the minimum curve generally falls, it does at times rise.

6.2.2 Final Q-value Distributions

Figure 6-10 to Figure 6-12 show the distribution of Q-values held in the look-up table after a set number of laps. For example, Figure 6-10 shows the contents of the look-up table after 2,000 laps of learning. Figure 6-10 is generated by extracting each cell of the look-up table that is not a default value (i.e. not the initial value, which is 800 in this experiment). These values are then sorted from lowest to highest, and then plotted. Therefore the graph must be monotonic. The values on the x-axis only serve to give an indication of the number of look-up table cells involved. The values on the y-axis show the range of the Q-values. A group of identical values will result in a flat, horizontal, area on the graph. The default values are omitted to make the graph more readable. In early learning most states

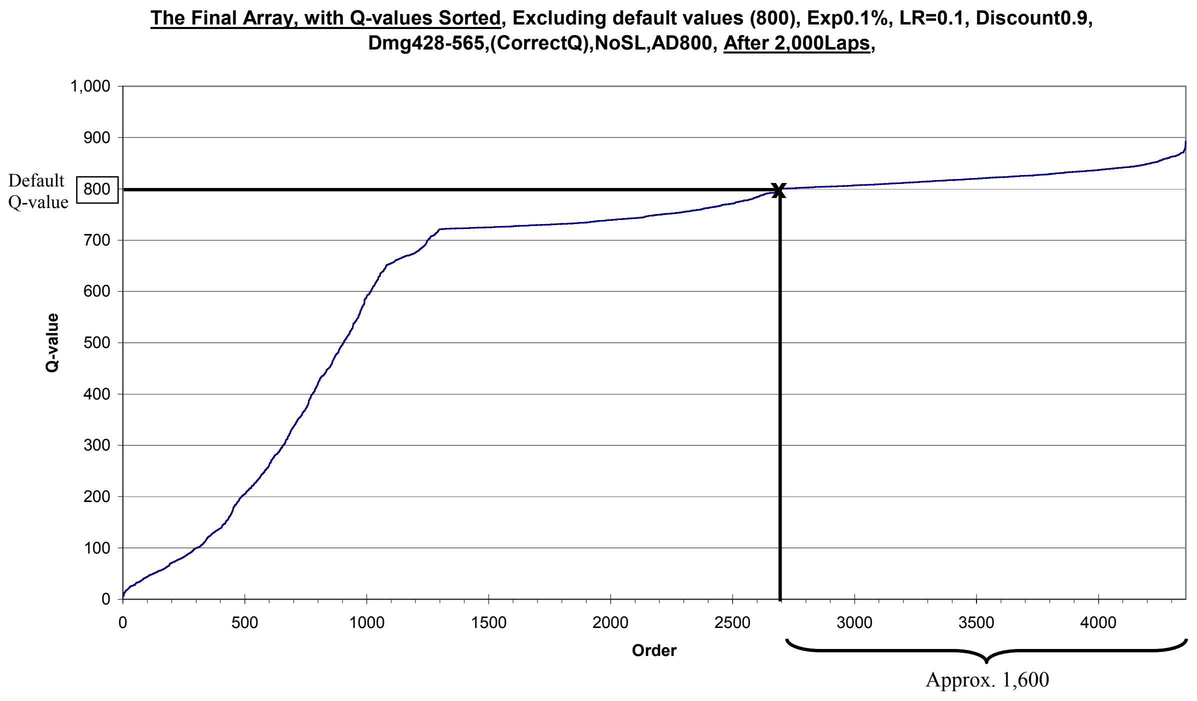
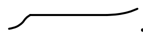


Figure 6-10 The Distribution of Q-values, After 2,000 Laps, Discount 0.9

are unvisited, so most values are still the default value and the full graph would appear something like: . A few of the state-action-pairs with the default Q-value of 800 may have been visited, but this is unlikely as the value is almost certain to be changed by a back up after a visit, even if only by a tiny amount (the Q-values are stored as type float, so a tiny change is recorded). The Q-values shown are those of any state-action-pair, (not only those of the optimal action in each state). The x-axis scales differ between Figure 6-10 to Figure 6-12.

The x-axis of Figure 6-10 shows there are about 4,350 table entries that no longer have the initial Q-value after 2,000 laps. It shows the initial value of 800 is probably pessimistic because roughly 2/3 of the values (on the x-axis) have been learnt to have a lower value than 800 (remember, lower is better—faster—in this domain). This is good because, as discussed earlier, optimistic initial values can cause too much unscheduled exploration. The large change in gradient at about 1,300 on the x-axis shows about 2/3 of the values are within about ±80 of the initial value (the right-hand 2/3 of the graph), while about 1/3 of the values have a much greater range and go as low as a Q-value of about 10 (the left-hand 1/3 of the graph). It would be instructive to add a third dimension to this graph, that of number-of-visits to each state action pair. This information is recorded in later

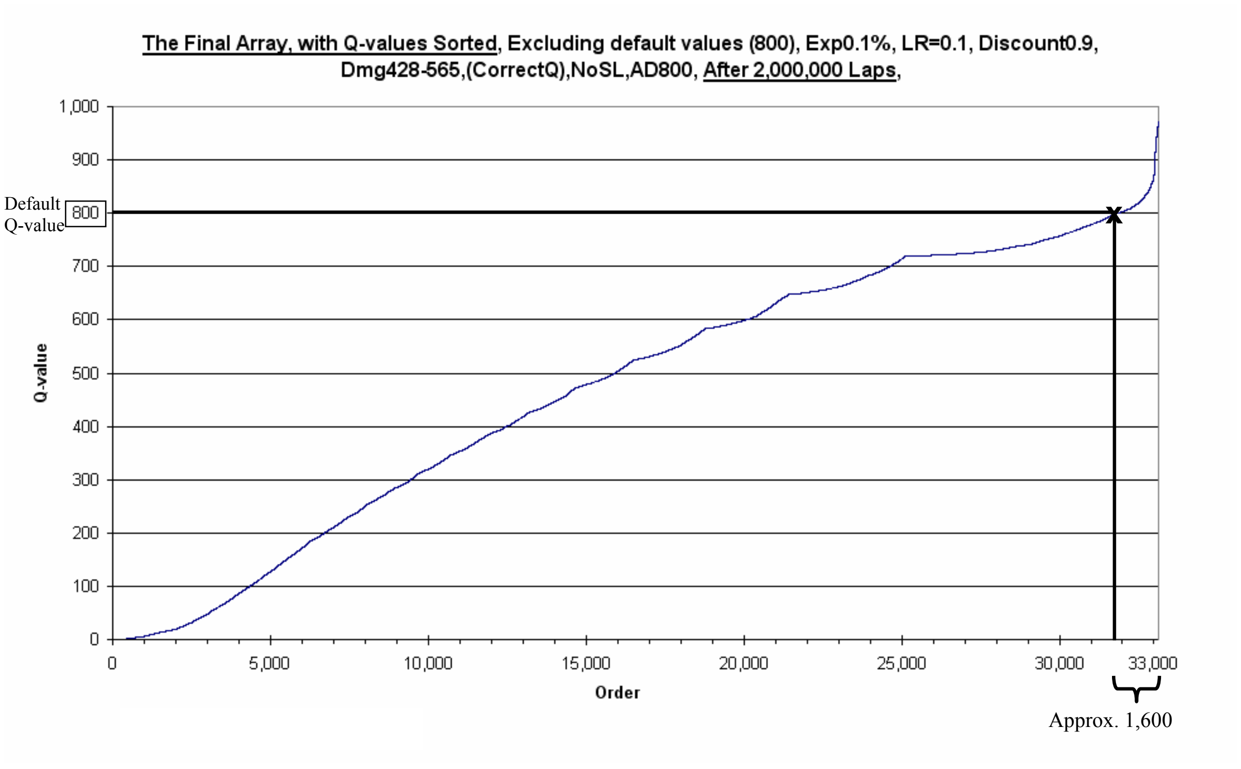


Figure 6-11 The Distribution of Q-values, After 2,000,000 Laps, Discount 0.9

experiments, but not in the experiments of Figure 6-10 to Figure 6-12. This information is expected to show that the actions with better (lower) Q-values are visited more often.

The experiments shown in Figure 6-10 and Figure 6-11 are identical, except Figure 6-10 shows the Q-value distribution after 2,000 laps while Figure 6-11 shows the Q-value distribution after 2,000,000 laps. Both experiments have a discount of 0.9, (and exploration 0.1%, learning rate 0.1, initial values of 800, damage rewards of 428 to 565).

The x-axis of Figure 6-11 shows that there are about 33,200 table entries that no longer have the initial Q-value after 2,000,000 laps. Nearly all of these are found to have better than the initial Q-value, and this confirms its pessimism. There are about the same number of state action pairs with values above the default, that have had at least one visit, after 2,000,000 laps as there are after 2,000 laps (about 1,600 in both cases, as marked on the graphs). This is because the agent has learnt to avoid states where the best action available is a poor one. Although the agent has taken a few more very poor actions (probably exploratory) which are seen in Figure 6-11 as values above 900 (these do not occur in Figure 6-10). The scallop shapes in the central part of Figure 6-11 are intriguing, and in the absence of further investigation their meaning and cause can only be guessed. As for Figure 6-10, it would be instructive to add a third dimension to Figure 6-11, that of number-of-visits to each state action pair.

The experiments shown in Figure 6-11 and Figure 6-12 are identical, except that the experiment of Figure 6-11 uses a discount of 0.9 while the experiment of Figure 6-12 uses a discount of 0.99. The general shapes of Figure 6-11 and Figure 6-12 have some similarities: it looks as if the point marked with a "**x**" in Figure 6-11 has been pulled to the left to make Figure 6-12. Both graphs show about the same number of state action pairs visited (33,000 and 34,000 on the x-axes). Figure 6-12 shows action values range up to about 1,400, whereas in Figure 6-11 they range up to about 1,000. A far larger proportion of the state action pairs have values above the initial value in Figure 6-12 (about 40%), than is the case in Figure 6-11 (about 5%). This indicates that the default value of 800 is not as pessimistic in the context of the experiment shown in Figure 6-12, as it is in the context of the experiment shown in Figure 6-11. A discount value of 0.99 gives *less* discounting than a value of 0.9 (a discount value of 1.0 gives no discounting). This means an agent using $\gamma = 0.99$ is more "farsighted" than one using $\gamma = 0.9$. That is, the Q-values in such an agent represent the sum of the value of future actions taken, if following the

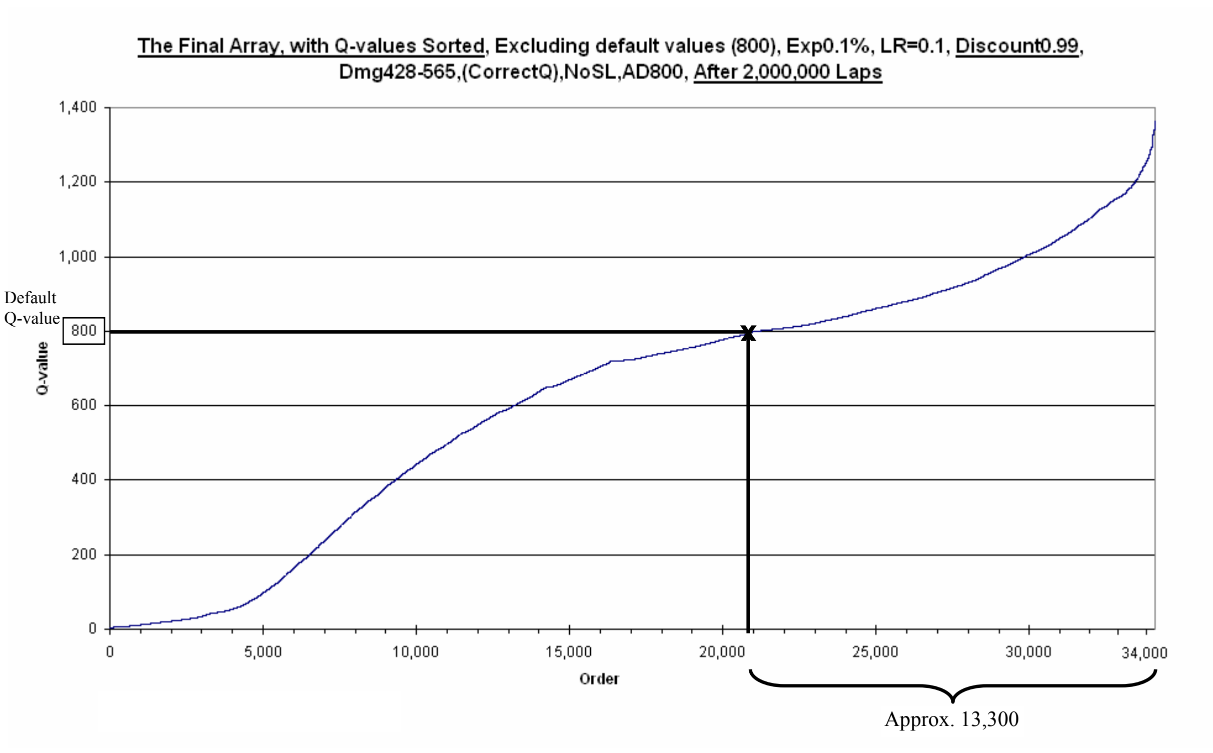


Figure 6-12 The Distribution of Q-values, After 2,000,000 Laps, Discount 0.99

optimal policy, but *with greater weight on the more distant actions* than is the case when using $\gamma = 0.9$. With this in mind, the comparison between Figure 6-12 and Figure 6-11 may indicate that fewer actions appear "good" when a more farsighted view is taken, although more "very good" actions are found (there are about 4,000 with Q-value 100 or less in Figure 6-11, compared to about 5,000 with Q-value 100 or less in Figure 6-12). These things may indicate that the agent works better using a discount of 0.99 than a discount of 0.9. Indeed, a comparison of the corresponding total-rewards-per-lap graphs show the experiment using a discount of 0.99 learns much faster and reaches a more stable performance than does the experiment using a discount of 0.9.

The more central location of the default (initial) Q-value in Figure 6-12, compared to its location in Figure 6-11, indicates it is a more optimistic value in the context of the experiment of Figure 6-12. This means there will be more "nonscheduled" exploration occurring due to the initial value being optimistic. This suggests running an experiment with a higher initial value, say 1,100, (but otherwise identical parameters) to test the usefulness of reducing "nonscheduled" exploration back to a similar amount as occurs in the experiment of Figure 6-11. Such an experiment was never tried (as is the case with a number of other interesting side-issues arising in this work) due to time constraints. Nonscheduled exploration is discussed in Section 3.9. As a broader remark, the observation of the location of the default (initial) Q-value, within the Q-value distribution graph, may be a useful method of automatically determining the pessimism/optimism of an initial Q-value (although this provides information *a posteriori*).

6.2.3 Statistical Presentation: Sampling versus Averaging Graphs

The graphing tool used in this work can handle a maximum of only 32,000 data. This is more than ample resolution for drawing graphs. However, most experiments are run over a much greater number of laps than 32,000. Therefore some method must be used to summarise/compress the data from long experiments before they can be displayed on a single graph. There are two simple ways of doing this: sampling the data or averaging the data. Each method produces very different looking graphs from the same data, and each highlights different aspects of the data. A decision was made to use averaging in this work, for reasons illuminated below. However, for the sake of completeness, a sampling graph is shown and compared to an averaging graph from the same data.

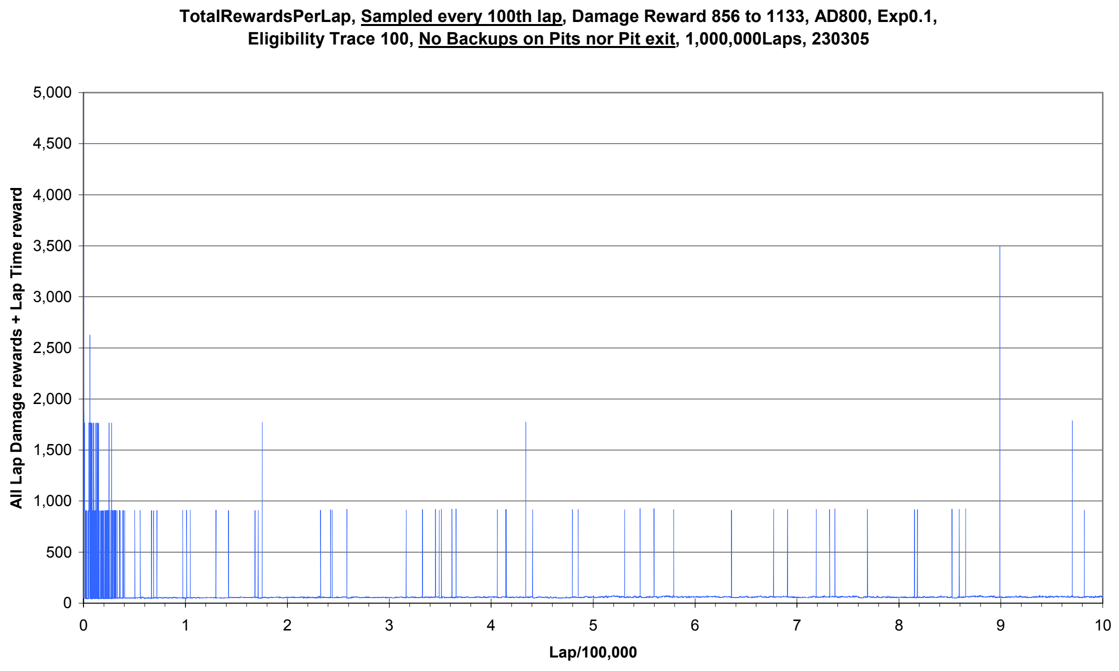


Figure 6-13 Total Rewards per Lap, Using Sampling Every 100th Lap to Compress the Data

Figure 6-13 shows the same data as Figure 6-7, yet the two graphs look very different. To summarise the data (so the graphing tool can cope with it) every 100 laps are averaged to produce one datum for Figure 6-7. To generate Figure 6-13 every 100th lap is sampled. Each method has different effects. Averaging guarantees the outliers are captured, although, unless there is a group of outliers or the value is extreme, an outlier comes to appear less sizable than it actually is if the averaging period is long. Sampling reveals the discretised nature of the crash rewards (that is why Figure 6-13 appears "blocky"). With sampling, the peaks are higher (note the larger y-axis scale of Figure 6-13 cf. Figure 6-7); and the peaks are much more sparse. This sparseness gives a better illustration of the sparsity of the damage rewards later in learning, because it more closely resembles what is seen if the raw data is inspected. The averaging graphs tend to make the rewards look more frequent because if averaging is done, say, each 100 laps only one of those laps need have a reward for it to show on the graph—the other 99 laps could be reward-free (damage reward, not lap reward).

The exaggeration of reward frequency is possibly the biggest disadvantage of using averaging to compress the data. This effect needs to be kept in mind when interpreting the graphs. For example, to graph an 8,000,000 lap experiment its rewards are averaged every

250 laps to provide 32,000 data. Adjacent points can barely be distinguished on a printed graph with a resolution of 32,000 on the x-axis, and certainly can not be distinguished on a computer display. If every fourth point shows a rise on the graph it appears as if the whole graph is raised, and this implies the agent is crashing continually. But in fact it means it may only be averaging one crash in every (4 × 250 =) 1,000 laps. That is about one crash per 1,000,000 time steps on track v01.trk! This is less than some expert human drivers.

However, using averaging to compress the data tends to give a much smoother graph, with outliers that are less radical. This is easier on the eye in that it makes it easier to judge trends in its shape. For these reasons, averaging is the method used in this work. However, the exaggeration of crash rewards must be kept in mind.

6.2.4 Number of Visits to State-action-pairs

The motivation for producing Figure 6-14 to Figure 6-17 is to illustrate how much information the action choice decisions are based on at various stages of the learning. This is to answer the question: When a state is visited and an action is chosen as the optimum (or randomly, on exploratory moves), then how many previous times has that state action pair been visited (i.e. how reliable is its Q-value)? Figure 6-14 to Figure 6-17 are generated by observing how many previous visits the state action pair has had each time a state action pair is visited (each state action pair has a number-of-visits parameter associated with it, in the later experiments). Tallies are kept of: how many times any state action pair with zero previous visits is chosen; how many times any state action pair with one previous visit is chosen; how many times any state action pair with two previous visits is chosen; and so on, up to those with 1,000 previous visits; and a total is also kept of how many times any state action pair with more than 1,000 previous visits is chosen. The number-of-previous-visits is shown on the x-axis, and the tally (number of occurrences) is shown on the y-axis.

Nearly all state-action-pairs will register more than once on these graphs. That is because if a state-action-pair is visited, say, 10 times then it must also have been visited once and twice and three times, etc. This may seem redundant, but the idea of these graphs is to indicate how much information *decisions* are based on (decisions are the subject of interest, not state action pairs). The least accurate decisions are based on state action pairs with zero previous visits (shown on the left hand end of the x-axis); more accurate decisions are based on state action pairs with, say 10, previous visits; better accuracy comes from those state action pairs with, say 100, previous visits; and so forth. The total number of decisions made (i.e. the total number of actions performed) is given by the area under the graph (coloured grey) including the tail not visible off the right hand end.

State-action-pairs with zero *previous* visits have been visited once. State-action-pairs that are never visited are not shown. Therefore, the 9,000,000 maximum on the y-axis of Figure 6-14 shows there were about 9,000,000 different state-action-pairs used in the run (i.e. 9,000,000 different state-action-pairs that were visited at least once).

Figure 6-14 shows the total visits over all 8,000,000 laps of an experiment. It shows millions of decisions are made during the course of the experiment that are based on Q-values of actions with 10 or less previous visits. The area under Figure 6-14 that falls below 10 on the x-axis is roughly 70,000,000. That means about 70,000,000 decisions were based on Q-values of actions with 10 or less previous visits. Most of these poorly informed decisions are made early in learning, as can be seen more clearly in the next couple of figures.

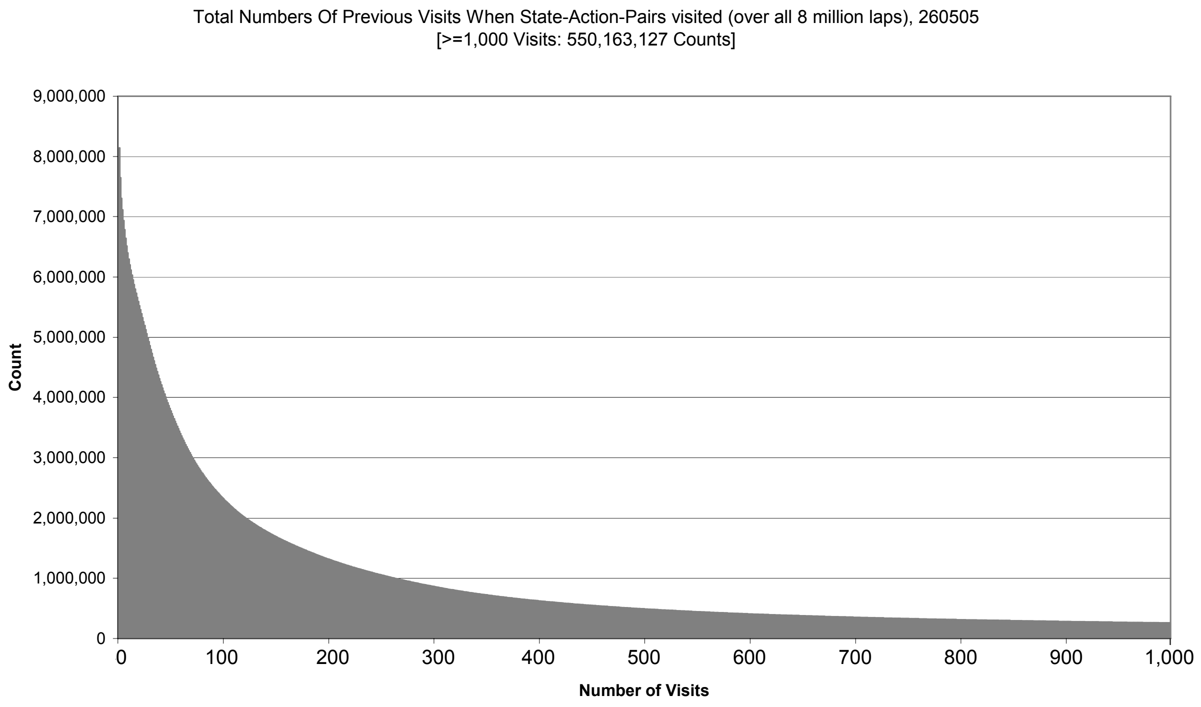


Figure 6-14 Number of Counts Versus Number of Previous Visits to State-action-pair, across 8M Laps

The shape of Figure 6-14 appears to show that the greater the number of previous visits to a state action pair, the less likely it is to be used. However, there are two indications that this is incorrect: Figure 6-14 levels-off asymptotically; and the area under the tail of Figure 6-14, off the right hand end of the graph, is very large. The area of the complete graph beyond 1,000 visits is 5,501,631,270 decisions. The area below 1,000 is roughly 1,500,000,000 decisions. This means that most of the data used to produce Figure 6-14 is not visible, and that most decisions are very well informed (because most decisions are based on state action pairs with 1,000 or more previous visits). This last point is shown more clearly by Figure 6-15 to Figure 6-17.

Figure 6-15 shows the decisions made during the first 400,000 laps of the 8 million lap experiment of Figure 6-14. It shows about 50,000,000 decisions based on state action pairs with 10 or less previous visits (this is the approximate area below 10 on the x-axis)—those are 50 million of the 70 million decisions observed for the same statistic in Figure 6-14 (Figure 6-14 includes the data from Figure 6-15). This shows that most of the decisions (of the 8 million lap experiment) made using state action pairs with 10 or fewer previous visits

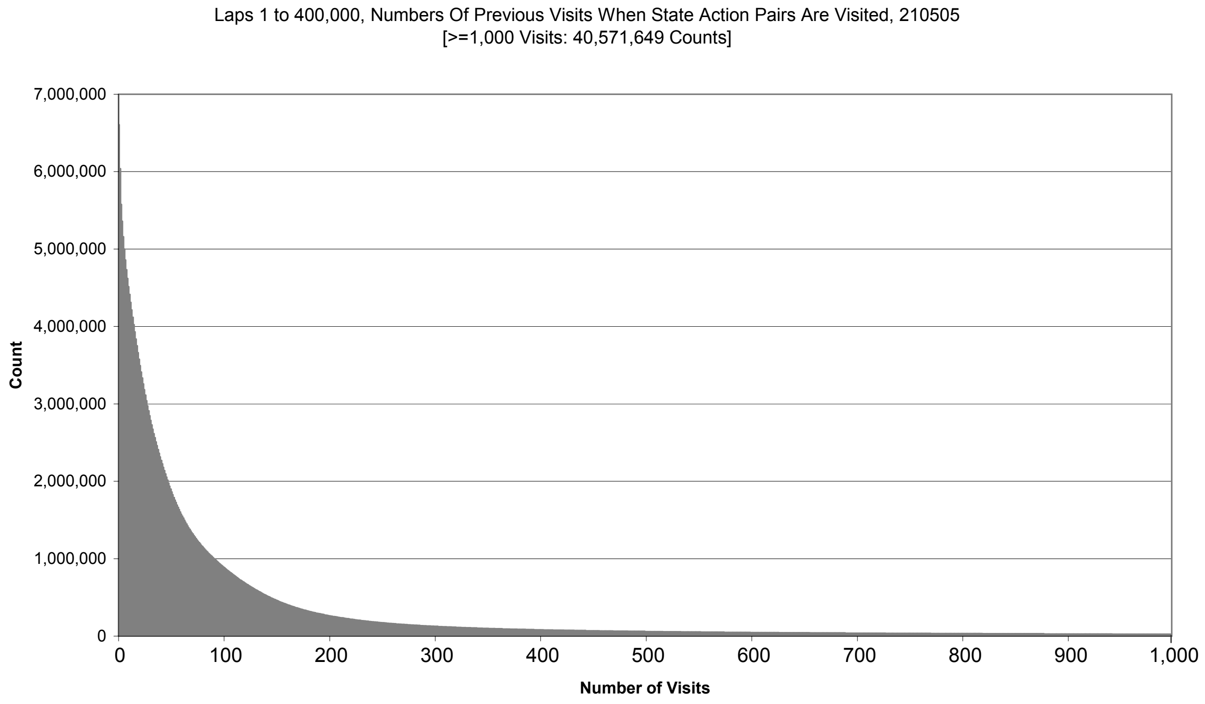


Figure 6-15 Number of Counts Versus Number of Previous Visits to State-action-pair, across First 400,000 Laps

are made in the first 400,000 laps. A similar observation can be made for decisions involving state action pairs with 30 or less previous visits. The left hand end of Figure 6-15 is close in area to the left hand end of Figure 6-14 (say, below 30 on the x-axes) and this confirms that the majority of lesser-informed decisions are made in the first 400,000 laps.

Figure 6-15 shows about 7,000,000 different state action pairs have been visited in the first 400,000 laps (shown by the maximum value on the y-axis). Figure 6-14 likewise shows about 9,000,000 different state action pairs have been visited in all of the 8 million laps. This means about 7/9$^{\text{ths}}$ of all state action pairs used are visited in the first 5% of the laps, in this experiment. The entire graph partly-shown by Figure 6-15 has an area above 1,000 on the x-axis of only 40,571,649 decisions. Figure 6-15 itself has an area of roughly 300,000,000 decisions, and nearly all of these involve state action pairs with less than 200 previous visits. This clearly shows that decisions made in the first 400,000 laps are nearly all based on state action pairs with 200 or less previous visits.

Figure 6-16 shows the decisions made during the 400,000 laps that fall between lap 1,600,000 and lap 2,000,000 of the 8,000,000 lap experiment of Figure 6-14. Note the

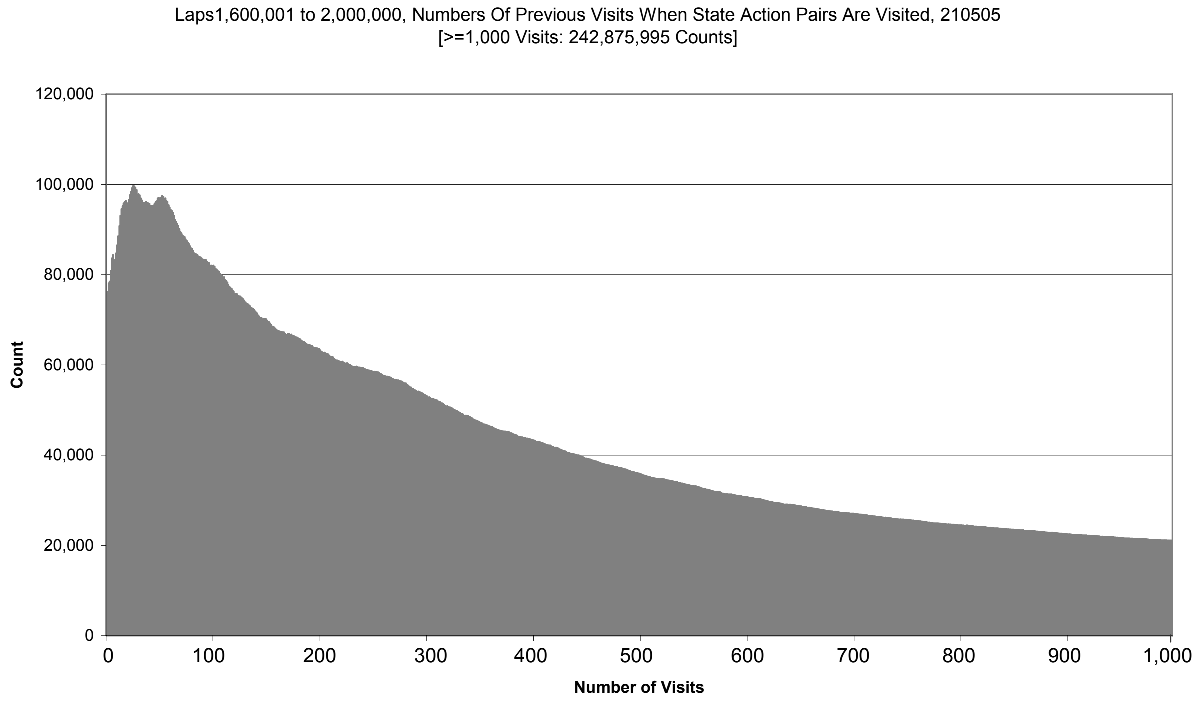


Figure 6-16 Number of Counts Versus Number of Previous Visits to State-action-pair, across Laps 1.6M to 2M

smaller y-axis scale in Figure 6-16 compared to Figure 6-15. When number-of-previous-visits (on the x-axis) is zero, the count is less than 80,000 (on the y-axis). This shows that less than 80,000 state action pairs are visited for the first time in this 400,000 lap period. This compares to 7,000,000 state action pairs visited for the first time in the first 400,000 laps.

Most of the area of the graph partly-shown by Figure 6-16 is in the tail above 1,000 on the x-axis (242,875,995 counts). The area of Figure 6-16 itself is roughly 45,000,000 decisions. The area under Figure 6-15, including the tail, is about 340,000,000 decisions, while the area under Figure 6-16, including the tail, is about 290,000,000 decisions. Each decision corresponds to one time step. Therefore, the 400,000 laps of Figure 6-16 take about 15% fewer time steps to complete than do the 400,000 laps of Figure 6-15. This is because the 400,000 laps between lap 1,600,000 and lap 2,000,000 are, on average, travelled at a speed about 15% faster than the first 400,000 laps—the result of successful learning.

Figure 6-16 has an interesting peak at roughly 50 on the x-axis. The peak is of about 100,000 on the y-axis. One explanation is that by laps 1.6 to 2 million the agent has learnt enough about its environment to ensure that on almost all states reached there is a good indication of the near-optimum action. In other words, most of the time the agent knows a "good" action to take (it may not be the optimum action) and therefore it seldom moves into a state it has never seen before (which is one with zero previous visits—the left-most end of Figure 6-16). A "good" action will not take the risk of moving into an unvisited state (a "good" action must have had some "good" values backed up to it have to become "good", therefore it must lead into a "good" next state). The agent also seldom moves into states where the best action has had only, say, 10 or fewer previous visits (the near-left-most end of Figure 6-16) because by this stage of the learning the seldom-visited states are probably poor ones. All this implies that by laps 1.6 to 2 million the amount of inherent/non-scheduled exploration, as discussed in Section 3.9, has largely stopped. Further to this, the amount of scheduled exploration has also begun to diminish by this stage of the learning. For these two reasons, the state action pairs with around 30 to 80 previous visits are seen more often than those with less than 30 previous visits, and this corresponds to the peak in Figure 6-16. However, the number of visits to state action pairs with more than 80 previous visits is by far the larger group, and this corresponds to the remaining part of Figure 6-16 (including the unseen tail) to the right of 80 on the x-axis.

From this analysis, it is reasonable to expect that as learning progresses the number of state action pairs visited for the first time will continue to decrease, as also will the number of visits to state action pairs with a "small" number of previous visits. This means the early part of the graph is expected to get lower (on the y-axis) and flatter as learning progresses. This is just what is shown by Figure 6-17.

Figure 6-17 shows the decisions made during the 800,000 laps that fall between lap 7,200,000 and lap 8,000,000 of the 8,000,000 lap experiment of Figure 6-14. Observe the much smaller y-axis scale in Figure 6-17 compared to either of Figure 6-14 or Figure 6-15 or Figure 6-16. Most of the area of the graph partly-shown by Figure 6-17 is in the tail (i.e. above 1,000 on the x-axis) where there are 558,293,237 counts. The area under Figure 6-17 is roughly 4,000,000 decisions. This means there are about 562,000,000 decisions (time steps) in the final 800,000 laps of the experiment.

Figure 6-17 shows less than 1,000 state action pairs visited for the first time. These visits are very likely to all occur on exploratory moves. By this stage of the learning, many

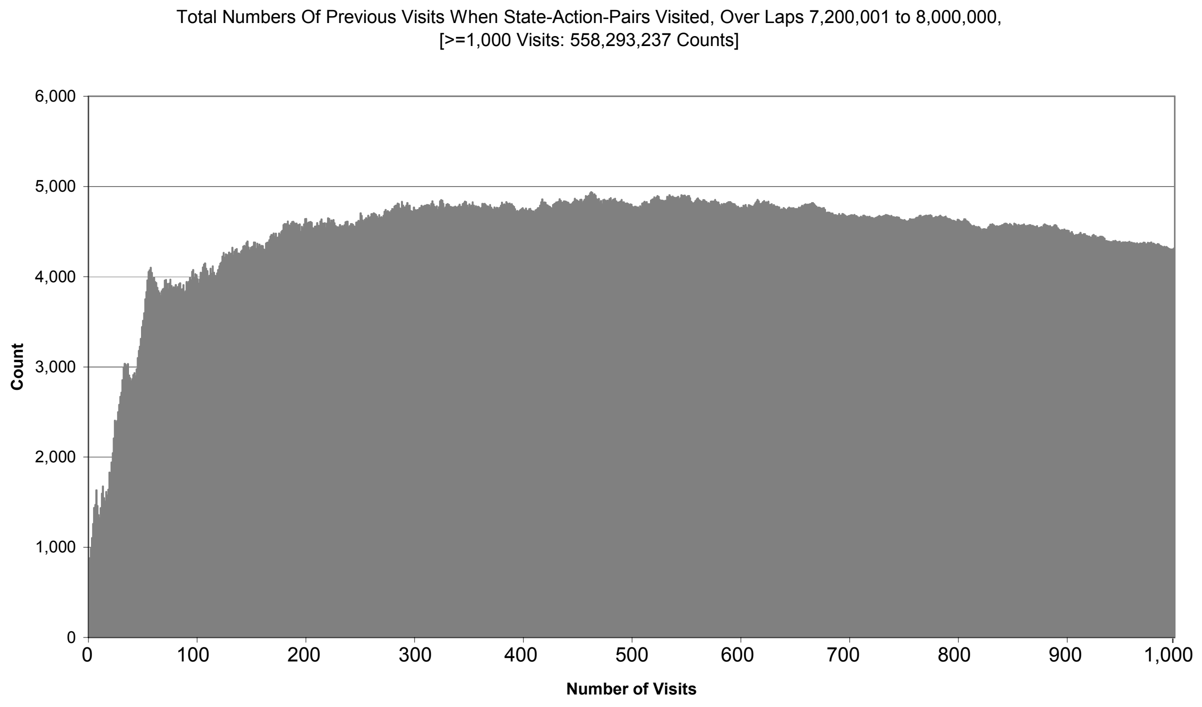


Figure 6-17 Number of Counts Versus Number of Previous Visits to State-action-pair, across Laps 7.2M to 8M

exploratory moves probably land the agent in a state that has a previously-visited optimal action (these are, nevertheless, useful exploratory moves); only some of the exploratory moves probably land the agent in a state that has either never been visited or all actions that have been previously visited (in that state) are suboptimal (in either case the optimal state action pair will not have been previously visited—and will show in the first column of Figure 6-17). Another explanation for a visit to a previously unvisited state action pair is the randomness in the simulation causing the move to be stochastic.

The shape of Figure 6-17 ties in nicely with the explanation given for the shape of Figure 6-16. This is given in detail below Figure 6-16. It is likely to be informative to extend Figure 6-17 (and Figure 6-16) well beyond 1,000 on the x-axis. It is guessed that there could be some sort of lump in the graph, perhaps a bell shape, which would show a group of state action pairs (with a sizeable number of previous visits) that are visited most often, falling away on the right to the left. The graph is possibly skewed so the lump is well to the right hand end ("skewed left"). These graphs were not made due to time constraints.

Figure 6-14 looks nothing like Figure 6-17, and has no peak like Figure 6-16. This is because Figure 6-14 shows the data from all 8 million laps, which includes the very first to the very last laps. On the other hand, Figure 6-15, Figure 6-16 and Figure 6-17 show groups of laps from specific periods of learning, and Figure 6-16 and Figure 6-17 do not happen to include any of the earliest laps (unlike Figure 6-15).

Another statistic useful for judging learning progress is to measure the reliability of the action choice decisions versus the number of laps completed (i.e. versus time). This is shown in Figure 6-18 and Figure 6-19. A comparison between Figure 6-15, Figure 6-16 and Figure 6-17 shows that as the number of laps increase, the number of states visited once or only a few times decreases greatly. This is also illustrated by Figure 6-18 and Figure 6-19. Figure 6-15 to Figure 6-19 all show that as learning progresses the amount of experience that decisions are based on (that is, the number of previous visits to visited state-action-pairs) is reasonably large.

6.2.5 Proportion of State-action-pairs Visited, with less than five previous visits

Figure 6-18 and Figure 6-19 show (on the y-axis) the proportion of state-action-pairs that, when visited, have had less than 5 previous visits; versus (on the x-axis) the number of laps completed. To clarify, the proportion is: (number of visits to state-action-pairs that have had less than 5 previous visits) / (number of visits to any state-action-pair, i.e. number of time steps). For Figure 6-18 this is measured every 10th lap (i.e. for lap 10, lap 20, lap 30, …), that is, this statistic is sampled. Figure 6-18 covers laps 10 to 320,000. Figure 6-18 and Figure 6-19 use data from the same experiment.

The x-axis shows laps completed, which in effect shows time passing (unlike the x-axis of Figure 6-14 to Figure 6-17). In effect, Figure 6-18 and Figure 6-19 use the same source of data as used in the first 4 columns of Figure 6-14 to Figure 6-17 (i.e. number of visits to state action pairs with 4 or less previous visits), but display it versus time.

Figure 6-18 shows that in very early learning nearly all action decisions are based on little information (i.e. most of the actions selected have been used less than five times before). However, this decreases rapidly as learning progresses, and by about 70,000 laps around

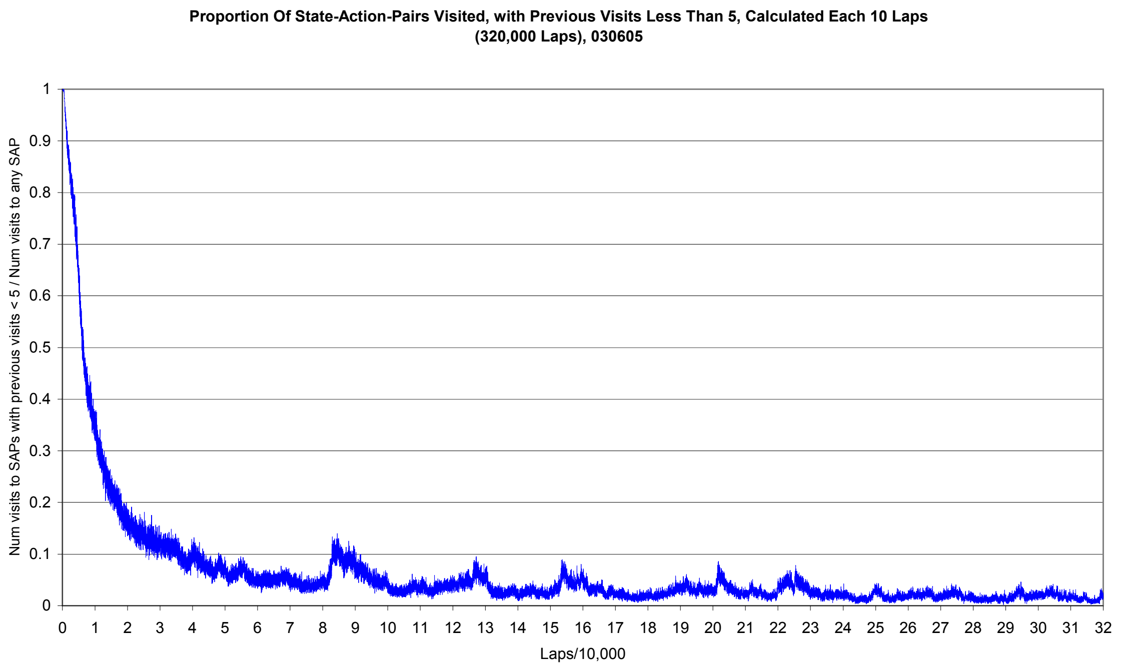


Figure 6-18 Proportion of State-action-pairs Visited That Have Less Than Five Previous Visits, across 320,000 Laps

5% of the actions selected have been used less than five times before. From lap 70,000 onwards the decrease is more gradual, with a few rises (possibly due to exploration), but consistent in general.

Figure 6-19 shows (as does Figure 6-18) the proportion of state-action-pairs that, when visited, have had less than 5 previous visits; versus the number of laps completed. For Figure 6-19 this is measured every 100th lap (i.e. for lap 100, lap 200, lap 300, …). Figure 6-19 covers laps 100 to 3,200,000. The two traces on Figure 6-19 show the same data, but at different resolutions on the y-axis: The light grey uses the left hand y-axis; the darker trace uses the right hand y-axis.

Figure 6-19 shows the trend over 3.2 million laps, of the data shown in Figure 6-18. Figure 6-19 shows the same tendency as seen in Figure 6-18, that is, a gradual decrease in the proportion of actions decisions that are based on little information. Enlarging the scale of Figure 6-19 (shown as the light grey line), or inspecting the raw data, shows that by lap 3 million most laps never use action decisions based on "little" information. Figure 6-18 and Figure 6-19, as do Figure 6-14 to Figure 6-17, help to confirm that as driving progresses, the amount of knowledge that decisions are based on increases, and is reasonably large. This gives some degree of confidence in the quality of the learning that is occurring.

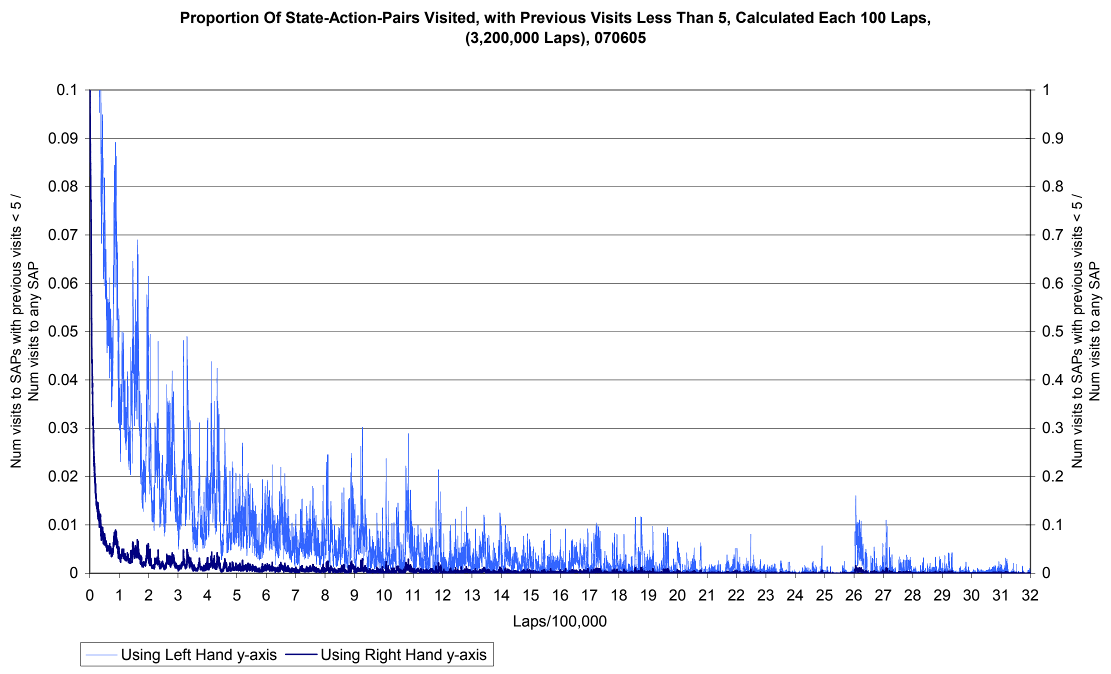
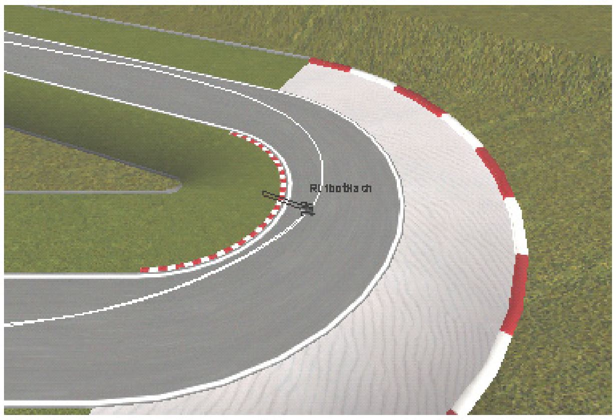
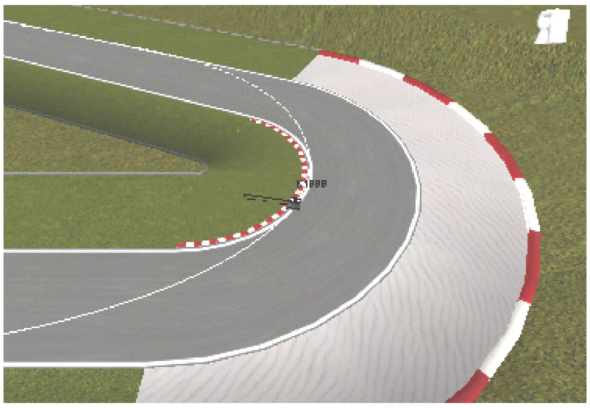


Figure 6-19 Proportion of State-action-pairs Visited That Have Less Than Five Previous Visits, across 3.2M Laps

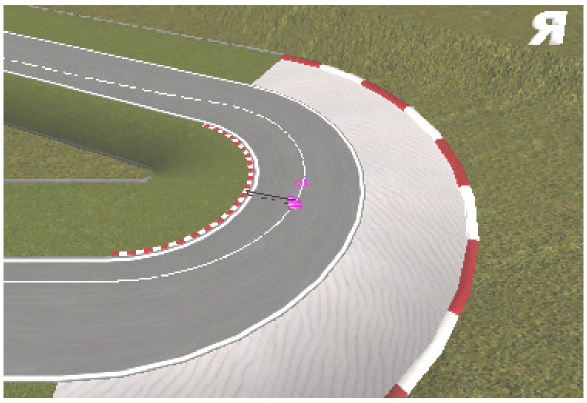
6.3 Screen Shots



A. The Reinforcement Learning Robot



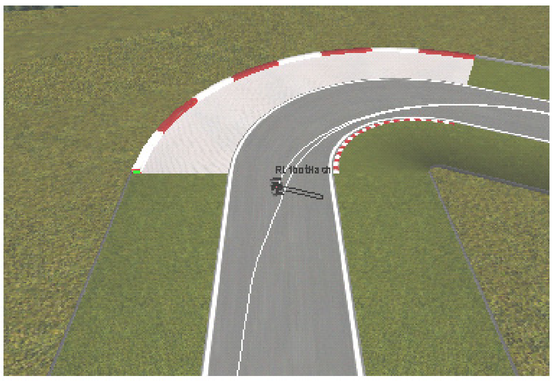
B. Robot k1999 (expert)



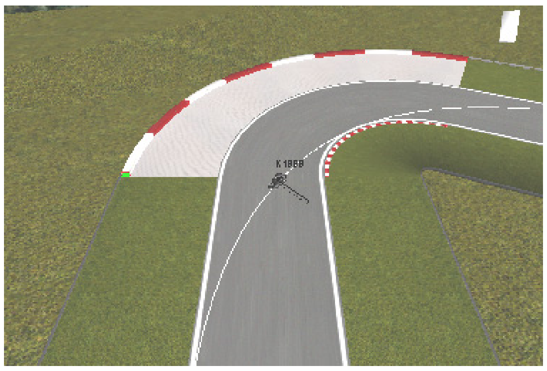
C. Robot 01 (basic)

Figure 6-20 The First Corner of Track v01.trk

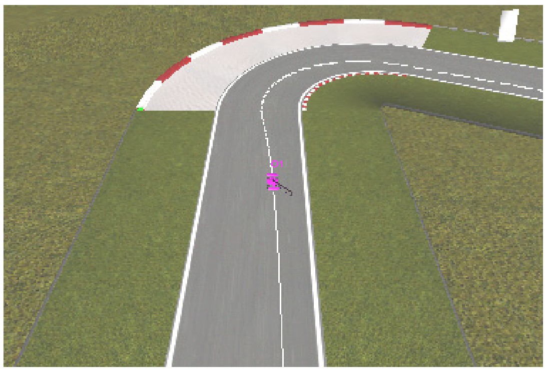
For the reinforcement learning robot, the screen shots shown in Figure 6-20-A, Figure 6-21-A and Figure 6-22-A are taken at about lap 6,100,000 in the same experiment as is shown in Figure 5-6, Figure 5-7, Figure 6-14 and Figure 6-17. Figure 6-20 to Figure 6-22 show that the driving lines taken by the reinforcement learning robot through the corners of track v01.trk are more similar in shape to those of the expert robot, k1999, than to those of the simple robot, 01. Robot 01 more-or-less just follows a fixed lane. Robot k1999 is proven to be an excellent driver as it has won several seasons of RARS competitions. There is no need to go into an analysis of the physics of the best driving line—k1999 graphically shows the near-ideal line. Without a detailed description, it should be clear from inspection of Figure 6-20 to Figure 6-22 that the reinforcement learning robot's driving lines are not very different to those of k1999, and are much improved from when it started learning, when all it could do was to drive in a straight line and crash.



A. The Reinforcement Learning Robot

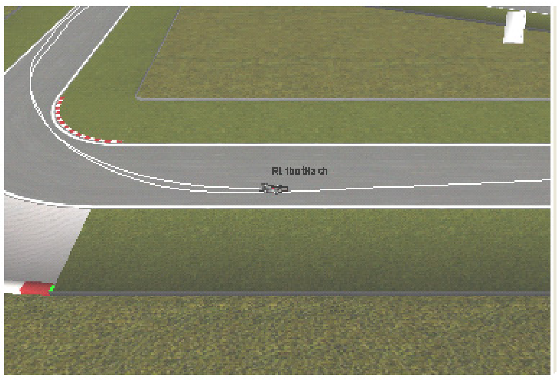


B. Robot k1999 (expert)

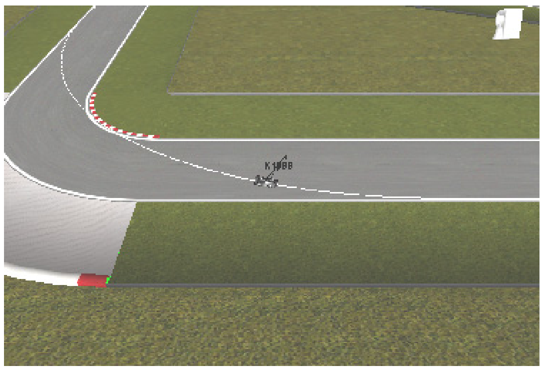


C. Robot 01 (basic)

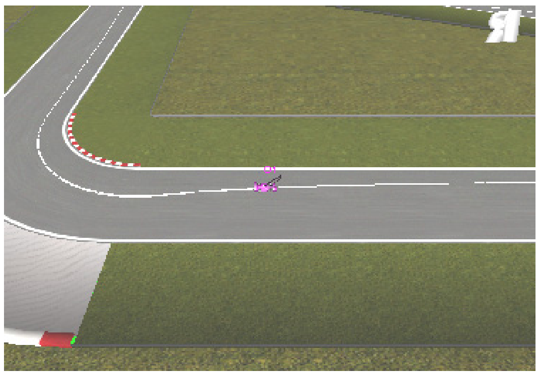
Figure 6-21 The Second Corner of Track v01.trk



A. The Reinforcement Learning Robot



B. Robot k1999 (expert)



C. Robot 01 (basic)

Figure 6-22 The Third Corner of Track v01.trk

6.4 Chapter Summary

This chapter first dealt with peculiarities of the RARS domain that give rise to implementational issues affecting the reproducibility of most of the experiments in this work. These tiresome issues are of little interest from a reinforcement learning perspective, but need to be dealt with if reinforcement learning is to be used within RARS. These matters include the use of a minimum crash reward; a modification to the simulator code that gives an execution speed-up of 16.4 times; and a range of issues concerning pit stops. Pit stops need to be hidden from the reinforcement learning parts of the robot’s code, if pit stops are to be excluded from the problem formulation. Pit stops need to be requested as seldom as possible to avoid increasing the run time; yet must be requested judiciously enough to ensure the robot is never excluded from the race. The lap time reward must not be given on pit laps, as it is meaningless on these occasions. RARS gives a damage reward on pit entry, and this must be ignored. During the pit entry and exit the robot is steered by the simulator, however the robot is called normally yet it’s steering and speed commands are ignored. Because the robot is not in control, learning must be specially prevented during these periods. This also means the eligibility trace must be “cut” on the pit exit.

Chapter 6 then details the variety of methods employed to judge the progress and performance of the learning algorithm. Graphs of lap times, lap damages, and total rewards given per lap are used throughout this thesis. Chapter 6 demonstrates some of the other measurements used, such as graphs of the Q-value range and distribution in the array at the end of the run; graphs of the number of visits to states, at different stages of the run, (to see how much data the decisions were based on); and statistics such as: the fastest lap time (with damage and lap number); the minimum and maximum damages; the lap with the minimum total-rewards; and the median, mean and standard deviation of the total-rewards-per-lap for the last 100,000 laps of the run. These measurements all indicate that the learning is well-founded. Finally, some screen shots of the RARS circuit are given that compare the driving of the reinforcement learning robot with that of an expert robot and a basic robot, and these show the reinforcement learning robot’s driving style approaches that of the expert robot.

7 Summary and Future Work

7.1 Summary

This thesis investigated the use of reinforcement learning to learn to drive a race car at the minimum lap time in the simulated environment of the Robot Automobile Racing Simulator. The essence of reinforcement learning is learning by interacting with an environment. The agent observes the effects of its actions and from this discovers how to select actions so as to achieve some desired goal. This learning framework is applicable to a large range of problem domains and requires no human input nor prior domain knowledge, and therefore has exciting ultimate potential such as for industrial, economic or social domains. The real world tends to be the most difficult, but often the most useful, domain to solve. However, although RARS is a simulator, it is reasonably complex, and its optimal solution has not yet been found by any method, including heuristic methods that utilise extensive prior domain knowledge (although some solutions found by these methods are close to optimal). These things make the use of reinforcement learning within RARS a subject worthy of study.

Chapter 3 set out to prove the feasibility of using reinforcement learning in the RARS domain—outlining the representation for the action-value function, the rewards and the manner of exploration. It showed that a simple tabular representation adequately handles the very large search space, and that the learner does manage to converge to a steady state; just not a very good one. The advantage of using a tabular method is that it is both simple to implement and has already been proven to converge for reinforcement learning. The experiments made use of sample training data (provided by a preprogrammed robot car) to bootstrap the system with some basic driving skills prior to the reinforcement learning. Unfortunately, this initial training stage introduced a bias in the direction of learning that seemed to prevent (or slow down considerably) the subsequent reinforcement learning from advancing to an optimal (or at least near-optimal) level of driving skill.

Chapter 4 removed the bootstrap process, forcing the reinforcement learning to proceed from scratch with no prior experience whatsoever. This resulted in slower initial learning, but the learning performance was improved over the long term.

Exploration turned out to be an important matter. Sufficient exploration is needed to force the agent to take risks and thereby discover profitable areas of the search space. This benefits the long term performance. Yet too much exploration was seen to degrade the immediate performance due to the random actions of the exploratory moves. This so called exploitation/exploration dilemma is successfully addressed by reducing the amount of exploration as the learning proceeds. This was achieved both explicitly (by laps completed) and by random proportional selection (which reduces exploration with experience). However as it turned out, initial experiments suffered from too much exploration in the earlier stages of learning due to the random tie-breaking method used to choose between equally good actions. An averaging method of tie-breaking was shown to improve learning performance in the earlier stages. Furthermore, it was found that using large initial Q-values much reduced the amount of non-scheduled (i.e. uncontrolled) exploration. This was important to allow control of the exploration.

Using more than one source of reward is unusual in reinforcement learning and has not yet been formally proven to converge. The aim of this thesis is for the robot to achieve the minimum lap time, therefore the correct reward is the lap time. However, this is only available once per lap. It was found, in Chapter 3, that using both lap time and crash damage as rewards resulted in much faster learning, and particularly helped the robot to learn crash avoidance. However in the long term increased lap time was traded-off against decreased damage. This made sense as the total reward was still falling, but did not help in the aim of minimum lap time. Altering the ratio of damage reward size to lap-time reward size did not solve the problem (Chapter 5). It turned out that using a third reward of instantaneous tangential speed did solve the trade-off problem. This allowed lap times to continue to fall in the long term. Speed rewards are the most frequent as they are available every time step, however maximum average speed is not the aim of the robot as it is not the same thing as minimum lap time. The reward regime, in particular the use of three different sources of rewards, is very much an open question and leaves plenty of scope for further investigation.

Methods were used to speed up the execution time of the simulation. The use of eligibility traces to update the 100 previously-visited state action pairs on every time step gave a learning speed-up of about 25 times. A modification in the actual RARS code gave an execution speed-up of about 16 times.

In Chapter 5 hashing was used to free-up memory that was then used to increase the discretisation resolution. The increased resolution slowed down the learning by a large factor, due to the increased search space. However, the robot's driving performance improved. Also, a rough form of generalisation was implemented using a simple nearest neighbour method. This improved the transfer of knowledge between different driving tasks. The gains shown by the use of finer-grained discretisation and generalisation confirms that the use of function approximation is probably worthwhile. A function approximator would replace the tabular representation, and would provide compression (as does the use of hashing) thus allowing increased resolution, and also good generalisation. It is known that function approximators need careful implementation with reinforcement learning. The preliminary use of hashing and nearest neighbour generalisation to imitate the effects of function approximation has served to indicate that it is probably worthwhile investing the time to implement function approximation within RARS in the future.

A number of domain-specific issues arose during the work—for example, discounting was needed because of the very long episodes. It turned out that pit stops needed special treatment and this and other loose-ends are covered in Chapter 6.

This thesis has demonstrated that it is possible to use reinforcement learning within the domain of RARS to produce a robot driver of fair ability. A number of issues were raised that require further investigation, and from this it appears likely that the ability of the reinforcement learning driver could be improved significantly.

Achievements

A tabular representation is successfully used in a domain of high dimensionality to represent the Q-value function of reinforcement learning. The domain has 7 state variables and 2 control variables, some of which are continuous.

The reinforcement learning robot achieves better lap times than the basic heuristic robots. The heuristic robots are programmed using knowledge of the domain, while the reinforcement learning robot starts with virtually no domain knowledge.

The driving lines taken by the reinforcement learning robot are in appearance more like those taken by the best heuristic robots (e.g. “k1999”) than those used by the basic heuristic robots (e.g. “01”). From personal domain knowledge it is known that the reinforcement learning robot’s lines are more conducive to speed than are the lines taken by the basic heuristic robots.

The use of extremely long episodes fortuitously revealed shortcomings that otherwise would not have been noticed. When using the highest discretisation resolution the learning is very slow, and millions or tens of millions of laps are needed to see results. This corresponds to billions (i.e. thousands of millions) or tens of billions of time steps. To allow the episodes to run for this long a number of minor peculiarities in both the RARS domain and the implementation of reinforcement learning had to be weeded out and addressed. Surprisingly, some of the matters that appeared trivial actually turned out to have a major affect on the learning performance in the long term. Had learning not been run over such long episodes these problems would not have been noticed, and the poor learning performance could have either been attributed to some other cause, or the RARS domain would have been considered unsuitable or too difficult for the methods used. This shows how easy it is to be led astray by one small rare error. Unless a piece of computer code has been proved correct by using formal methods, then that code can not be used to truly prove anything. Unfortunately, applying formal methods to anything as modestly complex as RARS and its robots is enormously difficult and time consuming.

7.2 Future Work

Numerous possible directions for further investigation are mentioned throughout each chapter of this work as the ideas arise in the discussion, and these include side-issues that were identified that could lead to a new thread of investigation. This section presents a broad overview. Much of the discussion here on future work relates closely to Section 2.2 (Previous Work on Learning within RARS) and Section 2.3 (Remaining Questions in the Field of Reinforcement Learning).

A couple of methods commonly used to improve performance in reinforcement learning applications are not used in this work. Domain knowledge may be used to transform the input state parameters into measurements that are easier to learn from. Redundant "features" can be added to the input representation to direct the learning. An example of both techniques are used in the work of Rémi Coulom [2002] on RARS. Two of the transformed parameters used by Coulom are the distance to the centre of the track and the angle of the car's velocity relative to the direction of the track. These parameters are not directly provided by RARS but are derived from those that are. Redundant features used by Coulom include the distance to the wall ahead and the angle of incidence with the wall ahead. These things are worth trying in the future work of this thesis because Coulom found their use improved learning performance; although they are slightly against the aim of this thesis in that they require the use of prior domain knowledge.

The best performing experiments in this thesis use three different sources of reward. The use of multiple reward sources is not common, and the best way to do this is not clear. There are many possibilities to test—for example: ways of distinguishing the reward sources; ways of combining the rewards; ways of changing the relative influence of each reward as the learning progresses, and how to do this automatically. Chapman and Kaelbling [1991] approach this problem by keeping track of more information: instead of using Q(*s, a*), they record D(*s, a, r*)—the discounted future probability of receiving reinforcement *r* after performing action *a* in state *s*. They call this D-learning and report superiority over Q-learning.

Another possibility is to use centripetal acceleration as a reward, either alternatively or in addition to the other rewards used. The idea is that minimum lap time might be achieved by maintaining maximum turning force (which is measured as centripetal acceleration), although this does not apply on the straights. This is discussed in more depth in [Cleland, 2003].

The use of hashing and nearest-neighbour generalisation in this thesis "tests the water" for using function approximation. As an intermediate step, tile coding could be tested [Sutton and Barto, 1998; Watkins, 1989]. This provides generalisation and is also a tabular method. It is also often hashed to save space. The amount of generalisation can be varied for testing by altering the arrangement and overlap of the tiles.

The success of hashing and nearest-neighbour generalisation suggests the use of function approximation. There are several possibilities, but in the reinforcement learning domain those that allow on-line incremental use are probably more elegant than those that require batch updates. Methods to try include the tree-based G-learning [Chapman and Kaelbling, 1991], and variations on G-learning [Uther and Veloso, 1998; Pyeatt and Howe, 1998c]. However, function approximation has proved to have difficulties when used with reinforcement learning: errors can accumulate to render them useless [Smart and Kaelbling, 2000; Gordon, 1995, 2001]. Function approximators have to be implemented carefully if they are to work with reinforcement learning.

Further possibilities for extension of the work of this thesis include many of the ideas presented as current topics of research in reinforcement learning as given in Section 2.3. In particular, those methods aimed at improving learning speed and efficiency appear attractive—for example: the various forms of hierarchal reinforcement learning, including the use of variable resolution (using finer discretisation step sizes only where higher precision is needed); and temporal abstraction (the use of "macro" actions). Fast Q-learning, which uses a lazy update rule, could prove useful.

Some of the ideas recognised in earlier chapters of this thesis as potentially useful but not investigated further (due to time constraints) include the following. The use of variable-depth eligibility traces may improve learning because different reward sources are attributable to different numbers of previous actions. For example, crashes are probably influenced by the actions chosen on the previous 100 or so time steps; whereas a lap time is equally attributable to all actions of the previous lap or more. The variable depth could be implemented as different values of the eligibility trace parameter, λ, and/or the trace could be cut at different lengths. Guided exploration involves trying out actions, or areas of the state space, that are likely to be profitable rather than simply making a random choice on exploratory moves. Some ideas for guided exploration in RARS are discussed in [Cleland, 2003].

The experiments in this thesis have modest success at using reinforcement learning to construct an agent that performs well in the Robot Auto Racing Simulator. The optimal or a near-optimal agent is not found. The experiments are successful in using minimal prior knowledge. There are many potential improvements of the agent, and what has been achieved in this thesis appears to only be a small part of what is possible.

References

Andre, D. (2003) *Programmable Reinforcement Learning Agents*. PhD thesis, U.C. Berkeley.

Andre, D. and Russell, S. (2001) *Programmable reinforcement learning agents*. Advances in Neural Information Processing Systems, volume 13.

Barreno, M and Liccardo, D. (2003) *Reinforcement Learning for RARS.* Technical Report, Department of EECS, University of California, Berkeley.

Barto, A.G. and Mahadevan, S. (2003) *Recent Advances in Hierarchical Reinforcement Learning.* Autonomous Learning Laboratory, Department of Computer Science, University of Massachusetts, Amherst MA.

Chapman, D. and Kaelbling, L.P. (1991) *Input Generalization in Delayed Reinforcement Learning: An Algorithm And Performance Comparisons.* In, Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Sydney, Australia.

Chrisman, L. (1992). *Reinforcement learning with perceptual aliasing: The perceptual distinctions approach.* Proceedings of the Tenth National Conference on Artificial Intelligence, Menlo Park, CA. AAAI Press/MIT Press. p183-188.

Cichosz, P. (1997) *Reinforcement Learning by Truncating Temporal Differences.* PhD, Warsaw University of Technology, Poland.

Cichosz, P. (1999) TD(λ) *Learning Without Eligibility Traces: A Theoretical Analysis.* Journal of Experimental & Theoretical Artificial Intelligence, 11(2). p239-263.

Cleland, B.G. (2003) *Application of Machine Learning to Real-Time Control of an Autonomy Operating within a Continuous System.* Honours dissertation. Department of Computer Science, University of Waikato.

Coulom, R. (2002) *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD Thesis. Institut National Polytechnique de Grenoble.

Dietterich, T.G. (2000) *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition.* Journal of Artificial Intelligence Research, 13. p227–303.

Gordon, G.J. (1995) *Stable function approximation in dynamic programming*. Technical Report CMU-CS-95-103, Carnegie Mellon University. A version also appeared in: Machine Learning (proceedings of the twelfth international conference), San Francisco, CA, 1995.

Gordon, G.J. (2001) *Reinforcement Learning with Function Approximation Converges to a Region.* Advances in Neural Information Processing Systems. The MIT Press.

Grounds, M. (2004) *Scaling-Up Reinforcement Learning*. Qualifying Dissertation, Department of Computer Science, University of York.

Hendler, J.A. (2000) *Probing the Pachyderm: A Plea for Proaction*. IEEE Intelligent Systems, volume 15, number 6, pages 40-41.

Hochreiter, S. and Schmidhuber, J. (1997). *Long short-term memory.* Neural Computation, 9(8). p1735-1780.

Isbell, C.L., Shelton, C.R., Kearns, M., Singh, S. and Stone P. (2001) *A Social Reinforcement Learning Agent*, International Conference on Autonomous Agents 2001, Montreal, Canada, p377-384. ACM.

James, M.R., Wolfe, B. and Singh, S. (2005) *Combining Memory and Landmarks with Predictive State Representations*. International Joint Conference on Artificial Intelligence (IJCAI), 2005.

Lin, L.J. (1992) *Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching.* Machine Learning, 8(3/4), p293-321.

Lin, L.J. (1993) *Reinforcement Learning for Robots Using Neural Networks*. Phd, Carnegie Mellon University. Also appears as CMU Technical Report CMU-CS-93-103.

Littman, M. L., Cassandra, A. R. and Kaelbling, L. P. (1995). *Learning policies for partially observable environments: Scaling up.* Proceedings of the Twelfth International Conference on Machine Learning, San Francisco. p362-370.

Littman, M.L., Sutton, R.S. and Singh, S. (2002) *Predictive representations of state.* Advances in Neural Information Processing Systems, 14. MIT Press, 2002.

Mataric, M.J. (1994) *Reward Functions For Accelerated Learning.* Machine Learning: Proceedings of the Eleventh International Conference. p181-189.

McCallum, A.K. (1993). *Overcoming incomplete perception with utile distinction memory.* Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann. p190-196.

McCallum, A.K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester.

Mitchell, M.W. (2003) *Using Markov-k Memory for Problems with Hidden-state.* School of Computer Science and Software Engineering, Monash University. In Machine Learning: Models, Technologies and Applications.

Moore, A.W. and Atkeson, C.G. (1995) *The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces*. Machine Learning, 21(3). p199–233.

Munos, R. and Moore, A.W. (2002) *Variable Resolution Discretization in Optimal Control.* Machine Learning, 49(2–3) p291–323.

Natarajan, S. and Tadepalli, P. (2005) *Dynamic Preferences in Multi-Criteria Reinforcement Learning.* International Conference on Machine Learning, Bonn, Germany, 2005.

Parr, R. and Russell, S. (1995). *Approximating optimal policies for partially observable stochastic domains.* Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, 1995.

Parr, R. and Russell, S. (1997) *Reinforcement Learning with Hierarchies of Machines*. Advances in Neural Information Processing Systems, volume 10. The MIT Press.

Perkins, T.J. and Precup, D. (2002) *A convergent form of approximate policy iteration*. Advances in Neural Information Processing Systems 13. MIT Press.

Pyeatt, L.D., Howe, A.E. and Anderson, C.W. (1996) *Learning Coordinated Behaviors for Control of a Simulated Race Car.* Technical report, Computer Science Department, Colorado State University, Fort Collins.

Pyeatt, L.D. and Howe, A.E. (1998a) *Learning to Race: Experiments with a Simulated Race Car.* Eleventh International Florida Artificial Intelligence Research Society Conference. p357–361.

Pyeatt, L.D. and Howe, A.E. (1998b) *Reinforcement Learning for Coordinated Reactive Control.* Fourth World Congress on Expert Systems, 1998.

Pyeatt, L.D. and Howe, A.E. (1998c) *Decision Tree Function Approximation in Reinforcement Learning.* Technical Report TR CS-98-112, Computer Science Department, Colorado State University, Fort Collins. Version also in: Third International Symposium on Adaptive Systems: Evolutionary Computation & Probabilistic Graphical Models, Havana, Cuba, 2001. p70-77.

Rafols, E.J., Ring, M.B., Sutton, R.S. and Tanner, B. (2005) *Using Predictive Representations to Improve Generalization in Reinforcement Learning.* Nineteenth International Joint Conference on Artificial Intelligence (IJCAI), 2005.

Reynolds, S.I. (2001) *Optimistic Initial Q-values and the max Operator*. Proceedings of the UK Workshop on Computational Intelligence, Edinburgh, UK.

Reynolds, S.I. (2002a) *The stability of general discounted reinforcement learning with linear function approximation*. In Proceedings of the UK Workshop on Computational Intelligence (UKCI-02), pages 139–146.

Reynolds, S.I. (2002b) *Experience Stack Reinforcement Learning for Off-Policy Control,* Technical Report. School of Computer Science, The University of Birmingham, UK.

RL3. Rutgers Laboratory For Real-Life Reinforcement Learning http://www.cs.rutgers.edu/rl3/index.html (As at February 2006).

Russell, S. and Norvig, P. (2003) *Artificial Intelligence a Modern Approach.* Second Edition. Prentice Hall, New Jersey.

Shelton, C.R. (2001) *Importance Sampling for Reinforcement Learning with Multiple Objectives*. Phd thesis, MIT.

Şimşek, Ö. and Barto, A.G. (2004) *Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning.* 21st International Conference on Machine Learning, 2004.

Smart, W. D. and Kaelbling, L.P. (2000) *Practical Reinforcement Learning in Continuous Spaces.* International Conference on Machine Learning, 2000.

Sutton, R.S. (1988) *Learning to Predict by Method of Temporal Differences*. Machine learning, 3(1), p9-44.

Sutton, R.S. (1990) *Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming*. Seventh International Conference on Machine Learning, Morgan Kaufmann. p216-224.

Sutton, R.S. (1991) *Dyna, an Integrated Architecture for Learning, Planning, and Reacting*. SIGART Bulletin, 2, p160-163.

Sutton, R.S. and Barto, A.G. (1998) *Reinforcement Learning*. Bradford Book, The MIT Press, Massachusetts.

Sutton, R.S. and Singh, S.P. (1994) *On Step-Size and Bias in Temporal-Difference Learning.* Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems. p91-96.

Sutton, R.S. and Tanner, B. (2005) *Temporal-difference networks*. Advances in Neural Information Processing Systems 17, MIT Press. p1377–1384.

Sutton, R.S., Precup, D. and Singh, S.P. (1999) *Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning*. Artificial Intelligence, 112(1–2), p181–211.

Szepesvari, C. and Smart, W.D. (2004) *Interpolation-based Q-learning*. International Conference on Machine Learning (ICML'04).

Tanner, B. and Sutton, R.S. (2005) *Temporal-Difference Networks with History.* IJCAI-05, p865.

Tesauro, G.J. (1994) *TD-Gammon, a self-teaching backgammon program, achieves master-level play.* Neural Computation. 6(2):215-219.

Tesauro, G.J. (1995) *Temporal difference learning and TD-Gammon.* Communications of the ACM. 38:58-68.

Thrun S. and Littman M.L. (2000) *A Review of Reinforcement Learning*. AI magazine, American Association for Artificial Intelligence, p 103-105.

Thrun, S. and Schwartz, A. (1993) *Issues in Using Function Approximation for Reinforcement Learning*. Proceedings of the Fourth Connectionist Models Summer School. Lawrence Erlbaum, Hillsdale, NJ.

Thrun, S.B. and Schwartz, A. (1994) *Finding Structure in Reinforcement Learning.* Advances in Neural Information Processing Systems, volume 7. p385–392. The MIT Press.

Timin, M. *Robot Auto Racing Simulator*. Available at http://rars.sourceforge.net/ (February 2006)

Tsitsiklis, J.N. and Van Roy, B. (1996) *An Analysis of Temporal Difference Learning with Function Approximation.* Technical Report LIDS-P-2322, MIT.

Uther, W.T.B. and Veloso, M.M. (1998) *Tree based discretization for continuous state space reinforcement learning.* In, Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI.

Watkins, C.J.C.H. (1989) *Learning from Delayed Rewards.* PhD Thesis. King's College, Cambridge.

Wiering, M. and Schmidhuber, J. (1998a) *Fast Online Q(λ)*. Machine Learning, 33(1). p105-115.

Wiering, M. and Schmidhuber, J. (1998b) *Speeding Up Q(λ)-Learning.* Tenth European Conference on Machine Learning (ECML'98). p352-363.

Wiering, M.A. (2004) *Convergence and Divergence in Standard and Averaging Reinforcement Learning.* 15th European Conference On Machine Learning, 2004, p 477-488.

Wikipedia, http://en.wikipedia.org/wiki/Reinforcement_learning (As at 19 Janurary 2006)

Witten, I.H. and Frank, E. (2000) *Data Mining, Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann Publishers, San Francisco.

Appendices

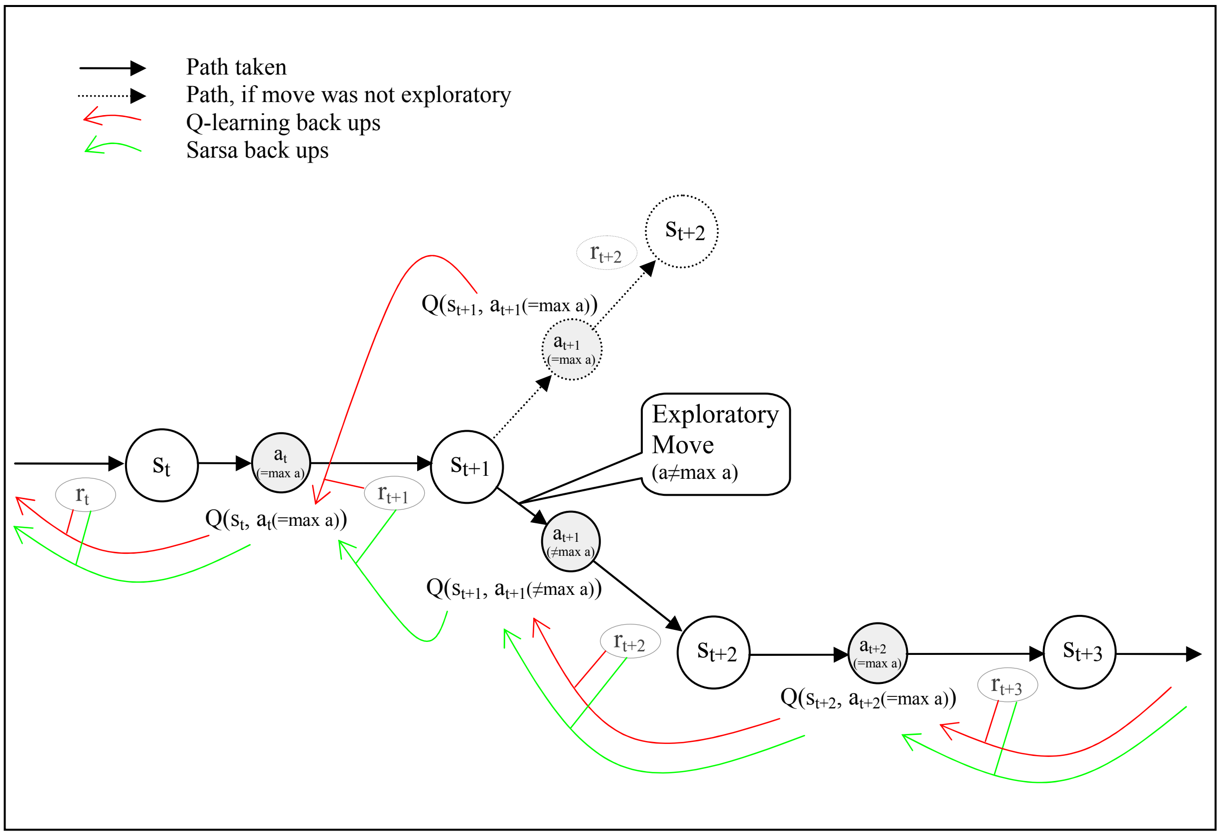
A. Comparison of Q-learning and Sarsa

The backup diagram below is used in the discussion in Section 3.4 and Section 3.9.5.

Back-up formulae:

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

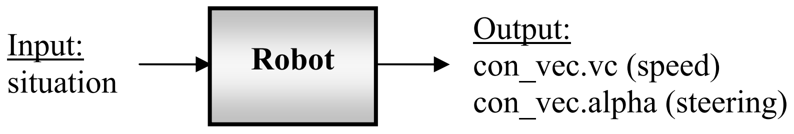
Sarsa: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$



B. Inputs and Outputs of a RARS Robot Driver.

The following explanations are adapted from the C++ code of RARS, particularly from the file "car.h", [Timin].

A robot driver is a function that takes a "situation" reference and returns a "con_vec":



Inputs:

This is the situation structure passed to the robot driver by the main RARS code. These are also referred to as the "environmental parameters":

struct situation	- a car's local situation as seen by the driver
{	
cur_rad	- radius of inner track wall, 0 = straight, minus = right
cur_len	- length of current track segment (angle if curve)
to_lft	- distance to left wall
to_rgt	- distance to right wall
to_end	- distance to end of current track seg. (angle or feet)
v	- the speed of the car, feet per second
vn	- component of v perpendicular to track direction
nex_len	- length of the next track segment (angle if curve)
nex_rad	- radius of inner wall of next segment (or 0)
after_rad	- radius of the segment after that one. (or 0)
after_len	- length of the segment after that one.
aftaft_rad	- radius of segment after that one. (or 0)
aftaft_len	- length of segment after that one.
cen_a, tan_a	- centripetal, tangential acceleration
alpha, vc	- wheel angle of attack and wheel command velocity
power_req	- ratio: power requested by driver to maximum power
power	- power delivered, divided by PwrMax
fuel	- lbs. of fuel remaining
fuel_mileage	- miles per lb.
time_count	- elapsed time since start of race
start_time	- value of time_count when first lap begins
bestlap_speed	- speed of cars best lap in that race fps
lastlap_speed	- speed of last lap, fps
lap_time	- time in seconds to complete most recent lap
distance	- distance travelled from start of first segment
behind_leader	- seconds behind leader on last SF crossing
dead_ahead	- set when there is a car dead ahead, else 0
damage	- accumulated damage units (out of race 30000)
stage	- What's happening, what stage of the competition?
my_ID	- to tell which car object you are driving
seg_ID	- current segment number, 0 to NSEG-1
laps_to_go	- laps remaining to be completed

laps_done	- laps completed
out_pits	- 1 if coming out from pits after stop - adjust your speed
go_pits	- becomes 1 when main program takes over for pitting
position	- 0 means leading, 1 means 2nd place, etc.
started	- position on the starting grid.
lap_flag	- changes from 0 to 1 on each crossing of finish line
backward	- set if cars motion is opposed to track direction
starting	- if not zero, robot knows to initialize data
side_vision	- allow cars alongside in s.nearby data
rel_state* nearby	- relative states of three cars in front of you (definition is below)
void* data_ptr	- pointer to driver's scratchpad RAM area

}

This is the definition of "rel_state", which is used in the "situation" structure above. This defines the relative position and velocity vectors of a car in front. These are with respect to a non-rotating frame of reference centred on the car behind. The y axis is in the direction the car is pointing.

struct rel_state - relative states of cars in front of you
{

rel_x	- how far to your right is the car ahead?
rel_y	- how far in front of you?
rel_xdot	- how fast cutting across from left-to-right?
rel_ydot	- how fast is he getting away from you?
Alpha	- car's current alpha
to_lft	- car's s.to_lft position
to_rgt	- car's s.to_rgt position
v	- car's velocity
vn	- car's velocity wrt track
dist	- distance ahead on track
who	- an identifier for that car.
Braking	- car is braking
for_position	- pass is for position
coming_from_pits	- flagged if car is just about to drive out

}

Outputs:

Control vector: steering & throttle, and pit-stop orders.

This is the structure returned by each robot driver. Alpha (steering) and vc (throttle/brake) are the most commonly used.

struct con_vec - the driver's requests to the environment
{

alpha	- wheel angle of attack (steering angle)
vc	- wheel command velocity (translates to throttle/brake)
fuel_amount	- how much fuel to add to the tank during next pitting
request_pit	- set to 1 to enter pits
repair_amount	- repair to do during next pitting

}