Fuzzy and tile coding approximation techniques for coevolution in reinforcement learning

Tokarchuk, Laurissa Nadia

Fuzzy and Tile Coding Approximation Techniques for Coevolution in Reinforcement Learning

Laurissa Nadia Tokarchuk

Submitted for the degree of Doctor of Philosophy

Department of Electronic Engineering

Queen Mary, University of London

December 2005

To Matthew

Abstract

This thesis investigates reinforcement learning algorithms suitable for learning in large state space problems and coevolution. In order to learn in large state spaces, the state space must be collapsed to a computationally feasible size and then generalised about. This thesis presents two new implementations of the classic temporal difference (TD) reinforcement learning algorithm *Sarsa* that utilise fuzzy logic principles for approximation, FQ Sarsa and Fuzzy Sarsa. The effectiveness of these two fuzzy reinforcement learning algorithms is investigated in the context of an agent marketplace. It presents a practical investigation into the design of fuzzy membership functions and tile coding schemas. A critical analysis of the fuzzy algorithms to a related technique in function approximation, a coarse coding approach called tile coding is given in the context of three different simulation environments; the mountain-car problem, a predator/prey gridworld and an agent marketplace. A further comparison between Fuzzy Sarsa and tile coding in the context of the non-stationary environments of the agent marketplace and predator/prey gridworld is presented.

This thesis shows that the Fuzzy Sarsa algorithm achieves a significant reduction of state space over traditional Sarsa, without loss of the finer detail that the FQ Sarsa algorithm experiences. It also shows that Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding learn similar levels of distinction against a stationary strategy. Finally, this thesis demonstrates that Fuzzy Sarsa performs better in a competitive multiagent domain than the tile coding solution.

Acknowledgements

I would like to thank Professor Laurie Cuthbert, Dr John Bigham and Dr Chris Phillips for the advice and guidance they have given me throughout my PhD at Queen Mary, University of London.

I would like to thank the staff of the Electronic Engineering department and especially Ho and Chris for the extra simulation support.

I would also like to thank my furry non-human friends, Nephis and Kila, for their unconditional love throughout all of my studies.

I would like to thank my friends and family for the love, guidance and encouragement that they have always provided. Finally, I especially want to thank my parents - it is your love and support that has made me the person I am today!

Table of Contents

List of Figures

1 Introduction

1.1 Research Motivation

While there are many different ways of dealing with reinforcement learning in large state spaces, function approximation promises to be one of the more powerful solutions: this is because all function approximation techniques deal with generalisation. They attempt to generalise from the information learnt in one state to determine a course of action to take in a newly visited state. This research looks at two different techniques for function approximation to determine their ability to represent and generalise, not just in large state space examples, but also in smaller dynamic state spaces.

One of the motivating factors for examining the generalisation capabilities of function approximation stems from recent interest in agent coevolution. Coevolution is a technique in which a learner or agent evolves in response to both the stationary and non-stationary elements in their environment. A stationary element is an element which stays the same over time, whereas a non-stationary element is one that changes. An example of a non-stationary element is another learning agent interacting with the same environment as the first learning agent.

Some of the work that has been done to address this problem is to use agents that model the other agents; between agents that directly compete in a marketplace [VIDD97], agents that compete in a game situation [RV00], or between agents that collaborate to achieve a specific goal [NG97]. Vidal and

Durfee [VIDD97] provide a framework for agents who learn about other agents in terms of agents who directly compete for the same product. Most modelling has typically been done with relatively simple learning algorithms.

Another way to address this problem is through the learning algorithm. This approach uses more complex learning algorithms that use minimax [LS96], Nash equilibriums [HW03], or hill climbing techniques [BV02]. These techniques are difficult to scale to large state/action spaces.

Learning algorithms capable of dealing with large state spaces, while retaining enough flexibility to cope with changing environments, are particularly relevant for real world applications. One potential application area is flexible resource management for telecommunication networks. For example, in third generation mobile systems (3G), the use of higher bandwidth services in a mobile environment has led to increased complexity in resource control and resource management because of the variable bandwidth requirements of the applications, the new radio architecture and the varying demands on the fixed part of the infrastructure. Management of resources is, therefore, one of the many interesting applications of agent technology in 3G mobile networks [BTetal00 and TAetal01]. A flexible learning algorithm capable of dealing with non-stationary problems would be beneficial in many areas of this domain, such as bandwidth brokering, power management, routing and even fraud detection.

This thesis investigates the potential benefits and capabilities of using function approximation in conjunction with reinforcement learning in competitive and dynamic environments.

1.2 Research Scope

This research looks at two different techniques of function approximation used with reinforcement learning algorithms in order to investigate their abilities of state space representation and generalisation capabilities. The first function approximation technique investigated uses fuzzy set theory. This technique is then compared with the coarse coding approach of tile coding. Finally, these two types of function approximation are investigated in terms of their generalisation capabilities when the learning problem includes information about other agents in the environment, in other words they coevolve.

To properly investigate these issues three different simulation environments were created in order to enable a thorough comparison of the different algorithms under different environment dynamics. These three environments are:

- *The mountain-car problem.* The mountain-car world serves as a benchmark environment. The world dynamics and the gradient descent Sarsa(λ) with tile coding algorithm implemented is a Java conversion of the C++ implementation provided by Richard Sutton [SuttonMC].

- *Predator/prey gridworld.* The predator/prey gridworld was chosen because of its similarity in basic dynamics to many classic reinforcement learning examples such as those given by Stone and Veloso [SV00] and because it is the base domain for more complicated applications such as robotic soccer [ST00 and SSK05].

- *Agent marketplace*. The agent marketplace was chosen in order to enable comparison between the coevolutionary modelling using function approximation and other techniques such as those presented by Vidal [VID98] and Hu [HW98].

1.3 Summary of Contribution

The primary aim of this research is to investigate methods of coevolution in a learning environment. To this end, the following novel contributions have been made:

- Two fuzzy reinforcement learning algorithms:
  - ✓ FQ Sarsa – a fuzzy learning accelerator for Sarsa learning; and
  - ✓ Fuzzy Sarsa – a "fuzzification" of Sarsa following Bonarini's guidelines [BON98].
- A detailed investigation of the above fuzzy techniques and a comparison of those methods to a related linear approximation method called tile coding in three separate environments were given. These investigations covered both stationary and non-stationary problems. These investigations showed:
  - ✓ In a stationary environment, both Fuzzy Sarsa and the tile coding technique perform similarly.
  - ✓ In the non-stationary environment Fuzzy Sarsa has better performance than tile coding in same goal scenarios. Furthermore, these investigations showed that Fuzzy Sarsa was robust with regards to variation in parameter settings and, also with regards to membership function design error.

1.4 Outline of Thesis

This research first presents the background theory of reinforcement learning and reviews some relevant techniques in function approximation identifying fuzzy set theory as a promising method (Section 2). It then presents the three simulation environments used in this research (Section 3) before further investigating fuzzy reinforcement learning. After modifying an existing fuzzy algorithm to deal with on-policy learning, it presents the results of using this algorithm in the agent marketplace and also presents a study into parameterisation in this environment (Section 4). These results are then compared with a related technique of tile coding in all three simulation environments. It also details the experiences in constructing fuzzy membership functions and the overlaying of tiles in tile coding (Section 5). It then presents a critical analysis Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding in a coevolutionary scenario is presented (Section 6). Finally the results of this research are summarised and several areas for future investigation are highlighted (Section 7).

2 Reinforcement Learning

The majority of simple decision-making functions utilised by agents are characterised in terms of some sort of method for the maximisation of expected utility. For an agent system, Russell and Norvig [RN95] provide methods through which the agent can calculate the *expected utility* given that it performs some action A. Any decision an agent makes is based on one or more variables. These variables exert different levels of influence on the decision point. For example, every day we are faced with simple, seemingly straightforward, decisions that when examined contain many variables. For example, the decision to eat lunch is intuitively based on whether we are hungry. However, this decision may also be based on whether there is any food available, or perhaps on whether we have a meeting within the next hour. Making a decision is one of an agents most important functions. Without the capability to make a decision, an agent is helpless to act in its environment. In addition, there are many other aspects about agents that are also important. A discussion of some of these areas is provided by [JSW98].

[RN95] provide a general definition of a learning agent. Accordingly, a general learning agent is composed of the four different elements:

Learning Element – This element is responsible for making improvements, such as improved strategies, to the agent.

Performance Element – This element selects external actions in accordance to the newly learnt improvements.

Critic – This is some sort of internal mechanism that the agent uses to measure how well it is doing.

Problem Generator – This is a generator that suggests actions to the agent that will lead to new and informative situations.

Due the complex nature of multiagent systems (MAS), it is only natural that the architects of such systems utilise machine learning (ML) techniques. Typically machine learning is used in MAS to provide agents with adaptively. There has been a significant amount of work done in the application of ML techniques to MAS. [G96, SV00 and TR96] provide a good review of the types of techniques that have been applied in this area. They adopt Parunak's taxonomy for MAS [P96], dividing MAS by the following three characteristics:

- System function
- Agent architecture (level of heterogeneity, reactive vs. deliberative)
- System architecture (communication, protocols and human involvement)

ML techniques are then divided into the type of MAS they have been used in. The MAS community is not the only one to find added benefit in ML.

One area that is particularly relevant to MAS is that of learning *moving target functions*. In any kind of dynamic environment, the assumption that an entity will not change its behaviour cannot be made. It is probable that agents, like people, will change their strategies if they observe that their current strategy no longer meets their needs. As a result of this, the type of learning an agent employs needs to be flexible enough to cope with changes. Just because an agent has learnt the behaviour at time t, does not imply that that behaviour will be the same at time t+1.

Reinforcement learning is a term that is attached to a family of *unsupervised learning* algorithms. Unsupervised learning, rather than supervised learning, is a type of learning that does not rely on the existence of an external supervisor. In supervised learning, the learner uses a pre-existing data set provided from the external supervisor for training. This means that supervised learning is not adequate for the learning of interactive data because the nature and multitude of possibilities is very large. This is significant because this makes it very difficult for the training data to be both accurate and representative of all necessary situations.

Unsupervised algorithms force the learner to try to learn from its experiences. These algorithms build a mapping of situations onto actions. In other words, a reinforcement learning algorithm observes the current state of its world, and learns the best possible action from that state. The learner is never told what actions to take but rather what results are desired. After that, it is up to the learner how they achieve the result. Reinforcement learners have 4 main elements: *a policy, a reward function, a value function* and optionally, *a model of the environment.*

In order to understand how reinforcement learning works, imagine a simple world where some agent, a stickman learner, walks down various different paths (Figure 2.1).

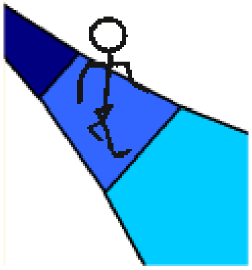


Figure 2.1: Stickman World

The stickman learner continues walking until it either reaches the goal, say a basket of oranges, or, it reaches a terminal penalty state, say getting crushed by a giant box.

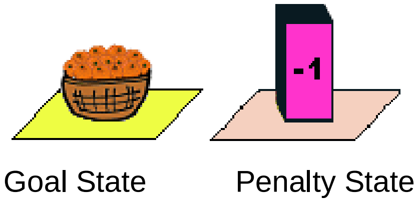
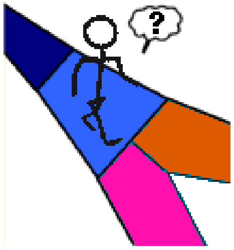


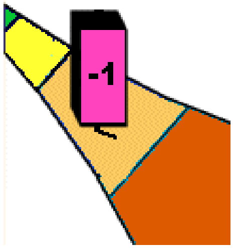
Figure 2.2: Stickman terminal reward and penalty states

With reference to this world, the three required elements of a reinforcement learner can be described as:

Policy: The policy defines how the learner reacts to the environment. A learner’s policy aids them in making decisions regarding their actions. For example if the learner is faced with choosing between two paths, the learner’s current policy will help it decide which path to take.



Reward Function: This function defines what is good and what is bad. The reward function provides immediate feedback in the form of a numerical value to the learner. In the stickman world, if the end result of some action is bad, it is indicated to the learner by a reward of -1 Conversely if the resulting state of the action is good, such as reaching the goal, a positive reward would be given.



Value Function: The value function defines what is good in the long run. In contrast to the reward, which is immediate depending on the learner’s current state, a value function describes all times the learner has been in a similar type of situation.

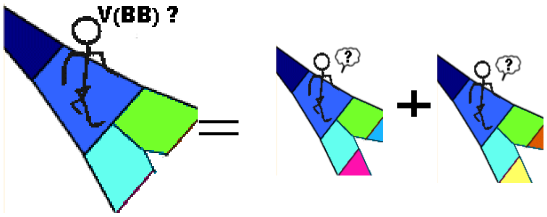


Figure 2.3: Stickman learning example

2.1 The Basics

This section discusses some of the basic techniques used in reinforcement learning. More detailed reviews are available in [SB98] and [KLM96].

2.1.1 Value Functions

In reinforcement learning, a state consists of a set of discrete values representing the current state of the world. As discussed above, reinforcement learners ultimately learn based on some sort of reward signal. This reward signal r directly influences the value of being in a particular state. $Q_t(a)$ indicates the value of taking action a at time t. Typically $Q_t(a)$ is calculated by averaging the observed rewards :

$$Q_t(a) = \frac{\left(r_1 + r_2 + \ldots r_{k_a}\right)}{(k_a)} \tag{2.1}$$

where k_a is the number of times action a has been chosen. Most reinforcement learning algorithms follow an incremental version of this update which requires less memory:

$$V(s) = V(s) + \alpha\left[V(s') - V(s)\right] \tag{2.2}$$

where $V(s)$ is the value of being in the original state s and $V(s')$ is the new value being in the next state s'. α is a step-sized parameter or the *learning rate*.

2.1.2 Policy Selection

Since the learner is only told what the desired result of learning is, they must attempt to balance two types of actions: *exploratory* actions and *exploitive* actions. An exploratory action is an action that the learner takes in order to discover the value of a potentially new solution, whereas an exploitive action is an action which makes use of the learner's best known available solution.

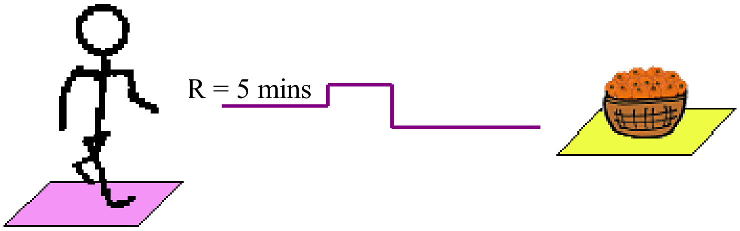


Figure 2.4: A path to the oranges in a stickman world

For example, in Figure 2.4 the learner knows that there is a route to the oranges that takes 5 minutes through the purple path. It knows this information because it has gone that way before. However, since the learner has never tried the other paths, it does not know the existence or value (time taken) of any other path. If the learner always chooses the exploiting action, once a positive reward is discovered, the learner will never discover any other path.

For stickman there are other paths that can be discovered by making exploratory moves. As depicted in Figure 2.5, there are actually three separate paths to the oranges; the purple path, a shorter green path and a longer blue path.

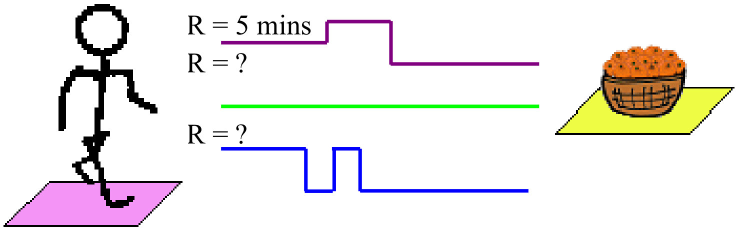


Figure 2.5: Stickman choices: Explore vs. Exploit

If the stickman chooses to explore, he will discover the value for other paths. If the value of the green path is 2 minutes, the next time the stickman exploits, he will choose the green path. Conversely, if the learner explores too much, it will revisit paths it already knows are bad and thus not benefit from the knowledge of which path is shortest.

Reinforcement learning algorithms address the issue of exploitation vs. exploration through implementation of a policy. The learners' current policy or *action selection* algorithm determines the action the learner takes at any given time of the learning process. There are several different types of policy selection methods, the more popular being *ε greedy* and *softmax* action selection. The ε greedy selection policy operates by choosing the most optimal action based on the current known rewards or Q-values for all possible actions. This means that the learner chooses which action to take based on maximising its reward. For every selection there is some probability ε that rather than choosing the optimal greedy action, the algorithm will choose randomly to explore other actions in the hope that they may lead to a better solution.

ε - Randomly explore a different action.

(1-ε) – Make the greedy choice.

The two main issues behind all action selection policies are firstly, when to explore and when to exploit, and secondly, how to select which action is chosen when exploring. Unlike ε greedy, action selection policies such as softmax concentrate on trying to choose actions in the exploratory phase that are more likely to lead to a positive outcome. This type of selection policy is particularly important in situations where bad actions are very bad.

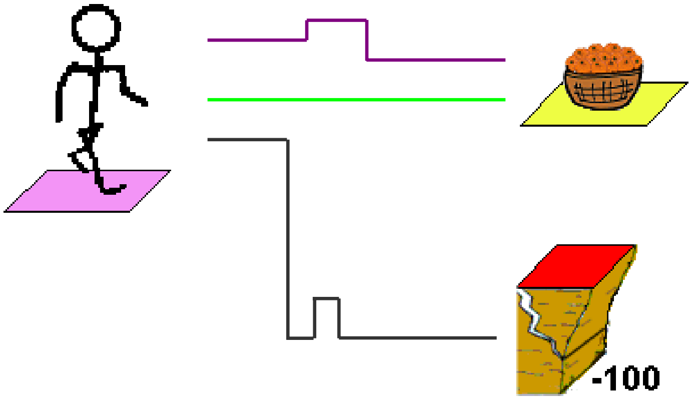


Figure 2.6: Stickman and the cliff

For example, as shown in Figure 2.6, imagine that one of the paths the stickman could follow leads to a cliff. In this case, choosing a potentially bad action is very bad indeed, as if the learner makes a bad decision, they fall off the cliff and die. If the learner was using ε greedy for policy selection, and was faced with making an exploratory move, it would fall off the cliff just as often as take any other action. If the learner was using softmax policy selection, it would attempt to minimize the exploratory choice of really bad actions such as the black path. Actions are weighted according to their value estimates. Selection typically uses distributions such as Gibbs or Bolzmann distribution and τ (a positive temperature parameter) to weight the estimated available action values. A summary of action selection is given by [SB98].

Balancing exploitation and exploration is an important area of research. More complex action selection methods have been investigated, such as methods that track the number of times an action has been selected [BON96a], and other algorithmic methods [SAH94]. An overview of the more complex types of action selection mechanisms, along with a behaviour based proposal is given by [HUM96]. However, many researchers find that ε greedy or softmax provide adequate action selection in their domain (such as [OFJ99], [KLM96] and [SB98]). Since action selection is not the focus of this research, further investigation is not pursued here.

2.1.3 Off-policy versus On-policy

There are two main styles of learning within reinforcement algorithms; *off-policy* and *on-policy*. Off-policy describes learners that learn about behaviours or policies other than the one currently being executed. An off-policy learner updates its value function by choosing an action according to the current policy. It judges the value of the current state based on the best possible value of all state/action combinations of the next state *irrespective* of the actual action taken. Therefore, the learner learns the best policy regardless of the policy actually being followed. In contrast, an on-policy learner learns only from actions that it actually takes during the episode.

To illustrate the difference, imagine the stickman arriving in the bright blue BB square. An off-policy learner will learn the value of being in the bright blue square $V(BB)$, based on the value of the light blue (LB) square ($V(LB)=0.5$) regardless of what action it next takes. So the update for the off-policy learner is $V(BB)=V(BB)+\alpha[V(LB)-V(BB)]$. This update is fixed regardless of whether the learner makes a greedy move to the BB square or an exploratory move to the light green LG square.

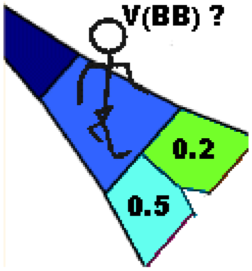


Figure 2.7: A Stickman faced with a decision

On the other hand, an on-policy learner learns *only based on the action* it takes. If the on-policy learner decides to take an exploratory action (a move to the green square), rather than an exploiting action (a move to the light blue square), it will learn based on the value of the light green square $V(LG)=0.2$ and thus its update is $V(BB)=V(BB)+\alpha[V(LG)-V(BB)]$. However, if it makes a greedy move to the LB square its update would be $V(BB)=V(BB)+\alpha[V(LB)-V(BB)]$.

2.2 Dynamic Programming

Dynamic programming (DP), as introduced by Bellman [BELL57], is a family of algorithms that solve the learning problem in a specific way. The following section presents a brief review, primarily focused at DP for solving a markov decision process (MDP). An MDP is a reinforcement learning problem that satisfies the *markov property*. A state is said to have the markov property if it succeeds in retaining all relevant information about the previous states.

DP techniques are well proven. However, they suffer from several practical problems. While dynamic programming algorithms are capable of computing the optimal policy for a learner to follow, they also need a perfect model of the learners' environment in order to do so. This model consists of a set of transitional probabilities, which describe the probability of transition from one state to any other state, and a set of immediate rewards.

One method of calculating the optimal policy is through *iterative policy evaluation*. In policy evaluation, a policy π is chosen and then the value of every state in the environment is approximated using the Bellman equation for V^{π}:

$$V_{k+1}(s) = \sum_{a \in \mathrm{A}(s)} \pi(s,a) \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V_k(s') \right] \tag{2.3}$$

where $a \in \mathrm{A}(s)$ is the actions belonging to the set of available actions for the state s, $P_{ss'}^{a}$ is the probability of transitioning from s to s' when action a is selected, and $R_{ss'}^{a}$ the reward, for all $s \in S$. The value of each state, $V_{k+1}(s)$, is updated with the sum of the values from all possible successor states. Figure 2.8 depicts a simple 3x3 gridworld with one terminal state. This gridworld is a simplified version of the example presented by [SB98].

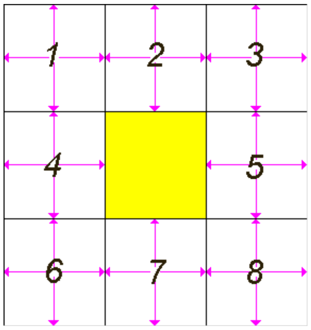


Figure 2.8: Gridworld 3x3 Example

The value of any state in the gridworld is calculated as the sum of all possible successor states based on the rules of the particular environment. In the case of the gridworld, movement rules are indicated as pink arrows on the grid [up, down, right, left]. If the allowed move is off the grid the agent's state remains unchanged. All states are non-terminal except for the central yellow state. Rewards are expressed as -1 on all transitions. In order to apply iterative policy evaluation, the probabilities $P_{ss'}^{a}$ for all $s, s' \in S$ for every action a are first calculated. In this example, these probabilities are simple to determine. All possible state transitions probabilities from square 1 can be expressed as:

$$P_{1,1}^{up} = 1\,,\quad P_{1,2}^{up} = 0\,,\; P_{1,3}^{up} = 0\,,\quad \ldots \quad ,\; P_{1,8}^{up} = 0$$

$$P_{1,1}^{right} = 0\,,\; P_{1,2}^{right} = 1\,,\; P_{1,3}^{right} = 0\,,\quad \ldots \quad ,\; P_{1,8}^{right} = 0$$

$$P_{1,1}^{down} = 0\,,\; P_{1,2}^{down} = 0\,,\quad \ldots \quad ,\; P_{1,4}^{down} = 1\,,\quad P_{1,5}^{down} = 0\,,\quad \ldots ,\; P_{1,8}^{down} = 0$$

$$P_{1,1}^{left} = 1\,,\quad P_{1,2}^{left} = 0\,,\; P_{1,3}^{left} = 0\,,\quad . \quad . \quad . \quad . \quad ,\; P_{1,8}^{left} = 0$$

All other non-terminal states have a similar list of probabilities. If the current policy of agent is that all actions are equiprobable, then the policy probabilities are:

$$\pi(s, up) = 0.25 \qquad \pi(s, right) = 0.25$$

$$\pi(s, down) = 0.25 \quad \pi(s, left) = 0.25$$

The calculation for the value of state 1, using a discounting rate γ = 1:

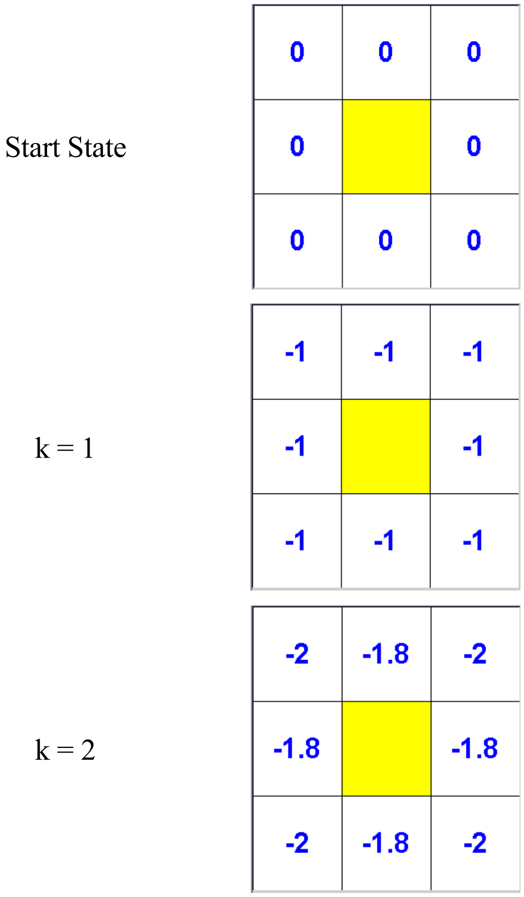
$$V_k(1) = \pi(1, up) * \left(P_{1,1}^{up}\left[R_{1,1}^{up} + \lambda V_k(1)\right] + P_{1,2}^{up}\left[R_{1,2}^{up} + \lambda V_k(2)\right] + \ldots + P_{1,8}^{up}\left[R_{1,8}^{up} + \lambda V_k(8)\right]\right)$$
$$+ \ \ldots \ + \pi(1, left) * \left(P_{1,1}^{left}\left[R_{1,1}^{left} + \lambda V_k(1)\right] + P_{1,2}^{left}\left[R_{1,2}^{left} + \lambda V_k(2)\right] + \ldots + P_{1,8}^{left}\left[R_{1,8}^{left} + \lambda V_k(8)\right]\right)$$

```
= 0.25(1[-1+1(0)] + 0[-1+1(0)] + … + 0[-1+1(0)]) +
  0.25(0[-1+1(0)] + 1[-1+1(0)] + … + 0[-1+1(0)]) +
  0.25(0[-1+1(0)] + 0[-1+1(0)] + … + 1[-1+1(0)]
                  + 0[-1+1(0)] + … + 0[-1+1(0)]) +
  0.25(1[-1+1(0)] + 0[-1+1(0)] + … + 0[-1+1(0)]) +

= 0.25(-1+0+0+0+0+0+0+0) + 0.25(0+-1+0+0+0+0+0+0) +
  0.25(0+0+0+-1+0+0+0+0) + 0.25(-1+0+0+0+0+0+0+0)

= -1
```

Figure 2.9 shows the all calculated $V(s)$ values for each grid location for *k*=0 to *k*=2.



Start State

0	0	0
0	🟨	0
0	0	0

k = 1

-1	-1	-1
-1	🟨	-1
-1	-1	-1

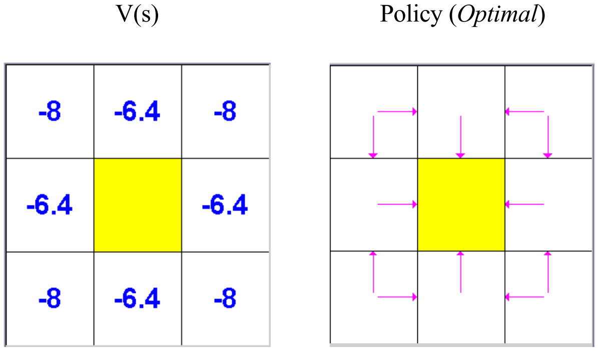
k = 2

-2	-1.8	-2
-1.8	🟨	-1.8
-2	-1.8	-2

Figure 2.9: Dynamic programming V(s) calculations for 3x3 gridworld.

The previous example is of policy evaluation. The value of each state is calculated based on the current policy. In the example, the policy being followed is equiprobable action selection, in other words, each possible movement from any state has an equiprobable chance of selection. The results displayed are a result of continually evaluating this policy. When examining the 3x3 gridworld, the move *up* from 1 is clearly not a desirable move, while the move *right* from 1 is desirable.

The primary reason for calculating $V(s)$ is to be able to shift the current policy towards the optimal policy. This procedure, called *policy improvement,* is typically performed by altering the policy to be greedy with respect to $V^{\pi}(s)$. The policy is shifted toward the greedy policy when the current policy is deemed to be stable, typically when $\max \Delta V(s) <$ some small number for $\forall s \in S$. For example, at $k = 14$ where $\max \Delta V(s) \geq 0.2$, changing the policy to be greedy with respect to $V^{\pi}(s)$ results in:



V(s)

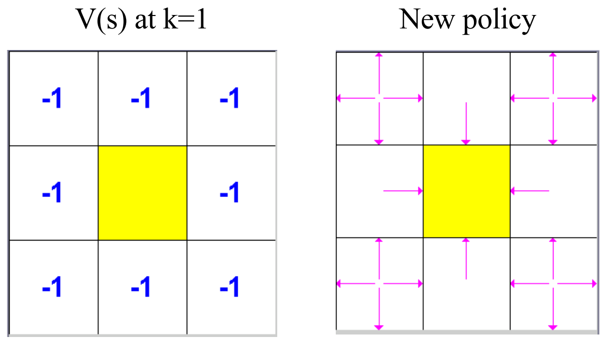
-8	-6.4	-8
-6.4		-6.4
-8	-6.4	-8

Policy (*Optimal*)

Figure 2.10: Shifting the policy π towards greedy at k = 14 where max ΔV(s) ≥ 0.2

In a problem of this size, policy iteration finds the optimal policy on the first sweep. However on larger problems with more complex transitional probabilities, several sweeps may be required before the optimal policy is found. In these more complex problems, this process continues until the policy is optimal (while $V^{\pi'}(s) \geq V^{\pi}(s)$). Policy evaluation can lead to drawn out iterative computations while waiting for $\max \Delta V(s)$ to decrease significantly. Furthermore in simple problems, the optimal policy is often found far earlier. In the 3x3 gridworld, the optimal policy is already found at $k = 2$, and thus waiting on $\max \Delta V(s)$ to be less than some small number is not very beneficial and could result in many unneeded iterations.

One method to decrease this calculation is that of *value iteration.* In value iteration, the policy is shifted towards the greedy policy at the end of each policy evaluation step. Rather than waiting until $\max \Delta V(s)$ is suitably small, the policy is changed immediately towards the greedy policy. At $k = 1$, the new policy in the gridworld becomes:



-1	-1	-1
-1		-1
-1	-1	-1

Figure 2.11: Shifting the policy π towards greedy at k = 2 for value iteration

This combination of policy evaluation and policy improvement is called *generalized policy iteration (GPI)*. This process describes the repetitive movement of the current policy towards the greedy policy. The majority of reinforcement learning methods, *including* DP, can be described in this manner.

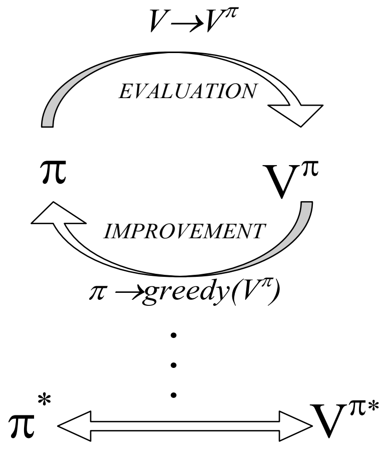


Figure 2.12: Generalized policy improvement.

The review presented here was based on a more detailed examination of DP for MDPs in [LCK95] and [SB98]. DP techniques require a full model of the environment, but since this is not feasible in most domains, DP is of primary interest as the theoretical basis of reinforcement learning.

2.3 Monte Carlo

Another approach to the reinforcement learning problem is a family of methods called Monte Carlo [RUB81]. Monte Carlo methods learn from sample sequences of their environment by averaging the complete returns of an episode.

Unlike dynamic programming methods of the previous section, Monte Carlo methods do not require a complete model of their environment. In order for the DP methods to work, a complete list of transitional probabilities would be required. Monte Carlo methods do not require these probabilities to be explicitly stated. Unlike DP, Monte Carlo methods do not bootstrap. This means that the value of an individual state does not rely on the values of any other states. To calculate the value of a state, sample episodes are generated. An episode is a set of state transitions from the start state to the terminal state. The value of each state along a single sample episode is the averaged return of all rewards received along that path.

This type of exploration based evaluation of the state space works extremely well in environments where the range of states required to solve the problem is actually a relatively small subset of the overall state space. Monte Carlo methods can be *focussed* to concentrate in these areas. One assumption that must be made in Monte Carlo applications is episode termination. This is because the averaged returns are not awarded to the state/action pairs until the end of the episode. For example, in the gridworld from Section 2.2 sample episodes would be generated starting from random locations within the gridworld. Each state/action pair appearing in the episode would be updated with the averaged return of the episode.

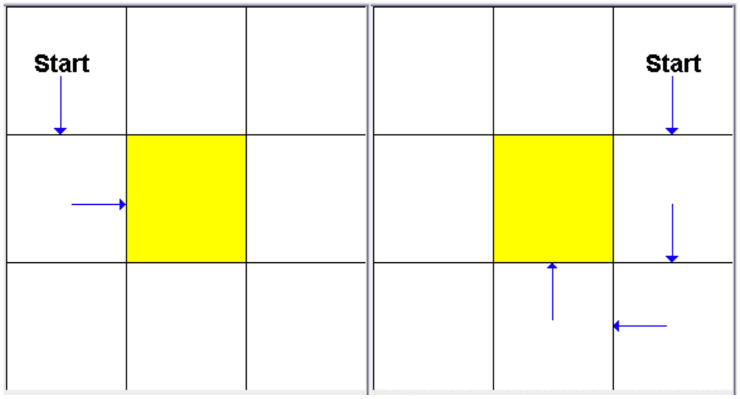


Figure 2.13: Two sample episodes in Monte Carlo evaluation.

This type of Monte Carlo evaluation is based on the assumption of *exploring starts;* the episode start is a randomly selected state-action pair, and every pair has a positive probability of being selected. This is necessary to ensure adequate exploration of the state space. In reality, this assumption is very restrictive. Many problems that have a specific start state. In order to get rid of this assumption, the principles of on-policy and off-policy as discussed in Section 2.1.3 can be applied. These principles and the implementation of some sort of *ε-soft* policy ensure adequate exploration of the state space without the exploring starts assumption. *ε-soft* policies are any type of policy that ensures that all actions have some positive probability of being selected. ε-greedy from Section 2.1.2 is an example of an ε-soft policy. Therefore, using Monte Carlo control and following generalised policy iteration, after each sample episode the policy is shifted towards the greedy policy and then the next episode is generated based on the new behavioural policy. A more detailed overview is provided in [SB98].

2.4 Temporal Difference Algorithms

Temporal difference (TD) algorithms combine the approaches of dynamic programming and Monte Carlo. They are similar to Monte Carlo approaches because they learn from sample sequences of their environment. Yet, unlike Monte Carlo methods, TD algorithms do not wait for the final outcome for learning to occur. Instead they learn from the partially learnt values of the next states they visit. The TD algorithms presented in this section are referred to as *tabular* learning, since they store their representation of the world discretely in a lookup table.

To elucidate this, consider a marketplace environment where a number of agents are participating in an auction, and where the goal for each agent is to purchase a number of items. Figure 2.14 illustrates the potential states for a marketplace agent. The agent has a look up table that contains combinations of discrete values which define its current state. These are the amount of money left, and the number of items still left to buy.

State	Money_Left	Items_to_Buy
S1	12	3
S2	5	1

Figure 2.14: State Representation

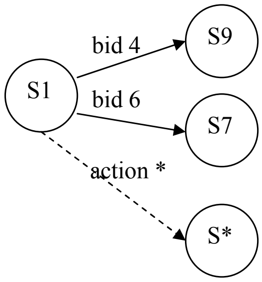


Figure 2.15: State action translation.

The current state of the agent's environment is represented by a particular state. The agent recognizes which state it is in (say state S1), and executes some action. This action causes a translation to another state. Figure 2.15 illustrates potential state translations. Tabular reinforcement learning algorithms such as Sarsa and *Q*-Learning attempt to learn the *Q*-value of a state-action pair-$Q(s,a)$. For the example state $S1$ in Figure 2.14, there would be several entries in the table corresponding to all the possible actions. If the available actions are bid 8, bid 6, and bid 4, the entries for $S1$ would become:

State		Action
Money_Left	Items_to_Buy	Bid
12	3	8
12	3	6
12	3	4

Figure 2.16: State action pairs

Finally as shown in Figure 2.16, the agent uses this table to store information about the value (*Q*-value) of each of these state/action combinations.

State(s)		Action	Q(s,a)
Money_Left	Items_to_Buy	Bid	
12	3	8	0.1
12	3	6	0.5
12	3	4	0.3

Figure 2.17: State action pairs with *Q*-values

The full lookup tables contain all possible state/action combinations.

2.4.1 Sarsa

Sarsa [SUTT96] is an on-policy TD learning algorithm originally called modified Q-Learning [RN94]. The general principle of Sarsa is summarized by its name: **S**tate, **A**ction, **R**eward, **S**tate, **A**ction. In Sarsa, an agent starts in a given state, from which it does some action. After the action, the agent receives a reward and has transitioned into a new state from which it can take another action.

Sarsa is an on-policy algorithm. This means that the learning occurs only from actual experience. An on-policy learner selects an action, receives a reward and observes the new state and again selects an action. As with all reinforcement style algorithms, there must be a trade off between exploration and exploitation. An exploratory action or exploiting action is chosen as a result of the current policy typically an action selection policy such as ε greedy. Recall that ε greedy policy selection operates on the simple guideline of choosing the most optimal action based on the current known rewards or Q-values for all possible state action pairs. At every time t, there is some probability ε that rather than choosing the optimal greedy action, the selection policy will choose randomly to explore from the set of possible state action pairs in the hope that they may lead to a better solution.

As discussed at the beginning of this section, after an agent has made an action from a state, the agent receives a reward. The learner typically receives a positive reward at the end of the episode if it has achieved its goal (i.e. it bought the number of required items) and a negative reward if it has not. At all other non-terminating state-actions, the agent receives the default reward.

The algorithm then proceeds as follows:

```
All Q(s,a) values are initialised.
Repeat for each episode (or auction game){
    Initialize $s_t$ (start state for the auction game).
    Choose $a_t$ from $s_t$ using $\varepsilon$ greedy selection policy.
    Repeat for each step(auction) in the episode(auction game){
        Take action $a_t$, observe $r$ and $s_{t+1}$
        Choose $a_{t+1}$ from $s_{t+1}$ using ε greedy selection policy
        $Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$
        $s_t = s_{t+1}$, $a_t = a_{t+1}$
    }
}
```

Figure 2.18: Sarsa Algorithm

2.4.2 Q-Learning

Watkins Q-Learning [WAT89] is a very similar algorithm to Sarsa. The primary difference between the two is that Q-Learning is an off-policy learning algorithm. In reference to the stickman world described previously, a Q-Learner learns based on the best state/action value at its new state. This action is not necessarily the action it takes.

Thus the update formula for the Q-Learner is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max\nolimits_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{2.4}$$

In a similar example to the cliff world in [SB98], imagine a gridworld where the agent starts at one end of the world and is required to find a path to a terminal point on the other side of the world. All non-terminal steps receive a -1 reward. There is a sink hole in the middle of this world, and if the learner falls in this hole, it receives a -100 reward and must start again. In this example the parameters γ and α are set as follows: γ=0.9 and α=0.1. To illustrate the differences between Sarsa and Q-Learning ε is initially set to 0. In this example, there is no difference between the two algorithms in the policies learnt. Setting ε to zero means that there is no exploration, both algorithms will always pick the greedy move. This results in both Sarsa and Q-Learning learning the policy illustrated by the black arrows of the Q-Learner of Figure 2.19.

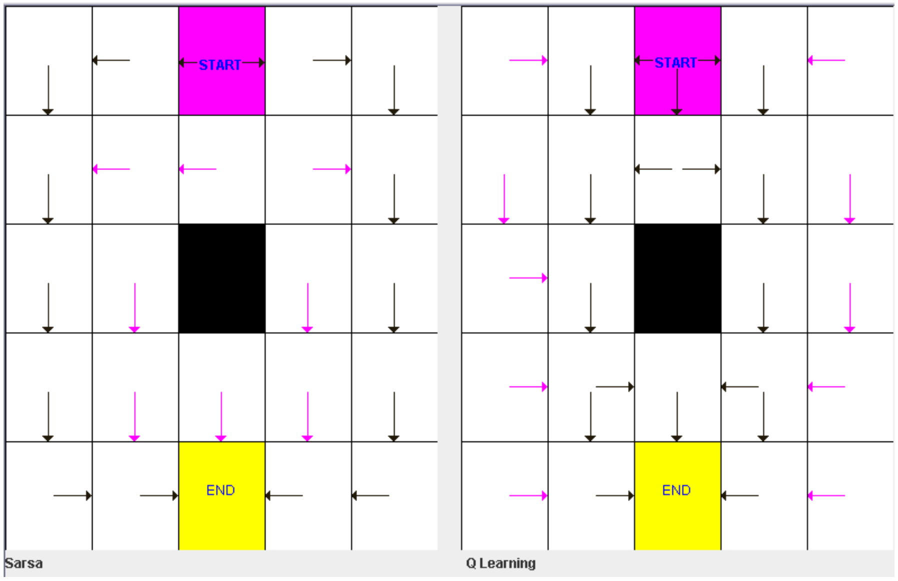


Figure 2.19: Sarsa and Q-Learning policies in a sink hole gridworld

If the ε is increased to 0.05, the policy learnt by Sarsa moves slightly away from the optimal policy. This is because Sarsa learns directly from the moves it actually makes. To illustrate how this affects the Sarsa learner, consider the episode highlighted by the blue arrows of Figure 2.20.

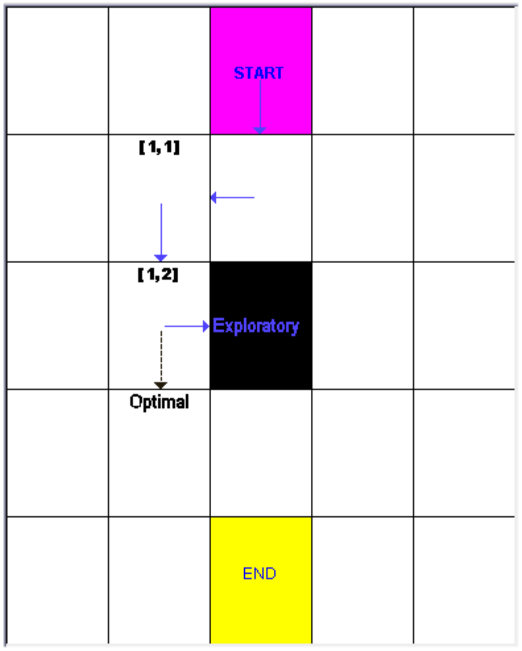


Figure 2.20: Exploratory vs. optimal action selection in a sink hole gridworld

At grid location [1,2], the current policy (ε-greedy) suggests an exploratory move which takes the learner into the sink hole rather than the optimal action indicated by the dashed arrow.

When the learner falls into the sink hole, each learner updates the Q(s,a) value of falling into the sink hole in the same manner:

$$Q(G[1,2],\ A_{EAST}) = Q(G[1,2],\ A_{EAST}) + 0.1(-100)$$

The difference comes from the update for the pairs before that final move. Sarsa updates its $\texttt{Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)}$ as:

$$\begin{aligned}\texttt{Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)} &= \texttt{Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)}\\ &\quad + \texttt{0.1(-1 + 0.9(}\boldsymbol{Q(G[1,2],A_{EAST})}\texttt{ - Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)}\end{aligned}$$

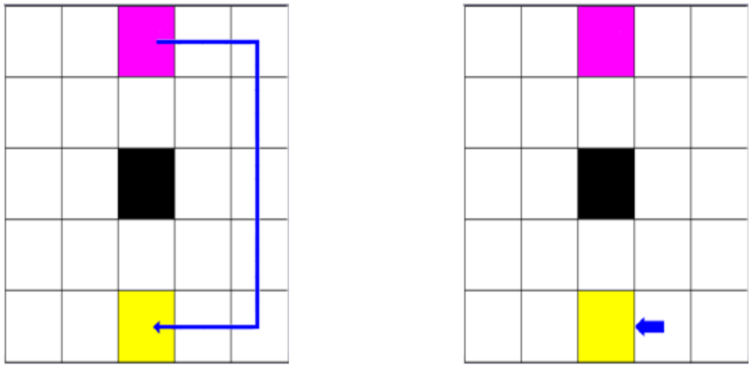
Where as Q-Learning updates its $\texttt{Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)}$ as:

$$\begin{aligned}\texttt{Q(G[1,1], A}_{\texttt{SOUTH}}\texttt{)} &= \texttt{Q(G[1,1],A}_{\texttt{SOUTH}}\texttt{)}\\ &\quad + \texttt{0.1(-1 + 0.9(}\boldsymbol{Q(G[1,2],A_{SOUTH})}\texttt{ - Q(G[1,1],A}_{\texttt{SOUTH}}\texttt{)}\end{aligned}$$

The Q-learner is learning the optimal policy even when it follows a different one, whereas the Sarsa learner learns from the policy it actually follows. Therefore the next time Sarsa will be less likely to even approach a state where one of the possible actions from that state is very bad. The end result of this difference in learning, is demonstrated by the difference in the policy learnt in Figure 2.19. The Sarsa learner learns the safe path around a really bad state.

2.5 Eligibility Traces

In the previous section, the algorithms discussed can be seen as one step algorithms. It is only the next step which influences the value of the current step. Furthermore the amount the next step influences the current step is dictated by γ. A γ of 1 indicates that the learning that takes place at the current state is heavily influenced by the next state, whereas lower settings of γ indicate that the learner is less influenced by events that happen in the future. Referring back to the sink hole example, if one of the example episodes follows the path indicated in the left hand grid of Figure 2.21, the one step algorithms of the previous section only update the blue arrowed state on the final transition. This means that several passes over a path are needed for the final rewards to affect the other states on the rest of the path.

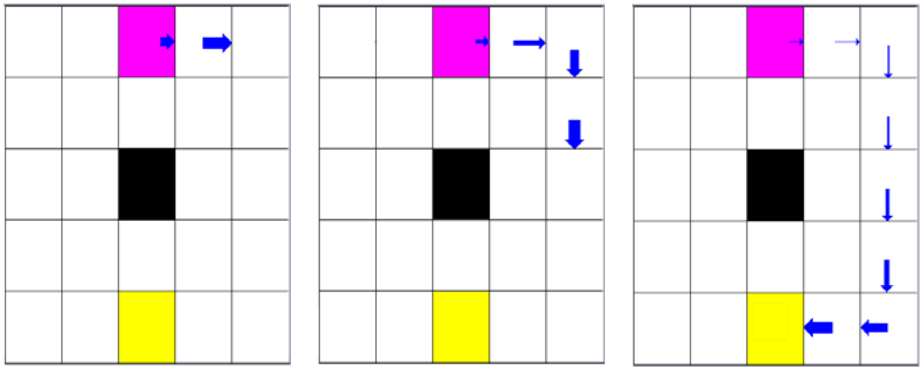


Path taken

Values affected by one-step algorithms

Figure 2.21: Sample episodes in a sink hole gridworld

Eligibility traces extend this idea of allowing things that happen in the future to more directly influence the original decision that the learner made [WAT89, SS96]. The learner keeps track of all the states that it visits during an episode. When it visits a state and chooses an action to take it saves the fact that it has been there by increasing the eligibility of the state/action pair ($e(s,a)$). The increase to $e(s,a)$ uses either *accumulating traces* or *replacing traces*. In the case of accumulating traces, 1 is added to $e(s,a)$ on each visit. Replacing traces simply resets $e(s,a)$ to 1 when it is visited again. In Figure 2.22, at t=1, the learner has visited two states and thus the blue arrows indicate the eligibility of those states. As the episode progresses, newly visited states are added to the eligibility trace. As indicated in the Figure 2.22, the eligibility of a state is also subject to decay over time. As shown at t=7, the most recently visited states are the most affected by the end reward of an episode. For eligibility tracing, a new parameter λ is introduced to control the decay of eligibility.



T=1 t=3 t=7

Figure 2.22: Eligibility traces in a sink hole gridworld

The eligibility traces decay at each time step $e(s,a) = \gamma\lambda e(s,a)$ and the update formula (for Sarsa) becomes:

$$Q(s,a) = Q(s,a) + \alpha\delta e(s,a) \tag{2.5}$$

Where $\delta = \left(r + \gamma Q(s',a') - Q(s,a)\right)$. δ is updated for the current step in the episode, whereas the Q update is carried out for all states where *e(s,a)* > 0 at every step,. For example with λ = 0.9, γ=0.9 and α=0.1, at the t=1:

$e(G[2,0],\ A_{EAST}) = 0.9*0.9*1.0 = 0.81$

$e(G[3,0],\ A_{EAST}) = 1$

At the end of t=1, the Q update is performed on these two states (as all other e(s,a) = 0):

$\delta = -1 + 0.9 * Q(G[3,0],\ A_{EAST}) - Q(G[2,0],\ A_{EAST})$

$Q(G[2,0],\ A_{EAST}) = Q(G[2,0],\ A_{EAST}) + 0.1* \delta * 0.81$

$Q(G[3,0],\ A_{EAST}) = Q(G[3,0],\ A_{EAST}) + 0.1* \delta * 0.81$

Conversely by the time the learner has reached the terminal state (in yellow), the eligibility of the first two states visited has decayed to:

$e(G[2,0], A_{EAST}) = 0.2288$

$e(G[3,0], A_{EAST}) = 0.2824$

At this final point there are 6 other active traces, and thus all 8 states will be updated as above.

It should be noted that the examples given above are applicable for on-policy algorithms. Eligibility traces can be applied to off-policy algorithms, however their application is not as straightforward due to the off-policy learning. Watkin's Q(λ) [WAT89] simply cuts the eligibility tracing whenever an exploratory action is taken, Peng's Q(λ) [PW96] mixes on-policy and off-policy learning to use eligibility traces. This work is primarily focused on on-policy learning, and thus further discussion of these methods is outside the scope of this research.

2.6 Linear Function Approximation in Reinforcement Learning

This section has reviewed the basic principles of reinforcement learning. The techniques presented are broadly applicable to many types of problems. However, due to their tabular nature they are not appropriate in most real domains. Real domains are often characterised by many *continuous*, rather than discrete, state variables. Tabular forms of reinforcement learning are therefore computationally unable to learn the problem without utilising some sort of function approximation.

One of the many challenges facing reinforcement learning in specifically large domain sizes is both the ability to deal with large state space sizes and also the ability to generalise about a new state based on the learner's experience of other visited states. The methods described in Section 2.1 are tabular. They rely on the ability to represent the state space as a giant lookup table. However, most interesting problems are often made up of state spaces that cannot be represented in pure tabular form. Consider a state space made up of two continuous variables. The number of possible combinations is infinite unless the designer knows what level of quantisation is required. For example, does the problem require precision in the range of 1 decimal place or is it 10 decimal places? If the state space can be designed in such manner, then it may be possible to use tabular forms of learning. Even if the state space can be reduced in this manner, it may still be too large.

The easiest technique used in function approximation is state aggregation. Possibly the earliest example is [CM68], but has been further developed in [SSR98]. This technique divides each state space variable into regions and considers each region as discrete variable. Recent research in function approximation techniques have concentrated on using non-tabular forms of Q-Learning [WAT89] and Sarsa [RN94] in a variety of different scenarios such as [PSD01, SASM99, SSK05]. This section investigates function approximation and associated learning algorithms.

2.6.1 Gradient Descent Learning

Gradient descent methods [SB98] rely on a parameter vector of features, $\vec{\theta}_n$ as depicted Figure 2.23. This vector is a large vector that contains all the features of the state space. Any state can be described by one or more of these features.

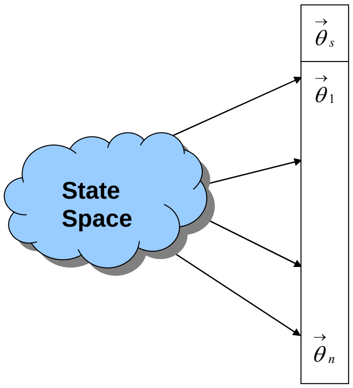


Figure 2.23: Parameter Vector for Gradient Descent

In the tabular case, a greedy action could be selected simply by choosing the best $Q(s,a)$ for all a_t. In Figure 2.24, the current state of the learner is described as [Money_Left = 12, Item_To_Buy = 3], and the greedy move is Bid 6.

State(s)		Action	Q(s,a)
Money_Left	Items_to_Buy	Bid	
12	3	8	0.1
12	**3**	**6**	**0.5**
12	3	4	0.3

Figure 2.24: State action pairs and Q values

In gradient descent methods, the state action pair [Money_Left = 12, Item_To_Buy = 3], a = [6] has its own vector of features $\vec{\phi}_n$ .Each entry in

$\vec{\phi}_n$ corresponds to the a basis functions $\{\phi_n(s,a)\}$ is the same size as the vector of parameters $\vec{\phi}_n$. The $Q(s,a)$ value is calculated as follows:

$$Q(s,a) = \sum_{j=0}^{n} \theta_j \, \phi_j(s,a) \tag{2.6}$$

In the following example, each feature is assumed to be binary. Thus $\vec{\phi}_n$ becomes a big binary vector, with each entry 1 to n indicating whether the corresponding feature in $\vec{\theta}_n$ is present in the state (1 for present, 0 for absent).

In the example, $Q(s,a)$ is the sum of all θ where $\vec{\phi}$= 1.

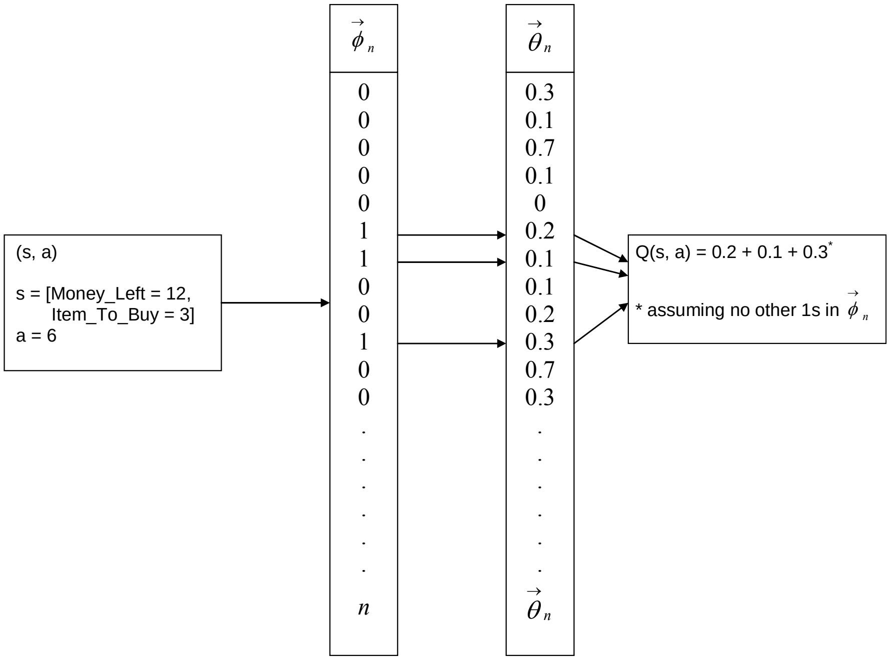


Figure 2.25: Calculating Q(s,a) using Parameter and Feature Vectors

Assuming this representation of the state space and assuming that the examples in the parameter vector appear with same distribution as the examples [SB98] suggests that a good approach is to try to minimise the mean squared error. This is done by adjusting the entry for the present feature(s) by a small amount. While there are both on-policy and off-policy gradient descent methods, the primary difference between on-policy and off-policy has been discussed in Section 1.4. Furthermore, Watkins Q Learning [WAT89, WATD92] may fail to converge when used with function approximation [TS93]. For the sake brevity only the linear, gradient-descent Sarsa(λ) is given here. (Figure 2.26)

```
Initialise $\vec{\theta}$ arbitrarily and $\vec{e} = \vec{0}$
Repeat for each episode {
   s ← initial state of episode
   For all a ∈ $\mathcal{A}$(s):
     $F_a$ ← set of features present in s,a
     $Q_a$ ← $\sum_{i \in F_a} \theta(i)$
   Choose $a_t$ using ε greedy policy.
   Repeat for each step of the episode{
     $\vec{e} \leftarrow \gamma\lambda \vec{e}$
     For all $\bar{a} \neq a$:                    (Replacing Traces)
        For all $i \in F_{\bar{a}}$:
           e(i) ← 0
     For all $i \in F_a$:
           e(i) ← e(i) + 1     or        1
                  (Accumulating or Replacing Traces)
     Take action a, observe r and s's'
     $\delta \leftarrow r - Q_a$
     For all a ∈ $\mathcal{A}$(s′):
       $F_a$ ← set of features present in s′,a
       $Q_a$ ← $\sum_{i \in F_a} \theta(i)$
     Choose a′ using ε greedy policy.
     $\delta \leftarrow \delta + \gamma Q_{a'}$
     $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta \vec{e}$
     a ← a′
   until s′ is terminal
```

Figure 2.26: Gradient descent Sarsa(λ) Algorithm

2.6.2 Linear Approximation

Feature selection is one of the most vital areas for gradient descent learning. The following sub-sections describe two different linear methods for selecting features.

2.6.2.1 Coarse Coding with Tile Coding

Cerebellar model articulation controller (CMAC), was first introduced by Albus [ALB81]. Over the last few years it has been adapted for use in reinforcement learning and renamed tile coding [SB98]. The basic principle behind tile coding is to overlay the state spaces with exhaustive partitions. Each partition is called a *tiling*, and every element in the partition a *tile*. Each tile makes up one feature and the total set of tiles in all tilings $\vec{\theta}$.

The resolution is divided into generalisation and granularity parameters. The generalisation parameter describes the shape of the tiles. The granularity parameter is described by the number of tilings overlaying the state space. These overlays are important in tile coding’s ability to make fine distinctions. The combination of generalisation and granularity is called the overall resolution.

For example, extending one of Sutton’s examples [*SuttonTC*], if a state space is described by two state variables x and y, one possible way to tile it is to create 4x4 regions across the state space. This creates broad generalisation between state values that are within 0.25 of each other (in both x and y). This level of generalisation is relatively coarse. To refine the detail of what is learnt, another tiling offset from the original can be placed over the state space. Figure 2.27

shows the original 2-dimensional state space with 2 offset 4x4 tilings. The example state lies in exactly one tile in each tiling. Generalisation of that state occurs with any other state that lies within that tile. Since the offset is different for each tiling, the cluster of states surrounding the original state differs.

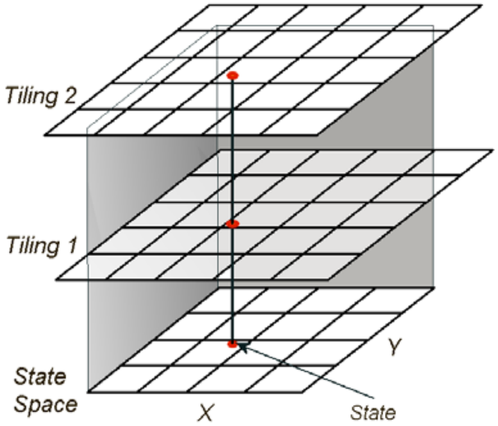


Figure 2.27: Calculating Q(s,a) using Parameter and Feature Vectors

The overall resolution of this example is 0.25/2 or 0.125. Finer resolution can be achieved by increasing the number of tilings. In summary, the shape and size of the tiles determines the type of generalisation that occurs between states, whereas the number of tiling overlays controls the distinctions made about them.

2.6.2.2 Radial Basis Functions

Radial basis functions [POW87] extend the idea of tile coding in that instead of a feature being present or not present, it can have a degree of belonging anywhere in the interval of [0,1]. Typically the feature has a Gaussian response function based on the distance between the current triggering state and the feature's "centre" (relative to the width).

2.6.3 Fuzzy Based Function Approximation

Knowledge representation is often represented in terms of binary opposites. *"The light is on, the light is off*". However, many things cannot be represented with this kind of binary logic. Knowledge works more along the lines of "*Give the plant a small amount of water every couple of weeks*" rather than "*Give the plant 15 ml of water every 14 days".* We are still able to deduce the right course of action even with the first statement, and in fact, more exact knowledge would not be any more helpful – as the example in [BEZ93] concludes, knowing that you should brake exactly 74 feet before you need to stop a motor vehicle is not actually useful in practice.

Representing and working with this type of knowledge is termed fuzzy logic. Fuzzy logic is capable of dealing with fuzzy data, vague rules and imprecise information. Systems that deal with "real" systems need to be able to cope with this kind of data.

The following two sub-sections investigate fuzzy set theory and its application in machine learning problems.

2.6.3.1 Fuzzy Sets

The idea of fuzzy sets is that, unlike binary logic where membership is described as 0 or 1, a fuzzy set contains several labels that describe different states of a variable. For example, suppose an agent wants to purchase a basket of oranges. When looking for oranges the agent may have different requirements; price, quality, and quantity. Some of these attributes are not things that are normally given binary values. For example, how does the agent judge what oranges of good quality are? Normally, we do not think of these types of requirements in binary terms.

A fuzzy variable consists of a set of symbolic labels called *fuzzy labels.* In the agent marketplace example, the variable price might be represented as [PRICE_LOW, PRICE_MEDIUM, PRICE_HIGH] and any specified attribute value (say price = £4) has a certain degree of membership to one or more labels in the price set. A set (or variable) is said to be *crisp* if the values it refers to are traditional discrete values. To determine how much a specified crisp value belongs to any given label, for example PRICE_LOW, a pre-defined membership function is applied. [BEZ93] defines *fuzzy sets* and *membership functions* as follows:

> If X is a collection of objects denoted generically by X, then a fuzzy set A in X is defined as a set of ordered pairs $A = \{(x, \mu_A(x)) \mid x \in X\}$, where $\mu_A(x)$ is called the membership function (or MF for short) for the fuzzy set A. The MF maps each element of X to a membership grade (or membership value) between 0 and 1 (included).

A fuzzy set is a mapping from a set of real numbers to a set of symbolic labels. For example, consider the world descriptor Money_Left from the states described Figure 2.14. The value of Money_Left in a crisp state consists of a discrete number, say ML(x), $x \in \mathbb{Z} = [0..15]$. However, in a fuzzy state, the same value x maps to one or more of the fuzzy labels associated with Money_Left =

[Lots_Money, Little_Money]. X's degree of belonging to any particular fuzzy label is defined by the *membership function* (μ) associated with the fuzzy set Money_Left. For example, the $\mu_{\text{Money\_Left}}$ and $\mu_{\text{Items\_to\_Buy}}$ might be described as:

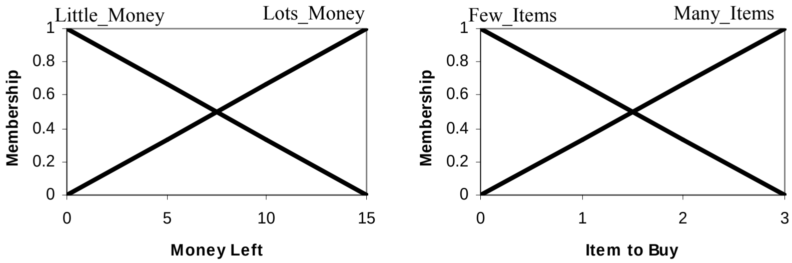


Figure 2.28: Membership function of Money_Left and Items_To_Buy

Crisp values are fuzzified using these types of membership function. Each crisp value will belong, to some degree, to one or more fuzzy set labels. In Figure 2.14, Money Left_{S1} = 12, fuzzification of this value results in:

$$\mu_{\text{Lots\_Money}}(12) = 0.87 \text{ and } \mu_{\text{Little\_Money}}(12) = 0.13$$

Therefore, fuzzy sets have *soft* or fuzzy boundaries, whereas the old form of state representation has *crisp* boundaries. Defining membership functions of fuzzy sets requires some level of knowledge engineering, i.e. the designer must have some intuition about the domain in order to make a reasonable mapping of crisp values onto the fuzzy labels. However, the design of fuzzy sets allows for flexibility in membership definitions. This flexibility allows for the existence of a *soft* boundary between labels.

The soft boundary can be used to illustrate the differences and highlight the relationship between fuzzy theory and probability theory. To further the examples from literature, this distinction will be illustrated with a marketplace example. The marketplace is again selling oranges and an agent wishes to purchase a basket of oranges. This agent can either purchase lot A or lot B:

$\mu_{\text{GOOD\_ORANGES}}(A) = 0.9$ $\Pr(B \in \text{GOOD\_ORANGES}) = 0.9$

Figure 2.29: Membership vs. Probability

The primary difference between the two types of information, is that A's 0.9 membership to the set of GOOD_ORANGES indicates that while the basket may contain some degree of rotten oranges, all in all it will still contain a fair amount of good ones. On the other hand, the probability statement that describes lot B, indicates that most of the time, B will contain good oranges. This statement says nothing about the quality of the oranges the remaining 0.1 of the time; the oranges could be all rotten during this period. Both probability and fuzzy membership express the level to which the basket belongs to the set of GOOD_ORANGES, however fuzzy membership also expresses the degree of belonging.

Further information about fuzzy set theory can be found in [BEZ93, BO82 and MUK01].

It is important to note that the membership functions can be described by any type of function that maps a variable X to a value between 0 and 1[1]. However,

[1] Including both straight line functions such as triangular or trapezoidal functions and curved line functions such as a generalised bell curves or sigmoidal (open left or right) functions.

membership functions that total 1 for any given crisp value ($\sum_{i=1..n} \mu_i(x) = 1$). It has been shown that systems that follow this rule are more robust to errors such as noise and design faults. [BBM99].

2.6.3.2 Fuzzy Reinforcement Learning

The history of reinforcement learning and fuzzy reinforcement learning can be traced through techniques developed for *learning classifier systems* (LCS). In a LCS an agent has a rule-based model of the world. It uses interactions with its environment to modify that rule base via some evolutionary process. LCS systems typically combine some type of *trial and error* learning[2] with a Darwinian evolutionary *survival of the fittest* mechanism. An introduction to LCS can be found in [HetAl00], and more recently [BK05]. [LR00] presents a review of some of the successful LCS systems.

In terms of LCS systems, the research presented in this thesis focuses on the techniques developed for the *learning fuzzy classifier system* (LFCS), especially on the fuzzy reinforcement learning aspects. An introduction to LFCS is provided by Bonarini in [BON00]. The reinforcement distribution concepts of these types of systems are particularly relevant. A variety of different researchers, including Bonarini, have proposed fuzzy extensions to the Q Learning algorithm. The following discussion will focus on three such proposals.

[2] I.e. Reinforcement Learning.

Glorennec proposed a version of Q-learning that uses fuzzy rules [GL94, GLJ97]. In this approach, the entire set of fuzzy rules is considered an agent that produces some action a. Each agent always triggers the same action. This architecture is described by a Q function $Q(s_t, a_t)$ and a rule quality $q(i, a_t)$. A Q-value is:

$$Q(s_t, a_t) = \frac{1}{2^n} \sum_{i \in H(s_t)} q(i, a_t) \tag{2.7}$$

where $H(s_t)$ is the set of all fuzzy rules that are triggered for the crisp state s_t and n is the number of input variables. Therefore, action selection is the $Q(s_t, a_t)$ with the largest summed rule quality. The update given to the rule quality is described as:

$$\Delta q(i, a_t) = 2^n act(i) \Delta Q \tag{2.8}$$

where $act(i)$ is the mean relative activity of the rule i for s_t, or the amount of contribution of rule i to a_t.

In this proposal, to select an action, the set of fuzzy rules activated in s_t needs to be evaluated for that action. In order to finalise action selection, all possible $Q(s_t, a_t)$ must be calculated. This can be computationally expensive, especially for large systems.

Berenji's Q-Learning [BER94, BER96, BERV01] deals with fuzzy constraints on the actions. An example of a fuzzy constraint is "the price of item A must not be *substantially* more than item B". This algorithm maintains an estimate for taking an action given the fuzzy constraint on the action. Therefore, the Q value update[3] becomes:

$$FQ(s_t, a_t) = FQ(s_t, a_t) + \alpha \left[\left(r + \gamma \max_b FQ(s_{t+1}, a) \right) \wedge \mu_c(s_t, a_t) - FQ(s_t, a_t) \right] \tag{2.9}$$

In this system, the action selected at t is the action with the maximum $FQ(s,a)$. Actions are selected rather than combined. This fuzzy extension only applies to the constraints $\mu_c(s_t, a_t)$. This system is quite different from most other fuzzy systems because it does not combine actions.

Bonarini [BON96, BON96a, BON97, BON98] presented a more truly fuzzy version of Q-Learning. In Bonarini's fuzzy LCS, the reinforcement fuzzy Q-learning section applies principles similar to Glorannec, extending the fuzzification to fuzzy goal states. In this algorithm, the states are fuzzified and the actions fuzzified. This creates a set of fuzzy rules, of which one or more fire for a specific *crisp* (non-fuzzy) state. From the rules that fire, the most appropriate fuzzy action is chosen. The set of fuzzy actions are recombined to produce a crisp action. In Bonarini's Q-Learning proposal [BON98] the update is given as follows:

$$\hat{Q}_{r_i}(s_t, a_t) = \hat{Q}_{r_i}(s_t, a_t) + \alpha \xi_{r_i}(reward + \gamma \max{}_j \hat{Q}_{r_j}(s_{t+1}, a)\xi_{r_{ji}}) - \hat{Q}_j(s_t, a_t) \tag{2.10}$$

[3] Where Q values become *fuzzy* Q-values or FQ values.

for all i, where $r_i \in R_t$ and R_t is the set of all fuzzy rules with $\mu > 0$ for the crisp state s_t; ξ_{r_i} is the relative contribution of the rule i (r_i) that matches a crisp state s, with respect to the total contribution of all rules that match s. This is given as:

$$\xi_{r_i} = \frac{\mu_{\tilde{s}_i}(s)}{\sum_{k=1,K} \mu_{\tilde{s}_k}(s)} \tag{2.11}$$

While this section has presented a review of Bonarini's Fuzzy Q-Learning, the paper [BON98] and others [BON96, BON96a, BON97] present other extensions such as Fuzzy-Q(λ), ELF[4] and a variety of successful experiment results mostly in robotic tasks such as navigation and pursuit. Bonarini proposes a methodology for applying RL algorithms to LFCS however, no on-policy algorithms are proposed.

2.7 Multiagent learning algorithms

One problem with most reinforcement learning algorithms is that the learning problem is *non-stationary*. In a non-stationary problem estimates, or Q values, never completely converge due to the changing nature of the environment. Tracking non-stationary behaviour has been dealt with in a variety of different ways. Some reviews in the area are provided by [TR96 and SV00].

[4] A LFCS using the fuzzy reinforcement learning algorithms presented.

2.7.1 Recursive Modelling Method

The Recursive Modelling Method Algorithm introduced by Gmytrasiewicz [GDW91, GLJ97 and NG97] is a formalism to represent and process models that one agent keeps of another agent. This method, based on game theory, is based on payoff matrices. It assumes that the agent doing the modelling will know the utility functions of all the other agents. The modelling agent calculates its payoff matrices based on the total expected payoff that all the agents will receive given that they follow a particular action. The modelling agent chooses an action based on what will maximise not only its own utility, but the utility of all the other agents. This algorithm is further extended in order to cope with recursive levels of modelling – how to model an agent when you know the agent is modelling you. The level these models descent is the agent's *knowledge depth*.

RMM can be solved using dynamic programming techniques. As commented on by Vidal in [VID98], RMM has several limitations. The first is the exponential growth of the matrices and modelling levels as the number of agents and knowledge depth is increased. This exponential growth leads to a high overhead in computational time to obtain a good solution. The second is that the nesting models assume that the agent doing the modelling has knowledge of other agents' payoff functions. [VID98] presents a knowledge dampening version of RMM called Limited Rationality Recursive Modelling Method, a framework for incorporating knowledge about other agents was presented in [VIDD97].

2.7.2 Minimax Q

Minimax-Q, proposed by Littman [L94] uses zero-sum games where the learner tries to maximise the payoff in the worst scenario. Essentially the algorithm switches between minimisation and maximisation depending on the state. This algorithm achieves good results with or without opponents. This algorithm utilises linear programming in each state and episode, causing it to be very slow in learning. Further details can be found in [L94 and LS96]

2.7.3 Nash Q

Nash-Q, proposed by Hu and Wellman [HW98, HW03], extends Q-learning to perform updates based on the existence of a *Nash equilibrium* over the Q values for all learners. A Nash equilibrium is defined as a set of strategies for each learner such that each learner's current strategy is optimal given the other learners' current strategies. This algorithm tries to learn the Nash-Q value; a value defined as the optimal Q-values in a Nash equilibrium. The major issue with this algorithm is the need to pre-calculate the Nash equilibrium values. Given these values, the algorithm will converge to Nash equilibrium policies under certain strict conditions to related to the Nash equilibriums.

2.7.4 WoLF

The research presented in [BV02] and [BV02a], combines tile coding, policy gradient ascent or policy hill climbing and a technique they call *Win or Learn Fast* (WoLF). The WoLF principle is a method for altering the learning rate (α)

depending on the current performance of the agent. If the learner is doing worse than expected, α is increased to encourage it to learn faster. If it is doing better than expect α is decreased because it is likely that the other learning agents will soon change their policy in response to their poor performance. Policy hill climbing and policy gradient ascent algorithms work similarly to the learning algorithms discussed, however they combine the learning principles with the ability to learn multiple policies. The principles of tile coding were reviewed in 2.6. The novelty is its combination of the three techniques and application to the multiagent learning problem. The results reported in stochastic games are promising. Intuitively this solution seems logical as it is more flexible than those that require calculations of equilibriums.

2.8 Summary

This section has provided a review of the theoretical background of reinforcement learning. It has introduced a variety of techniques that attempt to learn in non-stationary environments. The modelling techniques and multiagent learning algorithms are difficult to scale up to large games and are sometimes not applicable in adversarial environments.

To that effect the research presented in this thesis will investigate the use of function approximation techniques in adversarial environments. Specifically, the fuzzy techniques presented in Section 2.6.3 appear promising due to the diversity of rules that can be triggered for any particular state. However, although Q-learning has been popular choice with most algorithms including the fuzzy techniques, it has been shown that Watkins [WAT89] Q-learning may not converge correctly when used in function approximation. In particular, [TS93], have shown that because function approximation introduces *noise* into the calculation of *Q*-values, some values may be too large, and other too small. *Q*-learning uses the max operator, always picking the largest values, causing overestimation if the error intervals of several related *Q*-values overlap. Therefore, the research presented in this thesis will focus on on-policy methods of function approximation because they do not use the max operator.

To that effect, a further investigation into on-policy fuzzy methods and two separate proposals by the author of this thesis for on-policy fuzzy learning are provided in Section 4. Before that, Section 3 presents the three simulation environments used to evaluate and test the proposals made in Section 4.

3 Simulation System Design

The learning algorithms investigated and developed for the research presented in this thesis were tested in a variety of different learning problems. While Section 4 introduces the novel algorithms, this section presents some principles of simulation before presenting the three simulation systems used in this research and, finally, the techniques used for verification and validation of the systems presented, including the statistical methods used in data presentation and validation of results.

3.1 Random Numbers

In any type of simulation the use of random number generators (RNGs) is required. The sequence of numbers produced should be reproducible. This means that given the same seed, the RNG should produce the same sequence. This is important in order to aid debugging of the simulation and increase the reliability of the results [PJL02].

RNGs are pseudorandom, meaning that they rely on a deterministic mathematical process rather than some activity which is a fundamentally random natural process such as radioactive decay. [BZ03] provides a concise summary of the important properties of good random number generators (RNG). Some of these characteristics include reproducibility of the sequence, uniform distribution, a long period of numbers, independence (low levels of statistical correlation) and efficiency. Furthermore, for stochastic simulations, linear RNGs are the most widely used and much quicker than non-linear RNGs. There are several RNG algorithms that fulfil these requirements including Taus88, TT800, and Mersenne Twister [RNGTest].

The simulation systems developed for this thesis used version 7 of the Java version of the Mersenne Twister algorithm [MN98] available from Sean Luke [MTJava] for random number generation. This algorithm is available in both synchronised[5] and non-synchronised form. The Mersenne Twister algorithm has the following advantages: It has been designed with consideration of the flaws of various existing generators, it is freely available in a wide variety of languages. It has a long period and high order of equidistribution (period: 2^{19937}-1 and 623-dimensional equidistribution property is assured.) and finally, fast generation. .

3.2 A Marketplace Simulation

The agent marketplace was chosen for three reasons:

1. It is a scaleable domain;
2. It offers the potential for multiagent scenarios where both agents are competing for the same goal, and;
3. It has a close relationship to real world problems, such as bandwidth brokering and other auction scenarios.

[5] In a multi-threaded environment, if one or more threads share the same object, it must be synchronizable to provide safe access.

The following section details the design and setup of the marketplace simulator used in the research presented in this thesis. Section 3.2.1 investigates multi-agent platforms currently available before concluding that a simpler multi-agent platform was required. This platform is presented in Section 3.2.2 and the behavioural algorithms used for agent construction in Section 3.2.3.

3.2.1 Agent Platforms – FIPA-OS and Zeus.

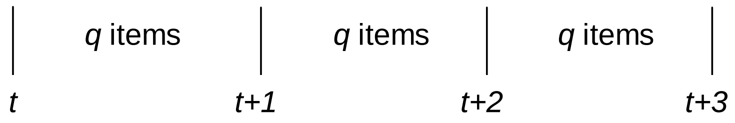
Several multi-agent technologies, such as FIPA-OS and Zeus were investigated thoroughly at the beginning of this project. FIPA-OS is an open source agent platform. This platform contains the mandatory elements required for agent interoperability that are specified in the FIPA[6] specifications. The FIPA-OS platform originated from Nortel Networks. The Zeus multi-agent platform was developed by BT Labs and is also FIPA compliant. Both systems are Java based, and include many useful interoperability features for building commercial agent systems. However, the learning-curve for both systems is quite steep and at the time, neither system offered pared down functionality for the purpose of simulation. It was decided that a better approach for this project was to write an auction simulator with only the functionality required.

[6] Foundation for Intelligent Physical Agents

3.2.2 The AgentSim Marketplace

The marketplace is modelled in order to cope with scarce items. Two possible marketplace auction mechanisms include: (1) An ascending price English Outcry, and (2) Sealed bid. The auctions themselves are modelled as a continuous episodically levelled task. By this it is meant that an amount of items being auctioned in a time period t and the number of auctions the agent knows will occur in each time period are fixed. Essentially time is broken up into equal periods, during each time period t, a certain quantity q of the item is available. During the first episode occurring at time t for q items must be completed before $t+1$.

Marketplace structure:



An episodically levelled task complicates the learning issue because there are two levels of learning possible. The first level is simply to learn what do in any one auction of an item. However, if the auctions are episodically levelled, a better solution would be to learn a strategy over the game of auctions. Thus in the following discussion the terms auction and auction game have particular meaning. An *auction* refers to one event of auctioning an item within an episode or an *auction game*.

In order to learn patterns of bidding in different auctions the game is modelled as a *general sum* stochastic game rather than a *zero sum* game. This means that the rewards the agents receive do not total to zero (i.e. one agent gets +1 and the other agent -1 in any given auction game). Instead the agents receive a reward based on the number of items received within a game (some set of auctions).

The AgentSim Marketplace system developed for this project is written in Java using Java 1.4.2 [JAVA] and SimJava [SJAVA]. SimJava is a freely available process based discrete event simulation package for Java. The package has animation capabilities which allow a visual display of the inner workings of the simulated entity. SimJava allows for the creation of multiple animation entities. These entities can then be joined up in order to send and receive events as required. The simulation is handled by a central controller, which manages all the simulation threads, collects and delivers simulation events to the appropriate entity, and finally advances the simulation time when appropriate.

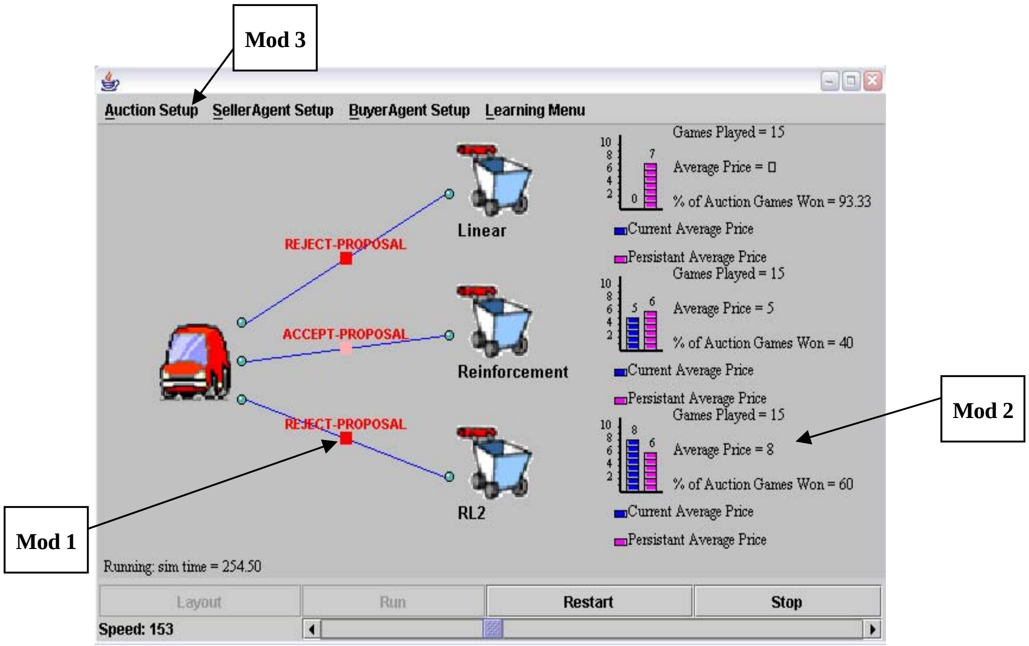
Utilising both this package and other Java packages, the AgentSim Marketplace was constructed in order to simulate an agent marketplace. The first task was to extend the SimJava animation capacity to allow detailed entity-entity messaging. This function is useful for debugging the auction simulator, as it allows information (such as agent bids and the type of message sent), to be immediately recognised by the user. It is also valuable for explaining the simulator to a third party.

The SimJava package was extended by the author of this thesis in the following ways:

- ✓To allow the user to specify both text, and colour coding to the visual message that are passed between the entities. (Mod 1)
- ✓The creation of a bar graph package and side labels to track data while the simulation is running. (Mod 2)
- ✓Addition of application option for simulation (original package is only intended for applets). (Mod 3)
- ✓Conversion of AWT graphics to Swing graphics[7]. (Mod 4)

Modifications 1-3 to the SimJava package can be seen in the agent marketplace simulation GUI:

[7] Swing package is a graphics API released after the original AWT package that provides lightweight widgets, that have pluggable look and feel are extensible and scaleable.



Games Played = 15
Average Price = 0
% of Auction Games Won = 93.33

Games Played = 15
Average Price = 5
% of Auction Games Won = 40

Games Played = 15
Average Price = 8
% of Auction Games Won = 60

Running: sim time = 254.50

Speed: 153

Figure 3.1: GUI Interface for Agent Simulator using Mod 4 graphics

Figure 3.1 illustrates the AgentSim Marketplace GUI. The GUI sets up and runs simulations between a seller agent and 2 to 5 buyer agents. The number of auctions and the agents are all configurable through the title bar menu.

The AgentSim interface also has a non-GUI option, allowing the simulation to run without the overhead of graphics processing. Regardless of which option is chosen (GUI or non-GUI) the simulations are configurable as follows:

The auctions are configurable with the following parameters:

- Number of auctions occurring in an episode.
- Standard deviation of number of auctions.

The Seller Agent Setup menu allows the following alterations to the Seller Agent:

- Start money.
- Minimum price accepted for an item.
- Market valuation of an item.

The Buyer Agent Setup menu allows the following alterations to the Buyer Agents:

- Number of Buyer Agents participating in the auctions. (2-5)
- Strategy type (bidding) of each agent – Linear, Greedy, Sarsa, Fuzzy Sarsa, FQ Sarsa and gradient descent Sarsa(λ) with tile coding.
- Number of items required to purchase during the episode.
- Start money.
- Maximum price to bid.
- Market valuation of an item.

The Learning menu allows the user to configure:

- Toggle learning (Learn vs. graphical debug run).
- Number of iterations for learning.(100 – 500,000)

Game Size	Items to Buy per Agent	Total Auctions	Min Price	Max Price
Very Small	2	4	5	6
Small	4	8	5	8
Medium	6	12	5	12
Large	10	20	5	14
Very Large	15	30	5	18
VVVLarge	20	40	5	24
Huge	35	70	5	30

Figure 3.2: Game sizes for marketplace simulations

Figure 3.2 depicts the game sizes used for all tests. For example, in a Game Size of Medium, an agent must buy 6 items from 12 auctions with an allowable price range of 5 to 12 units.

3.2.3 Seller and Buyer Agent Algorithms.

The agents in the marketplace simulation system compete in a first price sealed bid auction. The first price sealed bid auction follows the contract-net protocol set by FIPA [FIPA02]:

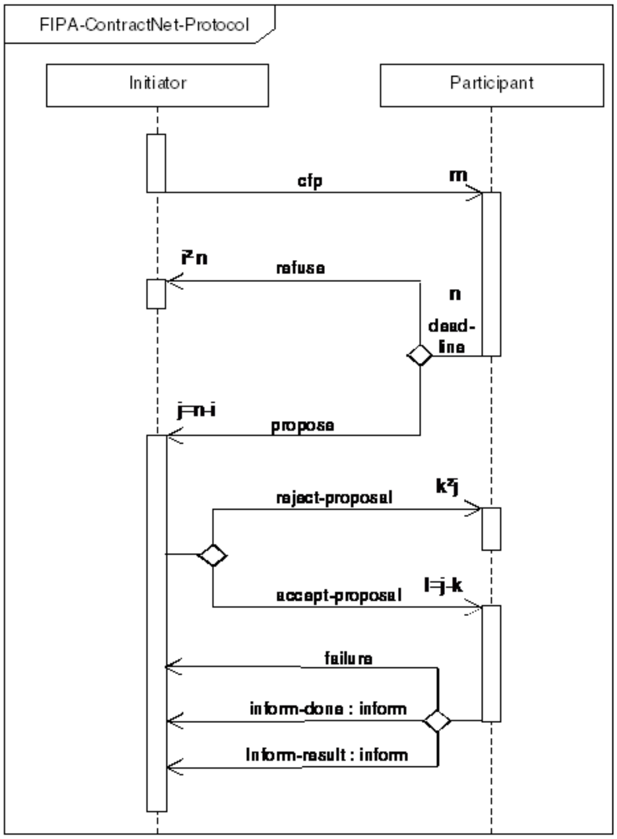


Figure 3.3: FIPA Contract-Net Protocol from [FIPA02]

Following that example, the seller agent follows the following algorithm:

```
Initialise agent.
 Repeat for each episode (or auction game){
   Initialise game (reset items to sell, etc).
   Issue CFP.
   Wait for bids
   If bids received
        Choose highest bid or randomly choose among
        highest bid.
   Else end auction.
   If auction not over {
        Issue accept-proposal and reject-proposal
        statements to appropriate agents.
        Receive payment.
   }
}
```

Figure 3.4: Seller Agent Algorithm

All buyer agents, regardless of their bidding strategy, follow this general algorithm:

```
Initialise agent.
 Repeat for each episode (or auction game){
   Wait for CFP.
   Receive CFP.
   Propose action (according to bidding strategy –
     propose or refuse)
   If receive accept-proposal
        Send inform.
   Update state.
}
```

Figure 3.5: Buyer Agent Algorithm

The rewards used in this simulation were based on overall achievement of the agents' goal. They can be summarized as follows:

$I(\frac{M_t}{M_{t=0}})$ `if all items purchased`

$-I_{Needed}$ `if all items NOT purchased`

0 `all other non-terminating steps`

Where I is the number of items, M is the amount of money at time t.

3.3 The Gridworld Pursuit Problem

The predator/prey gridworld was chosen for similar reasons to the agent marketplace:

1. It is an easily scaleable domain;
2. It offers the potential for multiagent scenarios where both agents are competing for different goals, and;

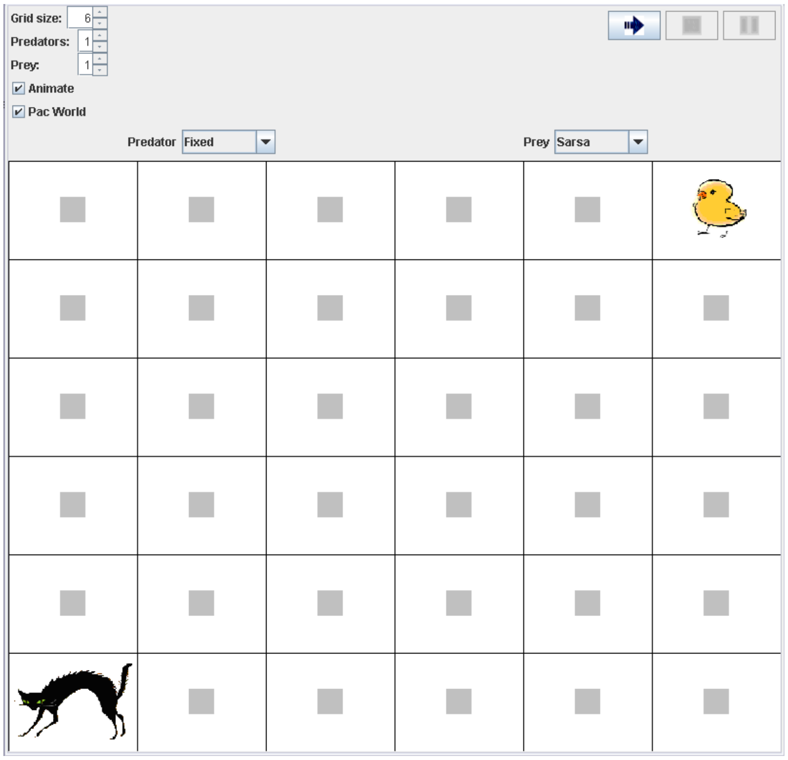
3. Its similarity in basic dynamics to many classic reinforcement learning examples such as those presented by Stone and Veloso [SV00] and because it is the base domain for more complicated applications such as robotic soccer [ST00 and SSK05].

This section details the design and setup of the predator/prey gridworld simulator used in the research presented in this thesis. This platform is presented in Section 3.3.1 and the behavioural algorithms used for agent construction in Section 3.3.2.

3.3.1 Predator/Prey Gridworld

As illustrated in Figure 3.6, in this domain there are n predators (cats) and m prey (chicks). The goal of the problem from the predator's standpoint is to catch the prey as quickly as possible. The goal of the prey depends on the type of grid chosen. If the checkbox *Pac World* is selected the goal of the chick is to eat all the chicken feed (grey dots) and avoid the predator. The episode is over if the chick eats all the chicken feed before being eaten by a cat, a cat eats the chick or MAX_STEPS is exceeded. If *Pac World* is not selected, the goal is simply to avoid the predator[8] and the episode ends when either the cat eats the chick or MAX_STEPS is exceeded.

[8] This setting is used as validation, since the learning problem facing the agent is much easier. It will not be discussed any further.



Grid size: 6

Predators: 1

Prey: 1

☑ Animate

☑ Pac World

Predator Fixed

Prey Sarsa

Figure 3.6: Predator/Prey Grid World

The simulation system is configurable in the following ways: it allows for variable grid size (3x3 to 20x20), variable number of predators and prey (1 to 4), animation on or off (off to learn quickly), game type selection and agent algorithm selection (Fixed, Sarsa, Fuzzy Sarsa, or tile coding).

3.3.2 Predator/Prey Agent Algorithms

```
Initialise agent tables.
 Repeat for each episode {
   Make move according to policy.
   If receive reward/penalty
   Update state.
}
```

Figure 3.7: Predator/Prey Agent Algorithm

Figure 3.7 gives the simple behavioural algorithm for both the predator and prey. In the case of a stationary agent, the agent simply ignores any reward signal. The rewards are summarised as follows:

For the prey:

$-g$	`if eaten by Predator`
$-g\left(\frac{s_{\min}}{s_t}\right)$	`if all dots have been eaten`
$+1$	`if dot eaten`
0	`if empty square`

For the predator:

0	`if eats prey`
-1	`all other states`

3.4 Mountain-car Problem

The mountain-car world was chosen as a benchmark environment for the final algorithm investigated in the research presented in this thesis, gradient descent Sarsa(λ) with tile coding. This is because this problem is described in detail by [SS96, SUTT96 and SB98]. Furthermore, the implementation used in the previous papers is publicly available.

The following section details the design and setup of the mountain-car simulator used in the research presented in this thesis. The dynamics and porting of this platform is described in Section 3.4.1. The mountain-car world is not a multi-agent platform and is primarily used for validation purposes.

3.4.1 Mountain-car World

In the problem the learner controls an underpowered car that is situated in a deep valley. The goal problem is to get the car to the top of the mountain. The difficulty is that the car is underpowered and thus cannot gain enough momentum by simply going forward to get it to the top of the mountain. In order to find a solution, the learner must first move away from the goal.

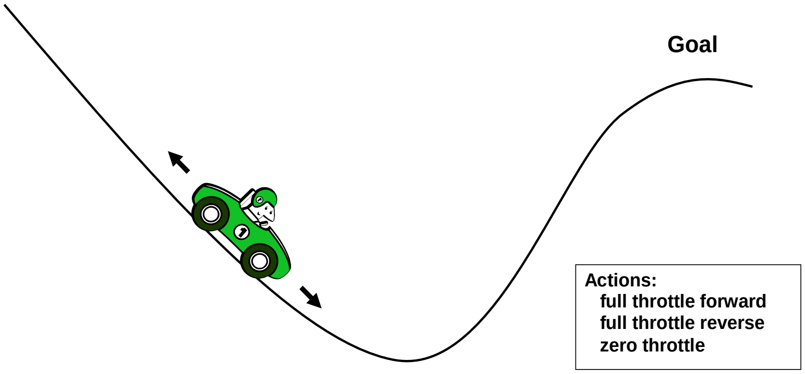


Figure 3.8: Mountain-car World

The car moves according to:

$$p_{t+1} = bound[p_t + p_{t+1}]$$

$$v_{t+1} = bound[v_t + 0.001a_t + -0.0025\cos(3v_t)]$$

Where p_t and v_t is the car's position and velocity at time *t* and a_t the action taken. The bound operation enforces $-1.2 \le p_{t+1} \le 0.6$ and $-0.07 \le v_{t+1} \le 0.07$. The rewards in this domain summarised as follows:

-1 `for all non-terminating states`

This environment serves as a control world, the implementation is a Java conversion of the original C++ Mountain-car world including the gradient descent Sarsa(λ) with tile coding learner provided by Sutton [SuttonMC]. The correctness of the ported Java code was compared by the author of this thesis with the original code by conducting identical simulations in both languages and comparing the results. In all cases the ported code had the same output as the original C++ code.

3.5 Verification and Validation

In order to determine accurate function of the simulation systems built, all three simulators needed to be verified and validated. [SAR98] defines validation as the substantiation that a computerised model exhibits a satisfactory range of accuracy with the intended application and verification as assurance that the program of the model and its implementation are correct.

[SAR98] summarises a variety of different validation techniques. The ones used to valid the simulation systems presented in this thesis were: Animation, Traces, Degenerate Tests, Event Validity, Fixed values and Internal Validity. Each simulator's operational behaviour was validated in function by using animation in the developed GUI[9], combined with breakpoint flow tracing checks. This allowed for verification of the behaviour flow and event validity of each simulation system.

[9] In the case of the agent marketplace and the predator/prey gridworld.

In the next step of validation, fixed value and degenerate testing was done with each learning algorithm to ensure the computerised version behaved as stated in the algorithm definition. Some examples of the kind of testing performed in this section are:

- Observation of the current state and rewards received on one simulation cycle and manual checking the expected new Q-values.
- Observation of the actual state of the system versus the learning agent's perception of the system.

Finally, extensive application of Java's runtime exception handling was implemented to ensure the system remained in a consistent state during runtime operation. Any unexpected values or illegal states trigger program alerts and termination.

The following discussion serves two purposes. The first is to demonstrate how interval validity testing was done, and the second, to describe the procedure undertaken in the mean calculations present in all future sections. As is often the case in experiments that require large amounts of computing power, it is impractical to have large number of sample experiments. Therefore, it is important that the appropriate statistical methods are employed for analysis of small sample sizes.

The results presented in the subsequent sections of this thesis fall into the small sample size category. The small sample size techniques are described in further detail in [JB92]. Although the tables presented in Figure 3.9 and Figure 3.10 only illustrate 5 experiment examples, unless otherwise stated, the results presented in subsequent sections of this thesis are over 10 experiments. The confidence intervals used are the 95% confidence intervals calculated using the *Student's t* distribution. There are several different types of data collection files, including win ratio statistics[10], and solution quality statistics[11].

Figure 3.9 gives the convergence or percentage win ratio of 5 trials of a Fuzzy Sarsa Agent in the marketplace world. In all 5 trials, the experiment setup is identical except for the seed given to the random number generator. The seed used is reflected in the column headings 2 to 6. The seventh column is the calculated mean of the trials and the final column the 95% confidence interval using the student's t distribution. The entries in each column represent the averaged win ratio of the agent over the previous 100 trials. Unless otherwise indicated, all graphs presented in further sections have been smoothed at an increment of 100[12]. Each simulation runs for 10000 trials unless otherwise stated. (Figure 3.9 and Figure 3.10 only give the first 1000 trials for the sake of brevity).

[10] How often the learner achieves the stated goal.

[11] The average price achieved by the learner or average number of moves taken to get to the goal.

[12] In other words 100 data points have been averaged to obtain the entry for each table entry.

Trial Num	*Buyer1 SEED-1*	*Buyer1 SEED-10*	*Buyer1 SEED-13*	*Buyer1 SEED-21*	*Buyer1 SEED-222*	*Buyer1 MEAN*	*Buyer1 95%-CI*
0	0	0	0	0	0	0	0
100	66.7702	62.9640	70.7302	67.3891	76.3037	68.8315	4.7720
200	72.3474	69.2209	73.1138	68.7822	73.4912	71.3911	2.1218
300	69.5010	69.0809	72.0394	71.1836	71.0312	70.5672	1.1782
400	70.7310	69.6275	72.5001	70.9231	70.2395	70.8042	1.0224
500	72.2346	70.1037	71.9283	70.9165	69.5233	70.9413	1.1037
600	73.3731	71.1936	71.4863	71.3805	69.6241	71.4115	1.2700
700	74.5867	72.7285	71.6324	72.3249	72.6120	72.7769	1.0466
800	75.6051	74.1008	73.4469	74.2146	74.8125	74.4360	0.7758
900	75.8267	75.7584	75.5080	74.8936	76.5001	75.6973	0.5533
1000	76.2164	77.2700	77.0455	76.1493	78.0193	76.9401	0.7437

Figure 3.9: Percentage win ratio from a Fuzzy Sarsa Agent

Figure 3.10 illustrates the data collected in 5 trials of a Fuzzy Sarsa Agent in the marketplace world. In all 5 trials, the experiment setup is identical except for the seed given to the random number generator. In simulations where more than one learner is present, different start seeds are given to each learner. The entries in each column represent the averaged price achieved for i items in n auctions. In the case where the agent did not win the required number of auctions, the calculation uses the maximum possible price for the missing items.

Trial Num	*Buyer0 SEED-1*	*Buyer0 SEED-10*	*Buyer0 SEED-13*	*Buyer0 SEED-21*	*Buyer0 SEED-222*	*Buyer0 MEAN*	*Buyer0 95% CI*
100	18.6917	18.4740	18.4800	18.7826	18.5780	18.6013	0.1284
200	18.5289	18.6860	18.5263	18.7629	18.5449	18.6098	0.1033
300	18.2560	18.7731	18.3389	18.2771	18.2349	18.3760	0.2149
400	18.3194	18.0469	18.1700	18.2791	17.9049	18.1441	0.1626
500	17.9969	17.7714	17.8291	17.8720	17.7797	17.8498	0.0874
600	17.1849	17.4340	17.5331	17.5689	17.8314	17.5105	0.2230
700	16.7514	17.2980	17.7609	18.7589	17.9143	17.6967	0.7123
800	16.4406	18.2437	17.1186	17.2089	17.8954	17.3814	0.6728
900	16.7491	17.0960	17.2806	16.3654	17.2429	16.9468	0.3688
1000	16.6897	16.2706	14.8837	16.6346	18.0580	16.5073	0.9815

Figure 3.10: Price data from a Fuzzy Sarsa Agent

The confidence intervals are calculated using the equation for small samples with a normal population using the student's t distribution. This interval is given by:

$$\left(\overline{X} - t_{\alpha/2}\frac{s}{\sqrt{n}}, \overline{X} + t_{\alpha/2}\frac{s}{\sqrt{n}}\right) \tag{3.1}$$

where $t_{\alpha/2}$ is the upper $1-\alpha/2$ percentile of the t distribution with degrees of freedom $= n-1$.

In terms of interval validity, the 95% confidence interval given in Figure 3.9 and Figure 3.10 seem to be within a tolerable limit for a learning agent. However to increase the certainty of the intervals, typically a minimum of 10 averaged experiments was used.

3.6 Summary

This section introduced some of the principles of simulation before presenting three simulation systems used in the research presented in this thesis. The first two simulation systems presented in Section 3.2 and 3.3, the agent marketplace and the predator/prey gridworld, enabled the learning algorithms introduced in Section 4 and the other algorithms for comparison purposes, to participate in a multiagent environment. The final simulation system presented in Section 3.4, the mountain-car world, served primarily as a validation domain. Finally, Section 3.5 gave the validation and verification techniques used at both the domain level and experiment level.

4 Fuzzy Learning in a Marketplace Environment

As introduced in Section 2.6, function approximation is a popular methodology to deal with large state spaces. Since one of the aims of the research presented in this thesis is to analyse algorithms that are capable of taking actions in completely new states based on generalising from the existing visited state space, function approximation seems like a promising way of achieving directed coevolution. The biological basis of coevolution in nature advocates that a group evolves according to their experiences with the other, evolving, actors in their environment. It therefore seems to follow that on-policy learning may provide the best solution for coevolution. This decision is reinforced by the fact that off-policy methods like Q-learning have been shown to be unstable with several types of function approximation [Watkins89]. Sutton however has shown that the approximation technique of tile coding combined with Sarsa [SUTT96], as initially done by [RN95], is capable of finding a robust solution to problems that had previously been shown to exhibit unstable behaviour [BM95] with function approximation.

As previously discussed, fuzzy sets also present an interesting way of creating a complex mapping of the state space. Bonarini has shown that the fuzzy Q-learning algorithm has been able to successfully use fuzzy rules in order to implement a variety of large state space problems in the controller domain [BON98], [BON97], [BON96]. The following two sub-sections describe two novel on-policy fuzzy methods. The first one is based on a basic fuzzy collapse of the state space and Berenji's [BER94] approach, and the second uses

Bonarini's methodology for extending crisp algorithms described in [BON98]. Section 4.3 presents the results of some experiments using these two new algorithms.

4.1 FQ Sarsa

The FQ Sarsa algorithm is based on the Sarsa algorithm. Essentially, it reduces the state space by storing the state representation in fuzzy sets. In all other respects, it behaves exactly like Sarsa. The algorithm does not consider fuzzy actions or goal states, leaving these in their original crisp representation and thus can be considered a hybrid algorithm. In this approach, a crisp state s matches a set of fuzzy states and these fuzzy states are paired with crisp action values. To determine the fuzzy state, a mapping from the set of real numbers representing the current state to a set of symbolic state labels is created.

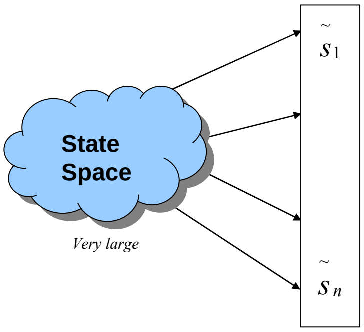


Figure 4.1: Fuzzy State Space Mapping

Consider the world descriptor Money_Left from the states described in Figure 2.14. The value of Money_Left in crisp state s consists of a discrete number, say $ML(x), x \in Z = [0..15]$. However, in a fuzzy state, the same value x maps to one or more of the fuzzy labels associated with

$Money\_Left = [Lots\_Money, Little\_Money]$. The degree to which x belongs to any particular fuzzy label is defined by the membership function (μ) associated with the fuzzy set *Money_Left*. So for example, μ_{Money_Left} might be described as:

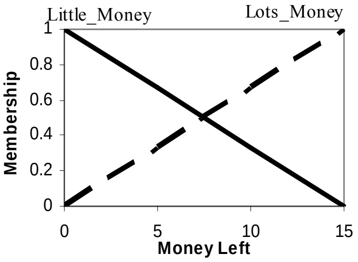


Figure 4.2: Membership function of Money_Left

The crisp values are then fuzzified using these membership functions. Each crisp value will belong to different degrees to one or more fuzzy label sets. Recall that in Figure 2.14 one potential state for a marketplace agent was $S1 = [Money\_Left = 12, Items\_To\_Buy = 3]$. The fuzzification of $Money\_Left_{S1} = 12$ results in $\mu_{Lots\_Money}(12) = 0.87$ and $\mu_{Lots\_Money}(12) = 0.13$. To fuzzify a crisp state, the membership of each state item is fuzzified, and typically, the *and* [13] is calculated to obtain the state's membership or degree of matching. In the case of state $S1$ of Figure 2.14, crisp state $S1$ belongs to fuzzy states $\hat{S}1_b$ and $\hat{S}1_d$ with membership 0.87 and 0.13 respectively. All fuzzy states $\hat{S}1_n$ (where n is the number of possible matches for $S1$) and there respective membership calculations are depicted in Figure 4.3.

[13] The minimum of the two values.

Fuzzy State	Money Left	$\mu_{\text{Money Left}}$	Auctions Left	$\mu_{\text{Auctions Left}}$	μ_{S1}
$\hat{S}1_a$	Lots_Money	0.87	Few_Auctions	0	0
$\hat{S}1_b$	Lots_Money	0.87	Many_Auctions	1	0.87
$\hat{S}1_c$	Little_Money	0.13	Few_Auctions	0	0
$\hat{S}1_d$	Little_Money	0.13	Many_Auctions	1	0.13

Figure 4.3: Fuzzification of Crisp State S1

As explained earlier, in FQ Sarsa the actions are not fuzzified. As a result, the selection mechanism operates greedily rather than utilising any sort of fuzzy calculation mechanism, such as the centre of mass approach presented in the next algorithm. At any given time t, the action that is selected is the best action (the one with the highest FQ value) for the most fit fuzzy state ($\max \mu(\hat{s}_t)$, where μ is the degree of matching of crisp state s to fuzzy state $\hat{s}$).

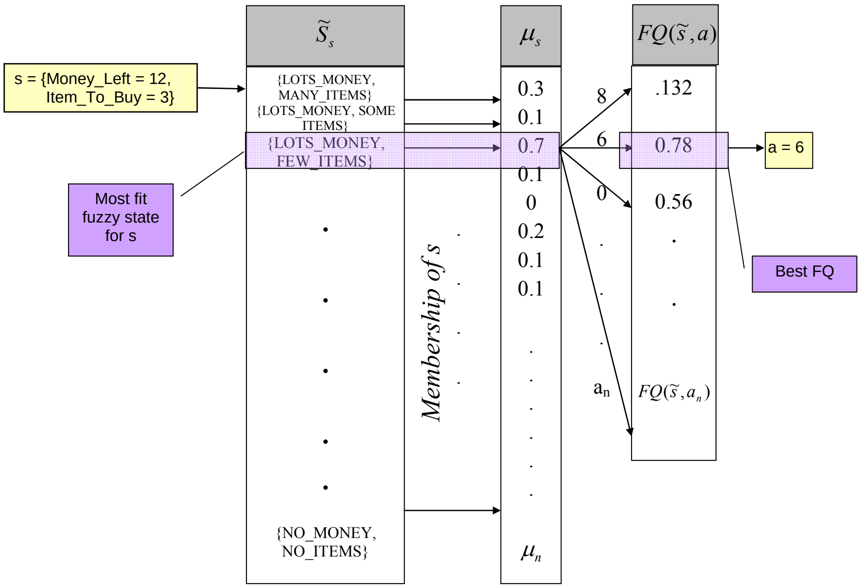


Figure 4.4: Fuzzy FQ Action Selection

The FQ value update formula is modified from Sarsa as follows.

$$FQ(\hat{s}_{t-1}, a_{t-1}) = FQ(\hat{s}_{t-1}, a_{t-1}) + \alpha(r + \lambda FQ(\hat{s}_t, a_t)\wedge \mu(\hat{s}_t) - FQ(\hat{s}_{t-1}, a_{t-1})) \qquad (4.1)$$

Rather than take the max of future rewards, it is replaced with the FQ value of the new state action pair reached by applying the current policy - $FQ(\hat{s}_t, a_t)$. In other words, $FQ(\hat{s}_t, a_t)$ is the state with the highest degree of matching (*max* $\mu(\hat{s}_t)$) and the a_t is action chosen following the current policy (i.e. ε-greedy). For this algorithm, Berenji's [BER94] fuzzy action constraints are considered to be the $\mu(\hat{s}_t)$. Therefore the algorithm also uses the fuzzy *and* operation of $FQ(\hat{s}_t, a_t)$ and $\mu(\hat{s}_t)$.

FQ Sarsa was presented by the author of this thesis in [TBC04]:

```
All $FQ(\hat{s}, a)$ values initialised.
Repeat for each episode (or auction game){
Initialize $\hat{s}_t$ (start state for the auction game).
 Choose $a_t$ from $\hat{s}_t$ using $\varepsilon$ greedy policy.
 Repeat for each step(auction) in the
                   episode(auction game){
  Take action $a_t$, observe $r$ and $\hat{s}_{t+1}$
  Choose $a_{t+1}$ from $\hat{s}_{t+1}$ using $\varepsilon$ greedy policy
   $FQ(\hat{s}_{t-1}, a_{t-1}) = FQ(\hat{s}_{t-1}, a_{t-1}) +$
          $\alpha(r + \lambda FQ(\hat{s}_t, a_t) \wedge \mu(\hat{s}_t) - FQ(\hat{s}_{t-1}, a_{t-1}))$
$\hat{s}_t$ = $\hat{s}_{t+1}$, $a_t$ = $a_{t+1}$
 }
}
```

Figure 4.5: FQ Sarsa Algorithm

4.2 Fuzzy Sarsa

The FQ Sarsa algorithm presented above does not utilise fuzzy principles to combine actions, it only selects them. Essentially it only concentrates on reducing the state space and is not capable of fuzzy rule interaction. To that effect, the Fuzzy Q-Learning algorithm presented by Bonarini [BON96] is described in further detail and extended by the author of this thesis to on-policy learning.

Fuzzy Q-Learning and Fuzzy Sarsa use fuzzy representation of both states and actions. Their state/action entries do not include crisp actions like FQ Sarsa or Berenji's Fuzzy Q-Learning [BER94]. Figure 4.6 illustrates the fully fuzzy state/action pair used by the Fuzzy Q-Learning and Fuzzy Sarsa. In FQ Sarsa, the degree of matching is still based on the fuzzy state. However, membership functions for the fuzzification and defuzzification of fuzzy actions are also required.

Fuzzy State		Fuzzy Action
Money Left	Auctions Left	Bid
Lots_Money	Many_Auctions	Bid_High
Lots_Money	Many_Auctions	Bid_Low

Figure 4.6: Fuzzy state action pairs

An example of defuzzification is Bid_High translating to the crisp action Bid 8. This type of fuzzy state action pair is referred to as a fuzzy rule where the fuzzy state corresponds to the antecedent of the rule and the fuzzy action proposed is the consequent. All fuzzy rules have a strength associated with them. It is this strength (FQ value) that the algorithms attempt to learn. A crisp state s matches a selection, or sub-population, of fuzzy states. A fuzzy rule is defined as the combination of fuzzy state $\hat{s}$ and the fuzzy action $\hat{a}$ that it proposes.

In the action selection portion, the rule chosen from a sub-population of rules is the one with the highest FQ value. Recall that since a crisp state s might match a number of fuzzy states (set $FS(s)$) as seen in Figure 4.3[14], a method is needed in order to determine what action to take when all rules could be proposing different actions. For all $\hat{s} \in FS(s)$, there will be at *least* one matching fuzzy state action pair, or fuzzy rule r. The action proposed for each $\hat{s}$, will be the greedy action (highest FQ-value) proposed by the fuzzy rule. The final action proposed is a weighted average of the actions proposed by each rule that is triggered. These actions are weighted by the degree of matching of the crisp state s with the antecedent of the rule. The weighted average is computed using the centre of mass approach:

$$a = \frac{\sum\limits_{i=1..n} \mu_i a_{\hat{s}_i}}{\sum\limits_{i=1..n} \mu_i} \tag{4.2}$$

where n is the number of fuzzy states matching crisp state *s,* and $a_{\hat{s}_i}$ is the best action (having been defuzzified) proposed by any rule matching $\hat{\mathrm{s}}_{\mathrm{i}}$. Any fuzzy state with membership > 0 is considered in the action calculation.

	Fuzzy State		Fuzzy Action	
μ	Money Left	Auctions Left	Bid	FQ(ŝ,â)
0.7	Lots_Money	Many_Auctions	Bid_High	0.4
	Lots_Money	Many_Auctions	Bid_Low	0.1
0.4	Little_Money	Few_Auctions	Bid_High	0.2
	Little_Money	Few_Auctions	Bid_Low	0.6

Figure 4.7: Fuzzy state action pairs with μ and FQ values

To clarify greedy action selection, consider the example from Figure 4.7. A crisp state matches two fuzzy states $[Lots\_Money, Many\_Auctions]$ with degree 0.7

[14] Both $\hat{S}1_b$ and $\hat{S}1_d$ match the fuzzy state S1.

and $[Little\_Money, Few\_Auctions]$ with degree 0.4. Each of these two fuzzy states has 2 rules associated with them. For the state $[Lots\_Money, Many\_Auctions]$, the greedy action will be to *Bid_High*, since that rule has the highest $FQ(\hat{s},\hat{a})$ value. Similarly, for the state $[Little\_Money, Few\_Auctions]$, *Bid_Low* will be selected. The fuzzy actions are now defuzzified to obtain a crisp output. *Bid_High* is translated via some defuzzification function as bid 8 units and *Bid_Low* as bid 4 units. Thus the actual action taken is calculated as follows:

$$a = \frac{((0.7x8) + (0.4x4))}{(0.7 + 0.4)} = 6.5$$

In summary, where FQ Sarsa chooses an action based on selecting the most fit fuzzy state ($\max \mu$) for the current s, and then choosing the action with the highest FQ value; Fuzzy Q-Learning and Fuzzy Sarsa use *all* fuzzy states with a $\mu > 0$ to suggest an action. Each fuzzy state suggests an action based on the highest FQ value and then all suggested actions are combined using Centre of Mass. A further example is given in Figure 4.8.

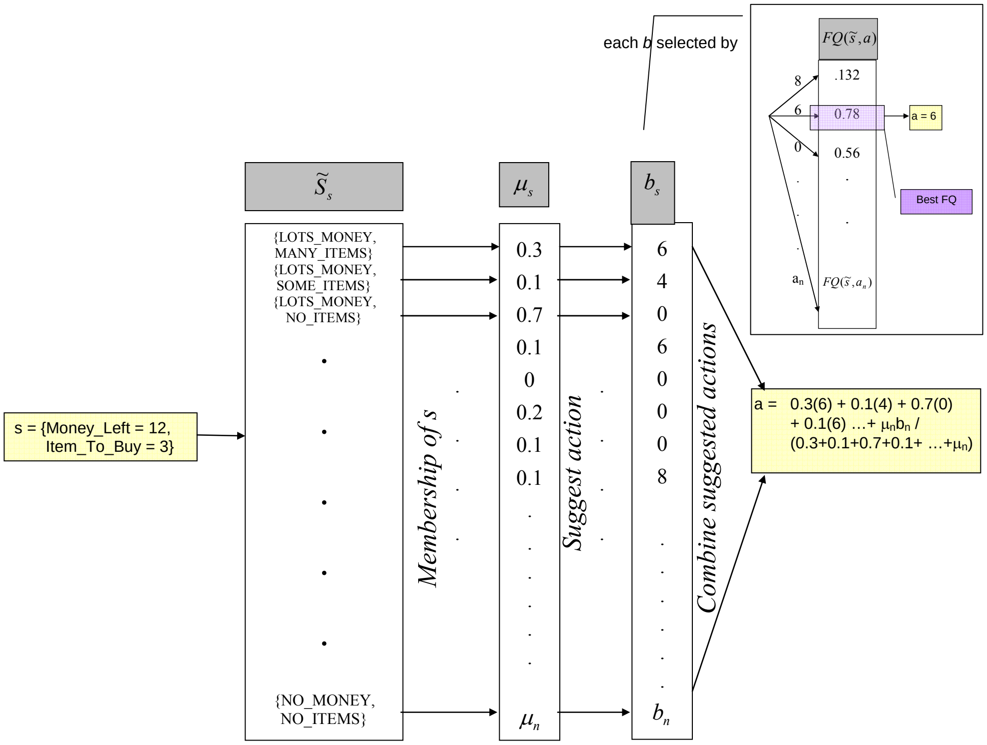


Figure 4.8: Fuzzy Sarsa Action Selection

Previously in this section, the description of fuzzification and action selection was applicable to both Q-Learning and Fuzzy Sarsa. For Fuzzy Sarsa, the update formula of Bonarini's Fuzzy Q-Learning given in Section 2.6.3.2 is updated by the author of this thesis as follows:

$$\begin{aligned} FQ\ (\hat{s}^i_{t-1},\hat{a}^i_{t-1}) &= FQ(\hat{s}^i_{t-1},\hat{a}^i_{t-1}) \\ &\quad + \alpha\xi_{(\hat{s}^i_{t-1},\hat{a}^i_{t-1})}(r_t + \gamma\sum_{\forall j} FQ(\hat{s}^j_t,\hat{a}^j_t)\xi_{(\hat{s}^j_t,\hat{a}^j_t)} - FQ_{t-1}(\hat{s}^i_{t-1},\hat{a}^i_{t-1})) \end{aligned} \tag{4.3}$$

for all i, where $\hat{s}^i_{t-1} \in \hat{S}_{t-1}$ and $\hat{S}_{t-1}$ is the set of all fuzzy states with $\mu > 0$ for the crisp state s_{t-1}. $FQ\ (\hat{s}^i_{t-1},\hat{a}^i_{t-1})$ is the value of being in the fuzzy state $\hat{s}^i_{t-1}$ and suggesting a fuzzy action $\hat{a}^i_{t-1}$. $\xi_{c^i_{t-1}}$ is the fuzzification factor, or the degree of belonging (μ) of the crisp state s_{t-1} to the fuzzy state $\hat{s}^i_{t-1}$. This is calculated as:

$$\xi_{(\hat{s}^i_{t-1},\hat{a}^i_{t-1})} = \frac{\mu_{(\hat{s}^i_{t-1})}}{\sum\limits_{i=1..n}\mu_i}. \qquad (4.4)$$

To avoid confusion in notation, the algorithm proposed by the author of this thesis uses $\hat{s}^i_{t-1},\hat{a}^i_{t-1}$ as the fuzzy rule, rather than Bonarini's $\bar{r}$ because in this notation a fuzzy state and suggested action is the definition of a fuzzy rule and r is already used in reference to the reward. In Q-learning, Q is updated using the largest possible reward (or reinforcement) from the next state, whereas in Sarsa, Q is updated with the value of the actual next state action pair as defined by the current policy.

Therefore the primary change from the Fuzzy Q-Learning algorithm is in the future contribution section. Fuzzy Q-Learning's future contribution section is defined as:

$$\gamma \sum_{\forall \hat{s}^j_t \in \hat{S}_t} \max FQ(\hat{s}^j_t,\hat{a})\xi_j \qquad (4.5)$$

for all FQ values where $\forall \hat{s}^j_t \in \hat{S}_t$, where $\hat{S}_t$ is the set of fuzzy state where $\mu > 0$ for the next crisp state s_t, and $\hat{a}$ is the action that the highest FQ values proposes.

In order to change this algorithm from off-policy to on-policy, rather than take the max of future rewards, the algorithm proposed by the research in this thesis uses the next set of $FQ(\hat{s}^j,\hat{a}^j)$ values. Fuzzy Sarsa's future contribution section is described as:

$$\gamma \sum_{\forall \hat{s}^j_t \in \hat{S}_t} FQ(\hat{s}^j_t,\hat{a}^j_t)\xi_j \qquad (4.6)$$

This is done for all FQ values where $\forall \hat{s}_t^j \in \hat{S}_t$, where $\hat{S}_t$ is the set of fuzzy state where $\mu > 0$ for the next crisp state s_t and the suggested action $\hat{a}_t^j$ is the action that would be applied using the current policy.

The Fuzzy Sarsa algorithm was originally presented by the author of this thesis in [TBC04]:

```
All $FQ(\hat{s},\hat{a})$ values initialised.
 Repeat for each episode (or auction game){
 Initialize $\hat{s}_t$ (start state for the auction game).
 Choose $\hat{a}_t$ from $\hat{s}_t$ by calculating the centre of mass using
  all $\hat{s}_t$ that match crisp $s$ and $\hat{a}_t$ following $\varepsilon$ greedy
  selection policy.
 Repeat for each step(auction) in the episode(auction
  game){
   Take action $a_t$, observe $r$ and ŝ$_{t+1}$
   Choose â$_{t+1}$ from $\hat{s}_{t+1}$ using $\varepsilon$ greedy selection policy for all
   $\hat{s}_{t+1}$match $s_{t+1}$.
   For all $\hat{s}_{t-1}^i \in \hat{S}_{t-1}$
      $FQ\ (\hat{s}_{t-1}^i,\hat{a}_{t-1}^i) = FQ(\hat{s}_{t-1}^i,\hat{a}_{t-1}^i) +$
                  $\alpha\xi_{(\hat{s}_{t-1}^i,\hat{a}_{t-1}^i)}(r_t + \gamma\sum_{\forall j} FQ(\hat{s}_t^j,\hat{a}_t^j)\xi_{(\hat{s}_t^j,\hat{a}_t^j)} - FQ_{t-1}(\hat{s}_{t-1}^i,\hat{a}_{t-1}^i))$
    $\hat{s}_t$ = $\hat{s}_{t+1}$, â$_t$ = â$_{t+1}$
   }
}
```

Figure 4.9: Fuzzy Sarsa Algorithm

The experiments presented in the research conducted for this thesis used an ε-greedy action selection policy. Regardless of what action selection mechanism is employed, it is not immediately clear how the algorithm should proceed in the exploratory case. In the crisp version of Sarsa, the exploratory action is chosen, say bid 8 units, and then the state/action pair corresponding to the current state and bid 8 is used directly in learning. As discussed earlier, in fuzzy learning the crisp state matches *n* fuzzy states. Therefore, there are two possible ways of making an exploratory move.

The first way is that for each match made, a random move is generated and then the centre of mass of all the random moves is calculated to determine the actual action. The second way is to make a random move instead and consider the set of state/action pairs to be updated the set of *all matching* fuzzy state/action pairs $(\hat{s}_t, \hat{a}_t)$, where $\hat{a}_t$ is the fuzzified crisp action, bid 8.

Since the agent is trying to learn the specific action required with regards to the total set of matching fuzzy states, the second method of exploratory action selection is used. Although only empirically tested, early experiments using both of these two methods indicated that the first method tends to cause instabilities in convergence. The remainder of fuzzy action selection is relatively straightforward: if a greedy action is taken, the algorithm observes the results and updates all fuzzy state/action pairs that contributed to the selection of $\hat{a}_t$.

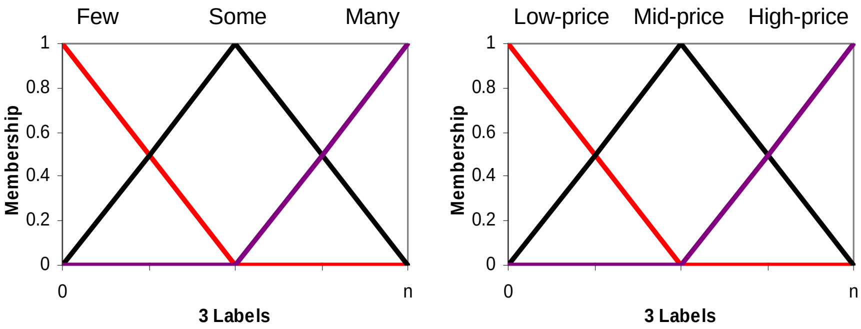
For example, in the random case $[\hat{s}1_a, \hat{s}1_b]$ matching the current state and the random action $\hat{a}_3$ being taken, the algorithm updates $[(\hat{s}1_a, \hat{a}_3), (\hat{s}1_b, \hat{a}_3)]$. If however, a greedy action is taken, then the action taken is calculated as the centre of mass of the actions proposed by $[\hat{s}1_a, \hat{s}1_b]$. Suppose $\hat{s}1_a$ proposed $\hat{a}_1$, and $\hat{s}1_b$ proposed $\hat{a}_3$ and that the centre of mass calculation returned $\hat{a}_2$. The pairs that are updated in the greedy case are the *contributing* pairs, i.e. $[(\hat{s}1_a, \hat{a}_1), (\hat{s}1_b, \hat{a}_3)]$. After the update is completed, the world is in a new state, and the algorithm repeats the above process.

4.3 Marketplace

The following section presents the results of the initial investigation into the different fuzzified Sarsa algorithms. Sarsa, FQ-Sarsa and Fuzzy Sarsa were implemented in an agent marketplace designed as discussed in Section 1.1. In the case of the Sarsa algorithm, the state of the world was considered to consist of 3 major categories*: Money_Left, Auctions_Left and Items_Left*. Actions included bids ranging from the offer price to the agent's maximum price and abstaining. Fuzzy states consisted of the same state categories as Sarsa. However, rather than storing the crisp representation of the state, states are stored as fuzzy labels rather than discrete values. The research presented in this thesis initially used four labels for each fuzzy category. Since membership functions are more robust when additive, $\sum_{i=1,n} \mu(x) = 1$, the functions used were triangular[15]. Triangular membership functions are popular and easy to use to design additive functions.

Confirmation of the robustness of additive membership functions came from the results of an earlier experiment using non-triangular and non-additive membership functions. During this test, the fuzzy algorithms were not able to find a solution, let alone an optimal one. All state variables are fuzzified according to the general membership functions given in Figure 4.10. FQ Sarsa does not use the action membership function since it utilises crisp bids.

[15] Other types of membership functions include trapezoidal, Gaussian and generalised bell.



State *Action*

Figure 4.10: Fuzzy Membership Functions for the Test bed

For the following sections, unless otherwise stated the following settings apply to all learners:

1. The exploration (ε) and learning (α) rates are both annealing parameters. Both annealing parameters anneal at the rate of:

$$p - \frac{p}{\rho} \tag{4.7}$$

   where p = {ε or α}, and ρ = 5. The annealing parameter continues to decay until it reaches 0.01.

2. γ = 0.1.

In the experiment, all games were played with two agents; a fixed strategy agent, and a learning agent. A *Zero Level Seller* is defined as following a fixed strategy for the entire game (see [HW98]). The seller allocates items to auctions with a fixed policy. The quantity q auctioned by the seller agent at each time interval is fixed and constant. Thus the seller agent holds q auctions in each interval to sell q items.

In the marketplace world initial testing is conducted by fixing the policy of any competitor agents. A policy π is fixed or a *stationary strategy* if $\pi_e = \overline{\pi}$ for all e (episodes) and $\overline{\pi}$ is the original policy. Stationary strategies are useful in domains where there is more than one agent interacting with the environment.

A *Zero Level Buyer Agent* also follows a fixed strategy for the entire game. For the agent marketplace there are two stationary strategies used to test the learning agents. These basic strategies are:

- *Greedy Bidding Strategy:* This agent bids its' maximum price immediately and continues to bid at its' maximum price in every auction thereafter until it has bought the number of items required.

  ```
  If ($I_{NEEDED}$ > I) $BID_t$ = $P_{MAX}$
  else BID = 0
  where P equals the price to bid.
  ```

- *Linear Bidding Strategy:* As time passes the maximum price this agent is willing to bid to, increases in a linear fashion.

  ```
  If ($I_{NEEDED}$ > I) $BID_t$ = $P_{MIN}$ + $INC_t$
  else BID = 0
  where $INC_t$ = $(P_{MAX} - P_{MIN)}A_t/A$. I.
  ```

ε and α are both annealing parameters and γ is set to 0.1.

Section	Purpose	Experiment Setup
4.3.1	This set of experiments compares Fuzzy Sarsa, FQ Sarsa and Sarsa in small agent marketplace auction games.	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *fixed*.
4.3.2	This set of experiments compares Fuzzy Sarsa, FQ Sarsa and Sarsa. These experiments detail the relative performance of each algorithm against a stationary strategy in an agent marketplace.	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *fixed*. ✓ Algorithms learn against a *stationary* strategy
4.3.3	This set of experiments compares Fuzzy Sarsa, FQ Sarsa and Sarsa. These experiments detail the relative performance of each algorithm against a non-stationary strategy in an agent marketplace.	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *fixed*. ✓ Algorithms learn against each other (*non-stationary* strategy)

Figure 4.11: Experiment Table for Section 4.3

The setup described in this section is applicable to all experiments done in Section 4.3. A summary of the experiments performed for the remainder of this section is given in Figure 4.11.

4.3.1 Fuzzy Label Partitions

In test 1 of the algorithms, each agent must obtain 2 items over the 4 auctions in the episode. In this test, the price of each item ranges from 5 to 6. The results presented in Figure 4.12 are the average over 20 auction games.

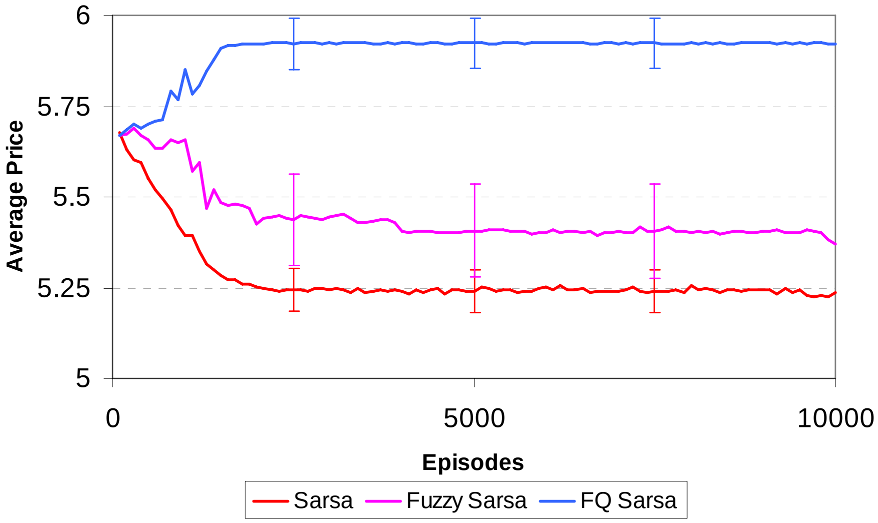


Figure 4.12: Learners vs. Fixed Linear Strategy in a Very Small Scale Auction Game

As seen from Figure 4.12, all three algorithms converge upon a solution at similar rates. The difference in the algorithms can be seen from the quality of solution found. Both FQ Sarsa and Fuzzy Sarsa find solutions that are somewhat worse than that of Sarsa. However, since fuzzy algorithms maximise learning around boundaries [BON98] and if the boundaries themselves do not represent a significant enough portion of the items, then the value of the solution may be affected. In test 1, the range of items to buy and the range of prices is less than or equal to the range of fuzzy labels used, and therefore do not represent appropriate partitions in the fuzzy label boundaries.

To determine if this was the case a second test was performed leaving all parameters constant except price, which now ranges from 5 to 8, and number of items to buy, which was increased to 4. Both changes if parameters are necessary in order to offer a range of values which are greater than the number of labels used.

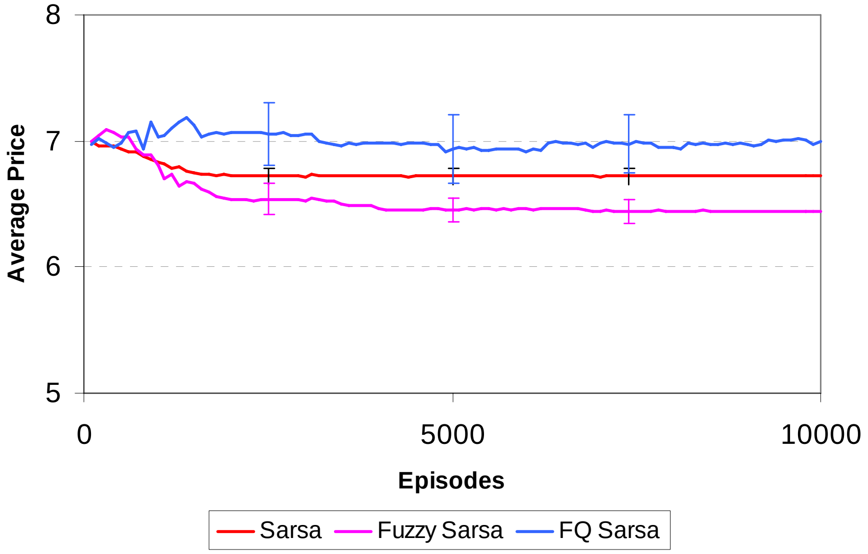


Figure 4.13: Learners vs. Fixed Linear Strategy in Small Scale Auction Game

As seen in Figure 4.13, the convergence rates remain similar to test 1. However, as suspected the value of the solution found for Fuzzy Sarsa and FQ Sarsa is closer to that of Sarsa. The original tests involving the fuzzy boundary problem were presented by the author of this thesis in [TBC04]. However, the original work with the fuzzy algorithms utilised 4 labels rather than 3. Therefore, in that work the small game size was adequate for highlighting this issue. For 3 labels, it is only in the very small game that this issue is apparent.

The reason for representing this work with 3 labels is that results presented in Section 5.1.1, indicated that the optimal number of labels for the marketplace situation is 3. To validate that nothing else about the fuzzy algorithms has been altered due to the change in number of labels, and to offer further validation of the fuzzy boundary problem solution, the Small test is rerun using 4 Labels.

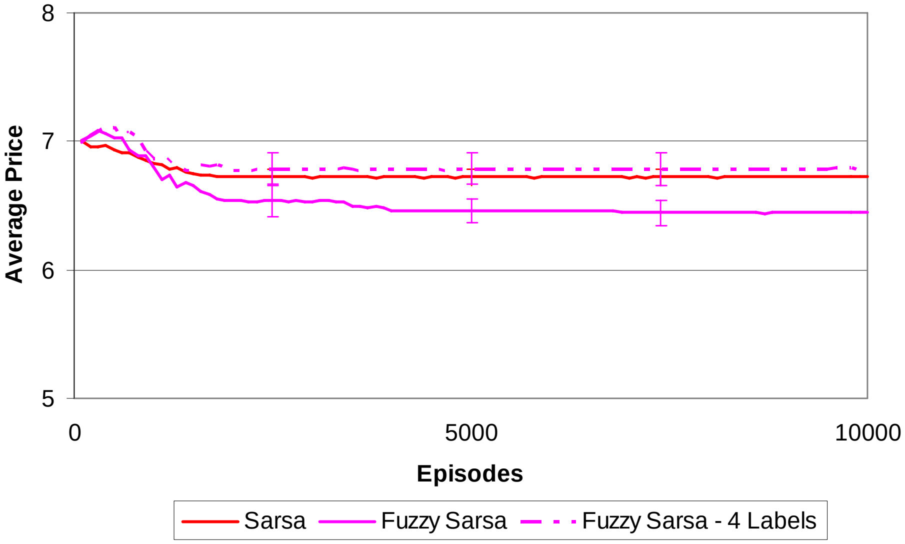


Figure 4.14: 3 and 4 Label Fuzzy Sarsa solutions in Small Scale Auction Game

Figure 4.14 shows that the 4 label Fuzzy Sarsa learner performs similarly to the Sarsa learner, while the 3 label learner finds a significantly better solution than either. Fuzzy Sarsa produces unpredictable and less optimal results when the number of fuzzy labels used to encode information is greater than the number of actual labels. For example, unpredictable behaviour is observed if there are three fuzzy labels, LOW, MEDIUM, and HIGH, and only two possible crisp values.

In these small game size tests, all three algorithms perform similarly.

4.3.2 Stationary Strategy Algorithm Performance

The tests presented so far deal with small state spaces. To fully compare the capabilities of the algorithms further tests are required in larger games. All tests presented in this section are against a linear fixed strategy agent. The first test conducted is in a large game size simulation. For this test, the number of items each agent must obtain is increased to 10 items and the number of auctions to 20 (price remains the same at 7 to 12).

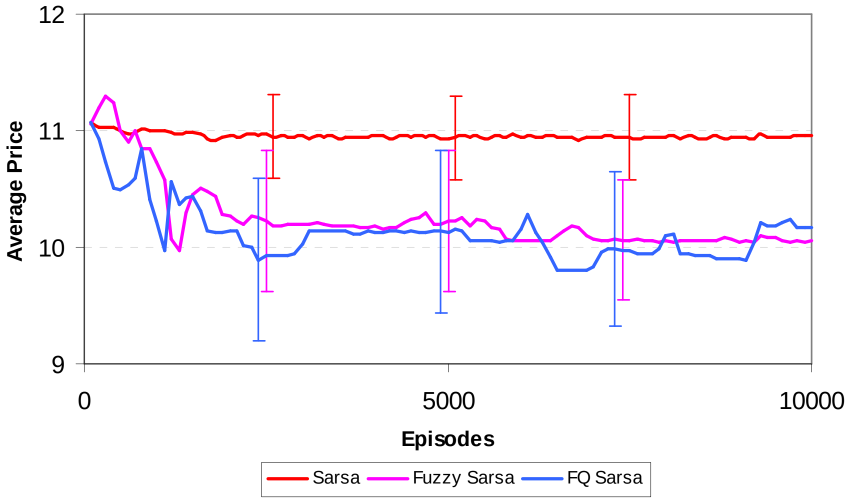


Figure 4.15: Large Game – Fixed Strategy Test

As observed from Figure 4.15, both FQ Sarsa and Fuzzy Sarsa now converge to a solution quicker than Sarsa. As a result of the increased state space during this test, Sarsa has a tendency to get caught in a local minimum if it does not come across a good solution during the exploratory stage of this algorithm. Furthermore, it is seen that the value of the solution found by Fuzzy Sarsa and FQ Sarsa is superior to the one found by Sarsa. In comparison with the results presented by the author of this thesis in [TBC04], the change in the number of fuzzification labels has indeed improved the performance of both FQ Sarsa and Fuzzy Sarsa. In fact, FQ Sarsa now seems to perform as well as Fuzzy Sarsa. To determine if there is any added benefit to the fully fuzzy approach of Fuzzy Sarsa, a further test was run in a VLarge[16] game.

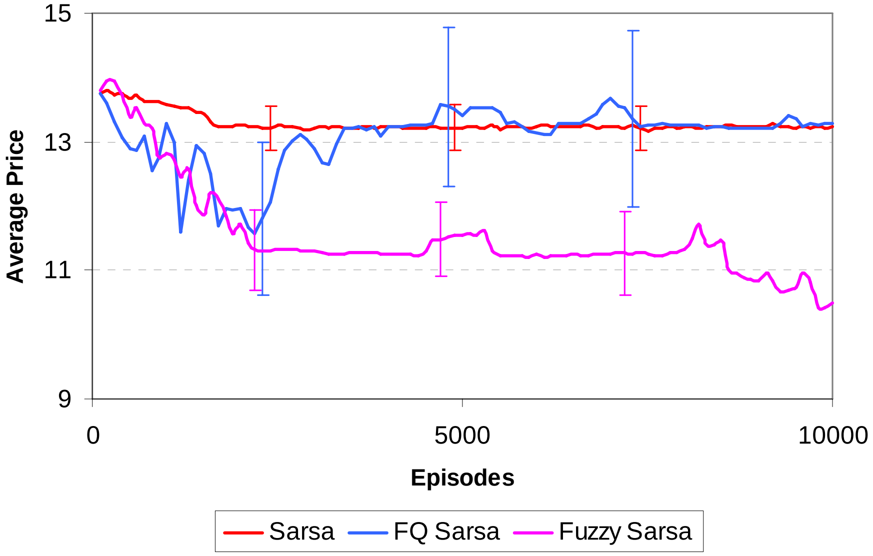


Figure 4.16: VLarge Game – Fixed Strategy Test

As expected, Fuzzy Sarsa finds a better solution than either FQ Sarsa or the Sarsa. The reason for this is apparent: while FQ Sarsa converges quicker, and in small games performs similarity to Fuzzy Sarsa (because it has a much reduced state space without the fuzzy action combination of Fuzzy Sarsa) its representational powers fail and it mimics the solution found by Sarsa. This confirms that the purer fuzzy solution presented by Fuzzy Sarsa, does seem to maximise transitions along fuzzy borders, allowing it to converge quicker and find a better solution than either Sarsa or FQ Sarsa.

[16] The complete details for all game size settings can be found in Figure 3.2.

4.3.3 Non-Stationary Algorithm Performance

In order to determine the flexibility of the learning agents, they were next put into direct competition with each other. For this test, all parameters (auction size, start money, rewards, etc) are kept the same as in the first test. Each learning algorithm was tested against the other competitive algorithms in turn. The results presented in Figure 4.17 are the averaged price achieved in the final 2000 of 10 trials.

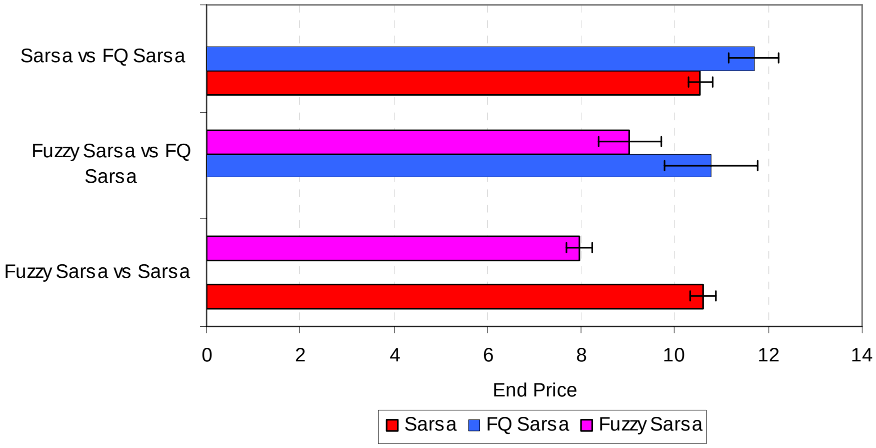


Figure 4.17: Large Game - Direct Competition Test

As shown in Figure 4.17, given the same learning parameters, Fuzzy Sarsa achieves a superior price than its competitors when in direct competition with either the Sarsa algorithm or the FQ Sarsa algorithm. The reason the Fuzzy Sarsa agent in the Fuzzy Sarsa vs. Sarsa game achieves a better price than that of the agent in the Fuzzy Sarsa vs. FQ Sarsa game is explained by Sarsa's inability to explore large state spaces sufficiently under these conditions. FQ Sarsa, because of its reduced state space, is more able to react to the Fuzzy Sarsa algorithm. In the Sarsa vs. FQ Sarsa test, it is interesting to note that Sarsa and its reduced state version, FQ Sarsa achieve an almost identical end price. However, the FQ Sarsa algorithm is more prone to oscillation due to the reduction of its state space. As shown in Figure 4.18, this oscillation causes the algorithm to start to fail. The average success rate for even FQ Sarsa's ability to find a solution, irrespective of the solution quality, has fallen to a final success rate of 78%.

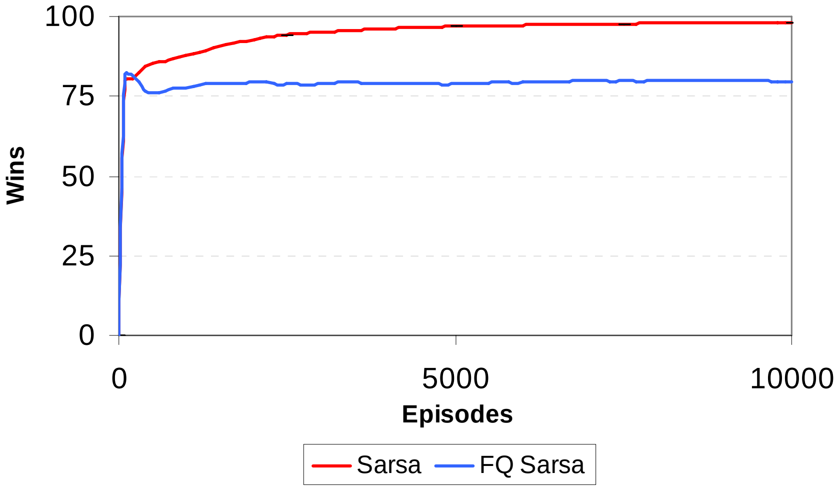


Figure 4.18: Large Game Convergence – Sarsa vs. FQ Sarsa

To further validate these results, the test was rerun in the VLarge game.

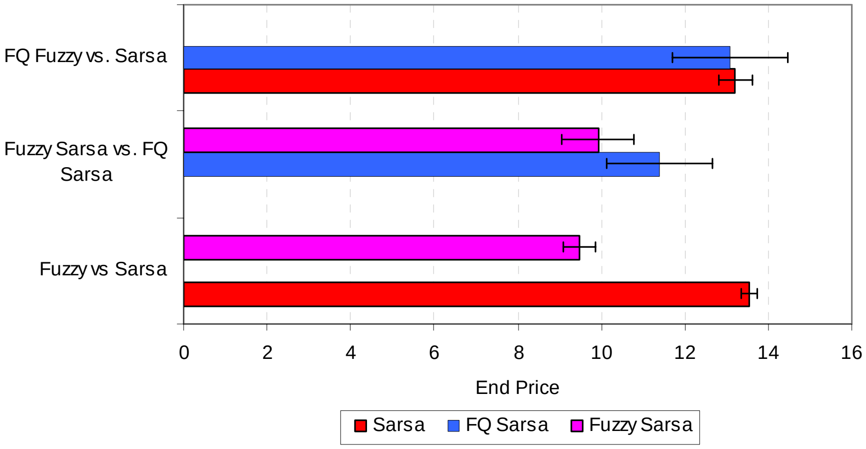


Figure 4.19: VLarge Game - Direct Competition Test

Figure 4.19 presents the results of direct competition in a VLarge game. The results confirm those presented for the Large game in Figure 4.17.

In the final test, all 3 learning algorithms were evaluated in the same game. Each agent must still win the required number of items; however, in order to provide the same framework as the previous test, the number of auctions must be increased. For example, in the Large game, each agent must win 10 items, when 3 agents are participating in the auction 30 items are available.

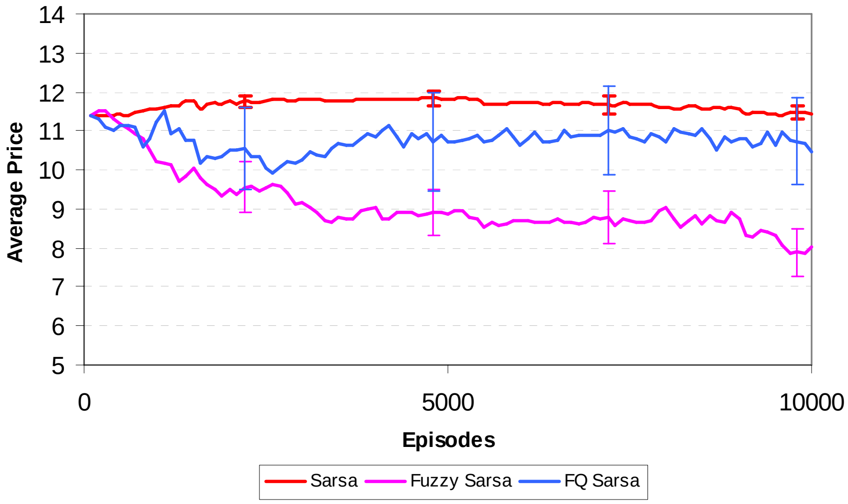


Figure 4.20: Sarsa vs. FQ Sarsa vs. Fuzzy Sarsa in Large Auction Game

In this final test, it is clear that Fuzzy Sarsa once again is the most optimal and flexible of the three algorithms. When competing directly with either of the other two algorithms it is able to consistently achieve a better price with minimal variability in end price. These results are significant because they demonstrate Fuzzy Sarsa's ability to learn effectively against a moving target, the other two learning agents. Under these settings the generalisation powers of Fuzzy Sarsa enable it to quickly take advantage of the current market conditions. Although not presented here for the sake of brevity, Fuzzy Sarsa experiences none of the convergence difficulties that FQ Sarsa encounters, as demonstrated in Figure 4.18.

4.4 Summary

Section 4.1 and 4.2 presented two novel on-policy fuzzy reinforcement learning algorithms. Section 4.3 has presented the results of initial testing in the marketplace domain:

The tests presented in Section 4.3.1 indicated a relationship between the number of fuzzy labels used and the size of the data set. The result show that the number of fuzzy labels used should be greater or equal to the actual data. However, it opens the question of how many fuzzy labels are appropriate. Is there a relationship between the number of labels used and generalisation abilities?

Section 4.3.2 and 4.3.3 presented experiments regarding the general capabilities of the fuzzy algorithms with respect to the tabular on-policy algorithm Sarsa. Specifically these tests indicated that as the state space increases, the purer fuzzy logic approach to reinforcement learning presented in Fuzzy Sarsa allows for a more robust and correct solution than the reduced state space algorithm presented by FQ Sarsa and than the traditional on-policy learning of Sarsa. This improvement is seen in both games against stationary learners (Section 4.3.2) and games that put the learning agents into direct competition with each other (Section 4.3.3).

This is an important result since Fuzzy Sarsa works with a significantly smaller state space than Sarsa. However, the results presented in this section only deal with one domain, does Fuzzy Sarsa perform similarly in other domains?

Furthermore, as presented in Section 2.6, there are other types of function approximation algorithms that are also capable of dealing with reduced state spaces and also with generalisation. How significant is Fuzzy Sarsa's performance in comparison?

5 Comparison of Function Approximation Techniques

After the experimentation in the marketplace with the fuzzy algorithms, two major issues became apparent:

1. While the fuzzy algorithms outperformed tabular Sarsa, this comparison was not deemed fair since the fuzzy algorithms deal with function approximation and Sarsa does not.

2. The fuzzy algorithms perform reasonably well in the Marketplace domain, but could it be extended to work in other domains, including domains with continuous state variables?

To address the first issue, another type of function approximation technique was required. After examining the function approximation techniques discussed in Section 2.6, tile coding was identified as a good candidate for comparison with fuzzy. There are two motivating factors behind this decision. First, the technique of overlaying the state space with tilings seemed intuitively similar to fuzzy membership functions and second, researchers using tile coding combined with Sarsa have recently reported a fair amount in implementations ranging from the mountain-car problem [SS96, SUTT96], the complex task of robotic soccer [SSK05] and stochastic games [BV02, BV02a]. Consequently, gradient descent Sarsa(λ) with tile coding appears to be a reasonable algorithm to investigate both its performance in large state space problems, but also its ability to generalise. Therefore, the gradient descent Sarsa(λ) with tile coding presented by Sutton [SUTT96] was chosen as the third test algorithm.

To address the second issue, the mountain-car domain and predator/prey gridworld domain simulators were introduced. The mountain-car domain was identified as the first test for the fuzzy algorithms, since the gradient descent Sarsa(λ) with tile coding had already been implemented [SuttonMC] and the results, [SUTT96] published. This domain served as a control domain for the gradient descent Sarsa(λ) with tile coding. The predator/prey world was chosen because of its differing game dynamics, in that both the predator and the prey while in competition, have fundamentally different goals. In the marketplace the goal of each competing agent is get the best price for a certain number of items, while in the predator/prey environment it is to either eat the prey, or avoid the predator while eating all the dots.

This section presents the results of implementing the three different types of function approximation algorithms in the three separate control domains described in Section 3. Section 5.1 investigates how to set up the generalisation parameters of fuzzy labels and the tile coding settings, Section 5.2 will present how the parameter settings were determined and Section 5.3 presents the results of stationary strategy tests in all three simulated domains. Of particular interest in the following experiments is: (1) Ease of implementation (2) Quality of control obtained, and (3) Scalability. Figure 5.1 summarises the experiments performed in this section. Further details are given in the relevant sections.

Section	Purpose	Experiment Setup
5.1	To determine the good generalisation parameters for fuzzy memberships and tile schemas.	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *change*.
5.2	To determine good learning parameters for each algorithm in each simulation domain.	✓ Learning parameters *change*. ✓ Fuzzy and tile schemas *fixed*.
5.3	This set of experiments compares Fuzzy Sarsa, gradient descent Sarsa(λ) with tile coding, and optionally FQ Sarsa. These experiments detail the relative performance of the algorithms in the three simulation domains.	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *fixed*. ✓ Algorithms learn against a *stationary* strategy.

Figure 5.1: Experiment Table for Section 5

5.1 Effects of Generalisation

One of the pertinent issues in both fuzzy and tile coding, is how to set up the type of generalisation that occurs. With fuzzy, generalisation is based on the design of the fuzzy membership functions. In tile coding, generalisation is primarily based on the shape and width of the tile. The following sub-sections demonstrate some of the initial work that went into designing the fuzzy memberships and tile coding schemas in each of the target domains.

5.1.1 Fuzzy Labels

5.1.1.1 Marketplace

In all three problems one of the initial issues in implementation of the fuzzy algorithms is the number of fuzzy labels that should be used in order to represent the data. The number of labels used must adequately represent the state space. For example, as depicted in Figure 5.2, in the case of the state space parameter Auction_Left, is 3, 4 or n labels sufficient granularity for describing how many auctions there are left in the game? Furthermore, is the required granularity affected by the state space of the problem?

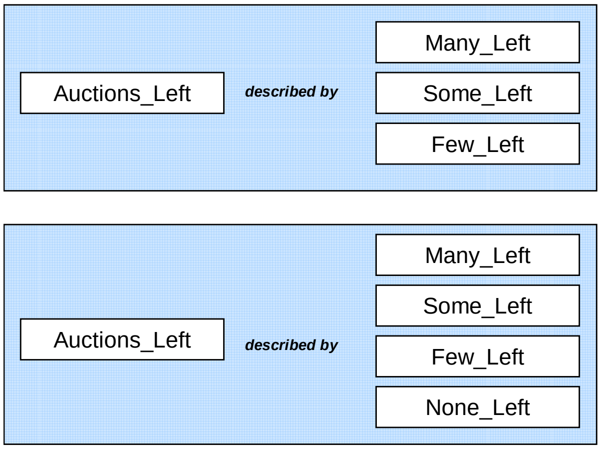


Figure 5.2: Possible Fuzzification for Auctions_Left

Researchers in control system theory advocate 3 [JAN91]. However, there seems to be no definitive recommendation for fuzzy learning systems. In order to investigate the appropriate number of labels to use in fuzzification of a learning system a variety of different label combinations are investigated in the marketplace and the mountain-car domain. In the marketplace domain the variables used are discrete whereas in the mountain-car domain they are continuous. This difference is important as it makes the state space of the mountain-car domain much larger than the marketplace.

For the marketplace domain, label combinations of 2, 3, 4 and 5 labels were investigated. In all cases, the membership functions are both triangular and additive. As a result of the greatly increased cost in the complexity of design, all items in the state space and action space for each test, all variables used in the system are fuzzified with the same number of labels as indicated in Figure 5.3.

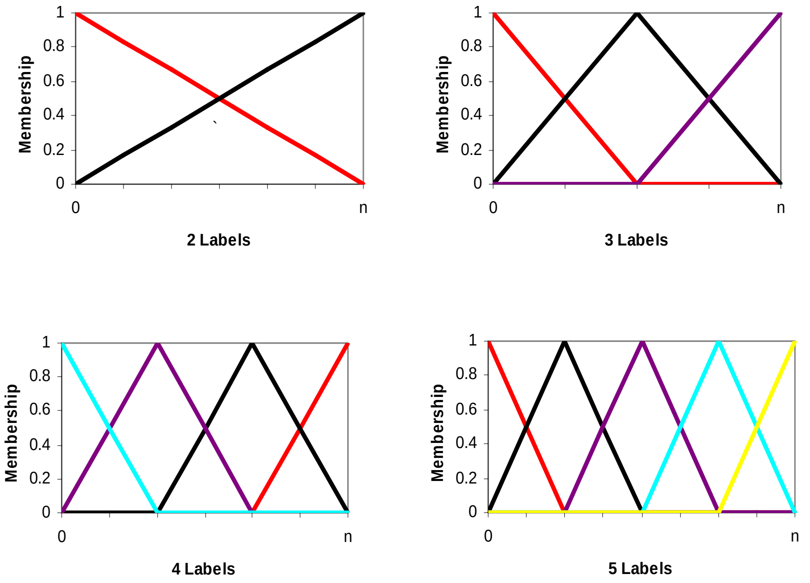


Figure 5.3: Triangular and additive membership functions for X Labels.

Three separate tests at four and five different game sizes were performed by each Fuzzy Sarsa learner (i.e. 2 label fuzzy learner, 3 label fuzzy learner, etc). The first two tests consist of the Fuzzy Sarsa learner playing an auction game against a fixed strategy agent. In the first test, Fuzzy Sarsa plays against a greedy agent, in the second against a linear agent, and in the third test, Fuzzy Sarsa plays against a Sarsa agent. The reason for conducting three separate tests is that the behaviour of each different agent competing against Fuzzy Sarsa is significantly different, and thus the game space that Fuzzy Sarsa must learn is different in each case. In all tests, each agent has enough money to buy the required number of items at the fixed maximum bid price and the allowable bids range from 5 to 25 and abstain. All other parameters, such as α, ε, and γ, remain as described in Section 3.2. The results presented are averaged over a minimum of 5 experiments and the error bars the 95% confidence intervals calculated as described in Section 3.5.

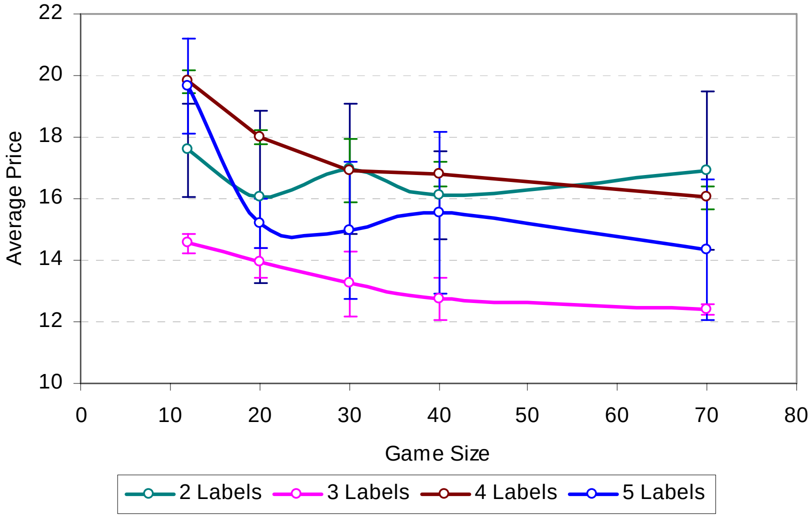


Figure 5.4: Marketplace Label Test I: Fuzzy Sarsa vs. Greedy.

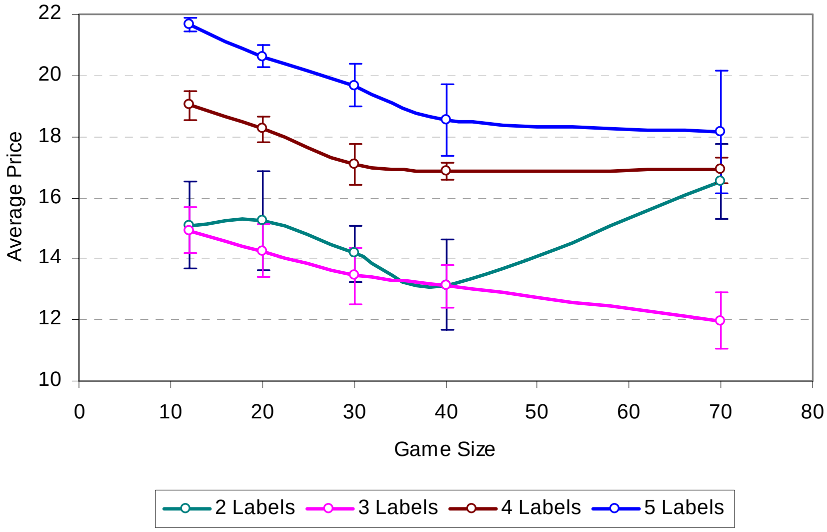


Figure 5.5: Marketplace Label Test II Fuzzy Sarsa vs. Linear.

In both test I and test II shown in Figure 5.4 and Figure 5.5, the 3 label combination appears to be the ideal choice for the marketplace game. However in test II the 2 label combination performs comparably to the 3 label combination. However, in the greedy test, the 2 label combination performed significantly worse than the 3 label combination. To determine the cause of the differing performance of the 2 label combination, consider the behaviour of each of the stationary strategies the agent plays against. Figure 5.6 illustrates the increasing bidding pattern of the linear strategy and the flat bidding pattern of the greedy agent.

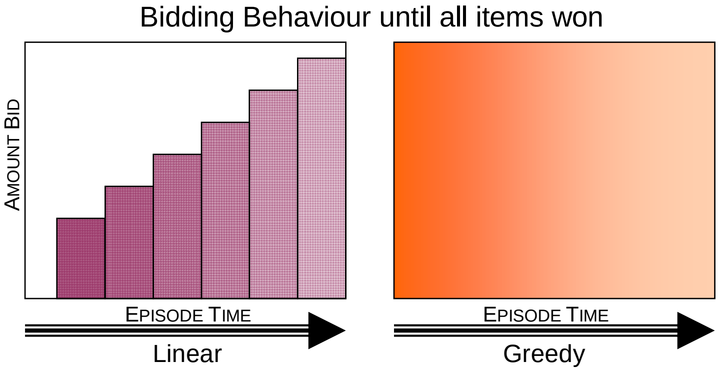


Figure 5.6: Marketplace stationary strategy behaviour

In each case, the shading indicates that the bidding behaviour continues until the stationary strategy agent has won the required items. By comparing the policy learned over the set of auctions for the 2 and 5 label combinations, it appears that although additive membership functions offer a more robust solution, the shape of the membership functions plays an important role in the optimality of the solution. In the case of the 2 label combination, the agent it is better able to play against the linear agent because the optimal competitor policy is to bid early and bid late. This bid pattern falls along the boundaries for the 2 label combinations, and thus the 2 label agent performs well against the linear strategy. Against the greedy agent, the optimal policy is to bid late and abstain early. This policy is further away from the boundaries of the 2 label combination and thus is not easy for the 2 label agent to learn. Similarly the 5 label combination is able to perform significantly better against the greedy combination because the midpoint boundary for the label representation falls at the actual midpoint of state space. As a result, the 5 label representation experiences a boost in performance when playing against the greedy strategy.

Based on the stationary strategy tests, the ideal choice so far is the 3 label combination. Not only does the 3 label combination perform significantly better than the 4 and 5 label combinations, but in both cases where there is a similar competitor, the error bounds on the solution found by the 3 label combination is considerably smaller. As a final check, a third test against a non-stationary strategy was run. Due to the size of state space in the Huge game size scenario, this test was only run to the VVVLarge game size.

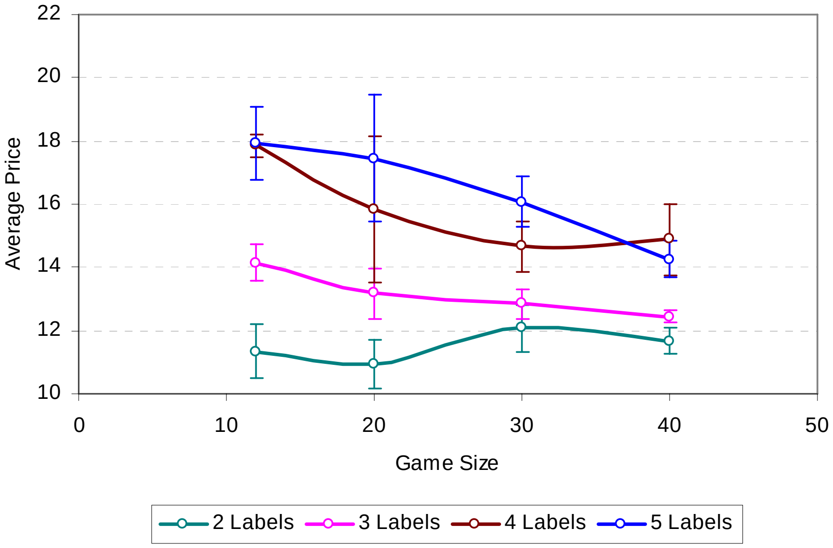


Figure 5.7: Marketplace Label Test III: Fuzzy Sarsa vs. Sarsa.

As shown in Figure 5.7, the 2 label combination actually outperforms the 3 label combination in 50% of the tests. While this result is promising, the representational abilities of the 2 label combination are limited. As seen in both stationary strategies tests (Figure 5.4 and Figure 5.5) at the Huge level (70 auctions) the 2 label significantly shifts towards a less optimal policy. In examination of the data files for the policy learnt over the set of 70 auctions rather than the average price achieved it is apparent that using 2 labels is no longer capable of a suitable representation of the state space.

Since in the majority of cases the 3 label combination is clearly better than any other combination, and because the error on the 3 label combination is much smaller than its closest competitor, the 2 label combination, further experiments in the marketplace domain presented in this thesis use 3 label membership functions.

5.1.1.2 Mountain-car World

In order to further investigate the effectiveness of the three different forms of function approximation in reinforcement learning, the three algorithms were implemented in the mountain-car world described in Section 3.4.

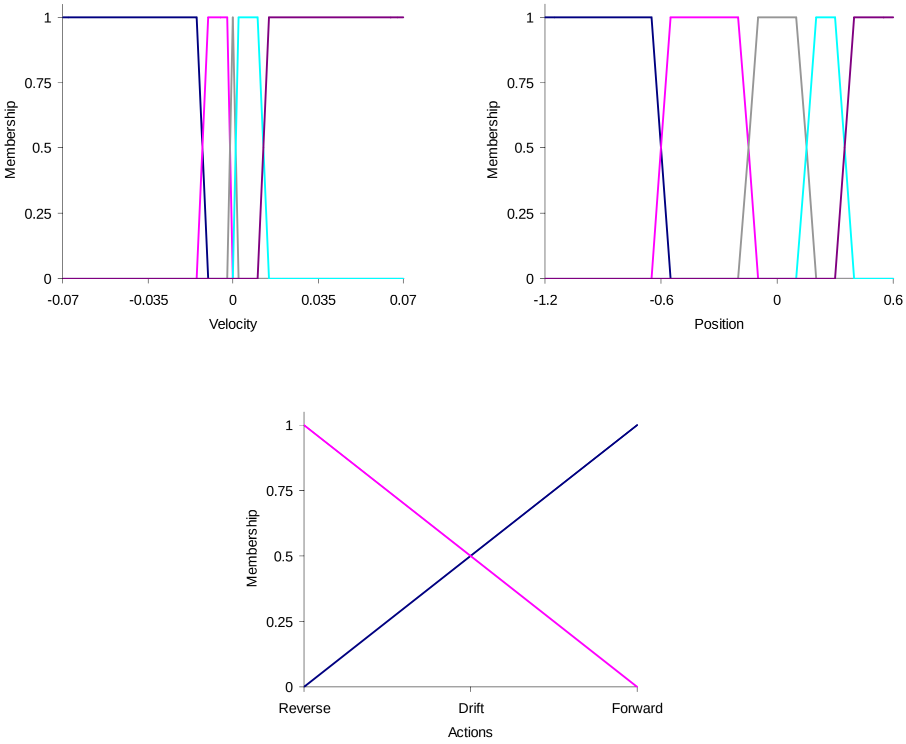


Figure 5.8: Membership functions for the Mountain-car world.

An initial study conducted by the author of this thesis indicated that 3 labels were insufficient to model this domain, with the initial attempts unable to find a solution. Furthermore, as demonstrated by the central label clustering of velocity in Figure 5.9, expert knowledge of the most frequently observed values was required. Since the majority of values in the mountain-car problem are clustered in the centre, membership function designs that did not take this into account were also unsuccessful at finding a solution. Figure 5.8 depicts the 5 label membership combination tested, with a 2 label action function. The 7 label combination has similar variable coverage, simply with more labels clustered centrally. It also uses a 2 label action function.

As illustrated in Figure 5.9, for FQ Sarsa there appeared to be little difference between the solutions found by 5 or 7 labels. In the case of Fuzzy Sarsa, the 5 label solution appears to be marginally better than the 7 label solution.

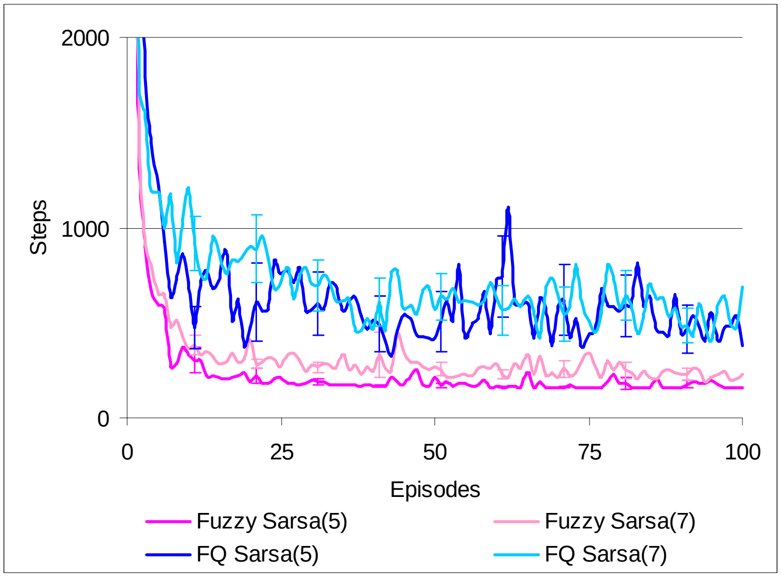


Figure 5.9: 5 and 7 Label tests in Mountain-car world.

One of the first issues in implementing the fuzzy algorithms is careful consideration of the design of the membership functions. A number of different label combinations were experimented with; the most effective for both position and velocity was found to be 5 labels.

5.1.2 How to Tile?

This section describes in detail some of the issues faced when applying tile coding techniques to the marketplace and the predator/prey gridworld. It is included to provide illustrative detail of some of problems and drawbacks of tile coding. The literature on tile coding provides some basic guidance on how to address the issues of tile shapes, tiling density and tiling width [SuttonTC and SB98]. In regards to tile shape, it appears that, like fuzzy memberships, tile shape is primarily based on expert knowledge of the system. Different tile shapes promote different types of generalisation. Since the aim is to compare the tile coding algorithm with the fuzzy algorithm, the basic tiling shape of a square was deemed appropriate for the marketplace world. This tiling shape provides a similar overlay to the additive trapezoidal membership functions used by Fuzzy Sarsa.

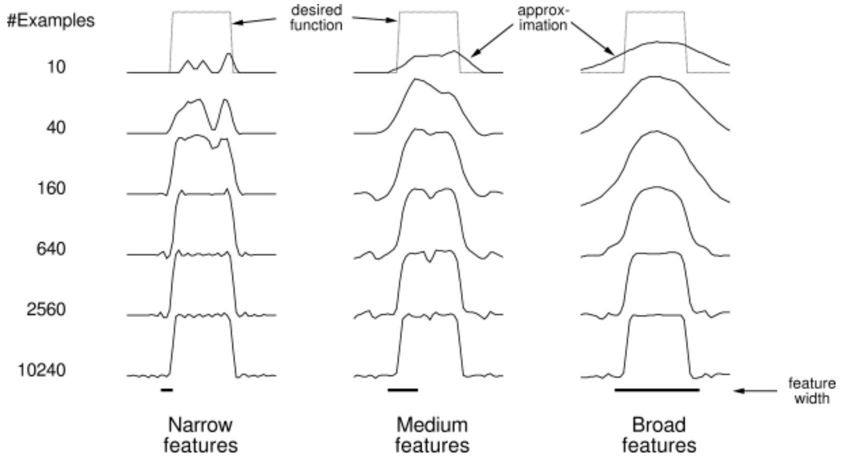


Figure 5.10: The effect of Narrow vs. Broad feature widths from [SB98]

With regard to tiling density and tiling width, [SB98] provides some insights with the example in Figure 5.10. In this example, narrow vs. broad feature width appears to have little effect on the overall function learnt (10240 examples). However, broad features have a much stronger effect on initial generalisation (10 examples). For both the marketplace and the predator/prey gridworld the way the tiles are overlaid is fixed and the resolution and generalisation parameters are altered.

5.1.2.1 Marketplace

For the marketplace domain, a tiling type of a simple grid was deemed to be most similar to the additive triangular membership functions already used by the fuzzy algorithms. Therefore each tile in the marketplace is described as $(I \bullet A \bullet M)$ where I is the number of items the agent currently has, A is the current auction number, and M is the amount of money the agent has. In order to determine the actual tiling width (generalisation) and resolution (number of tilings) settings in the marketplace, a scanning program was written to identify potentially suitable settings. This program scanned through different tiling numbers with different tiling widths. At each setting it ran one simulation of each game to 2000 episodes and compared it with the other results. The top 10 settings of two sizes of auction games were isolated and the top 6 in common were chosen to complete the full 10 game tests. In this experiment, each tiling combination competes only against the fixed strategy agent.

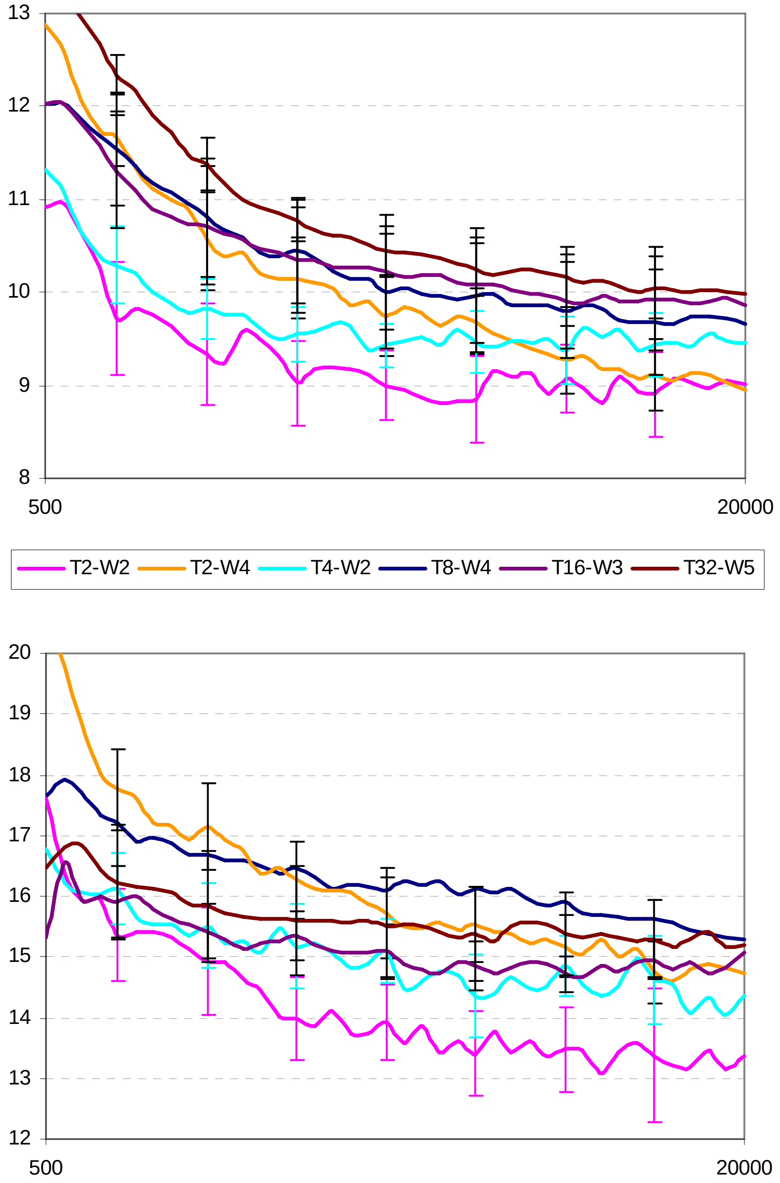


Figure 5.11: Tile Tests in Large (top) and VVVLarge (bottom) Marketplace.

Figure 5.11 presents the results of these tests in a Large and VVVLarge marketplace in a linear strategy game. T*x*-W*y* indicates the number of tilings used and their width. For example the first series labelled T2-W2 is of the tile coding agent with tile widths of 2 and the number of tilings made over the state space set to 2.

The first point to address in analysing these two experiments is the different entry points of the 6 different tile/width settings (at 500 episodes). All 6 settings are competing in the same marketplace setup, including the use of the same initial start seed. One of the advantages of function approximation methods is the ability to abstract newly learnt data to other states and this abstraction quickly affects the states visited.

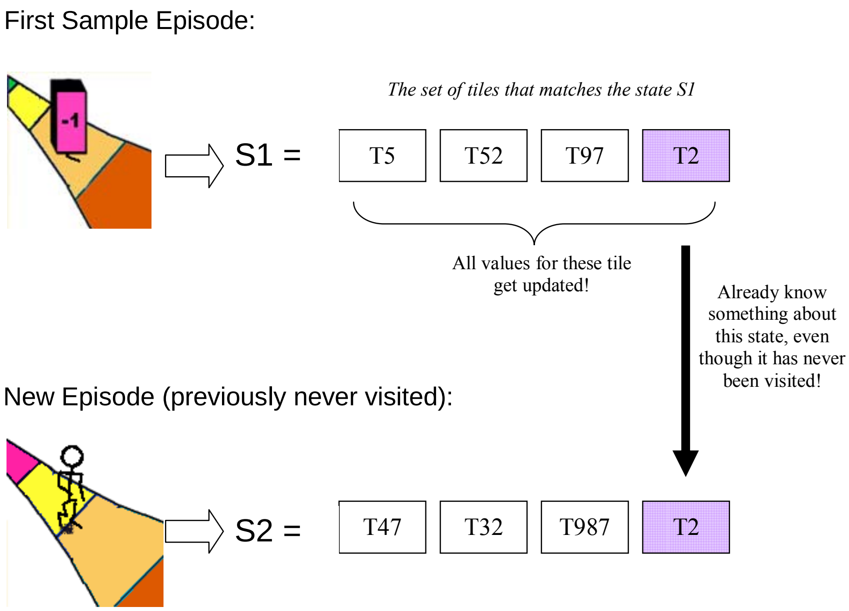

First Sample Episode:
The set of tiles that matches the state S1
-1
S1 =
T5
T52
T97
T2
All values for these tile get updated!
Already know something about this state, even though it has never been visited!
New Episode (previously never visited):
S2 =
T47
T32
T987
T2

Figure 5.12: Stickman Generalisation.

As depicted in Figure 5.12, when an agent receives a reward in a particular state, each tile that represents it receives a proportion of the reward. When the agent visits a completely new state, some of the information learnt in the original state may already be influencing it. In the case where the new state has one or more tiles in common with the previously visited state, that information will immediately affect action selection. The agent in the new state already has enough information to narrow down the possibilities in its greedy state selection. The section of state space initially affected by this generalisation is directly related to the width of the tiles overlaying the state space.

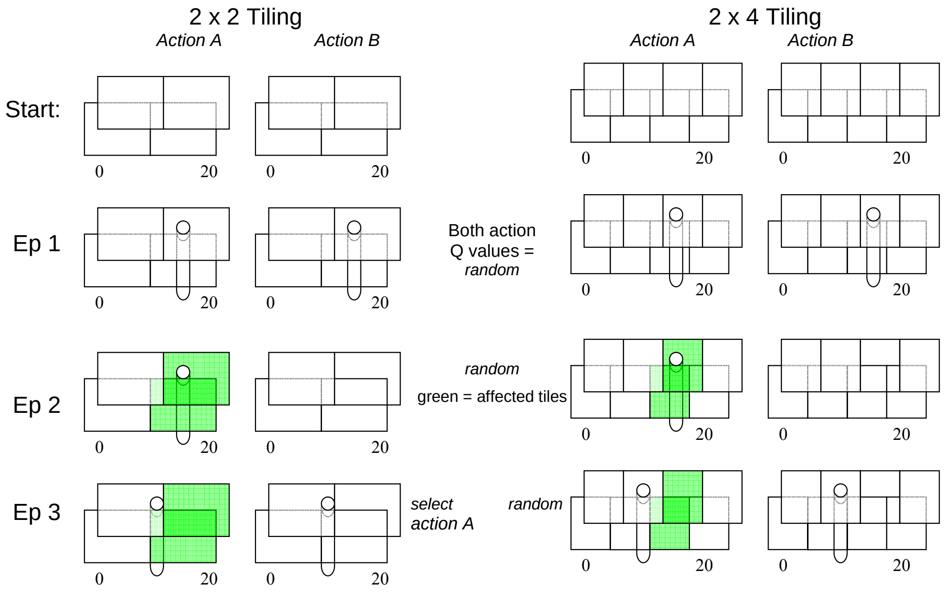


Figure 5.13: Tilings/Width Exploration Example.

Since 3-dimensional state/action selection is difficult to visualise, Figure 5.13 uses a very simplified state space example to illustrate the rapid effects on states visited as a result of differing state space overlays. The example uses only one state variable and two action variables. At the start, all tiles are initialised to their default value. In Ep 1, the current state is mapped to tiles as illustrated by the cylindrical intersections through the tiling space. There is an intersection through the tiling space for each possible action. In the 2x2 case, the state space is represented by the right most tile in both tilings. In the 2x4 case, the same state falls into the 3rd (out of 4) tile in both tilings. At this point, the greedy action would likely be chosen; however since all values are the same, a random action is chosen instead. This action is indicated by the single cylindrical intersection in the state/action space at Ep 2 corresponding to selecting Action A. The reinforced or affected tiles are highlighted in green. At Ep 3, although the start state is identical in both tiling combinations, the greedy actions chosen by each combination are already different. In the 2x2 case, the reinforced tiles already affect the greedy decision, whereas in the 2x4 case there are no affected tiles and thus a random decision is taken. In both cases, at an exploration factor of 0.03, the majority of future action selections will be greedy and thus occur around positively influenced tiles. This causes the agents to favour known solutions until exploration pushes them in another direction. This analysis agrees with the observation made by [SB98], that "*With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy.*" In the marketplace environment, the training of close neighbours at the stages analysed has a greater effect on the speed of overall movement towards an improved solution. Thus, the different starting points of the six combinations are explained by the different way each setting intersects the state space and then by the clustered exploration around the first solution found. In all cases exploration does eventually push them out of the local minima.

The final point to consider with regard to this issue is the fact that in an auction game the start state of the game is always the same. As stated, there is only a 3% chance of exploration once an initial solution is found. In the case of a coarse tiling over the state space, an exploratory move further into the game has a much greater chance of affecting the initial move, than the late exploratory move of a finer tiling. Figure 5.14 depicts the first 50 games of 4 of the combinations. The combinations have been chosen by their Width settings. The quicker move towards a smaller price indicates that the coarse width of 2 (T2W2) is greater affected by exploratory intermediary moves than the finer width of 5 (T32W5). As indicated by the confidence intervals in the figure, T2W2 also suffers from greater oscillation in its bids, and T32W5 less, as a result

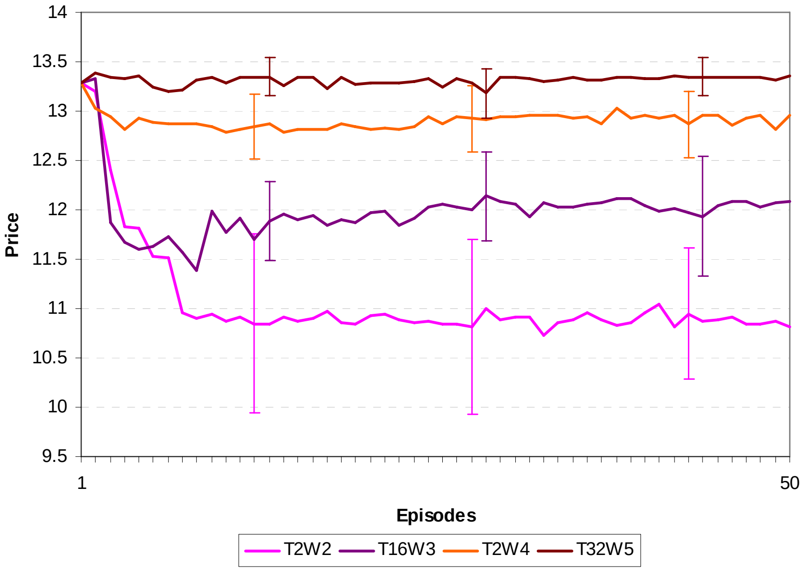


Figure 5.14: First 50 episodes of Tile Tests.

Finally in this section, based on the experiments of Figure 5.11, a tile/width combination must be chosen for further experimentation. In the VVVLarge test, the T2W2, corresponding to 2 Tilings with an overlay of 2x2x2 (or 0.5 granularity) performs better about 50% of the time, than any other tilings/width combination. In the Large test, this improvement was less marked, however it is the only tiling that appears to have any comparative improvement. A further reason for choosing such a coarse tiling is that it is expected that the ability to generalise quickly (i.e. have broad features) will be important in a coevolutionary scenario.

5.1.2.2 Predator/Prey Gridworld

Figure 5.15 shows the results of a study into tiling resolution in a gridworld. In this study, the general shape of the tiles has been fixed and the tile width and number of tilings has been altered. This experiment is run in a 5x5 tile world, with the state variables *x*: width of the current grid, *y*: height of the current grid and *t*: time. The tiles used in this world are $(X \bullet Time) + (Y \bullet Time) + (X \bullet Y \bullet Time)$. In the case of the first two combinations, because X and Y are finite (for this experiment 5, and others with the range 3 – 15), X and Y are used as is, one tile per value. This means that there is no generalisation between positions in the grid. In the final combination $(X \bullet Y \bullet Time)$, generalisation between tiles is used. The *x* axis of the surface maps indicates the width of the time tilings (g_t). For example, $g_t = 25$ results in fairly wide tilings of 25 time units each whereas $g_t = 5$ results in narrower tilings of 5 time units each. The y axis of each surface map in Figure 5.15 is the granularity of the XY tilings g_{xy}. For example, if $g_{xy} = 2.5$ the x and y state planes are divided into tiles that are 2.5 units wide. If the $g_{xy} = 1.0$, no generalization between x and y coordinates is used. For all three combinations, time is tiled.

The results presented in the surface map are the final average moves of 5 trials of each XY/time setting for 8 different tiling settings. The lighter shades indicate the better solutions and lower error.

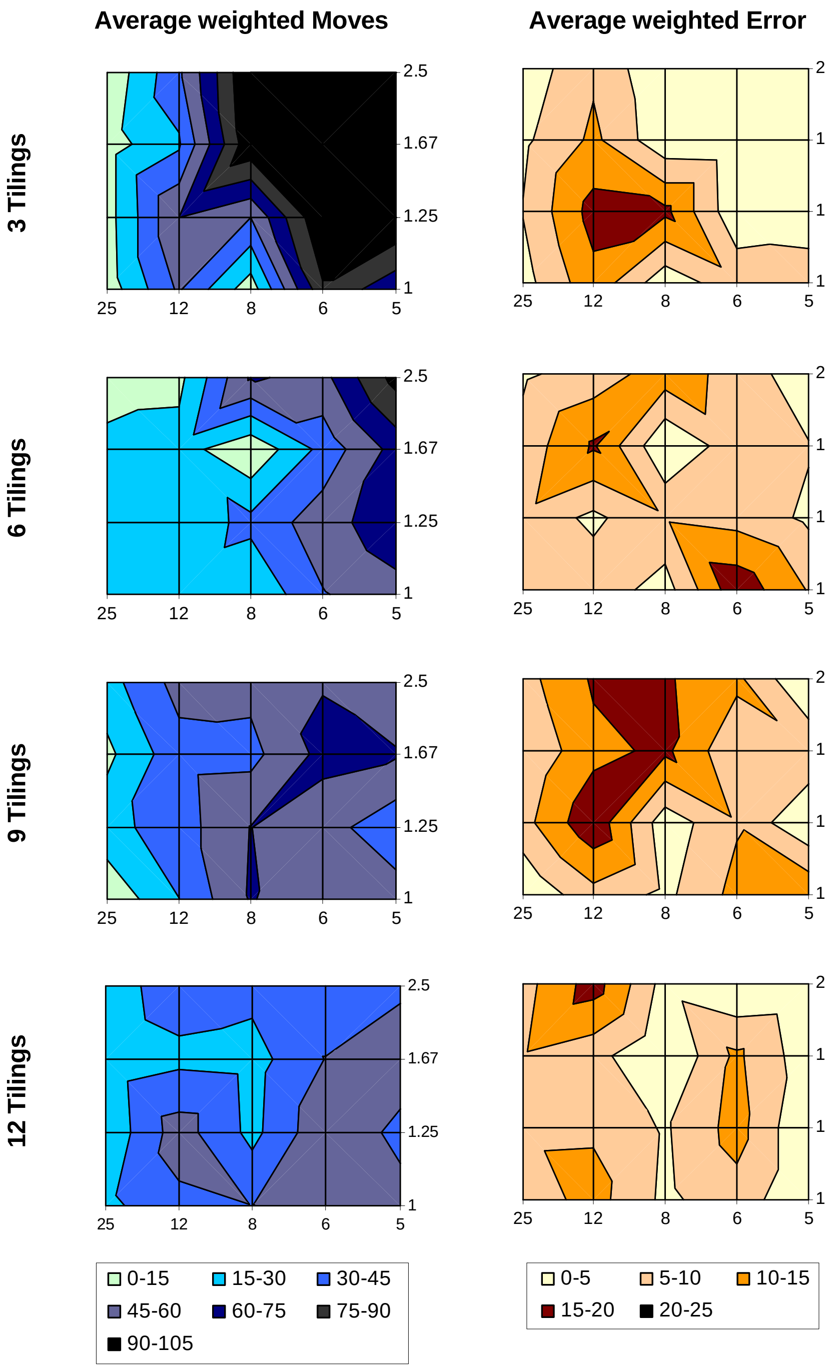


Figure 5.15: Tile Tests in Predator/prey gridworld.

The results presented Figure 5.15 are those that are useful in the current discussion. Further tests were performed at higher tiling levels, but these tests indicated that at the current settings learning was too slow to achieve a good solution. Since only 9 tilings were used in the mountain-car world, it is reasonable to expect that 12 tilings should be more than adequate for representing this domain. The full results for all tiling tests performed can be obtained in Appendix I. For all numbers of tilings it is evident that the width of the time tiles most affects performance, with the narrowest tiles ($g_t = 5$ or $g_t = 6$) being the worst performing.

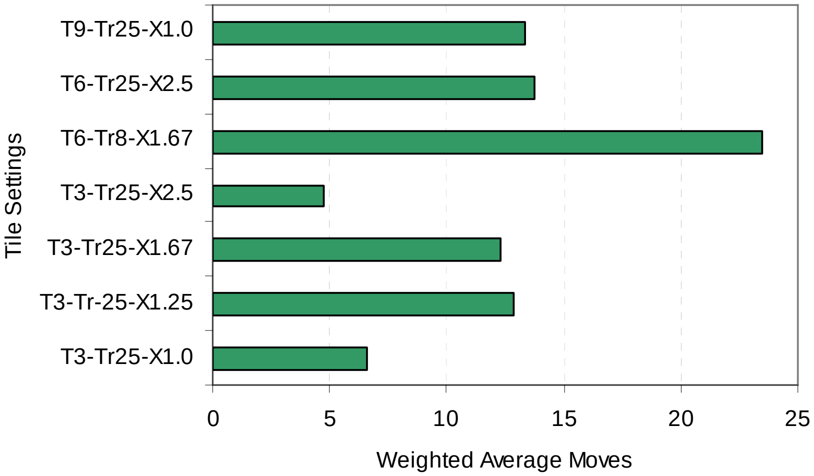


Figure 5.16: Tile Tests in Predator/prey gridworld.

From the results present in Figure 5.15, the 7 best settings were chosen for further tests. Figure 5.16 shows the weighted average moves of 10 experiments at each listed number of tilings (3, 6 or 9), g_t and g_{xy} settings. The results are weighted according to the stability of the solution – a setting that has a 95% success rate will get a lower weighted move rating than one with 65%. The tile setting of T3-Tr25-X2.5 corresponding to 3 tilings with $g_t = 25$ and $g_{xy} = 2.5$ clearly performs the best from the 7 tilings. However, one other tile setting was marked for extra test in parameterisation because 3 tilings only corresponds to one tile per input combination; 1 for each $(X \bullet Time), (Y \bullet Time)$ and $(X \bullet Y \bullet Time)$. Therefore the best setting where $T > 3$ was also chosen: T9-Tr25-X1.0.

5.2 Parameterisation

One of the main concerns regarding the previous work is in the issue of parameterisation. Although initial spot tests indicated that the original choice of α, ε, and γ was quite reasonable, further investigation was warranted. Following guidance from much of Suttons work, such as [SSK05], a "good enough" approach to parameterisation was adopted. This approach aims to find reasonable parameter settings, rather than necessarily the best ones. The parameters presented in Section 3.2 were originally chosen based on Vidal's work with agent marketplace learning [VIDD97 and VID98] to be "good enough" for all 3 learners, however because of the different behavioural nature of some algorithms the best "good enough" parameter for one algorithm may not be the best for another.

The following two sub-sections present the parameterisation tests for the marketplace and the predator/prey gridworld. Parameters for the mountain-car world were ported with the original code and fixed as such for benchmarking.

5.2.1 Marketplace Parameterisation

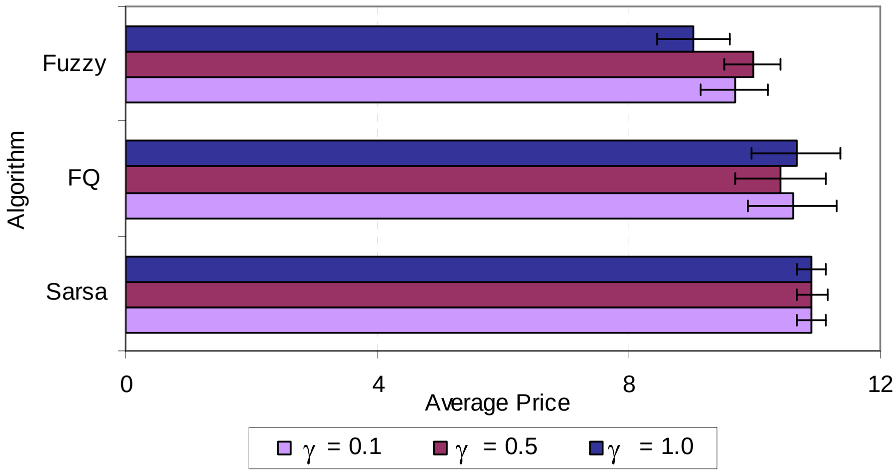


Figure 5.17: Effect of different γ values in Large Marketplace

In the first parameterisation test the following parameters were fixed: $\alpha = 0.15$ and $\varepsilon = 0.02$. Figure 5.17 illustrates the average price achieved by each algorithm over ten tests of 10,000 runs using three different γ settings. All three algorithms exhibit very little difference in average price due to changing γ. Fuzzy Sarsa is the only algorithm where any one setting might offer any improvement. However, since Fuzzy Sarsa appears to have a very minor improvement with $\gamma = 1.0$, and $\gamma = 1.0$ is the simplest case for debugging (a full one step backup), further marketplace experiments use this value.

For the parameter settings of α and ε, a scanning program was written to determine the appropriate targets settings. For all three algorithms, an ε range of 0.02 to 0.05 offered an adequate amount of exploration in this domain. For α, the range between 0.05 and 0.25 were identified as target values. In order to determine the relationship between these parameter settings, each algorithm was run against a linear fixed strategy, at each different parameter combinations in the target ranges, for 5 tests.

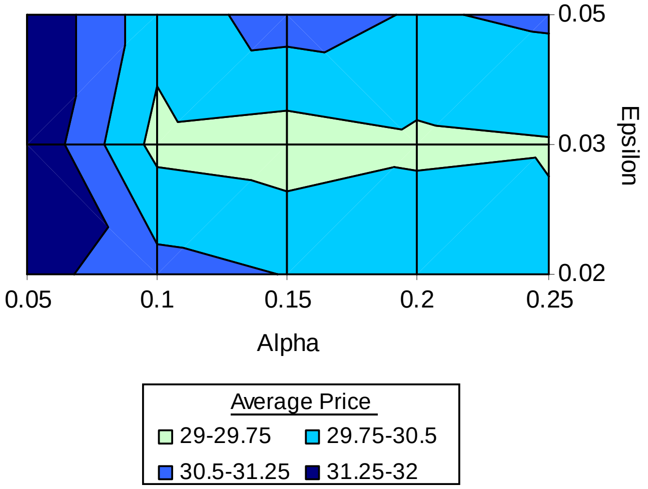


Figure 5.18: Effect of different α and ε values in Large Marketplace

The results in Figure 5.18 are the summed totals of the average price of each algorithm at each parameter combination. In the figure, lighter shades indicate a lower overall price, whereas darker shades indicate a higher overall price. As depicted, the learning algorithms perform best when ε is set to 0.03. For all three algorithms, the standard deviation of the prices achieved across the surface of the map is between 0.34 and 0.41. This indicates that there are no spots on the surface map where one algorithm performed significantly better than another.

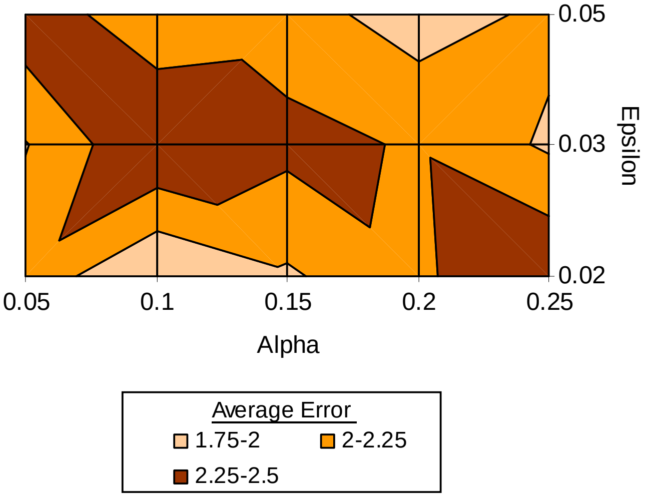


Figure 5.19: 95% confidence values of different α and ε values in Large Marketplace

The final setting that was chosen for further experiments in the marketplace was the α setting (at ε=0.03) of the best combination between overall price achieved while minimizing the overall deviation. Figure 5.19 gives the 95% confidence intervals around the overall price achieved from Figure 5.17: lighter shades indicate smaller error. For settings of α = 0.1 and 0.15, the overall price achieved is better than other settings, but the intervals are larger. For α = 0.25 the situation is reversed, the overall price is the worst of the good settings, but the error is minimal. The setting α = 0.2 is a good midpoint between the two. The final settings chosen are α = 0.02, ε = 0.03 and γ=1.0. While there may be settings not investigated that are superior, this setting is deemed as "good enough".

5.2.2 Predator/Prey Gridworld Parameterisation

The following section illustrates how the parameter settings for the predator/prey gridworld were determined. Unlike the marketplace world, the predator/prey gridworld is more volatile. Bad choices at specific times can have a large negative impact on the learning of the agent. Thus bad parameter choices exacerbate this problem. To that effect the following tests were performed.

As stated in Section 5.1.2.2, the final two tiling settings identified for parameterisation testing were as 3 tilings with $g_t = 25$ and $g_{xy} = 2.5$ and 9 tilings with $g_t = 25$ and $g_{xy} = 1.0$. These settings were chosen as the best performing out of 7 “good” settings, in turn identified from a larger tile test experiment that tested 16 different tile width combinations with 7 different numbers of tilings. As shown in Figure 5.20, the tile coding algorithm in the predator/prey world is especially sensitive to γ. However, Fuzzy Sarsa is less volatile with different setting of γ; with all results overlapping in their 95% confidence intervals.

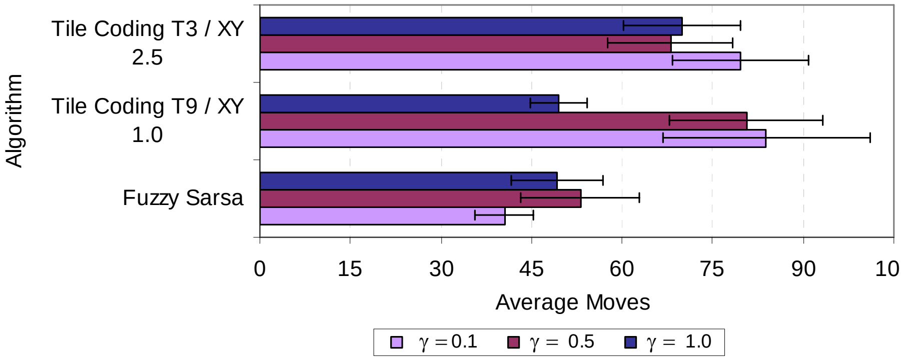


Figure 5.20: The effect of changing γ for Fuzzy Sarsa in 5x5 gridworld

The γ setting of 1.0 was chosen since this greatly boosts the tile coding setting for 9 tilings. However, due to rather rapid improvement of this setting both 3 and 9 tilings were used in further parameterisation tests. Figure 5.21 gives the averaged weighted results over 5 experiments of the last 5000 episodes at each setting for α and ε.

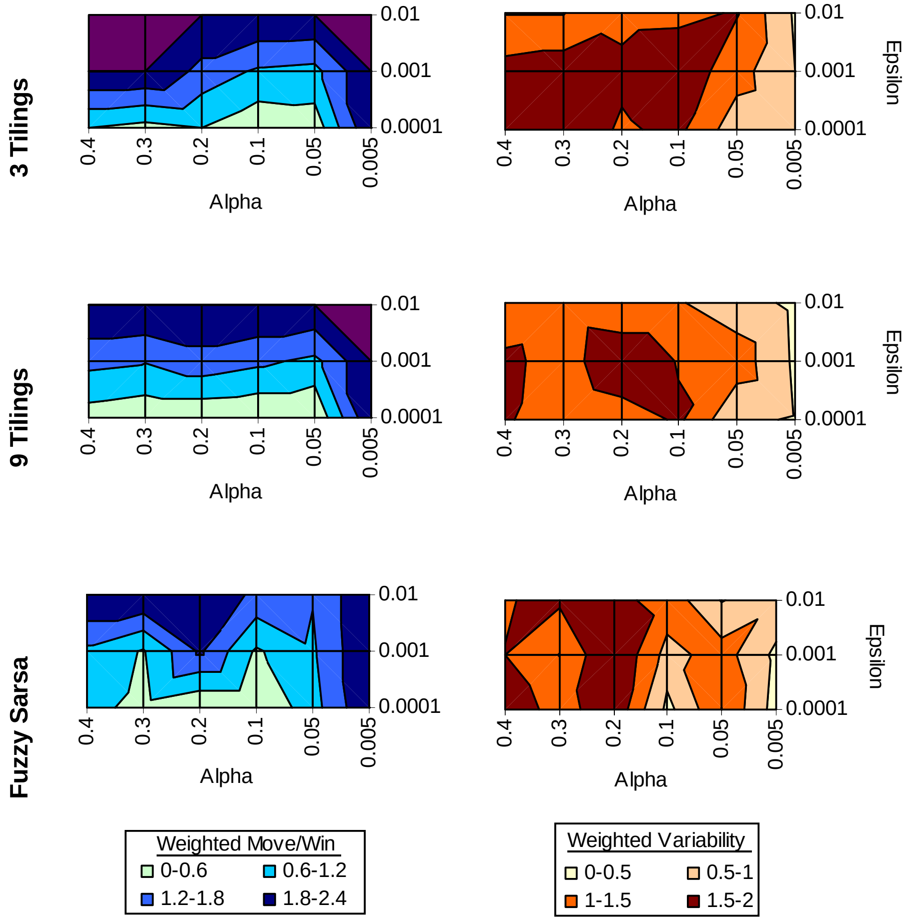


Figure 5.21: Weighted Move/Win Ratio of different α and ε for Tile Coding (3 & 9 Tilings) and Fuzzy Sarsa in 5x5 gridworld

As a result of the α and ε test results depicted in Figure 5.21, further experiments will use the tile coding setting of 9 tilings with $g_t = 25$ and

$g_{xy} = 1.0$, $\alpha = 0.05$ and $\varepsilon = 0.0001$ and Fuzzy Sarsa settings of $\alpha = 0.1$ and $\varepsilon = 0.0001$. These setting were chosen since they offered the best overall solution with the least variability.

5.3 Learning an optimal policy

This section is designed to validate the effectiveness of each type of function approximation by implementing the algorithms in all three simulation worlds. In the mountain-car world, the implementation is straightforward as the learner must simply learn a policy which allows them to get up the hillside. As seen earlier in the marketplace, but also in the predator/prey world, the problem is more complex when the learning agent is interacting with other agents in the world. To explore each algorithm's abilities to find a feasible policy in these worlds, the other agent(s) must follow a stationary strategy. The linear and greedy stationary strategies for the marketplace have already been introduced in Section 4.3, and the stationary strategies for the predator/prey gridworld will be introduced when presenting the relevant results.

5.3.1 Mountain-car World

In order to further investigate the effectiveness of the three different forms of function approximation in reinforcement learning, the three algorithms were implemented in the mountain-car world described in Section 3.4. Since the mountain-car domain is made up of continuous variables, implementing the tabular version of Sarsa is not practical.

The gradient-descent Sarsa(λ) with tile coding used 9x9 grid tilings across both velocity and position. The settings are further described in [SB98] and [SuttonMC]. Using the parameters $\gamma = 1.0$, $\alpha = 0.6$ and $\varepsilon = 0.025$ (with annealing for ε to allow the algorithms to settle on a greedy policy), all three algorithms find a solution. Figure 5.22 illustrates the stability of the solution found over 101 episodes. As expected, since FQ Sarsa is only a reduced state space solution, the coarse generalisation makes the action policy quite volatile and the furthest from the optimal solution. Fuzzy Sarsa does significantly better and is able to achieve a much more stable solution. However, gradient descent Sarsa(λ) with tile coding clearly achieves the most optimal and stable solution. The error on the solution is so small that it can not even be seen on the graph.

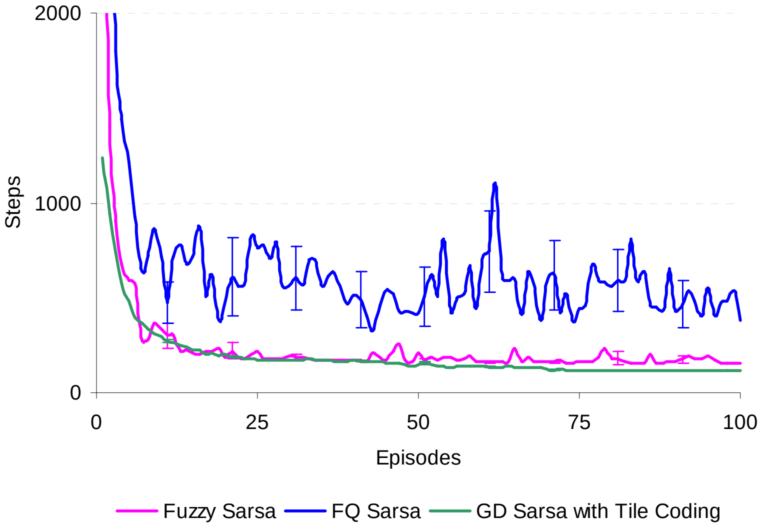


Figure 5.22: Stability of solution in Mountain-car problem

The final area investigated in the mountain-car world is the final action policy for each algorithm as illustrated by Figure 5.23. Although it is clear from the previous figure that gradient descent Sarsa(λ) with tile coding is clearly the more powerful algorithm it is also useful to investigate the final policy in order to illustrate its powerful generalisation capabilities in comparison with the fuzzy algorithms. As demonstrated by the three action selection policy surface maps, the policies learnt have broad areas of similarity. However, the policies learnt by the fuzzy algorithms lack the finer distinctions of the tile coding solution.

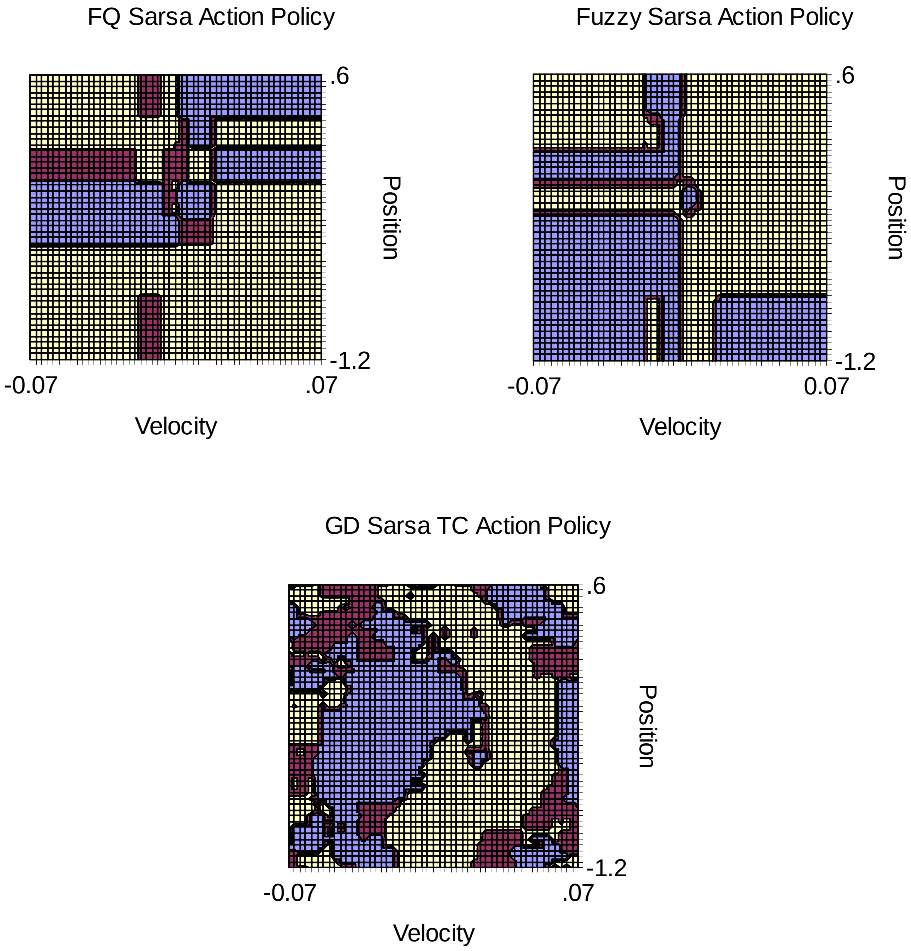


Figure 5.23: Final Action Policies for Mountain-car world.

5.3.2 Marketplace World

After the experiments described in Section 5.2, the parameters for the following section were fixed at $\alpha = 0.2$, $\varepsilon = 0.03$, and $\gamma = 1.0$. The membership functions for the fuzzy algorithm remain as described in Section 4.3 and the tile coding setting used has a granularity of 0.5 (as shown in Figure 5.24) with 2 overlaid tilings. This setting was determined as discussed in Section 5.1.2.

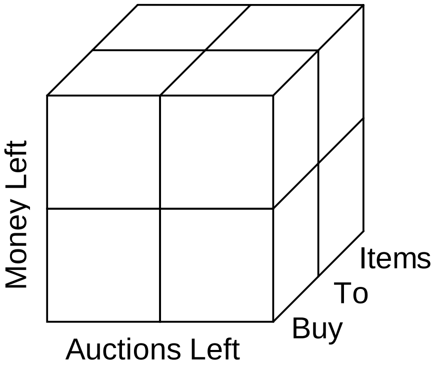


Figure 5.24: Tile Coding Strategy for the Marketplace

For the experiments in this section the VVVLarge setting in the marketplace is used. This setting gave a state action space of 8,787,366 combinations. Each function approximation learner is tested against each of the stationary strategies, Linear and Greedy. The total time of these trials was increased from 10000 to 20000 and thus the following graphs have been smoothed at an increment of 500.

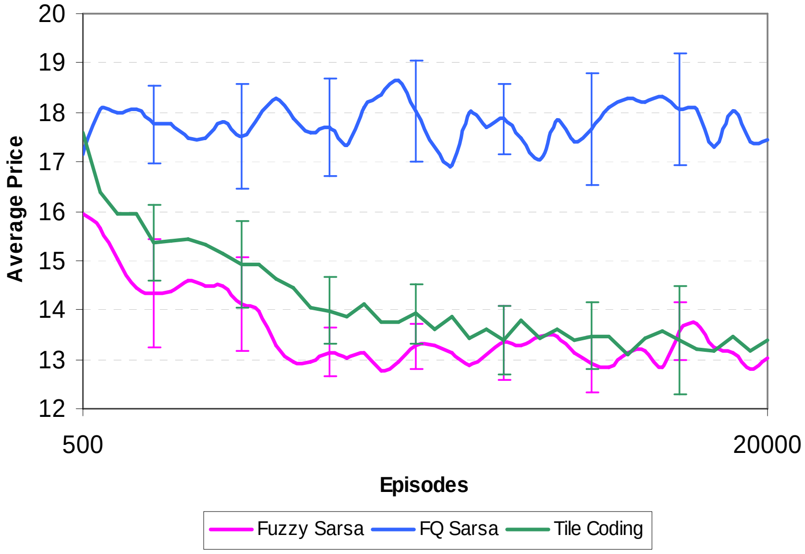


Figure 5.25: Agents against a Linear Strategy in VVVLarge Auction Game

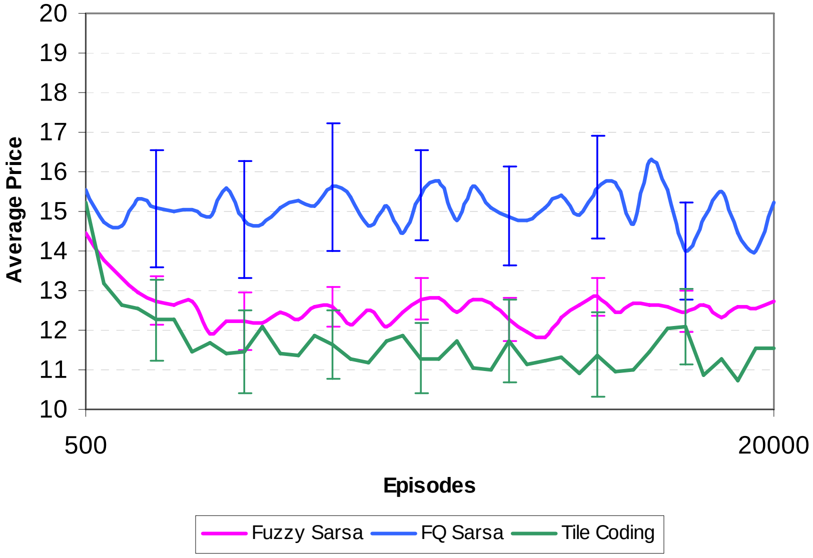


Figure 5.26: Agents against a Greedy Strategy in VVVLarge Auction Game

Both the Linear and Greedy Test show that the FQ Sarsa algorithm is not stable enough to warrant further investigation. The average price achieved oscillates and does not improve significantly from the start of the auction game. This is not a surprising, given the coarse nature of the state space for this algorithm.

In the case of Fuzzy Sarsa and the gradient descent Sarsa(λ) with tile coding, Figure 5.25 indicates that both algorithms perform similarly when competing with a fixed linear strategy[17], whereas Figure 5.26 indicates that the solution found by tile coding offers a small, but significant improvement over that of Fuzzy Sarsa when competing against a fixed greedy strategy. Figure 5.27 shows the results of increasing the complexity of the problem. In this test, both Fuzzy Sarsa and tile coding are required to find a solution with two stationary agents rather than just one. To add even more complexity to the game space, two different stationary strategies are selected. As indicated in Figure 5.27, when the game becomes more complex, Fuzzy Sarsa seems to gain some advantage over tile coding.

[17] The linear and greedy stationary strategies were defined in Section 4.3.

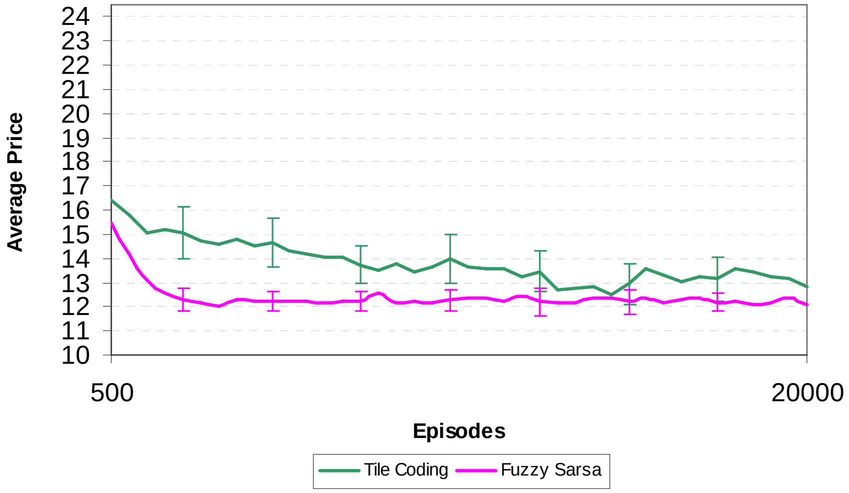


Figure 5.27: Agents against a Greedy and Linear Strategy in VVVLarge Auction Game

Before investigating these two algorithms in the context of coevolution in the marketplace domain, an investigation into their effectiveness in the predator/prey gridworld is given.

5.3.3 Predator/Prey Gridworld

From the previous sections, the reduced state space algorithm FQ Sarsa was deemed to be unstable, but both the Fuzzy Sarsa and the gradient descent Sarsa(λ) with tile coding seemed to offer similar benefits in the marketplace world. It was suspected by tile coding's slightly better performance in both the mountain-car world and the greedy stationary strategy test, that tile coding is the more powerful modeller, however further investigation into these two algorithms was warranted. The predator/prey gridworld environment has been described in Section 3.3. In order to investigate the two algorithms in the gridworld, a stationary strategy was defined.

The stationary strategies for both the predator and prey are defined as:

- *Corner Strategy:* This agent follows the boundaries of the maze and continually circles the grid.

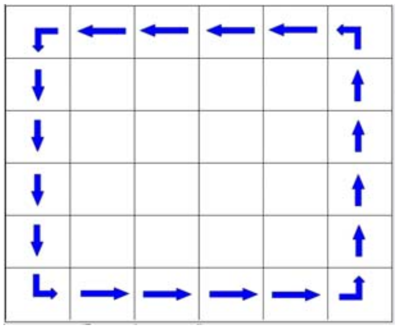


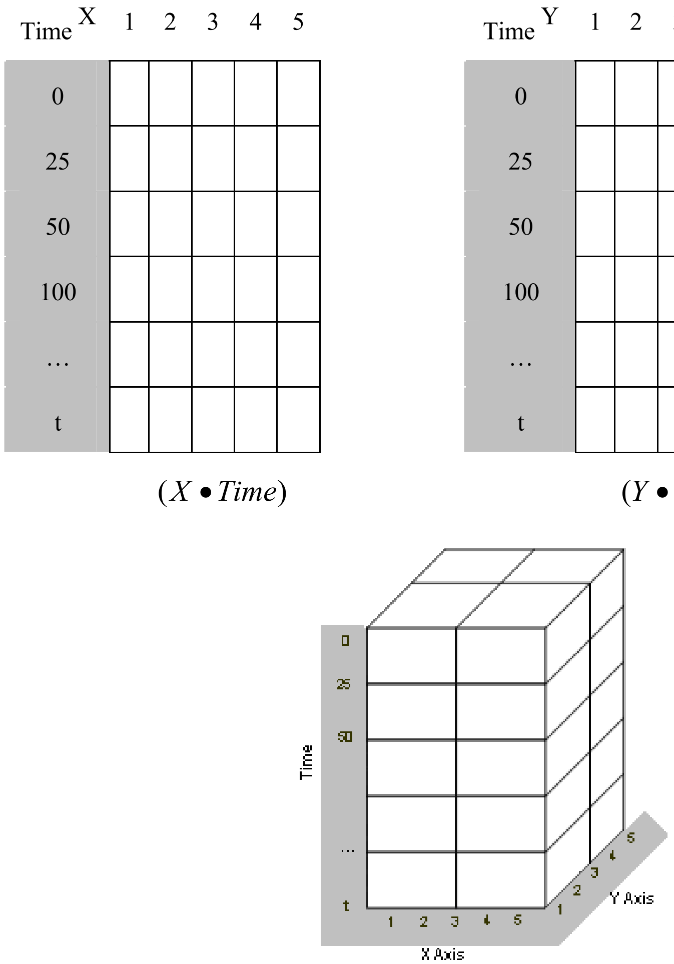
Figure 5.28: Corner Strategy in a Gridworld

As discussed in Section 5.1.2.2, the pursuit world is broken into three OR groups of tiles. All previous parameterisation for the predator/prey gridworld was done assuming the learning agent is the prey. The reason it was done this way is that being a predator is a much easier problem to learn than learning the more complex problem of avoiding the predator while still eating all the dots.

Similarly setting appropriate rewards in this domain can cause difficulty. In early tests, excessively punishing the prey agent for running out of time caused the agent to maximise its reward firstly by staying alive as long as possible and secondly by committing suicide if it thinks the allowable time is about to elapse. The prey agent becomes so concerned about the very negative rewards of being eaten and running out time, that it neglects the actions that would result in a more positive outcome[18].

The tiling scheme was designed as follows: The tilings are divided into 3 equal sized groups described as $(X \bullet Time) + (Y \bullet Time) + (X \bullet Y \bullet Time)$. The number of tilings indicates how many tiles are dedicated to each or generalisation. For example, if there are 3 tilings then 1 tile is dedicated to $(X \bullet Time)$, 1 tile to $(Y \bullet Time)$ and 1 to $(X \bullet Y \bullet Time)$. The final tiling scheme is illustrated in Figure 5.29.

[18] Such as eating all the dots!



$(X \bullet Time)$

$(Y \bullet Time)$

$(X \bullet Y \bullet Time)$

Figure 5.29: Tile density and width

This tiling structure allows the agent to make decisions base on its current X location and time, Y location and time and finally the XY location and time. From experiments presented in the previous 2 sections, the chosen settings are 9 tilings (3 of each tiling depicted in Figure 5.29) with $g_t = 25$ and $g_{xy} = 1.0$, $\alpha = 0.05$, $\varepsilon = 0.0001$ and $\gamma = 1.0$.

The fuzzy membership functions were chosen for their similarity to the tiling structure. To allow the fuzzy agent to have similar kinds of generalisation powers the following 4 label membership functions were needed:

- *X location:* The agent's location along the X axis.

- *Y location*: The agent's location along the Y axis.

- *Time with respect to the gridsize*: Time divided by the gridsize. This allows the agent to make the same inferences as the tile coding tile set for $(X \bullet Y \bullet Time)$.

- *Total time*: The current time with respect to the total time allowed.

The general membership for these four state variables and the action set is described by:

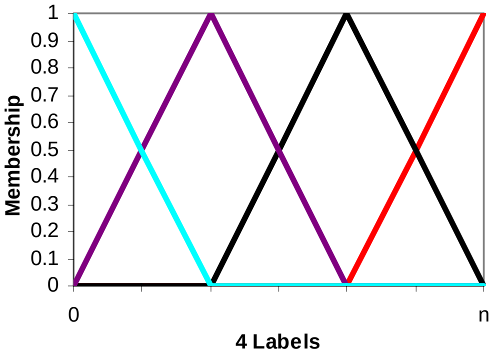


Figure 5.30: Fuzzy membership for 5x5 predator/prey gridworld

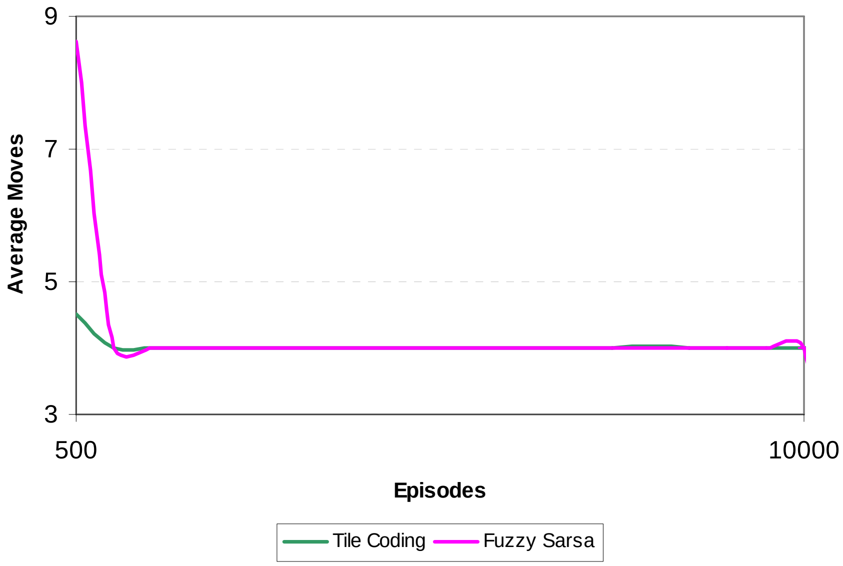


Figure 5.31: Tile Coding and Fuzzy Sarsa against fixed strategy prey in 5x5 predator/prey gridworld

Figure 5.31 gives the results over 10 experiments run to 10000 episodes of both agents learning to catch a stationary strategy prey. As shown, the 95% confidence intervals for both agents are insignificant – both agents reliably find the optimal solution quickly, with almost no deviation.

As mentioned learning to be the predator against a fixed stationary strategy in this domain is a simple task. A more complex one is for the agent to learn to be the prey. Figure 5.32 shows the results of each learning agent as the prey against a fixed strategy predator. Since this is a more complex task, the results presented are over 10 experiments run to 15000 episodes.

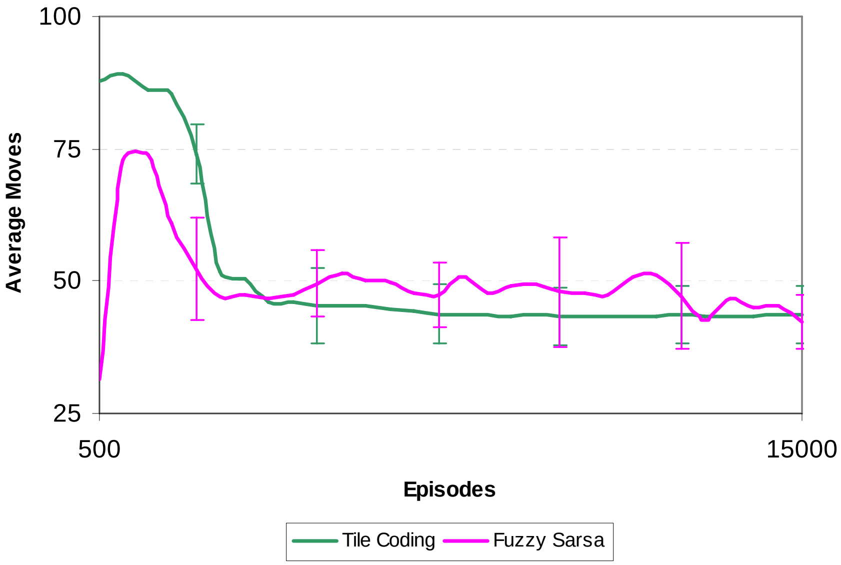


Figure 5.32: Tile Coding and Fuzzy Sarsa against fixed strategy predator in 5x5 predator/prey gridworld

As shown in Figure 5.32, both the tile coding agent and the Fuzzy Sarsa agent find a reasonable solution. The solution quality for both agents is very similar. The results for being the prey in the predator/prey gridworld again confirm the results seen in the mountain-car domain and in the marketplace domain.

Figure 5.33 illustrates further the effectiveness of the tile coding agent and the Fuzzy Sarsa agent. The results presented are the average win ratio of each agent. The average win ratio is defined by how often the agent completes the task at hand (eats all dots before time up). Fuzzy Sarsa is quicker to converge to a winning solution.

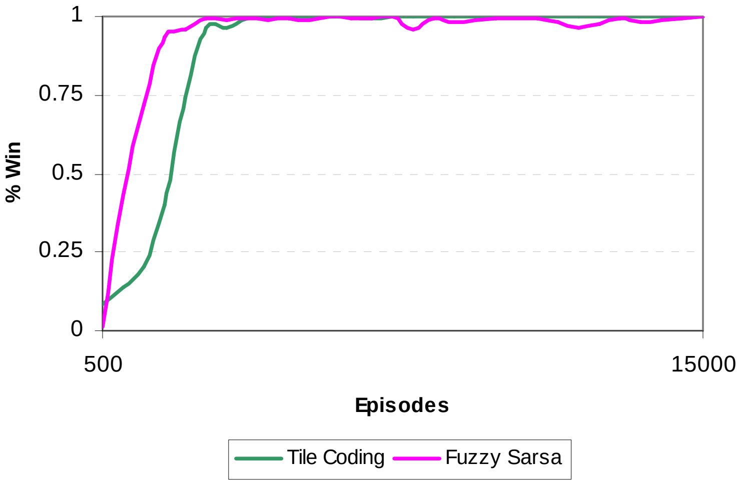


Figure 5.33: Tile Coding and Fuzzy Sarsa win ratio against
a fixed strategy predator in 5x5 predator/prey gridworld

5.4 Summary

This section has presented the results of stationary strategy tests of Fuzzy Sarsa, FQ Sarsa, and gradient descent Sarsa(λ) with tile coding. It has shown that FQ Sarsa is unstable for most large problem domains. Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding both offer good methods of function approximation in all three test domains. In a stationary environment, the approximation found by the tile coding technique appears to offer marginal improvement and better stability over the fuzzy technique.

The results presented in section also extend the recent (2005) finding of Booker in [BOOK05]. In that work, tile coding was compared with a learning classifier system called *XCS* [BW01]. There are three important differences between the research presented by Booker and the research presented in this section. Firstly, XCS is a learning classifier system, in that it learns *and evolves*, the research of this thesis has focused on the reinforcement learning aspect of LCS, and more specifically the fuzzy reinforcement learning of a LFCS. Secondly, this section has used control problems rather than prediction problems[19]. The final difference is an algorithmic one; as pointed out by [CCB04], an important difference between the LCS XCS and Bonarini's LFCS ELF [BON96a], is that in XCS there is consideration of competitive actions whereas in Bonarini's proposal the focus is on the interaction between the state portion of the rule[20].

Therefore, the results presented in the previous section have extended those presented by Booker in two ways. First, it shows similar amounts of function precision and smoothness for the tile coding technique in the control problems used in this research with the prediction problems used by Booker. Secondly, although the research presented in this thesis only uses the fuzzy reinforcement learning of a LFCS, the results indicate similar findings for the control function learnt by Fuzzy Sarsa, to that of the prediction problem learnt by XCS. In both

[19] This research has presented the background in terms of control problems, reinforcement learning for prediction problems is generally classed as an *easier* problem than that of control. [SB98] also include the relevant prediction algorithms for all the types (DP, Monte Carlo, etc.) presented in Section 2.

[20] A fuzzy rule is made up of the antecedent or *state*, and the consequent or *action*.

cases, the learnt function is more bumpy than the tile coding solution and does not exhibit the same fine details as the tile coding technique.

The research presented in this thesis has used control problems for algorithm testbeds. Therefore the next natural extension to the results presented in this section, is to analyse how the two algorithms, Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding, perform when placed in a multi-agent scenario. This is the topic of the next section.

6 Effects of Multiagent Competitive Coevolution

The next logical step after verifying that both function approximation algorithms could satisfactorily learn against a stationary strategy is to investigate the algorithms capabilities in the context of competitive coevolution. For these experiments, the adversarial simulation environments of the marketplace and the predator/prey gridworld were considered. In the case of the marketplace environment, the two agents competed directly for the same set of resource. In the predator/prey gridworld, the competing strategies each took their turn in both roles. In other words, in the first experimental set up, Fuzzy Sarsa adopted the role of the predator, while gradient descent Sarsa(λ) with tile coding played the role of the prey.

Section	Purpose	Experiment Setup
6.1 & 6.2	To determine the capabilities of Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding in non-stationary scenarios. ✓ Agent marketplace: *competing agents have same goal.* ✓ Predator/prey: *competing agents have different goals.*	✓ Learning parameters *fixed*. ✓ Fuzzy and tile schemas *fixed*. ✓ Algorithms learn against each other (*non-stationary* strategy).

Figure 6.1: Experiment Table for Section 6

Figure 6.1 summarises the experiments presented in this section.

6.1 Marketplace World

The initial setup for the coevolution tests in the Marketplace World consists of VVVLarge marketplace tests. The reason for choosing the VVVLarge marketplace was to give each algorithm two separate challenges. The first challenge was for the algorithm to be able to learn in the large state space and the second to be able to adjust to the other learner. The length of the test trial was increased to 20000 episodes to ensure that the algorithms have enough time to learn the gradation of both the combined learning problem.

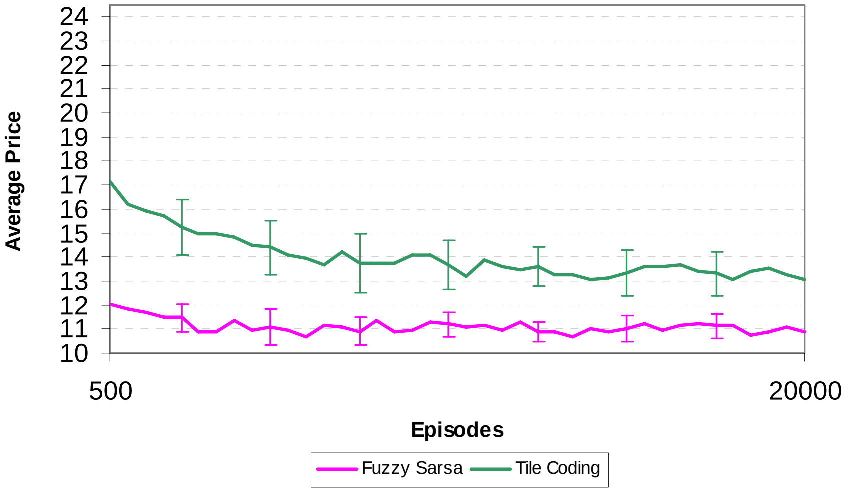


Figure 6.2: Fuzzy Sarsa vs. Tile Coding in VVVLarge Marketplace test

Figure 6.2 presents the results of a head to head test of Fuzzy Sarsa vs. tile coding in the VVVLarge Marketplace. In this experiment, the two algorithms were the only two agents competing in the market. There are enough items for both agents, and thus the goal of the two algorithms was to learn how to divide the items between them. Fuzzy Sarsa clearly achieved a significantly better price than tile coding throughout the experiment. Given the tile coding agent's better performance in fixed strategy experiments of Section 5.1.2.1, this poor performance was somewhat unexpected. Before concluding that Fuzzy Sarsa has more powerful modelling capabilities, the poorer performance of the tile coding agent must be investigated.

In the previous experiment in the marketplace, tile coding vs. a stationary strategy, since there was only one learner, the interactions of the game remained constant and therefore the tile coding agent was able to refine its coarse state space representation to learn a fairly good strategy to use against a stationary strategy. In the coevolution experiment, because more than one agent was learning at the same time, the interactions of the game fluctuated. This fluctuation makes it more difficult for each agent to learn an optimal solution. The confidence intervals of the tile coding agent in Figure 6.2 are much larger than that of the Fuzzy Sarsa agent. This indicates that prices achieved by the tile coding agent fluctuated more than those achieved by the Fuzzy Sarsa agent. Therefore one possible reason for the poorer performance of the tile coding agent is that the generalisation and resolution settings chosen by the stationary strategy experiment conducted in Section 5.1.2 are not sufficient for the coevolution experiment. To determine if the tile coding settings were adversely effecting the tile coding agent's results, the experiment was rerun with three of the other candidate settings from Section 5.1.2, and one random setting (T3W3).

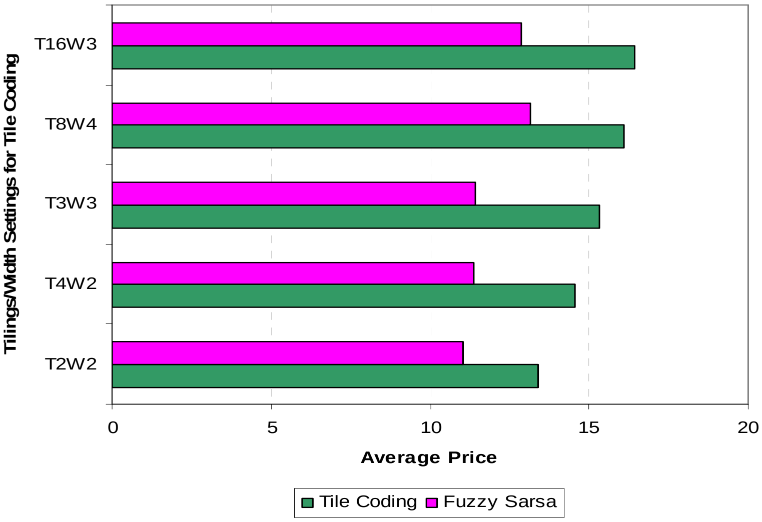


Figure 6.3: Other tiling settings in the VVVLarge Marketplace coevolution test

Figure 6.3 presents the averaged price achieved in the last 5000 episodes (out of 20000) by the tile coding agent and the Fuzzy Sarsa agent when in direct competition with each other. For both agents, the optimal price achieved is when the tile coding agent is at T2W2. In this environment, stiff competition increases the performance of Fuzzy Sarsa. It pushes the algorithm to carefully refine its distinctions between both cooperative state/action pairs (pairs that work together to generate a solution) and competitive ones (pairs that contain the same state portion).

Since the speed of learning in each individual tile in the tile coding method is dependant on the number of tilings, in one further attempt to boost the algorithm's performance the T8W4 test was rerun with increased α setting. Figure 6.4 illustrates the results of increasing the α for only the tile coding agent; the Fuzzy Sarsa agent remains fixed at 0.2. The figure shows the average price achieved by each agent over 10 tests of 20,000 runs each.

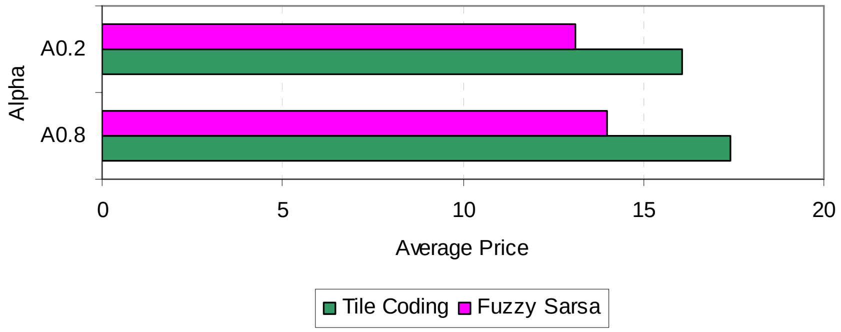


Figure 6.4: Increased α for T8W4 Tile Coding in
VVVLarge Marketplace coevolution test

These results indicate that the increased α caused the tile coding agent to become even more prone to getting stuck in a local minimum. The increased α is of no help to the tile coding agent.

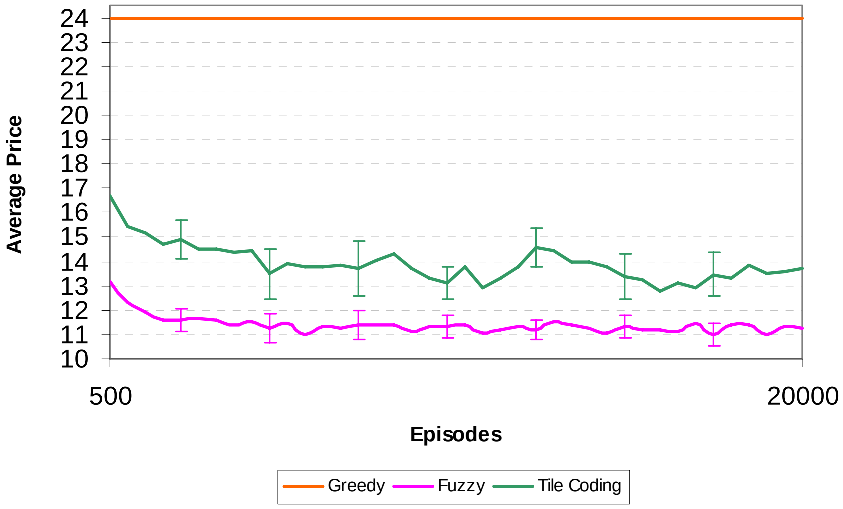


Figure 6.5: Fuzzy Sarsa, Tile Coding and Greedy in
VVVLarge Marketplace coevolution test

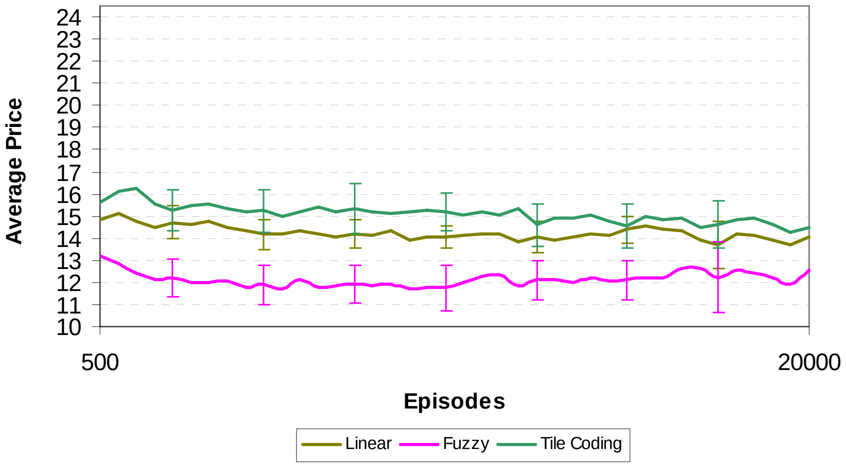


Figure 6.6: Fuzzy Sarsa, Tile Coding and Linear in VVVLarge Marketplace coevolution test

Returning to the original settings for the tile coding agent, Figure 6.5 and Figure 6.6 show the results of two 3-agent coevolution tests: the first with both function approximation methods and a greedy stationary strategy; the second, with both function approximation methods and a linear stationary strategy. In both cases, Fuzzy Sarsa clearly achieves a better overall price than any other agent.

The poor performance of the tile coding strategy is unexpected after its good performance against stationary strategies in all three domains. Before investigating this further, the algorithms were subjected to coevolution tests in the predator/prey domain.

6.2 Predator/prey gridworld

The multiagent tests in this environment consist of all combinations of agent algorithms in both the predator and prey roles. This results in four different combinations:

Test	Predator	Prey
1	Fuzzy Sarsa	Fuzzy Sarsa
2	Tile Coding	Tile Coding
3	Tile Coding	Fuzzy Sarsa
4	Fuzzy Sarsa	Tile Coding

Figure 6.7: Predator/prey gridworld agent combinations

In the initial tests, the total episodes per run is increased to 50000 to ensure that the results capture any latent emergent behaviours. All results presented in this section are averaged over a minimum of 5 experiments. In test 1, as shown in Figure 6.4, the Fuzzy Sarsa agents move towards an equilibrium win ratio for the first 20000 episodes, but after that, the predator agent is the clear winner. However, this graph does not show the full story.

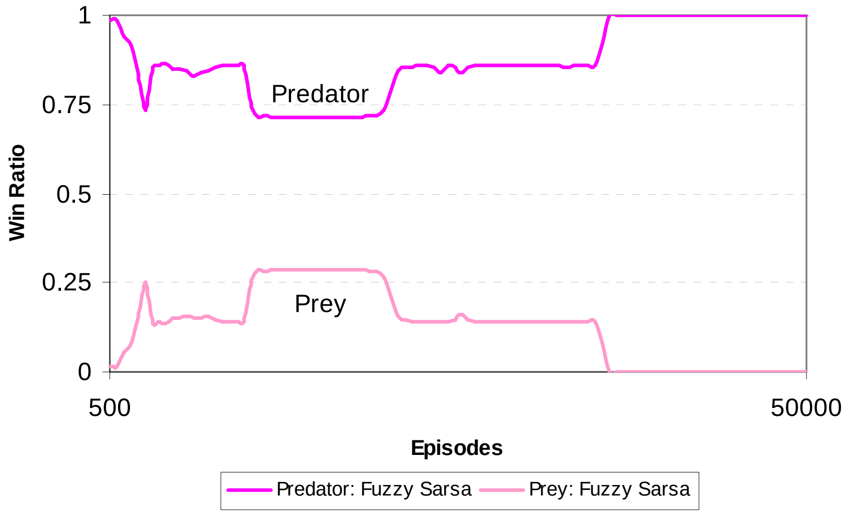


Figure 6.8: Win ratio of Fuzzy Predator vs. Fuzzy Prey in a 5x5 gridworld

As shown Figure 6.9, although the predator is the winner by win ratio, the game has settled onto an equilibrium. The maximum number of moves that the game has is 125. Therefore, what has occurred is that the prey agent has collected all of the dots it can – the last dot is guarded by the predator agent. This agent learns to always move into the wall over a dot adjacent to a wall. In this manner, the predator agent manages to stay stationary and guard the dot. It therefore has maximised its expected reward, as while its rewards are negative at every time step, the penalty for allowing the prey agent to complete the grid is even worse. The prey agent has also maximised its rewards, as it has generally collected 20 to 23 of the dots that the predator is not guarding. Therefore since the reward received in an empty square is 0, the longer it stays alive the better. This equilibrium point is the game's *Nash Equilibrium*. At Nash equilibrium, no player can do better by changing strategies unilaterally given that the other players don't change their Nash strategies. At least one Nash equilibrium exists in any game. [BO82]. At this point there is no move that will improve either the predator or prey's solution.

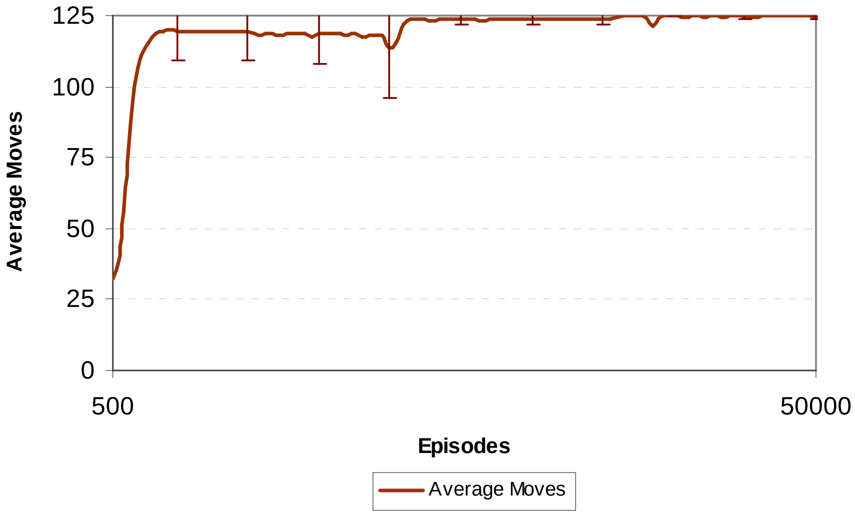


Figure 6.9: Average moves of Fuzzy Predator vs. Fuzzy Prey in a 5x5 gridworld

In test 2, the equilibrium reached is different from test 1. As shown in Figure 6.10 between the two tile coding agents, the prey is the clear winner. This seems to indicate that in this domain the tile coding agent is better at being prey than a predator.

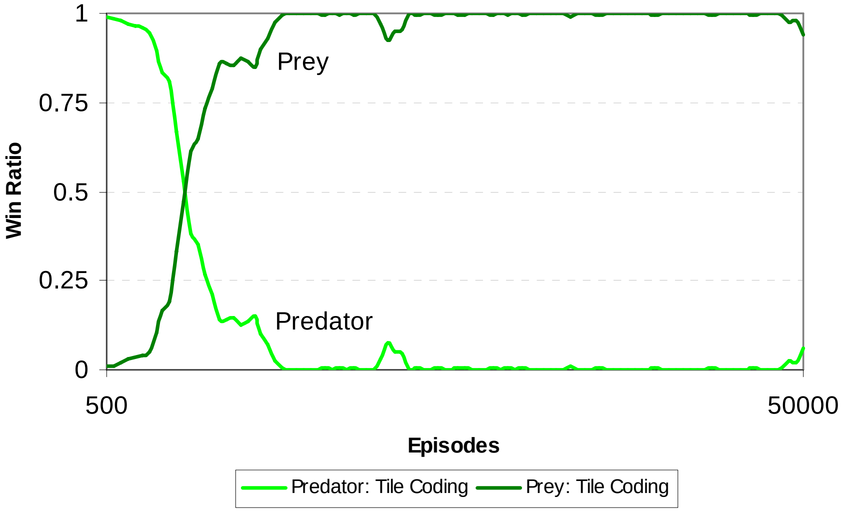


Figure 6.10: Average moves of Tile coding predator vs. Tile Coding Prey in a 5x5 gridworld

As shown in Figure 6.11, in test 3, the tile coding prey is the clear winner against the Fuzzy Sarsa predator. The results for this test are odd as, unlike when the Fuzzy Sarsa predator played against a Fuzzy Sarsa prey, it does not learn to effectively stay stationary and guard a dot. Instead the Fuzzy Sarsa predator guards a collection of 2 or 3 dots. Therefore due to the fine distinctions that the tile coding can make, the prey agent manages to navigate around the guarding predator and complete the grid.

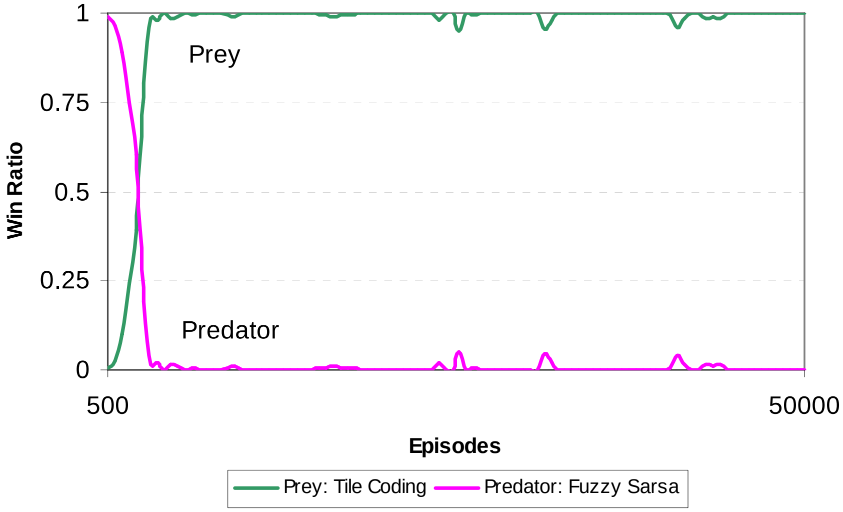


Figure 6.11: Win Ratio of Fuzzy Predator vs. Tile Coding Prey in a 5x5 gridworld

After the results of Section 6.1 the final multiagent results of test 4 shown in Figure 6.12 are somewhat unexpected. As indicated, the tile coding agent as predator clearly out performs the Fuzzy Sarsa prey.

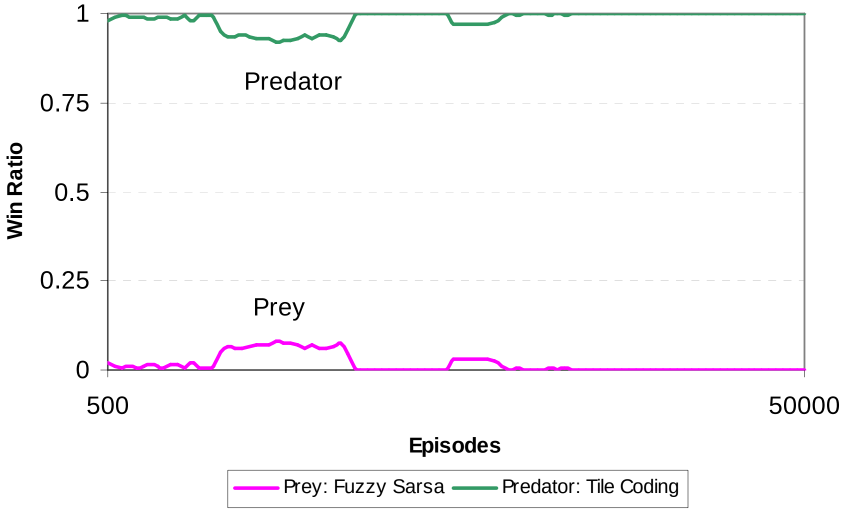


Figure 6.12: Win Ratio of Tile Coding Predator vs. Fuzzy Prey in a 5x5 gridworld

The results presented in test 3 and 4 (Figure 6.11 and Figure 6.12) are unexpected after Fuzzy Sarsa's better performance in the marketplace.

Test	Predator	Prey	Outcome
1	Fuzzy Sarsa	Fuzzy Sarsa	Nash Equilibrium
2	Tile Coding	Tile Coding	Prey
3	Tile Coding	Fuzzy Sarsa	Predator
4	Fuzzy Sarsa	Tile Coding	Prey

Figure 6.13: Predator/prey gridworld game outcomes

Figure 6.13 summarises the results of the four tests. In test 1, indicated by the Nash equilibrium observed, it was noted that the fuzzy algorithm was equally suited to playing either predator or prey. In test 2 where tile coding fulfils both predator and prey roles in the gridworld, it consistently solves the game as the prey. This indicates that the algorithm and/or settings used for the tile coding make it biased towards the prey.

Test 3 and 4 are still unexplained. In both tests tile coding is the clear winner. It is possible that this domain is simply better suited to the tile coding algorithm. However, the first observation made regarding this difference is that, in the predator/prey domain, the two algorithms use different α values: The Fuzzy Sarsa uses 0.1 whereas tile coding agent uses 0.05. The ideal α values were arrived at in a stationary strategy test with only the prey agent. Therefore, it is possible that the values are not actually "good enough" in this setting. Figure 6.14 shows spot check of the effects of altering α in test 3:

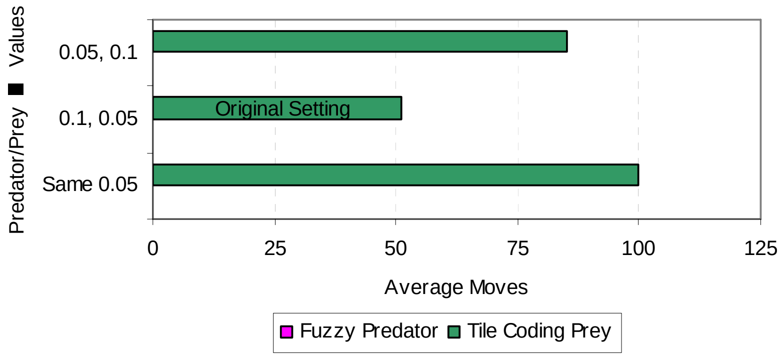


Figure 6.14: Changing α in test 3 of the predator/prey gridworld

Interestingly, changes to α in test 3 encouraged the game to move towards the Nash Equilibrium. Figure 6.15 presents the results of the same alterations in test 4. In this case, one change sways the balance towards Fuzzy Sarsa. The other setting does encourage better performance from Fuzzy Sarsa – it does actually win a small proportion of the time with the setting 0.05, 0.1.

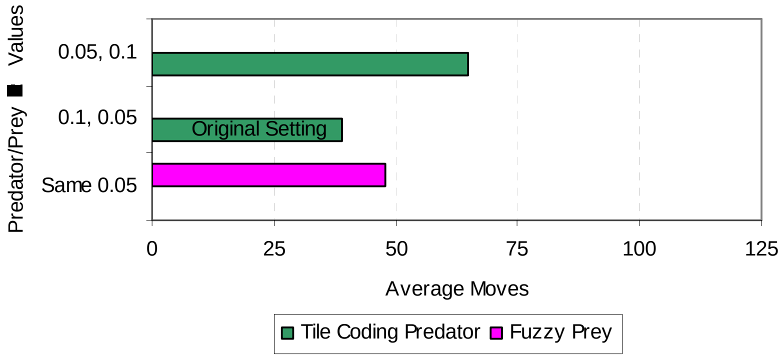


Figure 6.15: Changing α in test 4 of the predator/prey gridworld

During the tests in Figure 6.14 and Figure 6.15, one further point for consideration was identified. On empirical observation of the agents, the tile coding agent was able to learn very fine moves around the Fuzzy Sarsa agent. Fuzzy Sarsa's inability to learn the correct manoeuvre was interesting. In investigating possible reasons for this behaviour, the original setup of both algorithms was reviewed.

The predator/prey gridworld is extremely sensitive to different kinds of generalisation. When designing the tilings, standard tile patterns[21] were ineffective in this domain. However, although effort to retain similarity between the membership functions and the tilings led to the introduction of xy divided time and overall time, the resulting design significantly disadvantaged the algorithm in this domain. The tile coding agent uses $(X \bullet Time) + (Y \bullet Time) + (X \bullet Y \bullet Time)$ to generalise, whereas the fuzzy membership can be described as $(X \bullet Y \bullet Divided\_Time \bullet Time)$. This description of the state space for Fuzzy Sarsa lacks the distinction of the tile coding scheme. The tile coding scheme has three layers of distinctions, whereas the fuzzy scheme only has one.

While this resulted in a reasonable outcome in the stationary strategy environment, when used in a coevolutionary scenario, it is no surprise that the tile coding agent was able to outperform the Fuzzy Sarsa agent. Therefore further experimentation with α values would not be advantageous without redesigning the base membership function.

While the tests in this section indicate that tile coding clearly wins with the given inequalities between the two agents, important information about Fuzzy Sarsa's abilities in a multiagent scenario were still discovered. The

[21] Grid shaped tiles.

fact that Fuzzy Sarsa consistently finds the Nash Equilibrium in test 1, shows that is more flexible in design than tile coding; given the same settings it performing equally well as *both* predator and prey, whereas tile coding performs better as prey.

6.3 Summary

The previous sections have presented the results of Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding in two separate competitive domains. In the first domain, both agents are competing with each other to achieve the *same goal*. In the second domain, each agent is embroiled in a "to be or not to be" battle; each agent is trying to achieve *different goals*.

To that effect, Fuzzy Sarsa is a more robust algorithm when it comes to competing for the same goal. Although Fuzzy Sarsa loses against a tile coding agent with appropriate α values and good tilings, it is more robust to errors in generalisation design and has a wide range of modelling capabilities given a particular design.

7 Conclusions

7.1 Conclusions

The aim of this research was to identify new function approximation algorithms in a multiagent setting or coevolution and to analyse their performance. To that effect, the outcomes of this research are:

- The novel fuzzy on-policy reinforcement learning algorithm called Fuzzy Sarsa.
- A detailed evaluation of Fuzzy Sarsa in comparison with the popular technique of gradient descent Sarsa(λ) with tile coding in three separate simulation environments. This evaluation demonstrated that the performance that both fuzzy and tile coding techniques perform similarly in stationary environments.
- A critical analysis of the performance of both Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding in a coevolutionary setting was given. This analysis showed that Fuzzy Sarsa is more robust with regards to a competitive coevolution than the tile coding solution.

This results presented in this thesis has shown that Fuzzy Sarsa is able to produce a better and more robust solution in the context of a multiagent system where agents are competing for the same goal. It has also indicated that this robustness may extend to competition for different goals. Fuzzy Sarsa has also recently been successfully applied in resource management for IP networks

[SBP05]. Since Fuzzy Sarsa has performed well in this context, this research has identified the areas presented in the next section for further investigation.

7.2 Future Work

In Section 2.7 a variety of multiagent techniques were presented, some of which were developed in parallel to this research. Specifically the approach presented in [BV02 and BV02a], may improve the tile coding solution in the domains. While the tile coding approach used is similar, the use of a variable learning rate would be beneficial. Furthermore, the Fuzzy Sarsa algorithm could be adjusted to learn multiple policies and thus these principles could also be extended to the Fuzzy Sarsa algorithm.

Fuzzy Sarsa does not use state eligibility traces like gradient descent Sarsa(λ). Therefore an potential improvement to Fuzzy Sarsa is extending it to Fuzzy Sarsa(λ) and comparing it with both Fuzzy Sarsa and gradient descent Sarsa(λ) with tile coding. [BON98] provides a suggested methodology for this extension.

Further investigation into the different goal competitive coevoultion would be beneficial. However, another interesting avenue of further research includes investigating the algorithms in a cooperative framework.

Another recent investigation by [SST05], advocated online adjusting of the tile coding parameters. This technique could also potentially be used to improve the performance of the gradient descent Sarsa(λ) with tile coding algorithm.

A variety of issues in Fuzzy Sarsa provide interesting avenues for further research. One issue is to investigate other forms of T-norm for action combination. Another possibility is investigating the shape and form of membership function. The idea of online parameter adjusting over a set of pre-defined *reasonable* functions is also a possibility.

Appendix I. Complete Tile Tests

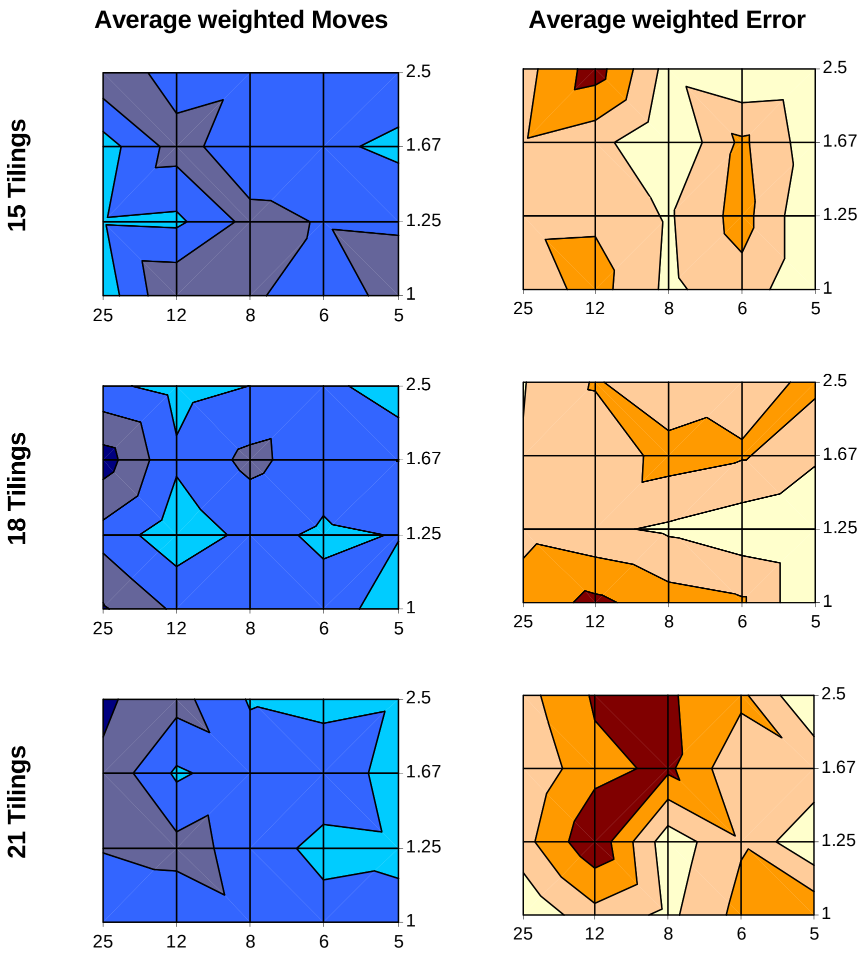


Figure I.1: Tile Tests in Predator/prey gridworld.

Figure I.1 illustrates the higher tiling sizes from the tiling experiments from Section 5.1.2. They are not included in that section because they do not add any benefit to the results there. As the tiling size went up, the % win ratio went down, and the average moves achieved went up. An increase in α may improve the results for the higher number of tilings. However, since good settings were found at lower settings, these tilings are not investigated any further.

8 References

8.1 Publications by Author

[BTetal00] J. Bigham, L. Tokarchuk, D.J. Ryan, L.G. Cuthbert, J. Lisalina, M. Dinis. (2000). Agent-Based Resource Management for 3G Networks. *In the Proceedings of the Second International Conference for 3G Mobile Technologies*.

[TAetal01] T. Tjelta, M. Annoni, L. Tokarchuk, A. Nordbotten, E. Scarrone, J. Bigham, C. Adamsand, K. Craig and M. Dinis. (2001). Future Broadband Radio Access Systems for Integrated Services with Flexible Resource Management. *IEEE Communications Magazine* 2-9.

[TBC04] L Tokarchuk, J Bigham, and L Cuthbert. (2004). Fuzzy Sarsa: An approach to fuzzifying Sarsa Learning. *Proceedings of the International Conference on Computational Intelligence for Modeling, Control and Automation*.

[TBC05] L Tokarchuk, J Bigham, and L Cuthbert. (2005). Fuzzy Sarsa: An approach to linear function approximation in reinforcement learning. To be published in *Proceedings of the International Conference on Artificial Intelligence and Machine Learning*.

[TBC06] L Tokarchuk, J Bigham, and L Cuthbert. (2006). Fuzzy and tile coding function approximation in agent coevolution. To be published in *Proceedings of the IASTED Conference on Artificial Intelligence and Applications.*

8.2 References

[ALB81] J. S. Albus. (1981). *Brains, Behavior, and Robotics.* Peterborough: Byte Books.

[BO82] T. Basar and G. J. Olsder. (1982). *Dynamic Noncooperative Game Theory*. London: Academic Press.

[BELL57] R.E. Bellman. (1957). *Dynamic Programming.* Princeton: Princeton University Press.

[BERV01] H. R. Berenji, D. Vengerov. (2001) On Convergence of Fuzzy Reinforcement Learning. *FUZZ-IEEE 2001,* (pp. 618-621).

[BER94] H. R. Berenji. (1994). Fuzzy Q-Learning: A new approach for fuzzy dynamic programming, *Proceedings of the Third IEEE Conference on IEEE World Congress on Computational Intelligence,* (pp. 26-29).

[BER96] H. R. Berenji.. (1996). Fuzzy Q-Learning for generalization of reinforcement learning. *Proc. of FUZZIEEE'96.* New Orleans.

[BEZ93] J.C. Bezdek; *Fuzzy Models – What are they, and why?*; IEEE Transactions on Fuzzy Systems; Vol. 1, No. 1; February 1993.

[BON96] A. Bonarini. (1996). Delayed Reinforcement, Fuzzy Q-Learning and Fuzzy Logic Controller. In Herrera, F., Verdegay, J. L. (Eds.), *Genetic Algorithms and Soft Computing, (Studies in Fuzziness, 8)*, D, (pp. 447-466). Berlin: Physica-Verlag.

[BON96a] A. Bonarini. (1996). Evolutionary learning of fuzzy rules: competition and cooperation. In: W. Pedrycz, Ed., *Fuzzy Modelling: Paradigms and Practice*, (pp. 265-283). Kluwer Academic Press.

[BON97] A. Bonarini. (1997). Anytime learning and adaption of structured fuzzy behaviors. In M. Mataric (Ed.), *Adaptive Behavior Journal; Special issue on "Complete agent learning in complex environments",* 5 (3-4), (pp. 281-315).

[BON98] A. Bonarini. (1998). Reinforcement distribution for fuzzy classifiers: a methodology to extend crisp algorithms. *Proceedings of the IEEE World congress on Computational Intelligence (WCCI) - Evolutionary Computation,* (pp. 51-56). Piscataway, NJ: IEEE Computer Press.

[BBM99] A. Bonarini, C. Bonacina and M Matteucci. (1999). Fuzzy and Crisp representation of real-valued input for learning classifier systems. *In Proceedings of IWLCS99*, Cambridge MA: AAAI Press.

[BON00] A. Bonarini. (2000). An Introduction to Learning Fuzzy Classifier Systems. *Lecture Notes in Computer Science*, Volume 1813, (pp. 83-106).

[BOOK05] L. B. Booker. (2005). *Approximating Value Functions in Classifier Systems*, Technical paper, The MITRE Corporation.

[BK05] L. Bull and T. Kovacs. (eds)(2005). Foundations of Learning Classifier Systems: An Introduction. *Foundations of Learning Classifier Systems*. Springer.

[BW01] M. Butz and S. W. Wilson. (2001). An Algorithmic Description of XCS. In *Revised Papers From the Third international Workshop on Advances in Learning Classifier Systems* (September 15 - 16, 2000). P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Lecture Notes In Computer Science, vol. 1996. Springer-Verlag, London, 253-272.

[BV02a] M. Bowling and M. Veloso. (2002). Scalable Learning in Stochastic Games. *In Proceedings of the AAAI-2002 Workshop on Game Theoretic and Decision Theoretic Agents*, Edmonton, Canada.

[BV02] M. Bowling and M. Veloso.(2002). Multiagent learning using a variable learning rate. *Artificial Intelligence.*

[BZ03] R. P. Brent and P. Zimmermann. (2003). Random number generators with period divisible by a Mersenne prime. *Computational Science and its Applications - ICCSA 2003, Lecture Notes in Computer Science, Vol. 2667*, (pp. 1-10). Berlin: Springer-Verlag.

[BM95] J. A. Boyan and A. W. Moore. (1995). Generalisation in reinforcement learning: Safely approximating the value function. In G. Tesauro, S. Touretzky, and T. Leen, editors, *Advances in Neural InformationProcessing Systems 7*. MIT Press.

[CCB04] J. Casillas, B. Carse and L. Bull. (2004) Fuzzy XCS: an Accuracy-based Fuzzy Classifier System. *In Proceedings of the XII Congreso Espanol sobre Tecnologia y Logica Fuzzy*.

[CM68] R. Chambers and D. Michie. Boxes: An experiment on adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence2*, (pp. 125-133. 1968).

[GL94] P.Y. Glorennec. (1994). Fuzzy Q-Learning and Dynamical Fuzzy Q-Learning. *Proc. of FUZZ-IEEE'9.,* Orlando.

[GLJ97] P. Y. Glorennec and L. Jouffe. (1997) Fuzzy Q-learning. *Proc. of FUZZ-IEEE'9,.* (pp.659-662).

[GDW91] P. J. Gmytrasiewicz, E. H. Durfee, and D. K. Wehe. (1991). A decision-theoretic approach to coordinating multi-agent interations. *In Proc. Int. Joint Conf. on Artif. Intell*., (pp 62-68).

[G96] P.J. Gmytrasiewicz. (1996). On reasoning about other agents; In M. Wooldridge, J. Müller, M. Tambe, editors, *Intelligent Agents II: Lecture Notes in Artificial Intelligence,* 1037. Springer-Verlag.

[GNK97] P.J. Gmytrasiewicz, S. Noh and T. Kellog. (1997). Bayesian Update of Recursive Agent Models. *In AAAI-97 Workshop on Multiagent Learning*.

[HetAl00] J. H. Holland, L. B. Booker, M. Colombetti, M. Dorigo, D. E. Goldberg, S. Forrest, R. L. Riolo, R. E. Smith, P. Luca Lanzi, W. Stolzmann, S. W. Wilson. (2000) What is a Learning Classifier System?, *Lecture Notes in Computer Science*, Volume 1813, (pp. 3-32).

[HW98] J. Hu and M. P. Wellman. (1998). Multiagent Reinforcement Learning: Theoretical Framework and an Algorithm. *Proceedings of the 15th International Conference on Machine Learning,* (pp. 242-250).

[HW03] J. Hu and M. P. Wellman. (2003). Nash Q-Learning for General-Sum Stochastic Games. *Journal of Machine Learning Research 4 (*pp. 1039-1069).

[HUM96] M. Humphrys. (1996). Action selection methods using reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior* (pp. 135-144). Cambridge, MA: MIT Press, Bradford Books.

[JAN91] J. Jantzen. (1991). Fuzzy Control. *Lecture notes in On-Line Proces Control (5343)* Publ no 9109. Technical University of Denmark.

[JSW98] N. Jennings, K Sycara and M. Wooldridge. (1998). A Roadmap of Agent Research and Development. A*utonomous Agents and Multiagent Systems,* (275-306). Kluwer Academic Publishers.

[JB92] R. A. Johnson and G. K. Bhattacharyya. (1992). *Statistics: principles and methods*, 2nd Ed. John Wiley & Sons, Inc.

[KLM96] L.P. Kaelbling, L.M. Littman and A.W. Moore. (1996). Reinforcement learning: a survey. *Journal of Artificial Intelligence Research, vol. 4,* (pp. 237—285).

[LR00] P. L. Lanzi, R. L. Riolo. (2000). A Roadmap to the Last Decade of Learning Classifier System Research. *Lecture Notes in Computer Science*, Volume 1813, (pp. 33-62).

[L94] M. L. Littman. (1994). Markov games as a framework for multi-agent reinforcement learning. In W. W. Cohen & H. Hirsh, eds, *Proceedings of the Eleventh International Conference on Machine Learning (ML-94)*, (pp. 157-163). New Brunswick, NJ: Morgan Kauffman Publishers, Inc.

[LCK95] M. Littman, A. Cassandra, and L. Kaelbling. (1995). Learning policies for partially observable environments: Scaling up. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, (pp. 362—370). San Francisco, CA: Morgan Kaufmann Publishers.

[LS96] M. L. Littman and C. Szepesv'ari. (1996). A generalized reinforcement-learning model: Convergence and applications. In Saitta, L., ed., *Proceedings of the Thirteenth International Conference on Machine Learning*, (pp. 310-318).

[MUK01] M. Mukaidono. (2001). *Fuzzy Logic for Beginners*. Singapore: World Scientific Publishing.

[MN98] M. Matsumoto and T. Nishimura. (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1* (pp 3-30).

[NG97] S. Noh and P. J. Gmytrasiewicz. (1997). Agent Modeling in Antiair Defense. *Proceedings of the Sixth International Conference on User Modeling.*

[OFJ99] E. Oliveira, J. M. Fonseca, N. R. Jennings. (1999). Learning to be Competitive in the Market. *Proc. AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities* (pp. 30-37). Orlando, FL.

[P96] H.V.D. Parunak. (1996). Applications of distributed artificial intelligence in industry. In G. M. P. O'Hare and N. R. Jennings, eds, *Foundations of Distributed Intelligence* (pp. 139-164). Wiley Interscience.

[PJL02] K. Pawlikowski; H.-D.J. Jeong and J. -S.R. Lee. (2002). On credibility of simulation studies of telecommunication networks. *Communications Magazine, IEEE,* Vol.40, Iss.1., (pp.132-139)

[PW96] J. Peng and R. J. Williams. (1996). Incremental multi-step Q-learning. *Machine Learning,* 22, (pp. 283-290).

[PSD01] D. Precup, R. S. Sutton,. S. Dasgupta. (2001). Off-policy temporal-difference learning with function approximation. *In Proceedings of the Eighteenth Conference on Machine Learning (ICML 2001)*, (pp.417-424). Morgan Kaufmann.

[POW87] M. Powell. (1987). Radial basis functions for multivariable interpolation : A review. J.C. Mason and M.G. Cox, eds, *Algorithms for Approximation*, (pp.143-167).

[RV00] P. Riley and M. Veloso. (2000). On Behavior Classification in Adversarial Environment. Proceedings of the Fifth International Symposium on Distributed Autonomous Robotic Systems (DARS-2000).

[RUB81] R. Rubinstein. (1981). *Simulation and the Monte Carlo Method*. New Your, NY: Wiley.

[RN95] S. Russel and P. Norvig. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall.

[RN94] G. A. Rummery and M. Niranjan. (1994). *On-line Q-learning using connectionist systems.* Technical Report; CUED/F-INFENG/TR166. Cambridge University Engineering Department.

[SAH94] M. K. Sahota. (1994). Action selection for robots in dynamic environments through inter-behaviour bidding. In Cliff, D., Husbands, P., Meyer, J.-A., and S, W., editors, *From Animals to Animats: The Third International Conference on Simulation of Adaptive Behavior* (pp. 138-142). Cambridge, MA: The MIT Press.

[SAR98] R. G. Sargent. (1998). Verification and Validation of Simulation Models, Ed. F. E. Cellier, *Chapter IX in Progress in Modelling and Simulation*, (pp. 159-169). London: Academic Press.

[SBP05] K. Shoop, J. Bigham, and C. Phillips. (2005). Resource management employing learned pseudo-delay for multi-service IP networks. *In Proceedings of the 10th European Conference on Networks and Optical Communications.*

[SSR98] J. C. Santamaria, R. C. Sutton, and A. Ram. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior,* 6(2).

[SS96] S. P. Singh and R. S. Sutton. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123-158.

[SST05] A. A. Sherstov and P. Stone. (2005). Function Approximation via Tile Coding: Automating Parameter Choice. In *SARA 2005*, pp. 194–205, Berlin: Springer Verlag.

[ST00] P. Stone. (2000). Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer. The MIT Press.

[SSK05] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement Learning for RoboCup-Soccer Keepaway. *Adaptive Behavior*, 13(3):165–188, 2005..

[SV00] P. Stone and M. Veloso. (2000). Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots; Vol 8, Issue 3..*

[SUTT96] R. S. Sutton. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *In Advances in Neural Information Processing Systems, volume 8*. The MIT Press.

[SB98] R. S. Sutton, A. Barto. (1998). *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

[SASM99] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. (1999). *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. Technical report, AT&T Labs—Research.

[TR96] M. Tambe and P. Rosenbloom. (1996). Architectures for agents that track other agents in multiagent worlds. In M. Wooldridge, J. Müller, M. Tambe, editors, *Intelligent Agents II: Lecture Notes in Artificial Intelligence.* Springer-Verlag.

[TS93] S.B. Thrun and A. Schwartz. (1993). Issues in using function approximation for reinforcement learning. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, Hillsdale, NJ: Lawrence Erlbaum

[VIDD97] J.M. Vidal and E.H. Durfee. (1997). Agents learning about agents: A framework and analysis. *In AAAI-97 Workshop on Multiagent Learning.*

[VID98] J.M. Vidal. (1998). Computational Agents that Learn about Agents: Algorithms for the design and a predictive theory of their behaviour. PhD Thesis. University of Michigan.

[WAT89] C. J. Watkins. (1989). *Learning from Delayed Rewards.* PhD thesis. King's College, Cambridge, UK.

[WATD92] C. Watkins, & P. Dayan. (1992). Q-learning. *Machine Learning*, 8, (pp. 279-292)

8.3 Internet Links

[MTJava]: Mersenne Twister Java Libraries
http://www.cs.umd.edu/users/seanl/gp/

[FIPA02] Fipa Specifications
http://www.fipa.org/specs/fipa00029/SC00029H.html

[JAVA] Java
http://java.sun.com/

[SJAVA] SimJava; http://www.dcs.ed.ac.uk/home/hase/simJava/index.html

[RNGTest] Random Number Generator Tests
http://www1.physik.tu-muenchen.de/~gammel/matpack/html/LibDoc/Numbers/Random.html

[SuttonMC] Mountain-car C++ and Lisp Implementation
http://www.cs.ualberta.ca/~sutton/MountainCar/MountainCar.html

[SuttonTC] Tile Coding Software
http://www.cs.ualberta.ca/~sutton/tiles.html