# Parallelizing Reinforcement Learning by Periodic Merging of Function Approximators

**David White**

**Supervisor: Dr. Daniel Kudenko**

4th Year MEng Project
Department of Computer Science
Universtiy Of York

Word Count: 19269, counted by WinEdt, all appendices excluded
70 Pages

May 10, 2005

**Abstract**

The reinforcement learning problem is that of an agent situated in an environment learning how to reach a goal. The agent performs an action on the environment at each time step and receives a reward for doing so. Using these rewards it estimates how valuable a state is according to the expected cumulative reward it would receive if it were in that state. As the number of actions it performs on the environment increases, the state valuations become increasingly more accurate and eventually converge to the optimal values. For complex domains, the values can converge very slowly.

This project considerers parallelization methods to speed up the convergence process by allowing many single agents to learn the same problem independently. The agents share the knowledge they have gained (the value estimates) at periodic intervals. To measure the success of the algorithm, it is evaluated on the mountain car problem. In this problem, a car situated in a steep sided valley must learn to escape by reaching the goal at the top of one side of the valley. Both the convergence speed and scalability of the parallel algorithms are evaluated. Considerable improvements are observed from the parallelization scheme, and even when the algorithm is used sequentially, faster learning is obtained.

# Contents

**1 Introduction** **5**

**2 Literature Review** **7**
2.1 The Reinforcement Learning Problem . . . . . . . . . . . . . . . . 7
2.1.1 Reinforcement Learning as a Markov Decision Process . . 8
2.2 Methods of Solving the Reinforcement Learning Problem . . . . 8
2.2.1 Dynamic Programming . . . . . . . . . . . . . . . . . . . . 9
2.2.2 Monte Carlo Methods . . . . . . . . . . . . . . . . . . . . . 9
2.2.3 Temporal Difference Learning . . . . . . . . . . . . . . . . . 10
2.3 Evaluation Domain . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Previous Work on Parallelizing Reinforcement Learning . . . . . 16

**3 Problem Analysis and Design** **18**
3.1 Basic Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Algorithm Evaluation Method . . . . . . . . . . . . . . . . . . . . 19
3.3 Maximum Altitude Approach . . . . . . . . . . . . . . . . . . . . 20
3.4 Best Function Approximator Determined by Evaluation . . . . . 22
3.5 Merge Best Function Approximators . . . . . . . . . . . . . . . . 24
3.5.1 Merge Algorithm 1: Match by Distance . . . . . . . . . . 24
3.5.2 Merge Algorithm 2: Sample Four Points . . . . . . . . . . 25
3.5.3 Merge Algorithm 3: Sample Five Points . . . . . . . . . . 27
3.6 Using Minimum Feature Values to Create a New Function Approximator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.7 All Function Approximators Represent the Same Set of Features 31
3.7.1 Algorithm for the Optimistic Formulation . . . . . . . . . 33
3.7.2 Algorithm for the Pessimistic Formulation . . . . . . . . . 33

**4 Evaluation** **37**
4.1 Evaluation Method . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 Parallel Algorithm Applied to the Optimistic Problem Formulation 38
4.2.1 Comparison Against a Single Agent . . . . . . . . . . . . 38
4.2.2 Scalability Results . . . . . . . . . . . . . . . . . . . . . . . 38
4.3 Parallel Algorithm Applied to the Pessimistic Problem Formulation 42
4.3.1 Comparison Against a Single Agent . . . . . . . . . . . . 42
4.3.2 Scalability Results . . . . . . . . . . . . . . . . . . . . . . . 44
4.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

**5 Implementation** **52**
5.1 High level Architectural Requirements . . . . . . . . . . . . . . . . 52
5.2 Reinforcement Learning Framework for a Single Agent . . . . . . 53
5.2.1 Choosing Which Action to Perform . . . . . . . . . . . . . 55
5.2.2 Performing an Action on the Environment . . . . . . . . . 56
5.2.3 Updating the Eligibility Traces . . . . . . . . . . . . . . . . 56
5.2.4 Updating the Feature Value Estimates . . . . . . . . . . . 56
5.2.5 Initializing the Function Approximator . . . . . . . . . . . 57
5.2.6 Computing the state-action Features . . . . . . . . . . . . 57
5.2.7 Detailed Descriptions of the Data Classes . . . . . . . . . 58
5.2.8 Logging . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.3 Extending the Single Agent RL Framework to Include Support for Multi Agent Experiments . . . . . . . . . . . . . . . . . . . . 61
5.3.1 Including an Evaluation Method . . . . . . . . . . . . . . 61
5.3.2 The Test Class . . . . . . . . . . . . . . . . . . . . . . . 61
5.3.3 Using a Configuration File to Specify the Parameters of an Experiment . . . . . . . . . . . . . . . . . . . . . . . . 62
5.3.4 High Level Procedures for Running Complete Experiments 63
5.4 Implementation of the Parallel Algorithms . . . . . . . . . . . . . 64
5.4.1 Select Agent with Maximum Altitude . . . . . . . . . . . 65
5.4.2 Select Best Function Approximator and Merge Algorithms 1,2 and 3 . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.4.3 Select Minimum Feature Value . . . . . . . . . . . . . . . 66
5.4.4 Select Minimum Native Feature Value . . . . . . . . . . . 66
5.5 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

**6 Conclusion** **68**
6.1 Further Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**A Code for the RL Framework** **73**
A.1 Action . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
A.2 ActionSet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
A.3 Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.4 ConfigFile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
A.5 EligibilityTrace . . . . . . . . . . . . . . . . . . . . . . . . . . 85
A.6 Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
A.7 FunctionApproximator . . . . . . . . . . . . . . . . . . . . . . 89
A.8 Logger . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
A.9 Random . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
A.10 Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
A.11 State . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
A.12 Statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
A.13 Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
A.14 main . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
A.15 Constants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
A.16 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

# List of Figures


2.1 CMAC uses overlapping tilings of binary features . . . . . . . . . 13
2.2 The Mountain-Car Task . . . . . . . . . . . . . . . . . . . . . . . . 15

3.1 A diagram depicting the operation of the parallel algorithms, the stages are explained in section 3.1 . . . . . . . . . . . . . . . . . . 19
3.2 Results for the "maximum altitude" parallel learning algorithm . 21
3.3 Results for the "evaluate and select best function approximator" algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.4 Distance calculation applied to two function approximators each with two tilings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.5 Results of merge algorithm 1 . . . . . . . . . . . . . . . . . . . . . 26
3.6 The points used to find the approximate value of a feature in merge algorithm 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.7 Results of merge algorithm 2 . . . . . . . . . . . . . . . . . . . . . 28
3.8 The points used to find the approximate value of a feature in merge algorithm 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.9 Results of merge algorithm 3 . . . . . . . . . . . . . . . . . . . . . 30
3.10 Results of the "minimum feature value" algorithm . . . . . . . . 32
3.11 Results of the "optimistic minimum feature value" algorithm where all agents use the same FA . . . . . . . . . . . . . . . . . . 34
3.12 Results of the "pessimistic minimum feature value" algorithm where all agents use the same FA . . . . . . . . . . . . . . . . . . 35

4.1 A comparison between a single agent and 5 parallel agents for a step cutoff of 50 using the optimistic formulation . . . . . . . . . 39
4.2 A comparison between a single agent and 5 parallel agents for a step cutoff of 75 using the optimistic formulation . . . . . . . . . 40
4.3 A comparison between a single agent and 5 parallel agents for a step cutoff of 100 using the optimistic formulation . . . . . . . . 41
4.4 Scalability results for the optimistic problem formulation with a step cutoff of 50 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.5 Scalability results for the optimistic problem formulation with a step cutoff of 75 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.6 Scalability results for the optimistic problem formulation with a step cutoff of 100 . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.7 A comparison between a single agent and 5 parallel agents for a step cutoff of 50 using the pessimistic formulation . . . . . . . . . 45
4.8 A comparison between a single agent and 5 parallel agents for a step cutoff of 75 using the pessimistic formulation . . . . . . . . . 46

4.9 A comparison between a single agent and 5 parallel agents for a step cutoff of 100 using the pessimistic formulation . . . . . . . . 47
4.10 Scalability results for the pessimistic problem formulation with a step cutoff of 50 . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.11 Scalability results for the pessimistic problem formulation with a step cutoff of 75 . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.12 Scalability results for the pessimistic problem formulation with a step cutoff of 100 . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.13 The divergent behavior observed on a particular run using the pessimistic problem formulation . . . . . . . . . . . . . . . . . . . 50

5.1 The class hierarchy for the RL implementation, the shaded boxes represent the high-level experiment procedures described in section 5.3.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.2 The true state space and the FA state space . . . . . . . . . . . . 58

# List of Algorithms

2.1 Procedural description of the Sarsa algorithm (Sutton and Barto, 1998, Section 6.4) . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Procedural description of the Q-learning algorithm (Sutton and Barto, 1998, Section 6.5) . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Full Sarsa algorithm (*linear, gradient-descent Sarsa($\lambda$)*) with eligibility traces and function approximaton (Sutton and Barto, 1998, Section 8.4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.1 Algorithm based on "maximum altitude" . . . . . . . . . . . . . . 20
3.2 Algorithm based on performing a full evaluation of each FA . . . . 22
3.3 Algorithm for the first merging solution, also a framework for the later merging algorithms . . . . . . . . . . . . . . . . . . . . . . . 25
3.4 Parallel Learning Algorithm that uses agent's experience to merge all the FAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.5 Algorithm for combining function approximators that represent the same features (optimistic case) . . . . . . . . . . . . . . . . . . 33
3.6 Algorithm for combining function approximators that represent the same features (pessimistic case) . . . . . . . . . . . . . . . . . 36

# Chapter 1

# Introduction

Reinforcement learning (RL) (Sutton and Barto, 1998) is the problem of an agent learning a policy to achieve a goal. Reinforcement is given to the agent through rewards which helps guide the agent through the environment to the goal. While moving through the environment, the agent builds up value estimates for particular states based on the rewards it receives. The value estimate for a state determines how *valuable* it is for the agent to be in that particular state. Eventually, the value estimates will converge on values that reflect the optimal policy. The optimal policy in reinforcement learning is defined as the policy that maximizes the total expected reward. Therefore, the agent must learn to forego immediate rewards to achieve the largest cumulative reward possible. This idea of dismissing early rewards for greater rewards in the future is generally a very difficult problem to overcome in learning techniques, however, reinforcement learning can successfully solve these types of problems. Furthermore, some RL methods allow optimal policies to be learnt simply through interacting with the environment.

However, the speed with which the value estimates converge to reflect the optimal policy scales exponentially in the size of the search space. As such, current RL techniques can only be applied to relatively simple tasks. For reinforcement learning to be applicable to real-world tasks, a significant speed up in convergence is required. Once such direction that could provide this increase in learning power is parallel reinforcement learning. This project discusses such an idea.

The basic premise behind the idea is that a number of agents are simultaneously performing reinforcement learning in isolated, but identical, environments. At periodic intervals the agents are asked for their current value estimates and these are merged in some way to combine the best estimates from each agent. The new value estimates are then distributed to the agents and the process is repeated. For this method to be successfully applied two problems must be solved, (1) how to determine which value estimates an agent is most knowledgable about and (2) how the best value estimates should be combined to produce even better estimates.

A number of algorithms that solve these two problems, with various levels of success, will be presented in this report. To measure the success of an algorithm, it will be evaluated on the mountain car problem. The mountain car problem considers a car stuck in a valley with two steep sides. The goal is to reach

the top of the mountain on one side of the valley. However, the car's motor does not provide sufficient power to simply drive up the side of the valley so it must learn to oscillate in the valley, thus building up sufficient momentum to reach the goal. It is this kind of task, where the agent must move further from the goal in order to eventually reach it, that reinforcement learning excels at. Furthermore, the state space of the mountain car problem is quite large among RL problem domains so it should prove to be a good test of the parallel algorithms' scalability.

The performance of the parallel algorithms developed in this project are very interesting. It appears that if the problem is structured in a certain way then the parallel algorithm can produce faster convergence than a single agent algorithm even when the parallel algorithm is implemented sequentially. This is a very promising result for improving sequential reinforcement learning. Obviously, when the algorithm is implemented in parallel the resulting speed-up is even more significant. Furthermore, it appears that the performance of the algorithm scales well in the number of parallel agents used.

The remaining report is structured accordingly. Chapter 2 contains a literature review exploring all the relevant topics that apply to this project. This includes a description of: the theory behind RL, various method of solving the RL problem, function approximation for applying RL to domains with a continuous state space, the mountain car problem and some work performed in similar subject areas. Chapter 3 describes the design of the various parallel algorithms developed in this project. Short evaluations are performed throughout to guide the design process. Chapter 4 describes a full evaluation of the best performing parallel algorithm. The convergence speed and scalability of the algorithm are measured over a number of tests. Chapter 5 describes the implementation of the framework required to carry out experiments involving the parallel algorithms. Since single agent algorithms are used in some evaluations, the implementation supports both traditional single agent RL algorithms and parallel algorithms. Chapter 6 concludes the project and discusses some further work that could be carried out in the field.

# Chapter 2

# Literature Review

## 2.1 The Reinforcement Learning Problem

The reinforcement learning problem is that of an agent interacting with an environment to achieve a goal. At each time step $t$ the agent receives information about the environment in the form of a state $s_t$. It then interacts with the environment by performing an action $a_t$, which is chosen according to the agents policy $\pi$. This generates a reward $r_t$ which is the only feedback the agent receives from the environment, and therefore the only information available for learning. The aim of the problem is to learn a policy that results in the agent receiving the largest cumulative reward. Naturally, this is not known or the problem would be solved. Instead an estimate, the expected return, is maximized which represents the amount of reward the agent expects to receive in the future.

A problem can either be structured in terms of episodes, where each episode ends when the goal is reached, or as a continuous task with no end-point defined. Although it appears that the two representations are different, they can be unified. When an episode finishes, the system enters into an absorbing state where it will loop forever, thereby making it into a continuous task. The expected reward can now be described in a uniform way if the notion of discounting is introduced. Discounting means that rewards received sooner are worth more than rewards received later. Equation 2.1 gives the cumulative expected discounted return $R_t$ for an agent at time t.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

Note that the reward at time $t$, $r_t$, is not included since it is the *future* rewards that are of interest. The parameter $\gamma \in [0, 1]$ is the discount rate and determines what weight rewards received far in the future have with respect to rewards received immediately. If $\gamma = 0$, the agent only takes into account the next reward received. Alternatively, as $\gamma \rightarrow 1$, future rewards become increasingly important. Clearly the expected return needs to be discounted since if $\gamma = 1$ the reward sequence for the continuous task would be infinite.

### 2.1.1 Reinforcement Learning as a Markov Decision Process

If the state signal $s_t$ is a Markov signal then the reinforcement learning problem can be described formally as a *Markov Decision Process* (MDP). A MDP is a tuple $< S, A, T, R >$, where $S$ is a set of problem states, $A$ is the set of available actions, $T(s, a, s') \rightarrow [0, 1]$ is a function which defines the probability that taking action $a$ in state $s$ will result in a transition to state $s'$, and $R(s, a, s') \rightarrow \mathcal{R}$ defines the reward received when such a transition is made. A state signal has the Markov property if it fully captures all current and past sensations the agent perceives. If the past sensations offer no useful information beyond that of the current sensations then a state signal containing only current sensations will retain the Markov property. In many domains, the state signal does not have the Markov property but so long as it is a close approximation to the Markov state signal, MDP theory can still be applied.

## 2.2 Methods of Solving the Reinforcement Learning Problem

Most reinforcement learning algorithms are based on estimating a value function, determining either the value of a state or a state-action pair. Using this information the agent can try to take actions that result in it being in more valuable states, and therefore receiving more cumulative reward. The choice of action is given by the agent's policy, $\pi$, which is formally defined as a mapping from each state $s \in S$ and action $a \in A$ of the MDP to the probability $\pi(s, a)$ of taking action $a$ in state $s$. The value function for a policy $\pi$ is defined as follows:

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{2.2}$$

where $E_{\pi}\{\}$ is the expected value given the agent follows policy $\pi$. Therefore $V^{\pi}(s)$ is the expected reward of starting in $s$ and then following $\pi$. In algorithms where the value of a state-action is considered instead of just the state, the value function is:

$$Q^{\pi}(s, a) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \tag{2.3}$$

Any value function must satisfy a recursive relationship, such that the expected reward is the immediate reward $r_t$ plus the discounted future reward given by the value function for the successive state: $\gamma V^{\pi}(s')$. This relationship is described by the Bellman equation (2.4) for $V^{\pi}(s)$ which averages the reward of all possible combinations of states and actions, weighted by the probability of each state and action occurring.

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi}(s')] \tag{2.4}$$

An optimal policy, $\pi^*$, can be described as the policy for which $V^{\pi^*}(s) \geq V^{\pi'}(s)$ for all possible policies $\pi'$ and $s \in S$. Therefore, the value function for

the optimal policy, $V^{\pi^*}(s)$, must equal the expected reward of the best action that can be chosen in $s$:

$$V^{\pi^*}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi^*}(s')] \tag{2.5}$$

where $\mathcal{A}(s)$ is the set of actions available in state $s$. When the optimal action-state value function is considered, $Q^{\pi^*}(s,a)$ must equal the immediate reward plus the value of the future rewards given the best action is chosen.

$$Q^{\pi^*}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^{\pi^*}(s',a')] \tag{2.6}$$

In all of the following algorithms for solving the RL problem, the ultimate aim is to learn an optimal policy. If the MDP parameters $T$ and $R$ are initially unknown then this is accomplished by *interacting with the environment* and observing what transitions and rewards result from these interactions. If all the parameters of the MDP are known then an optimal policy can be calculated by dynamic programming.

### 2.2.1 Dynamic Programming

The goal of dynamic programming methods are to find a solution to either of the Bellman equations (2.5 or 2.6) for each state $s \in S$.

One method for solving a reinforcement learning problem using DP involves creating $|S|$ simultaneous equations of 2.4, one for each state $s \in S$ and solving for all unknown values of $V^{\pi}(s)$. One of the solutions to this set of simultaneous equations will be the optimal policy. However, this is very computationally intensive so iterative methods are generally better suited to the task. Iterative methods generally work by creating a policy, known as *policy evaluation*, and then improving on it by applying *policy improvement*. Each time policy improvement is applied a better policy is guaranteed to be generated (provided the policy to be improved is not already optimal). Since each problem only has a finite set of policies, this repeated application of policy improvement, termed *policy iteration*, must converge to the optimal policy.

It turns out that policy iteration can be simplified with no loss of the optimal convergence guarantee. *Value iteration* places a restriction on the number of backups[1] allowed in the policy evaluation step, and as a result, it significantly speeds up convergence.

However, DP algorithms are of little practical use these days since they require a perfect model of the problem to backup actions taken (allowing them to explore the reward of all actions from a state). Furthermore, even with optimized algorithms such as value iteration, DP algorithms are very computationally intensive when compared to other algorithms available for solving the reinforcement learning problem.

### 2.2.2 Monte Carlo Methods

Monte Carlo methods differ from DP methods in that they learn from experience only (real or simulated interactions with the environment). Furthermore, Monte

[1] Performing a backup means undoing an action

Carlo methods learn from episode to episode - they are methods of averaging complete rewards, and so cannot be applied to continuous tasks. Learning from only experience using these methods has the same power as DP and optimal policies can still be learnt.

A state's value is learnt by averaging the rewards that result from the state, as given by interactions with the environment when following policy $\pi$. As the value is averaged over more rewards, the value tends to the expected value. If this is done for all the states $s \in S$ then the estimate of the state value function $V^\pi(s)$ will tend to the expected value. However, as there is no longer a model of the environment, it is not possible to backtrack in each state and choose the best action. For this reason $Q^\pi(s,a)$ is learnt instead. Now rewards are averaged for each combination of $s \in S$ and $a \in A$. To learn in this manner, experience must be available for all action-state combinations.

With $Q^\pi(s,a)$ known, iterative methods, similar to those applied to DP, can be used again. That is, over $k$ iterations a policy $\pi_k$ is evaluated, by learning $Q^{\pi_k}(s,a)$, then $Q^{\pi_k}(s,a)$ is used to generate a new policy $\pi_{k+1}$. The new policy $\pi_{k+1}$ is created from the expected value function as such:

$$\pi_{k+1}(s) = \arg\max_a Q^{\pi_k}(s,a)$$

The policy generated on the $k+1$ iteration is guaranteed to be better than the $k$ iteration policy by the policy improvement theorem (Sutton and Barto, 1998, Section 4.2), unless both policies are optimal. Therefore, the optimal policy will be found after a finite number of iterations since there are only a finite number of policies. In practice, some exploration must be included in the new policy so all combinations of $s$ and $a$ are still experienced. Methods of accomplishing this are discussed in the next section.

### 2.2.3 Temporal Difference Learning

Temporal difference methods use aspects of both dynamic programming and monte carlo approaches. They are similar to monte carlo methods in that they learn from experience that is derived from interacting with the environment. However, they are similar to DP methods in that they update the value function estimate at every time step instead of waiting to discover the true discounted reward. Because of this DP and TD methods are said to *bootstrap*.

Once an action has been performed on the environment, a TD method must only wait until the following time step to update its value function. The update rule uses the difference between the two estimates to form a more accurate estimate:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.7}$$

When learning the action-state value function the update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.8}$$

When $s_{t+1}$ is a terminal state, then $V(s_{t+1})$ and $Q(s_{t+1}, a_{t+1})$ have the value zero. This update rule uses experience tuples $< s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} >$ which gives rise to the name *Sarsa* (Rummery and Niranjan, 1994) for the algorithm. A procedural description of Sarsa is given in algorithm 2.1.

Initialize $Q(s, a)$ arbitrarily
Repeat for each episode:
  Initialize $s$
  Choose $a$ from $s$ using policy derived from $Q$ ($\epsilon$-greedy)
  Repeat for each step of the episode
    Take action $a$, observe $r$,$s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ ($\epsilon$-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'; a \leftarrow a';$
  until $s$ is terminal

**Algorithm 2.1**: Procedural description of the Sarsa algorithm (Sutton and Barto, 1998, Section 6.4)

The learning rate $\alpha \in [0, 1]$ determines the extent to which the existing estimate of $Q(s, a)$ contributes to the new estimate.

For any TD algorithm to converge to the optimal policy, experience must be provided for all states and possible actions. To accomplish this, when a policy is derived from Q (line 4 and 7 of algorithm 2.1), it is altered slightly to give some bias towards exploratory actions, thereby increasing the chance that all combinations of $s$ and $a$ will be experienced. This decision of when to take greedy (best) action choices and when to take random exploratory actions is known as an *exploration strategy.* The most common exploration strategy is the *$\epsilon$-greedy* policy Watkins (1989), which takes random actions with probability $\epsilon$ and greedy actions otherwise.

If over time the policy followed by the agent tends towards greedy choices in the estimated value function $Q(s, a)$, then the $Q(s, a)$ values will eventually converge to $Q^{\pi^*}(s, a)$, the value function for the *optimal* policy $\pi^*$.

It is said that the Sarsa algorithm is on-policy. This means that an action-state's value is updated using the action that was chosen by the policy, not the best action. Q-Learning (Watkins, 1989) is an off-policy TD control algorithm, i.e. the best action is always used in the update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.9}$$

Therefore, regardless of the policy being followed, $Q$ will converge to $Q^*$ providing all action-state combinations continue to be visited. A procedural description of Q-learning is given in algorithm 2.2.

### Eligibility Traces

*Eligibility traces* (Sutton, 1988) are an enhancement that can be applied to any TD algorithm to speed up learning. Depending on the trace parameters, a TD method using eligibility traces can replicate the behavior of MC methods (complete episode updates), one-step TD methods (such as those described in section 2.2.3) or any n-step update. For this reason they are considered as a method for unifying MC and TD methods.

```
Initialize Q(s,a) arbitrarily
Repeat for each episode:
    Initialize s
    Repeat for each step of the episode
        Choose a from s using policy derived from Q (ε-greedy)
        Take action a, observe r,s'
        Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
        s ← s';
    until s is terminal
```

**Algorithm 2.2**: Procedural description of the Q-learning algorithm (Sutton and Barto, 1998, Section 6.5)

An eligibility trace $e_t$is associated with every action-state pair $e_t(s,a)$ determining that pair's eligibility to benefit from any reward that the agent might receive. Informally, the eligibility trace for a pair determines the part that pair played in the agent receiving a reward. The eligibility trace for each action-state pair is initialized to zero, $e_0(s,a) = 0$.

When eligibility traces are applied to Sarsa (section 2.2.3) the update rule is changed to the following:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]e_t(s,a) \quad (2.10)$$

where $e_t(s,a)$ is defined as:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & : \text{ if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s,a) & : \text{ otherwise} \end{cases} \quad (2.11)$$

Equations 2.10 and 2.11 must both be updated for all pairs $(s,a)$ at each time step. The value of $\lambda \in [0,1]$ in equation 2.11 determines the *decay rate* of the eligibility traces. When the state-action pair an eligibility trace represents occurs during the agent's interactions with the environment, its value is incremented. This is to indicate that this state is partly responsible for any reward given in the near future (the near future is defined according to $\lambda$). Traces updated according to 2.11 are said to be *accumulating traces*.

A second type of trace methods are *replacing traces*. With accumulating traces, if a state-action pair is visited again before the trace has decayed to 0 then it will be incremented to more than 1. If the same situation is encountered when using replacing traces then the trace value will be reset to 1. The update rule for replace traces is as follows:

$$e_t(s,a) = \begin{cases} 1 & : \text{ if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s,a) & : \text{ otherwise} \end{cases} \quad (2.12)$$

Using replace traces in some domains can produce a significantly faster learning speed when compared to accumulate traces (Singh and Sutton, 1996).

### Function Approximation

In many domains, a problematic feature of reinforcement learning algorithms is the representation of the state space as a finite set $S$ of named states $s_i$.

For more complex learning domains, an explicit enumeration of all the possible states of the system will be enormous. Alternatively, the state may be described by continuous variables, resulting in an infinite number of possible states.

To use reinforcement learning in these domains, an enumeration of the state space is avoided by describing each state in terms of a finite set of *state features*. It is often the case that states with similar state features have similar value and/or require a similar action to be taken. To take advantage of this property, we can use *function approximation* techniques to learn an approximate version of $Q^{\pi^*}(s,a)$. Note that by using a function approximator the problem is no longer Markov, however as stated in section 2.1.1 so long as it is an approximation of an MDP, the theory can still be used.

*CMAC* (Albus, 1981) or *tile-coding* is a *linear* approximation method. A linear approximator uses a set of $n$ basis functions $\{\phi_f(s)\}$ and a set of $n$ parameters $\{\theta(f)\}$ to express the approximation of the value function as:

$$\widetilde{V}(s) = \sum_{f=1}^{n} \theta(f)\phi_f(s) \tag{2.13}$$

Each basis function $\phi_f$ can be interpreted as a single state feature. A CMAC approximator is based on several sets of features, each set being an *exhaustive partition* of the state space. Each of these sets is known as a *tiling*. A *tile* is a binary feature, which has value 1 when the state falls within its area, and value 0 otherwise. The tilings are all offset by different amounts in the state space (see Figure 2.1), improving the generalisation achievable by the approximator. The set of features $f$ *present* or *active* in a state $s$ are those that for which $\phi_f(s)$ has a value of 1.

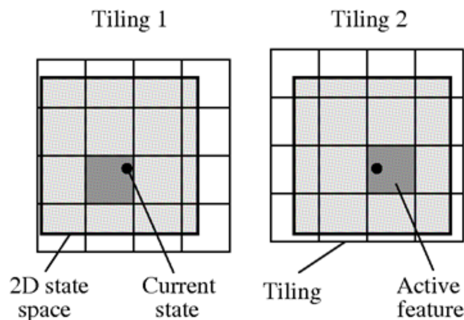


Figure 2.1: CMAC uses overlapping tilings of binary features

Since TD and DP methods use estimates to update the value function, one might be concerned that when further estimates are introduced (i.e. from a function approximator), the estimates worsen or even diverge. However, Sutton (1996) has shown that when CMACs are applied to TD methods the resulting behavior is very stable.

To increase the effectiveness of tile coding it can be combined with *hashing* techniques to reduce the memory requirements of the tiling. Furthermore, tilings can be represented by other structures instead of grids. For example, in problems where the state space spans many dimensions, the "curse of dimensionality"

(Bellman, 1961) can be avoided by ignoring some of the dimensions in the tiling or by considering some dimensions in less detail than others.

**Full Sarsa Algorithm with Enhancements**

The Sarsa algorithm described in section 2.2.3 can be extended with eligibility traces and function approximation to give an approach called *linear, gradient-descent Sarsa($\lambda$) with binary features.* A procedural version is shown in algorithm 2.3. This is the algorithm that will be adopted for the parallel learners studied in this project.

```
Initialize θ⃗ arbitrarily
Repeat for each episode:
    e⃗ = 0⃗
    s, a ← initial state and action of episode
    F_a ← set of features present in s, a
    Repeat for each step of the epsiode:
        For all i ∈ F_a:
            e(i) ← e(i) + 1 (accumulating traces)
            e(i) ← 1 (replacing traces)
        Take action a, observe reward, r, and resulting state s
        δ ← r − Σ_{i∈F_a} θ(i)
        With probability 1 − ε:
            For all a ∈ A(s):
                F_a ← set of features present in s, a
                Q_a ← Σ_{i∈F_a} θ(i)
            a ← arg max_a Q_a
        else
            a ←a random action ∈ A(s)
            F_a ← set of features present in s, a
            Q_a ← Σ_{i∈F_a} θ(i)
        δ ← δ + γQ_a
        θ⃗ ← θ⃗ + αδe⃗
        e⃗ ← γλe⃗
    until s is terminal
```

**Algorithm 2.3**: Full Sarsa algorithm (*linear, gradient-descent Sarsa($\lambda$)*) with eligibility traces and function approximaton (Sutton and Barto, 1998, Section 8.4)

## 2.3 Evaluation Domain

The performance of any parallelized learning algorithm developed will be assessed using the *Mountain-Car Task* (Moore, 1991; Sutton and Barto, 1998, Section 8.4), which has emerged as a well-known benchmark for reinforcement learning methods.

A car situated in a steep-sided valley between two mountains must learn how to reach the goal at the top of one of the mountains (see Figure 2.2). The car's engine is not powerful enough to accelerate up the mountain from rest. Instead the car must reverse part of the way up the opposite hill, then accelerate forward to achieve sufficient inertia to reach the goal.

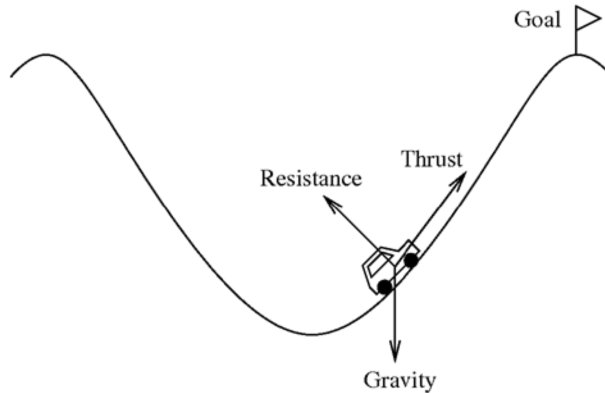


Figure 2.2: The Mountain-Car Task

The state of the mountain car problem is described by the position $x_t$ and velocity $v_t$ of the car. There are only three actions, which set the car's acceleration $a_t$ to either $-1$, 0 or $+1$. At each time step, the state is updated according to a simplified physical model:

$$x_{t+1} = bound[x_t + v_t]$$

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

where *bound* is a function that enforces $x \in [-1.2, 0.5]$ and $v \in [-0.07, 0.07]$. When the car reaches the opposite side of the mountain from the goal, $x = -1.2$, the car's velocity is reset to 0. The altitude of the car in the valley is given by:

$$altitude_t = \sin(3p_t)$$

The mountain car problem can be formulated using two different strategies. The first is an *optimistic formulation* of the reward structure. In this formulation the feature values are initialized to 0, and a reward of $-1$ is received on all time steps. Note that even when the agent reaches the goal a reward of -1 is still received. This is somewhat contrary to the idea reinforcement learning where one expects a positive reward to be received when the agent accomplishes a goal. However, by structuring the rewards in this way the agent is encouraged to reach the goal as quickly as possible so as to minimize the negative cumulative reward it will receive. The action choice used with this formulation is greedy (i.e. always select the best action). This results in *implicit* exploration as actions thought initially to be optimal turn out to be sub-optimal, and are no longer selected.

The second strategy uses a *pessimistic formulation* of the reward structure. The more natural rewards of 0 on all time steps and 1 when the goal is reached are used instead. The initial values of the features are again set to 0, but in this formulation the true value of all features will be greater than 0. Because of this no implicit exploration occurs during learning and exploratory actions must be introduced by using an $\epsilon$-greedy exploration strategy.

Given that the state space of this domain is continuous, function approximation must be used to estimate the value function. Tile coding will be used as the method of function approximation. For each of the three actions, a CMAC approximator is used to estimate the state value space. Therefore, a feature/tile in the approximator will represent a quantity of the position and velocity state space.

To solve this problem in a reasonable amount of time, eligibility traces are required. Since features are used to represent the state-action value function, eligibility traces must be applied to the features. This amounts to storing an eligibility trace for each feature the function approximator represents. The update and decay operations used with state-action pairs remain unchanged when applied to features.

## 2.4 Previous Work on Parallelizing Reinforcement Learning

This project focuses on parallelizing the reinforcement learning process to speed up convergence, an approach that has so far received relatively little attention. There are however some works in similar subject areas and a brief overview of these is given.

Early work by Whitehead (1991) proposed the sharing of experience amongst learning agents using "socicially" inspired algorithms. Two methods were suggested. *Learning with an external critic* allows a more knowledgable agent to watch the system and generate partial rewards for the agent based on the actions it performs. These rewards help guide the agent towards the true rewards in the environment. However, by introducing a more knowledgable agent domain specific knowledge is required for learning, therefore reducing reinforcement learning's simplicity. The second method allows an agent to gain experience by watching others similar to itself. But this does not result in true parallelization, since experience tuples have to be processed by each agent after each learning step and this results in additional computations.

Bagnell (1998) used a similar approach of sharing experience tuples between robots in the same environment to learn the behavior for a robotic controller. The consequence of processing additional experience tuples is less severe in this environment since experience is relatively sparse compared to processing power.

Recent work by Kretchmar (2002) on parallelizing Q-learning discusses a method of combining the state value estimates from multiple agents to produce a more accurate estimate. The number of times an agent visits a state is used to weight how much its value estimate is worth in the combination. However, the method was only applied to a single state task, the n-armed bandit problem.

Wingate and Seppi (2004) used similar ideas to solve MDPs by dynamic programming. The state space is partitioned and distributed to parallel processors

which then use value iteration to solve their part. Each processor is assigned multiple partitions and an "intelligent" priority ranking system is used to determine which partition should be partially solved. Once this is complete, the new values of the partition are communicated to the other parallel processors. The algorithm shows a significant speed up over many domains.

# Chapter 3

# Problem Analysis and Design

This chapter describes the various algorithms for parallel learning considered during this project. For each algorithm a description of its operation is given first, then results of its performance are presented and finally, a discussion of the results is given to guide the design of the next solution.

## 3.1 Basic Algorithm

The basic structure of all the solutions described in this chapter is very similar, and is shown diagrammatically in figure 3.1. That is, a set of independent learners are run in parallel and allowed to execute for a limited episode (stage 1 in figure 3.1). In this sense an episode is limited by imposing a cutoff on the maximum number of time steps the episode may run for. The agent's function approximators are transferred to an arbitrator (stage 2). A new function approximator (FA) is then generated using knowledge from one or more of the agent's function approximators (stage 3). This new function approximator is distributed to all the parallel agents (stage 4) and the process repeats. An arbitrator is introduced to control the collecting of function approximators, the generation of the new FA and the distribution of the new FA to the parallel agents.

There are two key problems that must be considered for an effective implementation of the above procedure to be possible. One is how to determine the quality of a function approximator so as to ascertain how close to the optimal an agent's action-state value estimates are. Or in other words, establish a method to determine where a function approximator's expertise lies. The second problem is how to combine the expertise of each FA to produce a new FA which contains the sum of the most useful knowledge.

The pessimistic formulation (section 2.3) was not considered until later designs, so all early designs are applied to the optimistic formulation only.

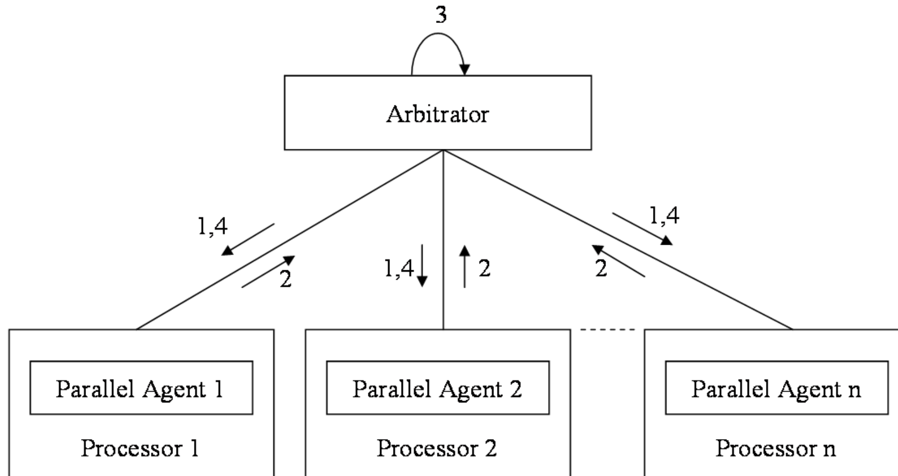


Figure 3.1: A diagram depicting the operation of the parallel algorithms, the stages are explained in section 3.1

## 3.2 Algorithm Evaluation Method

To evaluate the performance of an algorithm, the following performance measures will be used (the reader is referred to section 4.1 for a more complete description). In each parallel algorithm five parallel agents are used.

To compare the performance of the parallel algorithm against the single agent algorithm, three different approaches are used to compare the learning effort expended by the two methods: *equal computation time in episodes*, *equal computation time in steps* and *equal execution time*.

Suppose there are $l$ parallel learners, each of which has learned for $n$ episodes. The group of learners *as a whole* has consumed $ln$ episodes. Therefore one possible comparison is with a single agent learner which has learned from $ln$ episodes. This comparison is termed *equal computation time in episodes* and indicates the relative performance of the parallel method if each agent had to be implemented *sequentially* on a single processor. Note the scale of these graphs is in $ln$ episodes.

The *equal computation time in steps* method allows the single learner to only complete $n$ episodes but each episode lasts for $l$ times longer, i.e. the step cutoff of the single agent is the step cutoff of a parallel agent multiplied by $l$. Again this indicates the relative performance of the parallel algorithm if it were implemented sequentially.

The third method compares the parallel learner with a single learner which has only learned from $n$ episodes, i.e. the same number as *only one of the parallel agents*. This comparison is called *equal execution time*. This indicates the relative performance of the parallel method if each episode is performed *in parallel* on $l$ different processors. Note that the overhead resulting from the merging process is not included in any of the evaluations.

After learning, the resulting policy is evaluated by measuring its performance

over 100 episodes. Each method is evaluated on the same set of randomly generated initial states. No learning takes place during the evaluation, so a time limit of 5000 time steps per episode is imposed. At the end of the evaluation, the mean number of steps per episode is calculated for each method.

The learning and evaluation phases described above are repeated 50 times and the mean of the mean number of steps per episode is calculated. The standard error of this result is also calculated and is shown on the following graphs using error bars.

## 3.3 Maximum Altitude Approach

A very simple approach to solving the problems outlined at the beginning of this chapter is to use the maximum altitude an agent in the valley reaches during the learning stage as an indication of the quality of a function approximator. The best function approximator determined in this way will be distributed to all the other agents, overwriting their existing function approximator. Intuitively, the agent that reaches the highest altitude should be the most likely to escape the valley. A procedural description of this algorithm is given in algorithm 3.1. In the algorithm, $P$ is the set of parallel agents and $\theta_x$ represents the function approximator belonging to agent $x$. The function sarsaAlt($x$, stepCutoff) performs one episode, with a maximum of stepCutoff time steps, of the full sarsa learning algorithm (2.2.3) using agent $x$. The function returns the maximum altitude the agent reaches.

Initialize all $x \in P$
Repeat for each episode:
  $best = \arg\max_{x \in P} \text{sarsaAlt}(x, \text{stepCutoff})$
  For all $x \in P, x \neq best$:
    $\theta_x \leftarrow \theta_{best}$

**Algorithm 3.1**: Algorithm based on "maximum altitude"

Unfortunately, this algorithm does not perform well in practice. The results are given in figure 3.2 with equal execution time at the top, equal computation time in episodes in the middle and equal computation time in steps at the bottom. The poor performance of the parallel agents results from the fact that the altitude an agent reaches is not a good indication of a function approximator's quality. For example, as discussed in section 2.2.3 the starting states for each episode are randomly chosen. If an agent's initial position is one with very high altitude and it promptly escapes, then the resulting FA will be ranked with high quality, even though it does not contain as much useful information than a low altitude agent that has explored until the step cutoff limit. However, a positive point of this algorithm is the very short time it takes to determine the best FA since the altitude calculation is very simple. This leads to two conclusions: the FA quality calculation should be quick so as to not interfere with the learning speed-up resulting from the parallel aspect of the algorithm. Secondly, the quality calculation should be an accurate reflection of a FA's true quality. Therefore, a tradeoff exists between the quality calculation accuracy and speed.

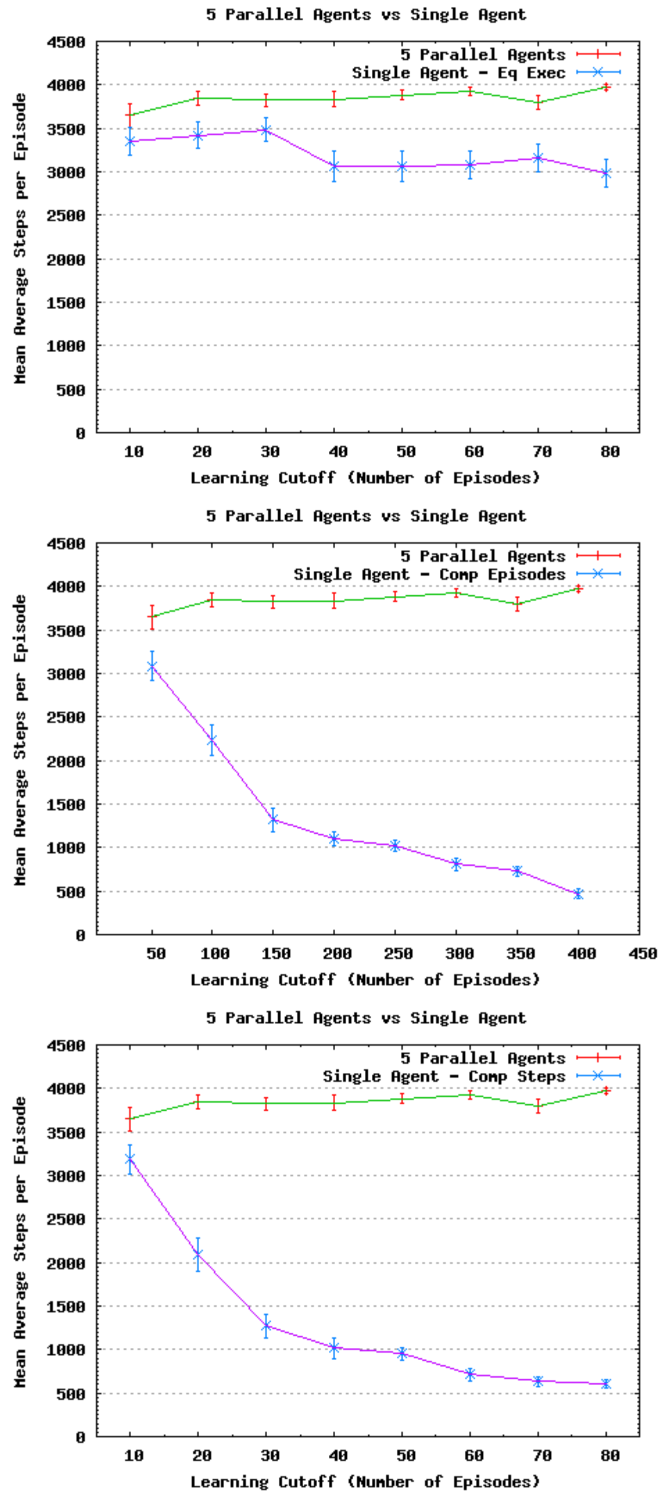


Figure 3.2: Results for the “maximum altitude” parallel learning algorithm

This test considered a “quick and dirty” calculation method sacrificing accuracy for speed.

## 3.4 Best Function Approximator Determined by Evaluation

The following tests weigh in favor of calculation accuracy to determine the best possible speedup given that the FA quality calculation time is not a factor. An FA’s quality is determined by performing sample episodes using that FA and computing the average number of steps taken per episode. The same (randomly generated) initial states are used with each FA’s quality calculation to ensure a fair comparison. A procedural description is given in algorithm 3.2. In this algorithm, sarsa($x$,stepCutoff) is as described in the previous section except it does not return the maximum altitude reached. The function evaluate($x$,$W$) performs a complete episode using agent $x$ for each test state in $W$ and returns the average number of steps per episode.

```
Initialize all x ∈ P
Repeat for each episode:
    For each x ∈ P
        sarsa(x, stepCutoff))
    Generate set W of test states
    best = arg max_{x∈P} evaluate(x, W)
    For all x ∈ P, x ≠ best:
        θ_x ← θ_best
```

**Algorithm 3.2**: Algorithm based on performing a full evaluation of each FA

The results of this approach are given in figure 3.3. The parallel agents do exceptionally well when learning has occurred for only a small number of episodes but fail to show any improvement as the number of learning episodes increases. This is probably due to the aggressive step cutoff value of 50 which allows a general approximation to be learned very quickly, but prevents the specialization required to further decrease the average steps per episode.

This algorithm ensures that the best function approximator is always chosen, at the expense of a very long accuracy calculation time. To obtain an accurate idea of how good an FA is it needs to be tested over at least 100 episodes. If five parallel agents are used then this is an overhead of 500 episodes per episode. However, the only intent of this experiment is to determine what is possible in the best case so the evaluation time can be ignored. Again, information from all the agents but one is discarded. However, in this algorithm the FA evaluation stage produces a reliable ordering between the quality of the various agent’s FAs, allowing a combination of the best to be produced. This will be shown in the next solution.

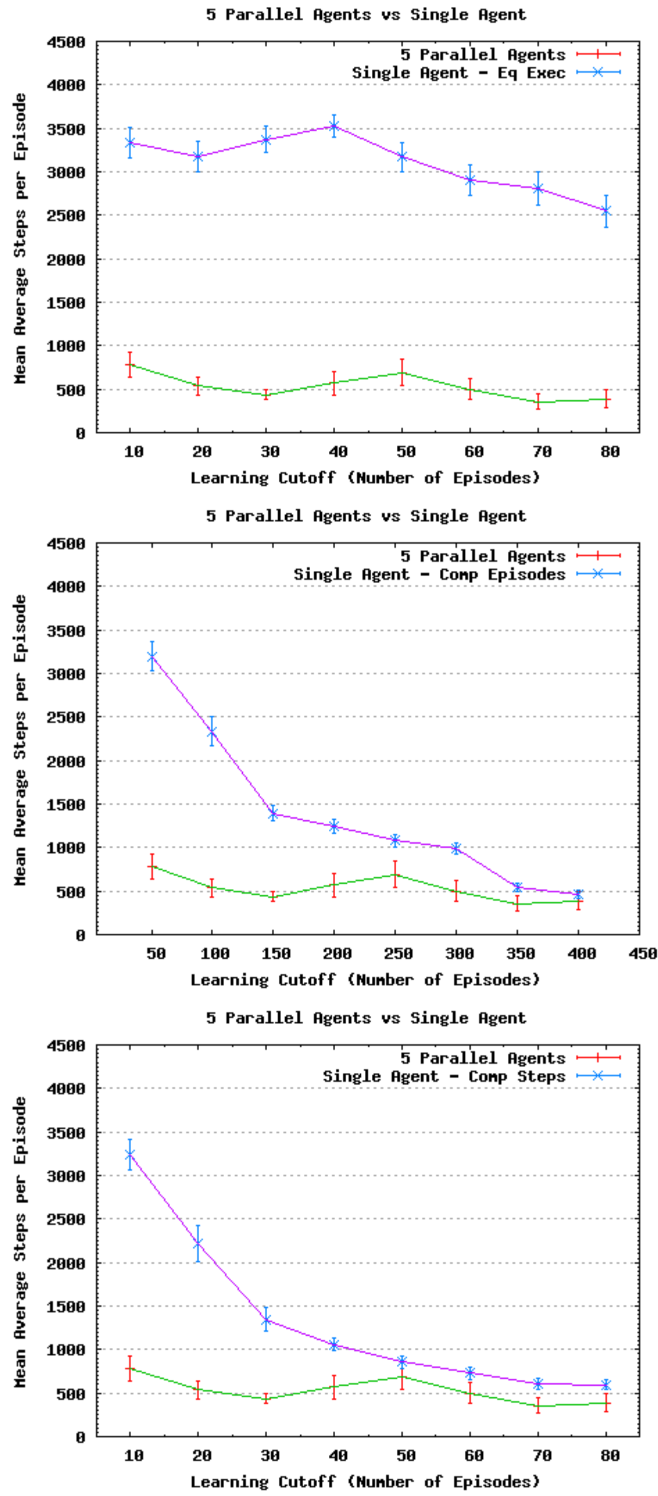


Figure 3.3: Results for the "evaluate and select best function approximator" algorithm

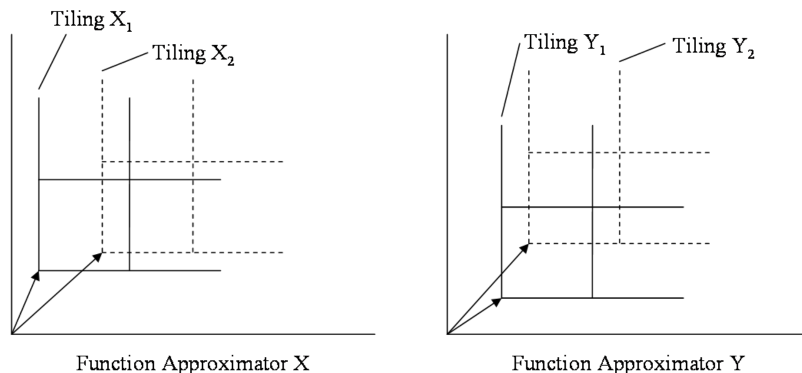


Figure 3.4: Distance calculation applied to two function approximators each with two tilings

## 3.5 Merge Best Function Approximators

### 3.5.1 Merge Algorithm 1: Match by Distance

This solution produces a new FA by combining knowledge from the best agent's FAs determined by the evaluation step described in the last section. However, as each agent is using a CMAC function approximator with its own tile offsets, the situation is complicated, i.e. the features represented by one function approximator are not those of another. To solve this a method of determining how similar two features are is required. However, all the features contained in a tiling are offset by the same amount (the tiling offset) so it is only necessary to determine how similar two tiles are. This can be implemented, with limited accuracy, as a distance calculation.

Therefore, the first iteration of the merging solutions involves matching up the tiles of CMACs as accurately as possible using the tiling's offset distance. For example, figure 3.4 shows a function approximator X with two offset tilings overlayed on it, and another function approximator Y with different offsets. The tiling labeled $X_1$ is at a smaller distance from the tiling origin than $X_2$ so FA X's tiling ordering is $[X_1, X_2]$. By the same logic FA Y's ordering is $[Y_1, Y_2]$. Therefore, $X_1$ should be matched up against $Y_1$ and $X_2$ against $Y_2$.

The updates are always applied to the best FA, so first the best FA is merged with the second best FA. The merging of the best and second best is then merged with the third best and so on in this fashion. The merging is performed in this way to give a higher implicit weight to the FAs merged earlier in the process. If $\theta_{best}$ is the best FA, $\theta_x$ is a different agent's FA, $f$ is a feature from $\theta_{best}$, $f'$ is a feature from $\theta_x$ and $f'$ is the closest feature to $f$ then the update is as follows:

$$\theta_{best}(f) \leftarrow \theta_{best}(f) + weight(\theta_x(f') - \theta_{best}(f)), \text{for all } f \in F_{best} \quad (3.1)$$

where $F_{best}$ is the set of features belonging to $\theta_{best}$. Weight determines how close $\theta_{best}$ will be brought to $\theta_x$. For example, if weight is set to 1 then $\theta_{best}$

will become equal to $\theta_x$. A procedural description of this algorithm is given in algorithm 3.3 for merging the top three function approximators.

Initialize all $x \in P$
Repeat for each episode:
  For each $x \in P$:
    sarsa($x$, stepCutoff))
  Generate set $W$ of test states
  $best = \arg\max_{x \in P}$ evaluate($x, W$)
  $second = \arg\max_{x \in P - \{best\}}$ evaluate($x, W$)
  $third = \arg\max_{x \in P - \{best, second\}}$ evaluate($x, W$)
  $\theta_{best}(f) \leftarrow \theta_{best}(f) + weight(\theta_{second}(f') - \theta_{best}(f))$, for all $f \in F_{best}$
  $\theta_{best}(f) \leftarrow \theta_{best}(f) + \frac{weight}{2}(\theta_{third}(f') - \theta_{best}(f))$, for all $f \in F_{best}$
  For all $x \in P, x \neq best$:
    $\theta_x \leftarrow \theta_{best}$

**Algorithm 3.3**: Algorithm for the first merging solution, also a framework for the later merging algorithms

The results of this algorithm are shown in figure 3.5. The second, third and fourth best FAs are merged into the best in that order with the respective weight values 0.5, 0.25 and 0.125. The results are very disappointing given that significantly better results were seen from the previous algorithm that performed no merging. Despite trying other merge combinations and weight values, the results could not be improved upon. Apart from the merging, the two algorithms are identical so in this case merging actually harms the performance of the algorithm. Exactly why the merging results in worse performance can be explained by the matching of tiles by distance. Depending on the tiling offsets in the merged FAs, this algorithm can end up matching tiles that are quite dissimilar, resulting in features being merged that do not represent the same state at all. Therefore, the next solution will consider a more accurate method of performing a merge operation.

### 3.5.2 Merge Algorithm 2: Sample Four Points

This approach is an attempt to improve the accuracy with which two features that represent slightly different aspects of the state-action value space can be merged. In the previous example this problem was solved by performing a distance calculation between the various tiles making up the function approximator. The new method works by computing points that lie within the feature that is to be merged and then sampling the values of these points on the function approximator. The values of these points are then averaged to determine the approximate value the function approximator assigns to the non-native feature. This is shown diagrammatically in figure 3.6. The dashed lines represent an overlay of the feature to be approximated (from the best FA) while the filled background represents the function approximator, $x$, that the feature value will be extracted from. A point is chosen at each corner of the feature, $p_i$, and the value of this point is given by $FA_x(p_i)$. Note that an action-state value is calculated by summing the value assigned to that action-state by each tiling in the

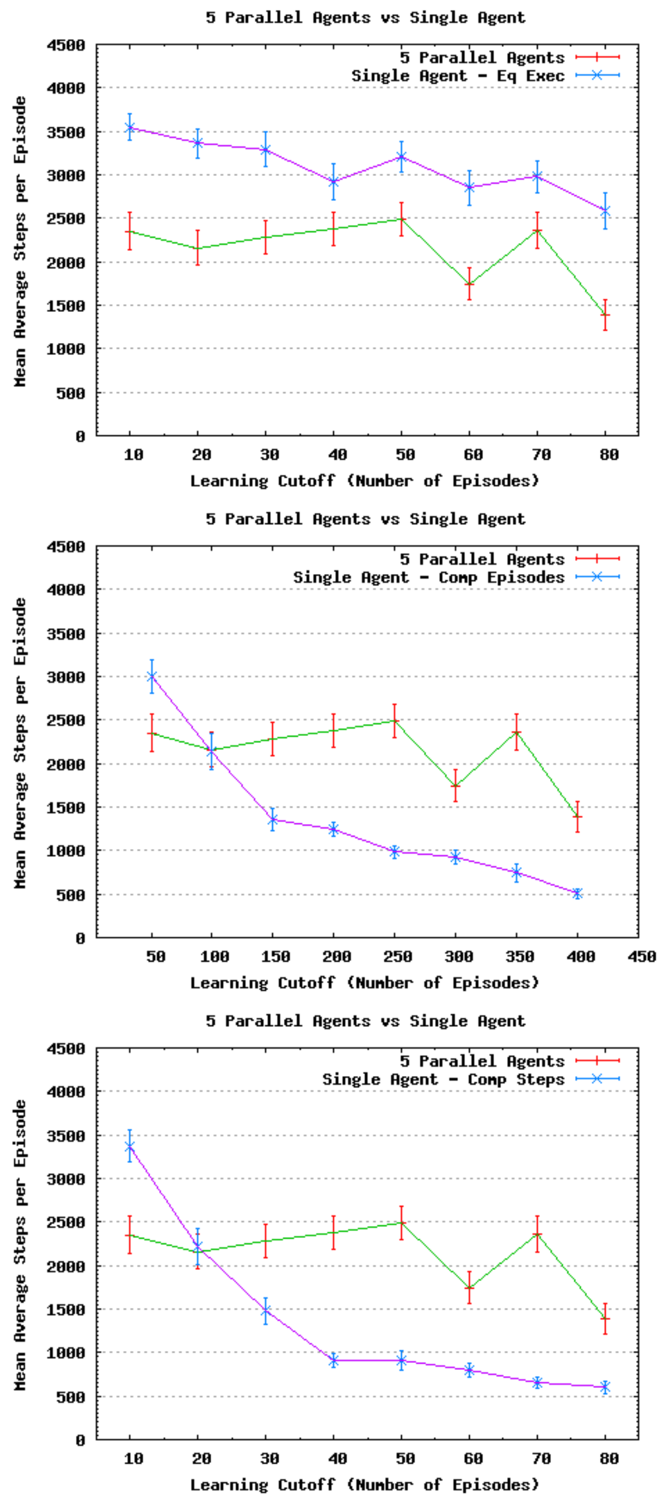


Figure 3.5: Results of merge algorithm 1

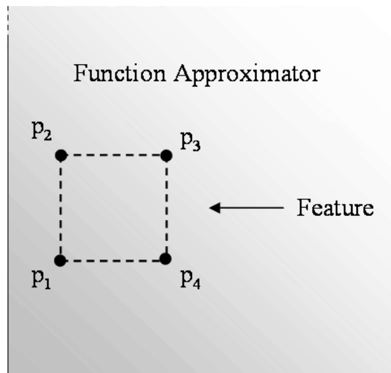


Figure 3.6: The points used to find the approximate value of a feature in merge algorithm 2

FA. Since it is a feature's value that is being updated,not the complete value of that state-action,the value must be scaled back into the appropriate range by dividing it by the number of titles present in the FA. Therefore, the update rule considered in the last section becomes the following:

$$\begin{aligned}\theta_{best}(f) \quad \leftarrow \quad & \theta_{best}(f) + \\ & weight(\frac{FA_x(p_1) + FA_x(p_2) + FA_x(p_3) + FA_x(p_4)}{4 * TILES} - \theta_{best}(f)), \\ & \text{for all } f \in F_{best}, \text{ where } p_1, p_2, p_3 \text{ and } p_4 \text{ are computed from } f\end{aligned}$$

Algorithm 3.3 remains unchanged in this approach with the exception of the new update rule. The results given in figure 3.7 are for merging the top four FAs with decreasing weight values as described in section 3.5.1. The results are again quite unimpressive however, some improvement is definitely seen over the previous merge method. It appears that the performance of the algorithm worsens slightly as the number of learning episodes increases. This is quite surprising and the only conclusion is that the merging procedure is detrimental to the overall quality of a function approximator. Nevertheless, the algorithm in the next section will try to remedy this by using a different feature approximation method.

### 3.5.3 Merge Algorithm 3: Sample Five Points

This method further refines the approach described in the two previous sections. The differences are (1) that a center point is sampled from the FA and (2) the positions of the points are chosen to give a more accurate estimate of the feature's value. The new points and positions are shown in figure 3.8. The reason for moving the sample points towards the center of the estimated feature is to reduce noise generated from tiles near the point. For example, in the previous example where the points are sampled right at the edge of the feature, a point's value is computed using the full number of tilings that make up the

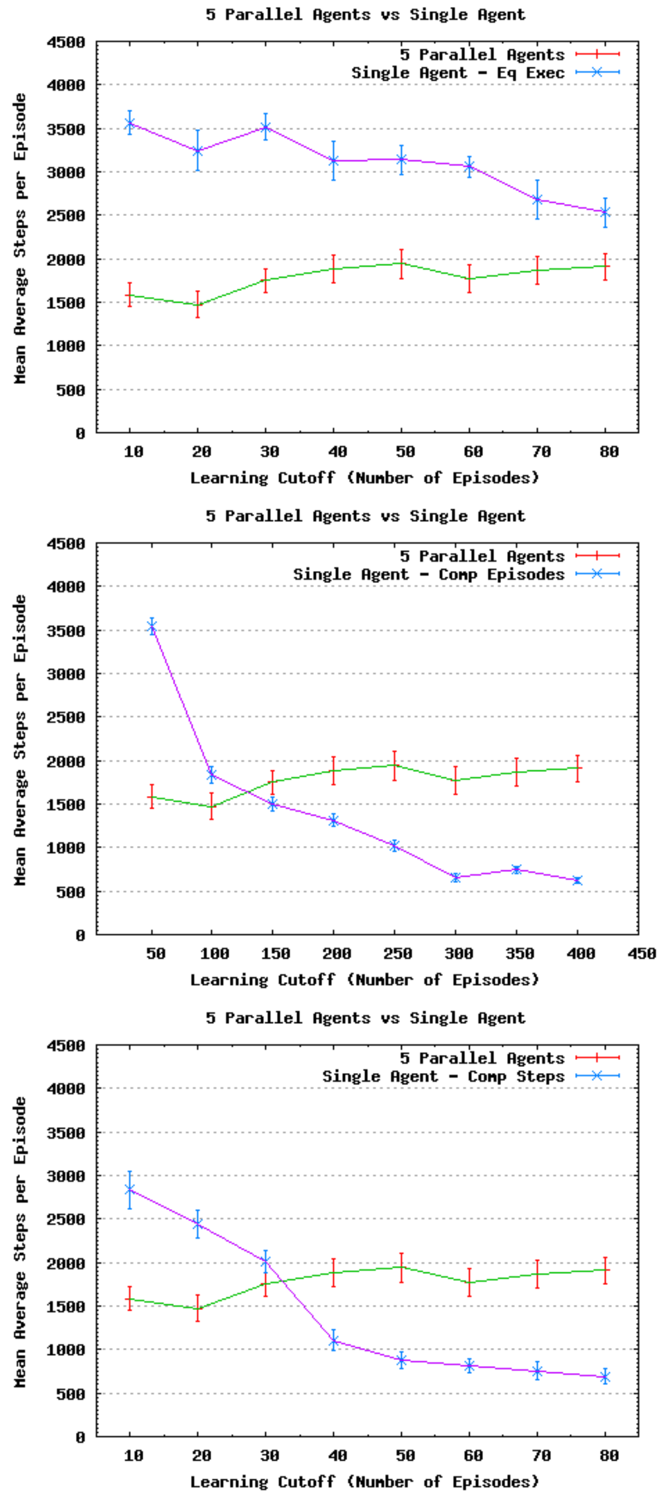


Figure 3.7: Results of merge algorithm 2

FA. This will probably include some tiles that only overlap with the feature a very small amount, thereby producing an error.

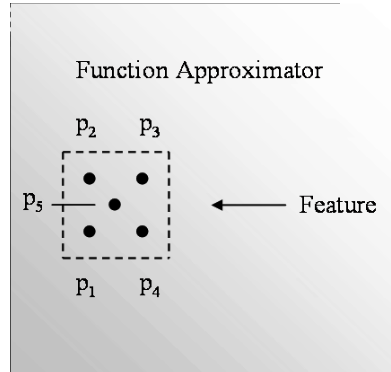


Figure 3.8: The points used to find the approximate value of a feature in merge algorithm 3

With the new points the update rule becomes the following:

$$\begin{aligned}\theta_{best}(f) \quad \leftarrow \quad & \theta_{best}(f) \\ & +weight(\frac{FA_x(p_1) + FA_x(p_2) + FA_x(p_3) + FA_x(p_4) + FA_x(p_5)}{5 * TILES} - \theta_{best}(f)), \\ & \text{for all } f \in F_{best}, \text{ where } p_1, p_2, p_3, p_4 \text{ and } p_5 \text{ are computed from } f\end{aligned}$$

Again, the algorithm remains unchanged from that described in the previous section apart from the update rule. Results are given in figure 3.9. Some improvement is observed over the previous merge algorithm in that the merge procedure no longer appears to worsen the quality of the function approximator over time. However, no there is also no improvement over time. Given the generally poor performance of the merge algorithms discussed in this section, a new solution will be investigated in the next section.

## 3.6 Using Minimum Feature Values to Create a New Function Approximator

In this section a technique is considered that uses a feature's value to determine how much experience an agent has interacting with the state space represented by that feature. The agent with the most experience will be the agent with the best estimate of the feature's value. This can then be used to build a new function approximator out of the best feature value estimates.

For example, if the optimistic problem formulation is used (section 2.3) then the initial value for each feature is set optimistically to zero. Therefore, the true value of any feature is less than zero and the agent with the lowest value estimate of a feature represents the agent with the most experience interacting with the environment designated by that feature. This agent therefore has the best estimate of that feature.

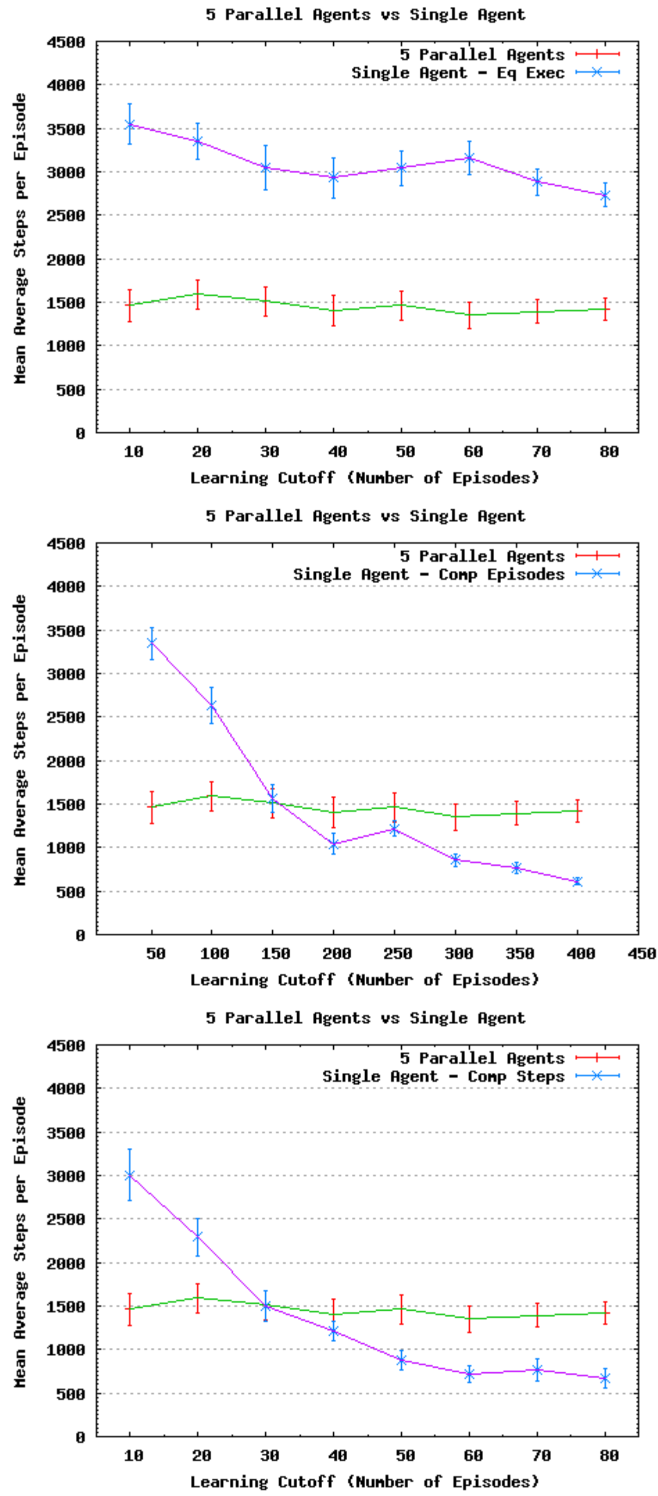


Figure 3.9: Results of merge algorithm 3

The merging stage occurs as follows, firstly, for a given feature each agent is asked for its current estimate of that feature. The feature's value in the new function approximator then becomes the lowest value returned. This is then repeated for each feature in the new function approximator. Note that since each function approximator has its own tiling offsets,the method described in section 3.5.3 must still be used to determine the approximate value of a feature not directly represented by a FA.

By using this method the new function approximator will contain the best knowledge each agent has to offer. By using this implicit experience calculation, all the time-consuming evaluation steps described in the previous sections are removed from the algorithm. Therefore, this approach provides a way to accurately and quickly approximate the evaluation stage held in the previous algorithms. A procedural description of this approach is given in algorithm 3.4.

Initialize all $x \in P$
Repeat for each episode:
  For each $x \in P$:
    sarsa($x$, stepCutoff))
  For each $f \in F_{new}$, where $p_1, p_2, p_3, p_4, p_5$ are computed from $f$:
    $\theta_{new}(f) = \min_{x \in P}(\frac{FA_x(p_1)+FA_x(p_2)+FA_x(p_3)+FA_x(p_4)+FA_x(p_5)}{5*TILES})$
  For each $x \in P$:
    $\theta_x \leftarrow \theta_{new}$

**Algorithm 3.4**: Parallel Learning Algorithm that uses agent's experience to merge all the FAs

The results of this algorithm are given in figure 3.10. Considering that this algorithm does not require the lengthly evaluation process required in the previous algorithm, the performance it manages to achieve is reasonable. The algorithm just manages to outperform the single learner with equal execution time which shows there is some potential for this type of merging algorithm. However, the method used in merge algorithm 3 for extracting a non-native feature's value from a FA is still required since the agents use FAs that represent different features. This problem will be addressed in the next solution.

## 3.7 All Function Approximators Represent the Same Set of Features

In the previous approaches each parallel agent's function approximator was setup independently, meaning that each FA represented a different set of features. This lead to the methods described above for merging two function approximators that do not share the same state features. By setting up the agents' function approximators in the same way, each FA will use the same set of state features $\{\phi_f(s)\}$. Therefore, estimating the value of a feature that is not native to a function approximator need no longer be considered, simplifying the merging problem to combining the parameter vector $\theta$ from each of the parallel learners. This removes the approximate merging performed in the previous algorithms and as a result this algorithm performs significantly better.

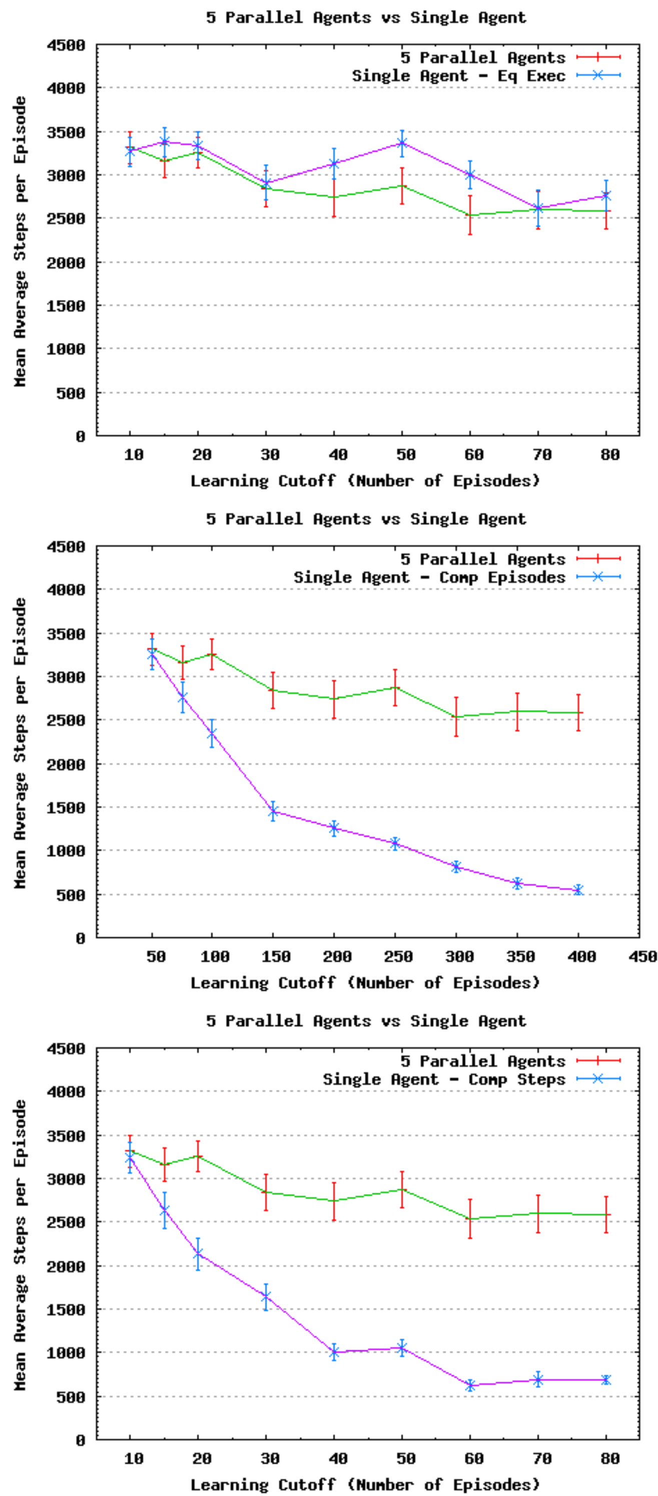


Figure 3.10: Results of the “minimum feature value” algorithm

### 3.7.1 Algorithm for the Optimistic Formulation

First a single set of random offsets are generated for the CMAC tilings, which forms the basis of a *template* approximator to be used for all the agents. Next, each learner completes a learning trial that is cut-off as discussed before. After the round of trials is complete, a new approximator is created, again based on the template. Each feature of this new approximator is set to the minimum value of that feature as given by the set of all parallel learners' approximators. The merged approximator is then distributed to all agents and the process repeated. The new function approximator's features are set according to the following rule:

$$\theta_{new}(f) = \min_{x \in P} \theta_x(f), \text{for all } f \in F$$

```
Initialize all x ∈ P using θ_template
ε = 0;
Repeat for each episode:
    For each x ∈ P:
        sarsa(x, stepCutoff, ε)
    For each f ∈ F:
        θ_new(f) = min_{x∈P} θ_x(f),
    For each x ∈ P:
        θ_x ← θ_new
```

**Algorithm 3.5**: Algorithm for combining function approximators that represent the same features (optimistic case)

The algorithm for this is given in algorithm 3.5. Note that a new Sarsa function, sarsa($i$, Cutoff, $\epsilon$), has been introduced to allow both the algorithms for the optimistic and pessimistic formulation to be expressed in the same notation. This new function takes an extra parameter $\epsilon$ which determines the proportion of exploration to exploitation in the $\epsilon$-greedy policy.

### 3.7.2 Algorithm for the Pessimistic Formulation

This algorithm is the first to consider the pessimistic formulation of the mountain car problem. It is identical to the algorithm for the optimistic formulation except it uses an exploratory strategy and merges based on maximum feature values (algorithm 3.6). This is because all the feature values are initialized pessimistically, so the maximum value for a feature represents the best estimate.

Preliminary results from these algorithms are shown in figure 3.11 and 3.12 for the optimistic and pessimistic formulations respectively. It is clear from the graphs that the performance of these algorithms is very impressive, especially for the pessimistic formulation which manages to outperform the single learners in all the tests - even when the single learner is allowed equal computation time. In the next chapter a full investigation into the performance of these algorithms will be performed.

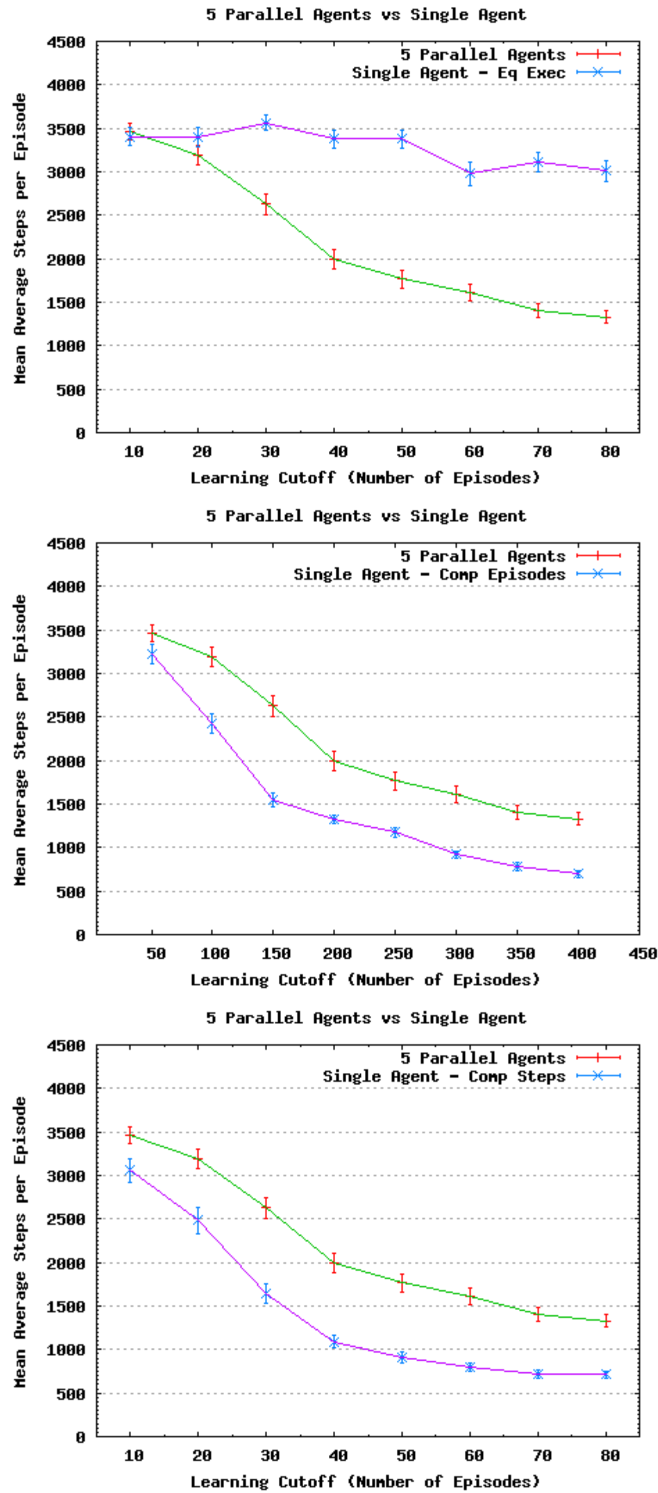


Figure 3.11: Results of the "optimistic minimum feature value" algorithm where all agents use the same FA

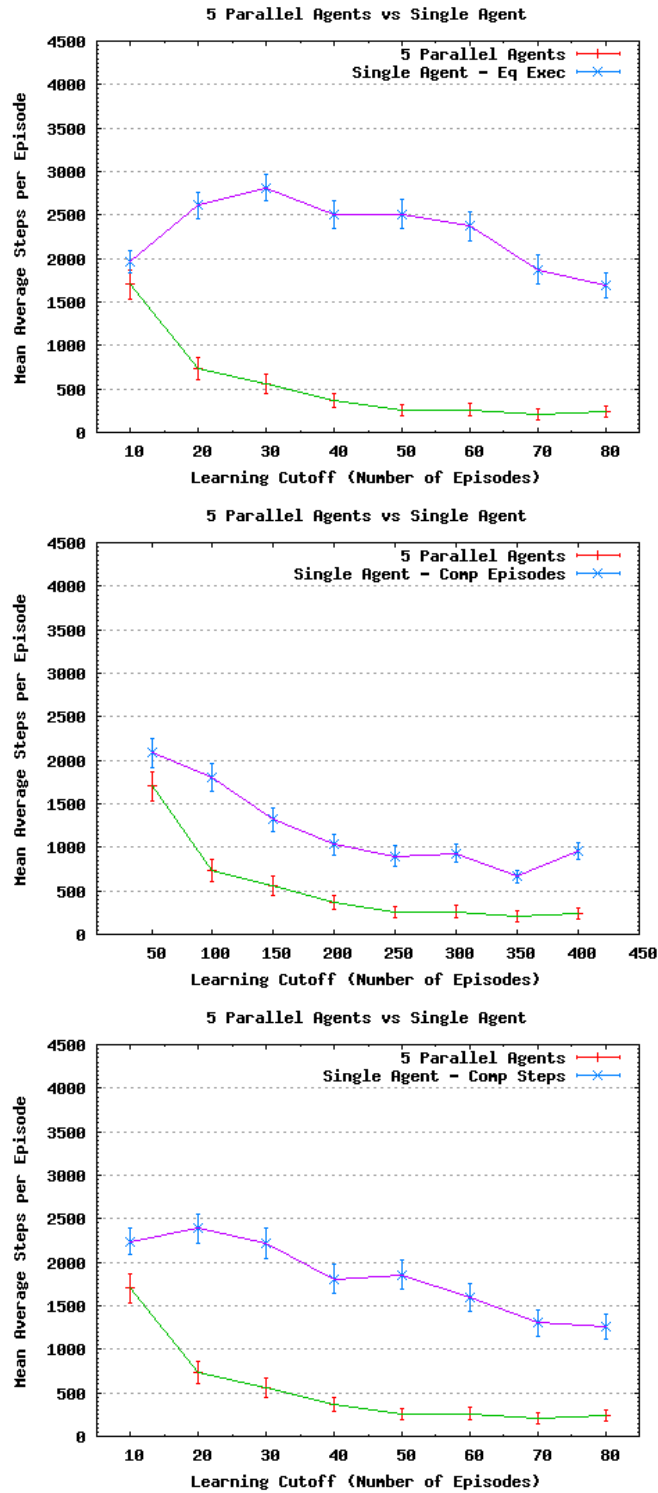


Figure 3.12: Results of the "pessimistic minimum feature value" algorithm where all agents use the same FA

Initialize all $x \in P$ using $\theta_{template}$
$\epsilon = 0.1$;
Repeat for each episode:
  For each $x \in P$:
    sarsa($x$, stepCutoff, $\epsilon$)
  For each $f \in F$:
    $\theta_{new}(f) = \max_{x \in P} \theta_x(f)$,
  For each $x \in P$:
    $\theta_x \leftarrow \theta_{new}$

**Algorithm 3.6**: Algorithm for combining function approximators that represent the same features (pessimistic case)

# Chapter 4

# Evaluation

In this chapter an indepth analysis of the best parallel reinforcement learning algorithm will be performed. The best performing parallel RL algorithm from the previous chapter was the algorithm that merges native features based on the parallel agent's minimum/maximum feature value estimate.

The evaluation proceeds as follows. A description of the evaluation method that was partially described in section 3.2 will be given first. The important points are reiterated and some new evaluation methods are introduced. Next, the performance of the algorithm on the optimistic formulation of the mountain car problem will be analyzed. After this a similar analysis will be performed on the pessimistic problem formulation. Finally conclusions are drawn about the performance of the two algorithms and some possible improvements are suggested.

## 4.1 Evaluation Method

The comparison methods described in the previous evaluation section, *equal computation time in episodes*, *equal computation time in steps* and *equal execution time* are again used to compare the performance of the parallel algorithm to the single agent algorithm.

A new evaluation measure is introduced that tests the scalability of the algorithm. In this method the parallel algorithm is run with 2, 5, 10 and 20 agents and a single agent with equal execution time is included for comparison. This gives an insight into the kind of improvement that the parallel algorithm is capable of if a set of parallel processors are available.

Furthermore, both types of experiments described above are run for various values of the step cutoff parameter, to investigate how it affects the convergence speed. The step cutoff values chosen are 50, 75 and 100. This analysis is performed because the choice of step cutoff in the algorithms is arbitrary and therefore, experimental evidence is required to show the best value for a particular situation.

As before, after learning is complete the resulting policy is evaluated, however, in these tests the performance is measured over 300 episodes instead of 50. The number of times the learning and evaluation process is repeated is increased to 200 repetitions. The increase in these values from the previous

section is to provide stronger evidence for the performance gains seen from the parallel algorithm.

As discussed in the evaluation domain description (section 2.3), a CMAC function approximator is used to represent each of the 3 actions the agent can perform. Each approximator uses 10 tilings of 9x9 square tiles. The choice of parameters $\alpha = 0.05$ and $\lambda = 0.9$ were proposed for this problem by Sutton and Barto (1998). A value of $\gamma = 1$ was chosen because all episodes finish after a finite number of steps. The $\epsilon$-greedy parameter was $\epsilon = 0$ for optimistic starting values and $\epsilon = 0.1$ for pessimistic starting values.

## 4.2 Parallel Algorithm Applied to the Optimistic Problem Formulation

### 4.2.1 Comparison Against a Single Agent

The graphs in figures 4.1, 4.2 and 4.3 show a comparison of the parallel algorithm against a single learner for step cutoffs of 50, 75 and 100 respectively. The results clearly show the parallel algorithm outperforming the single agent when equal execution time is considered. The performance of the parallel algorithm scales linearly when the step cutoff is increased whereas the single agent shows a greater than linear increase. Despite this it never comes close to matching the performance of the parallel algorithm.

In the case of equal computation time the results are less impressive but understandable since the parallel algorithm is not expected to outperform the single agent when the same amount of processing is available to both methods. When equal computation time in steps is considered, the parallel algorithm is quite close in performance and the gap closes as the step cutoff is increased. Since the single learner with equal computation time in steps already performs quite long episodes, the conclusion is that the increase in episode length due to the step cutoff increase is not very useful to a learner that already runs for long episodes. However, since the increase in episode length applies to *each* parallel agent, the results are significant for the parallel algorithm and in the case of a step cutoff of 100, the performance almost reaches that of the single agent.

The equal computation time in episodes method outperforms the parallel algorithms by a sizeable margin for all values of step cutoff. For each value of step cutoff the difference between the two algorithms remains the same and a linear increase in performance is observed from the equal computation time in episodes method as the step cutoff increases.

When the results of the parallel algorithm with different step cutoffs are compared, a clear relationship between step cutoff and convergence speed can be seen. The data suggests that the relationship is linear, when the step cutoff is increased, so is convergence speed.

### 4.2.2 Scalability Results

The results in figure 4.4, 4.5 and 4.6 show the scalability of the parallel algorithm for step cutoffs of 50, 75 and 100 respectively. The speed up received from 2 and 5 parallel agents is significant compared to the single learner with equal execution time however, when the number of parallel agents is increased to

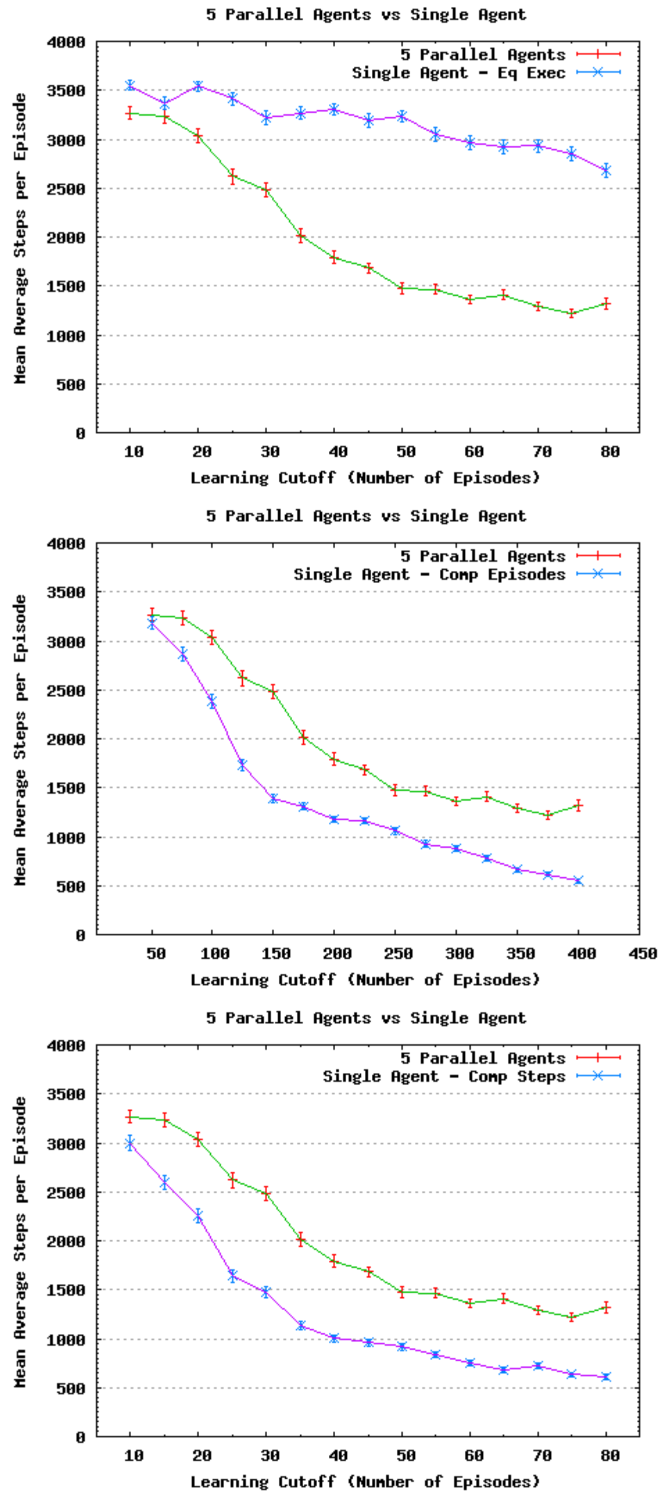


Figure 4.1: A comparison between a single agent and 5 parallel agents for a step cutoff of 50 using the optimistic formulation

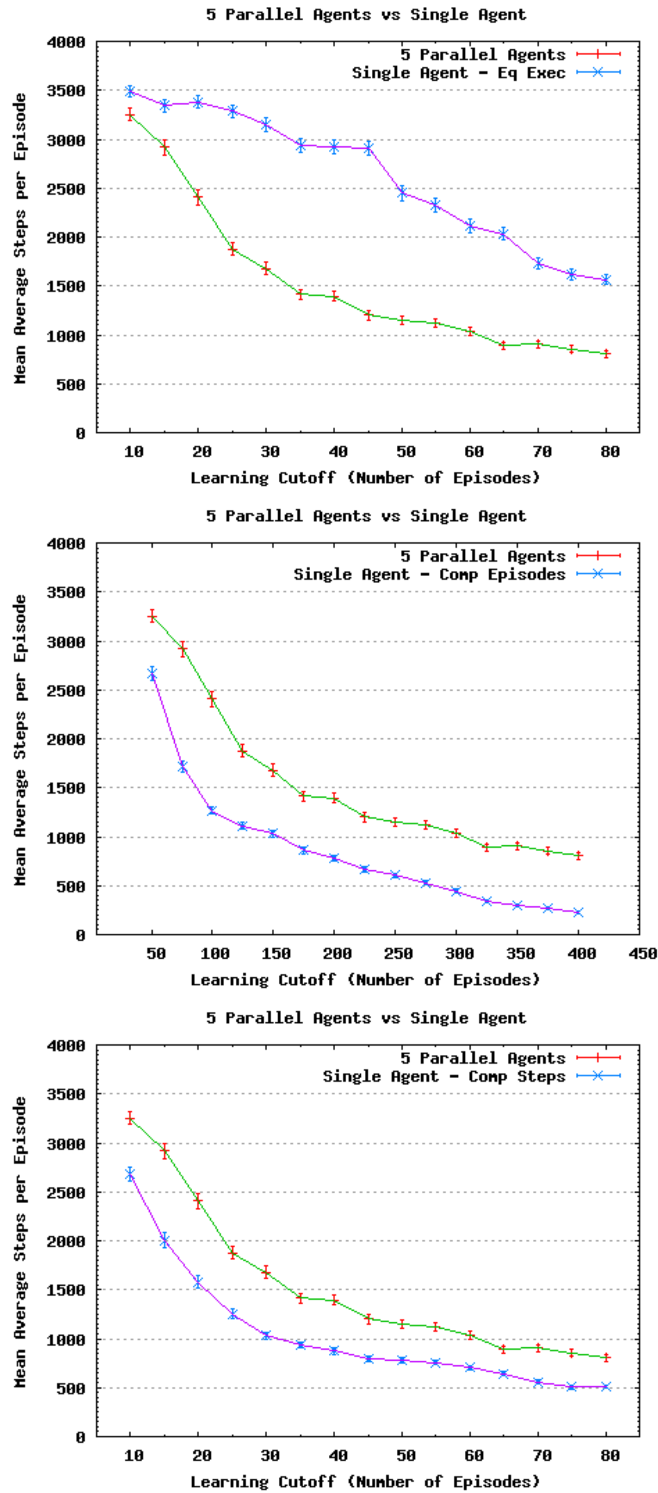


Figure 4.2: A comparison between a single agent and 5 parallel agents for a step cutoff of 75 using the optimistic formulation

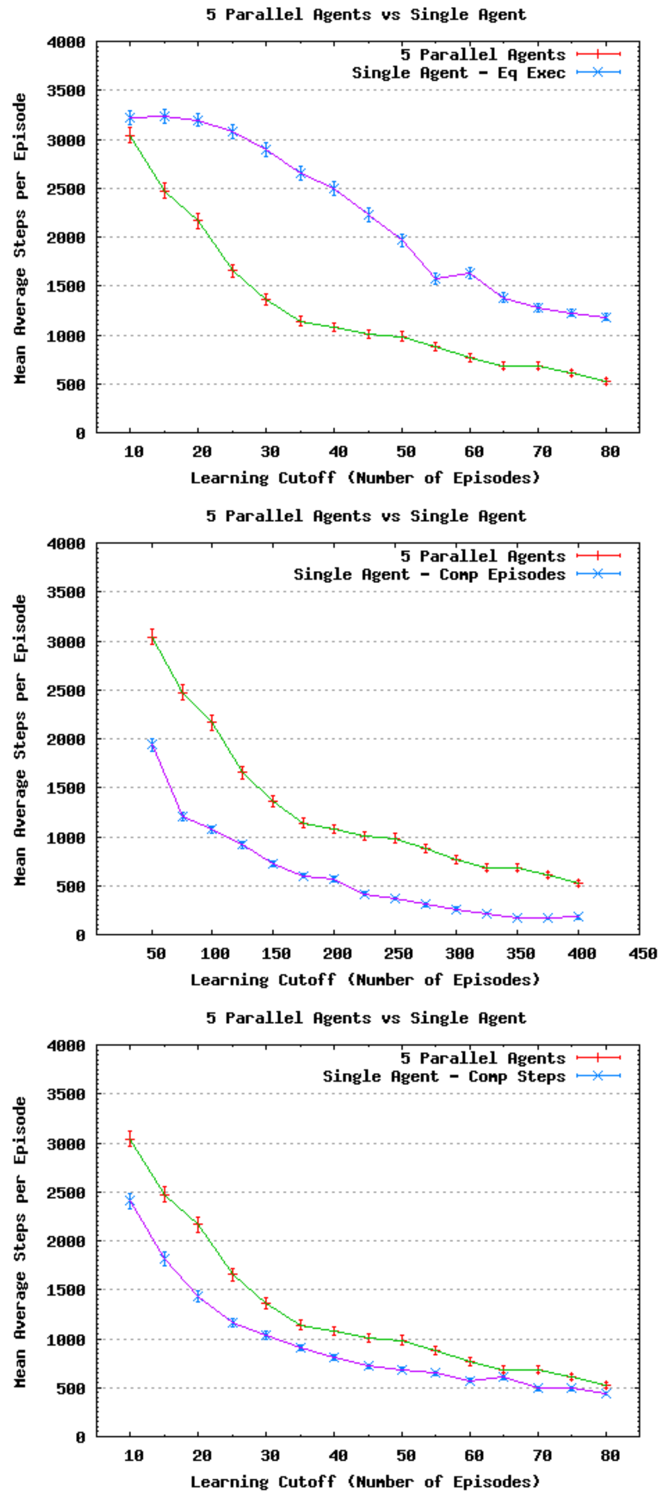


Figure 4.3: A comparison between a single agent and 5 parallel agents for a step cutoff of 100 using the optimistic formulation

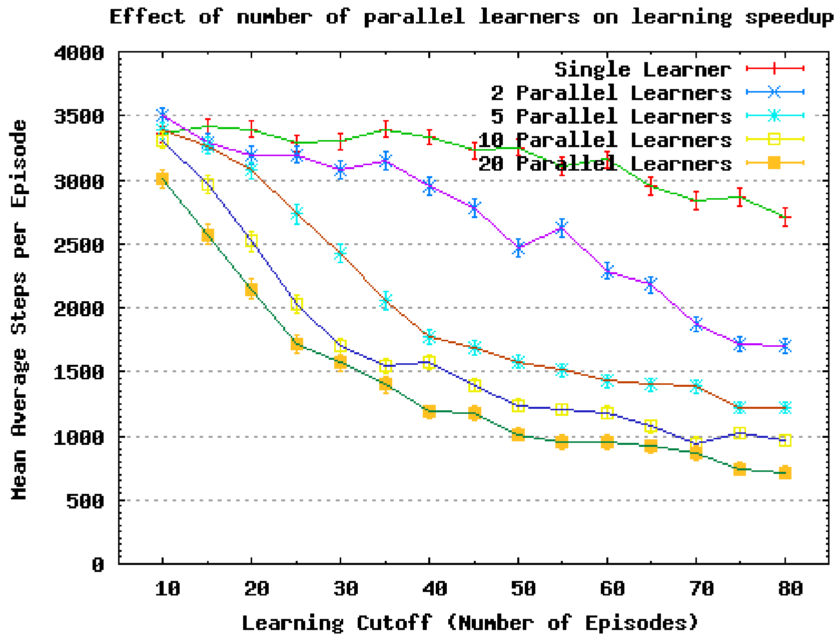


Figure 4.4: Scalability results for the optimistic problem formulation with a step cutoff of 50

10 and 20, far less improvement is observed. In fact, when the step cutoff is increased to 100, the algorithms with 5, 10 and 20 agents perform almost identically. Considering the amount of extra computation incurred by 20 parallel agents compared to 5 parallel agents, the scalability of the algorithm is not very impressive with high values of the step cutoff parameter. However, when the step cutoff is reduced the algorithm shows some potential for high scalability.

Given this and the results of the previous section it is clear that a balance must be archived between step cutoff and number of parallel learners in order to produce the maximum learning speed up for a limited set of resources. In this case resources are time and number of parallel processors.

## 4.3 Parallel Algorithm Applied to the Pessimistic Problem Formulation

### 4.3.1 Comparison Against a Single Agent

Figures 4.7, 4.8 and 4.9 show the results when the parallel algorithm is applied to the pessimistic problem formulation for step cutoffs of 50, 75 and 100 respectively. The results are surprisingly impressive this time, improving on or equaling the single agents performance in *all* evaluation methods.

In the equal execution time evaluation, the parallel algorithm outperforms the single agent by a significant margin, proportionally more than the parallel algorithm outperformed the single agent using the optimistic formulation. In

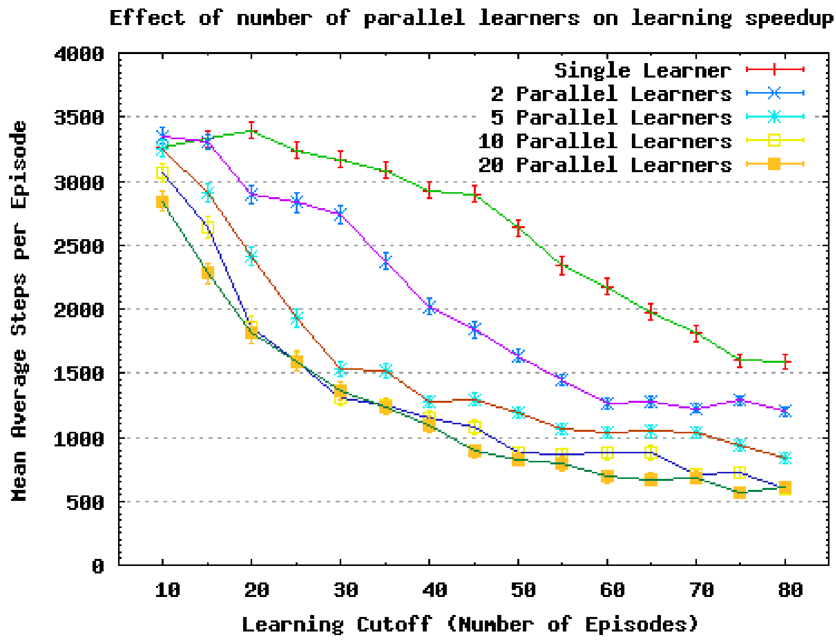


Figure 4.5: Scalability results for the optimistic problem formulation with a step cutoff of 75

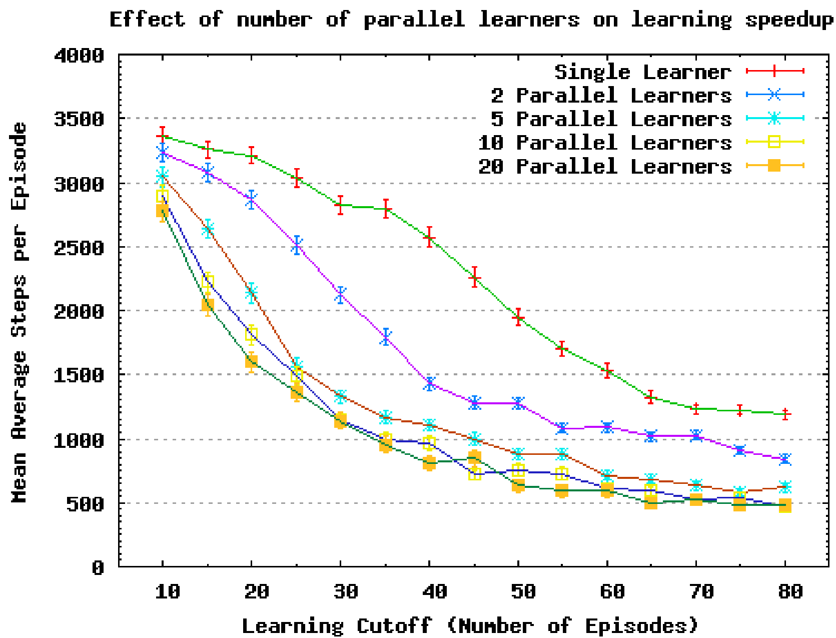


Figure 4.6: Scalability results for the optimistic problem formulation with a step cutoff of 100

most cases a final policy was learnt that resulted in the average number of steps per episode being less than 100. Also note the confidence intervals of these results, indicating that the algorithm is likely to reach these levels of performance on almost all runs.

When the parallel algorithm is compared to the single agent with equal computation time in steps, a very surprising result is observed, the parallel algorithm systematically outperforms the single agent even though both are receiving the same computational power. This is true for all the values of step cutoff that were tested however, as the step cutoff increases the difference in performance between the two algorithms decreases.

Only when the single agent with equal computation time in episodes is used as a comparison, is relatively equal performance observed. The parallel algorithm outperforms the single agent by a small margin for most learning episode cutoffs but in the 100 - 150 episode cutoff range, the single agent manages to perform slightly better than the parallel agents. Interestingly this result is observed for all values of step cutoff, indicating that the single agent generally has slightly faster convergence during this period. However, after this period the parallel agents regain their lead. Based on the graphs, the step cutoff only seems to affect the convergence over the initial 100 episodes, and after this the graphs are very similar for the various step cutoffs.

The results are especially interesting since they show it may be possible to apply the parallel algorithm in the sequential case to improve the speed up of single agent reinforcement learning. This issue will be discussed further in the chapter's conclusion.

### 4.3.2 Scalability Results

Scalability results of the parallel algorithm using the pessimistic formulation with step cutoffs of 50, 75 and 100 are given in figures 4.10, 4.11 and 4.12 respectively. Similar to the scalability results presented for the optimistic problem formulation, convergence speed is improved when going from 2 to 5 and 5 to 10 agents however, there is little improvement going from 10 to 20 agents. In fact when considering the results for 10 and 20 agents the convergence actually seems to be divergent at higher values of the learning episode cutoff (70 and 80). This is clearly affected by step cutoff since in the results for a step cutoff of 50 this divergent behavior is only observed for the 20 agent algorithm. When the step cutoff is increased to 75, the divergence is observed for both 10 and 20 agent algorithms. Furthermore, this divergence is not observed in the results of the optimistic formulation. Therefore, the divergence seems to be an artifact of the problem formulation, number of agents and the step cutoff. The number of agents and step cutoff can be combined into one factor, the total amount of experience the system receives. In this case it appears that more experience results in a higher chance of divergence.

To investigate how the divergence arises, after each learning episode is complete the resulting function approximators are evaluated. The results of this for a sample run are shown in figure 4.13 for a 20 agent simulation with a step cutoff of 100. One plot shows the averaged results of the evaluation for each of the parallel agent's function approximators, while the other plot shows the results of the merged function approximator's evaluation. After about 120 learning episodes a poor function approximator is produced. In the following

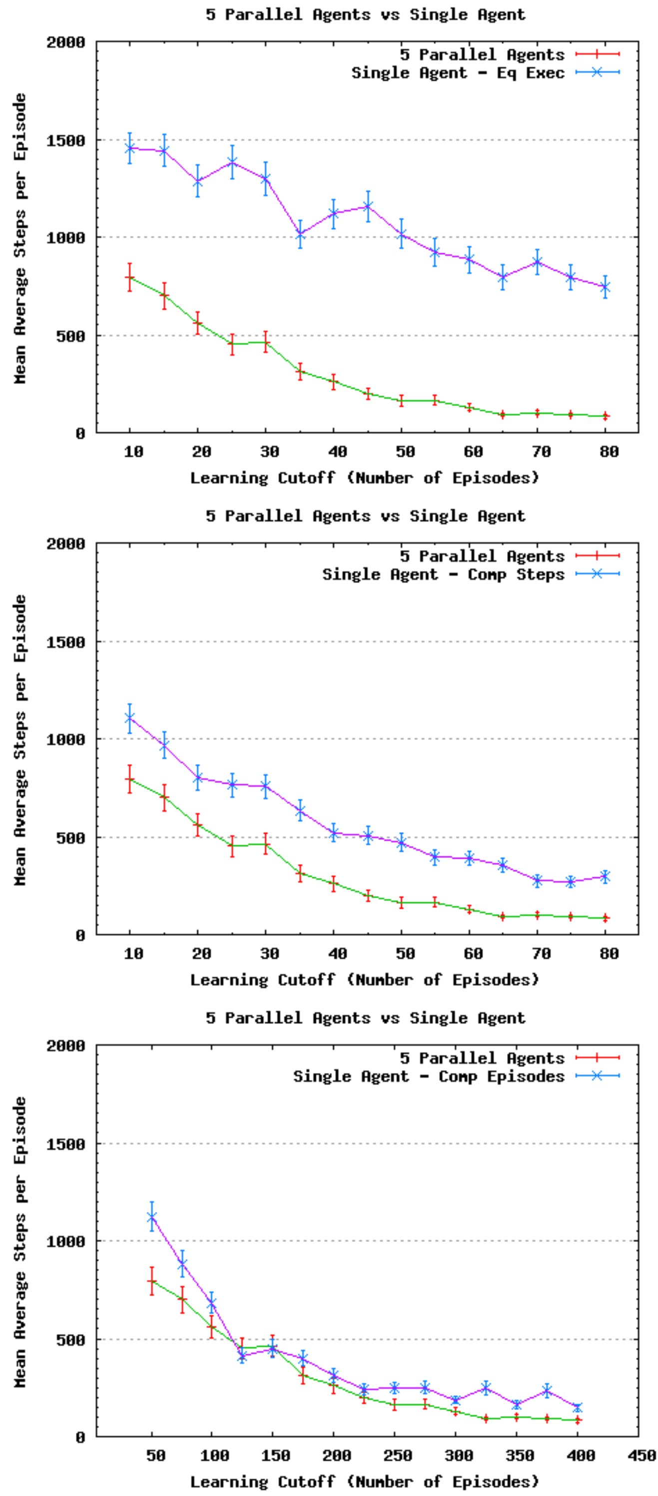


Figure 4.7: A comparison between a single agent and 5 parallel agents for a step cutoff of 50 using the pessimistic formulation

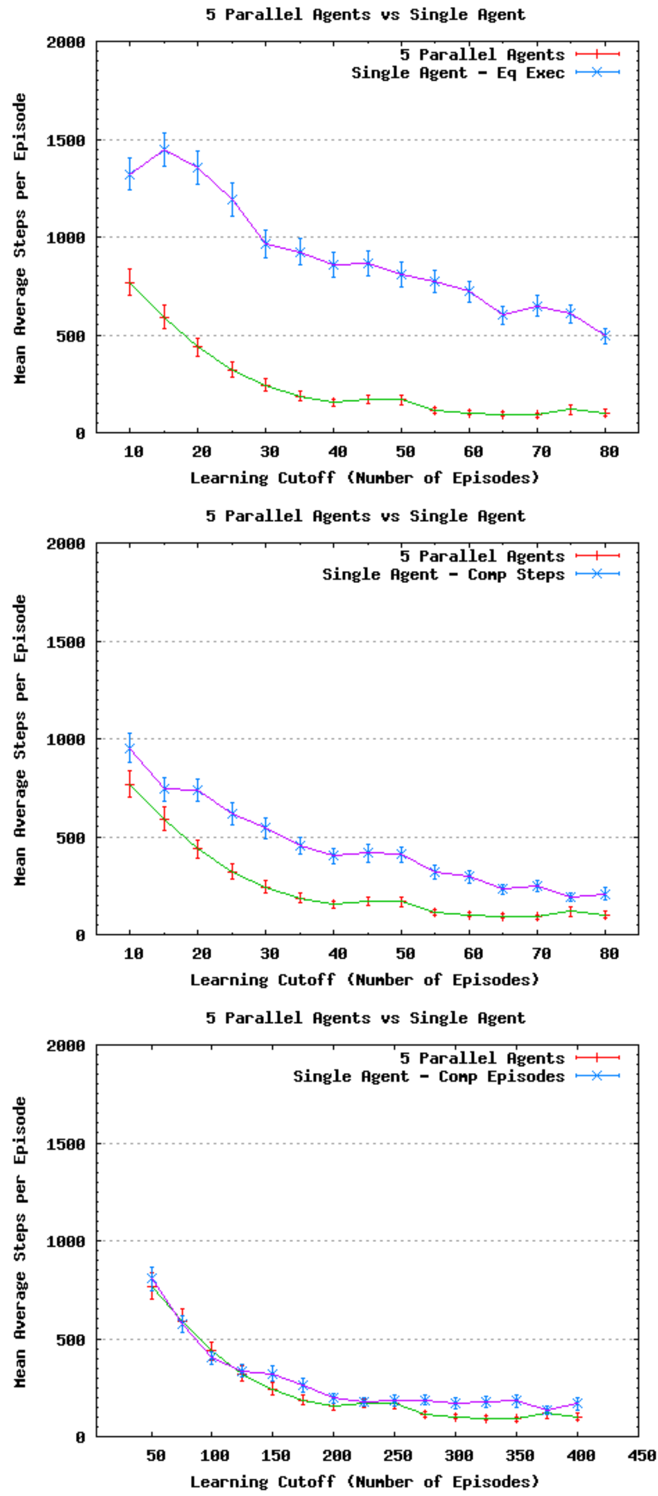


Figure 4.8: A comparison between a single agent and 5 parallel agents for a step cutoff of 75 using the pessimistic formulation

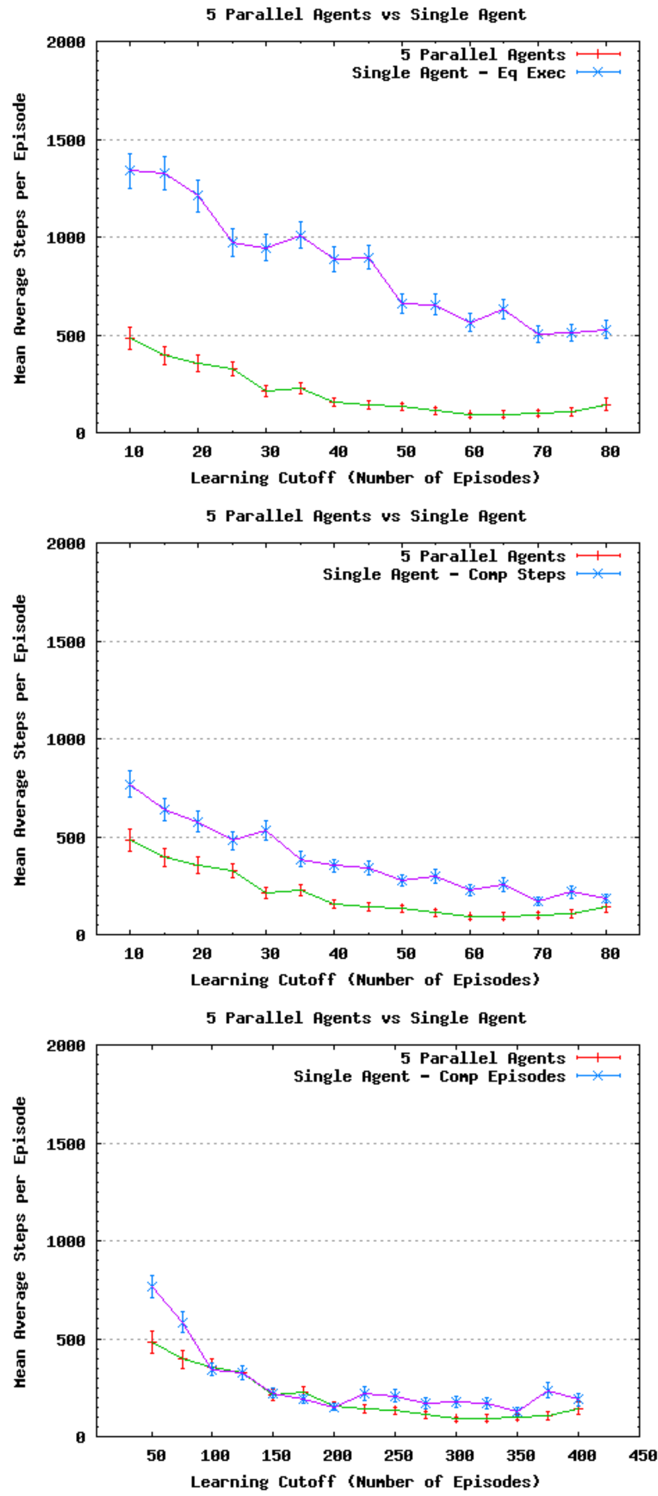


Figure 4.9: A comparison between a single agent and 5 parallel agents for a step cutoff of 100 using the pessimistic formulation

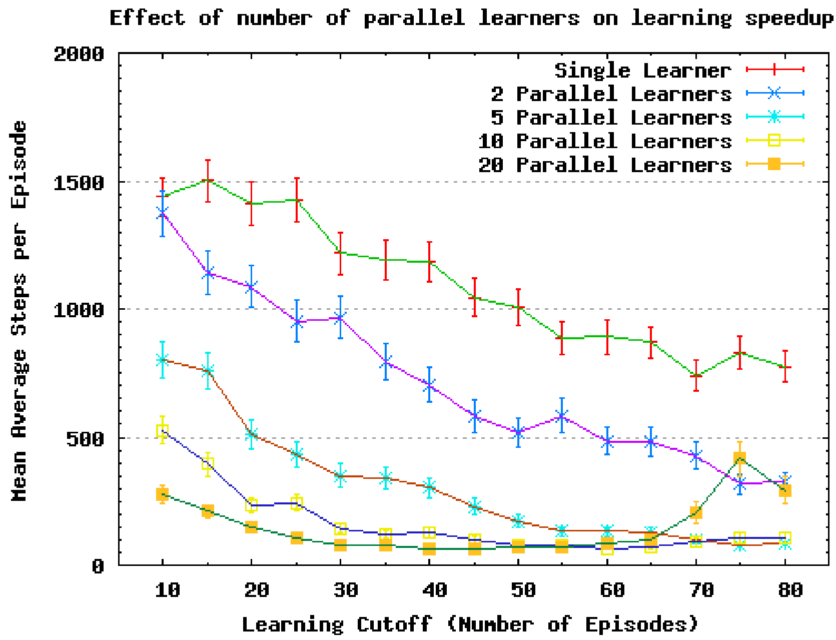


Figure 4.10: Scalability results for the pessimistic problem formulation with a step cutoff of 50

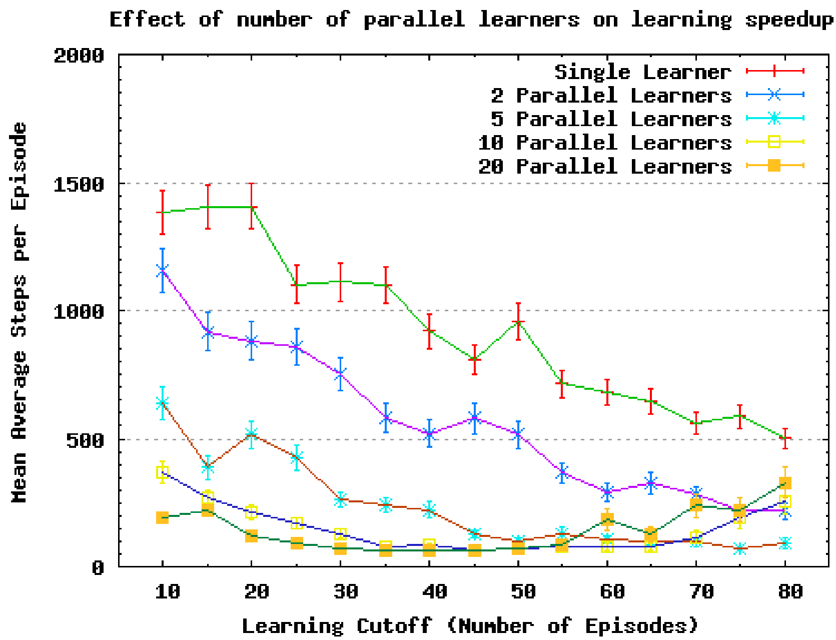


Figure 4.11: Scalability results for the pessimistic problem formulation with a step cutoff of 75

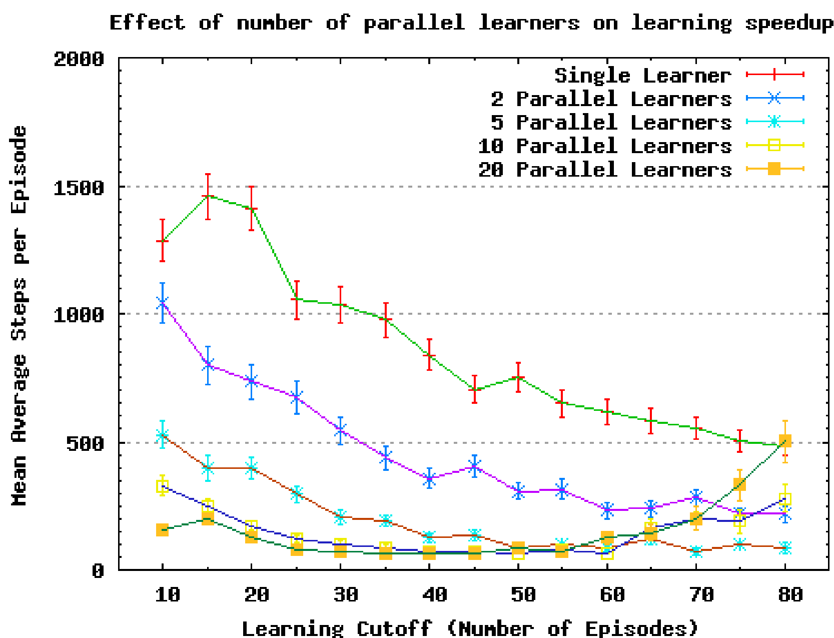


Figure 4.12: Scalability results for the pessimistic problem formulation with a step cutoff of 100

few episodes more poor function approximators are produced until the merging procedure begins to continually produce very poor function approximators. When the merged function approximator is used for learning with the agents, some manage to correct it and this is why the “Mean of Parallel Agent’s FAs” trace does not rise as sharply as the “Merged FA” trace.

It is clear that whatever was causing a function approximator to perform poorly was transferred during the merge stage to the new function approximator. Since it is only feature values that are transferred to a new function approximator, this must be the cause of the divergence.

It turns out that it is possible for a feature value to be increased past the optimal value, therefore invalidating the central idea behind the parallel algorithm: that the maximum value of a feature is the closest to the optimal (minimum value in the case of the optimistic formulation). This incorrect feature value will then persist since the maximum value is always chosen.

A feature can be increased above its maximum value because MDP’s can be stochastic and this is true in the mountain car problem because a function approximator is used for the value estimates. This means that a specific sequence of unlikely, but possible rewards can push a feature’s value above the optimal. This reward sequence only needs to be experience by one agent for it to included in the function approximator. However, another reward sequence would be able to correct this value. Unfortunately, since the value will be propagated to all agents as long as it exists in one agent’s function approximator, the reward sequence that corrects the value must be experienced by all the agents in the same episode. This correlates with the idea raised earlier in this section that

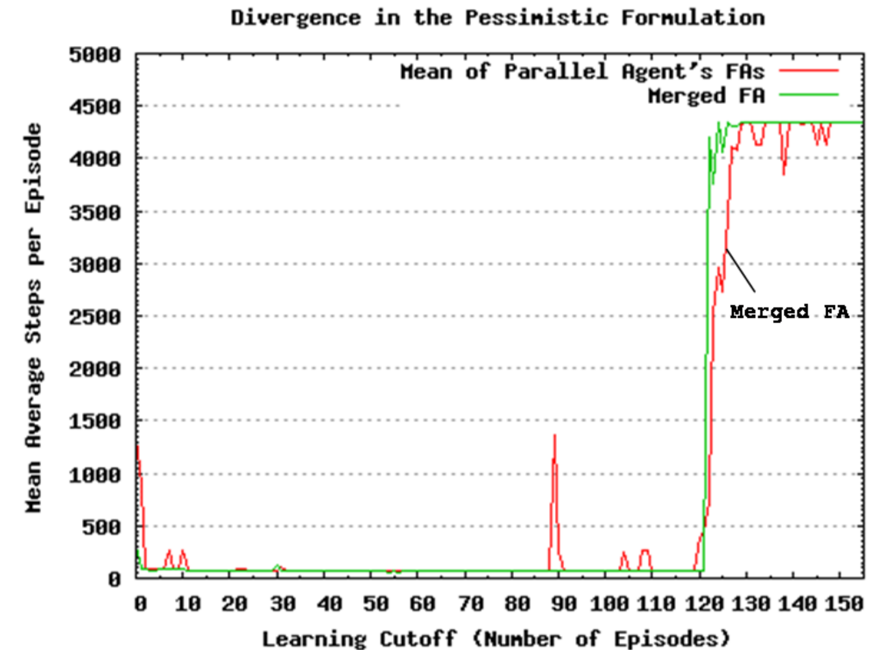


Figure 4.13: The divergent behavior observed on a particular run using the pessimistic problem formulation

more experience might produce divergence.

## 4.4 Conclusion

Both the pessimistic and optimistic parallel learning algorithms have been evaluated on a variety of tests. Significant performance gains have been shown in some situations, namely ones involving the pessimistic problem formulation, while less impressive gains have been observed for the optimistic problem formulation.

Nevertheless, the optimistic algorithm still shows an increase in convergence speed over the single learner allowed equal execution time and therefore demonstrates the applicability of parallel RL. Furthermore, the optimistic formulation generally converges to the optimum in a very stable manner over a long period of time, whereas the pessimistic formulation converges very much faster but more is more erratic given the observed divergent behavior.

Despite the divergence, the pessimistic algorithm manages to achieve better performance than traditional RL algorithms when implemented sequentially. These are very interesting and promising results for RL in general since any method that improves convergence speed increases reinforcement learning usefulness. The divergence is a problem however, since a RL algorithm must have a guarantee of optimality to be useable and this guarantee entails convergence. For this reason, two solutions are proposed that could be investigated in further

work to combat the divergent nature of the pessimistic algorithm.

One algorithm functions as follows, the value estimates from recent episodes are saved such that they may be restored at a later point. The average number of steps per episode is monitored during learning and when signs of divergent behavior emerge, the system restores the value estimates from a previous state before divergence occurred. At this point the system reverts into a single learner where no merging is performed on the value estimates. This method should be sound for two reasons: firstly, when the estimates diverge it happens very suddenly so only recent estimates would need to be retained. Secondly, the convergence speed-up to be gained by parallel learning mainly occurs in the initial episodes meaning that the benefits of parallel learning are still retained. This method is somewhat similar to applying hill climbing to genetic algorithms, where a genetic algorithm is used to find a solution close to the optimal and then hill climbing is applied to the solution to find the optimum. For this algorithm the method corresponds to an approximate solution being produced by the parallel algorithm and then fine tuning the policy to the optimal values using a single agent algorithm.

Another way to solve the divergence problem could be to apply the method described by Kretchmar (2002) to the parallel algorithm. This would involve each agent keeping a record of how frequently they visit a feature, and the merging procedure would be a combination of each agent's value estimate weighted by the number of times they had visited that feature. This would prevent a feature value that had been increased/decreased past the optimum solely determining the new feature value. Although the over-estimated value would contribute significantly, since it is only past the optimum value because the agent has visited that feature many times, this would be averaged out by the other agent's estimates thus preventing it passing the optimal value.

# Chapter 5

# Implementation

This project is concerned with an investigation of parallel reinforcement learning, not a parallel implementation. Therefore, a sequential solution approximating a true parallel system will suffice for the implementation. Furthermore, if the sequential parallel reinforcement learning algorithm is designed carefully then it can also be used to perform experiments with a single learner. This suggests that the system should be designed in terms of a single agent solving the reinforcement learning problem and then scaled up to cope with the parallel algorithms. For this reason, a basic RL framework will be considered before that of the parallel version. However, since decisions taken at this relatively low level will affect the elegance and quality of the high-level implementation, i.e. combining multiple agents to create a parallel learning algorithm, the high level architectural concerns of the system as a whole must be considered first. Following that, a description of the RL framework for a single agent will be given. In the next section an extension to this framework is discussed that allows function approximators to be evaluated. This provides the basics for the experiment interface which is described next. Following that, implementation details for the parallel learning algorithms discussed in chapter 3 will be given. Finally, the methods used to test the implementation will be discussed.

C++ was the clear choice of programming language. Since the system structure lends itself naturally to a hierarchy, it makes sense to organize this in a hierarchy of classes (see figure 5.1). Furthermore, a high amount of modularity can be achieved when using object oriented techniques. Lastly, while it supports less object orientated constructs than other object orientated languages, for example Java, very high performance can still be achieved.

## 5.1 High level Architectural Requirements

Even though the parallel aspects of the system will be executed sequentially, there is no reason why the structure of the system cannot reflect that of a parallel implementation. This is desirable because at some point the algorithm described in this report may be implemented on a parallel architecture, and the better the software architecture maps to a parallel architecture, the easier it will be to change the sequential implementation into a parallel implementation.

Another desirable property of the system is that is should be exceptionally

modular and extensible. This was required because after the structure of the system had been defined, algorithms were added to it which relied on aspects of the system that could not be predicted beforehand. For this reason the best solution is to make the system as modular and replaceable as possible.

The system should be easily reconfigurable since many different tests will be run on the system and the ease with which these tests can be run will significantly aid the usability of the software. Furthermore, when results are generated from the experiments, the results should be useable. The concern with this is the level of detail of results produced by the system. For example, if interest lies in a single run of an episode then the information returned from the experiment should be presented at the step level. However, if different learning algorithms are being evaluated over many episodes to try and determine the comparative performance of an algorithm then long-term statistical results are of interest.

Finally, the framework should be transferable to other reinforcement learning domains. This amounts to not making any assumptions about the style and structure of problem to be solved.

## 5.2 Reinforcement Learning Framework for a Single Agent

The implementation for a single learning agent consists of three core classes: the agent, it's environment and a simulation that provides the temporal logic of the system (see "single agent RL framework" in figure 5.1). The simulation class is the glue between the agent and the environment. An agent has a function approximator class and an eligibility trace class associated with it. While the agent operates at the step level (i.e. chooses an action to perform each time step), the simulation operates at the episode level (i.e. performs a learning episode). A top-down overview of how these three core components interact to complete a learning episode will now be given. After the overview, a more detailed presentation of some of the stages involved in the procedure will be given including a more detailed description of the role of the function approximator and eligibility trace classes.

The implementation follows the full Sarsa procedural algorithm given in algorithm 2.3 very closely.

The simulation object provides methods to an outer component for performing learning episodes. First, the simulation object must be initialized with the parameters for the particular simulation, that is: the problem formulation, the eligibility trace mode, a function approximator and various other learning parameters. The simulation object is also responsible for initializing the agent and environment objects. The function approximator is passed at this level to facilitate the initialization of multiple agents to the same function approximator as required by some of the parallel algorithms. During the agent's initialization, it is passed a pointer to the environment that it will be interacting with. Note, the agent and environment have a unidirectional relationship - the agent knows about the environment, but the environment knows nothing of the agent.

Once initialization is complete, a learning episode can be started by calling one of the `dotrial()` methods (in this section a trial is used interchange-

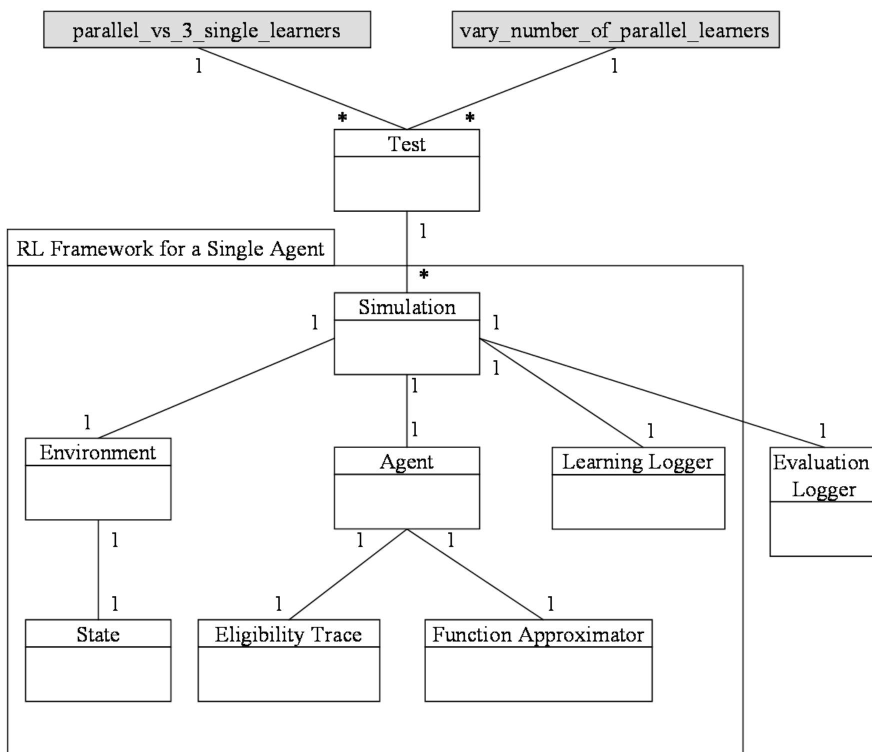


Figure 5.1: The class hierarchy for the RL implementation, the shaded boxes represent the high-level experiment procedures described in section 5.3.4

ably with an episode). Depending on the type of simulation taking place the episode can be set with a step cut-off or with no step cut-off. First, the agent must be informed that a new learning trial is about to start using `initializeLearningTrial()` and the environment reset to a random initial state. The agent prepares for the new trial by retrieving the current state of the environment and based on this information it *computes which features are currently active*. The episode's initial action is then chosen and the eligibility traces are reset.

The next stage of the `doTrial()` function is to call the agent's step function, `learningStep()`. By calling the agent's step function, the agent will perform all the necessary tasks to complete one time step of the system. The first task is to *update the eligibility traces of the currently active features*. Once this is complete, the *action chosen* in the previous step (or initialization step if this is the first step of the episode) is *performed on the environment* and a reward is returned. The agent then retrieves the resulting state from the environment. After this, the $\epsilon$-greedy choice is made of whether to take an exploratory action or the best action. Once this is known, the *feature values in the function approximator are updated* according to the reward just received, the difference in the estimates of the action taken and the action just chosen, and the value of each eligibility trace. Finally, eligibility trace decay is applied and the step function returns.

The `doTrial()` function checks if the agent has successfully reached the goal of the environment by calling the Environment's `goal()` method and if it returns true then the episode terminates. If not, then the `doTrial()` function calls the agent's step function again and the whole process repeats until either the agent reaches the goal or the step cut-off is hit.

The following sections describe the parts of the implementation that were *emphasized*, with particular consideration to the function approximator and eligibility traces.

### 5.2.1 Choosing Which Action to Perform

The agent may choose from two different types of actions to perform, either the action with the highest estimated return or an exploratory action. The choice of action is determined by the exploration strategy the agent is following. The methods used to compute these actions will now be described.

To compute an exploratory action the set of actions available to the agent in the current state must be available. The set of legal actions for the current state is maintained by the environment object and can be retrieved by calling `getLegalActions()`. Any action from the set may then be chosen as the exploratory action. The state features must be recomputed after the new action is chosen.

If the exploration strategy determines that the agent is to take the best action possible then the calculation of the best action proceeds as follows. First, the agent's function approximator's method `getBestActionAndValue()` is called, which returns the best action and its corresponding value estimate for the current state the agent is in. In this case the value of the state-action is discarded.

`getBestActionAndValue()` first retrieves the set of legal actions for the current state. Next it backs up the current active features set (the active features set is the set of features present in the current action-state). Then the function approximator's active features are set to those corresponding to an action in the

legal actions set and `computeValueOfFeatures()` is called which returns the FAs estimate for the set of active features. This is then repeated for each other action in the legal action set, before returning the action with the highest value determined in this manner. Before the method returns the features that were backed up as the first step are restored so the state of the function approximator has not been altered.

Once the agent has set the action returned from `getBestActionAndValue()`, the state features are recomputed again. This may seem like overhead since during `getBestActionAndValue()` the state features have already been recomputed multiple times, however, this is to allow independent use of the function `getBestActionAndValue()`.

### 5.2.2 Performing an Action on the Environment

The agent performs an action on the environment by calling the environment's method `performAction(Action)`. The first function of this method is to update the environment's internal state by calling the state's method `update(Action)`, which computes the agent's new position and velocity according to the rules described in section 2.3. Next, the set of legal actions is updated according to the new state, however, in the mountain car task there are never any illegal actions so this step is trivial. The option of having illegal actions is included so that the framework may be extended to other reinforcement learning domains beside the mountain car. The final stage is to compute the reward to be returned to the agent. According to section 2.3 this depends on the problem formulation being used and it is for this reason that the problem formulation is passed to the environment when it is initialized.

### 5.2.3 Updating the Eligibility Traces

The agent's eligibility traces are updated at two separate points in the step procedure. The first update `setFeatureEligibilities(FunctionApproximator*)` increments the eligibility trace of any currently active features according to the trace mode (replace/accumulate) that the eligibility trace object was initialized with. To determine which features are currently active requires access to the agent's function approximator. An agent's function approximator is decoupled as much as possible from the rest of the system so when a parallel implementation is considered, function approximators can be passed with no regard to what is linked to them. In this case, the decoupling simply amounts to passing a pointer to the agent's current function approximator.

The second update of the eligibility traces is performed by calling `applyDecay()` which decays the eligibility traces according to the parameters $\gamma$ and $\lambda$.

### 5.2.4 Updating the Feature Value Estimates

The feature value estimates contained in the function approximator are updated by calling the FA's method `updateActionStateValues(EligibilityTrace*, alpha, delta)`. As before, since the function approximator is decoupled as much as possible from the rest of the system, the eligibility trace object that is required to perform this update, is passed as a pointer. Also required are the

learning parameters $\alpha$ and the update value $\delta$, as described in the full Sarsa algorithm.

### 5.2.5 Initializing the Function Approximator

Initializing the function approximator consists of three tasks. The first is to calculate the amount of state space that a feature/tile covers. In the function approximator the $x$ direction corresponds to the car's position and $y$ direction corresponds to the car's velocity. The amount of position and velocity represented by a single tile/feature is calculated as follows:

$$tileSizeX = \frac{PositionRange}{NumberOfXTiles - 1}$$
$$tileSizeY = \frac{VelocityRange}{NumberOfYTiles - 1}$$

The second task requires an offset for each tiling in the function approximator to be generated. The y offset is randomly drawn from $[0, tileSizeY]$ and the x offset from $[0, tileSizeX]$.

Finally, each feature value in the function approximator must be set to the initial value. For both problem formulations the feature values are initially set to zero.

### 5.2.6 Computing the state-action Features

The set of active features represents the features present in the function approximator that correspond to the agent's current state and choice of action. Therefore, each time the agent's state changes, or a new action is chosen, the set of active features must be updated. To describe how this is accomplished first the various state spaces in the system must be considered.

The function approximator's state space begins at $(0, 0)$ to simplify the offset calculations. However, the state space for the particular variation of the mountain car problem in question begins at $(-1.2, -0.07)$ and extends to $(0.5, 0.07)$. The two state spaces are shown diagrammatically in figure 5.2. Therefore, computing which features correspond to an agent's state-action first involves a conversion from the true state space to the function approximator's state space.

In this case it involves adding 1.2 to the position, and 0.07 to the velocity of the true state. Next the state space offset must be added. Since the state space is offset by a full tiles's length then `tileSizeX` must be added to the position, and `tileSizeY` must be added to the velocity component of the coordinates. Therefore, the full calculation to convert a position value in true state space to a FA state space value is the following:

$$\begin{aligned} Position_{FAState} &= Position_{TrueState} + \\ &\quad AdjustPositionToPositiveRange + tileSizeX \end{aligned}$$

The result of this calculation gives an $x$ coordinate that lies in the FA state space. However, it is the particular tiles/features that this point represents that are of interest. `getXCell(x_coordinate,tiling)` is a private member of

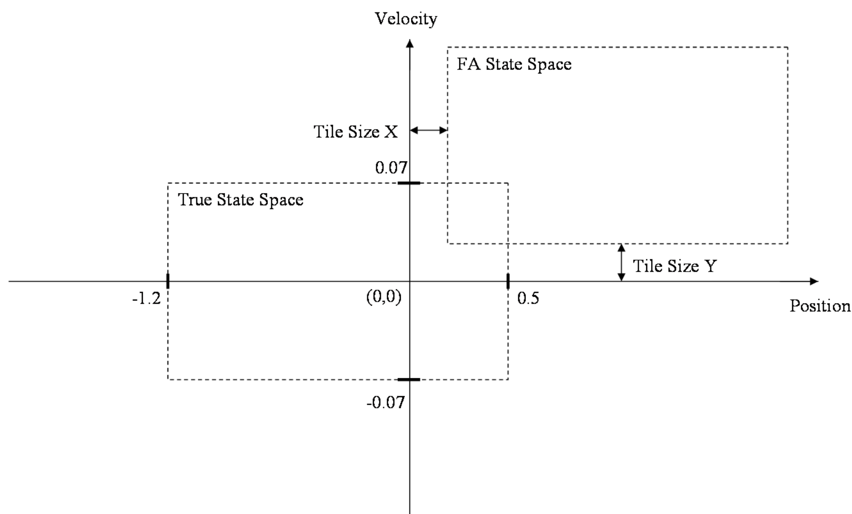


Figure 5.2: The true state space and the FA state space

the FA object and returns the X'th tile from the `tiling` corresponding to the `x_coordinate` passed. The same procedure is then applied to the velocity using `getYCell(float,tiling)` to retrieve the Y'th tile. Now the tile/feature for one of the tilings is known. All that remains is to apply the same procedure to the remaining tilings. Note that since a tiling is always an exhaustive partition of the state space, the number of active features is always equal to the number of tilings contained within the function approximator.

### 5.2.7 Detailed Descriptions of the Data Classes

The descriptions given in the previous sections mainly describe the implementation from an operational point of view. This section clarifies the details of the low level data holding classes.

#### State

A state in the system is represented by a `State` object. This allows a different state description to be chosen if the framework is applied to a different domain with little or no alteration to the objects that use the `State` object. The `update(Action)` method sets the state to the one resulting from performing `Action` in the current state. Three set methods are provided, one to set the value of position and velocity directly, `setState(position, velocity)`, one to set a random state within a range and one to set a truly random state, `setRandom()` . The method `goal()` returns a `bool` indicating whether the current state is a goal. Lastly, the altitude of a state can be retrieved by the `getAltitude()` method.

### Action

An action is represented by the `Action` object, which, in turn is represented by an integer which takes the value $-1$ for reverse, 0 for coast and 1 for accelerate. An index representation is also maintained for when data has to be indexed by action. For example, the function approximator's CMACs are indexed by action. The object provides three methods to set its value: `setRandom(ActionSet)` which sets the action object to an action from `ActionSet`, the constructor `Action(int)` which sets an action to its integer representation and finally, `setActionByIdx(int)` which sets the action by the corresponding index integer. The value of an action can be retrieved using `getValue()` and an action's index by `getIdx()`.

### ActionSet

When a set of actions is required, for example to represent the set of legal actions, an `ActionSet` object is used. The STL `set` class is used to store `Action`s and provides methods for inserting actions and testing membership. The actions in the set can be iterated through using `getNextAction()`. `resetIterator()` is used to reset the sequence of actions returned through `getNextAction()`.

### Function Approximator

Most of the relevant methods provided by the function approximator have already been discussed, or will be discussed in section 5.4 (the implementation of the merge methods for the parallel algorithms). For this reason, implementation of memory parameters will be discussed. The features are represented using a four-dimensional array indexed by tiling, then action, position feature index and finally velocity feature index. The array is created statically to improve performance since the values in a FA will be the most heavily manipulated and accessed in the entire program. To represent the active features, an array is used with storage space for a feature from each tile. A feature is identified using a `cell struct` that stores the action, position tile and velocity tile associated with the feature. The tiling the feature belongs to is implicitly specified by the array index. Tiling offsets are stored in an array as a float for each direction: position offset first, then velocity offset.

```
struct cell {int actIdx; int posBin; int velBin;};
cell features[TILES];   //contains the current active features
float tiles[TILES][ACTIONS][XCELLS][YCELLS];
float offset[TILES][2]; // [T][0] position offset
                        // [T][1] velocity offset
```

### Eligibility Trace

The eligibility trace for each feature is stored in an array with the same dimensions and sizes as the one in the FA that stores feature values.

```
float etrace[TILES][ACTIONS][XCELLS][YCELLS];
```

### 5.2.8 Logging

A logger class was included as a simple method of storing all the details of the learning experienced by an agent. As well as providing an easy method for testing, the logger can produce information about a learning episode at the step level or statistical information from a number of episodes. It is this choice of level of results details that was desired in the general aims of the implementation.

The logger object records traces in terms of episodes. Once a new episode is added using `newEpisode()`, a step trace can be added for every step of the episode using `addStep(stepTrace)`. A `stepTrace` is defined as follows:

```
struct stepTrace {State state; Action action; State newState;
                  int reward; actionType actionChoice;};
```

where `actionChoice` stores whether the action chosen was exploitative or exploratory. For each episode, an initial FA and a final FA can be added using `addInitialFA(FunctionApproximator)` and `addFinalFA(FunctionApproximator)` respectively. The structure that stores an episode is defined as such:

```
struct episodeTrace {vector<stepTrace> trace;
                     FunctionApproximator initialFA;
                     FunctionApproximator finalFA;};
```

Step level information can be printed out to a file for all episodes either with, or without, associated FAs using the method `printLog(ofstream, printFAs, printStats)`. Where `printStats` controls whether episode-level statistics are printed. Episode-level statistics can be retrieved independently using `getAvgStepsPerEpisode()` and `getProbabilityOfEscape()`. Finally, the log can be cleared using `reset()`.

The entire log is stored in a vector:

```
vector<episodeTrace> episodeTraces;
```

Vectors are used in this situation, despite the performance drop associated with them, because the log object should be made as general as possible. For example, the number of steps per episode is not known in advance if episodes are only allowed to finish when a goal is reached. Although this is not the case in the parallel algorithms, some of the algorithms used for comparison have this property. Furthermore, the framework was programmed to be applicable to as many RL problems as possible.

The logger is included in the RL framework as part of the simulation class, named `learningLogger`. To use the logger with the simulation object, the `doTrial()` method had to be altered. The first task it performs now is to create a new episode in the log and add the initial FA. To retrieve a step trace for each step the `learningStep()` method from the `Agent` object was altered to record the relevant information during its execution and then return a `stepTrace` composed of that information. The `doTrial()` method then adds the `stepTrace` returned by each learning step to the logger. When the episode completes the final FA is added to the logger.

## 5.3 Extending the Single Agent RL Framework to Include Support for Multi Agent Experiments

This section considers an extension to the single agent RL framework that provides support for parallel learning agents. As development continued it became apparent that a method of evaluating a function approximator was required. Details of the implementation of this evaluation method are included. Furthermore, a configuration file approach to experimental setup is described along with the high-level procedures required to utilize it.

### 5.3.1 Including an Evaluation Method

The evaluation method is implemented in the existing `Simulation` object. It allows a function approximator belonging to the agent in the simulation to be evaluated by assessing its performance over a number of test episodes. To obtain an accurate evaluation of an FA's performance, it was found that the FA needed to be used on about 100 test episodes. A second logger was introduced in the `Simulation` class, `evaluationLogger`, to record the performance of the agent over the evaluation episodes.

The `evaluate()` method is quite similar in structure to the `doTrial()` method. However, since no learning occurs during the evaluation, the non-learning step functions `initializeTrial()` and `step()` provided by the `Agent` class are used instead of `initializeLearningTrial()` and `learningStep()`. Further differences arise in the episode completion test: again the environment is queried on each step to check if the agent has reached the goal but no step cutoff is imposed since it is the number of steps required to complete the episode that is of interest. However, the evaluation procedure might end up being used on very poor function approximators and since no learning is performed to improve them, protection must be put in place to stop infinitely long episodes occurring. This is accomplished by imposing a step limit of 5000 steps. If the agent takes more steps than the step limit, then it is considered to have failed to reach the goal.

The evaluation function takes the set of test states (initial states of the test episodes) as a parameter so multiple function approximators can be evaluated against the same set of test states. Statistical outputs from the `evaluate()` method include the average number of steps taken per episode and the probability of completing an episode. The evaluation log can be printed by specifying a boolean parameter.

### 5.3.2 The Test Class

To facilitate the setup and execution of experiments involving parallel agents, a `Test` class was introduced. Two methods are provided, one to set the parameters of the experiment and a second to run the experiment, `init()` and `run()` respectively. The parameters of the experiment are specified though a configuration file which is passed to the `init()` method. The parameters are then read in and stored locally in the `Test` object.

The `Test` class maintains a vector of `Simulation` objects, so it can handle algorithms that require any number of agents. This vector of simulations represents the parallel aspect of the implementation, i.e. in a parallel implementation each simulation in this vector would be run on a different processor where function approximators are transferred between them as and when is necessary. Since each simulation contains its own environment, this carries over well into the parallel implementation since each simulation is effectively self contained.

The `run()` method takes the following parameters as inputs: the number of episodes the agents will be allowed to learn for, the step cutoff, a template FA (required for the parallel algorithms (chapter 3) but does no harm to the other algorithms), a set of test states and a directory that specifies where all output files should be placed. The `run()` method commences execution by pushing the specified number of agents (`Simulations`) onto the back of the simulations vector. Each simulation in the vector is then initialized. A while loop is executed next which loops the system for the number of learning episodes. Inside the while loop the first action taken is to perform one learning episode with each agent in the simulations vector. If the step cutoff parameter is set to zero then the learning occurs without a step cutoff. If the algorithm specified in the configuration file is one that does not redistribute/merge function approximators, then the loop repeats until all learning episodes have been completed. If it does however, then the next stage is to evaluate/merge the function approximators just generated in the learning step. Exactly what occurs in this step is algorithm dependent and will be explained in section 5.4, which specifically deals with the implementation of the parallel algorithms.

After learning is completed, the resulting function approximator is evaluated using one of the agents in the simulation vector and the results of this, the average steps per episode and the probability of escape, are returned through reference variables.

During the execution of the `run()` method there is the option to print out the function approximator as it evolves during the learning. This was found to be useful for analyzing the way the value space in a function approximator changes with time.

### 5.3.3 Using a Configuration File to Specify the Parameters of an Experiment

Using a configuration file allows many parameters to be specified in a quick and efficient manner. This helps achieve a goal outlined at the beginning of this chapter: that the framework should be easy to use and therefore it should be possible to setup experiments quickly. The `configFile` object is initialized with the name of the configuration file it is to read the parameters from. Methods are provided for extracting each of the different data types from the file. For example, the output directory name is given as a string and so is extracted using the `getStr()` method. Whereas, the `problem_formulation` parameter is of user defined type and so must be extracted with the `getProblemFormualtion()` method. The parameters specified in the configuration file will now be listed along with a short explanation of their function.

**test_mode** Specifies which type of experiment is being run as described in the evaluation section 4.1.

**test_prefix** Specifies the directory all result files generated during the experiment will be placed.

**number_of_parallel_agents** Specifies the number of parallel agents that will participate in the parallel learning algorithm. If the experiment is for a single learner, then this parameter should be set to 1.

**parallel_merge_method** The parallel algorithm specific merge/selction method to choose/create the function approximator for the next learning episode.

**step_cutoff_init, step_cutoff_max, step_cutoff_inc** The step cutoff value is initially set to step_cutoff_init. To produce each new step cutoff, step_cutoff_inc is added to the current value. When step cutoff becomes greater than step_cutoff_max, then all the values of step cutoff have been investigated.

**episode_cutoff_init, episode_cutoff_max, episode_cutoff_inc** Same as for the step cutoff values except applied to the number of learning episodes.

**number_of_repetitions** Number of tests that will be performed with each combination of step and episode cutoff.

**number_of_test_states** The number of test states that a function approximator is evaluated against. Note, this value does not affect the evaluation accuracy of the parallel learning algorithms that include evaluations as part of their merge/select FA stages.

**problem_formulation** The problem formulation as specified in section 2.3.

**etrace_mode** The eligibility trace mode as discussed in section 2.2.3.

**epsilon, alpha, gamma, lambda** The basic parameters of the reinforcement learning algorithm described in section 2.2.3.

**evaluate_all_fas** The parameter that controls whether the quality of function approximators are checked as they are created. Used to investigate the divergent behavior described in evaluation section 4.3.

**print_learning_log, print_evaluation_log** Controls which logs are printed to the output directory.

**print_fa_graphing_data, print_fa_graphing_data_frequency** Print FA graphing data prints out a function approximator's values to a file so the FA can be visualized using 3D plots. The frequency parameter determines how often the values are printed, e.g. a value of 1 means every episode and 4 means every four episodes.

### 5.3.4 High Level Procedures for Running Complete Experiments

The highest level of the program hierarchy is structured as procedures rather than classes. Only one argument is passed to the main function from the command line, the configuration file's name. A configuration file object is created and the file name is passed to the `configFile`'s initialization method. Next, the `configFile` is queried for the parameter `test_mode` and depending on its

value, one of two procedures is called. The two procedures represent the two different types of tests performed in the evaluation section. If the test_mode is parallel_vs_3_single_learners then the procedure parallel_vs_3_ single_learners() is called and passed the configFile object. If the test_mode is vary_number_of_ parallel_learners then vary_number_of_parallel_learners() is called and again, the configFile is passed. These two procedures are very similar in structure but differ in the types of tests they set up. parallel_vs_3_single_learners() will be described first.

First the output directory is created including the associated results files that will contain the high level statistical results data. The Test objects are setup next, one for the parallel algorithm, the single learner with equal execution time, the single learner with equal computation time in episodes and finally, the single learner with equal computation time in steps. All four tests are then initialized by passing the configFile object to Test's init() method. Two loops are used to control the tests that will be performed for each combination of step cutoff value and learning episode cutoff value. The values for each cutoff parameter are computed from step_cutoff_init, _inc and _max for the step cutoff and episode_cutoff_init, _inc and _max for the episode cutoff. The number_of_repetitions parameter in the configuration file determines the number of times the tests will be repeated for each combination of episode and step cutoff values.

The results of each episode/step cutoff combination are stored in a vector and statistical methods from Statistics.h are used to compute the mean and standard error of both the average steps per episode and probability of escape for each combination. The results are then written to an output file and the process repeated for the next combination of episode/step cutoff values.

vary_number_of_parallel_learners() works in a very similar manner except the Test objects that are created reflect the different number of parallel learners that are undertaking the task. One test object is created for a single learner with equal execution time and the rest for $n$ parallel agents where $n$ is 2, 5, 10 and 20.

## 5.4 Implementation of the Parallel Algorithms

To implement one of the parallel learning algorithms, the framework must be extended in two places. Support must be included in the Test object so the new merge method is recognized when it is retrieved from the configuration file. When the merge stage of the algorithm is reached in the run() method, a simple switch is performed depending on the value of the configuration file's parameter merge_method. Then the code specific to each merge method executes and the resulting function approximator is assigned to newFA. At this point the merge method specific part terminates and newFA is distributed to all the parallel agents. The set of function approximators belonging to the parallel agents is available during the merge stage as a vector agentsFAs should the merge method require the function approximators in this form.

The actual operations that occur in the run() method are generally to do with assessing the quality of a function approximator or creating a quality ordering between the FAs. The actual merging itself is implemented as a method of the FunctionApproximator class. This approach is adopted since one function

approximator is usually being combined with another and therefore it makes sense that the merge is performed by the object that the merge is applied to. When a new function approximator is created from scratch using all the parallel agent's FAs, the new function approximator, `newFA` is used. In this case a vector of function approximators is passed to the new function approximator that should be initialized according to some quality ranking (e.g. minimum feature value).

The implementation specific concerns of each merge method will now be discussed.

### 5.4.1 Select Agent with Maximum Altitude

To determine the agent that reached the maximum altitude during learning, the `Simulation` method `getMaxAltitude()` is called. In turn it calls the Agent's `getMaxAltitude()` method since the maximum altitude reached by an agent is a property of that agent. Once the simulation containing the agent that reached the maximum altitude has been determined, the function approximator retrieved from that simulation using `getFunctionApproximator()` is assigned to `newFA`.

### 5.4.2 Select Best Function Approximator and Merge Algorithms 1,2 and 3

The algorithms select best FA and merge the top $n$ FAs both have a very similar structure in the `run()` method. First a set of test states is generated and then this set is passed to each simulation's `evaluate()` method. The results of all the evaluations are stored in a vector and this vector is then sorted according to performance (FAs with low average steps per episode have high performance). At this point the four algorithms diverge in function.

The Select best function approximator algorithm completes by assigning the top performing FA to `newFA`.

Merge algorithm 1 merges the second, third and fourth best performing function approximators with the first, in that order. To perform the merge, the method `mergeFA` from the best performing `FunctionApproximator` is called. It is passed the function approximator to be merged and the weight of the merge (as described in section 3.5.1). `mergeFA` calls `getTilesOffsetOrder()` to compute the tile offset order for both FAs. The features are then merged according to the tiling ordering. Finally, the best function approximator (that has had the three others merged with it) is assigned to `newFA`.

Merge algorithm 2 is identical to merge algorithm 1 except `mergeFA2` is used to perform the merging. `mergeFA2` first computes the points $p_1, p_2, p_3$ and $p_4$ for a feature (see figure 3.6). These points are relative to the FA state space, not the true state space (as indicated in figure 5.2) so the value of a point must be retrieved using `getValueOfFAStateAction(FAState, Action)` where `FAState` is a `struct` describing a set of coordinates in the FA state space. Using these points, a feature's value is updated according to the equation given in section 3.5.2. The process is then repeated for all other features in the best FA.

Merge algorithm 3 uses the FA method `getValueOfTile(bottomLeftX, bottomLeftY, sampleTileSizeX, sampleTileSizeY, Action)` to compute the

value of a feature that is not natively represented by a function approximator. `getValueOfTile()` makes repeated calls to `getValueOfFAStateAction()` based on the passed bottom left corner coordinates and the size to compute the points $p_1, p_2, p_3, p_4$ and $p_5$ shown in figure 3.8. As with the other two merge algorithms this process is repeated for each feature of the best FA.

### 5.4.3 Select Minimum Feature Value

Since the quality of a FA in this algorithm is dependent on independent feature values, no evaluation step needs to be performed. In fact, the vector of agent's FAs is simply passed straight to the `initToMin()` method of `newFA`. For each feature in `newFA`, `initToMin()` calls `getValueOfTile()` to extract the value of that feature from each agent's FA in the vector and assigns the minimum value extracted in this way to the feature. After `initToMin()` returns, the merged function approximator is `newFA` *itself* so the task is finished.

### 5.4.4 Select Minimum Native Feature Value

The vector containing the agent's FAs is passed to the `mergeToMin()` method of `newFA`. `mergeToMin()` iterates through all the features of the function approximator it is part of and assigns each feature a value computed as follows. For a feature $f$, the value of $f$ is extracted from each function approximator in `agentsFAs` and the minimum value is chosen. To extract a feature's value the FA method `getValueOfCell(tiling, action, position_tile, velocity_tile)` is used. Because all function approximators used in this algorithm originate from a template, each represents the same set of features. Therefore, `getValueOfCell()` extracts a feature's value from a function approximator based solely on the storage index of that feature. `mergeToMax()` is the equivalent merge function for the pessimistic algorithm.

## 5.5 Testing

Since the system was developed in three stages, the testing that was applied to each stage will be described in that order.

The basic RL framework was tested mainly using the `Logger` class. Several very short example runs were calculated by hand and then manually setup using the framework. The logs could then be inspected for any deviation from the expected results. Since the `Logger` class includes the option of inspecting the function approximators as they are learnt by the agent, the feature value update code was checked using this method. Obviously using the logger would not be a good method of testing the system if there was an error in the logger itself. For this reason, the code of the logger class was checked very carefully by hand.

Clearly, testing by this method can not be considered an exhaustive test of the system since a few runs will not test every aspect of the system. However, with knowledge of the internal workings of the classes, the tests can be structured so as many methods/ object interactions are included in the tests as possible. Moreover, since the system will only be used to simulate one domain the likelihood of a situation arising that was not encountered during the tests

is low. To include as many possible situations as possible the tests were setup to explore the following areas:

- The agent's movement though the domain, i.e. the system is in the correct state after executing any of the three actions. The agent's movement was also tested in the extremities of the domain, for example, when the car reaches the far side of the valley from the goal the velocity of the car is set in a different way.
- The episode ends correctly, both through the agent reaching the goal and by the step cutoff being reached.
- The conversion between the true state space and the state space represented by the function approximator is performed correctly. This amounts to ensuring that the mapping from a state to a set of features is correct.
- Determining whether the best action is successfully retrieved from a function approximator.
- Checking that the feature value estimates are correctly updated after an action is performed.
- Checking that the eligibility traces are incremented and decayed correctly.

The extension to the RL framework was tested as follows. First, the evaluation extension was tested using an approach similar to the one described above except the `evaluationLogger` was used instead of the `learningLogger`. The `Test` class was tested using simple `cout` output statements to ensure the data passed during the execution of the `run()` method was correctly received and then correctly passed to the simulation methods. The `init()` member of the `Test` class was trivial to test. The top level experiment procedures (and associated `configFile` class) were again tested in a similar manner since they do nothing more than route data to the lower level parts of the hierarchy.

Since each parallel algorithm required code to be implemented in two places, testing of this code was performed by two methods, one for each of the places. The first test ensured that the ordering between function approximators, computed in the `run()` method of the `Test` class, was correctly established (if the algorithm required such a step). The merge methods themselves were tested by keeping a mini "merge log" of the various values that were extracted from the other FAs. The log also recorded the results of combining the values to produce the new merged feature values. This merge log was used in combination with the "final FAs" stored in the learning log to check the correct values were extracted. The initial FAs stored in the learning log from the episode following the merge could be inspected too to ensure they contained the values from the merge procedure.

# Chapter 6

# Conclusion

This project has demonstrated several methods of parallelizing the reinforcement learning process to speed up convergence. While some of these methods performed very poorly, they acted as a guide for reaching an algorithm that performed well. The best performing algorithm showed a significant speed up over a single learner with equal execution time in both the pessimistic and optimistic problem formulations. In the pessimistic case, the parallel algorithm showed a speed up over a single learner which is allowed equal computation time, a most surprising result that will be discussed in the further work section.

The best performing algorithm operates as follows: after a learning episode has been completed for each parallel agent, the resulting function approximators are merged therefore combining the best knowledge each of the parallel agents has to offer. This new function approximator is then distributed to the parallel agents and the process is repeated. The merging algorithm assigns each feature in the new function approximator to the minimum/maximum value of that feature as drawn from the group of function approximators belonging to the parallel agents. Since all the function approximators represent the same set of features, a feature value for the new function approximator can be looked up instead of approximated resulting in a more accurate merging.

This algorithm functions on the principle that the minimum value of a feature is the closest to the optimal when the feature values are initialized pessimistically, and the maximum value is closest to the optimal when the feature values are initialized optimistically. The rewards must also be structured to reflect the initialization i.e. positively reinforce the agent in the pessimistic case and negatively reinforce the agent in the optimistic case. Therefore, for this algorithm to function correctly, the reward structure and initial feature values must be set to complement each other.

During the evaluation of this algorithm, it became apparent that in the pessimistic case, the algorithm was showing divergent behavior. This was traced back to a false initial assumption about the nature of increasing a feature's value. Previously it was thought that the the highest feature value from the set of parallel agents represented the value closest to the optimum. It turns out that because MDPs are stochastic this is not the case. A feature value can be increased past the optimal value and, due to the nature of the algorithms, this value will very likely persist throughout the algorithms execution. Due to this problem, two enhancements were suggested for the algorithm to prevent

divergence. These will be discussed again in the future work section.

Note, that this problem also applies to the optimistic algorithm however, it was not seen in practice during the evaluation. This appears to be due to the stability of the optimistic algorithm, given that it approaches the optimal values more cautiously than the pessimistic algorithm.

When the step cutoff parameter was investigated it was found that with a low number of parallel learners an increase made a large difference to convergence speed. However, when many parallel learners are used an increase in step cutoff makes little difference to convergence speed.

The framework required to test the various algorithms was implemented sequentially since only experimental results were required for the parallel algorithms, not a full parallel implementation. There were three major requirements specified for the implementation. Firstly, that the framework should be applicable to any RL domain. Secondly, the implementation should be designed in such a way so that it could be easily transferred to a parallel architecture if this is required at some point in the future. This was achieved by encapsulating each agent in its own simulation object that contained everything the agent needed to work independently. Furthermore, function approximators were completely decoupled from the system so they could be passed between nodes in a parallel architecture with no alteration. The third major requirement of the system was that it should be easy to setup and collect results from experiments with the various algorithms. This was accomplished by introducing a configuration file to facilitate experiment setup and give the user the ability to control the level of detail output in the experiment's results.

## 6.1 Further Work

The results that show the pessimistic parallel algorithm outperforming the single agent algorithm are very interesting. This is because the same techniques used in the parallel algorithm could be applied in the sequential case to improve the performance of current RL algorithms. For example, it could be more effective to restart a single agent in a random initial state after a step cutoff has been reached instead of allowing it to complete the episode.

Two enhancements were suggested that could prevent the divergent nature of the parallel algorithm as it stands. One method works by detecting divergence and reverting the system to a previous state before divergence occurred. After the system is reverted an algorithm that does not diverge can be applied to complete the remaining learning. The other enhancement would be to use a method similar to that described by Kretchmar (2002) to store an additional value for each feature indicating how frequently the agent visits it. This value is then used to weight the agent's value estimate when it is combined with the other agent's estimates.

Finally, to truly demonstrate the applicability of the parallel algorithm to reinforcement learning, it should be tested on more domains. To apply the algorithm in a domain where variance is experienced in the rewards the agents receive, the enhancement based on Kretchmar's method would need to be applied. This is because when a high variance in the rewards is experienced, a feature's value would be reduced/increased past the optimal value far more often than it currently is in the mountain car domain. This is because the CMAC

tiling that is responsible for the stochasticity only produces a very small variance in the value estimates.

# Bibliography


J. S. Albus. *Brain, Behavior and Robotics*. Byte Books, Peterborough, NH, 1981.

J. A. Bagnell. A robust architecture for multiple-agent reinforcement learning. Master's thesis, 1998.

R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

R. M. Kretchmar. Parallel reinforcment learning. In *Proceedings of the 6th World Conference on Systemics, Cybernetics, and Informatics (SCI2002)*, 2002.

A. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the Eighth International Conference on Machine Learning*. Morgan Kaufmann, 1991.

N. NotFoundYet. 1996.

G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR166, Cambridge University Engineering Dept., 1994.

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1–3):123–158, 1996. URL `citeseer.ist.psu.edu/singh96reinforcement.html`.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

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

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, U.K., 1989.

S. D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 607–613, 1991.

D. Wingate and K. Seppi. P3VI: A partitioned, prioritized, parallel value iterator. In *Proceedings of the 21st International Conference on Machine Learning (ICML-2004)*, 2004.

# Appendix A

# Code for the RL Framework

## A.1 Action

### Action.h

```
--------------------------------------------------------------------------

#ifndef ACTION_H
#define ACTION_H

#include <cstdlib>
#include <cstdio>
#include <iostream>
using namespace std;

#include "ActionSet.h"

class ActionSet;

class Action {

public:

  Action();
  Action(ActionSet legalActions);
  Action(int a);
  void setActionByIdx(int idx);

  int getValue();
  int getIdx();

  void setRandom(ActionSet legalActions);

  bool operator<(Action a) const {
    return (moveDirection<a.getValue());
  }

  friend ostream &operator<<(ostream &stream, Action a);
```

```
private:

  int moveDirection;

  int getRandomDirection();

};

#endif
```

### Action.cpp

```
--------------------------------------------------------------------------

#include "Action.h"


Action::Action() {
}

Action::Action(ActionSet legalActions) {
  cout << "Action constructor called" << endl;
  setRandom(legalActions);
}

Action::Action(int a) {
  moveDirection = a;
}

void Action::setActionByIdx(int idx) {
  moveDirection = idx - 1;
}

int Action::getValue() {
  return moveDirection;
}

int Action::getIdx() {
  return moveDirection + 1;
}

void Action::setRandom(ActionSet legalActions) {
  moveDirection = getRandomDirection();
  while (!legalActions.member(this)) {
    moveDirection = getRandomDirection();
  }
}

int Action::getRandomDirection() {
  int rand_num;
  rand_num = (rand() % 3);
  return rand_num-1;
}

ostream &operator<<(ostream &stream, Action a) {
```

```
  stream << a.moveDirection;
  return stream;
}
```

## A.2 ActionSet

### ActionSet.h

```
--------------------------------------------------------------------------

#ifndef ACTIONSET_H
#define ACTIONSET_H

#include "Action.h"

#include <cstdio>
#include <set>
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
using namespace std;

class Action;

class ActionSet {

public:

  ActionSet();

  void resetIterator();

  Action getNextAction();

  int count();

  bool member(Action a);
  bool member(Action *a);

  void insert(Action a);

  void clear();

  void print();

private:

  set<Action, less<Action> > actions;
  set<Action, less<Action> >::iterator nextAction;

};

#endif
```

**ActionSet.cpp**

```
------------------------------------------------------------------------

#include "ActionSet.h"


ActionSet::ActionSet() {
  resetIterator();
}

void ActionSet::resetIterator() {
  nextAction = actions.begin();
}

Action ActionSet::getNextAction() {
  Action a;
  a = *nextAction;
  nextAction++;
  return a;
}

int ActionSet::count() {
  return actions.size();
}

void ActionSet::insert(Action a) {
  actions.insert(a);
}

void ActionSet::clear() {
  actions.erase(actions.begin(), actions.end());
}

bool ActionSet::member(Action a) {
  if (actions.count(a) > 0) {
    return true;
  }
  return false;
}

bool ActionSet::member(Action *a) {
  if (actions.count(*a) > 0) {
    return true;
  }
  return false;
}

void ActionSet::print() {

  cout << "Actionset contains " << actions.size() << " actions: ";

  set<Action, less<Action> >::iterator backupIterator;
  backupIterator = nextAction;
```

```
  resetIterator();
  for (unsigned i = 0; i<actions.size(); i++) {
    cout << getNextAction() << " ";
  }

  cout << endl;

  nextAction = backupIterator;
}
```

## A.3 Agent

### Agent.h

```
---------------------------------------------------------------------------

#ifndef AGENT_H
#define AGENT_H

#include <cstdlib>
#include <cstdio>
#include <iostream>
using namespace std;

#include "Logger.h"
#include "Constants.h"
#include "Environment.h"
#include "Action.h"
#include "State.h"
#include "FunctionApproximator.h"


class Agent {

public:

  void init(Environment *env,
            ETraceMode traceMode,
            float epsilonVal,
            float alphaVal,
            float gammaVal,
            float lambdaVal);

  void initializeTrial(); //No learning to be done so no initial choice
  stepTrace step();
  void initializeLearningTrial(actionType initialChoice);
  stepTrace learningStep();

  FunctionApproximator getFunctionApproximator();
  void setFunctionApproximator(FunctionApproximator fa);

  float getMaxAltitude();
```

```
  void printLegalMoves();

private:

  float epsilon;
  float alpha;
  float gamma;
  float lambda;

  void setBestAction();
  void setExploratoryAction();

  Environment *e;

  Action action;
  State state;

  FunctionApproximator funApprox;
  EligibilityTrace eTrace;

  float maxAltitude;

};

#endif
```

### Agent.cpp

```
--------------------------------------------------------------------------
#include "Agent.h"


void Agent::init(Environment *env, ETraceMode traceMode, float epsilonVal,
                 float alphaVal, float gammaVal, float lambdaVal) {

  e = env;

  eTrace.init(traceMode);
  funApprox.init();

  epsilon = epsilonVal;
  alpha = alphaVal;
  gamma = gammaVal;
  lambda = lambdaVal;

}


void Agent::printLegalMoves() {
  ActionSet testactions = e->getLegalActions();
```

```
  testactions.print();
}


void Agent::initializeLearningTrial(actionType initialChoice) {
  eTrace.reset();



  state = e->getState();
  maxAltitude = state.getAltitude();

  #ifdef SETUPDEBUG
    cout << "Initial State: " << state << "\n";
  #endif

  if (initialChoice == best) {
    setBestAction();
  }
  else {
    setExploratoryAction();
  }

  #ifdef SETUPDEBUG
    cout << "Initial Action: " << action << "\n";
  #endif
}

stepTrace Agent::learningStep() {
  int reward;
  float delta;
  stepTrace trace;

  #ifdef STEPDEBUG
    printf("Calculating e traces\n");
  #endif
  //cout << "ActiveCells:";
  eTrace.setFeatureEligibilities(&funApprox);
  //cout << "\n";

  #ifdef STEPDEBUG
    cout << "Performing action " << action << " on state " << state << "\n";
  #endif

  trace.state = state;
  trace.action = action;

  reward = e->performAction(action);
  state = e->getState();

  trace.reward = reward;
  trace.newState = state;

  if (state.getAltitude() > maxAltitude) {
```

```
    maxAltitude = state.getAltitude();
  }

  delta = reward - funApprox.computeValueOfFeatures();

  if (generateRandomNumber(0, 1) > epsilon) {
    #ifdef STEPDEBUG
      cout << "Best action route taken\n";
    #endif

    trace.actionChoice = best;
    setBestAction();
  }
  else {
    #ifdef STEPDEBUG
      cout << "Random action route taken\n";
    #endif

    trace.actionChoice = exploratory;
    setExploratoryAction();
  }

  delta = delta + gamma*funApprox.computeValueOfFeatures();

  funApprox.updateActionStateValues(&eTrace, alpha*delta);

  eTrace.applyDecay(gamma*lambda);

  return trace;
}

void Agent::initializeTrial() {
  state = e->getState();
  maxAltitude = state.getAltitude();
  #ifdef SETUPDEBUG
    cout << "Initial State: " << state << "\n";
  #endif

  setBestAction();

  #ifdef SETUPDEBUG
    cout << "Initial Action: " << action << "\n";
  #endif
}


stepTrace Agent::step() {
  int reward;
  stepTrace trace;

  trace.state = state;
  trace.action = action;

  reward = e->performAction(action);
```

```
  state = e->getState();

  trace.reward = reward;
  trace.newState = state;

  if (state.getAltitude() > maxAltitude) {
    maxAltitude = state.getAltitude();
  }

  trace.actionChoice = best;
  setBestAction();

  return trace;
}


void Agent::setExploratoryAction() {
  action.setRandom(e->getLegalActions());
  funApprox.setFeatures(state, action);
}


void Agent::setBestAction() {
  float value;
  // action is class variable
  funApprox.getBestActionAndValue(e, action, value); //discard the value
  funApprox.setFeatures(state, action);
  //cout << "Best Action: " << action << endl;
}


void Agent::setFunctionApproximator(FunctionApproximator fa) {
  funApprox = fa;
}


FunctionApproximator Agent::getFunctionApproximator() {
  return funApprox;
}


float Agent::getMaxAltitude() {
  return maxAltitude;
}
```

## A.4 ConfigFile

### ConfigFile.h

```
--------------------------------------------------------------------------
```

```
#ifndef CONFIGFILE_H
#define CONFIGFILE_H

#include <cstdio>
#include <cmath>
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;

#include "Types.h"

class ConfigFile {

public:

  ConfigFile(string configFileName);

  int getInt(string key);
  float getFlt(string key);
  string getStr(string key);
  bool getBool(string key);
  FAMergeMethod getFAMergeMethod(string key);
  ProblemFormulation getProblemFormulation(string key);
  ETraceMode getETraceMode(string key);

  string getConfigFileName();

private:

  string configFileName;

};

#endif
```

### ConfigFile.cpp

```
---------------------------------------------------------------------------

#include "ConfigFile.h"

ConfigFile::ConfigFile(string fileName) {
  configFileName = fileName;
}

int ConfigFile::getInt(string key) {
  return atoi(getStr(key).c_str());
}

float ConfigFile::getFlt(string key) {
  return atof(getStr(key).c_str());
}
```

```
string ConfigFile::getStr(string key) {
  ifstream configFile(configFileName.c_str());
  string buffer;

  while (!configFile.eof()) {
    configFile >> buffer;
    if (buffer == key) {
      configFile >> buffer;
      //cout << "Key: " << key << " Value: " << buffer << endl;
      return buffer;
    }
    else {
      configFile >> buffer;
    }
  }

  cout << "Key " << key << " not found" << endl;
  exit(0);
  return "empty";
}

bool ConfigFile::getBool(string key) {
  string valueStr = getStr(key);
  if (valueStr == "true") {
    return true;
  }
  else if (valueStr == "false") {
    return false;
  }
  else {
    cout << "Invalid choice for " << key << " in param file: " << getConfigFileName() << endl;
  }
  exit(0);
  return false;
}

FAMergeMethod ConfigFile::getFAMergeMethod(string key) {
  string valueStr = getStr(key);
  if (valueStr == "minmerge") {
    return minmerge;
  }
  else if (valueStr == "maxmerge") {
    return maxmerge;
  }
  else if (valueStr == "maxaltitude") {
    return maxaltitude;
  }
  else if (valueStr == "inittomin") {
    return inittomin;
  }
  else if (valueStr == "evaluate_select_best") {
    return evaluate_select_best;
  }
  else if (valueStr == "evaluate_merge1") {
```

```
    return evaluate_merge1;
  }
  else if (valueStr == "evaluate_merge2") {
    return evaluate_merge2;
  }
  else if (valueStr == "evaluate_merge3") {
    return evaluate_merge3;
  }
  else if (valueStr == "none") {
    return none;
  }
  else {
    cout << "Invalid choice for " << key << " in param file: "
         << getConfigFileName() << endl;
  }
  exit(0);
  return none;
}

ProblemFormulation ConfigFile::getProblemFormulation(string key) {
  string valueStr = getStr(key);
  if (valueStr == "optimistic") {
    return optimistic;
  }
  else if (valueStr == "pessimistic") {
   return pessimistic;
  }
  else {
    cout << "Invalid choice for problem formulation in param file: "
         << getConfigFileName() << endl;
  }
  exit(0);
  return optimistic;
}

ETraceMode ConfigFile::getETraceMode(string key) {
  string valueStr = getStr(key);
  if (valueStr == "replace") {
    return replacetrace;
  }
  else if (valueStr == "accumulate") {
    return accumulatetrace;
  }
  else {
    cout << "Invalid choice for eligibility trace mode in param file: "
         << getConfigFileName() << endl;
  }
  exit(0);
  return replacetrace;
}

string ConfigFile::getConfigFileName() {
  return configFileName;
}
```

## A.5 EligibilityTrace

### EligibilityTrace.h

```
---------------------------------------------------------------------------

#ifndef ETRACE_H
#define ETRACE_H

#include <cstdio>
#include <cmath>
#include <iostream>
using namespace std;

#include "Types.h"
#include "FunctionApproximator.h"
#include "Constants.h"

class FunctionApproximator;

class EligibilityTrace {

public:

  void init(ETraceMode mode);
  void reset();

  void applyDecay(float gammaxlambda);
  void setFeatureEligibilities(FunctionApproximator *funApprox);

  float val(int tile, int action, int pos, int vel);

private:

  float etrace[TILES][ACTIONS][XCELLS][YCELLS];

  ETraceMode traceMode;

};

#endif
```

### EligibilityTrace.cpp

```
---------------------------------------------------------------------------

#include "EligibilityTrace.h"

void EligibilityTrace::init(ETraceMode mode) {
  traceMode = mode;
  reset();
}


void EligibilityTrace::reset() {
  for (int t = 0; t<TILES; t++) {
```

```
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          etrace[t][a][p][v] = 0.0;
        }
      }
    }
  }
}

void EligibilityTrace::applyDecay(float gammaxlambda) {

  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          etrace[t][a][p][v] = gammaxlambda*etrace[t][a][p][v];
        }
      }
    }
  }
}

void EligibilityTrace::setFeatureEligibilities(FunctionApproximator *funApprox) {
  cell feature;

  for (int t = 0; t<TILES; t++) {
    feature = funApprox->getFeature(t);
    if (feature.actIdx != -1) {
      if (traceMode == accumulatetrace) {
        etrace[t][feature.actIdx][feature.posBin][feature.velBin] += 1;
      }
      else if (traceMode == replacetrace) {
        //Reset feature trace to 1 and all other traces corresponding to that
        //action to 0.
        etrace[t][feature.actIdx][feature.posBin][feature.velBin] = 1;
        for (int i = 0; i<ACTIONS; i++) {
          if (feature.actIdx != i) {
            etrace[t][i][feature.posBin][feature.velBin] = 0;
          }
        }
      }
      else {
        cout << "Unknow eligibility trace mode: " << traceMode << endl;
      }
    }
  }
}

float EligibilityTrace::val(int tile, int action, int pos, int vel) {
  return etrace[tile][action][pos][vel];
}
```

## A.6 Environment

### Environment.h

```
--------------------------------------------------------------------------

#ifndef ENVIRONMENT_H
#define ENVIRONMENT_H

#include "Types.h"
#include "State.h"
#include "Action.h"
#include "ActionSet.h"



class Environment {

public:

  void init(ProblemFormulation pf);
  void resetToRandomState();
  void reset(State initialState);

  int performAction(Action a);
  State getState();

  ActionSet getLegalActions();

  bool goal();

  float getP();
  float getV();

private:

  State state;
  ProblemFormulation problemFormulation;

  int reward(Action a);

  ActionSet legalActions;

  void updateLegalActions();

};

#endif
```

### Environment.cpp

```
--------------------------------------------------------------------------

#include "Environment.h"
```

```
void Environment::init(ProblemFormulation pf) {
  problemFormulation = pf;
  updateLegalActions();
}

int Environment::performAction(Action a) {
  state.update(a);
  updateLegalActions();
  return reward(a);
}

State Environment::getState() {
  return state;
}

void Environment::updateLegalActions() {
  Action reverse(-1);
  Action idle(0);
  Action accelerate(1);

  legalActions.clear();

  legalActions.insert(idle);
  legalActions.insert(accelerate);
  legalActions.insert(reverse);
}

void Environment::resetToRandomState() {
  state.setRandom();
  updateLegalActions();
}

void Environment::reset(State initialState) {
  state = initialState;
  updateLegalActions();
}

int Environment::reward(Action a) {
  if (problemFormulation == pessimistic) {
    if (state.goal()) {
      return 1;
    }
    else {
      return 0;
    }
  }
  else if (problemFormulation == optimistic) {
    return -1;
  }
  else {
    cout << "Unknown problem formulation: " << problemFormulation << endl;
    exit(0);
  }
}
```

```
bool Environment::goal() {
  return state.goal();
}

float Environment::getP() {
  return state.getP();
}

float Environment::getV() {
  return state.getV();
}

ActionSet Environment::getLegalActions() {
  return legalActions;
}
```

## A.7 FunctionApproximator

### FunctionApproximator.h

```
--------------------------------------------------------------------------

#ifndef FUN_APPROX_H
#define FUN_APPROX_H

#include <cstdio>
#include <cmath>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;

#include "Action.h"
#include "State.h"
#include "EligibilityTrace.h"
#include "Environment.h"
#include "Constants.h"

class EligibilityTrace;
struct cell {int actIdx; int posBin; int velBin;};
struct FAState {float x; float y;};

class FunctionApproximator {

public:

  void init();

  //Basic operations
  void updateActionStateValues(EligibilityTrace *et, float alphaxdelta);
  void getBestActionAndValue(Environment *e, Action &a, float &Value);
```

```
  void setFeatures(State s, Action a);
  void setFeatures(FAState fas, Action a);
  float computeValueOfFeatures();
  cell getFeature(int t);   //used by the eligibility trace update step to
                            //determine which features should be incremented
  float getValueOfFAStateAction(FAState fas, Action a);
  float getValueOfStateAction(State fas, Action a);
  float getValueOfCell(float bottomLeftX, float bottomLeftY, float sampleTileSizeX,
          float sampleTileSizeY, Action tileAction);
  float getValueOfCell(int t, int a, int p, int v);

  //Specific to merge methods requiring an ordering based on distance between tiles
  void getTilesOffsetOrder(int *offsetOrderedTiles);

  //Methods for combining FAs
  void mergeToMax(vector<FunctionApproximator> &agentsFAs, ofstream &out, bool print);
  void mergeToMin(vector<FunctionApproximator> &agentsFAs, ofstream &out, bool print);
  void initToMin(vector<FunctionApproximator> &agentsFAs, ofstream &out, bool print);
  void mergeFA(FunctionApproximator fa, float weight);
  void mergeFA2(FunctionApproximator fa, float weight, ofstream &out, bool print);
  void mergeFA3(FunctionApproximator fa, float weight, ofstream &out, bool print);

  //Print/display methods
  void print(ofstream &out);
  void printFAconfig(ofstream &out);
  void printFAforGraphing(string fileName, int resolution, Environment *e);
  void displayFeatures();

  //void setEligibilityTrace(EligibilityTrace *et);

private:

  cell features[TILES];   //holds the currently active features, one on each tile

  float tiles[TILES][ACTIONS][XCELLS][YCELLS];  //holds the value for a cell/feature
  float offset[TILES][2]; // offset of each tile, 0 - pos, 1 - vel

  float stateSpaceXoffset;
  float stateSpaceYoffset;
  float cellSizeX;
  float cellSizeY;

  State convFAStoState(FAState fas);

  int getCellX(int t, float X);
  int getCellY(int t, float Y);

};

#endif
```

**FunctionApproximator.cpp**

```
--------------------------------------------------------------------------
```

```
#include "FunctionApproximator.h"

void FunctionApproximator::init() {

  cellSizeX = ABSPOS_RANGE / (XCELLS-1);
  cellSizeY = ABSVEL_RANGE / (YCELLS-1);

  stateSpaceXoffset = cellSizeX;
  stateSpaceYoffset = cellSizeY;

  #ifdef TILESETUPDEBUG
    printf("\tcell size - X: %f Y: %f\n", cellSizeX, cellSizeY);
    printf("\tstateSpaceOffset - X: %f Y: %f\n", stateSpaceXoffset, stateSpaceYoffset);
    printf("\tGenerating X tile offsets between %f and %f\n", 0.0, cellSizeX);
    printf("\tGenerating Y tile offsets between %f and %f\n", 0.0, cellSizeY);
  #endif
  for (int t = 0; t<TILES; t++) {
    offset[t][0] = generateRandomNumber(0, cellSizeX);
    offset[t][1] = generateRandomNumber(0, cellSizeY);
    #ifdef TILESETUPDEBUG
      printf("\tOffset tile %d - X: %f Y: %f\n", t, offset[t][0], offset[t][1]);
    #endif
  }

  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          tiles[t][a][p][v] = 0.0;
        }
      }
    }
  }
}

void FunctionApproximator::initToMin(vector<FunctionApproximator> &agentsFAs,
        ofstream &out, bool print) {
  init();

  for (int a = 0; a<ACTIONS; a++) {
    for (int t = 0; t<TILES; t++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
      float min;
      int best;
          Action cellAction;
          cellAction.setActionByIdx(a);

          float cellBottomLeftX = offset[t][0]+(p*cellSizeX);
          float cellBottomLeftY = offset[t][1]+(v*cellSizeY);

          min = agentsFAs[0].getValueOfCell(cellBottomLeftX, cellBottomLeftY,
           cellSizeX, cellSizeY, cellAction);
          best = 0;
```

```
      for (unsigned s = 1; s<agentsFAs.size(); s++) {
            //retreive all FA's approximations of a cell and take the min
       float cellValue = agentsFAs[s].getValueOfCell(cellBottomLeftX, cellBottomLeftY,
           cellSizeX, cellSizeY, cellAction);
            if (cellValue < min) {
          min = cellValue;
          best = s;
            }
      }
      tiles[t][a][p][v] = min;
      if (print) {
      out << t << a << p << v << " : (" << cellBottomLeftX << ", " << cellBottomLeftY << ") : "
                << best << " : " << min << endl;
      }
        }
      }
    }
  }
}


void FunctionApproximator::mergeToMax(vector<FunctionApproximator> &agentsFAs,
        ofstream &out, bool print) {

  for (int t = 0; t<TILES; t++) {
   for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          float max;
          int best;

          max = agentsFAs[0].getValueOfCell(t,a,p,v);
          best = 0;
      for (unsigned s = 1; s<agentsFAs.size(); s++) {
       float cellValue = agentsFAs[s].getValueOfCell(t,a,p,v);
       if (cellValue > max) {
         max = cellValue;
         best = s;
            }
      }
      tiles[t][a][p][v] = max;
      if (print) {
      out << t << a << p << v <<  " : " << best << " : " << max << endl;
      }
        }
      }
    }
  }
}

void FunctionApproximator::mergeToMin(vector<FunctionApproximator> &agentsFAs,
        ofstream &out, bool print) {

  for (int t = 0; t<TILES; t++) {
```

```
   for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          float min;
          int best;

          min = agentsFAs[0].getValueOfCell(t,a,p,v);
          best = 0;
      for (unsigned s = 1; s<agentsFAs.size(); s++) {
       float cellValue = agentsFAs[s].getValueOfCell(t,a,p,v);
       if (cellValue < min) {
         min = cellValue;
         best = s;
            }
      }
      tiles[t][a][p][v] = min;
      if (print) {
      out << t << a << p << v <<  " : " << best << " : " << min << endl;
      }
        }
      }
    }
  }
}




float FunctionApproximator::getValueOfCell(int t, int a, int p, int v) {
  return tiles[t][a][p][v];
}


void FunctionApproximator::updateActionStateValues(EligibilityTrace *et, float alphaxdelta) {
  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          tiles[t][a][p][v] += alphaxdelta*et->val(t,a,p,v);
        }
      }
    }
  }
}


int FunctionApproximator::getCellX(int t, float X) {
  float cellX = offset[t][0];
  if (X < cellX) {
    return -1;
  }
  for (int i = 0; i<XCELLS; i++) {
    if ( X >= cellX && X <= cellX+cellSizeX)
      return i;
```

```
    else
      cellX += cellSizeX;
  }
  return -1;
}

int FunctionApproximator::getCellY(int t, float Y) {
  float cellY = offset[t][1];
  if (Y < cellY) {
    return -1;
  }
  for (int j = 0; j<YCELLS; j++) {
    if ( Y >= cellY && Y <= cellY+cellSizeY)
      return j;
    else
      cellY += cellSizeY;
  }
  return -1;
}

void FunctionApproximator::setFeatures(State s, Action a) {

  float adjustXtoPositiveRange;
  float adjustYtoPositiveRange;

  // Compute position and velocity representation on tile's grid

  if (POSRANGE_LOW<0)
    adjustXtoPositiveRange = fabs(POSRANGE_LOW);
  else
    adjustXtoPositiveRange = (-1)*POSRANGE_LOW;

  if (VELRANGE_LOW<0)
    adjustYtoPositiveRange = fabs(VELRANGE_LOW);
  else
    adjustYtoPositiveRange = (-1)*VELRANGE_LOW;

  float X = s.getP() + // car position
    adjustXtoPositiveRange + stateSpaceXoffset;
    // +/- the lower range bound to bring the pos into the correct range
  float Y = s.getV() + // car position
    adjustYtoPositiveRange + stateSpaceYoffset;
    // +/- the lower range bound to bring the pos into the correct range

  #ifdef STEPDEBUG
    cout << "setfeatures(): state mapped on tiles (" << X << ", " << Y << ")\n";
  #endif

  for (int t = 0; t<TILES; t++) {
    features[t].posBin = getCellX(t, X);
    features[t].velBin = getCellY(t, Y);

    if (features[t].posBin == -1) {
      cout << "STATE NOT IN FA RANGE" << endl;
```

```
      features[t].actIdx = -1;
      features[t].velBin = -1;
    }
    else if (features[t].velBin == -1) {
      cout << "STATE NOT IN FA RANGE" << endl;
      features[t].actIdx = -1;
      features[t].posBin = -1;
    }
    else {
      features[t].actIdx = a.getIdx();
    }
  }
}

void FunctionApproximator::setFeatures(FAState fas, Action a) {
   for (int t = 0; t<TILES; t++) {
    features[t].posBin = getCellX(t, fas.x);
    features[t].velBin = getCellY(t, fas.y);

    if (features[t].posBin == -1) {
      features[t].actIdx = -1;
      features[t].velBin = -1;
    }
    else if (features[t].velBin == -1) {
      features[t].actIdx = -1;
      features[t].posBin = -1;
    }
    else {
      features[t].actIdx = a.getIdx();
    }
  }
}


void FunctionApproximator::displayFeatures() {
  printf("features:\nTile Act Pos Vel\n");
  for (int t = 0; t<TILES; t++) {
    printf("%3d  %3d %3d %3d\n", t, features[t].actIdx, features[t].posBin, features[t].velBin);
  }
}


float FunctionApproximator::computeValueOfFeatures() {
  float sum = 0;
  for (int t = 0; t<TILES; t++) {
    if (features[t].actIdx != -1) {
      int action = features[t].actIdx;
      int posBin = features[t].posBin;
      int velBin = features[t].velBin;
      sum += tiles[t][action][posBin][velBin];
    }
  }
  return sum;
}
```

```
float FunctionApproximator::getValueOfStateAction(State s, Action a) {
  cell backupFeatures[TILES];
  for (int i = 0; i<TILES; i++) {
    backupFeatures[i] = features[i];
  }
  setFeatures(s,a);
  float value = computeValueOfFeatures();
  for (int i = 0; i<TILES; i++) {
    features[i] = backupFeatures[i];
  }
  return value;
}

float FunctionApproximator::getValueOfFAStateAction(FAState fas, Action a) {
  cell backupFeatures[TILES];
  for (int i = 0; i<TILES; i++) {
    backupFeatures[i] = features[i];
  }
  setFeatures(fas,a);
  float value = computeValueOfFeatures();
  for (int i = 0; i<TILES; i++) {
    features[i] = backupFeatures[i];
  }
  return value;
}


cell FunctionApproximator::getFeature(int t) {
  return features[t];
}

void FunctionApproximator::printFAconfig(ofstream &out) {
  out << "Cell Size: (" << cellSizeX << ", " << cellSizeY << ")" << endl;
  out << "State Space Offset: (" << stateSpaceXoffset << ", " << stateSpaceYoffset << ")" << endl;
  for (int t = 0; t<TILES; t++) {
    out << "Tile  " << t << " offset: (" << offset[t][0] << ", " << offset[t][1] << ")" <<endl;
  }
}



void FunctionApproximator::print(ofstream &out) {
  int width = 6;
  out.precision(2);

  for (int a=0; a<ACTIONS; a++) {
    float xDiv = ABSPOS_RANGE/10;
  float yDiv = ABSVEL_RANGE/10;
  float x = stateSpaceXoffset;
  float y = stateSpaceYoffset;

    out << endl << "FunctionApproximator " << a << endl;
    out.width(width);
```

```
    out << "\t";
    for (int i=0; i<10; i++) {
      out.width(width);
      out << x << "\t";
      x += xDiv;
    }
    out << endl;


    for (int j=0; j<10; j++) {
      out.width(width);
      out << y << "\t";
      x = stateSpaceXoffset;
      for (int i=0; i<10; i++) {
        FAState fas;
        Action actionFA;
        actionFA.setActionByIdx(a);
        fas.x = x;
        fas.y = y;
        out.width(width);
        out << getValueOfFAStateAction(fas, actionFA) << "\t";
        x += xDiv;
      }
      out << endl;
      y += yDiv;
    }
  }
}


void FunctionApproximator::printFAforGraphing(string fileName, int resolution, Environment *e) {

  ofstream out(fileName.c_str());

  float xDiv = ABSPOS_RANGE/resolution;
  float yDiv = ABSVEL_RANGE/resolution;
  float x = POSRANGE_LOW;
  float y = VELRANGE_LOW;

  for (int j=0; j<(resolution+1); j++) {
    x = POSRANGE_LOW;
    for (int i=0; i<(resolution+1); i++) {
      State s;
      Action a;
      s.setState(x, y);
      e->reset(s);
      float value;
      getBestActionAndValue(e, a, value); //discard a - not interested in best action
      out << s << " " << value << endl;
      x += xDiv;
    }
    out << endl;
    y += yDiv;
  }
```

```
}

State FunctionApproximator::convFAStoState(FAState fas) {
  State s;
  float adjustXtoPositiveRange;
  float adjustYtoPositiveRange;

  // Compute real state space position and velocity from FAState

  if (POSRANGE_LOW<0)
    adjustXtoPositiveRange = fabs(POSRANGE_LOW);
  else
    adjustXtoPositiveRange = (-1)*POSRANGE_LOW;

  if (VELRANGE_LOW<0)
    adjustYtoPositiveRange = fabs(VELRANGE_LOW);
  else
    adjustYtoPositiveRange = (-1)*VELRANGE_LOW;

  float realX = fas.x -
    adjustXtoPositiveRange - stateSpaceXoffset;
  float realY = fas.y -
    adjustYtoPositiveRange - stateSpaceYoffset;

  s.setState(realX, realY);
  return s;
}


void FunctionApproximator::mergeFA(FunctionApproximator fa, float weight) {

  //cout << "Control FA: ";
  int offsetOrderedTilesThis[TILES];
  int offsetOrderedTilesFA[TILES];
  getTilesOffsetOrder(offsetOrderedTilesThis);
  //cout << endl << "Passed FA: ";
  fa.getTilesOffsetOrder(offsetOrderedTilesFA);

  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          float difference = fa.getValueOfCell(offsetOrderedTilesFA[t],a,p,v) -
            tiles[offsetOrderedTilesThis[t]][a][p][v];

          tiles[offsetOrderedTilesThis[t]][a][p][v] += difference*weight;
        }
      }
    }
  }
}

void FunctionApproximator::getTilesOffsetOrder(int *offsetOrderedTiles) {
  float scaleRatio;
```

```
  if (cellSizeX > cellSizeY) {
    scaleRatio = (cellSizeX/cellSizeY);
  }
  else {
    scaleRatio = (cellSizeY/cellSizeX);
  }

  bool addedTiles[TILES];
  for (int i = 0; i<TILES; i++) {
    addedTiles[i] = false;
  }

  for (int i = 0; i<TILES; i++) {
    float minOffset = sqrt(pow(cellSizeX,2)+pow(cellSizeY*scaleRatio,2));
    //set min offset to maxium so any other offset we test will be smaller
    int minOffsetTile = 0;
    for (int t = 0; t<TILES; t++) {
      if (addedTiles[t] == false) {
        float tileOffset = sqrt(pow(offset[t][0],2)+pow(offset[t][1]*scaleRatio,2));
        if (tileOffset <= minOffset) {
          minOffset = tileOffset;
          minOffsetTile = t;
        }
      }
    }
    addedTiles[minOffsetTile] = true;
    offsetOrderedTiles[i] = minOffsetTile;
  }
}


void FunctionApproximator::mergeFA2(FunctionApproximator fa, float weight,
        ofstream &out, bool printMergeLog) {

  if (printMergeLog) print(out);

  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          float sum = 0;
          float fasValue;
          out.precision(5);
          if (printMergeLog) out << t << a << p << v << "\t";
          FAState fas;
          Action tileAction;
          tileAction.setActionByIdx(a);
            // bottom left
          fas.x = offset[t][0]+p*cellSizeX;
          fas.y = offset[t][1]+v*cellSizeY;
          fasValue = fa.getValueOfFAStateAction(fas, tileAction);
          if (printMergeLog) out << "bl (" << fas.x << ", " << fas.y << ") -> " << fasValue;
          sum += fasValue;
          // bottom right
```

```
          fas.x = offset[t][0]+(p+1)*cellSizeX;
          fas.y = offset[t][1]+v*cellSizeY;
          fasValue = fa.getValueOfFAStateAction(fas, tileAction);
          if (printMergeLog) out << " br (" << fas.x << ", " << fas.y << ") -> " << fasValue;
          sum += fasValue;
          // top left
          fas.x = offset[t][0]+p*cellSizeX;
          fas.y = offset[t][1]+(v+1)*cellSizeY;
          fasValue = fa.getValueOfFAStateAction(fas, tileAction);
          if (printMergeLog) out << " tl (" << fas.x << ", " << fas.y << ") -> " << fasValue;
          sum += fasValue;
          // top right
          fas.x = offset[t][0]+(p+1)*cellSizeX;
          fas.y = offset[t][1]+(v+1)*cellSizeY;
          fasValue = fa.getValueOfFAStateAction(fas, tileAction);
          if (printMergeLog) out << " tr (" << fas.x << ", " << fas.y << ") -> " << fasValue;
          sum += fasValue;

          float difference = sum/(4*TILES) - tiles[t][a][p][v];

          if (printMergeLog) out << " thistile: " << tiles[t][a][p][v] <<
                 " fatile: " << sum/(4*TILES) <<
                 " diff: " << difference <<
                 " Diff*weight: " << difference*weight;

          tiles[t][a][p][v] += difference*weight;
          if (printMergeLog) out << " newvalue: " << tiles[t][a][p][v] << endl;

         // if ( tiles[t][a][p][v] < -10)
         //   exit(0);
        }
      }
    }
  }
  out << endl << endl << "finalfa:" << endl;
  if (printMergeLog) print(out);
}


void FunctionApproximator::mergeFA3(FunctionApproximator fa, float weight,
        ofstream &out, bool printMergeLog) {

  for (int t = 0; t<TILES; t++) {
    for (int a = 0; a<ACTIONS; a++) {
      for (int p = 0; p<XCELLS; p++) {
        for (int v = 0; v<YCELLS; v++) {
          float sum = 0;
          out.precision(5);
          if (printMergeLog) out << t << a << p << v << "\t";
          Action tileAction;
          tileAction.setActionByIdx(a);

          float cellBottomLeftX = offset[t][0]+(p*cellSizeX);
          float cellBottomLeftY = offset[t][1]+(v*cellSizeY);
```

```
          float difference =
            (
              getValueOfCell(cellBottomLeftX, cellBottomLeftY,
                cellSizeX, cellSizeY, tileAction)
              /
              (5*TILES)
            )
            - tiles[t][a][p][v];

          if (printMergeLog) out << " thistile: " << tiles[t][a][p][v] <<
                " fatile: " << sum/(5*TILES) <<
                " diff: " << difference <<
                " Diff*weight: " << difference*weight;

          tiles[t][a][p][v] += difference*weight;
          if (printMergeLog) out << " newvalue: " << tiles[t][a][p][v] << endl;

        }
      }
    }
  }
}



float FunctionApproximator::getValueOfCell(float bottomLeftX, float bottomLeftY,
        float sampleTileSizeX, float sampleTileSizeY, Action tileAction) {

  float sum = 0;
  float fasValue;
  float xOffsetFromCenter = sampleTileSizeX/4;
  float yOffsetFromCenter = sampleTileSizeY/4;
  FAState fas;

  // center
  fas.x = bottomLeftX+sampleTileSizeX/2;
  fas.y = bottomLeftY+sampleTileSizeY/2;
  fasValue = getValueOfFAStateAction(fas, tileAction);
  sum += fasValue;

  // bottom left
  fas.x = bottomLeftX+sampleTileSizeX/2-xOffsetFromCenter;
  fas.y = bottomLeftY+sampleTileSizeY/2-yOffsetFromCenter;
  fasValue = getValueOfFAStateAction(fas, tileAction);
  sum += fasValue;

  // bottom right
  fas.x = bottomLeftX+sampleTileSizeX/2+xOffsetFromCenter;
  fas.y = bottomLeftY+sampleTileSizeY/2-yOffsetFromCenter;
  fasValue = getValueOfFAStateAction(fas, tileAction);
  sum += fasValue;

  // top left
```

```
  fas.x = bottomLeftX+sampleTileSizeX/2-xOffsetFromCenter;
  fas.y = bottomLeftY+sampleTileSizeY/2+yOffsetFromCenter;
  fasValue = getValueOfFAStateAction(fas, tileAction);
  sum += fasValue;

  // top right
  fas.x = bottomLeftX+sampleTileSizeX/2+xOffsetFromCenter;
  fas.y = bottomLeftY+sampleTileSizeY/2+yOffsetFromCenter;
  fasValue = getValueOfFAStateAction(fas, tileAction);
  sum += fasValue;

  return sum/(5*TILES);
}



void FunctionApproximator::getBestActionAndValue(Environment *e, Action &a, float &value) {
  float bestValue;
  ActionSet legalActions = e->getLegalActions();
  Action bestAction;
  State state = e->getState();

  cell backupFeatures[TILES];
  for (int i = 0; i<TILES; i++) {
    backupFeatures[i] = features[i];
  }

  legalActions.resetIterator();

  for (int i = 0; i<legalActions.count(); i++) {
    if (i == 0) {
      bestAction = legalActions.getNextAction();
      setFeatures(state, bestAction);
      bestValue = computeValueOfFeatures();
    }
    else {
      Action trialAction = legalActions.getNextAction();
      setFeatures(state, trialAction);
      float trialValue = computeValueOfFeatures();

      if (trialValue > bestValue) {
        bestAction = trialAction;
        bestValue = trialValue;
      }
      else if (trialValue == bestValue) {
        if (generateRandomNumber(0,1) > 0.5) {
          bestAction = trialAction;
          bestValue = trialValue;
        }
      }
    }
  }
  a = bestAction;
  value = bestValue;
```

```
  for (int i = 0; i<TILES; i++) {
    features[i] = backupFeatures[i];
  }
}
```

## A.8 Logger

### Logger.h

```
--------------------------------------------------------------------------

#ifndef LOGGER_H
#define LOGGER_H

#include <cstdlib>
#include <cstdio>
#include <vector>
#include <iostream>
#include <fstream>
using namespace std;

#include "Constants.h"
#include "State.h"
#include "Action.h"
#include "FunctionApproximator.h"


enum actionType {best, exploratory};
struct stepTrace {State state; Action action; State newState;
                  int reward; actionType actionChoice;};
struct episodeTrace {vector<stepTrace> trace; FunctionApproximator initialFA;
                     FunctionApproximator finalFA;};

class Logger {

public:

  void reset();
  void newEpisode();
  void removeCurrentEpisode();
  void addStep(stepTrace trace);
  void addInitialFA(FunctionApproximator fa);
  void addFinalFA(FunctionApproximator fa);
  float getAvgStepsPerEpisode();
  float getProbabilityOfEscape();
  void printLog(ofstream &out, bool printFAs, bool printStats);

private:

  vector<episodeTrace> episodeTraces;

};
```

```
#endif
```

### Logger.cpp

```
--------------------------------------------------------------------------

#include "Logger.h"

void Logger::reset() {
  episodeTraces.clear();
}

void Logger::newEpisode() {
  episodeTrace newtrace;
  episodeTraces.push_back(newtrace);
}

void Logger::removeCurrentEpisode() {
  episodeTraces.pop_back();
}

void Logger::addStep(stepTrace step) {
  episodeTraces.back().trace.push_back(step);
}

void Logger::addInitialFA(FunctionApproximator fa) {
  episodeTraces.back().initialFA = fa;
}

void Logger::addFinalFA(FunctionApproximator fa) {
  episodeTraces.back().finalFA = fa;
}

void Logger::printLog(ofstream &out, bool printFAs, bool printStats) {

  for(unsigned e = 0; e<episodeTraces.size(); e++) {
    out << "Episode: " << e << "\n";

      if (printFAs) {
        out << "FA Details: " << endl;
        episodeTraces[e].initialFA.printFAconfig(out);
        out << endl << "InitialFA:" << endl;
        episodeTraces[e].initialFA.print(out);
      }
      out << endl;

    if (episodeTraces[e].trace.size() != NUM_STEPS_FOR_FAILURE) {
      for (unsigned s = 0; s<episodeTraces[e].trace.size(); s++) {
        out << s
            << "\t"<< episodeTraces[e].trace[s].state
            << "\t" << episodeTraces[e].trace[s].action
            << "\t" << episodeTraces[e].trace[s].newState
```

```
          << "\t" << episodeTraces[e].trace[s].reward
          << "\t" << (episodeTraces[e].trace[s].actionChoice == best ? "b" : "e")
          << "\n";
      }
      out << endl;
      if (printFAs) {
        out << endl << "FinalFA:" << endl;
        episodeTraces[e].finalFA.print(out);
      }
      out << endl;
    }
    else {
      out << "****************************************************************" << endl;
      out << "FAILED EPISODE (ran for > " << NUM_STEPS_FOR_FAILURE << " steps), Start State: "
        << episodeTraces[e].trace[0].state << endl;
      out << "****************************************************************" << endl;
    }
  }
  if (printStats) {
    out << "Number of episodes: " << episodeTraces.size() << endl;
    out << "Average number of steps per episode: " << getAvgStepsPerEpisode() << endl;
    out << "Chance of escape: " << getProbabilityOfEscape() << endl;
  }
}


float Logger::getAvgStepsPerEpisode() {
  int sum = 0;
  for (unsigned e = 0; e<episodeTraces.size(); e++) {
    sum += episodeTraces[e].trace.size();
  }
  return (sum/(float)(episodeTraces.size()));
}

float Logger::getProbabilityOfEscape() {
  int sum = 0;
  for (unsigned i = 0; i<episodeTraces.size(); i++) {
    sum += episodeTraces[i].trace.back().newState.goal();
  }
  return (float)sum/(float)episodeTraces.size();
}
```

## A.9 Random

### Random.h

```
--------------------------------------------------------------------------

#ifndef RANDOM_H
#define RANDOM_H

#include <ctime>
#include <cstdlib>
```

```
#include <iostream>
using namespace std;


void setupRandomNumberGenerator();

float generateRandomNumber(float low, float high);

#endif
```

### Random.cpp

```
--------------------------------------------------------------------------

#include "Random.h"

void setupRandomNumberGenerator() {
  unsigned timeNow = (unsigned)time( NULL );
  cout << "The time is: " << timeNow << "\n";
  srand(timeNow);
}

float generateRandomNumber(float low, float high) {

  float range;

  if (low<0 && high<0) {
    range = abs(low-high);
    return ((float) (rand()/ (RAND_MAX/range))) + low ;
  }
  else if (low<0) {
    range = abs(high-low);
    return ((float) (rand()/ (RAND_MAX/range))) + low ;
  }
  else {
    range = abs(high-low);
    return ((float) (rand()/ (RAND_MAX/range))) + low ;
  }
}
```

## A.10 Simulation

### Simulation.h

```
--------------------------------------------------------------------------

#ifndef SIMULATION_H
#define SIMULATION_H

#include <cstdio>
#include <vector>
using namespace std;
```

```
#include "Types.h"
#include "Logger.h"
#include "Agent.h"
#include "Environment.h"
#include "ActionSet.h"

class Simulation {

public:

  void init(FunctionApproximator fa,
            ProblemFormulation pf,
            ETraceMode traceMode,
            float epsilon,
            float alpha,
            float gamma,
            float lambda);

  void doTrial(int maxSteps);
  void doTrial();    //Starts a trial with best action

  float getMaxAltitude();
  int getSteps();

  void evaluate(vector<State> const &testStates,
                float &avgStepsPerEpisode,
                float &probabilityOfEscape,
                bool printEvaluationLog,
                string logFileName);

  void printLearningLog(ofstream &out);

  FunctionApproximator getFunctionApproximator();
  void setFunctionApproximator(FunctionApproximator fa);

  void printLegalMoves();

private:

  Environment environment;
  Agent agent;
  Logger learningLogger;
  Logger evaluationLogger;

  ProblemFormulation problemFormulation;
  int step;

};

#endif
```

**Simulation.cpp**

```
------------------------------------------------------------------------
#include "Simulation.h"


void Simulation::init(FunctionApproximator fa, ProblemFormulation pf,
                      ETraceMode traceMode, float epsilon, float alpha,
                      float gamma, float lambda) {

  environment.init(pf);

  agent.init(&environment, traceMode, epsilon, alpha, gamma, lambda);
  agent.setFunctionApproximator(fa);

  learningLogger.reset();
  evaluationLogger.reset();

}

void Simulation::printLegalMoves() {
  cout << "Sim: ";
  ActionSet testactions = environment.getLegalActions();
  testactions.print();
  cout << endl;

  cout << "Agent: ";
  agent.printLegalMoves();
  cout << endl;
}


void Simulation::doTrial(int maxSteps) {

  step = 0;
  learningLogger.newEpisode();

  environment.resetToRandomState();

  if (problemFormulation == optimistic) {
    agent.initializeLearningTrial(best);
  }
  else {
    agent.initializeLearningTrial(exploratory);
  }
  learningLogger.addInitialFA(agent.getFunctionApproximator());
  while (!environment.goal() && (step < maxSteps)) {
    learningLogger.addStep(agent.learningStep());
    step++;
  }
  learningLogger.addFinalFA(agent.getFunctionApproximator());
}
```

```
void Simulation::doTrial() {
  step = 0;
  learningLogger.newEpisode();
  environment.resetToRandomState();
  if (problemFormulation == optimistic) {
    agent.initializeLearningTrial(best);
  }
  else {
    agent.initializeLearningTrial(exploratory);
  }
  learningLogger.addInitialFA(agent.getFunctionApproximator());
  while (!environment.goal()) {
    learningLogger.addStep(agent.learningStep());
    step++;
    if (step > 5000) {
      cout << "warning 5000 steps reached" << endl;
    }
  }
  learningLogger.addFinalFA(agent.getFunctionApproximator());
}

void Simulation::printLearningLog(ofstream &out) {
  learningLogger.printLog(out, true, true);
}

float Simulation::getMaxAltitude() {
  return agent.getMaxAltitude();
}

int Simulation::getSteps() {
  return step;
}

FunctionApproximator Simulation::getFunctionApproximator() {
  return agent.getFunctionApproximator();
}

void Simulation::setFunctionApproximator(FunctionApproximator fa) {
  agent.setFunctionApproximator(fa);
}

void Simulation::evaluate(vector<State> const &testStates, float &avgStepsPerEpisode,
                          float &probabilityOfEscape, bool printEvaluationLog,
                          string logFileName) {

  evaluationLogger.reset();

  for (unsigned i = 0; i<testStates.size(); i++) {
    evaluationLogger.newEpisode();
    environment.reset(testStates[i]);
    agent.initializeTrial();
    evaluationLogger.addInitialFA(agent.getFunctionApproximator());
    int step = 0;
    while (!environment.goal() && step < NUM_STEPS_FOR_FAILURE) {
```

```
      evaluationLogger.addStep(agent.step());
      step++;
    }
  }
  avgStepsPerEpisode = evaluationLogger.getAvgStepsPerEpisode();
  probabilityOfEscape = evaluationLogger.getProbabilityOfEscape();

  if (printEvaluationLog) {
    ofstream out(logFileName.c_str());
    //don't print out fa's as they have not been added
    evaluationLogger.printLog(out, false, true);
  }

  evaluationLogger.reset();
}
```

## A.11 State

### State.h

```
---------------------------------------------------------------------------

#ifndef STATE_H
#define STATE_H

#include <cstdio>
#include <cmath>
#include <iostream>
using namespace std;

#include "Action.h"
#include "Random.h"

class State {

public:

  State();

  void update(Action a);

  void setRandom();
  void setRandom(float lowPositionBound, float highPositionBound,
      float lowVelocityBound, float highVelocityBound);
  void setState(float p, float v);
  bool legalAction(Action a);

  bool goal();

  float getV();
  float getP();
  float getAltitude();
```

```
  friend ostream &operator<<(ostream &stream, State s);

private:

  float carVelocity;
  float carPosition;

  float boundVelocity(float x);
  float boundPosition(float x);


};

#endif
```

### State.cpp

```
---------------------------------------------------------------------------

#include "State.h"

State::State() {
}

float State::getP() {
  return carPosition;
}

float State::getV() {
  return carVelocity;
}

void State::setState(float p, float v) {
  carPosition = p;
  carVelocity = v;
}

float State::getAltitude() {
  return sin(3*carPosition);
}

bool State::legalAction(Action a) {
  return true;
}

void State::update(Action a) {
  carVelocity = boundVelocity(carVelocity + 0.001*a.getValue() + (-0.0025*cos(3*carPosition)));
  carPosition = boundPosition(carPosition + carVelocity);
  if (carPosition <= -1.2) { //at left
    carVelocity = 0;         // so reset the velocity
  }
}

bool State::goal() {
```

```
  if (carPosition >= 0.5)
    return true;
  else
    return false;
}

void State::setRandom() {
   carPosition = generateRandomNumber(-1.2, 0.5);
   carVelocity = generateRandomNumber(-0.07, 0.07);
}

void State::setRandom(float lowPositionBound, float highPositionBound,
      float lowVelocityBound, float highVelocityBound) {
   carPosition = generateRandomNumber(lowPositionBound, highPositionBound);
   carVelocity = generateRandomNumber(lowVelocityBound, highVelocityBound);
}

float State::boundVelocity(float x) {
  if (x < -0.07) {
    return -0.07;
  }
  if (x > 0.07) {
    return 0.07;
  }
  return x;
}

float State::boundPosition(float x) {
  if (x < -1.2) {
    return -1.2;
  }
  if (x > 0.5) {
    return 0.5;
  }
  return x;
}

ostream &operator<<(ostream &stream, State s) {
  stream << "" << s.carPosition << "\t" << s.carVelocity << "";
  return stream;
}
```

## A.12 Statistics

### Statistics.h

```
--------------------------------------------------------------------------

#ifndef STATISTICS_H
#define STATISTICS_H

#include <cstdlib>
#include <iostream>
```

```
#include <vector>
using namespace std;

float meanFun(vector<float> results);

float standardError(vector<float> results);

#endif
```

### Statistics.cpp

```
------------------------------------------------------------------------

#include "Statistics.h"

float meanFun(vector<float> results) {
  float sum = 0.0;
  for (unsigned i = 0; i<results.size(); i++) {
   sum += results[i];
  }
  return sum/results.size();
}

float standardError(vector<float> results) {
  float sum = 0.0;
  unsigned numResults = results.size();
  float mean = meanFun(results);
  for (unsigned i = 0; i<numResults; i++) {
   sum += pow(results[i]-mean,2);
  }
  return sqrt(sum / (float)(numResults-1)) / sqrt((float)numResults);
}
```

## A.13 Test

### Test.h

```
------------------------------------------------------------------------

#ifndef TEST_H
#define TEST_H

#include <cstdio>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

#include "Types.h"
#include "Simulation.h"
#include "FunctionApproximator.h"
#include "State.h"
#include "ConfigFile.h"
#include "Statistics.h"
```

```
class Test {

public:

  void init(int numberOfAgents,
            FAMergeMethod faMergeMethod,
            ConfigFile config);

  void run(int episodeCutoff,
           int stepCutoff,
           int extraLearningStepCutoff,
           FunctionApproximator templateFA,
           vector<State> const &testStates,
           string outputDirPrefix,
           float &avgStepsPerEpisode,
           float &probabilityOfEscape);

private:

  vector<Simulation> simulations;

  int numberOfAgents;
  FAMergeMethod faMergeMethod;
  ProblemFormulation problemFormulation;
  ETraceMode eTraceMode;

  bool performExtraLearningUsingSingleAgent;
  bool evaluateAllFAs;

  float epsilon;
  float alpha;
  float gamma;
  float lambda;

  bool printLearning;
  bool printEvaluation;
  bool printFAdata;
  bool printFAdataFrequency;

};

#endif
```

### Test.cpp

```
--------------------------------------------------------------------------

#include "Test.h"

void Test::init(int numAgents, FAMergeMethod mergeMethod, ConfigFile config) {

  numberOfAgents = numAgents;
  faMergeMethod = mergeMethod;
  problemFormulation = config.getProblemFormulation("problem_formulation");
```

```
  eTraceMode = config.getETraceMode("etrace_mode");

  performExtraLearningUsingSingleAgent =
      config.getBool("perform_extra_learning_using_single_agent");
  evaluateAllFAs = config.getBool("evaluate_all_fas");

  epsilon = config.getFlt("epsilon");
  alpha = config.getFlt("alpha");
  gamma = config.getFlt("gamma");
  lambda = config.getFlt("lambda");

  printLearning = config.getBool("print_learning_log");
  printEvaluation = config.getBool("print_evaluation_log");
  printFAdata = config.getBool("print_fa_graphing_data");
  printFAdataFrequency = config.getInt("print_fa_graphing_data_frequency");

}


void Test::run(int episodeCutoff, int stepCutoff, int extraLearningStepCutoff,
               FunctionApproximator templateFA,
               vector<State> const &testStates, string outputDirPrefix,
               float &avgStepsPerEpisode, float &probabilityOfEscape) {

  ofstream mergeFile("mergeFile.txt");


  //Setup
  /////////////////

  simulations.clear();

  Simulation templateSimulation;
  for (int i = 0; i<numberOfAgents; i++) {
    simulations.push_back(templateSimulation);
  }
  for (int i = 0; i<numberOfAgents; i++) {
    if ((faMergeMethod != maxmerge) && (faMergeMethod != minmerge)) {
      FunctionApproximator nonTemplateFA;
      nonTemplateFA.init();
      simulations[i].init(nonTemplateFA, problemFormulation, eTraceMode,
        epsilon, alpha, gamma, lambda);
    }
    else {
      simulations[i].init(templateFA, problemFormulation, eTraceMode,
        epsilon, alpha, gamma, lambda);
    }
  }

  //Learn
  ///////////////////
  cout << "\tTest - Learning" << endl;

  for (int e = 0; e<episodeCutoff; e++) {
```

```
vector<float> FAfitness; //if needed (eval all fas = true)

for (unsigned s = 0; s<simulations.size(); s++) {
  cout << "\t\tLearning Simulation: " << s << endl;
  if (stepCutoff == 0) {
    simulations[s].doTrial();
  }
  else {
    simulations[s].doTrial(stepCutoff);
  }
}

if (printFAdata && ((e % printFAdataFrequency) == 0)) {
  stringstream graphFileName;
  graphFileName << outputDirPrefix << "graphing_" << e << "_first.dat";
  Environment environmentForGraphing;
  simulations[0].getFunctionApproximator().printFAforGraphing(graphFileName.str(),
    100, &environmentForGraphing);
}

if (evaluateAllFAs) {
  ofstream faEvalFile((outputDirPrefix + "fa_values.dat").c_str(), ofstream::app);
  faEvalFile << e << "\t";
  for (unsigned s = 0; s<simulations.size(); s++) {
    float avgSteps, probEscape;
    //cout << "\t\tEvaluating FA: " << s << endl;
    simulations[s].evaluate(testStates, avgSteps, probEscape, false, "");
    FAfitness.push_back(avgSteps);
    faEvalFile << avgSteps << "\t";
}
  faEvalFile.close();
}

if (faMergeMethod != none) {

FunctionApproximator newFA;
newFA = templateFA;
vector<FunctionApproximator> agentsFAs;

  //Collect all FAs in an vector
for (unsigned s = 0; s<simulations.size(); s++) {
  agentsFAs.push_back(simulations[s].getFunctionApproximator());
}

if (faMergeMethod == maxmerge) {
  newFA.mergeToMax(agentsFAs, mergeFile, false);
  }
  else if (faMergeMethod == minmerge) {
    newFA.mergeToMin(agentsFAs, mergeFile, false);
  }
  else if (faMergeMethod == inittomin) {
    newFA.initToMin(agentsFAs, mergeFile, false);
  }
```

```
else if (faMergeMethod == maxaltitude) {
  int maxAltSim = 0;
  float maxAlt = simulations[0].getMaxAltitude();
  for (unsigned s = 1; s<simulations.size(); s++) {
    if (simulations[s].getMaxAltitude() >= maxAlt) {
      maxAltSim = s;
      maxAlt = simulations[s].getMaxAltitude();
    }
  }
  newFA = simulations[maxAltSim].getFunctionApproximator();
}
else if (faMergeMethod == evaluate_select_best) {
  vector<State> quickEvalTestStates;
  for (int i = 0; i<20; i++) {
    State s;
    s.setRandom();
    quickEvalTestStates.push_back(s);
  }
  float avgSteps, probEscape;
  simulations[0].evaluate(quickEvalTestStates,avgSteps,probEscape,false,"");
  int bestSim = 0;
  float bestAvgSteps = avgSteps;
  for (unsigned s = 1; s<simulations.size(); s++) {
    simulations[s].evaluate(quickEvalTestStates,avgSteps,probEscape,false,"");
    if (avgSteps <= bestAvgSteps) {
      bestSim = s;
      bestAvgSteps = avgSteps;
    }
  }
  newFA = simulations[bestSim].getFunctionApproximator();
}
else if (faMergeMethod == evaluate_merge1 || faMergeMethod == evaluate_merge2
        || faMergeMethod == evaluate_merge3) {
  vector<State> quickEvalTestStates;
  for (int i = 0; i<100; i++) {
    State s;
    s.setRandom();
    quickEvalTestStates.push_back(s);
  }

  float avgSteps, probEscape;
  vector<simResultT> simResults;
  for (unsigned s = 0; s<simulations.size(); s++) {
    simulations[s].evaluate(quickEvalTestStates,avgSteps,probEscape,false,"");
    simResultT tmpResult;
   tmpResult.agent = s;
   tmpResult.avgSteps = avgSteps;
   simResults.push_back(tmpResult);
  }

 for (unsigned i = 0; i<simResults.size(); i++) {
   for (unsigned j = 0; j<(simResults.size()-1); j++) {
    if (simResults[j].avgSteps >= simResults[j+1].avgSteps) {
      simResultT tmpResult = simResults[j];
```

```
          simResults[j] = simResults[j+1];
          simResults[j+1] = tmpResult;
        }
      }
    }

      ofstream mergeFile("mergeLog.txt");
      newFA = simulations[simResults[0].agent].getFunctionApproximator();
      if (faMergeMethod == evaluate_merge1) {
        newFA.mergeFA(simulations[simResults[1].agent].getFunctionApproximator(), 0.5);
        newFA.mergeFA(simulations[simResults[2].agent].getFunctionApproximator(), 0.25);
        newFA.mergeFA(simulations[simResults[3].agent].getFunctionApproximator(), 0.125);
      }
      else if (faMergeMethod == evaluate_merge2) {
        newFA.mergeFA2(simulations[simResults[1].agent].getFunctionApproximator(), 0.5,
          mergeFile,false);
        newFA.mergeFA2(simulations[simResults[2].agent].getFunctionApproximator(), 0.25,
          mergeFile,false);
        newFA.mergeFA2(simulations[simResults[3].agent].getFunctionApproximator(), 0.125,
          mergeFile,false);
      }
      else if (faMergeMethod == evaluate_merge3) {
        newFA.mergeFA3(simulations[simResults[1].agent].getFunctionApproximator(), 0.5,
          mergeFile,false);
        newFA.mergeFA3(simulations[simResults[2].agent].getFunctionApproximator(), 0.25,
          mergeFile,false);
        newFA.mergeFA3(simulations[simResults[3].agent].getFunctionApproximator(), 0.125,
          mergeFile,false);
      }
    }

    for (unsigned s = 0; s<simulations.size(); s++) {
      simulations[s].setFunctionApproximator(newFA);
    }

    if (printFAdata && ((e % printFAdataFrequency) == 0)) {
      stringstream graphFileName;
      graphFileName << outputDirPrefix << "graphing_" << e << "_merged.dat";
      Environment environmentForGraphing;
      newFA.printFAforGraphing(graphFileName.str(), 100, &environmentForGraphing);
    }

    if (evaluateAllFAs) {
      ofstream faEvalFile((outputDirPrefix + "fa_values.dat").c_str(), ofstream::app);
      float avgSteps, probEscape;
      //cout << "\t\tEvaluating Merged FA" << endl;
      simulations[0].evaluate(testStates, avgSteps, probEscape, false, "");
      faEvalFile << "\t" << meanFun(FAfitness) << "\t" << avgSteps << "\t" <<
                    meanFun(FAfitness)-avgSteps << endl;
      faEvalFile.close();
  }
  }
}
```

```
  if (performExtraLearningUsingSingleAgent) {
    //cout << "\tTest - Extra Learning" << endl;
    for (int e = 0; e<(episodeCutoff/5); e++) {
      simulations[0].doTrial(extraLearningStepCutoff);
    }
  }

  // Print Learning Log
  if (printLearning) {
    for (unsigned s = 0; s<simulations.size(); s++) {
      stringstream tmpName;
      tmpName << outputDirPrefix << "learning_agent_" << s << ".txt";
      ofstream outputFile(tmpName.str().c_str());
      if (outputFile.fail()) {
        cout << "Error opening file: " << tmpName << endl;
        exit(0);
      }
      simulations[s].printLearningLog(outputFile);
    }
  }

  //Evaluate
  ////////////////

  // (first agent always performs eval in case of just single agent)
  simulations[0].evaluate(testStates, avgStepsPerEpisode, probabilityOfEscape,
      printEvaluation, (outputDirPrefix + "evaluation.txt"));

}
```

## A.14 main

### main.cpp

```
--------------------------------------------------------------------------

#include "Types.h"
#include "Test.h"
#include "State.h"
#include "ConfigFile.h"
#include "Statistics.h"

#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <fstream>
using namespace std;
```

```
void parallel_vs_3_single_learners(ConfigFile config);
void vary_number_of_parallel_learners(ConfigFile config);

int main(int argc, char *argv[]) {

  ConfigFile config(argv[1]);

  setupRandomNumberGenerator();
  cout << "jing0.2" << endl;

  unsigned startTime = (unsigned)time( NULL );
  cout << "jing0.21" << endl;

  string testMode = config.getStr("test_mode");

  if (testMode == "parallel_vs_3_single_learners") {
    parallel_vs_3_single_learners(config);
  }
  else if (testMode == "vary_number_of_parallel_learners") {
    vary_number_of_parallel_learners(config);
  }
  else {
    cout << "Unrecognised test more" << endl;
  }


  cout << "Duration: " << ((unsigned)time(NULL)) - startTime << "\n";
}

void parallel_vs_3_single_learners(ConfigFile config) {

  cout << "jing0.4" << endl;

  string testDir = config.getStr("test_prefix");
  system(("mkdir " + testDir).c_str());
  testDir += "//";

  string resultsAveragedFileName(testDir + config.getStr("test_prefix") + "_avg.dat");
  string resultsDetailsFileName(testDir + config.getStr("test_prefix") + "_det.dat");

  ofstream resultsAveraged;
  ofstream resultsDetails;

  cout << "jing0.5" << endl;

  Test parallelAgentsTest;
  Test singleAgentEqExec;
  Test singleAgentEqCompBySteps;
  Test singleAgentEqCompByEpisodes;
  //Test singleAgent;

  cout << "jing1" << endl;

  int numberOfParallelAgents = config.getInt("number_of_parallel_agents");
```

```
  parallelAgentsTest.init(numberOfParallelAgents,
    config.getFAMergeMethod("parallel_merge_method"), config);
  singleAgentEqExec.init(1, none, config);
  singleAgentEqCompBySteps.init(1, none, config);
  singleAgentEqCompByEpisodes.init(1, none, config);
  //singleAgent.init(1, none, config);

  for (int stepCutoff = config.getInt("step_cutoff_init");
       stepCutoff <= config.getInt("step_cutoff_max");
       stepCutoff += config.getInt("step_cutoff_inc")) {

cout << "jing2" << endl;

   stringstream stepCutoffDirName;
   stepCutoffDirName << "step_cutoff_" << stepCutoff << "//";
   system(("mkdir " + testDir + stepCutoffDirName.str()).c_str());

    resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
   resultsAveraged << "#GRAPH_DETAILS: Parallel | Eq Comp Steps | Eq Comp Episodes | Eq Exec"
                   << endl;
   resultsAveraged << "#" << stepCutoff << endl;
   resultsAveraged << "#" << config.getInt("number_of_repetitions") << endl;
   resultsAveraged << "#" << config.getInt("episode_cutoff_max") << endl;
   resultsAveraged << "#" << numberOfParallelAgents << endl;
   resultsAveraged.close();

    for (int episodeCutoff = config.getInt("episode_cutoff_init");
         episodeCutoff <= config.getInt("episode_cutoff_max");
         episodeCutoff += config.getInt("episode_cutoff_inc")) {

      stringstream episodeCutoffDirName;
      episodeCutoffDirName << "episode_cutoff_" << episodeCutoff << "//";
      system(("mkdir " + testDir + stepCutoffDirName.str()
              + episodeCutoffDirName.str()).c_str());

      string parallelResultsDir(testDir + stepCutoffDirName.str()
              + episodeCutoffDirName.str() + "parallel_rep_0//");
      system(("mkdir " + parallelResultsDir).c_str());

      string singleEqCompStepsResultsDir(testDir + stepCutoffDirName.str()
              + episodeCutoffDirName.str() + "single_eqcompsteps_rep_0//");
      system(("mkdir " + singleEqCompStepsResultsDir).c_str());

      string singleEqCompEpisodesResultsDir(testDir + stepCutoffDirName.str()
              + episodeCutoffDirName.str() + "single_eqcompepisodes_rep_0//");
      system(("mkdir " + singleEqCompEpisodesResultsDir).c_str());

      string singleEqExecResultsDir(testDir + stepCutoffDirName.str()
              + episodeCutoffDirName.str() + "single_eqexec_rep_0//");
      system(("mkdir " + singleEqExecResultsDir).c_str());

      //string singleAgentResultsDir(testDir + stepCutoffDirName.str()
      //        + episodeCutoffDirName.str() + "single_agent_rep_0//");
```

```
      //system(("mkdir " + singleAgentResultsDir).c_str());

      vector<float> results[4][2];  // [x][0] for avgStepsPerEpisode, [x][1] prob of escape

      for (int repetition = 0;
           repetition < config.getInt("number_of_repetitions");
           repetition++) {

        float avgStepsPerEpisode, probabilityOfEscape;

        FunctionApproximator templateFA;
        templateFA.init();

        vector<State> testStates;
        for (int i = 0; i<config.getInt("number_of_test_states"); i++) {
          State s;
          s.setRandom();
          testStates.push_back(s);
        }

        cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " Parallel Learner" << endl;
        parallelAgentsTest.run(episodeCutoff, stepCutoff, stepCutoff,
                               templateFA, testStates, parallelResultsDir,
                               avgStepsPerEpisode, probabilityOfEscape);
        results[0][0].push_back(avgStepsPerEpisode);
        results[0][1].push_back(probabilityOfEscape);

cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " Single (Eq Comp Steps) Learner" << endl;
        singleAgentEqCompBySteps.run(episodeCutoff, stepCutoff*numberOfParallelAgents, stepCutoff,
                                     templateFA, testStates, singleEqCompStepsResultsDir,
                                     avgStepsPerEpisode, probabilityOfEscape);
        results[1][0].push_back(avgStepsPerEpisode);
        results[1][1].push_back(probabilityOfEscape);

cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " Single (Eq Comp Episodes) Learner" << endl;
        singleAgentEqCompByEpisodes.run(episodeCutoff*numberOfParallelAgents, stepCutoff, stepCutoff
                                        templateFA, testStates, singleEqCompEpisodesResultsDir,
                                        avgStepsPerEpisode, probabilityOfEscape);
        results[2][0].push_back(avgStepsPerEpisode);
        results[2][1].push_back(probabilityOfEscape);

cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " Single (Eq Exec) Learner" << endl;
        singleAgentEqExec.run(episodeCutoff, stepCutoff, stepCutoff,
                              templateFA, testStates, singleEqExecResultsDir,
                              avgStepsPerEpisode, probabilityOfEscape);
        results[3][0].push_back(avgStepsPerEpisode);
        results[3][1].push_back(probabilityOfEscape);

//cout << "E: " << episodeCutoff << " S: " << stepCutoff
        //     << " R: " << repetition << " Single Learner (No restrictions)" << endl;
```

```
        //singleAgent.run(episodeCutoff, NUM_STEPS_FOR_FAILURE, NUM_STEPS_FOR_FAILURE,
        //                templateFA, testStates, singleAgentResultsDir,
        //                avgStepsPerEpisode, probabilityOfEscape);
        //results[4][0].push_back(avgStepsPerEpisode);
        //results[4][1].push_back(probabilityOfEscape);

        resultsDetails.open(resultsDetailsFileName.c_str(), ofstream::app);
        resultsDetails << stepCutoff << "\t" << episodeCutoff << "\t" << repetition << "\t";
        for (int i = 0; i<4; i++) {
          resultsDetails << results[i][0][repetition] << "\t"
                         << results[i][1][repetition] << "\t";
        }
        resultsDetails << endl;
        resultsDetails.close();
      }

      resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
      resultsAveraged << stepCutoff << "\t" << episodeCutoff
                      << "\t" << episodeCutoff*numberOfParallelAgents;
      for (int i = 0; i<4; i++) {
        resultsAveraged << "\t" << meanFun(results[i][0])
                        << "\t" << standardError(results[i][0])
                        << "\t" << meanFun(results[i][1])
                        << "\t" << standardError(results[i][1]);
      }
      resultsAveraged << endl;
      resultsAveraged.close();

    }
    resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
    resultsAveraged << endl << endl;
    resultsAveraged.close();
  }
}

void vary_number_of_parallel_learners(ConfigFile config) {

  string testDir = config.getStr("test_prefix");
  system(("mkdir " + testDir).c_str());
  testDir += "//";

  string resultsAveragedFileName(testDir + config.getStr("test_prefix") + "_avg.dat");
  string resultsDetailsFileName(testDir + config.getStr("test_prefix") + "_avg.dat");

  ofstream resultsAveraged;
  ofstream resultsDetails;

  Test singleAgentEqExec;
  Test parallelTest_2agents;
  Test parallelTest_5agents;
  Test parallelTest_10agents;
  Test parallelTest_20agents;

  singleAgentEqExec.init(1, none, config);
```

```
parallelTest_2agents.init(2, config.getFAMergeMethod("parallel_merge_method"), config);
parallelTest_5agents.init(5, config.getFAMergeMethod("parallel_merge_method"), config);
parallelTest_10agents.init(10, config.getFAMergeMethod("parallel_merge_method"), config);
parallelTest_20agents.init(20, config.getFAMergeMethod("parallel_merge_method"), config);

for (int stepCutoff = config.getInt("step_cutoff_init");
     stepCutoff <= config.getInt("step_cutoff_max");
     stepCutoff += config.getInt("step_cutoff_inc")) {

 stringstream stepCutoffDirName;
 stepCutoffDirName << "step_cutoff_" << stepCutoff << "//";
 system(("mkdir " + testDir + stepCutoffDirName.str()).c_str());

  resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
  resultsAveraged << "#GRAPH_DETAILS: Eq Exec | 2 Par | 5 Par | 10 Par | 20 Par" << endl;
 resultsAveraged << "#" << stepCutoff << endl;
 resultsAveraged << "#" << config.getInt("number_of_repetitions") << endl;
 resultsAveraged << "#" << config.getInt("episode_cutoff_max") << endl;
 resultsAveraged << "#" << endl;  //nothing - normaly num of parallel agents
 resultsAveraged.close();

  for (int episodeCutoff = config.getInt("episode_cutoff_init");
       episodeCutoff <= config.getInt("episode_cutoff_max");
       episodeCutoff += config.getInt("episode_cutoff_inc")) {

    stringstream episodeCutoffDirName;
    episodeCutoffDirName << "episode_cutoff_" << episodeCutoff << "//";
    system(("mkdir " + testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str()).c_str());

    string singleEqExecResultsDir(testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str() + "single_eqexec_rep_0//");
    system(("mkdir " + singleEqExecResultsDir).c_str());

    string parallel_2agentsResultsDir(testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str() + "2agents_rep_0//");
    system(("mkdir " + parallel_2agentsResultsDir).c_str());

    string parallel_5agentsResultsDir(testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str() + "5agents_rep_0//");
    system(("mkdir " + parallel_5agentsResultsDir).c_str());

    string parallel_10agentsResultsDir(testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str() + "10agents_rep_0//");
    system(("mkdir " + parallel_10agentsResultsDir).c_str());

    string parallel_20agentsResultsDir(testDir + stepCutoffDirName.str()
            + episodeCutoffDirName.str() + "20agents_rep_0//");
    system(("mkdir " + parallel_20agentsResultsDir).c_str());

    vector<float> results[5][2];  // [x][0] for avgStepsPerEpisode, [x][1] prob of escape

    for (int repetition = 0;
         repetition < config.getInt("number_of_repetitions");
```

```
           repetition++) {

        float avgStepsPerEpisode, probabilityOfEscape;

        FunctionApproximator templateFA;
        templateFA.init();

        vector<State> testStates;
        for (int i = 0; i<config.getInt("number_of_test_states"); i++) {
          State s;
          s.setRandom();
          testStates.push_back(s);
        }

cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " Single (Eq Exec) Learner" << endl;
        singleAgentEqExec.run(episodeCutoff, stepCutoff, stepCutoff,
                              templateFA, testStates, singleEqExecResultsDir,
                              avgStepsPerEpisode, probabilityOfEscape);
        results[0][0].push_back(avgStepsPerEpisode);
        results[0][1].push_back(probabilityOfEscape);

        cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " 2 Parallel Learners" << endl;
        parallelTest_2agents.run(episodeCutoff, stepCutoff, stepCutoff,
                                 templateFA, testStates, parallel_2agentsResultsDir,
                                 avgStepsPerEpisode, probabilityOfEscape);
        results[1][0].push_back(avgStepsPerEpisode);
        results[1][1].push_back(probabilityOfEscape);

        cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " 5 Parallel Learners" << endl;
        parallelTest_5agents.run(episodeCutoff, stepCutoff, stepCutoff,
                                 templateFA, testStates, parallel_5agentsResultsDir,
                                 avgStepsPerEpisode, probabilityOfEscape);
        results[2][0].push_back(avgStepsPerEpisode);
        results[2][1].push_back(probabilityOfEscape);

        cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " 10 Parallel Learners" << endl;
        parallelTest_10agents.run(episodeCutoff, stepCutoff, stepCutoff,
                                  templateFA, testStates, parallel_10agentsResultsDir,
                                  avgStepsPerEpisode, probabilityOfEscape);
        results[3][0].push_back(avgStepsPerEpisode);
        results[3][1].push_back(probabilityOfEscape);

        cout << "E: " << episodeCutoff << " S: " << stepCutoff
             << " R: " << repetition << " 20 Parallel Learners" << endl;
        parallelTest_20agents.run(episodeCutoff, stepCutoff, stepCutoff,
                                 templateFA, testStates, parallel_20agentsResultsDir,
                                 avgStepsPerEpisode, probabilityOfEscape);
        results[4][0].push_back(avgStepsPerEpisode);
        results[4][1].push_back(probabilityOfEscape);
```

```
        resultsDetails.open(resultsDetailsFileName.c_str(), ofstream::app);
        resultsDetails << stepCutoff << "\t" << episodeCutoff << "\t" << repetition << "\t";
        for (int i = 0; i<5; i++) {
          resultsDetails << results[i][0][repetition] << "\t"
                         << results[i][1][repetition] << "\t";
        }
        resultsDetails << endl;
        resultsDetails.close();

      }

      resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
      resultsAveraged << stepCutoff << "\t" << episodeCutoff;
      for (int i = 0; i<5; i++) {
        resultsAveraged << "\t" << meanFun(results[i][0])
                        << "\t" << standardError(results[i][0])
                        << "\t" << meanFun(results[i][1])
                        << "\t" << standardError(results[i][1]);
      }
      resultsAveraged << endl;
      resultsAveraged.close();

    }
    resultsAveraged.open(resultsAveragedFileName.c_str(), ofstream::app);
    resultsAveraged << endl << endl;
    resultsAveraged.close();
  }
}
```

## A.15 Constants

### Constants.h

```
------------------------------------------------------------------------

//#define SETUPDEBUG
//#define TILESETUPDEBUG
//#define STEPDEBUG

#define TILES 10
#define ACTIONS 3
#define XCELLS 9
#define YCELLS 9

#define NUM_STEPS_FOR_FAILURE 5000
#define NUM_SIMULATIONS 5

#define ABSPOS_RANGE 1.7
#define ABSVEL_RANGE 0.14
#define POSRANGE_LOW -1.2
#define POSRANGE_HIGH 0.5
#define VELRANGE_LOW -0.07
#define VELRANGE_HIGH 0.07
```

## A.16 Types

### Types.h

```
--------------------------------------------------------------------------

#ifndef TYPES_H
#define TYPES_H

enum ProblemFormulation {optimistic, pessimistic};
enum FAMergeMethod {minmerge,maxmerge,inittomin,maxaltitude,evaluate_select_best,
                    evaluate_merge1,evaluate_merge2,evaluate_merge3,none};
enum ETraceMode {accumulatetrace, replacetrace};
struct simResultT {int agent; float avgSteps;};
#endif
```